

May 01, 2013

Scaling up scientific computations by using map-reduce-like control flow on NUMA architectures

Vlad Slavici
Northeastern University

Recommended Citation

Slavici, Vlad, "Scaling up scientific computations by using map-reduce-like control flow on NUMA architectures" (2013). *Computer Science Dissertations*. Paper 28. <http://hdl.handle.net/2047/d20002952>

This work is available open access, hosted by Northeastern University.

Scaling up Scientific Computations
by using Map-Reduce-like Control Flow
on NUMA Architectures

A dissertation presented by

Vlad Slavici

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM


THESIS TITLE: SCALING UP SCIENTIFIC COMPUTATIONS BY USING
MAP-REDUCE-LIKE CONTROL FLOW ON NUMA ARCHITECTURES

AUTHOR: VLAD SLAVICI


Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.


Thesis Advisor

2/6/13
Date


Thesis Reader

1/11/12
Date


Thesis Reader

11/5/2012
Date


Thesis Reader


11/19/2012
Date

GRADUATE SCHOOL APPROVAL:


Director, Graduate School

2/12/13
Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:


Recipient's Signature

2/12/2013
Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Abstract

The clock speed of current CPUs and RAM has stopped scaling with Moore’s Law. Yet the scale of applications in science and engineering continues to increase. In order to address this scaling of applications, newer NUMA architectures are emerging. These include parallel disks, hybrid CPU-GPU, and many-core CPUs.

Existing CPU-based algorithms, as well as legacy sequential code, need to be migrated to NUMA in order to stay on the curve of Moore’s Law.

Migrating applications from traditional sequential architectures to NUMA is difficult, because NUMA architectures penalize data access patterns and computation patterns that are often used by programs on traditional architectures. Specific inefficient operations on NUMA architectures include: random access in hash table lookups; tensor computations involving many small, independent matrix multiplications; collecting together the inputs for a task from multiple previous tasks; and slow access to remote memory. Specific issues for hybrid CPU-GPU architectures are: transferring data from CPU to limited GPU global memory (up to 6 GB for 512 cores, in comparison with up to 2 GB/core on the CPU), and the limited size of GPU on-chip cache (typically 768 KB today).

This dissertation analyzes three real-world applications in computational science, each important to a particular community. The three applications are: the determinization and minimization of large finite state automata (important in permutation patterns), a port of the integral operator of the MADNESS scientific library from CPU-only architectures to hybrid CPU-GPUs (important in computational chemistry), and efficient multiplication of large permutations (important in computational group theory).

In all three cases, traditional architectures were either too slow, or did not support problem sizes of sufficient scale. For each application, NUMA solutions based on adapting existing CPU-based algorithms are proposed. NUMA architectures were adopted for the sake of greater speed (through greater parallelism) and/or greater data storage.

Two common techniques used across all three NUMA-based solutions are delayed data access (to allow for reordering of accesses) and delayed task execution (to hide the overhead of on-demand task launch).

In hindsight, we observe that the usage of delayed operations leads to a map-reduce-like control flow in each of the three NUMA-based solution implementations.

Acknowledgments

I would like to thank my advisor Gene Cooperman for his invaluable collaboration and support, and for his guidance throughout the long and rewarding journey of my PhD. I am grateful to my collaborators Xin Dong, Robert Harrison, Daniel Kunkle, Steve Linton, and Raghu Varier. Working with each of them was an intellectually fulfilling experience that helped me grow as a researcher. Many thanks to my PhD committee members Pete Manolios, Mirek Riedewald, and Steve Linton for their valuable suggestions and comments. I also wish to thank my colleagues in the High-Performance Computing Lab and the Solid-State Storage Lab for the productive discussions throughout the years.

I am grateful to my mother Monica and my father Stefan for their love, support, and advice in difficult moments; to my grandparents Omama, Otata, and Tataia for always being such a loving and caring presence in my life; to my grandmother Mamaia for motivating me to work hard and strive to achieve my goals; to my brother Mihai for his youthful enthusiasm and cheerful student-life stories; to my aunt Tanti Adi, my uncle Nenea Titi, and my cousin Viorel for their encouragement and for believing in me; to Adrian for his support and for always being interested in my research; and to Lavinia for her encouragement and warm hospitality. Finally, I would like to thank Simona for being my source of positive energy each and every day.

Contents

Abstract	iii
Acknowledgments	v
Contents	vii
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Overview of the Three NUMA-based Solutions	5
1.2 Algorithms for Large Finite State Automata on Parallel Disks	9
1.3 Multiresolution Analysis for CPU-GPU Clusters	15
1.4 Permutation Multiplication for Many-Core and Parallel Disks	20
1.5 Emerging NUMA Architectures used in Computational Science	23
1.6 Thesis Organization	27
2 Algorithms for Large Finite State Automata	29
2.1 Terminology and Background	30
2.2 Stacks, Token Passing Networks and Forbidden Patterns . .	34
2.3 NFA Determinization via Subset Construction for large NFAs	38
2.4 Finding the Unique Minimal DFA	41

2.5	Implementation Issues for the FSA Algorithms	44
2.6	Multi-threaded Implementations for Shared Memory	46
2.7	NFA Minimization: an Alternative Approach	47
2.8	Experimental Results	49
2.9	Related Work	53
3	Multiresolution Analysis in MADNESS for Hybrid CPU-GPU Clusters	57
3.1	Background: GPUs in High-Performance Computing	58
3.2	MADNESS — a Scientific Simulation Framework based on Multiresolution Analysis	60
3.3	Migrating MADNESS Operators to Hybrid CPU-GPU Architectures	67
3.4	The Apply Operator	71
3.5	Implementation Issues for CPU-GPU Apply	80
3.6	Experimental Results	82
3.7	Related Work	88
4	Permutation Multiplication for Many-Core Machines and Parallel Disks	93
4.1	Problem Background	97
4.2	Permutation Multiplication using External Memory	97
4.3	Permutation Multiplication in RAM	103
4.4	Permutation Inverse. Multiplication by an Inverse	105
4.5	Performance Analysis	106
4.6	Experimental Results	111
5	Lessons learned from the Migration of CPU-based Algorithms to NUMA Architectures	119

5.1	The Control Flow of Standard Map-Reduce Programs	120
5.2	The Map-Reduce Control Flow in the NUMA-based Algorithms	123
5.3	Recursive Algorithms on Parallel Disks	142
5.4	Join-based Operations	145
5.5	Impact of Research	147
A	Map-Reduce-like Control Flow in the Proposed NUMA-based Algorithms	151
A.1	Map-Reduce-like Control Flow in the Parallel Disk-based Sub- set Construction	151
A.2	Map-Reduce-like Control Flow in the Parallel Disk-based DFA Minimization	152
A.3	Map-Reduce-like Control Flow in the Hybrid CPU-GPU Apply Operator	155
A.4	Map-Reduce-like Control Flow in the Permutation Multiplica- tion Algorithms	157
	Bibliography	163

List of Figures

1.1	The control flow of the hybrid CPU-GPU Apply operator in MAD- NESS	19
2.1	A 2-stack followed by a k -stack represented as a token passing network	36
2.2	A 3-buffer followed by a k -stack represented as a token passing network	37
2.3	Frontier sizes and already-visited hash table sizes for each BFS level of the implicit graph corresponding to subset construction.	51
3.1	The NVIDIA Fermi class of GPUs	61
3.2	The MRA approach viewed as a telescoping series of grids . . .	63
3.3	Multiresolution Analysis over benzene dimer	64
3.4	The Layered MADNESS Architecture	65
3.5	The control flow of a hybrid CPU-GPU MADNESS operator . .	71
3.6	Data flow patterns for the Hybrid CPU-GPU system	72
3.7	Tensor Product: Generalization of Matrix Multiplication	74
3.8	Rank reduction for tensors in MADNESS	79
5.1	The <i>standard map-reduce</i> control and data flow	122
5.2	Map-reduce-like control flow in the hybrid CPU-GPU Apply algo- rithm	139

List of Tables

1.1	Local/Non-local memory components for three NUMA architectures	2
2.1	Parallel Disk-based Subset Construction	50
2.2	Multi-threaded RAM-based timings for stack depth 11	52
2.3	Multi-threaded RAM-based results for stack depth 9, 10, and 11	53
3.1	CPU scale-up vs. GPU scale-up for application <i>Coulomb</i>	83
3.2	CPU with 16 threads vs. GPU for application <i>Coulomb</i>	84
3.3	Timings for 3-dimensional <i>Coulomb</i> using our custom CUDA kernels and using cuBLAS 4.1	84
3.4	Timings for larger instances of 3-dimensional <i>Coulomb</i> using our custom CUDA kernels and using cuBLAS 4.1	85
3.5	CPU-only, GPU-only and hybrid computation scale-up with the increase in number of compute nodes	86
3.6	Timings for the Apply part of the MADNESS 4-dimensional Time Dependent Schrodinger Equation	87
4.1	Measured system parameters for external memory	111
4.2	Running times of our new RAM/disk and RAM/flash algorithms and comparison with estimated running times	112

4.3	Comparison of the traditional algorithm and the buffered traditional algorithm with disk-based and flash-based external memory	112
4.4	Comparison of three parallel-disk permutation multiplication algorithms for increasing permutation size	113
4.5	Comparison of three parallel-disk permutation multiplication algorithms for increasing parallelism	113
4.6	Measured system parameters for cache/RAM	116
4.7	Comparison of traditional and new algorithms, using thread or process-based parallelism	116
4.8	Running times (seconds) of our new implicit indices permutation multiplication for cache/RAM	117
5.1	Fundamental algorithms, techniques and data structures for algorithms in various environments: RAM-based sequential, RAM-based parallel, disk-based parallel	145

List of Algorithms

1	RAM-based, Recursive Subset Construction	38
2	Parallel Disk-based Subset Construction	39
3	Parallel Disk-based Subset Construction (continued, part 2) .	39
4	Parallel Disk-based Subset Construction (continued, part 3) .	40
5	Sequential RAM-based Forward Refinement	42
6	Parallel Disk-based Forward Refinement	43
7	Parallel Disk-based Partitions Collapse	44
8	The “Apply” Algorithm	76
9	The CPU-GPU Version of the “Apply” Algorithm	77
10	The compute and postsprocess functions of the “Apply” Algo- rithm	78
11	Permutation Multiplication Using External Sort	99
12	Permutation Multiplication Using RAM buckets	100
13	Permutation Multiplication using Implicit Indices	101
14	Multi-threaded cache/RAM Permutation Multiplication using Implicit Indices	104
15	Permutation Inverse Using Implicit Indices	106
16	Permutation Multiplication by an Inverse Using Implicit Indices	107

CHAPTER 1

Introduction

Today, the parallel computing landscape is diverse and dynamic, including a variety of emerging Non-Uniform Memory Architectures (NUMA), such as the parallel disks of a cluster, hybrid CPU-GPU architectures, or many-core machines, amongst others.

In NUMA architectures, memory access time is non-uniform (memory that is closer to a processing unit can be accessed faster than memory that is farther from the processing unit). Each NUMA architecture has multiple levels of memory with respect to the access time (e.g. in commodity clusters the levels are: Level-1 cache, Level-2 cache, main memory, local disk, remote memory, and remote disk). For the simplicity of exposition, we categorize all memory components of an architecture into two categories: *local memory* and *non-local memory* (see Table 1.1). An algorithm for a two-level memory hierarchy can also be used in an architecture with multiple hierarchical levels, by applying it to any two adjacent levels.

As the single-core CPU frequency peaked in the mid-2000s, further performance gains were no longer possible with new generations of single-core CPUs. Moreover, since the density of RAM is not increasing as fast as earlier, problems with very large data sets will overflow the RAM of commodity

NUMA Architecture	Local Memory	Non-local Memory
Parallel-disks Cluster	Local RAM	Local Disk Remote RAM Remote Disk
Hybrid CPU-GPU Cluster	Local CPU RAM	Local GPU RAM Remote CPU RAM Remote GPU RAM
Many-core Machine with multiple memory levels	Level-1 cache Level-2 cache	RAM

Table 1.1: For a parallel-disks cluster, RAM local to a compute node can be considered local memory, whereas local disk, remote RAM, and remote disks are non-local memory. For a cluster with hybrid CPU-GPU nodes, a node’s CPU RAM is local, whereas the node’s GPU RAM and memory on the other nodes is non-local. A node’s GPU RAM is non-local because of the high access latency over PCIeExpress between the CPU and the GPU. The case of a many-core machine, although traditionally considered an SMP (Symmetric Multi-Processor) architecture, can also be viewed as a NUMA architecture with respect to the cache-RAM hierarchy. For a CPU core, accessing Level-1 cache requires up to 10 CPU cycles, accessing Level-2 cache requires up to 40 CPU cycles. Both Level-1 cache and Level-2 cache are local memory. Accessing RAM requires around 50–100 CPU cycles. RAM is non-local memory.

machines.

This dissertation studies three different real-world applications in three different scientific domains. For each application, a novel NUMA-based solution is proposed.

The three applications were chosen because they each solve an important problem in a scientific domain. The domains are: permutation patterns (using parallel disks), computational chemistry (using hybrid CPU-GPU nodes in a cluster), and computational group theory (using many-core CPUs and parallel disks).

The application in the permutation patterns domain is concerned with a series of problem instances of increasing size. Each problem instance requires 3.5 times as much memory as the previous instance. The largest problem instance we solved required 1.1 terabytes of storage (see Table 2.1). Hence,

parallel disks were needed to accommodate the large data sets of these problem instances.

For the application in computational chemistry, the decision of using hybrid CPU-GPU clusters was driven by the domain scientists. Their decision is based on the need to scale up to larger problems, which is dependent on the current trend of using more and more hybrid CPU-GPU nodes in supercomputers. This trend is itself motivated by the need to keep energy consumption in supercomputers within reasonable limits, so that the cost of running a supercomputer is kept within reasonable limits.

For the application in computational group theory, the choice of the many-core architecture was imposed by the need to speed up the application solution. The many-core solution can be used for problem instances whose data sets fit in RAM. For problem instances whose data sets do not fit in RAM, we propose a parallel-disks version of the solution.

A unifying theme across the three applications is that it was no longer possible to scale up by using existing algorithms on traditional architectures. In order to scale up to larger problem instances, or to solve problem instances faster (therefore allowing more problem instances to be solved in a fixed amount of time), we had to use NUMA architectures.

NUMA architectures offer significant performance or scale improvements compared to traditional architectures. On the flip side, NUMA architectures penalize various access patterns and computational patterns (as a consequence of the non-uniform access time):

- parallel disks penalize random access;
- hybrid CPU-GPU architectures penalize complex data access to the GPU RAM and on-demand transfers between the CPU and GPU, as

well as computations with a high data-access-to-arithmetic-operation ratio; and

- many-core machines penalize non-cache-local data access.

The proposed solutions for NUMA architectures avoid these penalties, while harnessing the architecture strengths.

For each application, we migrated existing algorithms from traditional CPU-based architectures to a targeted NUMA architecture by using *delayed data access* and *delayed task execution*, along with architecture-specific and problem-specific solutions. Using delayed operations, the application can avoid the forms of data access and computation that are penalized by the NUMA architectures. The benefits of using *delayed data access* and *delayed task execution* are presented for each of the three applications in their own dedicated chapters.

In each case, we show experimentally that our migration of existing algorithms for traditional architectures to NUMA leads to significant performance improvements or to greater scale.

In addition, in Chapter 5 we identify a common *map-reduce* pattern unifying the NUMA-based solutions for the three applications discussed in the dissertation. Each of the three solution implementations extends the *standard map-reduce* control flow in its own particular way, as described in Section 5.2. In Chapter 5 we speculate that *map-reduce*-like solutions could be applied successfully in the future to migrate CPU-based algorithms for other scientific computations to NUMA.

Thesis Statement

Three novel solutions are proposed for three different real-world problems in computational science. In all three cases, traditional architectures were either too slow, or did not support problem sizes of sufficient scale. Each of the three solutions is achieved by migrating existing CPU-based algorithms to a targeted NUMA architecture. Each NUMA-based solution improves upon previously available CPU-based solutions for the corresponding problem either by being able to solve larger problem instances (parallel disk-based algorithms for finite state automata) or by being able to solve problem instances faster (a hybrid CPU-GPU operator in the MADNESS scientific framework, and a permutation multiplication algorithm for many-core machines and parallel disks).

1.1 Overview of the Three NUMA-based Solutions

The first real-world problem addressed in the dissertation is the determinization and minimization of large finite-state automata. Existing algorithms for determinization and minimization operate in RAM on commodity machines. However, there are determinization problems that overflow the available RAM of commodity machines and therefore cannot be completed with existing RAM-based approaches. To address this issue, we propose a solution that takes advantage of the storage space offered by the aggregate disks of a commodity cluster. The contributions of the dissertation in this area are:

- a parallel disk-based algorithm for NFA determinization (it is intended

for large determinizations that overflow RAM on commodity machines);

- a parallel disk-based algorithm for DFA minimization (the minimal equivalent DFA of a large DFA is often much smaller than the large DFA and, thus, easier to process);
- a new scalable RAM-based multi-threaded implementation for NFA determinization and DFA minimization;
- an application of the parallel disk-based algorithms to a challenging permutation-patterns problem involving stack-sortable permutations encoded as token passing networks; the experimental evidence gathered from the application of the disk-based algorithms led to the formulation of a conjecture for that specific stack-sortable permutations problem.

These contributions are published in [SKCL11] and [SKCL12].

The second real-world problem is the migration of the integral operator in MADNESS (a multiresolution-based general-purpose scientific framework) from CPU-only architectures to CPU-GPU clusters. This is a challenging problem, because MADNESS computations involve complex data access and many small compute-intensive tasks as opposed to simple, regular data access and a few large compute-intensive tasks. Complex data access is slow on GPUs. Also, on-demand invocation of the small compute-intensive tasks on the GPU suffers from high GPU access latency, low CPU-GPU transfer bandwidth and high CUDA kernel launch overhead. This overhead can dominate the application running time. Moreover, in the case of small tasks, a single CUDA kernel executing on the GPU leaves a part of the GPU idle. We propose a solution for migrating the MADNESS integral operator from CPU-based architectures to CPU-GPU clusters that alleviates these issues.

Even though these computations in MADNESS are not a great fit for the hybrid CPU-GPU architecture, the current architectural trends in high-performance computing impose the usage of CPU-GPU. In order to scale up MADNESS, domain scientists have to use hybrid CPU-GPU clusters, even though the architecture would not be their first choice. The reason is that, currently, large high-performance clusters scale up by adding GPUs, rather than CPUs. This architectural trend is itself imposed by the high increase in energy consumption that adding CPUs would incur compared to adding GPUs. This trend has similarities with the shift from sequential CPUs to many-core CPUs: the architectural and resource limitations imposed the change, rather than the programmers.

The contributions of the dissertation in this area are:

- harnessing the full computational power of the hybrid system, by distributing work to both the CPU and the GPU and by simultaneously running compute-intensive code on both the CPU and the GPU;
 - this is achieved by a reorganization of the control flow of the MADNESS integral operator, including the separation of compute-intensive and data intensive code. This control flow reorganization includes splitting the operator into three parts: *preprocess*, *compute*, and *postprocess*.
- replacing on-demand data access and on-demand GPU task execution with delayed data access and delayed GPU task execution in order to hide the high GPU access latency and low CPU-GPU transfer bandwidth inherent in on-demand operations:
 - data inputs for *compute* are asynchronously aggregated and then copied to pre-page-locked buffers that are transferred to the GPU

(thus, the latency penalty is paid only once for the entire buffer and transfer bandwidth is up to 3-times higher than for on-demand transfer due to the pre-page-locked buffer);

- multiple small CUDA kernels are launched on the GPU concurrently using CUDA streams, to fully occupy the GPU cores, once the aggregated input data has been transferred to the GPU;
- implementing MADNESS Library extensions that support the modified control flow and the aggregation of multiple *compute* data inputs;

These contributions are published in [SVCH12].

The third real-world problem is the multiplication (composition) of large permutations. The widely-used standard in-RAM permutation multiplication algorithm suffers from many non-cache-local data accesses, due to the random data access pattern of the algorithm. In the case of random permutations, each 4-byte permutation element access is very likely non-cache-local. Therefore, for each 4-byte element access, the standard algorithm pays the price of a cache miss, a cache line eviction, and a full 64-byte cache line load. We address these issues by designing a multi-threaded cache-aware permutation multiplication algorithm that employs only streaming data access patterns. The same issues arise for the standard permutation multiplication algorithm running on parallel disks, but with much higher penalties. Therefore, we also implement a parallel disk-based version of the streaming-based multi-threaded cache-aware algorithm for permutation multiplication. The contributions of the dissertation in this area are:

- a new multi-threaded permutation multiplication algorithm that is up to 50% faster than the standard multi-threaded permutation multiplication algorithm for permutations larger than CPU cache;

- a parallel disk-based version of the new multi-threaded permutation multiplication algorithm, which makes multiplication of very large permutations practical; and
- analytical formulas for the estimation of the running time of the new algorithm based on the size of the input data.

These contributions are published in [SDKC10].

A more detailed description of the three computational science problems and their solutions is presented next. Full details are then presented in each application’s dedicated chapter. Algorithms for the determinization and minimization of finite state automata for parallel disks are presented in Chapter 2. Multiresolution analysis for the hybrid CPU-GPU architecture is presented in Chapter 3. Permutation multiplication for many-core and for parallel disks is presented in Chapter 4. Implementation challenges and solutions are presented in each chapter.

1.2 Algorithms for Large Finite State Automata on Parallel Disks

We propose a parallel disk-based NFA determinization algorithm and a parallel disk-based DFA minimization algorithm that are applicable to larger finite state automata compared to RAM-based approaches for commodity architectures. This work attempts to relieve the critical bottleneck in many automata-based computations, specifically the combinatorial explosion of states that characterizes NFA determinization.

Parallel disks were chosen as the target parallel architecture for the finite state automata algorithms, because they provide large amounts of inexpen-

sive storage. The large amount of storage was necessary to generate the large intermediate DFAs produced by the determinization algorithm: the largest NFA determinization we ran used 1.1 terabytes of storage and resulted in a 1.9 billion-state DFA. Disks are about 100 times less expensive per storage unit than RAM. The disadvantages of disks are high-latency data access (about 1 million times higher than RAM) and relatively low bandwidth (about 100 times lower than RAM). However, the bandwidth disadvantage can be greatly alleviated by accessing multiple disks in parallel, while the high-latency disadvantage can be offset by eliminating random access to data whenever possible [Rob08].

The determinization algorithm was adapted from the well-known sequential subset construction algorithm. The minimization algorithm was adapted from a shared-memory parallel algorithm designed for the CM-5 machine in the 1990s.

The two proposed algorithms have been applied to an important problem in *forbidden permutation patterns*, an active research topic in the algebraic field of permutation patterns (see [LRV10]).

This research topic is concerned with answering the question: given a data structure (or a series of connected data structures), what are the permutations of a stream of input objects that can be achieved by passing the stream through the data structure? When the data structure is a sequence of stacks, the problem is known in the permutation patterns community as “stack sorting”. Stack sorting is one of four important problems presented by Murray Elder and Vince Vatter at the 2005 International Conference on Permutation Patterns [Eld].

Forbidden permutation patterns have been an active research area throughout the past fifteen years. Problems that have been solved in recent years by

the forbidden permutation patterns community include: *permutations sortable by two pop-stacks in parallel* [SV09, AS99], *permutations generated by a stack of depth 2 and an infinite stack in series* [Eld06], *permutations for which the number of repeated letters in a reduced decomposition equals the number of occurrences of 321 and 3412* [Tenb], and *permutations that can be drawn on an X* [Wat07, Eli], among others. A comprehensive list can be found at the Database of Permutation Pattern Avoidance [Tena]. Research results in permutation patterns are presented annually at the International Conference on Permutation Patterns [ICP].

Instances of *forbidden permutation patterns* problems can be solved by modeling a data structure as a token-passing network (see [ALT97]). The token-passing network is then represented as a finite state automaton. The finite state automaton is the input of a chain of processing steps. NFA determinization and DFA minimization are central to some of these processing steps (see Section 2.2).

The specific problem analyzed here is determining the *forbidden permutation patterns* of a 3-buffer connected with an infinite stack in series (details are presented in Chapter 2). The following methodology was applied: solve the problem for a 3-buffer connected with a k -stack and increase k until a convergent solution is observed.

Each increasing instance in the series requires 3.5 times more memory than the previous instance. This exponential storage requirement means that even using parallel disks we can only scale to a few larger instances more than an in-memory approach. However, scaling up to a few extra instances on parallel disk was enough to observe a convergent solution.

Concretely, for this series of problems solutions were previously achieved up to $k = 8$ using in-memory approaches on commodity computers, but these

solutions were not enough to observe convergent behavior. For larger k , the available determinization techniques on commodity computers required more than the available memory. Our NFA determinization and DFA minimization algorithms were applied to the next four instances in the series, and required a peak disk usage of 24 GB (for $k = 9$), 90 GB (for $k = 10$), 327 GB (for $k = 11$), and 1,136 GB (for $k = 12$).

Our parallel disk-based solution was able to solve the largest instance in 2 days and 20 hours using the aggregate disks of a 29-node commodity cluster. During this process, an intermediate DFA with 1,899,715,733 states needed to be generated for the $k = 12$ instance. To the best of our knowledge, this is the largest determinization ever reported in the literature. This also represents a 38-times larger intermediate DFA than the intermediate DFA generated by the $k = 9$ instance, which is the largest problem instance that fits in RAM on a commodity machine. (The $k = 9$ instance generates an intermediate DFA of 49,722,541 states.) The minimal equivalent DFA for the $k = 12$ instance has 774,172 states, a more than 1,000-fold reduction in size.

Based on these four new results, convergent permutation patterns were observed, and a conjecture was formulated. The conjecture states that there are 12,636 forbidden permutations of lengths between 7 and 18 for a 3-buffer and an infinite stack in series (see Section 2.2 for details). This is an important result in permutation patterns. An equivalent experimental result for the case of a 2-stack serially connected with a k -stack was used to theoretically prove the convergence property in [Eld06].

For comparison, we have also implemented multi-threaded versions of the traditional NFA determinization algorithm (subset construction) and the *forward refinement* DFA minimization algorithm (which were both also used for parallel disks). This implementation scales almost linearly with an increase

in the number of worker threads (see Section 2.6). Using this version on a high-end shared memory machine with 128 GB of RAM, we were able to solve the $k = 9$, $k = 10$, and $k = 11$ instances of the 3-buffer serially connected with a k -stack in series. The $k = 9$ instance used up to 12 GB of RAM, so it could also have been solved on a commodity machine with 16 GB of RAM. However, the $k = 10$ instance used up to 40 GB of RAM, which is 2.5-times the RAM size of a commodity machine.

In addition, two NFA minimization algorithms were run on the input NFAs (see Section 2.7). One of the NFA minimization algorithms (proposed in [GP08b]) reduced small automata in the series (for $k = 5$ or $k = 6$) by less than 10%. In both cases the minimization was slow (see Section 2.7). For large automata (such as the $k = 9$ or $k = 10$ cases) the NFA minimization program execution crashed with segmentation fault after peaking at 30 GB, and respectively 34 GB of memory usage. The other NFA minimization algorithm we tried was proposed in [AL04] and implemented in the GAP Automata package [DLM]. This second NFA minimization algorithm had no effect on the input NFAs.

Algorithms Overview The parallel disk-based NFA determinization algorithm is adapted from the well-known sequential subset construction algorithm. The subsets are constructed in the parallel-disks algorithm by performing a breadth-first search (BFS) of subsets of NFA states. Subsets of NFA states correspond to the resulting DFA states. Successive levels in the BFS are constructed from previous levels. At each breadth-first iteration, the subsets in the new BFS frontier are compared to the previously-discovered subsets for duplicate detection and removal. For each BFS frontier, each generated subset (regardless of whether it is new or previously discovered) determines a new transition to be added to the partial intermediate DFA.

The parallel disk-based DFA minimization algorithm is an iterative partition-refinement algorithm. Initially, all states in the DFA belong to one of the two initial coarse partitions: the accepting partition and the non-accepting partition. Based on the source and destination of the DFA transitions, the state partitions are iteratively refined. In each iterative step, the transition function with a fixed transition label is applied to each DFA state. For two different DFA states in the same current partition, if the transition function takes them to two different partitions, then the two DFA states will belong to two different partitions after the current iteration. Applying the transition function with a fixed label to all the DFA states in a current partition results in DFA states that are part of a number of current partitions. This is the number of partitions that the current partition is refined to. The iterative process continues by fixing the next transition label, until a fixed point is reached (no partitions are refined in the last iteration). The algorithm is guaranteed to converge to a partition graph that is isomorphic to the equivalent minimal DFA.

The parallel disk-based finite state automata algorithms were implemented in C, using the Roomy extension library. Roomy was chosen because it provides a broad range of primitive data structures including bit arrays, hash tables, or unordered sets (`RoomyList`).

The algorithm descriptions, as well as experimental results for the application of the algorithms to the problem of finding the *forbidden permutations* of a 3-buffer serially connected with a k -stack, are presented in Chapter 2.

1.3 Multiresolution Analysis for CPU-GPU Clusters

MADNESS (Multiresolution ADaptive Numerical Environment for Scientific Simulation) [FBHJ04, HFYB03, HFY⁺04, HF, HF07] is a scientific simulation framework that implements a set of integral and differential operators. MADNESS is used in computational chemistry, computational physics and other related fields employing quantum mechanics. More specifically, MADNESS has been used to solve problems in nuclear physics [FPH⁺09], time-evolution quantum chemistry [VHKac12], nanoscale photonics [RHH12], Hartree-Fock and density functional theory [SMYH08], and electronic structure [Tho11], to cite just a few domains.

MADNESS was designed to sustain petascale computations [HF07]. The MADNESS project is important for the high-performance computing community. (In 2011, the project received an R&D 100 award [SJ].) MADNESS is supported and developed by an active team, whose members are based at Oak Ridge National Laboratory, University of Tennessee at Knoxville, Virginia Tech, Argonne National Laboratory, Ohio State University and Northeastern University.

The four most common operators in MADNESS are the integral operator (**Apply**), the compression operator (**Compress**), the reconstruction operator (**Reconstruct**) and the truncation operator (**Truncate**). These four operators suffice for many MADNESS applications. Only **Apply** is CPU-intensive and the time for calls to **Apply** dominates over the other three.

We migrated the CPU-only **Apply** operator of MADNESS to hybrid CPU-GPU architectures as a first important step in migrating the MADNESS framework to hybrid CPU-GPU architectures. In an experiment for 4-dimensional

tensors on a 500-node cluster partition, we demonstrate a speedup of 1.9 times by using the GPUs of the cluster over using only the CPUs of the cluster. Further, we demonstrate a speedup of 2.3 times over the CPU-only version by dispatching tensor operations to both the CPU and the GPU (see Table 3.6 of Section 3.6). In an experiment on 3-dimensional tensors, we demonstrate a 4-times speedup of the hybrid CPU-GPU implementation compared to an equivalent MADNESS CPU-only implementation over 8 nodes of a cluster, with 16 cores per node (see Table 3.5 of Section 3.6).

MADNESS operators are highly optimized for CPU-computing. The compute-intensive MADNESS kernels achieve up to 6 GFLOPS on a single core in double precision [SHM⁺11].

However, in recent years the high-performance community’s focus has been shifting from scaling out using more and more CPUs to scaling out by adding GPUs to supercomputers (see [Topb] and [Topa]). The main reason is that GPUs have a high ratio of computational power to energy consumption. By contrast, CPUs consume more energy, and have a lower computational power than GPUs.

With this architectural shift, high-performance computing frameworks need to migrate their code to hybrid CPU-GPU architectures in order to continue to scale out and utilize the emerging architecture to its full power.

Therefore, MADNESS needs to scale out to hybrid CPU-GPU clusters in order to stay on track to achieve one of its main goals: petascale computing.

Another goal of MADNESS is portability. MADNESS is currently portable to Linux and Mac desktops, commodity clusters, and to various supercomputer architectures, such as Cray XT-5, Cray XK-6, IBM BG/Q, and IBM BG/P. Hybrid CPU-GPU clusters is another important high-performance architecture that MADNESS needs to support, in order to extend its portability in

today’s dynamic supercomputing landscape.

However, porting compute-intensive MADNESS operators to hybrid CPU-GPU clusters is challenging, because MADNESS operators implement many small tensor products, rather than a few large tensor products. (Tensors are higher dimensional generalizations of standard 2-dimensional matrices.) A tensor products is represented as a series of matrix multiplications in MADNESS. The **Apply** operator involves many small matrix multiplications often running into the hundreds. This results in a less regular organization of computations on GPUs.

Substituting each on-demand matrix multiplication operation on the CPU with a similar on-demand operation on the GPU is inefficient, because it would result in low GPU occupancy, high CPU-GPU transfer latency, low CPU-GPU transfer bandwidth (since the page-locking overhead for each small matrix is too large), and high GPU kernel launch overhead.

Our CPU-GPU port of **Apply** replaces the on-demand data access and task execution from the CPU-based **Apply** implementation with delayed data access and delayed task execution (for parts of the operator that use the GPU).

In adapting **Apply** to hybrid CPU-GPUs, the main changes made to the control flow of the CPU-based **Apply** algorithm are:

1. the separation of compute-intensive code from data-intensive code;
2. the aggregation of computation; and
3. the aggregation of data inputs.
4. the overlapping of CPU computation with GPU computation.

Data input aggregation alleviates the on-demand transfer latency (by transferring a few large buffers instead of many small ones) and the page-locking

overhead (by page-locking fewer, contiguous buffers). Computation aggregation (running multiple tasks to occupy all processors of the GPU) helps obtain a high GPU utilization ratio. Overlapping the CPU and the GPU computation utilizes both the CPU and the GPU fully.

Algorithm Overview To separate data-intensive code from compute-intensive code, the **Apply** operator was split into three parts:

- Part 1. A **preprocess** part. This gathers the necessary data inputs for the computation and prepares them for the compute-intensive processing. Multiple CPU threads can perform **preprocess** at the same time.
- Part 2. A **compute** part. The data inputs prepared by **preprocess** are processed (this is the compute-intensive part of the operator). Multiple **compute** tasks are distributed to the CPU and the GPU by a dispatcher thread. Each **compute** task produces a result.
- Part 3. A **postprocess** part. This processes the results produced by **compute** and prepares them for the rest of the computation. **Postprocess** serves as the glue between the operator and the rest of the computation.

The control flow of the CPU-GPU **Apply** operator is presented in Figure 1.1.

We extended the MADNESS framework so that the outputs of **preprocess** tasks are aggregated into a few large buffers. There is one large buffer per task “kind”. The task “kind” is determined uniquely by a combination of task parameters and user-supplied hash function.

A dispatcher thread processes each per-“kind” buffer. For each per-“kind” buffer:

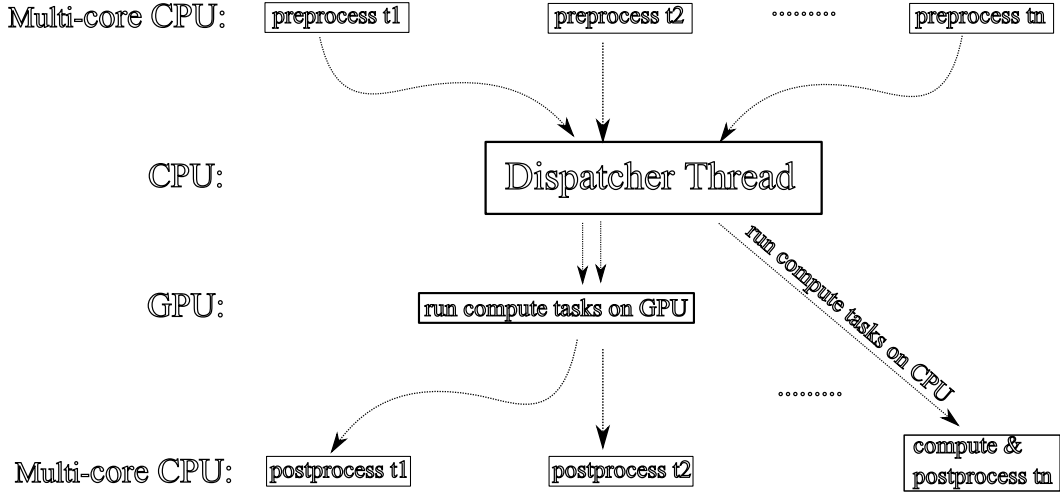


Figure 1.1: The control flow of the hybrid CPU-GPU Apply operator in MADNESS.

- The dispatcher will dispatch a fraction of the data as input to `compute` CPU-tasks. This dispatch process is non-blocking and the `compute` CPU-tasks can start executing on the CPU-cores immediately after they are dispatched;
- The dispatcher will then invoke a CPU-task that prepares the rest of the data for GPU processing. This involves copying the data into a few pre-allocated, pre-page-locked buffers. (These buffers now contain the inputs for the `compute` GPU-tasks.) The buffers are transferred to the GPU;
- The CPU-task invoked by the dispatcher launches `compute` GPU-tasks asynchronously (using CUDA streams) for each input in the transferred buffers. Multiple `compute` GPU-tasks run concurrently if one `compute` GPU-task does not occupy all the Streaming Multiprocessors (SMs) of the GPU (this is the case for 3-dimensional instances of `Apply`).

Chapter 3 presents in detail the hybrid CPU-GPU `Apply` operator. The solutions presented in Chapter 3 represent a promising first step towards ef-

ficiently running MADNESS applications on hybrid CPU-GPU architectures, and ultimately on Titan, which is projected to be the largest or second-largest supercomputer in the world.

1.4 Permutation Multiplication for Many-Core and Parallel Disks

Permutation multiplication (along with permutation inverse and permutation multiplication by an inverse) is an important operation in computational group theory, especially in the area of permutation groups. It is provided as a primitive operation in algebra packages such as GAP [GAP08] or Sage [S⁺12]. Permutation multiplication is a heavily data-intensive operation and it consists of only data accesses and no numerical computation. The standard permutation multiplication algorithm can be described as:

```
X, Y, Z arrays of size N with indices  $0 \dots N - 1$ .
X[i]  $\in \{0 \dots N - 1\}, \forall i \in \{0 \dots N - 1\}$ .
for i  $\in \{0 \dots N - 1\}$  Z[i] = Y[X[i]].
```

In the case of random permutations, by using the traditional one-line algorithm above, each data access is very likely to be non-local, and thus expensive.

We propose algorithms for efficiently executing the basic permutation operations for large permutations (permutations that range in size from 4 million points to permutations with billions of points). The size of the permutation dictates the preferred architecture. At the high end of our regime (billions of points), the preferred architecture consists of parallel disks. Using parallel disks, we are able to efficiently multiply permutations with 12.8 billion points in under one hour using the 16 local disks of a 16-node cluster (see Section 4.6). At the low end of our regime (millions of points), the preferred

architecture is represented by a many-core shared-memory machine. For this architecture, the proposed algorithm has an advantage over the standard algorithm for permutations large enough that they overflow the CPU cache. In the case of a multi-threaded algorithm, we demonstrate a speedup of the new algorithm of almost 50% over the standard algorithm on a recent eight-core commodity computer for permutations with 32 million points (see Table 4.8 in Section 4.6). In the case of single-threaded processes, we run eight competing processes simultaneously, and demonstrate the same 50% speedup over the traditional permutation algorithm. In this single-threaded case, the speedup is observed for permutations with as few as 4 million points (see Table 4.7 in Section 4.6).

The importance of these new methods for computational group theory is immediately evident by considering a group membership permutation computation for Thompson’s group from 2003. Thompson’s group acts on 143,127,000 points [CR03]. Those 143 million points from nine years earlier are well within the regime of interest discussed in this work: between 4 million points and billions of points. That computation now fits on today’s commodity computers, including the in-RAM technique of this work.

In addition to permutations being given directly, permutations arise frequently as the output of a Todd-Coxeter coset enumeration algorithm. There are several excellent descriptions of this algorithm [CDHW73, Fel61, Neu82, TC36]. In those cases, the first description of the group is as a finite presentation, and one employs coset enumeration to convert this into a more tractable permutation representation. The group can then be efficiently analyzed through such algorithms as Sims’s original polynomial-time group membership and the rich library that has grown up around it. Examples of such large coset enumerations include parallel coset enumeration [CH97] used to

find a permutation representation of Lyons’s group on 8,835,156 points, sequential coset enumeration [HS99] used to find a different permutation representation of Lyons’s group also on 8,835,156 points, and a result [HSW01] finding a permutation representation of Thompson’s group on 143,127,000 points.

The following paragraphs present our new permutation multiplication algorithm for many-core machines at a high level. The parallel-disks algorithm version is similar, but for a different memory hierarchy: disk/RAM instead of RAM/cache.

Algorithm Overview This is a four-step algorithm.

In step 1, elements of array X are distributed into buckets following the criterion “element $X[i]$ is assigned to bucket number $X[i]/\text{bucket size}$. When multiple threads execute simultaneously, each thread will own a sub-bucket in each bucket. In that case, the distribution criterion is “element $X[i]$ is assigned by the current thread to its own sub-bucket into bucket number $X[i]/\text{bucket size}$.”

A preprocess step (step 0) is necessary for the threads to compute the sizes of the sub-buckets and the offsets of the sub-buckets within a bucket in parallel, via a parallel prefix.

In the case of a sequential computation (single-threaded), the bucket size is set to half the size of L2 cache or less, so that two buckets fit in L2 cache at the same time. In the case of a multi-threaded computation, two buckets for each thread must fit in L2 cache at the same time. The reason is that step 2 of the algorithm requires that an X bucket and the corresponding Y bucket fit in L2 cache at the same time for fast random access.

In step 2, each thread loads both an X bucket and the corresponding Y bucket in L2 cache, repeatedly. The value $Y[X[i]]$ is computed in step 2 by

inspecting only cache-local data.

Step 3 is the inverse of step 1, in which elements are “de-bucketized” and distributed to their natural indices. This distribution computes the final result.

Chapter 4 presents the efficient many-core and parallel disk-based algorithms for permutation multiplication in detail.

For parallel disk-based permutation multiplication, we have designed and implemented three different versions for the purpose of exploring the design space. The first version is based on our multi-threaded RAM-based cache-aware algorithm, the second one uses external sorting, and the third one uses a RAM-buckets technique. The version based on our cache-aware algorithm outperforms the external-sorting-based algorithm by a factor of 11 and the RAM-buckets-based algorithm by a factor of 2. These results are observed on a 16-node cluster for permutations on 1.6 billion elements. More details and experimental results are presented in Chapter 4.

1.5 Emerging NUMA Architectures used in Computational Science

We briefly review the emerging NUMA architectures and their programming models, as used in high-performance computational science:

Parallel disks. There are a variety of programming language extensions and libraries for parallel disk-based programming, such as: MapReduce [DG04], Hadoop [Had], as well as their extensions (MapReduce Online [CCA⁺10], Twister [ELZ⁺10], HaLoop [BHBE10], Map-Reduce-Merge [YDHP07], etc.); Roomy [Kun10]; or Pregel [MAB⁺10].

MapReduce and Hadoop employ the *map* and *reduce* operations. *Map*

applies a user-defined function to each element of a data collection and produces key-value pairs. *Reduce* applies another user-defined function to accumulate all outputs of *map* that have the same key. The MapReduce and Hadoop runtime systems invoke a global barrier after each *map* and *reduce* stage. MapReduce Online makes these barriers optional. Twister and HaLoop introduce iterative constructs to MapReduce, such as setting the number of iterations and checking for fixed point conditions. Map-Reduce-Merge adds the *merge* operation to MapReduce. *Merge* can accumulate two different types of *reduce* output.

Roomy provides an interface to large distributed data structures to the user. These distributed data structures include: RoomyList, RoomyArray, and RoomyHashTable. Any element in a data structure can be accessed using the *access* functions and updated using the *update* functions. The *Roomy_sync* function invokes a global barrier and forces the execution of pending accesses and updates.

In Pregel, the programmer defines the behavior of the vertices of an explicit graph. The programmer implements a Vertex class and its methods, such as *Compute*, *vertex_id*, *SendMessageTo* (and others). The computation is executed in supersteps, until a halting condition is met.

NVIDIA GPUs. An NVIDIA GPU has hundreds of small, simple SIMD cores on a single card. A group of cores on a single chip defines a Streaming Multiprocessor (SM).

The widest-used programming model for NVIDIA GPUs is CUDA [NV1b], which is based on CUDA *kernels*. In CUDA, *kernels* organize the computation as a grid of many threads. The grid is split into thread blocks. The basic scheduling and execution unit is a warp (consisting of 32 threads). All threads in a warp execute the same instruction in lockstep. A single SM will

execute exactly one thread block at any given moment. The threads have access to global GPU RAM, Level-2 cache shared between the SMs and shared memory/Level-1 cache local to each SM.

The OpenCL [Khr08] library is related to CUDA, but it can be compiled for multiple architectures, including AMD Fusion. OpenCL also uses kernels, that are similar to CUDA's. The OpenCL *work-group* concept is similar to the CUDA thread block, while the OpenCL *work-item* concept is similar to the CUDA thread. In OpenCL, concepts such as global memory, local memory, and private memory are abstractions of GPU global memory, per-SM shared memory/Level-1 cache, and registers in CUDA.

OpenACC [Opea] and HMPP [DBB] are compiler-directive-based approaches for GPU programming. In both cases, C code is annotated with pragma directives that will be interpreted by the compiler and translated into GPU code. Examples of pragmas in OpenACC are: `#pragma acc parallel` (which launches multiple worker groups; each group can have either vector or SIMD operations; the programmer specifies the number of work groups (gangs), the number of workers per group, as well as the type of parallelism), `#pragma acc kernels` (which defines a GPU-code region), and `#pragma acc loop` (which defines loops in a kernel), among others. HMPP also uses pragmas similar to OpenACC (such as `#pragma hmppcg parallel`), but introduces extra pragmas, such as: `#pragma hmpp codelet` (that allows programmers to define multiple architecture-specific implementations for the same task), or `#pragma hmpp map, args` (for sharing data between codelets).

For a more detailed description of NVIDIA GPUs and CUDA, see Section 3.1.

AMD GPUs. Newer generations of GPUs produced by AMD (the “Southern Islands” architecture) consist of around 20 SIMD engines. (The previous

generation of AMD GPUs used a VLIW (very long instruction word) architectures, as opposed to the SIMD approach.) Each SIMD engine consists of multiple thread processors, and each thread processor contains a small number of stream cores. The stream cores on a thread processor share a set of registers, while all the thread processors on a SIMD engine share a small local memory and an L1 cache. Conceptually, newer AMD GPUs are relatively similar to NVIDIA GPUs.

The AMD Fusion architecture places an AMD GPU and a CPU on the same die, as opposed to NVIDIA GPUs, which are connected to the CPU via a PCIExpress bus on the motherboard. This way, as opposed to NVIDIA GPUs, AMD Fusion avoids one of the problems of CPU-GPU computing, which is the inefficiency of data transfers between the CPU and the GPU.

The widest-used programming language for AMD GPUs is OpenCL [Khr08]. OpenACC [Opea] and HMPP [DBB] will also be available for AMD GPUs. As of now, AMD GPUs have a much smaller share of the High-Performance Computing market, as opposed to NVIDIA GPUs. Time will tell if AMD GPUs are successful in high-performance computing.

Many-core CPUs. Current commodity many-core CPUs consist of 4 or 8 cores on a chip. Each of the cores has a few kilobytes of local Level-1 cache. All cores on the chip share a Level-2 cache with a size of a few megabytes. Many-core machines that place 2 or 4 of these many-core CPUs on the motherboard are common.

The Intel Many-Integrated Core (MIC) architecture is a many-core CPU with up to 80 *x86*-compatible cores. Intel MIC is a direct competitor of GPUs: the Intel MIC promises similar performance to a high-end GPU, but its advantage is that large parts of legacy code can be reused, since its cores are *x86*-compatible. Other vendors have not yet developed many-core CPUs with

such a high level of parallelism. Intel MIC architectures have not yet been used on a large scale, so it is too early to say how the HPC community views them in comparison with GPUs.

OpenMP [Opeb] is a compiler-directive-based approach for writing multi-threaded code for many-core architectures. Some OpenMP directives are: `#pragma omp parallel` for generating worker threads from a master thread, `#pragma omp barrier` for global synchronization among threads, `#pragma omp critical` to define an exclusive-access critical section, or `#pragma omp master` to define the behavior of the master thread. Cilk [BJK⁺96, Int] is a library and programming model for many-core CPUs. Cilk employs two primitives: *spawn* and *sync*. *Spawn* generates sub-tasks of the current task, while *sync* waits for multiple sub-tasks to complete. Cilk implements a *work-stealing* mechanism for load balancing. Cilk and OpenMP are available for both commodity many-core CPUs and the Intel MIC.

1.6 Thesis Organization

The rest of the dissertation is organized as follows: Chapter 2 presents the parallel disk-based NFA determinization and DFA minimization, along with an application of the algorithms to a forbidden-permutations problem; Chapter 3 presents our port of the MADNESS integral operator (known as **Apply**) to Hybrid CPU-GPU architectures; Chapter 4 presents our efficient permutation multiplication algorithms for many-core CPUs and for parallel disks; and Chapter 5 presents lessons learned from the NUMA-based solutions for the three real-world problems addressed in this dissertation, including the fact that all three solution implementations exhibit *map-reduce*-like control flow. Appendix A presents a step-by-step correspondence between the NUMA-based

algorithms proposed in this dissertation and the high-level control-flow components of *map-reduce*.

CHAPTER 2

Algorithms for Large Finite State Automata

We present parallel disk-based algorithms for the determinization of non-deterministic finite state automata (NFA) and minimization of deterministic finite state automata (DFA).

It is well-known that the determinization of an NFA can lead to a combinatorial explosion of states and, thus, to DFAs much larger than the available RAM size (in our case, the largest DFA obtained had 2 billion states and 14 transitions per state).

To address the combinatorial explosion issue, a parallel disk-based algorithm for NFA determinization has been implemented. In order for the resulting DFAs to be of practical use, they needed to be minimized to their canonical minimal DFA. For this purpose a parallel disk-based DFA minimization algorithm has been implemented.

The two algorithms for the parallel-disks NUMA architecture were adapted from existing algorithms for uniform memory architectures. The parallel-disks determinization algorithm was adapted from the well-known sequential *subset construction* algorithm. The parallel-disks minimization algorithm was adapted from a parallel algorithm designed for the CM-5 machine.

This work was motivated by a series of problems in the area of permu-

tation patterns, that can be reduced to processing of token passing networks represented as finite state automata. Central to this series of problems was the determinization of input non-deterministic finite state automata (FSA), since part of the processing required deterministic finite state automata (DFA) as input.

This chapter is organized as follows: Section 2.1 presents background on finite state automata, determinization and minimization; Section 2.2 presents background on permutation patterns and token passing networks; Section 2.3 introduces our parallel disk-based determinization algorithm; Section 2.4 introduces our parallel disk-based minimization algorithm; Section 2.5 briefly describes a few challenges that were overcome in the implementation of the parallel disk-based algorithms; Section 2.6 introduces multi-threaded algorithms for determinization and minimization for shared-memory architectures; Section 2.7 discusses the application of two NFA minimization algorithms to some of our input NFAs; Section 2.8 presents experimental results for implementations of the parallel disk-based determinization and minimization algorithms; and Section 2.9 presents related work relevant to the determinization and minimization of finite-state automata.

2.1 Terminology and Background

Finite state automata and the closely related concepts of regular languages and regular expressions form a crucial part of the infrastructure of computer science. Among the rich variety of applications of these concepts are natural language grammars, computer language grammars, hidden Markov models, digital logic, transducers, models for object-oriented programming, control systems, and speech recognition.

This section motivates the need for efficient, scalable algorithms for *finite state automata* (FSA), by noting that they are usually the most computationally tractable form in which to analyze the regular languages that arise in many branches of computer science. That analysis requires efficient algorithms both for determinization of NFA (conversion of NFA to DFA) and minimization of DFA.

Recall that a *deterministic finite state automaton* (DFA) consists of a finite set of states with labelled, directed edges between pairs of states. The labels are drawn from an associated alphabet. For each state, there is at most one outgoing edge labelled by a given letter from the alphabet. So, a transition from a state dictated by a given letter is *deterministic*. There is an initial state and also certain of the states are called *accepting*. The DFA accepts a word if the letters of the word determine transitions from the initial state to an accepting state. The set of words accepted by a DFA is called a *language*.

A *non-deterministic finite state automaton* (NFA) is similar, except that there may be more than one outgoing edge with the same label for a given state. Hence, the transition dictated by the specified label is non-deterministic. The NFA accepts a word if there exists a choice of transitions from the initial state to some accepting state.

More formally, a DFA is a *5-tuple* $(\Sigma, Q, q_0, \delta, F)$, where Σ is the input alphabet, Q is the set of states of the automaton, $q_0 \in Q$ is the initial state, and there is a subset of Q , called the *final* or *accepting* states, F . $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, which decides which state the control will move to from the current state upon consuming a symbol in the input alphabet.

An NFA is a *5-tuple* $(\Sigma, Q, q_0, \delta, F)$. The only difference from a DFA is that $\delta : Q \times \Sigma \rightarrow \mathfrak{P}(Q)$. Upon consuming a symbol from the input alphabet, an NFA can non-deterministically move control to any one of the defined next

states.

Recall that the *subset construction* allows one to transform an NFA into a corresponding DFA that accepts the same words. Each state of the DFA is identified with a subset of the NFA states. Given a state A of the DFA and an edge with label α , the destination state B consists of a subset of all states of the NFA having an incoming edge labelled by α and a source state that is a member of the subset A.

Finite state automata are an important computationally tractable representation of *regular languages*. This class of languages has a range of valuable closure properties, including under concatenation, union, intersection, complementation, reversal and the operations of (not necessarily deterministic) transducers. (A *transducer* is a DFA or NFA that also produces output letters upon each transition.) The above properties have algorithmic analogues that operate on finite state automata. For instance, given an FSA representing a language it is easy to construct one for the reversed language. So, one can compute various operations on regular languages by computing the analogous operations on their finite state automaton representations.

Using these operations to manipulate regular languages forces one to choose between a DFA and an NFA representation. But neither representation suffices. Some of the above operations on finite state automata, such as complementation, require input in the form of a DFA. And yet, some operations may transform a DFA into an NFA.

From a computability standpoint, there is no problem. The subset construction converts between an NFA and the more specialized DFA. But while the subset construction is among the best known algorithms of an undergraduate curriculum, it may also lead to an exponential growth in the number of states. This is usually the limiting factor in determining what computations

are practical. In some cases this problem is completely unavoidable, since there are families of non-deterministic automata whose languages cannot be recognized by any deterministic automata without exponentially many states. In many cases of interest, however, much smaller equivalent deterministic automata do exist. But the determinization process alone is not enough to reduce the DFA to the equivalent unique minimal DFA.

It is this large intermediate data which motivates us to consider parallel disk-based computing. To address the state-space explosion, we developed a parallel disk-based FSA package in Roomy, which can determinize much larger NFAs compared to what can be done in RAM even on a large shared memory machine [SKCL11]. The package also minimizes the intermediate DFA obtained by subset construction, thus making it small enough to be used in RAM for the remaining steps of the computation. The results obtained from the parallel disk-based computations we ran provided important new information for a series of problems in the area of permutation patterns.

This work attempts to relieve the critical bottleneck in many automata-based computations by providing a scalable disk-based parallel algorithm for computing the minimal DFA accepting the same language as a given NFA. This requires the construction of an intermediate non-minimal DFA whose, often very large, size has been the critical limitation on previous RAM-based computations. Thus, researchers may use a departmental cluster or a SAN (storage area network) to produce the desired minimal DFA off-line, and then embed that resulting small DFA in their production application. As a motivating example, Section 2.8 demonstrates the production of a two-billion state DFA that is then reduced to a minimal DFA with less than 800,000 states — a more than 1,000-fold reduction in size. Part of the difficulty of producing the 2 billion-state DFA by the subset construction is that each DFA state consists

of a subset that includes up to 20 of the NFA states. Hence, each DFA state needs a representation of 80 bytes (4×20).

As for the DFA minimization, Hopcroft [Hop71] provided an efficient $O(n \log n)$ algorithm, but the algorithm does not adapt well to parallel computing. An efficient parallel $O(n \log^2 n)$ algorithm has been used in the 1990s [RX96], but ultimately the lack of intermediate storage for the subset construction has prevented researchers from adapting these techniques for use within the varied applications described above.

The DFA minimization algorithm we use is based on an iterative, RAM-based parallel algorithm used on supercomputers of the 1990s. We adapt that algorithm both to clusters of modern commodity computers and to a multi-threaded algorithm for modern many-core computers.

Disk-based parallel algorithms are presented for both NFA determinization (Section 2.3) and DFA minimization (Section 2.4), both using *streaming* access to data distributed evenly across the parallel disks of a cluster. This avoids the latency penalty that a random access to disk incurs. Separately, Section 2.6 presents a depth-first based algorithm for determinization and minimization suitable for large shared-memory computers.

2.2 Stacks, Token Passing Networks and Forbidden Patterns

The study of what permutations of a stream of input objects could be achieved by passing them through various data structures goes back at least to Knuth [Knu68, Section 2.2.1], who considered the case of a stack and obtained a simple characterization and enumeration in this case. Knuth's characterization uses the notion of *forbidden substructures*: a permutation can be achieved by a stack

if and only if it does not contain any three numbers (not necessarily consecutive) whose order within the permutation, and relative values match the pattern high-low-middle (usually written 312). For instance 41532 cannot be achieved because of 4, 1 and 2. This work has spawned a significant research area in combinatorics, the study of permutation patterns [LRV10] in which much beautiful structure has been revealed. Nevertheless, many problems very close to Knuth's original one remain unresolved: in particular there is no similar characterization or enumeration of the permutations achievable by two stacks in series (it is not even known if 2-stack achievability can be tested in polynomial time). A number of authors have investigated restricted forms of two-stack achievability [Bon03] including the case of interest here, where the stacks are restricted to finite capacity. In this case, the stacks can be modeled as *token passing networks* (see for example Figure 2.1), as introduced in [ALT97]. These techniques allow the classes of achievable permutations and the forbidden patterns that describe them to be encoded by regular languages and manipulated using finite state automata using a collection of GAP [GAP08] programs developed by Steve Linton and Michael Albert.

A token passing network is a directed graph with designated input and output vertices. Numbered tokens are considered to enter the graph one at a time at the input vertex, and travel along edges in the appropriate direction. At most one token is permitted at any vertex at any time. The tokens leave the graph one at a time at the output vertex. A permutation $\pi \in S_n$ is called *achievable* for a given network if it is possible for tokens to enter in the order $1, \dots, n$ and leave in the order $1\pi, \dots, n\pi$.

In previous work, Steve Linton explored the cases of stacks of depths 2 and depth k (as seen in Figure 2.1) for a range of values of k and observed that for

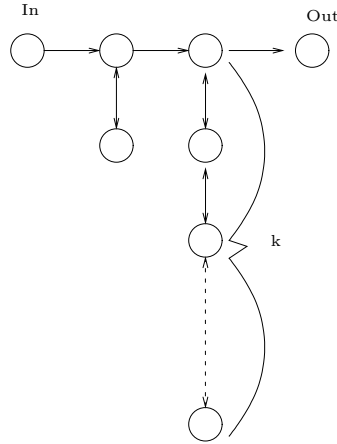


Figure 2.1: A 2-stack followed by a k -stack represented as a token passing network.

large enough k the sets of minimal forbidden patterns appeared to converge to a set of just 20 of lengths between 5 and 9, which were later proved [Eld06] to describe the case of a 2-stack and an infinite stack.

The application that motivates the calculations in this chapter is a step towards extending this result to a 3-stack and an infinite stack, by way of the slightly simpler case of a 3-buffer (a data structure which can hold up to three items and output any of them). This configuration is shown in Figure 2.2.

Computations had been completed on various sequential computers for a 3-buffer and a k -stack for $k \leq 8$, but this was not sufficient to observe convergence. The examples considered in this chapter are critical steps in the computations for $k = 9$, $k = 10$, $k = 11$ and $k = 12$. Based on the results of these computations we are now able to conjecture with some confidence a minimal set of 12,636 forbidden permutations of lengths between 7 and 18 for a 3-buffer and an infinite stack.

The computations required for these investigations are those implied by Corollary 1 of [AAR03, p. 96]. By modelling the token passing network in the style of [ALT97], slightly optimized to avoid constructing so many redundant

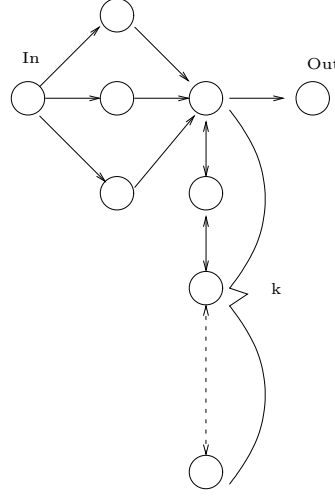


Figure 2.2: A 3-buffer followed by a k -stack represented as a token passing network.

states, we can construct (an automaton representing) a language L describing the permutations achievable by our network, and we wish to construct a language B describing the minimal forbidden patterns. Each state of L represents a configuration of the network and the labels on the transitions represent (rank encoded) output symbols, if any. Combining results from [AAR03] and simplifying the notation a little we find

$$B = \left(L^{RC} \cap (L^{RC} D^t)^C \right)^R$$

where R denotes left-to-right reversal, C denotes complementation and D is the deletion transducer described in [AAR03]. Each step of this computation can be realized by standard algorithms using finite state automata, but, as observed above, with frequent recourse to determinization (to allow complements) and minimization (to control explosion in the number of states). As the computations become larger, the limiting step turns out to arrive after the application of the transposed deletion transducer and before the next complementation, and it is this step that is the object of this chapter.

2.3 NFA Determinization via Subset

Construction for large NFAs

To generate the intermediate, large DFA, the recursive subset-construction algorithm was migrated to parallel disks. A high-level description of sequential recursive subset construction is presented in Algorithm 1.

Input: Initial *NFA*, with initial state s_i , transition-label set T , transition function δ and accepting states A_s .

Output: *DFA*, represented as a list, equivalent to *NFA*.

- 1: Create subset $I \leftarrow \{s_i\}$, and an integer Id for it (Id_I).
- 2: Insert pairs (I, Id_I) in a hash table of visited subsets, *visited*.
- 3: Create *DFA* — a list that will store the resulting DFA, containing 3-tuples of the form (Id, t, Id_{next}) , meaning transition t takes state with Id to state with Id_{next} .
- 4:
- 5: Call *subset_construct*(I, Id_I).
- 6:
- 7: **func** *subset_construct*(subset S, Id_S):
- 8: **for** each *NFA* transition label $t \in T$ **do**
- 9: Create empty subset S_{next} .
- 10: **for** each state s in subset S **do**
- 11: **for** each transition label $t \in T$ **do**
- 12: $s_{next} \leftarrow \delta(s, t)$ (in the *NFA*).
- 13: $S_{next} \leftarrow S_{next} \cup \{s_{next}\}$.
- 14: **end for**
- 15: **end for**
- 16: **if** S_{next} is already in *visited* **then**
- 17: Get $Id_{S_{next}}$ from *visited* and add 3-tuple $(Id_S, t, Id_{S_{next}})$ to *DFA*.
- 18: **else**
- 19: Create a new Id for S_{next} , add pair $(S_{next}, Id_{S_{next}})$ to *visited* and add $(Id_S, t, Id_{S_{next}})$ to *DFA*.
- 20: Recursively call *subset_construct*($S_{next}, Id_{S_{next}}$).
- 21: **end if**
- 22: **end for**
- 23: **end func**

Algorithm 1: RAM-based, Recursive Subset Construction.

Note that the recursive call from line 17 of Algorithm 1 is made for each newly discovered subset, which makes the branching factor of the recursion

variable.

We migrated Algorithm 1 to Algorithm 2, which we implemented in Roomy and ran on a computer cluster with 29 nodes. Experimental results which, to our knowledge, report the largest successfully carried out subset construction compared to previous literature, are presented in Section 2.8.

Input: Initial *NFA*, with initial state s_i , transition-label set T , transition function δ and accepting states A_s .

Output: *DFA*, represented as a list, equivalent to *NFA*.

- 1: Create subset $I \leftarrow \{s_i\}$, and an integer Id for it (Id_I).
- 2: Insert pairs (I, Id_I) in a hash table of visited subsets, *visited*.
- 3: Create *DFA* — a list that will store the resulting DFA, containing 3-tuples of the form (Id, t, Id_{next}) , meaning transition t takes state with Id to state with Id_{next} .
- 4: Add pair (I, Id_I) into *current_frontier* and create *next_frontier*, initially empty.
- 5: Call *pardisk_subset_construct*().

Algorithm 2: Parallel Disk-based Subset Construction.

- 6: **func** *pardisk_subset_construct*():
- 7: **while** *current_frontier* is not empty **do**
- 8: Call *neighbors*(S, Id_S) for each pair (S, Id_S) in *current_frontier*, which returns a BATCH of S_{next} subsets.
- 9: Perform delayed duplicate detection on the BATCH of S_{next} subsets with respect to *visited* and, for already visited subsets, obtain their Id, while for newly discovered subsets, create new Ids.
- 10: Insert all new $(S_{next}, Id_{S_{next}})$ in *visited*.
- 11: Add all $(Id_S, t, Id_{S_{next}})$ to *DFA*, for all t , regardless of whether S_{next} is new or not.
- 12: Add all new S_{next} to *next_frontier*.
- 13: Remove contents of *current_frontier* from parallel disks.
- 14: Rename *next_frontier* to *current_frontier*.
- 15: Create an empty list with name *next_frontier*.
- 16: **end while**
- 17: **end func**

Algorithm 3: FUNCTION *pardisk_subset_construct* of Parallel Disk-based Subset Construction.

For subset construction on parallel disks, three parallel disk-based Roomy hash tables are used: *visited*, *frontier* and *next_frontier*. Hash table keys

are sets of states of the NFA, and hash table values are unique integers.

Parallel breadth-first search (BFS) is used to compute the states of the intermediate DFA. Duplicate states in each BFS frontier are removed by delayed duplicate detection. The parallel disk-based computation follows a *scatter-gather* pattern in a loop: local batch computation of neighbors; send results of local computation to other nodes; receive results from other nodes; and perform duplicate detection.

Data that needs to be sent to other compute nodes by Roomy is first buffered in local RAM, in *buckets* corresponding to each compute node. For a given piece of data, Roomy uses a hash function to determine which compute node should process that data and, hence, in which bucket to buffer that data. Once a given buffer is full, the data it contains is sent over the network to the corresponding compute node (or to the local disk, if the data is to be processed by the local node).

```

18: func neighbors(subset  $S$ ,  $Id_S$ ) returns neighbor subsets:
19: Create ordered set of subsets  $SET$ , initially empty.
20: for each NFA transition  $t$  do
21:   Create empty subset  $S_{next}$ .
22:   for each state  $s$  in subset  $S$  do
23:     for each transition  $t$  do
24:        $s_{next} \leftarrow \delta(s, t)$  (in the NFA).
25:        $S_{next} \leftarrow S_{next} \cup \{s_{next}\}$ .
26:     end for
27:   end for
28:   Add  $S_{next}$  to  $SET$ .
29: end for
30: Return  $SET$ .
31: end func

```

Algorithm 4: FUNCTION *neighbors* of Parallel Disk-based Subset Construction.

Algorithm 2 was adapted from Algorithm 1. In Algorithm 3 (the continuation of Algorithm 2), function *pardisk_subset_construct* is executed by all the nodes of the cluster in parallel.

2.4 Finding the Unique Minimal DFA

The algorithm used for computing the minimal DFA on parallel disks is based on a parallel RAM-based algorithm used on the CM-5 supercomputers in the late 1990s and early 2000s [JR93, RX96, TSG02]. We call this the *forward refinement* algorithm. The central idea of the algorithm is to iteratively refine partitions of states of the given DFA. The algorithm is proved to converge to a stable set of partitions. Upon convergence, the set of partitions, together with the transitions between partitions, form a graph that is isomorphic to the minimal DFA.

At the beginning of the algorithm, the DFA states are split into two partitions: the accepting states and the non-accepting states. A hash table of visited partitions, *parts*, is used, with pairs of integers as keys and integers as values. For the pair of integers, the first integer represents the partition number of the current state i , while the second integer represents the partition number of $DFA[i][T]$, where T is the current transition being processed. If two states i and j in the DFA are equivalent, then for any transition T , at any time during the iterative process, the pairs corresponding to i and j for the same T should have the same first integers and the same second integers.

Algorithm 5 describes a sequential RAM-based version of *forward refinement*, while Algorithm 6 describes the parallel disk-based version.

The major differences between Algorithms 5 and 6 are that lines 7–17 and line 20 of Algorithm 5 are parallelized and that Roomy’s principles of parallel disk-based computing are used: all large data structures are split into equally-sized chunks that are kept on the parallel disks of a cluster; and all access and update operations to the *current_parts* and *prev_parts* arrays and to the *parts* hash table are delayed and batched for efficient streaming access to disk. Also, duplicate detection, which in the sequential RAM-based algorithm appears in

Input: A DFA $init_{DFA}$, with N states, with initial states I_s and accepting states A_s .

Output: The minimal canonical DFA min_{DFA} , equivalent to $init_{DFA}$.

- 1: Initialize array *current_parts*: $current_parts[i] \leftarrow 0$ if i is a non-accepting state of $init_{DFA}$, and $current_parts[i] \leftarrow 1$ if i is an accepting state.
- 2: Initialize array *next_parts* to all 0.
- 3: $prev_num_parts \leftarrow 0$; $curr_num_parts \leftarrow 2$.
- 4: **while** $prev_num_parts < curr_num_parts$ **do**
- 5: $prev_num_parts \leftarrow curr_num_parts$.
- 6: **for** each transition T of $init_{DFA}$ **do**
- 7: Initialize hash table *parts* to \emptyset .
- 8: $next_id \leftarrow 0$.
- 9: **for** $i \in \{1 \dots N\}$ **do**
- 10: $next_part \leftarrow current_parts[init_{DFA}[i][T]]$.
- 11: $pair \leftarrow new\ Pair(current_parts[i], next_part)$.
- 12: $id \leftarrow parts.getVal(pair)$.
- 13: **if** id was not found in *parts* **then**
- 14: Insert $(pair \rightarrow next_id)$ in *parts*.
- 15: $id \leftarrow next_id$.
- 16: $next_id \leftarrow next_id + 1$.
- 17: **end if**
- 18: $next_parts[i] \leftarrow id$.
- 19: **end for**
- 20: $current_parts \leftarrow next_parts$.
- 21: **end for**
- 22: $curr_num_parts \leftarrow next_id$.
- 23: **end while** // For each state i of $init_{DFA}$, $current_parts[i]$ defines what partition state i is in.
- 24: Collapse each partition to just one state to obtain the minimal DFA.

Algorithm 5: Sequential RAM-based Forward Refinement.

lines 12–17, is replaced by delayed duplicate detection.

Note that in Algorithm 6 each compute node k keeps its own part of the *parts* hash table ($parts^k$) and owns a part of the intermediate DFA states ($states^k$). As with subset construction, the parallel disk-based computation follows a scatter-gather pattern, denoted in the pseudocode by most of the *for* loops: local computation (line 5), *scatter* (line 6), *gather* (line 8), local computation (lines 9–11), *scatter* (line 12), *gather* (line 13), local computation (lines 15–16), *scatter* (lines 17, 19), *gather* (line 22) and local computation

Input: A DFA $init_{DFA}$, with N states, with initial states I_s and accepting states A_s .

Output: The minimal canonical DFA min_{DFA} , equivalent to $init_{DFA}$.

- 1: // Initialization and outer loop are the same as lines 1–6 in Algorithm 5
- 2: // Disk-based parallel loop (parallelization of lines 7–17 in Algorithm 5)
 - each node k does:
- 3: Initialize hash table $parts^k$ to \emptyset .
- 4: **for** $i \in \{states^k\}$ **do**
- 5: $k_1 = hash(init_{DFA}[i][T])$.
- 6: Send tuple $(i, current_parts[i], init_{DFA}[i][T])$ to node k_1 (batched operation).
- 7: **end for**
- 8: Receive batches of tuples $(i, current_parts[i], init_{DFA}[i][T])$ from all nodes.
- 9: $next_part[i] \leftarrow current_parts[init_{DFA}[i][T]]$.
- 10: $pair[i] \leftarrow new\ Pair(current_parts[i], next_part[i])$.
- 11: $next_id[i] \leftarrow new\ Id$.
- 12: Send new entry $(pair[i] \rightarrow next_id[i])$ and state id i to node $k_2 = hash(pair[i])$ (batched operation).
- 13: Receive batches of $pair \rightarrow id$ entries from all nodes.
- 14: **for** each received entry $pair \rightarrow recv_id$ and associated state id i **do**
- 15: **if** an entry $pair \rightarrow id$ was not found in the local $parts$ **then**
- 16: Insert $(pair \rightarrow recv_id)$ in the local $parts$.
- 17: Send key–value pair $i \rightarrow recv_id$ to node $k = hash(i)$ (batched operation).
- 18: **else**
- 19: Send key–value pair $i \rightarrow id$ to node $k = hash(i)$ (batched operation).
- 20: **end if**
- 21: **end for**
- 22: Receive batches of $i \rightarrow id$ entries from all nodes.
- 23: **for** each received entry $i \rightarrow id$ **do**
- 24: $current_parts[i] \leftarrow id$.
- 25: **end for**

Algorithm 6: Parallel Disk-based Forward Refinement.

(line 24).

The last part of finding the minimal DFA, in which each partition collapses to one state, is presented separately, in Algorithm 7.

```
1: // Collapsing partitions to  $min_{DFA}$  (parallelization of line 20 in
   Algorithm 5) – each node  $k$  does:
2: for  $i \in \{indices^k\}$  do
3:   Get  $partition[i]$  (the partition of state  $i$ ) from  $current\_parts^k$ .
4:   for each transition  $T$  of  $init_{DFA}$  do
5:     Get  $partition[init_{DFA}[i][T]]$  from node that owns it.
6:   end for
7:   // Now all the transitions of  $partition[i]$  in  $min_{DFA}$  are known.
8: end for
```

Algorithm 7: Parallel Disk-based Partitions Collapse.

2.5 Implementation Issues for the FSA

Algorithms

Large parallel computations are often subject to practical issues that need to be resolved in order for the computations to succeed. These are generally implementation aspects that need to be optimized.

In the case of the parallel disk-based FSA algorithms, the first important implementation question we faced was the representation of the sets of NFA-states. For determinizations in which sets tend to be small, an explicit state representation is preferable. For determinizations in which sets tend to be large (i.e. a significant fraction of the total NFA states), a bitmap representation of the sets is desired.

For the series of permutation patterns problems that are the object of the solution, small problems in the series were solved using sequential implementations by Steve Linton (see Section 2.2). Those solutions showed that the determinizations for this series of problems tend to produce small sets of NFA-states. Therefore, the explicit set of states representation was chosen.

Moreover, by observing a pattern of the largest produced set when increasing the size of the problem, we were able to make confident assumptions on the size of the largest produced set for the next problems in the series. If the

determinization produced a set larger than the maximum specified set size, the program would print this information and it would exit. This behaviour is a consequence of the usage of Roomy as an implementation platform. In Roomy, elements of a data structure have a fixed size. Any element smaller than the fixed size will be padded until it reaches the fixed size. This is a consequence of using files on disk. The problem is that elements larger than the fixed size are not accepted. So we need to over-provision for the largest element. In practice, this over-provisioning leads to an allocated to used space ratio of up to 2 for the sets of states for the series of problems that we solved. For input NFAs in general, the allocated to used space ratio using this representation may be much higher.

Another implementation issue occurs for the detection of duplicate sets in Algorithm 6 (lines 12–16). For the series of NFAs that we are concerned with, in the intermediate DFAs more than half of the states are accepting states. For the DFAs that we minimized as part of the series of permutation pattern problems, all the accepting states were determined by the DFA minimization to belong to the same partition. This means they collapse to a single accepting state at the end of the DFA minimization. In line 12 of Algorithm 6 all accepting states are sent to the same cluster node (because the hash function produces the same result for the same input) for duplicate removal. Therefore, the disk of that specific cluster node can fill up. At that point, the Roomy computation will crash. This has happened for the largest computation described in Section 2.8. The solution was to partition the accepting states into a few sets, and process the sets on the target cluster node one at a time. This ensured that there was never too much data to fill up the disk. Based on the observation that, for these cases, all accepting states collapse into one state in the minimal DFA, Steve Linton proved that this is true of all problems in the

series. A significant optimization was then to recognize that an accepting state is a duplicate of a canonical state upon discovery. This significantly reduces the amount of storage necessary for the discovered states. The experimental results in Section 2.8 do not take this last optimization into account, because these results were obtained before the proof was completed.

2.6 Multi-threaded Implementations for Shared Memory

For comparison with the parallel disk-based algorithms, multi-threaded shared-memory implementations of subset construction and DFA minimization are provided. A shared-memory architecture almost always has less storage (128 GB RAM in our experiments) than parallel disks. To alleviate the state combinatorial explosion issue, depth-first search (DFS) is used here for the subset construction instead of breadth-first search (BFS).

The smallest instance of the four NFA to minimal DFA problems considered can be solved on a commodity computer with 16 GB of RAM, the second instance needs 40 GB of memory for subset construction, while the third largest instance needs a large shared-memory machine with at least 100 GB of RAM. The largest instance considered cannot be solved even on a large shared-memory machine, thus requiring the use of parallel disks on a cluster.

A significant problem for both subset construction and DFA minimization in a multi-threaded environment is synchronization for duplicate detection. For subset construction, this issue arises when a thread discovers a new DFA state and checks whether the state has been discovered before by itself or another thread. The data structure keeping the already-discovered states (usually a hash table) has to be locked in that case, so that the thread can check

whether the current state has already been discovered and, if not, to insert the new state in the data structure. However, such an approach would lead to excessive lock contention (many threads waiting on the same lock).

Hence, the solution employed was to use a partitioned hash table to keep the already discovered states instead of a regular hash table. For large problem instances, the hash table was partitioned into 1024 separate hash tables — each with its own lock. So long as the number of partitions is much larger than the number of threads, it is unlikely that two threads will concurrently discover two states that will belong to the same partition, thus avoiding most of the lock contention. Experiments (see Section 2.8, Table 2.2) show significant speedup with the increase in number of threads.

A similar solution was used for the forward refinement algorithm, which minimizes the DFA obtained from subset construction. In this case, read accesses significantly dominate over write accesses. The implementation took advantage of this by implementing a lock only around writes to the corresponding hash table. The valid bit was written last in this case. A write barrier is needed to guarantee no re-ordering of writes. In the worst case, a concurrent read may read the hash entry as invalid, and that thread will then request the lock, verify that the hash entry is still invalid, and if that is the case, then do the write. This is safe.

2.7 NFA Minimization: an Alternative Approach

NFA minimization attempts to reduce the size of the input NFA, which should, in turn, lead to a decrease in number of states of the intermediate DFA.

Computing the minimal equivalent NFA of a given NFA is PSPACE-

hard [BM08], but a series of methods have been developed that, for many input NFA, compute a smaller equivalent NFA. This smaller NFA is often the minimal NFA with respect to a certain simulation relation.

One NFA minimization method uses simulation equivalence [GP08b], that establishes a simulation relation used to minimize the input NFA prior to determinization. In Glabbeek and Ploeger's experiments [GP08b], this approach produced NFAs up to four times smaller than the input NFA. The intermediate DFAs produced by subset construction from the reduced NFAs were up to 3 times smaller than the intermediate DFAs produced by subset construction from the non-reduced NFAs. When NFA reduction was followed by a *compression* method in the determinization step, the experiments in [GP08b] show an up to 100 times reduction in the intermediate DFA size.

Simulation equivalence for NFA minimization is generally preferred over other methods, such as bisimulation equivalence or trace inclusion (see [GPP03, Section 3.2]). According to [GP08a], simulation preorder “*is the coarsest pre-order included in trace inclusion that is known to be decidable in polynomial time*”.

The NFA minimization method of [GP08b] was not successful on our input NFAs. The minimized NFA still had more than 90% of the original number of states. More importantly, the NFA minimization process for our input NFAs was slow: it took 11 minutes to minimize an NFA with 3,500 states and more than 24 hours to minimize an NFA with 50,000 states. By contrast, the parallel disk-based NFA determinization and minimization took 2 minutes for the 3,500-state NFA and 1 hour for the 50,000-state NFA when using 4 disks.

When following NFA minimization by determinization using *compression*, the resulting intermediate DFA was much smaller than the intermediate DFA resulted from the parallel disk-based determinization. For example, for the

3,500-state NFA, minimizing the NFA and determinizing using compression resulted in a 5,000-state intermediate DFA, compared to a 650,000-state DFA resulted from parallel disk-based subset construction.

However, since using compression during determinization requires that the NFA be first minimized using simulation equivalence [GP08b], the size reduction benefit is offset by the long time it takes to minimize the NFA. Moreover, determinization using compression is significantly slower than simple determinization on the input NFAs resulted from the permutation patterns problems in Section 2.2.

Another NFA minimization method we tried is the *ReducedNFA* operation in the GAP [GAP08] package. This NFA minimization method is presented in [AL04] and it minimizes the NFA modulo the coarsest right-invariant equivalence relation on the states of the input NFA. The right-invariant equivalence relation was first described in [IY03]. This method had no effect on the size of our input NFAs.

2.8 Experimental Results

2.8.1 Parallel Disk-based Computations

Parallel disk-based computations were carried out on a 29-node computer cluster, each node's processor being a 4-core Intel Xeon CPU 5130 running at 2 GHz. Nodes on the cluster had either 8 or 16 GB of RAM and at least 200 GB of free disk storage and ran Red Hat Linux kernel version 2.6.9.

Table 2.1 presents the sizes of the intermediate DFAs produced by subset construction and the sizes of the minimal DFAs produced by the minimization process for the four considered token passing network problems (corresponding to stack depths 9, 10, 11 and 12).

Table 2.1 also shows the running time and aggregate disk-space used by the subset construction results for the four problem instances. Each state in the intermediate DFA is a subset of states in the original NFA and needs to be kept as such until the subset construction phase is over, for the purpose of exact duplicate detection. Hence, for each newly discovered DFA state, the entire corresponding subset needs to be stored on disk. The average subset size (the sum of all subset sizes divided by the number of subsets) increases slightly with stack depth, from an average of 8.48 states per set for stack depth 9 to 10.06 states per set for stack depth 12. The maximum subset sizes were 17, 18, 19, and 20 for stack depths 9, 10, 11, and 12, respectively.

The intermediate DFA (produced by subset construction) was then minimized using the forward refinement algorithm. Experimental results for DFA minimization are presented also in Table 2.1. For each of the four problem instances, the computation required exactly five forward refinements (five of the outer iterations described in Algorithm 6).

Table 2.1: Parallel Disk-based Subset Construction.

Stack depth	NFA size (#states)	Intermediate DFA			Minimal DFA		
		Size (#states)	Peak disk	Time	Size (#states)	Peak disk	Time
9	167,143	49,722,541	24 GB	9min	32,561	6 GB	38min
10	537,294	175,215,168	90 GB	29min	95,647	22 GB	2h 42min
11	1,667,428	587,547,014	327 GB	3h 40min	274,752	81 GB	9h 20min
12	5,035,742	1,899,715,733	1,136 GB	1day 12h	774,172	295 GB	1day 8h

Figure 2.3 presents the breadth-first search frontier sizes for the largest case ($k = 12$). This and the other three cases exhibit a thin bell-shaped curve, in contrast to the pear-shaped curve seen for many other implicit graph enumerations.

The DFA minimization times, reported in Table 2.1, grow steadily, almost linearly, with the increase in number of states of the intermediate DFA. On

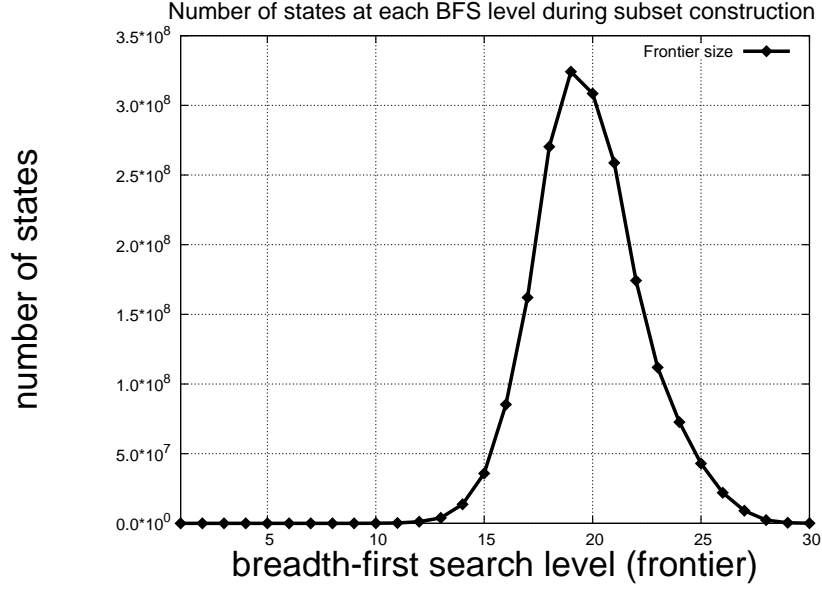


Figure 2.3: Frontier sizes and already-visited hash table sizes for each BFS level of the implicit graph corresponding to subset construction.

the other hand, the subset construction times from Table 2.1 increase much more rapidly. The two smaller cases run faster because the distributed subset construction fits in the aggregate RAM of the nodes of the cluster. The peak disk usage from Table 2.1 divided by 29 (the number of compute nodes) is significantly less than 8 GB per node (the size of RAM) for the two smaller cases and significantly larger than 8 GB for the largest case. It is just above 8 GB per node for the second largest case.

Also note that the peak disk usage is about 4 times lower for DFA minimization compared to subset construction, although the running time is usually higher for minimization and, for the largest case, about the same as for subset construction. This is because although the peak usage is smaller for DFA minimization, it is reached for each of the inner iterations of Algorithm 6. For the largest computation, there are 5 outer iterations, each consisting of 14 inner iterations, so there are $5 \times 14 = 70$ total inner iterations.

2.8.2 Multi-threaded RAM-based Computations

Multi-threaded computations were run on a large shared-memory machine with four quad-core 1.8 GHz AMD Opteron processors (16 cores), 128 GB of RAM, running Ubuntu 9.10 with a SMP Linux 2.6.31 server kernel.

Only the first three computations could be completed on the large shared-memory machine used. The fourth computation requires far too much memory. Table 2.2 shows how the running time of the subset construction and DFA minimization scales with the number of worker threads. The reported timings are for the stack depth 11 problem instance. The size of the intermediate DFA produced by subset construction for this instance is 587,547,014 states. The minimal DFA produced by forward refinements has 274,752 states. For any number of worker threads, the peak memory usage for subset construction was 98 GB, while for minimization it was 36.5 GB.

Table 2.2: Multi-threaded RAM-based timings for stack depth 11.

	Num. threads				
Subset	1	2	4	8	16
constr.	15h 30min	8h 10min	3h 50 min	2h 5min	1h 15min
Minimiz.	8h	5h 5min	2h 40 min	1h 25min	57min
Total	23h 30min	13h 15min	6h 30 min	3h 30min	2h 12min

The timings in Table 2.2 show that both the multi-threaded subset construction and the DFA minimization implementations scale almost linearly with the number of threads. DFA minimization scales almost linearly for up to 8 threads. From 8 to 16 threads it scales sub-linearly due to significant lock contention.

Table 2.3 presents the timings for the problem instances with stack depths 9, 10, and 11, using 16 worker threads. For the stack depth 9 case, the peak memory usage was 12 GB for subset construction and 5 GB for DFA

minimization. For stack depth 10, the peak memory usage was 40 GB and 11 GB, respectively. For stack depth 11, the peak memory usage was 98 GB for determinization and 36 GB for minimization.

Table 2.3: Multi-threaded RAM-based results for stack depths 9, 10, and 11, with 16 worker threads and parallel disk-based results for stack depths 9, 10, and 11, on a 29-node cluster. The parallel disk-based results are presented here for the purpose of comparison.

Stack depth	Multi-threaded Algo. Time			Parallel-disks Algo. Time		
	Subset constr.	DFA min.	Total	Subset constr.	DFA min.	Total
9	8 min	4min 10s	12min 10s	9 min	38 min	47 min
10	25 min	15min	40min	29min	2h 42min	3h 11min
11	1 hr 15 min	57min	2hr 12min	3h 40min	9h 20min	13h

2.9 Related Work

Finite state automata are ubiquitous in mathematics and computer science, and have been studied extensively since the 1950s. Applications include pattern matching, signal processing, token passing networks (including sorting networks), compilers, and digital logic.

While the algorithms for large FSA presented in this chapter have been applied to permutation patterns problems, large finite state machines are also an important tool in natural language processing, and have been used for a wide variety of problems in computational linguistics. In a work presenting new applications of finite state automata to natural language processing [Moh96], Mohri cites a number of examples, including: lexical analysis [Sil94]; morphology and phonology [Kos96]; syntax [Moh94, Roc96]; text-to-speech synthesis [Spr95]; and speech recognition [MPR02, PRS94]. Speech recognition, in particular, can benefit from the use of very large automata. In [Moh97], Mohri predicted:

“More precision in acoustic modeling, finer language models, large lexicon grammars, and a larger vocabulary will lead, in the near future, to networks of much larger sizes in speech recognition. The determinization and minimization algorithms might help to limit the size of these networks while maintaining their time efficiency.”

While the subset construction for determinization has been a standard algorithm since the earliest years, this is not true for the minimization algorithm. For any DFA there is an equivalent minimal canonical DFA [HMU06, Chapter 4.4]. Fast sequential RAM-based DFA minimization algorithms have been developed since the 1950s. A taxonomy of most of these algorithms can be found in [Wat95]. The first DFA minimization algorithms were proposed by Huffman [Huf54] and Moore [Moo56]. Hopcroft’s minimization algorithm [Hop71] is proved to achieve the best possible theoretical complexity ($O(|\Sigma|N \log N)$ for alphabet Σ and number of states N). Hopcroft’s algorithm has been extensively revisited [BC05, Gri73, Knu01]. There exist alternative DFA minimization algorithms, such as Brzozowski’s algorithm [CKP02], which, for some special cases, performs better in practice than Hopcroft’s algorithm [TV05]. However, none of these sequential algorithms parallelize well (with the possible exception of Brzozowski’s, in some cases).

Parallel DFA minimization has been considered since the 1990s. All existing parallel algorithms are for shared memory machines, either using the CRCW PRAM model [TSG02], the CREW PRAM model [JR93], or the EREW PRAM model [RX96]. All of these algorithms are applicable for tightly coupled parallel machines with shared RAM and they make heavy use of random access to shared memory. In addition, [RX96] minimized a 525,000-state DFA on the CM-5 supercomputer.

When the DFA considered for minimization is very large (possibly obtained

from a large NFA by subset construction), it must be stored on disk. To our knowledge, the proposed work represents the first disk-based algorithms for determinization and minimization.

CHAPTER 3

Multiresolution Analysis in MADNESS for Hybrid CPU-GPU Clusters

This chapter presents the adaptation of an important operator in the MADNESS [FBHJ04, HFYB03, HFY⁺04, HF, HF07] scientific framework to large hybrid CPU-GPU clusters. This is a first significant step towards migrating MADNESS to hybrid CPU-GPU clusters.

MADNESS (Multiresolution ADaptive Numerical Environment for Scientific Simulation) is a simulation framework that implements a set of basic operations that are used in computational chemistry, nuclear physics and other related fields employing quantum mechanics.

Most applications running on MADNESS use 3- and 4-dimensional tensors¹. The four most common operators in MADNESS are **Apply**, **Compress**, **Reconstruct** and **Truncate**. These four operators suffice for many MADNESS applications. Only **Apply** is CPU-intensive and the time for calls to **Apply** dominates over the other three. Hence, migrating the **Apply** operator to hybrid CPU-GPU clusters is the most important step towards migrating MADNESS to hybrid CPU-GPU.

¹Tensors are higher dimensional generalizations of standard 2-dimensional matrices.

3.1 Background: GPUs in High-Performance Computing

Traditionally restricted to graphics processing, Graphical Processing Units (GPUs) have, in the past few years, become a major technology in High-Performance Computing (HPC). Due to their massive computational power and reduced energy consumption, GPUs are quickly making inroads into supercomputing, where the goal is to achieve high scalability while also keeping the energy consumption within reasonable limits. For a typical Fermi-class NVIDIA GPU, the theoretical peak performance for double precision computations is 515 GFLOPS (although usually practical applications will achieve much less than the peak). This can be compared to a single CPU core, which peaks at 12 GFLOPS. As of this writing, 16-core CPUs are the largest widely-available CPUs.

As of November 2011, 3 of the 10 fastest supercomputers in the world used GPUs [Topb]. Also, 5 of the world's 10 most power-efficient supercomputers used GPUs [Topa].

Figure 3.1 presents the NVIDIA Fermi architecture, one of the current commodity state-of-the-art GPUs. The high-end of the Fermi architecture contains 16 streaming multiprocessors (SMs) for parallel computations. Each SM contains 32 cores (also known as stream processors (SPs)) that can collaborate to perform a computation. The computational unit is the NVIDIA thread, which is run by one of the SPs. 32 threads comprise a warp, which is the scheduling unit on NVIDIA Fermi GPUs (although half a warp can be scheduled independently of the other half). Warps can be efficiently context-switched in and out of an SM. The Fermi card has a register file totaling 32,768 32-bit registers, used for rapid context switching between warps.

Within an SM, all local SPs have access to a 64-kilobyte fast memory. The fast memory logically consists of the L1 cache and the Shared Memory. The fast memory can be configured in two ways: either 16 of the 64 kilobytes act as L1 cache and the other 48 kilobytes act as Shared Memory, or 48 of the 64 kilobytes act as L1 cache and the other 16 kilobytes act as Shared Memory. Shared Memory is user-managed, whereas L1 cache is transparent and coherent. In addition, there are 63 4-byte registers on each SP. The register access latency is about 1 clock cycle (various Fermi cards' frequencies are around 1.5 GHz), while the access latency for L1 cache and Shared Memory is on the order of 2-3 clock cycles. The 768-kilobyte L2 cache is shared between the 16 SMs and has access latency on the order of 200-400 cycles. L2 cache is transparent and coherent. The global memory latency is on the order of 600-800 cycles. The high memory latency is an issue in GPU computations: the computation has to be organized so that most data is accessed from the cache levels closest to the SMs. Algorithms that exhibit complex data access are, thus, not suitable for GPUs. This contrasts with a commodity CPU, in which L2 cache latency is around 10-15 clock cycles, and RAM latency is on the order of 50-100 clock cycles.

Computation on a GPU must follow a SIMD (Single Instruction Multiple Data) model (NVIDIA's terminology is SIMT - Single Instruction Multiple Threads). All 32 threads in a warp will execute the same instruction at the same time. In the case of branching instructions, the threads following the true branch will execute code, whereas the others will execute a NO-OP (or exhibit a functionally equivalent behavior to a NO-OP). This restriction is imposed by the architecture — in order for the GPU to be power-efficient, the SM with its 32 cores has a simple design, that does not allow efficient complex data access.

Computational kernels in the CUDA language (the NVIDIA extension to C) are specified as operations organized in thread blocks. The size of a thread block is user-specified. Each thread in a thread block executes the same instruction stream. A thread block will be assigned to exactly one SM. Thread blocks are organized by the user in grids of blocks. The entire kernel is executed by a grid of thread blocks. Usually any two thread blocks are independent and there is no inter-block communication. However, CUDA allows some inter-block synchronization via CUDA atomics. Multiple thread blocks executing in parallel can be viewed as a form of task parallelism.

The applications that usually fit the GPU computing paradigm are computationally intensive (they have a large computation to global memory access ratio). Large matrix-matrix multiplication is the representative algorithm for general-purpose GPU computing. Scientific simulations are generally a good fit, since they require a high-number of complex numeric calculations. However, GPUs have been used in other areas as well, such as inexpensive highly-efficient RAID [CWSB10].

3.2 MADNESS — a Scientific Simulation Framework based on Multiresolution Analysis

MADNESS is a general purpose software framework for scientific computations based on integral and differential equations. It belongs to the classes of analysis-based solvers and basis-free methods.

MADNESS improves the complexity of solutions to scientific problems by relying on the fact that optimal mathematical representations of particles

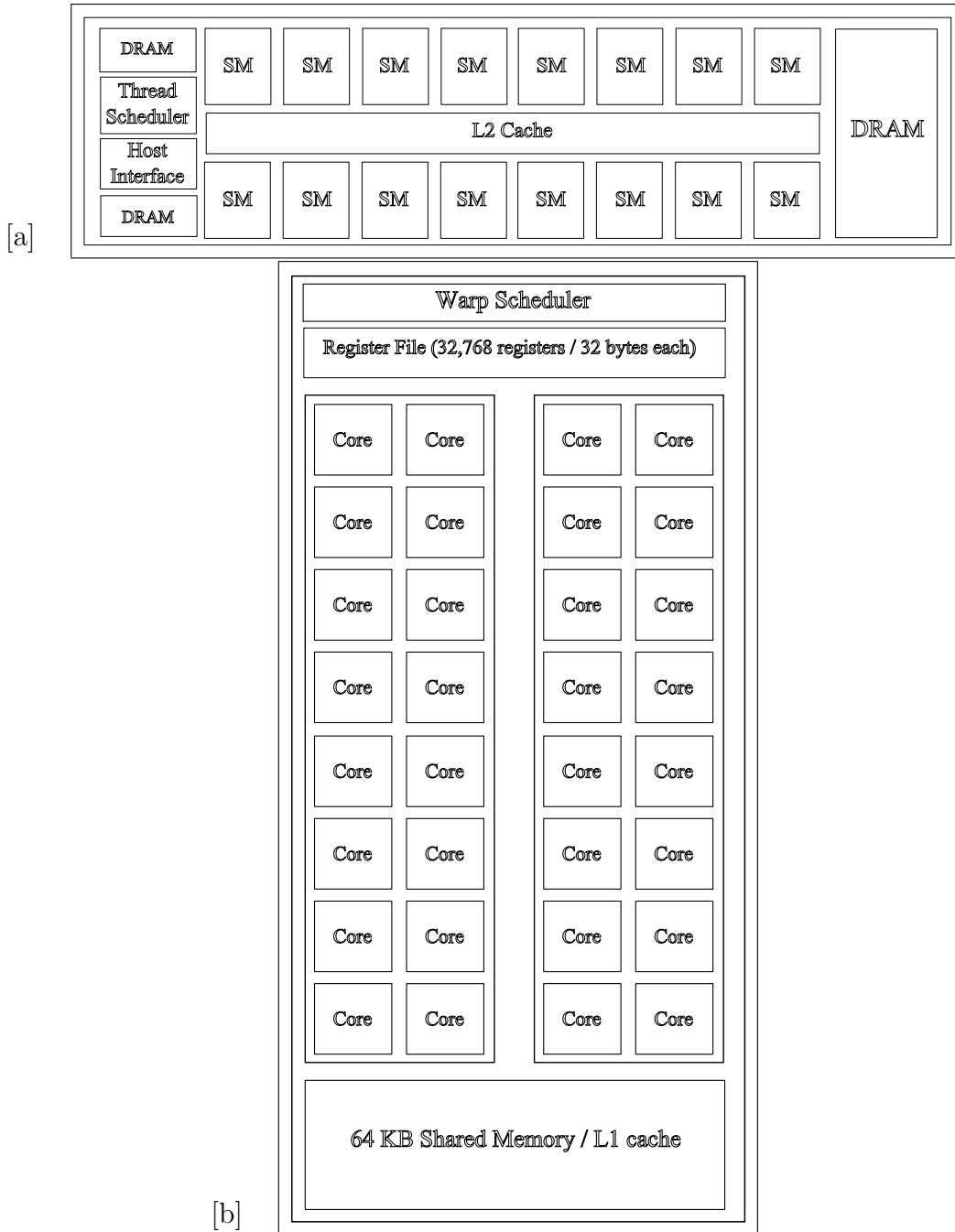


Figure 3.1: The Fermi class of GPUs from NVIDIA. [a] The Fermi GPU card has 16 SM (streaming multiprocessors), DRAM attached to the card, an L2 cache common to all SMs and a thread scheduler capable of scheduling kernels to run concurrently. [b] Each SM has 32 cores (also known as SPs — stream processors), a warp scheduler (a warp consists of 32 threads), a total of 32,768 registers of size 32 bytes, and Shared Memory and L1 cache, that together add up to 64 kilobytes.

such as molecules exhibit varying degrees of detail for different parts of the particle [HF07]. If the highest available degree of detail were used for all parts of the particle, the complexity of the system would grow non-linearly. But if the particle consists, alongside of some highly irregular parts, of many smoother parts, then higher detail is not necessary everywhere.

MADNESS employs a Multiresolution Analysis (MRA) methodology to reduce the computational complexity of the problem and achieves high precision by performing an adaptive mesh refinement over the simulation volume.

The different levels of refinement have different accuracy and the number of levels depends on the precision (accuracy) requested by the user. At a high-level, MRA can be seen as a telescoping series of grids, in which the information of the lower-ranked grids is included in the information of higher-ranked grids. (see Figure 3.2).

For some parts of a particle or system more detail might be necessary, which means that the corresponding sections of a higher-ranked grid will be used, while for some other parts of the system less detail is enough. Thus the corresponding sections of a lower-ranked grid suffice.

Figure 3.3 shows grids having finer resolution over the regions of a molecule where the probability of finding electrons (electron density) is higher and have coarser resolution over empty regions.

This helps in concentrating most of the compute power towards relevant computations, thereby reducing overall computational complexity. The level of refinement is improved by progressively generating new levels using the previous levels until the desired accuracy is achieved.

A multiresolution grid is represented in MADNESS as a highly unbalanced tree (see Figure 3.2). The nodes of the tree are distributed across the nodes of a cluster. The distribution is done using a tree-node to compute-node mapping.

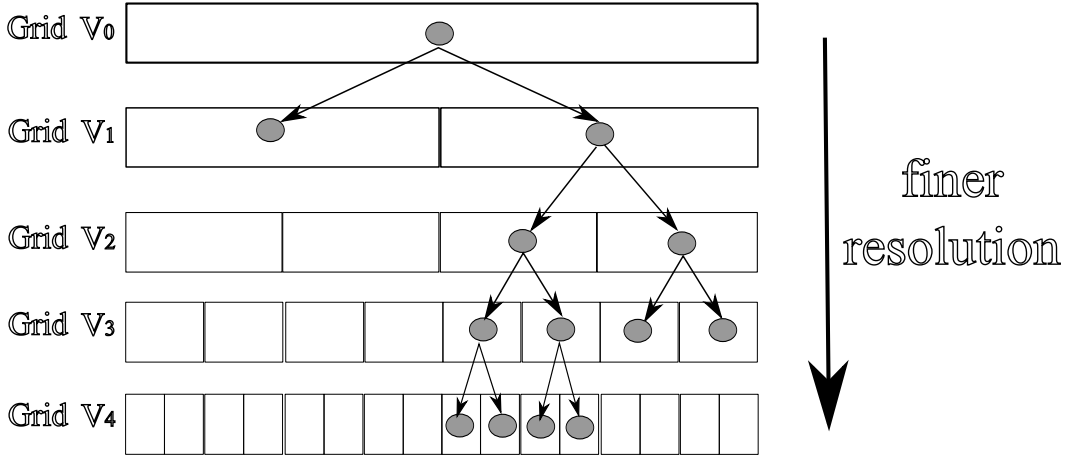


Figure 3.2: The MRA approach can be viewed as a telescoping series of grids. Grid V_0 is the coarsest. The finer the grid, the more information is captured about the wave function in the specific area. The more irregular the function in a certain area, the finer the grid that is necessary to represent it.

There are much more tree-nodes than compute-nodes and a tree-node resides on a single compute-node. Distributed trees are implemented in MADNESS with distributed hash tables.

MADNESS operators (such as **Apply**, **Compress**, **Reconstruct**, or **Truncate**) take as input a distributed tree, which they explore and modify.

3.2.1 The MADNESS Framework Implementation

The MADNESS framework implementation uses ideas from other HPC frameworks such as Cilk [BJK⁺96] and Charm++ [KRSG95].

Above the communication layer built around MPI and Global Arrays [NPT⁺06] there are three layers, as presented in Figure 3.4.

Right above the communication layer is the MADNESS *Parallel Runtime*, which exposes to the developer distributed data structures such as the **WorldContainer** (a distributed hash table) and the *Distributed Stack* mechanism, through constructs such as **Tasks** and **Futures** (placeholders for values yet to be returned by a remote task). Futures were developed originally in

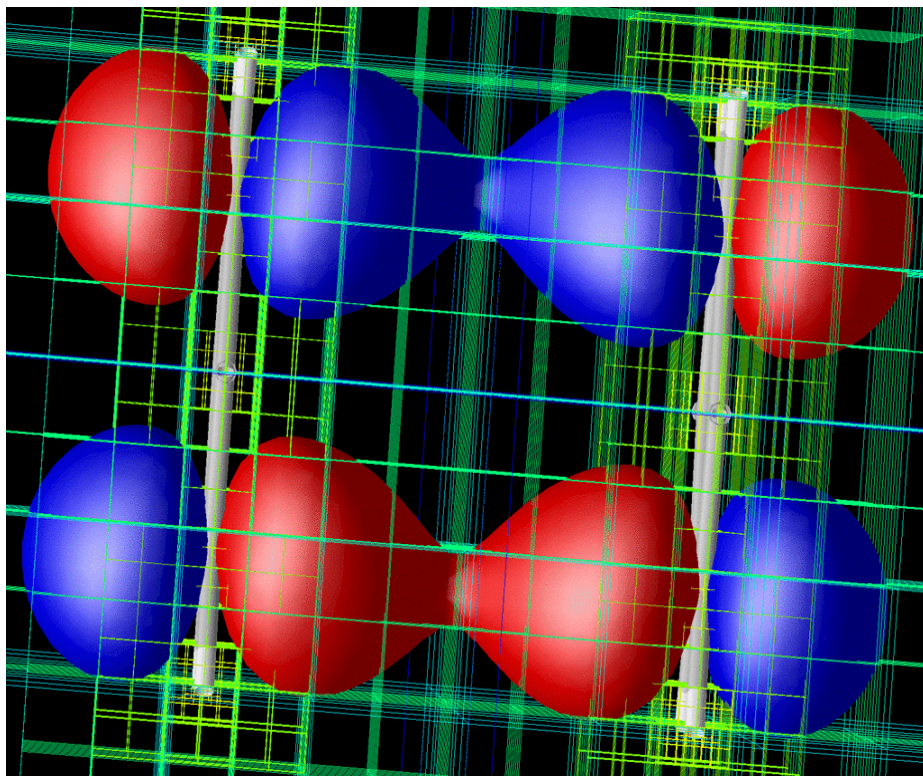


Figure 3.3: Multiresolution Analysis over benzene dimer (See Figure 6 of article [HF07]).

BBN Butterfly LISP in the 1980s [SKL88]. In MADNESS, a specific task may be executed locally or remotely. If it is executed locally, the `task` function call in MADNESS will block waiting for the function to return. If it is executed remotely, the `task` function call will return immediately with a `Future` value, which is not immediately evaluated. `Futures` can be passed as parameters to other `task` invocations, in which case the specific `task` will not start executing until all `Future` parameters have been evaluated. This is similar to the Cilk [BJK⁺96] programming model.

On top of the *Parallel Runtime* are the *Mathematical and Numerical* al-

MADNESS architecture

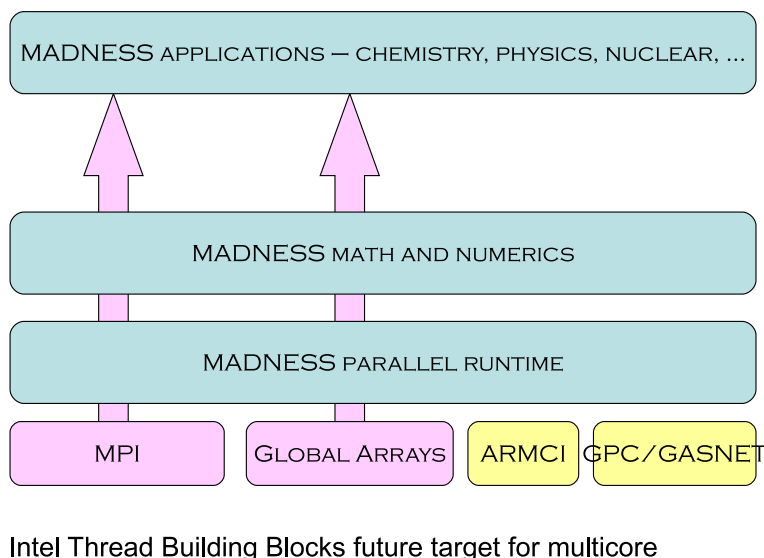


Figure 3.4: The Layered MADNESS Architecture (see [Har, page 27])

gorithms based on Multiresolution Analysis. The main data structure used in the *Mathematical and Numerical* layer is the n -ary distributed tree (implemented using the `WorldContainer`), which fits the hierarchical representation obtained from applying Multiresolution Analysis to wave functions. The tree nodes contain function coefficients. Most of the layer’s algorithms involve some manner of traversing the tree.

3.2.2 Adapting Scientific Simulation Frameworks for CPU-GPU clusters

Among packages supporting molecular dynamics, MADNESS is unique in using adaptive multiresolution analysis to solve molecular dynamics problems larger than ever before. The adaptively refined tree (see Figure 3.2) leads to

computations at multiple scales. This involves irregular computations over small matrices.

Because other packages do not use adaptive multiresolution methods, they have a tendency to do additional work for the same accuracy. Because the additional work is more regular than that for MADNESS, it tends to parallelize well on GPUs. *Parallelization of MADNESS tends to suffer from many small tasks.* Hence, other packages will achieve higher speedups on GPUs because they begin with a lower performance on CPUs for a given accuracy. Hence, for a given accuracy, MADNESS already achieves good performance on CPU-based clusters. The challenge for MADNESS is to achieve enough speedup on GPUs so as to retain its performance advantage for a given accuracy.

The multiple levels and scales for MADNESS are one of its strengths. The only competitive approach is BigDFT [Big]. However, BigDFT only has two levels of resolution (coarse and fine). This makes it significantly less flexible than MADNESS, which has multiple levels of resolution that are computed dynamically.

Some of the competing scientific simulation frameworks that have been ported to CPU-GPU architectures are [NVIa]: BigDFT (a pseudopotential density functional theory (DFT) framework that expresses Kohn-Sham wavefunctions in Daubechies wavelet bases), TeraChem [Ter] (Gaussian orbitals that implement ab initio molecular dynamics and DFT methods), GAMESS [GAM] (also an ab initio computational chemistry framework), AMBER [AMB] (a framework for classic molecular dynamics), NAMD [NAM] (molecular dynamics), GROMACS [GRO] (molecular dynamics for protein and bio-molecule simulation), LAMMPS [LAM] (classic molecular dynamics) and QMCPACK [QMC] (continuum quantum Monte Carlo methods).

All these simulation frameworks use large matrices, as described before, so

they are well-suited for GPUs. MADNESS computations use smaller, rectangular matrices and the ratio of computation to data access is lower than for the other scientific frameworks. This makes MADNESS highly suitable for many-core architectures, but leads to challenges when porting to GPUs, as described next. Nevertheless, MADNESS needs to be ported to CPU-GPU architectures in order to take advantage of the resources offered by new hybrid CPU-GPU clusters, such as Titan.

3.3 Migrating MADNESS Operators to Hybrid CPU-GPU Architectures

A naive CPU-GPU port of a MADNESS operator would replace each call to a matrix multiplication on the CPU with an equivalent GPU routine. However, this would result in low GPU occupancy and high CPU-GPU transfer latency. Also, CPU-GPU transfer would optimally use page-locked memory, but the overhead of page-locking for the transfer of a single matrix would be excessive. The solution is:

- to separate compute-intensive code from data-intensive code;
- to aggregate the computation; and
- to aggregate the data inputs.

Due to the irregular nature of the computations, even hand optimization of the MADNESS operators for the GPU does not utilize the full power of the GPU. Hence, it was important:

- to overlap CPU computation with GPU computation.

Extending MADNESS to CPU-GPU Clusters requires modifications to the underlying control flow of MADNESS operators. Keeping the current control flow for a CPU-GPU execution would be inefficient, since launching a GPU kernel on-demand is slow, because of: high CPU-GPU transfer latency, low CPU-GPU transfer bandwidth without page-locking, slow on-demand page-locking, and low GPU occupancy. Our modified **Apply** operator is fully compatible with the rest of the framework, that was not modified.

3.3.1 Modifications to the Control Flow of a MADNESS Operator

As opposed to other scientific frameworks (see 3.2.2), MADNESS employs many small tasks instead of a few large tasks. One MADNESS task applies an operator to a single node in the multiresolution tree and it can pass the result to other tree-nodes, modify the current tree-node, remove it and even create new tree-nodes. A node of an n -ary tree consists of an n -dimensional tensor along with extra information.

We made modifications to the control flow of **Apply** to enable the aggregation of data inputs for GPU tasks. Data inputs are aggregated into a few large pre-allocated buffers, which are then transferred to the GPU in a single step. The CPU-GPU latency penalty is thus paid only once for the entire batch as opposed to once for each task input.

Moreover, the pre-allocated transfer buffers are page-locked at the beginning of the computation. Page-locking ensures that the buffer stays resident in RAM, and it also leads to at least double the transfer speed. Page-locking can efficiently be done only on a few large buffers, since it is slow (0.5 milliseconds). Page-unlocking (the reverse of page-locking) is even slower (2 milliseconds), in the context in which the execution of a single typical MADNESS

3-dimensional CUDA kernel is on the order of 1 millisecond.

These modifications to the control flow of **Apply** are critical for solving the high latency, slow transfer and GPU occupancy issues.

Besides input data, the compute-intensive MADNESS tasks are also aggregated. A batch of tasks can be more efficiently scheduled on the streaming processors (SMs) of the GPU than individual tasks. Having a batch of tasks readily available allows launching multiple tasks of the same kind on the GPU concurrently in the case of small tasks, that individually occupy only a fraction of the GPU. Moreover, informed decisions can be made about how to divide work between the CPU and the GPU when a large computation batch is available.

Both the CPU and CPU-GPU versions of the **Apply** operator are presented in Section 3.4.

MADNESS Library Extensions for Efficient CPU-GPU Computing

It is not desirable to perform batching of all MADNESS tasks. The goal is to batch only those MADNESS tasks that are compute-intensive. The MADNESS algorithms developer has to identify these tasks and expose them to the MADNESS Library extensions. Specifically, the developer can split a task of interest into three sub-tasks: *preprocess*, *compute* and *postprocess*. The MADNESS Library extensions will ensure that the *preprocess* sub-task will be executed by a CPU thread. It will also ensure that the output data of *preprocess* is batched together with other output data of the same kind, to serve as input data for *compute* tasks.

The execution of the multiple *compute* tasks waiting for input data is delayed until a timer expires. At this point there are multiple batches of *compute* waiting to be executed (one batch per kind of *compute* task). The “kind” of a task is given by a combination of the memory address of the

compute function and the result of a user-defined hash function applied to the input data. Batches of *compute* tasks will be executed one by one at this point.

A dispatcher CPU thread will split each batch of *compute* tasks between the CPU threads and the GPU. A *compute* task must have both a CPU- and a GPU-version. By knowing the relative performance of the GPU code compared to the CPU code for a certain operator, a MADNESS developer can decide what is the ratio of CPU to GPU work. Consider that a CPU-only run takes time m and a GPU-only run takes time n . The minimal computation time can be achieved by an optimal CPU-GPU computation overlap. The minimal time is calculated by minimizing $\max(mk, n(1 - k))$, with $k \in [0, 1]$. This means that a k -fraction of tasks are sent to the CPU and the rest to the GPU. The optimal CPU-GPU work overlap is achieved when $mk = n(1 - k)$, so $k = n/(m + n)$. The minimal runtime is thus $\frac{m \times n}{m + n}$. This method of work distribution between the CPU and the GPU has its limitations: it only takes into account computation; it does not take into account data access. The method works best when the computation clearly dominates data access. However, if data access performed by the dispatcher thread or the worker threads requires time comparable to the computation time, then the computation overlap between the CPU and GPU will be sub-optimal. In these cases, the distribution of work between the CPU and the GPU can be tuned to compensate for the unaccounted data access time.

Newly generated *compute* tasks continue to be batched while the CPU and the GPU do work. The dispatcher also drives the execution of the GPU computation batches.

The control flow of the CPU-GPU **Apply** operator is presented in Figure 3.5. (This figure again presents the overview previously seen as Figure 1.1

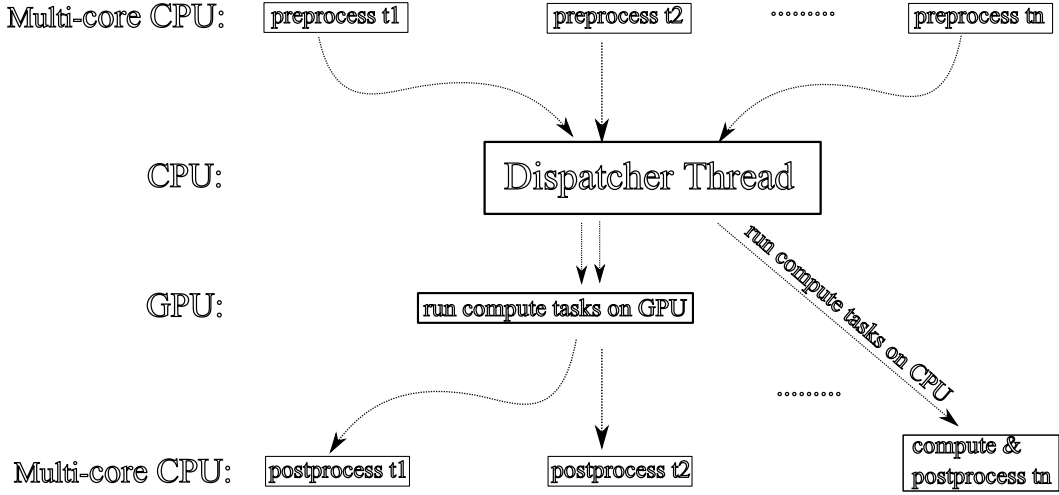


Figure 3.5: The control flow of a hybrid CPU-GPU MADNESS operator.

in Chapter 1).

Figure 3.6 describes the data flow behavior of the Hybrid CPU-GPU system, including the dispatcher thread. The implementation is optimized by using two task queues implemented as hash tables: the active and inactive hash table. The two hash tables are the implementation support of the double-buffering feature.

Next we present the CPU-GPU version of the MADNESS **Apply** operator that uses our MADNESS Library extensions.

3.4 The Apply Operator

The most computationally intensive operator in MADNESS is **Apply**. It consists of computing a Gaussian operator on each tensor node in a tensor tree and accumulating the local results of the Gaussian to compute an approximation of a version of Green’s function.

The computational part of the **Apply** operator can be expressed as:

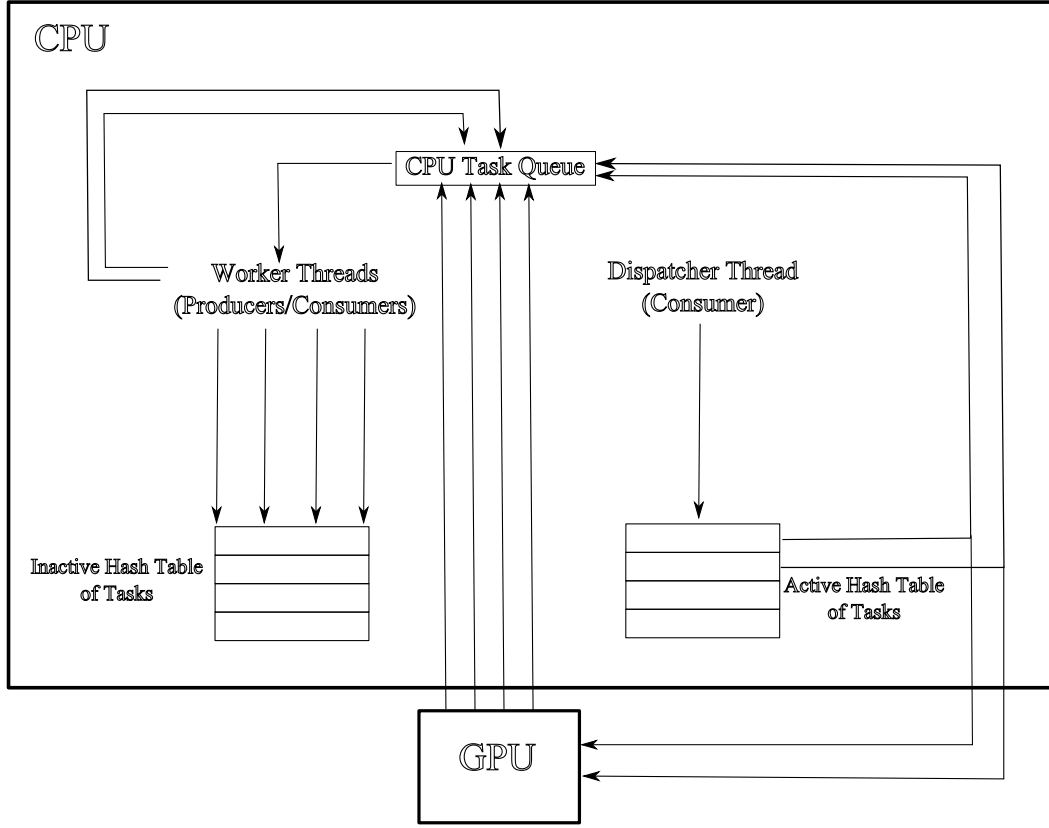


Figure 3.6: Data flow patterns for the Hybrid CPU-GPU system. Double-buffering is supported by the use of two task queues implemented as hash tables: the active and inactive hash tables. Once all data from the active hash table has been processed by the dispatcher thread, the active and inactive hash tables are swapped, so that processing can continue from an already-filled hash table.

Formula 3.4.1. *Integral Kernel for the Apply Operator*

$$r_{i_1 i_2 \dots i_d} = \sum_{\mu=1}^M \sum_{j_1=0}^{2k-1} \sum_{j_2=0}^{2k-1} \dots \sum_{j_d=0}^{2k-1} s_{j_1 j_2 \dots j_d} h_{j_1 i_1}^{(\mu,1)} \times h_{j_2 i_2}^{(\mu,2)} \times \dots \times h_{j_d i_d}^{(\mu,d)}$$

Here r and s are output/input d -dimensional tensors, respectively. s is a tensor from the input MADNESS multiresolution tree, while r is a tensor that will be used to update the input tree. The h operators are 2-dimensional tensors that are either computed as needed, or obtained from a software cache. Typical values of M and k are 100 and 10–20, respectively. Typical values of d are 3 and 4, but some computations may use $d = 6$.

A tensor product in notation $a_{i_1 i_2 \dots i_k \dots i_d} \times b_{i_k j_1}$ means multiplying along dimension k , in the same way as a matrix multiplication $A_{i_1 i_2} \times B_{i_2 j_1}$ performs element-by-element multiplication along dimension 2.

If we isolate the first step in the transformation and view the indices $j_2 \dots j_d$ as one compound index, then the first step in the tensor product can be viewed as $R_1 = r_{j_2 \dots j_d i_1} = \sum_{j_1=0}^{2^k-1} s_{j_1 j_2 \dots j_d} \times h_{j_1 i_1}$. The second step can be viewed as $R_2 = r_{j_3 \dots j_d i_1 i_2} = \sum_{j_2=0}^{2^k-1} r_{j_2 j_3 \dots j_d i_1} \times h_{j_2 i_2}$. The final result is obtained from this iterative process. The d-dimensional tensors are represented in memory as 2-D matrices of size $n^{d-1} \times n$. Since the first step multiplies along dimension-index j_1 and the second step multiplies along dimension-index j_2 , in order to preserve the memory layout of d-dimensional tensors, the tensor dimension-indices have to be cyclically permuted between iterative steps. For example, after computing R_1 , the indices are permuted as follows: $j_1 j_2 \dots j_{d-1} j_d \rightarrow j_2 j_3 \dots j_d j_1$. The same efficient cache-aware access patterns can thus be used in each iterative step.

Figure 3.7 presents the product of a 3-dimensional $2 \times 2 \times 2$ tensor with a 2-dimensional 2×2 tensor across all 3 dimensions.

Figure 3.7 [a] presents the matrix multiplication of two 3×3 matrices (across dimension i_2). The figure shows how indices are selected for multiplication to obtain $C[i_1, i_3] = \sum_{i_2=0}^2 A[i_1, i_2] \dot{B}[i_2, i_3]$ and highlights how $C[1, 0]$ is obtained.

Figure 3.7 [b] presents the tensor multiplication for a 3-dimensional $3 \times 3 \times 3$ tensor and a 2-dimensional 3×3 tensor (a matrix) across dimension i_1 . The result tensor C is computed as follows: $C[j_1, i_2, i_3] = \sum_{i_1=0}^2 A[i_1, i_2, i_3] \dot{B}[i_1, j_1]$. Note that the 3-D memory tensor layout in the figure is row-major (i_1 major followed by i_2 minor and i_3 minor).

Figure 3.7 [c] presents the tensor multiplication of the result from Fig-

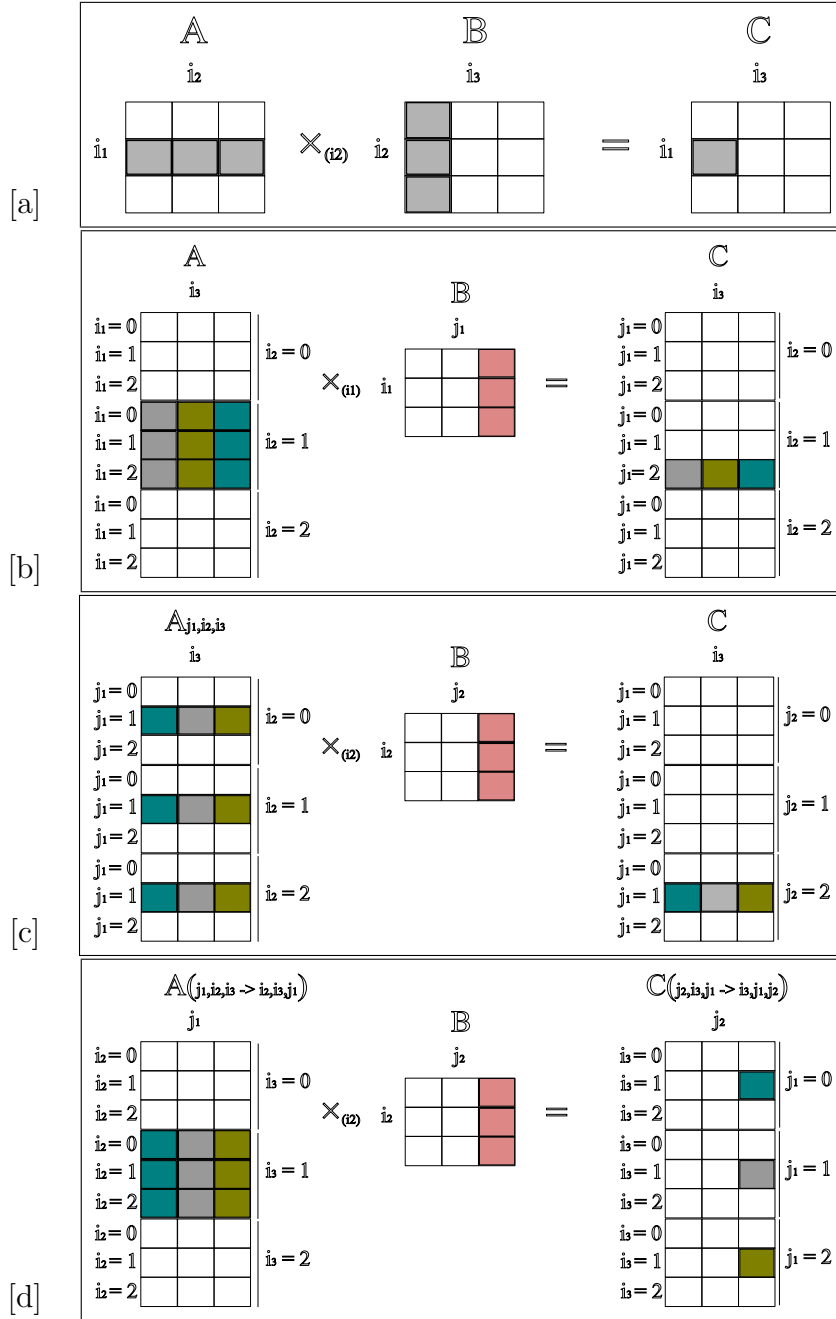


Figure 3.7: Tensor Product: Generalization of Matrix Multiplication.

[a] Matrix multiplication of two 3×3 matrices (across dimension i_2).

[b] Tensor multiplication for a 3-dimensional $3 \times 3 \times 3$ tensor and a 2-dimensional 3×3 tensor (a matrix) across dimension i_1 . The tensors are represented in row-major format.

[c] The result of the multiplication from [b] is multiplied by a 2-D 3×3 tensor across dimension i_2 . In this case, the input tensor is in row-major order j_1, i_2, i_3 .

[d] The same multiplication from [c], but this time both the input and output tensors are cyclically permuted so that they are in row-major format, with the first dimension the next one to perform multiplication over.

ure 3.7 [b] by a 2-D tensor across dimension i_2 . Note that the access patterns are now different from the ones in Figure 3.7 [c]. Having to change access patterns for each dimension change allows less cache-aware control, which leads to a significant increase in the cache miss ratio.

For this reason, the memory layout is changed after each tensor multiplication by applying a cyclic permutation of the indices of the result tensor, which leads to optimal cache behavior. Thus, the same highly optimized tensor multiplication function can be applied in each step, since this is a layout-preserving transformation (as seen in Figure 3.7 [d]). The cyclic permutation is not applied as a separate operation, but rather the result of a multiplication is written directly in the index-permuted order.

Algorithm 8 describes the MADNESS **Apply** algorithm for CPUs at a high level. Here we are interested in identifying the data-intensive and compute-intensive parts of the code, in order to reorganize them for efficient CPU-GPU execution.

To take advantage of the MADNESS Library extensions for CPU-GPU, the **Apply** operator presented in Algorithm 8 has to be split into three parts that are implemented by three tasks: *preprocess*, *compute* and *postprocess*.

The *preprocess* task obtains the addresses of all the 2-dimensional h tensor operators. In some cases, *preprocess* also calculates a low-rank approximation of the 2-dimensional tensors by singular value decomposition. The *compute* task uses the preprocessed inputs, performs the necessary $s \times h$ tensor products and adds the results into tensor r . The *postprocess* task accumulates the result tensor r into a neighbor tensor in the tree.

The bulk of the data is comprised of the two-dimensional tensor operators h . Many of the 2-D tensor operators will be reused multiple times by transformations applied to various different input d -dimensional tensors. Therefore,

Input: The current *coefficients* tree.
Output: The *coefficients* tree after computing a Green’s convolution.

- 1: **for** each *node* $s_{j_1 j_2 \dots j_d}$ in the *coefficients* tree **do**
- 2: Obtain displacements.
- 3: **for** each displacement **do**
- 4: $neighbor = \text{Compute neighbor of } s_{j_1 j_2 \dots j_d} \text{ based on displacement.}$
- 5: Tensor $r_{i_1 i_2 \dots i_d} = \text{integral_operator}(s_{j_1 j_2 \dots j_d})$.
- 6: Accumulate tensor $r_{i_1 i_2 \dots i_d}$ into *neighbor*.
- 7: **end for**
- 8: **end for**
- 9:
- 10: **function** *integral_compute*(*Key key*, *FunctionNode node*, *Key disp*, *Key dest*) **returns** *Tensor*:
- 11: Initialize result tensor $r_{i_1 i_2 \dots i_d}$.
- 12: **for** each $\mu = 0$ to convolution *rank* **do**
- 13: Obtain the h 2-D tensors $(h_{j_1 i_1}^{(\mu,1)}, h_{j_2 i_2}^{(\mu,2)}, \dots, h_{j_d i_d}^{(\mu,d)})$.
- 14: Apply *Formula 3.4.1* to $s_{j_1 j_2 \dots j_d}$ and the h tensors and add the result to $r_{i_1 i_2 \dots i_d}$.
- 15: **end for**
- 16: **return** $r_{i_1 i_2 \dots i_d}$.
- 17: **end function**

Algorithm 8: The “Apply” Algorithm.

in order to avoid redundant data transfers to the GPU, a write-once software cache containing the already transferred 2-D tensors has been implemented. This write-once cache has been modeled after a CPU software cache present in MADNESS for similar purposes.

After all the necessary data has been transferred to or located on the GPU, independent CUDA custom computational kernels (that were designed and implemented by Raghu Varier and presented in [SVCH12]) are launched on the NVIDIA device in separate CUDA streams. For 3-dimensional tensors, the use of CUDA streams helps occupy the GPU fully. 4-dimensional tensor products are much larger, and they occupy the GPU fully without the use of CUDA streams.

The custom kernels for 3-D problems take advantage of shared memory, L1 and L2 cache, and register locality on NVIDIA GPU devices, as well as other

Input: The current *coefficients* tree.

Output: The *coefficients* tree after computing a Green's convolution.

```

1: for each node  $s_{j_1 j_2 \dots j_d}$  in the coefficients tree do
2:   Obtain displacements.
3:   for each displacement disp do
4:     Asynchronously call integral_preprocess( $s_{j_1 j_2 \dots j_d}$ , disp).
5:   end for
6: end for
7:
8: function integral_preprocess(source_tensor, displacement):
9:   neighbor = Compute neighbor of source_tensor based on displacement.
10: for each  $\mu = 0$  to convolution rank do
11:   Obtain the h 2-D tensors  $(h_{j_1 i_1}^{(\mu,1)}, h_{j_2 i_2}^{(\mu,2)}, \dots, h_{j_d i_d}^{(\mu,d)})$ .
12: end for
13: Add the tuple (source_tensor, neighbor, h tensors) to the source data
    batch that corresponds to the “kind” of the data.
14: end function
15:
16: function integral_all_compute(batches of tuples (source_tensor, neighbor,
    h tensors)): // One batch of tuples (source_tensor, neighbor, h tensor)
    per data “kind”.
17: for each batch of tuples do // Initiate the CPU sub-batch processing:
18:   Split source data batch into a sub-batch for the GPU and a sub-batch
    for the CPU.
19:   for each tuple (source_tensor, neighbor, h tensors) in the CPU
    sub-batch do
20:     Asynchronously call integral_CPU_compute(source_tensor, neighbor,
    h tensors).
21:   end for
22:   Synchronously call integral_GPU_compute(batch of tuples
    (source_tensor, neighbor, h tensors)).
23: end for
24: end function

```

Algorithm 9: The CPU-GPU Version of the “Apply” Algorithm.

low-level facilities, such as inter-block synchronization using CUDA atomics and coalesced global memory access.

The hybrid CPU-GPU version of **Apply** is presented in Algorithms 9 and 10.

```

25: function integral_GPU_compute(batch of tuples (source_tensor,
    neighbor, h tensors)):
26: Initialize batch of result tensors  $r_{i_1 i_2 \dots i_d}$ .
27: for each tuple in the batch of tuples (source_tensor, neighbor, h tensors)
    do
28:   Asynchronously launch an Apply CUDA kernel on the GPU with tuple
    as input.
29: end for // Block waiting for all the CUDA kernels to finish.
30: When all CUDA kernels have finished, update the batch of result tensors
     $r_{i_1 i_2 \dots i_d}$ .
31: for each result tensor  $r_{i_1 i_2 \dots i_d}$  do
32:   Asynchronously call integral_postprocess( $r_{i_1 i_2 \dots i_d}$ , neighbor).
33: end for
34: end function
35:
36: function integral_CPU_compute(source_tensor, neighbor, h tensors):
37: Initialize result tensor  $r_{i_1 i_2 \dots i_d}$ .
38: for each  $\mu = 0$  to convolution rank do
39:   Obtain the h 2-D tensors  $(h_{j_1 i_1}^{(\mu,1)}, h_{j_2 i_2}^{(\mu,2)}, \dots, h_{j_d i_d}^{(\mu,d)})$ .
40:   Apply Formula 3.4.1 to  $s_{j_1 j_2 \dots j_d}$  and the h tensors and add the result
    to  $r_{i_1 i_2 \dots i_d}$ .
41: end for
42: Asynchronously call integral_postprocess( $r_{i_1 i_2 \dots i_d}$ , neighbor).
43: end function
44:
45: function integral_postprocess(result, neighbor):
46: Accumulate tensor  $r_{i_1 i_2 \dots i_d}$  into neighbor.
47: end function

```

Algorithm 10: The compute and postsprocess functions of the “Apply” Algorithm.

3.4.1 CPU Optimization: Rank Reduction

MADNESS introduces a separated representation that speeds up higher-dimensional computations, but also expands the rank [HF07]. For this reason, some of the *h* tensors in Formula 3.4.1 can be approximated by matrices of lower rank. MADNESS implements a CPU optimization in which, for each $s \times h$ multiplication, certain rows and columns of *s* and *h* can be omitted (see Figure 3.8). This optimization is called *rank reduction*. This MADNESS optimization can reduce the amount of computation *on the CPU only* by up to 2.5-times in

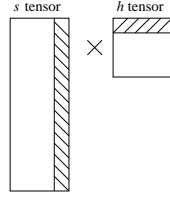


Figure 3.8: Rank reduction for an s and h tensor. h is a 2-dimensional tensor. s is a higher-dimensional tensor projected onto 2 dimensions. The hashed areas mark the rows and columns that are reduced. Note that reducing the rows and columns does not change the dimension of the result matrix.

typical cases.

Rank reduction was also implemented for the custom CUDA kernel, but did not have a noticeable effect on performance. The reason is that, unlike for the CPU, GPU resources are allocated at CUDA kernel launch time. The custom CUDA kernel typically performs hundreds of multiplications using the same input s tensor, but hundreds of input h tensors, as needed by Formula 3.4.1. (Recall that typical values of M and k are 100 and 10–20, respectively, in Formula 3.4.1.) Each multiplication uses two or three SMs, due to the lack of enough storage for the computation in the shared-memory and registers of a single SM. The custom kernel must reserve in advance the two or three SMs. For some of the multiplications, rank reduction allows the multiplication to be computed by a single SM. However, the GPU gains nothing from this, since the two or three SMs were already reserved by the kernel.

The dynamic parallelism featured in the future CUDA 5 release could help alleviate some of the rank reduction issues on GPUs. This future facility allowing the launch of sub-kernels from running kernels seems the most helpful for rank reduction. However, this will only be available for the Kepler GPU.

3.5 Implementation Issues for CPU-GPU

Apply

The implementation of the hybrid CPU-GPU **Apply** algorithm raised a number of issues that had to be resolved in order to obtain an efficient hybrid operator.

One issue was the bandwidth of CPU-GPU data transfers. Transferring a data buffer without page-locking it and registering it with the CUDA Library is slow (up to three times slower than for page-locked and CUDA-registered buffers).

The first solution attempted was on-demand page-locking and CUDA-registering CPU-memory that holds tensors and other data that needs to be transferred to the GPU. This was followed by the actual data transfer, and page-unlocking and CUDA-unregistering. This solution offset any benefits resulted from faster transfer, since on-demand page locking requires approximately 0.5 milliseconds and page-unlocking requires approximately 2 milliseconds. These times are often comparable to the execution times of the CUDA kernels for the transferred input batch.

The second solution attempted was to use a custom memory allocator that allocates a large chunk of memory upon execution startup. All memory allocation requests would then be served from the pre-allocated large chunk. The allocator page-locks the entire large chunk immediately after it is allocated. Preliminary tests resulted in segmentation faults for various types of memory allocation using this approach. Moreover, page-locking needs to be followed by CUDA-registering of the page-locked memory. Since the CPU memory allocator is not aware of the CUDA Library, registering the pre-allocated chunk is difficult to achieve. Also, implementing a robust custom allocator that works for all types of allocations is a complex task.

The chosen solution was to use large memory buffers that the application pre-allocates, pre-page locks and pre-registers with the CUDA Library. This introduces an additional processing step: the copying of CPU data from their existing locations spread throughout virtual memory to these few continuous pre-allocated buffers. This solution performs well. With the chosen solution, data transfer is fast and the page-locking times are not a problem, since page-locking is only performed once per buffer and the buffer is reused many times.

However, the extra copying step to the large buffers introduces a visible, but not dominant performance penalty. The memory copy is performed with the *memcpy* operation, which achieves lower than expected memory bandwidth. The actual memory bandwidth was around 4 GB per second on a system whose theoretical maximum is close to 16 GB per second. The problem is that *memcpy* does not prefetch large chunks of memory into cache. Therefore, it pays the price of a cache miss upon loading each next 64 bytes from the source buffer into cache. Since the price of a cache miss is around 20 nanoseconds, this explains the 4 GB per second actual observed bandwidth. The solution would be to implement a custom *memcpy* that prefetches a large chunk of the source buffer and a large chunk of the destination buffer in L2 cache in only a few assembly instructions, and then performs the partial memory copy entirely in L2 cache. The full memory copy would be significantly faster than the traditional *memcpy*, since it would suffer from fewer cache misses. However, it is unclear whether pre-fetching such a large memory chunk can be done in assembly language. There exist custom *memcpy* implementations, but it looks like none of them implement such a mechanism. The custom *memcpy* implementations report around 30% improvement over the traditional implementation, but this improvement is still short of the 2 or 3 times improvement theoretically possible. Two alternatives to the tradi-

tional *memcpy* implementations were tested (Agner Fog’s *memcpy* [Fog] and Daniel Vik’s *memcpy* [Vik]). Neither of the alternative *memcpy* implementations showed any improvement over the traditional implementation.

Implementing an efficient *memcpy* that achieves significant speedup over the traditional implementation is a large project in itself and is outside the scope of the proposed solution. The extra copying step is not a dominating factor in the computation, so no other attempts were made to make it more efficient.

A third issue encountered during the implementation of the **Apply** algorithm was the implementation of the custom CUDA kernels for **Apply**. The complex custom CUDA kernels were developed by Raghu Varier and are not the subject of this dissertation. Some of the issues that surfaced during the CUDA kernels development are described in [SVCH12].

3.6 Experimental Results

Experiments were run on nodes of the Titan supercomputer at Oak Ridge National Laboratory. One compute node consists of a 16-core AMD Opteron 6200 Interlagos at 2 GHz frequency, 16 or 32 GB of DDR3 RAM and an NVIDIA Tesla M2090 (of the Fermi class) at 1.6 GHz with 6 GB of GDDR5 connected via a PCIe 2.0 \times 16 slot.

One of the applications that relies on **Apply** is the computation of a *Coulomb* operator. The CPU-GPU implementation of **Apply** was compared with a highly-optimized CPU implementation (in which tensor multiplications were programmed in assembly language to achieve good cache behavior). The *Coulomb* application has among the inputs the dimension of the input tensors (d), the size of the tensor per dimension (k) and the desired precision of the

result.

Tables 3.1 and 3.2 present a comparison between the running times of a CPU computation, a GPU computation, and a hybrid CPU-GPU computation of the *Coulomb* operator. The GPU and hybrid CPU-GPU versions both used our MADNESS Library extensions that automatically schedule tasks on the GPU and CPU. These results are observed for a computation batch of 60 independent tasks. Also note that our approach of dedicating two or three SMs on the GPU to one lengthy, compute-intensive task that multiplies many small tensors, is more effective than using cuBLAS (see Tables 3.3 and 3.4) for 3-dimensional problems. cuBLAS distributes a tensor product across all 16 SMs of the GPU, which is efficient for operators on larger tensor sizes, such as the 4-dimensional TDSE operator presented in Table 3.6.

Application Coulomb with input parameters $d = 3$, $k = 10$ and precision 10^{-8} (no rank reduction)					
CPU-only compute		GPU-only compute		CPU and GPU compute	
CPU threads	CPU time (sec)	GPU streams	GPU time (sec)	CPU + GPU time (sec) using 10 CPU threads & 5 CUDA streams	
1	132.5	1	71.3	Optimal CPU-GPU Actual Overlap 14.4 12.1	
2	66.5	2	41.5		
4	45.7	3	31.5		
6	35.6	4	26.4		
8	28.5	5	24.3		
10	24.3	6	24.7		
12	22.8				
14	18.5				
16	19.9				

Table 3.1: CPU scale-up vs. GPU scale-up for *Coulomb* with input parameters $d = 3$, $k = 10$ and precision 10^{-8} . The GPU and hybrid CPU-GPU versions used our MADNESS Library extensions for work scheduling. For the GPU version 12 CPU threads for data access were used.

Table 3.2 shows experimental results for running *Coulomb* with $d = 3$, $k = 20$ and precision= 10^{-10} . Note that, compared to the $k = 10$ case, here the GPU performs even better compared to the CPU. The larger the tensor size, the better the GPU fares compared to the CPU. The main reason is that

the CPU incurs more cache misses for larger tensor sizes. Also, in this case cuBLAS routines were used — with tensors 8 times larger than for $k = 10$, we enter the regime in which cuBLAS performs well.

Application Coulomb (no rank reduction) with $d = 3$, $k = 20$ and precision 10^{-10}	
CPU 16 threads time	173.3 sec
GPU time	136.6 sec
CPU + GPU time (actual)	99.0 sec
CPU + GPU time (optimal CPU-GPU overlap)	76.2 sec

Table 3.2: CPU 16 threads vs. GPU for *Coulomb* with input parameters $d = 3$, $k = 20$ and precision 10^{-10} . The GPU-version used 15 CPU threads for data access. The hybrid version used 15 CPU threads.

Tables 3.3 and 3.4 compare the performance of a version of the 3-dimensional *Coulomb* application that uses our custom CUDA kernels with a version that uses cuBLAS 4.1. In both versions the computationally intensive part was processed only by the GPU. For this test only we used a MADNESS process map that distributes work evenly among all compute nodes.

	Application Coulomb (no rank reduction) with $d = 3$, $k = 10$ and precision 10^{-10}		
Compute nodes	Time (seconds)		Speedup ratio
	Custom kernel	cuBLAS version 4.1	
2	88	247	2.80
4	56	126	2.25
8	31	71	2.29
16	19	42	2.21

Table 3.3: Timings for 3-dimensional *Coulomb* using our custom CUDA kernels and using cuBLAS 4.1. Work was distributed evenly to all compute nodes. Below 2 nodes the data per node is too large for the GPU RAM. Above 16 nodes the amount of work in a batch of tasks is insufficient for good parallelism.

Table 3.5 presents the running times of a slightly larger *Coulomb* application. In this case 3-D tensors with $k = 30$ were used and the desired result precision was set to 10^{-12} . This application stops scaling above six compute

Compute nodes	Application Coulomb (no rank reduction) with $d = 3$, $k = 10$ and precision 10^{-11}		
	Time (seconds)		Speedup ratio
	Custom kernel	cuBLAS version 4.1	
16	27.6	43.2	1.56
32	15	24.2	1.61
64	10.2	15.6	1.52
100	7.6	11	1.44

Table 3.4: Timings for 3-dimensional *Coulomb* using our custom CUDA kernels and using cuBLAS 4.1. Work was distributed evenly to all compute nodes. Below 16 nodes the data per node is too large for the GPU RAM. Above 100 nodes the amount of work in a batch of tasks is insufficient for good parallelism.

nodes, because there is not enough work. Also, in this case MADNESS does not distribute work evenly between compute nodes, but rather attempts to achieve work locality on compute nodes depending on the shape of the highly unbalanced tree. The work distribution in MADNESS is done according to a process map specification.

Experimental results for a much larger application (a 4-dimensional Time-Dependent Schrodinger Equation — TDSE) are presented in Table 3.6.

3.6.1 Analysis of Results

Table 3.3 presents results for running a *Coulomb* application with precision 10^{-10} . This application scales well up to 16 nodes. Above this threshold there are too few tasks for good load balancing. MADNESS uses static load balancing. To scale above 16 nodes, a larger *Coulomb* application (with precision 10^{-11}) was needed (see Table 3.4). This second application consists of 154,468 tasks, and it scales up to 100 nodes. Larger applications would scale beyond 100 nodes. The 4-dimensional TDSE application (see Table 3.6) scales up to 500 nodes. It consists of 542,113 tasks, but these tasks have more computation than the tasks for the 3-dimensional *Coulomb* application, since

Compute nodes	Application Coulomb with input parameters $d = 3$, $k = 30$ and precision 10^{-12}				
	Time (seconds)				
	CPU-only compute		GPU-only compute	CPU-GPU compute no rank red.	
	rank red.	no rank red.		Actual	Optimal CPU-GPU Overlap
1	147	447	212	172	144
2	115	299	90	60	69
4	114	234	55	39	45
6	96	201	35	25	30
8	102	205	37	25	31

Table 3.5: CPU-only, GPU-only and hybrid computation scale-up with the increase in number of compute nodes. 3-D tensors with $k = 30$ are used. The desired precision is set to 10^{-12} . In the presented results, the CPU-only compute version uses 16 threads, while the GPU-only compute and hybrid versions use 6 CUDA streams and 15 CPU threads. The CPU-GPU dispatcher thread is also active. For the hybrid CPU-GPU case, the CPU-only compute and GPU-only compute times were taken into account in order to divide work optimally between the CPU and the GPU.

the matrices are 2-dimensional projections of 4-dimensional tensors.

Notice that the speedup of the computation with respect to the number of compute nodes in Table 3.5 is not linear because work is not distributed evenly to all compute nodes. MADNESS uses the concept of a process map to specify distribution of tasks to nodes. In the presented tests (except Table 3.3 and Table 3.4), the process map assigns more work to some of the nodes. Note that there is no speedup from 6 to 8 compute nodes — under the current process map there is not enough work to distribute to 8 compute nodes in this test.

Also, for both 6 and 8 compute nodes, the theoretical optimal CPU-GPU computation overlap, calculated taking into account the time for CPU-only compute and GPU-only compute, is higher than the actual time obtained. This can happen because the formula for calculating the optimal CPU-GPU

Compute nodes	4-D Time-Dependent Schrodinger Equation $k = 14$, precision 10^{-14} (with rank reduction)				
	Time (seconds)				
	CPU-only compute	GPU-only compute (using cuBLAS)	CPU-GPU compute		Speedup of CPU-GPU version over CPU-only version
			Actual	Optimal CPU-GPU Overlap	
100	985	873	664	463	1.4
200	759	580	524	329	1.4
300	739	533	308	310	2.3
400	718	448	299	276	2.4
500	648	339	277	223	2.3

Table 3.6: Timings for the **Apply** part of the MADNESS 4-dimensional Time Dependent Schrodinger Equation (TDSE) for $k = 14$ and threshold 10^{-14} on Titan. Various runs used between 9 and 14 CPU threads. Observations showed no significant scale-up when using more than 9 threads, so the variable number of CPU threads used does not impact reported results.

overlap considers the applications to be 100% compute-intensive. However, this is not the case in practice: the tested applications also have data-intensive parts that account for a non-dominating, but still significant fraction of the running time.

An issue for the CPU-only version for larger tensors (as in the case of Table 3.5) is that the computation is saturated by 10 threads, because the working set size is much larger than 16 MB, which is the aggregate size of the L2 cache on the compute nodes of Titan.

The same statement is true for the CPU-only version of an even larger computation, the 4-dimensional Time Dependent Schrodinger Equation (TDSE).

A 4-dimensional TDSE computation (see Table 3.6) requires hundreds of compute nodes. For the operations with larger tensors employed in this application we used cuBLAS, since this is the regime in which cuBLAS performs well (large matrix multiplications). Aside from custom CUDA kernels, all other MADNESS extensions were used for these tests. As discussed before, the scale-up is not linear, because of the way MADNESS distributes work

using process maps.

The results in Table 3.6 show that the GPU version scales better than the CPU version. The reason is that some compute nodes are assigned too few tasks at one time to constantly keep all the CPU cores busy. Currently there is no MADNESS CPU implementation of multiple threads working on the same multiplication, whereas for the GPU there is (by using cuBLAS). Therefore, by using the GPU we can efficiently perform large multiplications that are slow on the CPU. The overlapping of the CPU and GPU computation yields good results for 4-dimensional TDSE, as presented in Table 3.6.

Once again we notice the “super-optimal” overlap of CPU and GPU computation in Table 3.6, for the case of 300 nodes. As explained before, the reason is the contribution of the data-intensive part of the computation, which is not taken into account when the work division between the CPU and the GPU is performed. The CPU executes all *preprocess* and *postprocess* tasks, which are heavily data-intensive. In addition, the dispatcher CPU thread has to rearrange and batch data for the GPU, which also incurs extra overhead.

3.7 Related Work

There has been significant related work that addresses both aspects (code reorganization and GPU execution) of migrating an existing HPC scientific framework to hybrid CPU-GPU computing. Since this chapter is concerned with data and control flow reorganization, we focus on related work for this aspect.

MapReduce-like Frameworks for Hybrid CPU-GPU A number of MapReduce frameworks have been proposed for GPU programming in recent years: for NVIDIA GPUs He et al. proposed Mars [HFL⁺08] and Catan-

zaro et al. proposed [CSK08], while for AMD GPUs Elteir et al. proposed [ELcFS11]. For multi-GPU programming, GPMR [SO11] was proposed. These frameworks only address designing the GPU computation phase as a MapReduce operation, and do not provide solutions for hybrid CPU-GPU computing. They are orthogonal to the solution proposed in this chapter.

MapCG [HCC⁺10] is a MapReduce-like frameworks for hybrid CPU-GPU computations. Users can write a MapCG or GPMR program and the framework will translate it into either a many-core CPU implementation, a GPU implementation, or a hybrid CPU-GPU implementation. Both MapCG and GPMR have mostly been employed for applications that heavily favor either the CPU or the GPU.

Dataflow Graphs for Hybrid CPU-GPU An interesting approach to providing a high-level programming model for hybrid CPU-GPU systems is CnC-CUDA [GSBS11], a declarative deterministic coordination language that is an extension of Intel’s Concurrent Collections (CnC) [Kno09]. Programming in CnC-CUDA involves expressing the computation as a static graph in which the nodes can be dynamic computation items, data items and control items.

Closely related to our work is the SkePU [DKT11] skeleton-based framework for hybrid CPU-GPU computing. SkePU provides a set of complex programming constructs, named skeletons, that are well-suited for GPU programming (such as *Map*, *Reduce*, *MapReduce*, *MapOverlap*, *MapArray*, *Scan* — for data parallelism, and *Farm* — for task parallelism). SkePU can be used in conjunction with StarPU [ATNW11] (a runtime system for heterogeneous multicore platforms).

Approaches to General Heterogeneous Computing General heterogeneous computing frameworks, such as EXOCHI [WCC⁺07], Merge [LCWM08],

Harmony [DY08] or StarPU [ATNW11] provide a unified interface for programming on heterogeneous systems. All four frameworks require the developer to express the computation in terms of high-level tasks. The developer also provides architecture-specific implementations for each high-level task for each architecture of interest. Each of these four frameworks implements a scheduler that schedules tasks to all components of the heterogeneous system.

A general approach to programming on heterogeneous multi-threaded systems is presented by EXOCHI [WCC⁺07]. This framework consists of two parts: EXO (Exoskeleton Sequencer) and CHI (C for Heterogeneous Integration). EXOCHI generalizes the concept of a CPU-only multi-threaded system to hybrid “multi-core” architectures such as CPU-GPU, under the name multi-shredded. A “shred” is a generalization of a CPU thread, and can be executed on any type of “core” provided by the hybrid system. Users are responsible for providing the code for each architecture in the system. All instances of architecture-specific code is compiled into a single fat binary. The runtime system determines which architecture-specific code is chosen for a specific shred. EXOCHI does not implement aggregation of data inputs. To improve data access performance between devices, it implements a shared virtual memory between all devices. An address translation mechanism is used to provide a uniform addressing scheme. This mechanism only works for devices that share the same physical memory (although they may have different address spaces). For devices that do not share the same physical memory (such as CPUs and NVIDIA GPUs in the new hybrid clusters) this mechanism cannot be used. The EXOCHI framework can be used in these cases as well, but data transfer between the devices will be less efficient. EXOCHI also needs some operating-system support for the features it offers, including the shared virtual memory mechanism.

Merge [LCWM08] is a MapReduce-like framework built on top of EXOCHI. It exposes a high level of abstraction to the programmer, thus increasing productivity while also leveraging the high-efficiency benefits of EXOCHI. Architecture-specific task implementations still need to be available. A solution is specified by invoking map and reduce operations over a set of data inputs. Each separate task executed in parallel by a map operation can either be decomposed into other smaller tasks by the task code, or it can be processed without decomposition. This execution model creates a dependency tree. When a task is no longer decomposed, the runtime library automatically invokes one of the architecture-specific variants to process the task data. Tasks in Merge are functional closures and cannot access global data that has not been copied to the closure.

Harmony [DY08] is a programming and execution model for heterogeneous computing that is based on dynamically detecting and tracking dependencies between compute kernels. The approach is inspired by out-of-order instruction scheduling in superscalar processors. Harmony uses compute-kernel abstractions that allow the runtime environment to build directed acyclic graphs that define data and control flow. Harmony provides the possibility of speculative execution. This requires a mechanism to recover from mis-predicted branches. Harmony does not address aggregation of data inputs and tasks. Therefore, it is orthogonal to the solution proposed for MADNESS.

StarPU [ATNW11] is a runtime system for heterogeneous multicore platforms that provides task abstraction, data management and dynamic scheduling. As in the case of Harmony, StarPU also does not address aggregation of data inputs and tasks. Both Harmony and StarPU can be seen as orthogonal to the proposed solution for the **Apply** operator, and could theoretically act as schedulers for the MADNESS CPU and GPU tasks.

High-level Solutions for obtaining Efficient GPU Code Writing efficient GPU code is a cumbersome task. For linear algebra, libraries such as cuBLAS have been developed. We have found that cuBLAS is efficient for some irregular kernels, as long as the computation to data access ratio is not very small.

Recent hybrid computing solutions such as the CAPS Hybrid Multi-core Parallel Programming (HMPP) [DBB], OpenACC [Opea] or the OpenMP extensions of [ABC⁺09] rely on the developer inserting compiler directives in the code for GPU processing. Both HMPP and OpenACC have been inspired by OpenMP [Opeb], a compiler-directive approach to multi-threaded CPU computing. While compiler-directive approaches are less efficient than custom kernels, they increase the programmer productivity and allow the quick testing of multiple kernel algorithms. A mixed approach, in which development starts with HMPP kernels to detect the best algorithm, and continues with implementing that algorithm in low-level CUDA code, can be envisioned. There is currently a MADNESS project that focuses on coupling the work presented in this chapter with HMPP kernels for **Apply**.

Acknowledgments We wish to thank Judith C. Hill for some test cases and for valuable assistance in using Titan.

CHAPTER 4

Permutation Multiplication for Many-Core Machines and Parallel Disks

This chapter presents an extension of a cache-aware sequential data-intensive computation (permutation multiplication) to many-core machines and parallel disks.

Algorithms are introduced for efficiently executing the basic operations for large permutations, that range in size from 4 million points to billions of points [SDKC10].

The standard sequential permutation algorithm can be described as:

X, Y, Z arrays of size N with indices $0 \dots N - 1$.

$X[i] \in \{0 \dots N - 1\}, \forall i \in \{0 \dots N - 1\}$.

for $i \in \{0 \dots N - 1\}$ $Z[i] = Y[X[i]]$. ,

for input permutation arrays $X[]$ and $Y[]$, and output permutation array $Z[]$.

The object of this chapter is multiplication of random permutations. In this regime, almost every iteration of the traditional algorithm incurs a cache miss.

The size of the permutation dictates the preferred architecture. At the high end of our regime (billions of points), the preferred architecture consists

of parallel disks. At the low end of our regime (millions of points), the preferred architecture consists of a commodity many-core machine (the data must fit in the RAM of the machine).

When using the many-core architecture, one has a choice of using a multi-threaded algorithm or multiple independent single-threaded processes. Both regimes of computation are useful. Where independent computations from a parameter sweep are performed, or where a parallelization of the higher algorithm is available, independent single-threaded processes are preferred. Where a single inherently sequential algorithm is the goal, the multi-threaded algorithm is preferred. Experimental results show a 50% speedup in both cases. This novel algorithm has its primary advantage for permutations large enough that they overflow the CPU cache.

In addition to the problem of permutation multiplication, two other standard permutation operations are typically supported by permutation subroutine packages: *permutation inverse* and *permutation multiplication by an inverse*. The last problem, $X^{-1}Y$, is often included as a primitive operation because there exists a more efficient implementation than composing inverse with permutation multiplication:

for $i \in \{0 \dots N - 1\}$ $Z[X[i]] = Y[i]$

More formally, the problems are:

Let X and Y be two arrays with the same number of elements N , both indexed from 0 to $N - 1$, such that:

$$0 \leq X[i] \leq N - 1, \forall i \in \{0 \dots N - 1\}$$

Problem 4.0.1 (Multiplication). *Compute the values of another array, Z , with N elements, defined as follows:*

$$Z[i] = Y[X[i]], \forall i \in \{0 \dots N - 1\}$$

Problem 4.0.2 (Inverse). *Compute X^{-1} such that:*

$$X[X^{-1}[i]] = X^{-1}[X[i]] = i, \forall i \in \{0 \dots N - 1\}$$

Problem 4.0.3 (Multiply by Inverse). *Compute the result of multiplying a permutation by an inverse $X^{-1} \times Y$:*

$$Z[i] = Y[X^{-1}[i]], \forall i \in \{0 \dots N - 1\}$$

Terminology We present three permutation multiplication algorithms for architectures with at least two levels of memory, in increasing order of performance: the “*external sort algorithm*”, the “*buckets algorithm*” and the “*implicit indices algorithm*”.

Definition 1. Memory Hierarchy

The terminology “fast-memory/slow-memory” refers to an algorithm that uses slow-memory as the slower, much larger lower-level memory (the one on which the permutation arrays are stored), and fast-memory as the faster, much smaller higher-level memory (that cannot hold the entire permutation arrays).

The rest of the chapter is organized as follows: Section 4.1 presents the problem background, Sections 4.2 and 4.3 present our new fast algorithms, along with some theoretical considerations on their performance. Section 4.4 presents new fast algorithms for permutation inverse and multiplication by an inverse. Section 4.5 presents formulas for the optimal running time, under the assumption that the CPU cores are infinitely fast and that the single bus from CPU to RAM is the only bottleneck (or time to access disk). Section 4.6 presents the experimental results.

Overview of the Algorithms Seven algorithms are presented. The algorithms are:

Sequential Implicit Indices for Multiplication Algorithm 13 reviews an older method for permutation multiplication [CM02, CR03], here called *implicit indices*.

Multi-threaded Implicit Indices for Multiplication Algorithm 14 constitutes the central novelty of this work. It presents a multi-threaded parallel permutation multiplication algorithm.

Parallel disk-based Implicit Indices for Multiplication Tables 4.4 and 4.5, along with Section 4.2.2, present a generalization of the Multi-threaded Implicit Indices algorithm to parallel disks.

Sequential Implicit Indices for Inverse Algorithm 15 reviews an older algorithm for permutation inverse [CM02, CR03] that is analogous to Algorithm 13. The generalization to the multi-threaded case is analogous to Algorithm 14.

Sequential Implicit Indices for Multiplication by Inverse Algorithm 16 reviews an older algorithm for permutation multiplication by inverse [CM02, CR03], that is analogous to Algorithm 13. The generalization to the multi-threaded case is analogous to Algorithm 14.

Parallel disk-based External Sort for Multiplication Algorithm 11 uses external sorting and is intended solely to explore the design space.

Parallel disk-based RAM-buckets for Multiplication Algorithm 12 uses a buckets technique and is intended solely to explore the design space.

Section 4.5 presents a new timing analysis applicable to Algorithms 13, 14, 15 and 16 and their parallel generalizations. Experimental results are presented in Table 4.8.

4.1 Problem Background

The current work builds upon [CM02]. In that work, the authors present a fast RAM-based permutation algorithm that worked well on the Pentium 4, due in part to the 128-byte cache line on that CPUs. Most later CPUs have 64-byte cache lines, and so that algorithm, which is reviewed in Algorithm 13, later achieved mixed results. Algorithm 13 was also used as a sequential disk-based algorithm in [CR03]. Related sequential algorithms for permutation inverse and permutation multiplication by inverse were also described in [CM02, CR03].

4.2 Permutation Multiplication using External Memory

New algorithms for large permutations are presented. For many problems in computational group theory, the size of a permutation is in the range of tens to hundreds of gigabytes.

The first case presented below deals with permutations that fit on a single disk, with a permutation occupying at least 10 GB of space, but not more than 50 GB. These same algorithms can be run on flash memory. Table 4.2 presents experimental results obtained by running our implicit indices algorithm both on flash and on disk. In the following three subsections one can replace disk with flash and everything remains correct.

4.2.1 Local Disk and Flash

The traditional implementation for permutation multiplication is: `for (i = 0; i < N; i++) Z[i] = Y[X[i]]`. Using this implementation would be

impractical. The reason is that, for large enough pseudo-random permutations, most array accesses are to random locations on disk. Thus a memory page would be swapped in from disk at almost every array element access. On most current systems a memory page is on the order of 4 KB. If the element size is 8 bytes, then for each 8 bytes the traditional algorithm accesses, the system would actually transfer 4 KB of data, which results in a $4\text{ KB}/8\text{ bytes} = 512$ times ratio of transferred to useful data. This was indeed observed for naive permutation multiplication running in virtual memory (see Table 4.3).

A few important notions are defined before discussing the details of the three new algorithms for external memory.

Definition 1. System and Algorithm Parameters

The values in each permutation array X , Y and Z can be represented on b bytes.

$Hlms$ = the size of the higher-level memory component, in number of elements of b bytes.

Any arrays used in the algorithms can be divided into blocks of length $Bl = (Hlms/2)$ number of elements. Two blocks must simultaneously fit in $Hlms$.

$Nb = N/Bl$ is the total number of blocks in an array.

4.2.1.1 Using External Sort

The disk-based permutation multiplication method using external sorting is described in Algorithm 11.

Using the concept of buckets that fit in RAM one can significantly improve the performance of the algorithm. RAM buckets are an alternative to

Input: Permutation arrays X and Y , of size N

Output: Z , s.t. $Z[i] = Y[X[i]]$, $\forall i \in \{0 \dots N - 1\}$

Phase 1: Scan X and, for each index i , save the pair $(i, X[i])$ to an array D on disk.

Phase 2: Externally sort all pairs $(i, X[i])$ in array D increasingly by $X[i]$. Now $\forall j \in \{0 \dots N - 1\} \exists i \in \{0 \dots N - 1\}$ such that $D[j] = (i, X[i])$ and $X[i] = j$.

Phase 3: Scan both array Y and the pairs $(i, X[i])$ in the array D at the same time. $\forall j \in \{0 \dots N - 1\}$ we have $D[j] = (i, X[i])$, such that $X[i] = j$. Save the pair $(i, Y[j])$ to an array D' on disk.

Phase 4: Externally sort the array D' increasingly by the index i in pairs $(i, Y[j])$. Now the D' array contains pairs $(i, Y[X[i]])$ in increasing order of i . For each index i , copy $Y[X[i]]$ to the i index in the Z array.

Algorithm 11: Permutation Multiplication Using External Sort.

external sorting which trades the $n \log n$ running time of sorting for random access within RAM. RAM buckets have significantly sped up computations that previously used external sorting [KC09].

4.2.1.2 Using RAM Buckets

The RAM buckets method is described in Algorithm 12.

The RAM bucket size has to be chosen such that two RAM buckets simultaneously fit in RAM. Considering that both the index i and the value $X[i]$ are represented using the same number of bytes, one needs $2 \times N/Hlms$ buckets (here $Hlms$ is the size of RAM).

Algorithm 12 presents a few important improvements over Algorithm 11. Note that in phase 2 of Algorithm 12, there is no need to save the index in the buckets of array Y , since it is implicit in the ordering. Thus a bucket of array Y occupies twice as little space as a bucket of pairs $(i, X[i])$. In phase 3, Z is also divided into $2 \times N/Hlms - 1$ buckets, and all indices from the j -th bucket of D' correspond to positions in the j -th bucket of Z .

Algorithm 12 completely eliminates sorting and, in practice, shows a 4 times

Input: Permutation arrays X and Y , of size N
Output: Z , s.t. $Z[i] = Y[X[i]]$, $\forall i \in \{0 \dots N - 1\}$

- 1: All arrays are split into Nb equally sized buckets, each containing $B_l = N/Nb$ elements. The bucket size can be at most one-half the size of RAM. Bucket i of array A is denoted A_i . Bucket b contains indices in the range $[b * B_l, (b + 1) * B_l)$.
// **Phase 1: bucketize**
- 2: Scan array X and, for each index i , save the pair $(i, X[i])$ in the bucket $D_{X[i]/B_l}$.
// **Phase 2: permute buckets**
- 3: **for** each bucket b **do**
- 4: Load buckets D_b and Y_b into RAM.
- 5: **for** each index i in this bucket **do**
- 6: Let $D_b[i] = (j, X[j])$.
- 7: Save the pair $(j, Y_b[X[j]])$ to bucket D'_{j/B_l} .
- 8: **end for**
- 9: **end for**
// **Phase 3: combine buckets**
- 10: **for** each bucket b **do**
- 11: Load buckets D'_b and Z_b into RAM.
- 12: **for** each index i in this bucket **do**
- 13: Let $D'_b[i] = (j, Y[X[j]])$.
- 14: Set $Z_b[j] = Y[X[j]]$.
- 15: **end for**
- 16: **end for**

Algorithm 12: Permutation Multiplication Using RAM buckets.

(or more) speedup over the External Sort-based algorithm if the computation is disk-bound (see Table 4.5, the 1 node case).

Both algorithms 11 and 12 need to save the index of each value of the X permutation, thus resulting in disk arrays as large as twice the size of the initial arrays. The implicit indices RAM/disk algorithm (Algorithm 13) avoids saving the indices to disk arrays.

4.2.1.3 With Implicit Indices

The implicit indices method is described in Algorithm 13.

The correctness of Algorithm 13 can be proved by following the three

Input: Permutation arrays X and Y , of size N

Output: Z , s.t. $Z[i] = Y[X[i]]$, $\forall i \in \{0 \dots N-1\}$

- 1: All arrays are split into Nb equally sized buckets, each containing $Bl = N/Nb$ elements. The bucket size can be at most one-half the size of RAM. Bucket i of array A is denoted A_i . Bucket b contains indices in the range $[b * Bl, (b+1) * Bl)$.

// **Phase 1: bucketize**

- 2: Traverse the X array and distribute each value $X[i]$ into bucket $D_{X[i]/Bl}$ on disk.

// **Phase 2: permute buckets**

- 3: **for** each bucket b **do**
- 4: Load buckets D_b and Y_b into RAM.
- 5: **for** each index i in this bucket **do**
- 6: Set $D_b[i] = Y_b[D_b[i]]$.
- 7: **end for**
- 8: **end for**

// **Phase 3: combine buckets**

- 9: For each value $X[i]$, let j be the next value in bucket $D_{X[i]/Bl}$. Note that $j = Y[X[i]]$. Set $Z[i] = j$ and remove that value from bucket $D_{X[i]/Bl}$.

Algorithm 13: Permutation Multiplication using Implicit Indices.

phases for a generic index $i \in \{0 \dots N-1\}$: in phase 1 value $X[i]$ is distributed into bucket $j = X[i]/Bl$ of array D at position k , so that $D[k] = X[i]$. In phase 2, $D[k] = Y[D[k]]$, which can be written $D[k] = Y[X[i]]$. In phase 3, $Z[i] = D[k]$, which can be written $Z[i] = Y[X[i]]$.

The implicit indices version runs about twice as fast as the buckets version (see Table 4.4). The implicit indices RAM/disk algorithm performs the following steps: a sequential read of the X array and a sequential write of the D (temporary) array in phase 1 (*2 sequential accesses*); a sequential read of the D (temporary) array, a sequential read of the Y array and a sequential write of the D array (*3 sequential accesses*); a sequential read of the X array, a sequential read of the D (temporary) array and a sequential write of the Z array (*3 sequential accesses*). In total, there are 8 sequential accesses.

It is interesting to compare the running time of the implicit indices algorithm and the running time of a permutation multiplication algorithm that we

implemented in Roomy [Kun10]. Roomy is a general framework for disk-based and parallel disk-based computing which provides a high-level API for manipulating large amounts of data. The disk-based implicit indices algorithm is generally twice as fast as the Roomy implementation.

4.2.2 Parallel Disks

Here we describe how the three disk-based algorithms for permutation multiplication, presented in Section 4.2.1, can be used with the parallel disks of a commodity cluster.

Serial permutation multiplication using external sort is described in Algorithm 11. To parallelize it, all arrays are first split into sub-arrays. Each sub-array is placed on the disk of a single compute node in the cluster. All operations on those arrays are performed in parallel. In cases where one node generates data that references a sub-array on another node, that data is first batched locally, then sent over the network. Upon arrival at the destination cluster node, it is saved to disk. In our implementation, there is a separate thread of execution on each node that handles the writing of this remote data to the local disk. Finally, there is a synchronization point after each phase, to insure that all nodes are done with one phase before beginning the next.

Permutation multiplication using buckets (Algorithm 12) is made parallel in the same way. The arrays are split into sub-arrays (buckets), and the same methods are used for data distribution, parallel processing, and synchronization.

There is one additional modification necessary to parallelize permutation multiplication using implicit indices (Algorithm 13). Because the algorithm depends on the specific ordering of elements in each bucket, the buckets can not be written to in parallel. This is solved in the same way that Algorithm 14

extends Algorithm 13: each bucket is further split into sub-buckets, so that each node has its own sub-bucket to write to. Unlike the multi-threaded RAM case, the parallel disk case does not need an extra phase to compute the sizes of the sub-buckets, since the buckets are represented with files, which are dynamically sized.

4.3 Permutation Multiplication in RAM

The traditional permutation multiplication algorithm for cache/RAM allows for a trivially-parallel version. Each thread processes a contiguous region of the $X[]$ permutation array. Although this incurs frequent cache misses, it tends to scale linearly on current commodity computers *until one goes beyond four cores*. This is because the single bus to RAM becomes saturated by the pressure of the several cores. In Table 4.8 of Section 4.6, one sees this happening approximately with 3 threads for permutation multiplication and for 4 threads for inverse and multiplication by inverse.

Algorithm 13 of Section 4.2 presented a single-threaded disk-based algorithm to overcome the many page faults. The same algorithm can be implemented for cache/RAM to minimize cache misses. That algorithm's cache/RAM version is preferred for permutation algorithms that can be parallelized at a higher level and then call a single-thread permutation multiplication algorithm. Here, we consider a multithreaded version for the case when the higher algorithm does not parallelize easily. The corresponding results at the level of eight cores are presented in Table 4.7 in Section 4.6. As described in the extrapolation in Section 4.6.3, both the new single-threaded and the new multi-threaded algorithms are expected to have a similar advantage at the 16-core and higher level in the future.

- Input:** Permutation arrays X and Y , of size N , the number of cache buckets Nb , the number of threads T .
- Output:** Z , s.t. $Z[i] = Y[X[i]]$, $\forall i \in \{0 \dots N - 1\}$
- 1: All arrays are split into Nb equally sized buckets, each containing $Bl = N/Nb$ elements. The bucket size can be at most one-half the size of cache. Bucket i of array A is denoted A_i . Bucket b contains indices in the range $b \times Bl$ to $(b + 1) \times Bl$.
 - 2: Each thread t , $0 \leq t \leq T - 1$, will handle indices in the range $t \times N/T$ to $(t + 1) \times N/T - 1$.
// Phase 1: create sub-buckets
 - 3: Create a temporary array D , split into $T \times Nb$ sub-buckets. $D_{b,t}$ is the sub-bucket corresponding to bucket b and thread t . The bucket D_b is the concatenation of all sub-buckets $D_{b,t}$. The size of a sub-bucket is first determined by an additional scan of X .
// Phase 2: bucketize
 - 4: Each thread scans the portion of X that it is responsible for, and saves each $X[i]$ to sub-bucket $D_{t,X[i]/Bl}$.
// Phase 3: permute buckets
 - 5: Each thread locally permutes each bucket b that it is responsible for, setting $D_b[i] = Y_b[D_b[i]]$.
// Phase 4: combine buckets
 - 6: Each thread computes the final values $Z[i]$ that it is responsible for. For each such index i , let j be the next value in sub-bucket $D_{t,X[i]/Bl}$ that has not been removed (Note that $j = Y[X[i]]$). Set $Z[i] = j$ and remove that value from sub-bucket $D_{t,X[i]/Bl}$.

Algorithm 14: Multi-threaded cache/RAM Permutation Multiplication using Implicit Indices.

Algorithm 14 provides the multi-threaded version for multiplication using cache/RAM. Intuitively, it operates by splitting the buckets of Algorithm 13 into sub-buckets. Within a given bucket, each thread “owns” a contiguous region (a sub-bucket) for which it has responsibility. Algorithm 14 requires one extra phase (Phase 1) in order to determine in advance the size of the sub-bucket to allocate for each thread.

Some alternative designs were also explored. A brief summary of the alternatives considered is presented along with our reasons for rejecting them:

- Using pthread private data via “_thread” (problem: uses too much

memory);

- Using pthread locks to synchronize memory access (problem: synchronization delays);
- Using an atomic add operation to a single global counter (problem: internally, it still uses a lock); and
- Exploiting L1 cache via a two-level algorithm, similar to two-level external sort (problem: delays due to extra passes).

There are optimizations one can apply to Algorithm 14. In phase 2 the most important optimization is saving the results of the divisions $X[i]/Bl$ to position i in array Z . In phase 4, instead of redoing some of the computations from phase 2, it is enough to read the value in $Z[i]$. X is not accessed in phase 4 anymore. We implemented these optimizations and they prove to be significant on some architectures. Having to scan one array less in phase 4 provides better L1 cache behavior, which results in a faster algorithm.

Section 4.6.3 presents experimental results for the cache/RAM multi-threaded implicit indices algorithm.

4.4 Permutation Inverse. Multiplication by an Inverse

While Algorithms 15 and 16 are not new [CM02, CR03], their multi-threaded generalizations analogous to Algorithm 14 are novel. Experimental results for running permutation inverse and multiplication by an inverse, as well as theoretical estimates for these runs, can be found in Table 4.8.

4.4.1 Permutation Inverse

To compute permutation inverse the traditional algorithm looks like this:

```
for (i = 0; i < N; i++) Y[X[i]] = i;
```

The bottleneck is still the random access (this time write access) to the Y array. We are proposing a 2-phase fast RAM/disk algorithm for permutation inverse (we need two additional arrays, D and D'):

Input: Permutation array X , of size N

Output: $Y[X[i]] = i, \forall i \in \{0 \dots N - 1\}$

Phase 1: Scan array X and distribute each value $X[j]$ in array D at block number $k = X[j]/Bl$. At the same time write value j at the same index in block k of D' as $X[j]$ was written at in block k of D .

Phase 2: Scan the D' and D arrays sequentially at the same time and, for each index j , write $Y[D[j]] = D'[j]$.

Algorithm 15: Permutation Inverse Using Implicit Indices.

4.4.2 Permutation Multiplication by an Inverse

If X and Y are two permutations, we call a multiply-by-inverse $Y^{-1}[X[i]], \forall i \in \{0 \dots N - 1\}$.

For the multiply-by-inverse, the traditional algorithm is:

```
for (i = 0; i < N; i++) Z[X[i]] = Y[i];
```

At the end of the loop $Z[i] = Y[X^{-1}[i]], \forall i \in \{0 \dots N - 1\}$. We have developed a 2-phase algorithm for multiply-by-inverse, which looks very much like the 2-phase algorithm for inverse.

Phase 2 for multiply-by-inverse is identical to phase 2 for inverse.

4.5 Performance Analysis

The analysis presented here can be used to estimate the running time for the implicit indices algorithms, when using any 2-level memory hierarchy, includ-

Input: Permutation arrays X and Y , of size N

Output: $Z[i] = Y[X^{-1}[i]]$, $\forall i \in \{0 \dots N - 1\}$

Phase 1: Scan array X and distribute each value $X[j]$ in its corresponding block of array D . At the same time write value $Y[j]$ at the same index in D' as $X[j]$ was written at in D .

Phase 2: Scan the D' and D arrays sequentially and, for each index j , write $Y[D[j]] = D'[j]$.

Algorithm 16: Permutation Multiplication by an Inverse Using Implicit Indices.

ing cache/RAM, RAM/flash, RAM/disk. The implicit indices algorithms include Algorithm 13, its generalization to Algorithm 14, Algorithms 15 and 16, and their parallel generalizations.

Definition 2. System and Algorithm parameters (Analysis)

Hrl = higher-level read memory latency (*seconds*)

Lrl = lower-level read memory latency (*seconds*)

Lwl = lower-level write memory latency (*seconds*)

Lwb = lower-level write memory bandwidth (*bytes/second*)

Lrb = lower-level read memory bandwidth (*bytes/second*)

Es = array-element size (*bytes*)

N = array length (*bytes*)

Nb = number of blocks per array

Bs = bucket size (*bytes*)

$Bl = N/Nb$ (block length) (*bytes*)

We refer to permutation multiplication as PM, to permutation inverse as PI, and to permutation multiplication by an inverse as PMI. The next three formulas estimate the running time when memory is the bottleneck for PM, PI, and PMI, respectively. Note that in the case of cache/RAM N/Lrb must be added to each formula, due to the extra pass.

4.5.1 Permutation Multiplication: Performance

An evaluation of the implicit indices algorithm performance phase by phase follows. In phase 1, the running time can be estimated by:

$$Nb \times Bl/Lrb + (Nb \times (Lwl + Bs/Lwb)) \times (Bl/Bs)$$

Since $Bl = N/Nb$, this results in:

Formula 4.5.1 (PM Phase 1 Running Time).

$$\frac{N}{Lrb} + Lwl \times \frac{N}{Bs} + \frac{N}{Lwb}$$

The only algorithm parameter the formula is dependent on is Bs . The time decreases as Bs increases, so it is better if one chooses Bs as large as possible, keeping in mind that phase 1 needs $Nb \times Bs$ memory only for the buckets. Usually Bs has to be much smaller than Bl for the buckets to fit in the higher-level memory.

In phase 2, the running time can be estimated by:

$$2 \times Nb \times \frac{Bl}{Lrb} + Nb \times \left(\frac{Bl}{Es} \times Hrl \right) + Nb \times \frac{Bl}{Lwb}$$

This can be reduced to:

Formula 4.5.2 (PM Phase 2 Running time).

$$N \times \left(\frac{2}{Lrb} + \frac{1}{Lwb} \right) + \frac{N}{Es} \times Hrl$$

In Phase 3 the estimate is:

$$Nb \times \frac{Bl}{Lrb} + \left(Nb \times \left(Lrl + \frac{Bs}{Lrb} \right) \times \left(\frac{Bl}{Bs} \right) + Nb \times \frac{Bl}{Lwb} \right)$$

which can be rewritten as:

Formula 4.5.3 (PM Phase 3 Running time).

$$Lrl \times \frac{N}{Bs} + N \times \left(\frac{2}{Lrb} + \frac{1}{Lwb} \right)$$

By adding up the estimated times for the three phases, we obtain a total estimated time:

Formula 4.5.4 (PM Total estimated time).

$$N \times \left(\frac{Lwl + Lrl}{Bs} + \frac{3}{Lwb} + \frac{5}{Lrb} + \frac{Hrl}{Es} \right)$$

The complexity of the implicit indices permutation multiplication algorithm is linear in the size of the permutation array, as long as all the other parameters don't change. In practice, observed system parameters fluctuate between runs, but not too much, making observed system parameters (read or write bandwidth, latency) stable enough to base the timing estimates on. Similarly, we have produced formulas for estimates of the running times of permutation inverse and permutation multiplication by an inverse.

4.5.2 Permutation Inverse: Performance

Phase 1 performs one sequential read of the X array and writes to blocks of two arrays D and D' by means of buckets (this means we have to take lower-level memory latency into account for each used bucket). The formula is:

Formula 4.5.5 (PI Phase I Running time).

$$\frac{N}{Lrb} + 2 \times Nb \times \frac{Bl}{Bs} \times Lwl + 2 \times \frac{N}{Lwb}$$

Phase 2 scans both arrays D and D' and writes to blocks of array Z , one after the other. Writing to a single block is done at random positions in that block (the entire block will be written, but in random order), so we have to take into account the latency of a random access in RAM. The estimate for this phase:

Formula 4.5.6 (PI Phase II Running time).

$$2 \times \frac{N}{Lrb} + Nb \times \frac{Bl}{Es} \times Hrl + \frac{N}{Lwb}$$

Adding the estimates for phases 1 and 2 the result is:

Formula 4.5.7 (PI Total estimated time).

$$3N \times \left(\frac{1}{Lrb} + \frac{1}{Lwb} + \frac{2 \times Lwl}{Bs} + \frac{Hrl}{Es} \right)$$

4.5.3 Permutation Multiplication by an Inverse:

Performance

As we have seen in Section 4.4, the new permutation multiplication by an inverse algorithm is similar to the new permutation inverse algorithm, with the exception of writing values $Y[i]$ in array D' instead of indices i . From a performance standpoint this only means that an extra array is read sequentially in phase 1. The estimate for this phase is:

Formula 4.5.8 (PMI Phase I Running time).

$$2 \times \frac{N}{Lrb} + 2 \times Nb \times \frac{Bl}{Bs} \times Lwl + 2 \times N/Lwb$$

The estimate for phase 2 is:

Formula 4.5.9 (PMI Phase 2 Running time).

$$2 \times \frac{N}{Lrb} + Nb \times \frac{Bl}{Es} \times Hrl + \frac{N}{Lwb}$$

And the total:

Formula 4.5.10 (PMI Total estimated time).

$$N \times \left(\frac{4}{Lrb} + \frac{3}{Lwb} + \frac{2 \times Lwl}{Bs} + \frac{Hrl}{Es} \right)$$

Table 4.1: Measured system parameters for external memory.

	Disk	Flash	Cluster disk
Read BW (MB/s)	85	200	51
Write BW (MB/s)	82	26	51
Latency (ms)	10	14	39
Latency RAM (ns)	233	211	169

4.6 Experimental Results

4.6.1 Local Disk and Flash

Tests were ran on an AMD Phenom 9550 Quad-Core at 2.2 GHz with 4 GB of RAM, running Fedora Linux with kernel version 2.6.29. The machine has both a disk drive (Seagate Barracuda 7200.10 250GB) and 2 RAID-ed flash SSD drives ($2 \times$ INTEL SSD SSDSA2MH080G1GC, 80 GB each).

Table 4.1 contains the measured system parameters of this machine. Table 4.1 also contains the measured system parameters for one of the disks of the cluster that was used to run the “parallel RAM/parallel disk” algorithms. The parallel disk bandwidth assumes that network bandwidth is not a limiting factor. Table 4.4 shows this to be the case for permutation arrays of size up to 20 GB.

Table 4.2 shows a comparison between the new RAM/disk algorithm and the new RAM/flash algorithm, both based on implicit indices. The estimates from the formulas of Section 4.5 are also presented, to confirm that the algorithm is limited by the bandwidth of disk and flash.

Table 4.3 details our findings about the traditional permutation multiplication algorithm ran in virtual memory on the same machine. The experimental results confirmed our expectations: when the working set is at least twice the size of available RAM, using the traditional algorithm in virtual memory is infeasible. We also implemented a buffered traditional algorithm and ran par-

Table 4.2: Running times of our new RAM/disk and RAM/flash algorithms and comparison with estimated running times. Element size is 8 bytes. Bucket size is 2 MB, block size is 1 GB.

Nr. elts. (billions)	Running Time (<i>seconds</i>)					
	Using Disk					
	PM		PI		PMI	
	real	est	real	est	real	est
1.25 (10 GB)	1609	1388	1002	1149	1253	1269
2.5 (20 GB)	3205	2776	2259	2298	2736	2538
	Using flash					
	PM		PI		PMI	
	real	est	real	est	real	est
1.25 (10 GB)	1584	1849	1212	1747	1348	1798
2.5 (20 GB)	2807	3698	2604	3494	2711	3596

Table 4.3: Comparison of the traditional algorithm and the buffered traditional algorithm with disk-based and flash-based external memory. Element size: 4 bytes. RAM size is 4 GB. Arrays X, Y and Z are the work set.

Nr. elts (millions)	traditional algorithm time (seconds)			
	sequential		parallel	
	disk	flash	disk	flash
750 (3.0 GB)	3476	1198	1802	489
825 (3.5 GB)	> 4hrs	> 4hrs	> 4hrs	> 4hrs
	Buffered algorithm time (seconds)			
	sequential		parallel	
	disk	flash	disk	flash
750 (3.0 GB)	150	130	142	115
825 (3.5 GB)	> 4hrs	11762	> 4hrs	3561

allel versions of both the simple traditional and buffered traditional algorithm. While the parallel buffered traditional algorithm clearly outperforms the parallel simple traditional one, the first is still infeasible when the working set overflows RAM by a significant percentage.

Table 4.4: Comparison of three parallel-disk permutation multiplication algorithms for increasing permutation size, using 16 nodes of a cluster. Elements are 8 bytes each. A * indicates that the estimated time is not accurate, because the network became a bottleneck.

Nr. elts. (billions)	Algorithm Time (seconds)			
	Sort	Bucket	Implicit Indices	
			real	estimated
0.8 (6 GB)	538	105	77	70
1.6 (12 GB)	1151	202	100	139
3.2 (24 GB)	3440	490	270	279
6.4 (48 GB)	7484	2364	1571	*
12.8 (95 GB)	15697	6838	3228	*

Table 4.5: Comparison of three parallel-disk permutation multiplication algorithms for increasing parallelism, using from 1 to 16 nodes of a cluster. Elements are 8 bytes each. Permutations have 1.6 billion elements each (12 GB).

Nr. nodes	Algorithm Time (seconds)			
	Sort	Bucket	Implicit Indices	
1	28952	7069		5576
2	13555	3627		2861
4	6197	677		354
8	2227	336		167
16	1185	202		100

4.6.2 Parallel Disks

These experiments were run on a cluster of computers, each with two dual-core 2.0 GHz Intel Xeon 5130 CPUs and 16 GB of RAM, a locally attached 500 GB disk, running Linux kernel version 2.6.9. Only one process was used per node, to avoid competition for the single disk.

Tables 4.4 and 4.5 give a comparison of the three disk-based permutation algorithms presented in 4.2.1, based on: external sorting; RAM buckets; and implicit indices.

Table 4.4 shows the results of using 16 nodes of a cluster, with permutation sizes ranging from 800 million elements (6 GB) to 12.8 billion elements

(95 GB). In general, the three algorithms scale roughly linearly with permutation size. The most notable exception is a 5-fold increase in the running times of the bucket and implicit indices algorithms when moving from 24 GB to 48 GB permutations. We believe that this is due to network traffic on an older Fast Ethernet switch. Until that point, the bottleneck was likely disk bandwidth. The sorting based algorithm does not see a similar effect because its time is dominated by the in-RAM sorting process, not inter-node communications.

Table 4.4 shows the results of using between 1 and 16 nodes of the cluster, with permutations having 1.6 billion elements (12 GB). Again, the time for each algorithm scales roughly linearly with the number of nodes. The non-linear scaling when moving from 2 to 4 nodes is likely due to the bottleneck moving between disk and the network.

In general, the bucket algorithm takes about 1.5 to 2 times longer than the implicit indices algorithm, with the largest differences occurring with larger permutations and more parallelism. The implicit indices algorithm is more efficient because of the smaller amount of data that must be saved to disk. The sorting based algorithm takes roughly 5 to 10 times longer than the implicit indices algorithm, largely due to the time needed to sort data in RAM.

4.6.3 RAM

For cache/RAM, the performance of permutation multiplication, inverse and multiplication by an inverse was demonstrated on a recent 8-core commodity machine: two Quad-core Intel Xeon E5410 CPUs running at 2.33 GHz, with a total of 24 MB L2 cache — 12 MB L2 cache per socket and 16 GB of RAM made up of four memory modules. Table 4.6 lists the system parameters measured on this system.

Table 4.7 concerns the case of independent permutation computations running in parallel, with one computation per core. We believe that the traditional algorithm is close to saturating the bandwidth from CPU to RAM, both in the case of 8 threads and 8 processes. Table 4.8 provides confirming evidence of bandwidth saturation in comparing 4 threads versus 8 threads. As described in Section 4.5, the new algorithm is more bandwidth-efficient. We see that benefit for 8 processes but not for 8 threads. We speculate that is due to cache poisoning as the threads compete for the same cache.

While we did not have physical access to the 8-core machine, the CPU (2 Quad-core Xeon E5410) and the measured maximum read bandwidth of 5.85 GB/s strongly imply that it uses DDR2-800 RAM, with a theoretical maximum read bandwidth is 6.4 GB/s, but in practice we can achieve at most 5.85 GB/s. The maximum expected memory bandwidth for the implicit indices algorithm is $(3 \times WriteBW + 5 \times ReadBW)/8$. Using Table 4.6 the maximum expected new algorithm bandwidth is 5106 MB/s. The actual bandwidth with 8 threads for the new algorithm is 3968 MB/s. The multi-threaded traditional algorithm performs one random read, one sequential read and one sequential write of arrays of size N . The random read is equivalent to 16 sequential reads, because it incurs a cache miss during most accesses (therefore for each 4-byte element access 64 bytes are actually transferred). Hence, the multi-threaded traditional algorithm performs operations equivalent to 17 sequential reads and one sequential write of the array. This in turn implies, based purely on hardware specifications, that the traditional algorithm is limited by memory bandwidth.

In Table 4.8 one can find running times for Algorithm 14 and the multi-threaded generalizations of Algorithms 15 and 16, as well as theoretical estimates of these running times based on the formulas in Section 4.5.

Table 4.6: Measured system parameters for cache/RAM. Latency for cache is negligible.

Read bandwidth	5859 MB/s
Write bandwidth	3850 MB/s
Latency 1 random access	302 ns

Table 4.7: Comparison of traditional and new algorithms, using thread or process-based parallelism. Permutations have 4 million 4-byte elements each.

	Eight Threads	Eight Processes
Traditional	0.042 s	0.048 s
New	0.054 s	0.026 s

The new permutation multiplication algorithm is faster by about 50% than the traditional algorithm for permutations of 32 million elements or more, when using 8 threads. Our new algorithm is also faster than performing 8 multi-threaded traditional permutation multiplications in a row by at least a factor of 1.6. In contrast, when using only one thread (with seven cores idle), the time represents a mixture of RAM bandwidth and CPU power. Hence, the traditional and new algorithms have similar performance.

Extrapolation on memory bandwidth results In the near future, commodity machines will continue to gain additional CPU cores at a rate based on Moore’s Law. But the number of memory modules on the motherboard is likely to remain fixed (while the density of each memory module continues to rise). Hence the memory bandwidth is unlikely to grow significantly.

Table 4.8 shows the times for the traditional algorithm already approaching an asymptotic value for the transition from 4 threads to 8 threads. Furthermore, the timings for 8 threads is close to the timing for the theoretically optimal case for bandwidth limited computation.

The new algorithm shows a significant improvement in time in the transition from 4 threads to 8 threads. In the case of permutation multiplication,

Table 4.8: Running times (seconds) of our new implicit indices permutation multiplication for cache/RAM. Comparison with estimated running times is only provided for the 8-thread case. As explained, we need a machine with at least 8 cores working on the new algorithm in parallel for the CPU to be a less significant factor. Element size is 4 bytes. A bucket here is a cache line, the block size is variable between runs.

Nr. elem. (millions)	Running Time (seconds)									
	1 thread		2 threads		4 threads		8 threads		Optimal	
32 (128 MB)	Permutation Multiplication									
	new	trad.	new	trad.	new	trad.	new	trad.	new	trad.
	0.81	0.81	0.70	0.51	0.43	0.45	0.29	0.42	0.25	0.39
	1.98	1.67	1.47	1.27	0.85	0.95	0.62	0.88	0.50	0.77
64 (256 MB)	4.68	4.09	2.98	2.52	1.72	2.10	1.16	1.81	1.01	1.54
128 (512 MB)										
32 (128 MB)	Permutation Inverse									
	new	trad.	new	trad.	new	trad.	new	trad.	new	trad.
	1.03	1.83	0.86	1.04	0.54	0.63	0.34	0.59	0.18	0.53
	2.39	3.70	1.75	2.06	0.96	1.31	0.66	1.21	0.37	1.06
64 (256 MB)	5.33	7.48	3.52	4.15	2.00	2.65	1.35	2.51	0.75	2.11
128 (512 MB)										
32 (128 MB)	Permutation Multiplication by an Inverse									
	new	trad.	new	trad.	new	trad.	new	trad.	new	trad.
	1.06	1.84	0.87	1.10	0.53	0.66	0.35	0.60	0.20	0.55
	3.72	2.46	1.77	2.24	0.99	1.33	0.70	1.23	0.41	1.10
64 (256 MB)	5.57	7.52	3.62	4.22	2.08	2.72	1.43	2.55	0.83	2.19
128 (512 MB)										

the timing for 8 threads approaches that of the theoretically optimal memory bandwidth limited case. On the other hand, the algorithms for permutation inverse and permutation multiplication by an inverse show the potential for additional improvements in timings as more cores become available. This is seen by comparing the numbers for 8 threads and the optimal case.

Acknowledgments We gratefully acknowledge CERN for making available an 8 core machine for testing.

CHAPTER 5

Lessons learned from the Migration of CPU-based Algorithms to NUMA Architectures

Each of the three solutions for NUMA architectures uses *delayed data access* and *delayed task execution* to hide architecture-specific problems. These delays are used to hide the latency of random access in the case of parallel disks, the latency of GPU data access and the low CPU-GPU transfer bandwidth in the case of hybrid CPU-GPU architectures, or the latency of non-cache-local data access in the case many-core CPUs.

In hindsight, we also observe that the three NUMA-based solutions each exhibit control and data flow that is related to *standard map-reduce* and *standard map-reduce* extensions. (We define the *standard map-reduce* control flow as the control flow of programs written in Google’s original MapReduce [DG04] or in Hadoop [Had].) However, each of the three solutions employs a different type of extension to *standard map-reduce*.

Section 5.1 presents a high-level description of the control and data flow of *standard map-reduce* programs. In Section 5.2 we show that the control and data flow of each of the three solutions of this dissertation are related to the

standard map-reduce control and data flow, even though none of them is implemented using a *map-reduce*-based library. We also present the extension(s) to the *standard map-reduce* control flow used by each of our three solutions. In each case, our extension(s) to *standard map-reduce* are also compared with existing extensions from the literature in Section 5.2. Other lessons learned from the contributions of this dissertation are presented in Section 5.3 (recursive algorithms for parallel disks) and Section 5.4 (a connection between the permutation multiplication operation and join-based operators). Section 5.5 briefly discusses high-level ideas related to the future impact of this dissertation. Appendix A presents the step-by-step correspondence between the control flow of previously described algorithms and the components of the *map-reduce* control flow.

5.1 The Control Flow of Standard Map-Reduce Programs

We define the *standard map-reduce* control flow as the control flow of programs written in Google’s original MapReduce [DG04] or in Hadoop [Had].

The *map-reduce* programming paradigm consists of two operations: mapping and reducing. Mapping consists of multiple control threads executing the same operation (a user-defined function) on the elements of a data collection independently. Each mapping thread will compute (key, value) data pairs independent of the other threads. All values with the same key will be distributed to the same batch. Reducing consists of accumulating all the values from the same batch in some way (specified by an often associative and commutative user-defined function).

In the original *MapReduce* framework [DG04], mapping and reducing are

represented by the operations *map* and *reduce*, respectively. Concretely, as presented in [DG04], the input of *map* is a collection of (k_1, v_1) key-value pairs (k_1 is the key type, and v_1 is the value type), that are distributed among the mapper control threads. Each mapper control thread produces a list of key-value pairs: *list-of* (k_2, v_2) . The (k_2, v_2) pairs are distributed, based on the key k_2 , to reducer control threads. (In *MapReduce*, this distribution is called *shuffling*.) All (k_2, v_2) pairs with the same k_2 will be processed by the same reducer thread. Thus, the input of a reducer thread consists of data $(k_2, \text{list-of}(v_2))$. The output of a reducer thread is a *list-of* (v_3) . The outputs of the *reduce* phase can be used as inputs for a subsequent *map-reduce* stage.

Figure 5.1 presents a high-level description of the control and data flow of *standard map-reduce* programs.

The *standard map-reduce* control flow is characterized by the *Map* and *Reduce* phases, as follows:

- *Map*: *Input data* items are assigned evenly to multiple control threads (mappers). Each control thread processes its own chunk of the *input data* items, and the results of the processing are distributed, according to problem-specific user-defined keys, into *reducer input data* batches (see Figure 5.1). (Here, *input data* refers either to the problem input or to data generated by the current computational step.)
- *Reduce*: Each *reducer input data* batch is processed by a control thread (a reducer). The results are written by each reducer in a *reducer output data* batch (see Figure 5.1).

Multiple mapper threads simultaneously process *input data* and aggregate the results of this processing into multiple *reducer input data* batches (in *Map*). But each *reducer input data* batch is then processed by a single reducer thread

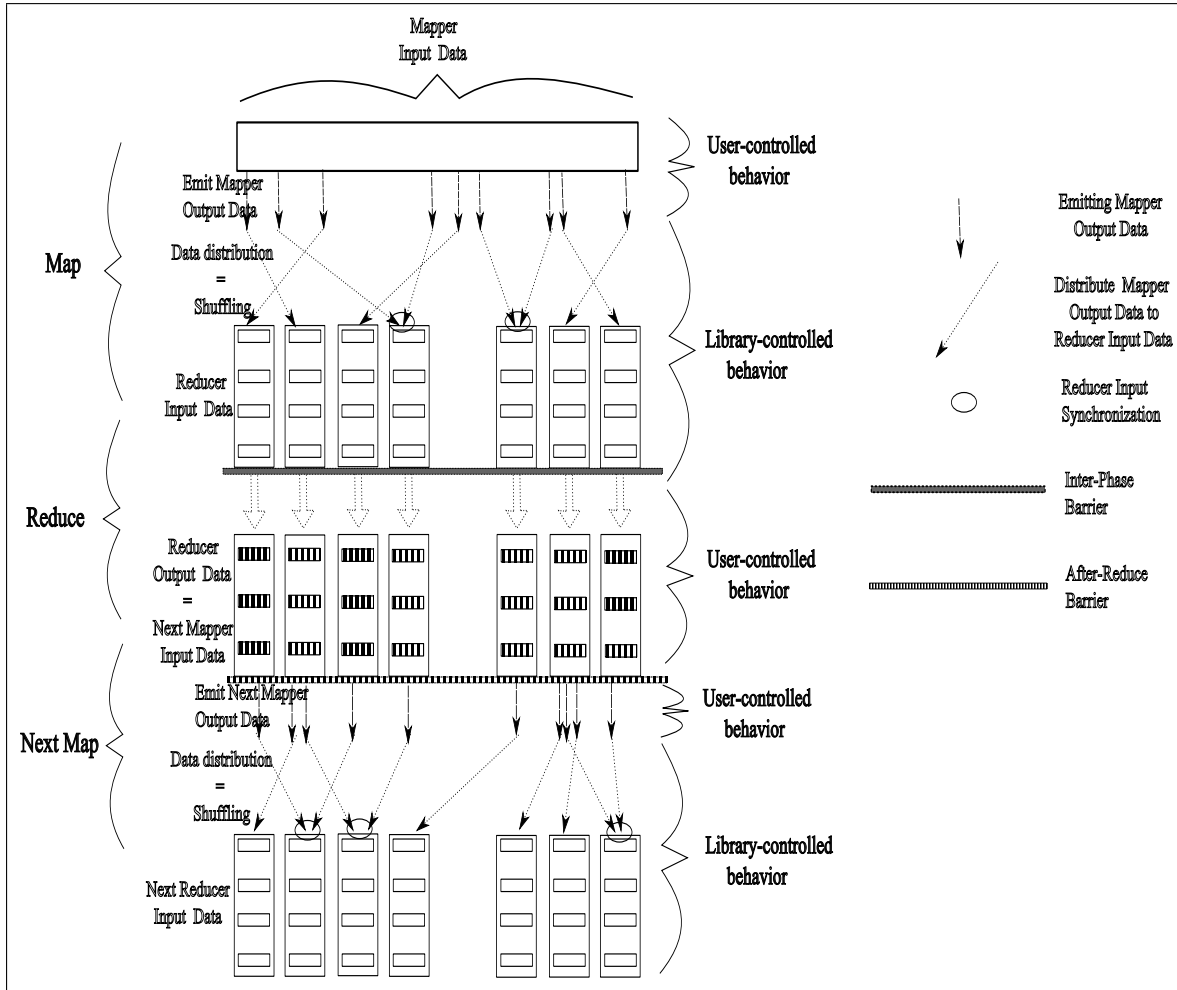


Figure 5.1: The *standard map-reduce* control and data flow, as exhibited by programs written in Google’s MapReduce or in Hadoop. *Map* applies a user-defined operation to each element in the *mapper input data*. The results of the operation are distributed to *reducer input* batches based on problem-specific criteria. In *Reduce*, each control thread (*Reducer*) processes its own *reducer input* batch and saves the results of the processing to its own *reducer output data* batch. In many cases, the *reducer output data* batches that result from *Reduce* become input data for the *Map* phase of the next *map-reduce* round. The *map-reduce* control flow also exhibits synchronization mechanisms: *Reducer Input Synchronization* (exclusive access to a *reducer input data* batch), *Inter-Phase Barrier* (all the mapper control threads must finish the *Map* phase before any reducer control threads starts), and *After-Reduce Barrier*.

(in *Reduce*).

Map-reduce programs use three synchronization mechanisms. Each of them is handled by the library and is not exposed to the *map-reduce* program developer. These synchronization mechanisms are:

- *Reducer Input Synchronization*: A mapper thread may request exclusive access to one of the *reducer input data* batches.
- *Inter-Phase Barrier*: All threads may need to switch between phases at the same time. Mapper threads cannot execute concurrently with Reducer threads.
- *After-Reduce Barrier*: All reducer threads must finish executing before the computation can advance to the next step.

5.2 The Map-Reduce Control Flow in the NUMA-based Algorithms

In this section we show that each of the three NUMA solutions is an extension of *standard map-reduce* control flow. We also discuss relevant extensions to *standard map-reduce* that are implemented by existing libraries. Also, for each of the three NUMA solutions, we briefly present a one-to-one correspondence between components of the solution and components of *map-reduce* (in Section 5.2.4). Appendix A presents a more detailed description of the components of the *map-reduce*-like control flow, as exhibited by each of the proposed algorithms.

5.2.1 The Map-Reduce Control Flow in NFA

Determinization

The parallel disk-based NFA determinization algorithm was presented in Algorithm 2 of Section 2.3. It consists of parallel breadth-first search (BFS) over the sets of NFA-states that make up the intermediate DFA.

In *standard map-reduce*, each BFS iteration would correspond to a *map-reduce* round: scanning the current BFS frontier (along with all previous BFS frontiers) and generating the next BFS frontier corresponds to the *Map* phase; comparing the next BFS frontier with the current and all previous BFS frontiers, and removing duplicate DFA states corresponds to the *Reduce* phase.

In our solution, the set of mappers is identical to the set of reducers: one process per each cluster-node will act both as a mapper and as a reducer, depending on the phase of the computation. This can also be achieved in *standard map-reduce*.

Extensions Used in Our Solution

Basically, the *map-reduce* extensions used by our NFA determinization solution are the caching of previous reducer data (also achievable by using the reducer cache concept in HaLoop [BHBE10]) and the merging of the *Reduce* and *Map* tasks from two consecutive iterations into a single, *Reduce-Map* tasks (this extension is also available in Map-Reduce-Merge [YDHP07]). We next describe how these two extensions are used in our NFA determinization.

Difference from Standard Map-Reduce The first difference is that in our solution, in each BFS iteration the *Reduce* tasks (that perform duplicate detection) are merged with the next *Map* tasks (that generate the neighbors of the current BFS frontier). We call these *Reduce-Map* tasks. In addition,

the output of these *Reduce-Map* tasks is shuffled directly into the input of the next *Reduce-Map* tasks. We can bypass the *Map* phase in each iteration except the first iteration. A separate *Map*-only phase is not needed except for the first iteration. This extension is also implemented by Map-Reduce-Merge [YDHP07]. By contrast, in *standard map-reduce* the *Map* phase cannot be bypassed, even though all it may do is pass input data to the output. This leads to more data being read and written from/to disk in *standard map-reduce*.

The second difference from *standard map-reduce* is that, in each iteration, the *Reduce-Map* tasks only emit the states of the next BFS frontier. These states are shuffled over the network to the next round of *Reduce-Map* tasks. The reducers cache all previously-generated BFS frontiers. Therefore, for duplicate detection reducers read the cached previous frontiers from disk, and read only the next BFS frontier from the output of the last round of *Reduce-Map* tasks. Each reducer reads cached data only from its local disk.

By contrast, in *standard map-reduce*, *Map* tasks need to emit all previous BFS frontiers and the current BFS frontier in addition to the next BFS frontier. In BFS, this leads to extra disk scans over large amounts of data. Also, more data will be shuffled across the network. Most of this data is not actually modified by *Map* tasks. This data is just read by *Map* and emitted unchanged. In this sense, accessing previous BFS frontiers in *Map* is redundant. However, these extra data scans are required, because in *standard map-reduce* reducers can only process data that was emitted by *Map* (aside from global shared data), since they do not cache previously generated data.

This difference is significant for performance: scanning all previous frontiers in each BFS iteration by *Map* tasks represents overhead not intrinsic to the algorithm, but required by *standard map-reduce*-based libraries.

Existing Map-Reduce Extensions Relevant to Our Solution As described, the two differences between our solution’s control flow and *standard map-reduce* control flow are:

- Merging *Reduce* and *Map* into *Reduce-Map*; and
- Caching of previously processed data by the reducers.

Creating *Reduce-Map* tasks is possible even in *standard map-reduce*, but there are no gains in efficiency: the output of the *Reduce-Map* tasks still has to pass through the next round of *Map* tasks on its way to the next *Reduce-Map* tasks. These *Map* tasks would just copy the input data to their output. There is no way to bypass the *Map* operation in *standard map-reduce*.

There exist *map-reduce* library extensions that implement *Reduce-Map* tasks that can communicate directly with other *Reduce-Map* tasks. One such library is Map-Reduce-Merge [YDHP07]. Aside from *Reduce-Map*, Map-Reduce-Merge also provides operations such as *Reduce-Merge* or *Reduce-Merge-Map*. Map-Reduce-Merge also introduces a third API operation in addition to *Map* and *Reduce*: *Merge*. The *Merge* operation can aggregate outputs from *Reduce* operations that produce different types. This extension supports heterogeneous data joins without the pre-homogenization step that is necessary in *standard map-reduce*.

While *standard map-reduce*-based libraries do not cache data from previous *map-reduce* rounds, the HaLoop [BHBE10] library extension does.

HaLoop introduces three caching mechanisms to *map-reduce*:

- the *reducer input cache*, which caches loop invariant reducer inputs, thus reducing the amount of input data scanned by mappers, as well as the amount of mapper output data shuffled over the network;

- the *reducer output cache*, which caches the most recent reducer output data on each disk; and
- the *mapper input cache*, which reduces the amount of non-local data scanned by mappers.

HaLoop also has constructs for detecting when an iteration reaches a fixed point.

Twister [ELZ⁺10] is another *map-reduce* extension for iterative programs. Twister caches data that is read-only for the entire duration of the iteration. The read-only data can be cached on the aggregate disks, or even in the distributed RAM of the cluster, if it fits. The authors of Twister report that this method provided performance gains for various problems in data clustering, machine learning, and computer vision, among others. However, Twister does not cache data that is invariant starting with a certain iteration until the end of the iterative process, as HaLoop does (and as used by our NFA determinization solution).

5.2.2 The Map-Reduce Control Flow in Hybrid CPU-GPU Apply

Recall Formula 3.4.1:

$$r_{i_1 i_2 \dots i_d} = \sum_{\mu=1}^M \sum_{j_1=0}^{2k-1} \sum_{j_2=0}^{2k-1} \dots \sum_{j_d=0}^{2k-1} s_{j_1 j_2 \dots j_d} h_{j_1 i_1}^{(\mu,1)} \times h_{j_2 i_2}^{(\mu,2)} \times \dots \times h_{j_d i_d}^{(\mu,d)}$$

In this formula, r and s are output/input d -dimensional tensors, respectively. s is a tensor from the input MADNESS multiresolution tree, while r is a tensor that will be used to update the input tree. Both r and s are 3-dimensional tensors or higher. The h operators are 2-dimensional tensors.

The hybrid CPU-GPU **Apply** operator is presented in Algorithm 9 of Section 3.4. An application of Formula 3.4.1 is split into three tasks: **preprocess**, **compute** and **postprocess**. The **preprocess** and **postprocess** tasks are data-intensive and are executed only on the CPU. The **compute** task is compute-intensive and it has CPU-only and GPU-only versions.

In the *Map* phase, the many-core CPU executes multiple **preprocess** tasks in parallel. The mapper input data consists of the tensor inputs of these tasks. On each cluster-node, there are as many mapper threads as cores, minus one. (The remaining core is used by a dispatcher thread that will act as a CPU-GPU reducer.) Since there are much fewer mapper threads compared to *mapper inputs*, and the same data-intensive operation needs to be applied to each input, waiting for all *mapper input data* to be processed before sending the *mapper output data* to reducers is inefficient. Therefore, reducers start executing as soon as some of the *mapper outputs* are available. The first set of available *mapper outputs* is called tier 1. Tier k of *mapper outputs* consists of the *mapper outputs* that become available during the execution of the *Reduce* phase for tier $k - 1$. (The *Map* phase of the next tier executes simultaneously with the *Reduce* phase of the current tier.)

Before shuffling, the *mapper output data* is sent to a per-node scheduler that will assign a fraction of the mapper inputs to a hybrid CPU-GPU reducer, and the rest to CPU reducers.

In the *Reduce* phase, each CPU reducer (represented by a CPU core) applies a **compute** task to multiple reducer inputs sequentially and sends each individual result to a **postprocess** task. The CPU-GPU reducer prepares the reducer input data for GPU processing (this processing includes copying all necessary data to a few large pre-allocated, pre-page-locked buffers that are transferred to the GPU), schedules the GPU-only **compute** tasks, transfer the

computation results to the CPU, and launches **postprocess** tasks for each of the results.

Extensions Used in Our Solution

The extensions used by our solution are: different architecture-specific implementations for reducers (also available in the Merge [LCWM08] library), and removing the *After-Reduce Barrier* (also available in MapReduce Online [CCA⁺10]). In addition, we implemented problem-specific features in the CPU-GPU reducer and rearranged the code so that the mappers execute data-intensive code, while the reducers execute compute-intensive code. A more detailed description of the differences between our solution's control flow and the *standard map-reduce* control flow follows.

Difference from Standard Map-Reduce Our solution uses two different kinds of reducers for two different architectures, CPUs and GPUs. The two reducers consume the same type of input, produce the same result for the same input, but they each have implementations suited to the specific architecture. This is a difference from *standard map-reduce*.

In addition, the *After-Reduce Barrier* is not used by either the CPU-only or CPU-GPU reducers, and the individual outputs of the reducers are sent forward to the **postprocess** tasks as soon as they are available. In *standard map-reduce*, the *After-Reduce Barrier* is mandatory.

Our CPU-GPU **Apply** solution could have also been achieved in a *map-reduce*-based library for CPU-GPU architectures. However, the efficiency of our solution is in large part a consequence of the separation of data-intensive and compute-intensive code (which would also have to be similarly implemented in a *map-reduce*-based library) and of the low-level implementation features of the CPU-GPU reducer. The low-level implementation features

are:

- retrieving and adding data from/to a GPU software cache;
- organizing data into pre-allocated, pre-page-locked buffers;
- occupying the GPU fully; and
- making sure that the GPU data does not overflow the GPU RAM.

All these features would have to be implemented in the *map-reduce*-based library as well, since they are not already provided by the library. But customizing an existing library written by other researchers to fit our needs is more difficult than writing our own custom code.

Existing Map-Reduce Extensions Relevant to Our Solution Having different *Reduce* operation implementations for each separate architecture in a hybrid CPU-GPU system is also implemented by Merge [LCWM08], a *Map-Reduce* library extension for CPU-GPU. Merge is built on top of the EXOCHI system for programming on heterogeneous architectures. EXOCHI schedules the multiple parallel tasks on all components of the heterogeneous system. EXOCHI also provides a unified memory view of the CPU memory and GPU memory to Merge. However, the unified memory interface of EXOCHI does not address the aggregation of multiple small inputs into a few large buffers, it does not use the “kind” of the data for scheduling, and it uses on-demand data access. Therefore, for problems in which the GPU access latency and CPU-GPU data transfer are dominant factors (like in **Apply**), the user has to manually implement aggregation mechanisms. A disadvantage of Merge is that all tasks are true functional closures, which means that one data instance cannot be shared between multiple tasks. Even read-only data needs to be replicated for each separate task that accesses it.

Another possibility is to have the same reducer code for both CPU and GPU, and have the library perform automatic code translation. This is the approach taken by MapCG [HCC⁺10]. In MapCG, CUDA is chosen as the unifying language between architectures. Unlike most *map-reduce*-based libraries, MapCG does not sort reducer input data, but organizes it in a GPU-based hash table. This technique is related to the RAM-buckets technique used by Roomy. The MapCG API includes, aside from the *Map* and *Reduce* operations, the *Splitter* operation (which splits *mapper input data* into multiple data tiers), and the *Hash* operation, which hashes a *mapper output key* to an entry in the hash table on the GPU. In order to support a hash table on the GPU, MapCG implements a lightweight GPU memory allocator.

We could have used either Merge of MapCG for the implementation of CPU-GPU **Apply**, but some of the most difficult challenges in CPU-GPU **Apply** are not related to the *map-reduce*-like control flow: separation of data-intensive and compute-intensive code, implementing a GPU software cache, using pre-allocated pre-page-locked buffers, and handling the amount of data that is transferred to the GPU so that it does not overflow its RAM, while also making sure that the GPU cores are fully occupied are all challenges outside of the high-level *map-reduce*-like control flow. Therefore, the use of an existing *map-reduce* library would likely have been more difficult than our from-scratch solution, since the library would have had to be significantly modified and adapted to the algorithm's needs.

One extension used in our solution consists of removing the *After-Reduce* barrier. This extension is also available in the MapReduce Online [CCA⁺10] library for parallel disks. MapReduce Online also provides the option to remove the *Inter-Phase Barrier* (which is also available in [VZC⁺10]).

5.2.3 The Map-Reduce Control Flow in Implicit-Indices Permutation Multiplication

Recall the permutation multiplication operation:

```

X, Y, Z arrays of size N with indices 0...N - 1.
X[i] ∈ {0...N - 1}, ∀i ∈ {0...N - 1}.
for i ∈ {0...N - 1} Z[i] = Y[X[i]].

```

The implicit-indices permutation multiplication algorithm (presented in Section 4.3) follows a *map-reduce*-like control flow. Both mappers and reducers are represented by Linux Pthreads. Both the *mapper input data* batches and *reducer input data* batches are buckets in the X , Y , Z and a temporary *TMP* array.

Map is represented by phase 2 of the algorithm, in which the values of array X are distributed to the buckets of array *TMP*. *Reduce* is represented by phase 3 of the algorithm, in which each reducer thread performs an in-cache partial permutation multiplication.

Extensions Used in Our Solution

Our solution's control flow is similar in many ways to *standard map-reduce*. However, in *Map* we only emit $X[i]$ (as opposed to the $(X[i], i)$ and $(i, Y[i])$ pairs in *standard map-reduce*). Also, in *Reduce* we only scan the X -values and Y -values (as opposed to also scanning their corresponding initial indices in *standard map-reduce*). The extension we use in our solution is the fine control over the placement of the *reducer input data* to make the implementation more cache-efficient. A more detailed description of the differences between our solution and a *standard map-reduce* solution follows.

Differences from Standard Map-Reduce In our solution, each mapper thread owns a sub-bucket in each bucket of array TMP . The sizes of these sub-buckets are computed in phase 1 of the algorithm, via a parallel prefix. In the *Map* phase of our solution, the mapper threads scan the X array, emit $X[i]$ values and shuffle them into the buckets of array TMP .

Our solution is different from *standard map-reduce*. In *standard map-reduce*, both the X and Y arrays have to be scanned in the *Map* phase, and both their values and the corresponding indices have to be emitted. However, our solution does not keep indices explicit in the *reducer input data* (array TMP). This can be done because we deterministically assign an exact location in array TMP to each $X[i]$. The deterministic relation uses the rank of the mapper thread, the offset of the sub-bucket owned by the mapper thread in array TMP , and the deterministic order of scanning the X array by the mapper threads. After the *Reduce* phase, the indices can be recreated by re-scanning the X array in the same deterministic order as scanned initially, and using the same deterministic relation to find each $Z[i]$.

Concretely, in standard *map-reduce*, one way to implement permutation multiplication is:

- *Map* emits key-value pairs $(X[i], i)$ and $(i, Y[i])$ for each i . The i in $(X[i], i)$ is needed so that the algorithm knows the initial index of the value, which will be used again in phase 4 of the algorithm. The $Y[i]$ in $(i, Y[i])$ is used in the *Reduce* operation (phase 3).
- *Reduce* aggregates based on the key. Therefore, the key-value pairs $(X[i], i)$ and $(j, Y[j])$, with $j = X[i]$, are sent to the same reducer. The reduce operation will output key-value pairs $(i, Y[j])$, where $j = X[i]$. The reducer output is $(i, Y[X[i]])$, which represents the result of permu-

tation multiplication.

The above *map-reduce* version of permutation multiplication is not cache-aware. A cache-aware version of permutation multiplication can be written in *standard map-reduce*, as follows:

- *Map* emits key-value pairs
 $(\text{quotient}(X[i]/\text{num_buckets}), (\text{remainder}(X[i]/\text{num_buckets}), i))$ and
 $(\text{quotient}(i/\text{num_buckets}), (\text{remainder}(i/\text{num_buckets}), Y[i]))$ for each i .
 This ensures that each *reducer input* bucket contains the same $X[i]$ and $Y[i]$ values as the *reducer input* buckets in our implicit-indices solution. However, in the *map-reduce* version, the *reducer input* buckets also contain the corresponding array indices for each $X[i]$ value and $Y[i]$ value, which leads to more *mapper input data* being scanned and shuffled compared to our solution.
- Each *reduce input batch* now contains multiple values of X whose values are close to the initial indices of the Y -values also present in the same batch. This increases the cache locality of the *Reduce* operation at the cost of the user having to write the local per-batch index-value matching.

As seen in both versions based on *standard map-reduce*, the indices of X and Y have to be emitted by *Map* and used by *Reduce*. These indices cannot be kept implicit, as in our custom solution. Also, in *standard map-reduce* the user has no control over the exact placement of the data in the *reduce input data* buffer, so a deterministic relation between the mapper location and the exact reducer location in the *reducer input* buffer cannot easily be achieved. This translates into more cache misses incurred by a *standard map-reduce* implementation compared to the implicit-indices algorithm, because it accesses up to twice as much data in the *Map* and *Reduce* phases.

Basically, the implicit-indices algorithm is obtained by modifying the second *standard map-reduce*-based algorithm presented above as follows:

- add an extra scan of array X to compute the sizes and offsets of each mapper thread's sub-bucket within each bucket;
- in *Map* scan only array X , emit its values, and shuffle them to deterministic locations in the sub-buckets (as opposed to *standard map-reduce*, in which both X and Y are scanned, and both their values and indices are emitted as key-value pairs and the exact placement of reducer input data cannot be specified in the absence of sorting); and
- in *Reduce* scan only the values of X and Y entirely in cache (as opposed to *standard map-reduce*, in which the pairs of index-values from both X and Y are scanned).

Existing Map-Reduce Extensions Relevant to Our Solution One of the first *map-reduce*-like libraries for many-core machines was presented in [CKL⁺06]. The authors propose a lightweight multi-threaded *map-reduce*-like engine for machine learning algorithms. This library does not implement specific features for increasing the cache-locality of data.

Phoenix [YRK09] is the state-of-the-art *Map-Reduce* library for many-core machines. The Phoenix runtime library is cache-aware and attempts to keep most of the computation cache-local. Outstanding map and reduce tasks are kept in *locality group task queues*, based on the location of the data used in the task. This minimizes the amount of work performed on non-cache-local data. Phoenix also uses contiguous *keys arrays* and *values arrays*. The pointers to each of the *keys arrays* are stored in a pre-allocated 2-dimensional array. The row index in the 2-D array represents a mapper thread's rank, and the column

index represents a reducer thread's rank. This data layout increases the cache locality of the computation.

Metis [MMK10] is another cache-aware *Map-Reduce* library for many-core machines. In Metis, each mapper thread emits *mapper output data* into a fixed-size mapper-local hash table. Each entry in the hash table is a b+tree. This hybrid data structure was adopted because it exhibits good lookup complexity for a wide range of workloads. To decide what the fixed number of hash table entries will be for the duration of a *map-reduce* computation, Metis runs the *map* phase over a sample of the input data. This is called the *prediction* phase. Based on its results, the Metis library computes the size of the per-mapper hash table. The aggregation of mapper output values that have the same key into batches is achieved via a sorting phase.

Neither Phoenix nor Metis can implement an implicit-indices algorithm, and they are not different in this regard from *standard map-reduce*. The extensions both Phoenix and Metis make to *standard map-reduce* are related to the format of *reducer input data*. So while both Phoenix and Metis implementations of permutation multiplication would likely be more efficient than a *standard map-reduce* implementation, they will still scan up to twice more data than our implicit-indices algorithm in both the *Map* and *Reduce* phases.

Also, in both Phoenix and Metis the user does not have full control over the placement of *reducer input data*. This is normal for a general-purpose library. In our custom solution, we can precisely control the offset of each element in the reducer input data buffer, therefore creating a one-to-one correspondence between the offset of each *X*-array value and the offset of each *TMP*-array value. Also, the overhead intrinsic in any *Map-Reduce* library would likely prove too large in the case of permutation multiplication, which consists of only data accesses and no numeric computation. Our algorithm was designed

to avoid explicit synchronization (locks) and to scan as little data as possible.

These features are difficult to achieve in a high-level general-purpose library, and they would likely need large amounts of customization.

5.2.4 The Map-Reduce Components in the Three NUMA-based Solutions

The Map-Reduce Components in NFA Determinization We present the correspondence between components of *map-reduce*-like control flow and our NFA determinization solution. Mapper and reducer control threads are, in this case, Linux processes. There is one such process per cluster node.

- *Map*: It is only used in the initial BFS iteration. The *mapper input data* consists of the initial DFA state. The *mapper output data* batches consists of DFA states at the second BFS level.
- *Reducer Input Synchronization*: This synchronization mechanism is not used in this case, since there is only one control thread per cluster node.
- *Reduce-Map*: At each non-initial BFS iteration, we use tasks that perform the job of both *Reduce* and the next *Map*. The distinction between reducers and mappers is non-existent at this point. The *reducer input data* batches (containing DFA states at the next BFS level) are compared against previously found DFA states that have been cached in previous *Reduce-Map* steps. The duplicate DFA states are removed. The new states become the current BFS frontier. Each *Reduce-Map* task will next generate the states at the next BFS frontier. These next states are shuffled across the network, and they become the input of the next *Reduce-Map* phase.

- *Inter-Phase Barrier*: In this case, this is the barrier between two consecutive *Reduce-Map* phases. It is implemented by global barriers provided by Roomy. In this case, the global barriers are necessary for the efficiency of the parallel disk-based algorithm, and not for algorithm correctness.
- *After-Reduce Barrier*: This mechanism is not used, because immediately after *Reduce*, the *Reduce-Map* task executes the *Map* part of the task.

The Map-Reduce Components in CPU-GPU Apply For this algorithm, mapper threads are MADNESS worker threads, while reducer threads are dispatcher threads. MADNESS threads are built on top of Linux Pthreads. Multiple such threads will run on the same node in the cluster. There is exactly one dispatcher thread per cluster node. The dispatcher is a CPU thread that drives the GPU, and distributes work to the CPU and the GPU.

Figure 5.2 presents the *map-reduce*-like control flow of our CPU-GPU Apply algorithm.

The CPU-GPU Apply operator exhibits a *map-reduce*-like control flow, as described next (and presented in Figure 5.2):

- *Map*: This algorithm uses tiered *mapper input data*, because the mapper inputs are not available all at once. The `preprocess` tasks represent the *Map* tasks. Their output passes through the local scheduler that distributes data to CPU reducers and a CPU-GPU reducer.

The current tier of *mapper output data*, represented by multi-dimensional tensors and 2-dimensional tensors, along with extra information, is copied to *reduce input data* batches, based on the “kind” of the input data. Each *reduce input data* batch will hold data of the same kind. The “kind” is determined via a combination of framework and user-supplied parameters.

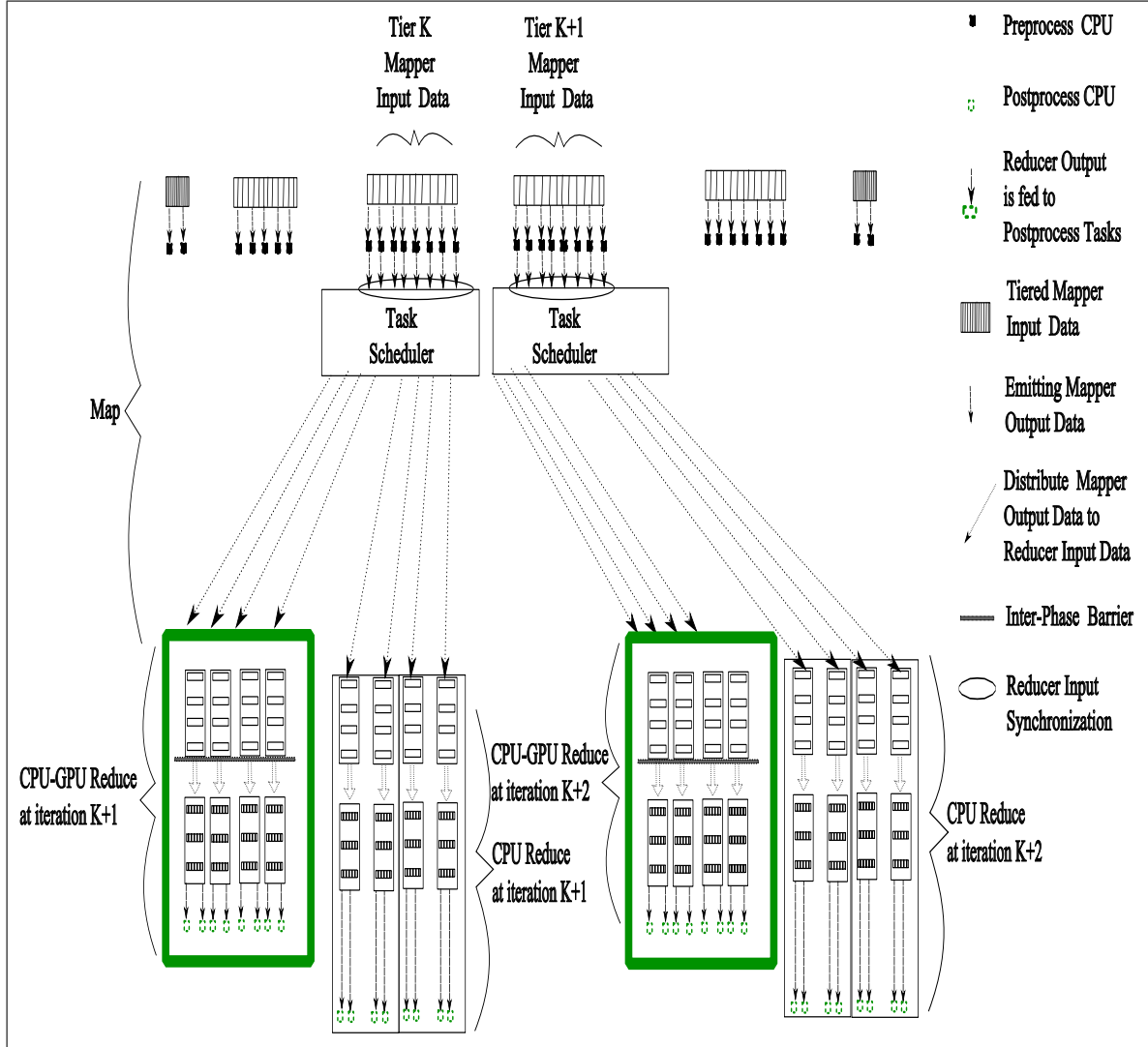


Figure 5.2: Map-reduce-like control flow in the hybrid CPU-GPU Apply algorithm.

- *Reducer Input Synchronization:* Multiple mapper threads write data to the scheduler's buffer. We use a lock to synchronize these multiple writes.
- *Reduce:* There are CPU-only reducers (one per core) and one CPU-GPU reducer. Each CPU reducer sequentially processes multiple independent inputs using `compute`. Once an input has been processed, the result is immediately sent to a `postprocess` task and `compute` is applied to the next input. The CPU-GPU reducer usually processes more inputs than all the CPU reducers combined, because the GPU can do more computation than 16 cores. As part of the CPU-GPU reducer, the reducer inputs are divided into smaller chunks that fit in the GPU memory. Each of the chunks is copied to pre-allocated, pre-page-locked buffers that are then transferred to the GPU. Additionally, the CPU-GPU reducer inspects and manages a GPU software cache for object reuse. This reducer also schedules multiple `compute` tasks simultaneously on the GPU, copies their result to the CPU and send the results to `postprocess` tasks.
- *Inter-Phase Barrier:* This synchronization mechanism is implemented by a barrier that applies only to the processing of elements in a tier.
- *After-Reduce Barrier:* This synchronization mechanism is not used, since it is not needed by the algorithm.
- The *reducer output data* batches are used by `postprocess`, which reorganizes the results for the rest of the computation.

The Map-Reduce Components in Implicit-Indices Permutation Multiplication We present the components of the *Map-Reduce*-like control flow as exhibited by the implicit-indices permutation multiplication algorithm:

- *Map*: The input data is represented by the X array. The *reducer input data* batches are represented by the buckets that the elements of array X are distributed into.
- *Reducer Input Synchronization*: This synchronization mechanism is not used. The initial phase of the algorithm computes the exact size of the sub-buckets. Therefore, multiple threads can simultaneously write to the same *reducer input data* batch (bucket). Each thread has its own area (sub-bucket) within each *reducer input data* batch.
- *Reduce*: The *reducer input data* batches (buckets) are processed by reducer threads asynchronously. This phase performs many small, in-cache permutation multiplications. The *reducer output data* batches consist of the results of the many small, in-cache permutation multiplications.
- *Inter-Phase Barrier* and *After-Reduce Barrier*: The four phases of the algorithm cannot be overlapped. All threads must finish a phase before starting the next step. Therefore, this synchronization mechanism is used via a global barrier.
- After all threads finish the *Reduce* phase, a postprocess step will arrange the contents of the *reducer output data* batches such that they represent the exact result of the overall permutation multiplication.

Our implicit-indices permutation multiplication algorithm avoids *Reducer Input Synchronization*, by computing a sub-bucket for each mapper thread in each bucket of array TMP . Therefore, multiple mapper threads can write values from the X array to the same bucket, but different sub-buckets, without using locks. This increases the parallelism of the algorithm.

5.3 Recursive Algorithms on Parallel Disks

Chapter 2 presents a parallel disk-based subset construction algorithm and a DFA minimization algorithm. The subset construction algorithm for parallel disks is adapted from a sequential, n -ary recursive algorithm. An n -ary recursive function makes n recursive calls to itself. This type of recursion corresponds to a dependency graph in which a sub-problem can be solved by aggregating the results of n sub-sub-problems. The branching factor is n .

The sequential subset construction algorithm can also be viewed as a *dynamic programming* problem. It can be formulated as a reachability problem: finding all the states in the intermediate DFA reachable from the initial state $\{q_0\}$ and all the transition paths that lead to these states, knowing only the NFA. Any sub-problem is formulated as: finding all the states in the intermediate DFA reachable from a given state, knowing only the NFA and the previously discovered states and paths.

Two n -ary recursive, *dynamic programming* problems implemented for parallel disks in the past are the parallel disk-based *apply* algorithm for binary decision diagrams, presented by Kunkle et al. in [KSC10], and the parallel disk-based enumeration of large implicit graphs, presented by Robinson in [Rob08].

The two previously mentioned problems and the subset construction problem are all n -ary recursions and *dynamic programming* instances. This forms the basis of a discussion about the usage of parallel-disks solutions for n -ary recursion instances that are also *dynamic programming* problems. Such problems need to have a small $\frac{\text{graph diameter}}{\text{branching factor}}$ ratio to benefit from a parallel-disks implementation.

Next, we briefly discuss a high-level idea for a methodology for the adaptation of n -ary recursions to parallel disks by using parallel breadth-first search

(PBFS) to explore the directed acyclic graph (DAG) defined by the recursion:

1. The first BFS frontier contains an encoding of the problem.
2. For each sub-problem on the current BFS frontier, generate all n sub-sub-problems and add them to the next frontier.
3. Delayed duplicate detection is applied on the next frontier with respect to all previous frontiers, to see if any problems in the next frontier had already been discovered.
4. While duplicates are eliminated, their parent nodes in the call-graph are updated, so that parents point to the duplicates' canonical representatives in the next frontier.
5. After BFS finishes, each frontier is scanned in bottom-up order.
6. For each node in the current frontier, information from the node's children is aggregated into the parent.
7. The algorithm finishes when the bottom-up scan reaches the root.

Some problems may allow representations in which nodes in the next frontier need not be compared to nodes from the previous frontiers [KSC10]. In these problems, only sub-problems at the same BFS level can be duplicates of one another. The fewer frontiers that need to be checked at any one time during the program execution, the more efficient the algorithm.

This high-level idea for a methodology for the development of parallel disk-based algorithms for recursive problems is based on the components of the *map-reduce* control flow presented in Section 5.1: step 2 in the methodology corresponds to the *Map* phase; step 3 corresponds to the *Reduce* phase; step 4 is a *Postprocess* step; and step 6 corresponds to another application

of the *Map* phase. Note that an actual implementation would consist of repeated applications of steps 2 and 3 for the top-down BFS scan and repeated applications of step 6 for the bottom-up BFS scan.

RAM-based recursive approaches (especially for *dynamic programming* problems) often use an important performance enhancing feature: memoization. Memoization generally means saving the results of already solved sub-problems. This way, if they are encountered again during the computation, their result is readily available. Any sub-problem is only solved once. For divide-and-conquer problems, memoization can reduce the complexity from exponential to polynomial.

The efficiency of memoization lies in the efficiency of in-RAM random access: looking up a stored result is fast. On parallel disks, the same technique is not applicable because of the high latency of a data access. For parallel disk-based programs, this problem is solved by delayed duplicate detection [Kor03, Kor04].

It is worth noting that memoization only helps with detecting sub-problems that were already solved. Memoization cannot detect distinct sub-problems that have the same result.

Detecting equal sub-problems is done by pointer equality, while detecting distinct sub-problems with equal answers is done by deep value equality. For the latter, disk-based parallel programs need to perform an additional scan of the dependency graph, usually bottom-up (step 6 of the disk-based recursion methodology), so that equivalent sub-problems can be detected. When DFS is used in RAM, since problems are solved depth-first, any sub-problems that have the same result can be detected while DFS runs. An instance of a problem which needs deep value equality detection on parallel disks is described in [KSC10].

Table 5.1 shows the general methods and data structures usually employed for various memory and CPU layouts, so that one can recognize what patterns are employed by a given program and what the equivalent patterns would be in a different memory environment.

Table 5.1: Fundamental algorithms, techniques and data structures for algorithms in various environments: RAM-based sequential, RAM-based parallel, disk-based parallel.

Environment	Algorithm	Data structure	Duplicate detection	
			pointer equality	deep equality
RAM-based, sequential	recursion (DFS)	stack	memoization table	immediate (DFS)
	iteration (BFS)	queue		delayed
RAM-based, parallel	recursion (multi-threaded DFS)	per-thread stack	synchronized memoization table	immediate
	iteration (multi-threaded BFS)	per-thread queue		delayed
disk-based, parallel	latency-tolerant distributed BFS	latency-tolerant parallel queue	delayed duplicate detection	delayed, bottom-up scan

5.4 Join-based Operations

Chapter 4 presents efficient solutions for permutation multiplication on many-core machines and parallel disks.

Permutation multiplication is an important operation in computational group theory, especially in the area of permutation groups. More generally, a variant of permutation multiplication applies to object rearrangement:

```
Object Z[N], Y[N]; int X[N]; for  $i \in \{0 \dots N-1\}$  Z[i] = Y[X[i]]
```

The permutation multiplication algorithms presented in Chapter 4 could be adapted to improve the performance of object rearrangement. When the size of an object remains small compared to the size of a disk block, flash block, or cache line, then the presented algorithms can be used on disk, flash, or RAM, respectively. Further, the algorithms described in Chapter 4 could generalize to the case when Y is near to a permutation, but whose values may

include duplicate entries from $\{0 \dots N - 1\}$, while omitting other entries from $\{0 \dots N - 1\}$.

Permutation multiplication is, in fact, an instance of the equi-join database operator. In databases, an equi-join is a join of two tables based on one or multiple equality conditions for values in one column of the first table and one column of the second table. If arrays X and Y are represented by tables with two columns (index and value), then permutation multiplication can be written in an equi-join form as follows:

```
Z = SELECT X.index, Y.value FROM Y, X WHERE Y.index == X.value
```

Note that in permutation multiplication the index and value columns contain unique values.

For the specific equi-join represented by permutation multiplication, the most efficient of the algorithms presented in Chapter 4 is the implicit indices algorithm. However, the implicit indices algorithm relies on the Y table being pre-sorted in index order (which is true for permutation multiplication). If the Y table is not pre-sorted in index order, then the index of each value has to be written explicitly to disk. In these cases, the RAM-buckets algorithm of Chapter 4 can be used with a few modifications: in phase 1, both the X and Y tables are scanned and pairs $(X.value, X.index)$ and $(Y.index, Y.value)$ are distributed to buckets $D_{X.value/Bl}$ and $D_{Y.index/Bl}$, respectively; in phase 2, pairs $(X.value, X.index)$ and $(Y.index, Y.value)$ with $Y.index = X.value$ reside in same source data batch column, so the source data batch column can be loaded into RAM, where the local partial join is performed by random access. In the general equi-join case, there might be multiple $(X.value, Y.index)$ pairs with the same $X.value$ value and multiple $(Y.index, Y.value)$. In this case, a cross product is computed between the set of X pairs with the same value and the set of Y pairs with the same index and $Y.index = X.value$. This general equi-join

algorithm also corresponds to a *map-reduce*-based implementation of the equi-join operator, which is well-known in the literature (see, for example, [OR11]).

5.5 Impact of Research

Based on the evidence presented in Section 5.2, we believe that a *map-reduce*-like solution may be applicable to many other computational science problems that use NUMA architectures. This statement is supported by the fact that each of the three proposed application solutions was designed independently of the other two and the common solution pattern was observed only in hindsight, without it being a goal initially. (Three independent NUMA-based solutions led to the same pattern.)

Therefore, researchers could, in the future, migrate existing solutions from CPU-based architectures to NUMA by using *map-reduce*-like constructs.

The *map-reduce*-like control flow was observed for three applications on three NUMA architectures, but it will likely be used efficiently in other NUMA architectures as well. For example, a port of the MADNESS **Apply** operator to clusters of Intel MIC will face some of the same challenges we did: the high latency of accessing the MIC from the CPU, the low transfer bandwidth between the CPU and the MIC over PCIeExpress, and low occupancy of the up to 80 cores of the MIC. Such a port can use the same solutions as the CPU-GPU port does.

Delayed data access and *delayed task processing* seem to be intrinsic to NUMA-based solutions. The reason is that any on-demand access to non-local memory incurs high latency. Real-world applications usually perform many non-local memory accesses. Therefore, paying the latency price for each such access is inefficient. By batching these multiple accesses, the latency

price is only paid once for each batch (instead of once for each element).

Aside from the common *map-reduce* pattern, the individual solutions of this dissertation could also have an impact on future research.

The parallel disk-based algorithms for finite state automata are applicable to any case in which NFA determinization overflows RAM, but can fit on the aggregate disks of an available cluster. Such large automata problems could occur in speech recognition (as described by Mohri in [Moh97]), lexical analysis, or morphology, amongst other areas.

More generally, as described in Section 5.3, the parallel disk-based NFA determinization algorithm is an instance of an *n-ary* recursion migrated to parallel disks. Previous instances of *n-ary* recursion on parallel disks are [KSC10], and [RC06], amongst others.

Following the guidelines of Section 5.3, more *n-ary* recursions whose call-graph does not fit in RAM can be migrated from the CPU to parallel disks (this includes large *dynamic programming* instances). The ultimate goal in this area would be an automatic translator from sequential CPU-based recursive code to parallel disk-based BFS.

Our MADNESS library extensions can be used to migrate other compute-intensive operators to hybrid CPU-GPU architectures. The next candidate for migration is the differential operator. It would also be worth investigating what are other scientific frameworks that use many small computations instead of a few large computations. The lessons learned from migrating MADNESS to CPU-GPU are likely to also be applicable in those cases.

Our proposed MADNESS library extensions are currently used in a project that implements MADNESS GPU kernels in HMPP. The library extensions and the modified CPU-GPU control flow proposed here are used to perform data aggregation and the separation of data-intensive and compute-intensive

code, while HMPP is used to implement GPU code. HMPP code is more compact, maintainable, and adjustable compared to CUDA code, but it is also less efficient, since it is higher-level. However, for a large project like MADNESS, maintaining complex and portable low-level CUDA code is difficult, because NVIDIA GPU architectures evolve at a rapid pace.

The modified control flow of MADNESS operators for CPU-GPU architectures can also support multiple GPUs attached to the same CPU. We did not have access to such an architecture, but it is likely that CPU-multi-GPU architectures will be common in the future, since GPUs are much more energy-efficient compared to CPUs.

Section 5.4 discusses the applicability of the proposed permutation multiplication algorithms to join database operators. Another area where the proposed permutation multiplication algorithms might have applicability is high-radix Fast Fourier Transform (FFT). FFT algorithms include a permutation multiplication step, which, in the case of radix-2 FFT, is called bit reversal. There exist efficient bit reversal algorithms. However, for higher-radix FFTs, a general permutation multiplication algorithm might be useful.

APPENDIX A

Map-Reduce-like Control Flow in the Proposed NUMA-based Algorithms

A.1 Map-Reduce-like Control Flow in the Parallel Disk-based Subset Construction

The control and data flow of the parallel disk-based subset construction algorithm (Algorithm 2) exhibits a *map-reduce*-like control flow, as follows:

- *Reduce-Map*: The current BFS frontier, containing DFA states at the current BFS level, represents the input data of the current *map-reduce* application round. The DFA states of the current BFS frontier are distributed evenly to all nodes of the cluster, via a Roomy hash function. Each cluster node represents a reducer. Each cluster node processes its own chunk of the current BFS frontier. The processing consists of comparing the current BFS frontier with all previous BFS frontiers and removing duplicates. Next, each reducer computes, for each new state in the current BFS frontier, the neighboring DFA states. Each DFA state is represented as a subset of the NFA states. Each of the produced subsets of states is saved by Roomy in a batch, based on the destination node

that the subset will be sent to (the owner node). Each batch is then sent to the batch owner node. Each node receives batches from all the other nodes, and one batch from itself. This collection of received batches is the next reducer input data. The reducer input data batches contain sets of NFA-states, that represent states in the resulting intermediate DFA.

- *Reducer Input Synchronization*: Each cluster node receives batches of sets of NFA-states from all other cluster nodes. Each received batch is saved to a different file on disk, thus avoiding the need for explicit synchronization.
- *Inter-Phase Barrier*: All cluster nodes must finish a *Reduce-Map* phase before any cluster node starts the next *Reduce-Map* phase. This ensures consistent identification of duplicate states.
- *After-Reduce Barrier*: This mechanism is not used, due to the merge of the *Reduce* and *Map* phases into the *Reduce-Map* phase.

A.2 Map-Reduce-like Control Flow in the Parallel Disk-based DFA Minimization

One iteration of the *forward refinement* algorithm is represented by Algorithm 6. Each such iteration contains three consecutive *map-reduce* rounds.

The first of the three *map-reduce* rounds in an iteration of *forward refinement* is represented by lines 4–11 of Algorithm 6, as follows (the variable names are the ones from Algorithm 6):

- *Map*: The input data of the current *map-reduce* round consists of the current-partition array (array *current_parts* in Algorithm 6). The indices

of the current-partition array represent the DFA states. The values of the current-partition array represent the identifier of the partition that DFA state belongs to. Each cluster node works on the DFA states that it owns (node k works on the batch of states $states^k$). For each local state i , the algorithm accesses the current partition of i and of $init_{DFA}[i][T]$, where T is the current transition. Usually, the state $init_{DFA}[i][T]$ will be owned by another cluster node k_1 . The tuple $(i, curr_parts[i], init_{DFA}[i][T])$ is saved to a local batch, that will be sent to node k_1 (line 6 of Algorithm 6). Each node k_1 receives all batches sent to itself (line 8). These batches represent the *reducer input*.

- *Reducer Input Synchronization*: Explicit synchronization is avoided, since all received batches are saved to separate files on disk.
- *Reduce*: Each node computes, for each received $(i, curr_parts[i], init_{DFA}[i][T])$ tuple, the partition identifier $current_parts[init_{DFA}[i][T]]$ (line 9). A unique identifier is assigned to the pair $(curr_parts[i], curr_parts[init_{DFA}[i][T]])$ (line 11). The reducer output consists of the following data: for each received tuple, the unique identifier of the partition pair, the DFA state i . The reducer output becomes the mapper input data for the second *map-reduce* round in this *forward refinement* iteration.
- *Inter-Phase Barrier and After-Reduce Barrier*: Although these synchronization mechanisms are theoretically not necessary, the usage of Roomy imposes a barrier both between *Map* and *Reduce*, and after *Reduce*.

The second of the three *map-reduce* rounds in an iteration of *forward refinement* is represented by lines 12–16 of Algorithm 6, as follows:

- *Map*: The mapper input data is represented by the reducer output resulted from the first *map-reduce* round in this iteration. Each cluster node sends the data of each tuple and the corresponding unique identifier to the owner of *pair[i]* (line 12). Upon arrival, this data represents the *reducer input data* (line 13).
- *Reducer Input Synchronization*: Explicit synchronization is avoided, since all received batches are saved to separate files on disk.
- *Reduce*: Delayed duplicate detection is performed (lines 14–16) on the pairs of partitions. Each new pair and its corresponding identifier is stored in the local *parts* hash table. For each new pair, the original *i* DFA state and the pair’s identifier are stored in reducer output data. For each old pair, the original *i* DFA state and the pair’s canonical identifier are stored in reducer output data. The reducer output data become the input data for the next *map-reduce* round.
- *Inter-Phase Barrier* and *After-Reduce Barrier*: These synchronization mechanisms are imposed by the usage of Roomy. They are not needed theoretically.

The third of the three rounds of *map-reduce*-like control flow completes one *forward refinement* iteration. *Map* is represented by lines 17, 18 and 22 of Algorithm 6. *Reduce* is represented by lines 23–24.

A.3 Map-Reduce-like Control Flow in the Hybrid CPU-GPU Apply Operator

The **Apply** operator consists of multiple invocations of Formula 3.4.1:

$$r_{i_1 i_2 \dots i_d} = \sum_{\mu=1}^M \sum_{j_1=0}^{2k-1} \sum_{j_2=0}^{2k-1} \cdots \sum_{j_d=0}^{2k-1} s_{j_1 j_2 \dots j_d} h_{j_1 i_1}^{(\mu,1)} \times h_{j_2 i_2}^{(\mu,2)} \times \cdots \times h_{j_d i_d}^{(\mu,d)}$$

In this formula, r and s are output/input d -dimensional tensors, respectively. s is a tensor from the input MADNESS multiresolution tree, while r is a tensor that will be used to update the input tree. Both r and s are 3-dimensional tensors or higher. The h operators are 2-dimensional tensors.

The CPU-GPU **Apply** operator exhibits *map-reduce*-like control flow, as follows:

- *Map*: Worker threads (mappers) on each compute node are executing the *preprocess* part of apply (the *integral_preprocess* function of Algorithm 9). The input data is represented by all the tensors contained in the nodes of a multiresolution tree. Each thread works on one tensor (s) at a time. Each worker thread computes a neighbor node in the multiresolution tree. The neighbor tensor will accumulate results from its neighboring tensors. During *preprocess*, each worker thread also computes or locates in the CPU RAM all the h 2-D tensors that will be used in a convolution product with the s higher-dimensional tensor. The s tensor, the h tensors, and the neighbor node, along with extra information, are copied to source data batches, based on the “kind” of the input data. Each source data batch will hold data of the same kind. The “kind” is determined by a combination of the *compute* function pointer and a user-supplied hash function. This ensures that the right *compute* function will be used to process the right *reducer input* batch. More-

over, this leads to more efficiently utilizing the GPU : the simultaneous execution of multiple tasks of the same “kind” should result in the tasks finishing executing at about the same time (there is some variability, though, that is a consequence of the values in the h tensors of each task). Using the input data “kind” also helps the static work distribution between the CPU and the GPU, since the relative execution time between tasks of the same “kind” is low.

- *Reducer Input Synchronization:* Mappers on the same cluster node use an explicit lock to synchronize when writing the s tensor, the h tensors and the neighbor node to the same source data batch.
- *Reduce:* The *reducer input* data batches are processed by a dispatcher thread per node and multiple other worker threads (reducers). There is one dispatcher thread per cluster node. Each dispatcher thread processes the local *reducer input* data batches one by one. Some elements in the *reducer input* batches are processed by a GPU version of *compute* (function `integral_GPU_compute` of Algorithm 10), while others by a CPU version of *compute* (function `integral_CPU_compute` of Algorithm 10). Formula 3.4.1 is applied in this phase to each element in each *reducer input* data batch. The results of the formula are placed in *reducer output* batches.
- *Inter-Phase Barrier:* This synchronization mechanism is used to separate the *Map* and *Reduce* phases for the same data tier.
- *After-Reduce Barrier:* This barrier is not needed by the algorithm and it is not used. As soon as an element of the *reducer output* is ready, it gets sent to a `postprocess` task.

A.4 Map-Reduce-like Control Flow in the Permutation Multiplication Algorithms

Recall the permutation multiplication operation:

```

X, Y, Z arrays of size N with indices 0...N - 1.
X[i] ∈ {0...N - 1}, ∀i ∈ {0...N - 1}.
for i ∈ {0...N - 1} Z[i] = Y[X[i]].

```

The multi-threaded RAM-based permutation multiplication algorithm (presented in Algorithm 14), as well as the three disk-based algorithms (Algorithms 11, 12 and 13) and their parallel disk-based counterparts all exhibit a *map-reduce*-like control flow.

Permutation Multiplication in RAM Here, batches are buckets (represented as files on disk) and control threads are Linux Pthreads.

The multi-threaded RAM-based permutation multiplication algorithm exhibits *map-reduce*-like control flow, as follows:

- *Preprocess*: Each thread will potentially write to each bucket. In order to avoid the performance penalty incurred by explicit bucket locking, the threads collaborate to compute, for each (thread, bucket) pair, the number of elements the thread will write in the bucket, and the offset at which the thread will start writing in the bucket. Thus, each thread will own a sub-bucket into each bucket. The sub-buckets do not intersect and they cover the entire bucket. This preprocessing phase is implemented by a parallel prefix algorithm. Preprocessing is presented in phase 1 of Algorithm 14. It contributes significantly to the efficiency of the *Map* phase of the algorithm. A temporary array D is used in addition to the

X , Y , and Z arrays.

- *Map*: The input data is represented by the X array. Each thread processes a contiguous chunk (bucket) of array X . All contiguous chunks have similar size. Each thread scans its own bucket of X and, for each number $X[i]$, determines the bucket of array D that $X[i]$ is distributed to. The thread will actually write the $X[i]$ value to the next available slot in the sub-bucket that it owns within the bucket. The *reducer input data* batches are represented by the buckets that the elements of array X are distributed into, in the manner presented above. The *Map* phase is represented by phase 2 of Algorithm 14.
- *Reducer Input Synchronization*: This synchronization mechanism is not used. The preprocessing phase computes the exact size of the sub-buckets. Each thread has its own area (sub-bucket) within each source data batch. Therefore, multiple threads can simultaneously write to the same source data batch (bucket) without explicitly synchronizing.
- *Reduce*: The buckets of arrays TMP and Y are processed by threads asynchronously. Each thread loads a bucket of array D and the corresponding bucket of array Y in L2 cache. Each thread performs in-cache permutation multiplication using the D -array bucket and the Y -array bucket. *Reduce* is represented by phase 3 of Algorithm 14. The *reducer output data* batches produced by each thread consists of the result of the small, in-cache permutation multiplication performed by the thread entirely in L2 cache.
- *Inter-Phase Barrier*: The *Map* and *Reduce* phases cannot overlap in this case. A barrier must ensure that all information has been written to array D before the in-cache permutations (*Reduce*) are performed.

- *Postprocess*: After all threads finish the *Reduce* phase, a postprocess step arranges the contents of the *reducer output data* batches such that they represent the exact result of the overall permutation multiplication. The postprocess is performed by phase 4 of Algorithm 14.

Permutation Multiplication on Parallel Disks Here, both *mapper input* batches and *reducer input* batches are buckets represented by files on disk. Both mappers and reducers are processes on the cluster nodes. There is one such process per cluster node. In all cases, the data array X is evenly distributed to the disk of each cluster node (in contiguous batches named buckets).

The external sorting algorithm exhibits control flow consisting of two *map-reduce* application rounds, as described next.

The first *map-reduce* round is:

- *Map*: Input data is represented by array X . Each cluster node scans its own bucket of array X . Each node saves pairs of integers $(i, X[i])$ to local batches. These local batches are sent to their owner nodes based on the value $X[i]$. The batches received by each node from all other nodes represent the *reducer input data* batches.
- *Reducer Input Synchronization*: Synchronization is not needed, since each received batch is saved to a separate file.
- *Reduce*: The pairs $(i, X[i])$ are sorted by $X[i]$ and saved in the sorted order into array D . The D array represents the *reducer output data*.
- *Inter-Phase Barrier and After-Reduce Barrier*: A barrier between the *Map* and *Reduce* phases is necessary, as well as a barrier after the *Reduce* phase. External sorting needs to be performed on the entire data.

The second *map-reduce* round is:

- *Map*: *Mapper input data* is represented by array D (the destination data batches resulted from the first *map-reduce* round). Each cluster node scans its own buckets of array D and Y . Pairs $(i, Y[j])$ (such that $D[j] = (i, X[i])$) are saved to local batches, which are then sent to their owner nodes based on i . The received batches represent the *reducer input data*.
- *Reducer Input Synchronization*: This synchronization mechanism is not needed, since each received batch is saved to a separate file.
- *Reduce*: The pairs $(i, Y[j])$ are sorted by i and saved in the sorted order into array D' . The D' array represents the *reducer output data* batches (and the result of the permutation multiplication).
- *Inter-Phase Barrier* and *After-Reduce Barrier*: A barrier between the *Map* and *Reduce* phases is necessary, as well as a barrier after the *Reduce* phase. External sorting needs to be performed on the entire data.

The RAM-buckets algorithm exhibits *map-reduce*-like control flow as follows:

- *Map*: The X buckets are the *mapper input data*. Each cluster node scans its own X bucket and saves each pair $(i, X[i])$ to a local batch. Each local batch will be sent to the node that owns bucket $X[i]/Bl$ of the arrays. Upon arrival to the destination node, the pair $(i, X[i])$ is saved to the local D -array batch. The D -array batches represent the *reducer input data batches*.
- *Reducer Input Synchronization*: Synchronization is not needed, since each received batch is saved to a separate file.

- *Reduce*: Each cluster node loads its own D -array bucket and its own Y -array bucket into RAM. Local in-RAM permutation multiplication is performed. The results are saved locally to D' -array buckets.
- *Inter-Phase Barrier* and *After-Reduce Barrier*: These synchronization mechanisms are performed with a global barrier.
- *Postprocess*: The permutation multiplication results (which are in array D') are sent to their owner nodes and arranged by index order.

The implicit indices algorithm exhibits *map-reduce*-like control flow similarly to the RAM-buckets algorithm, with the difference that no indices are saved explicitly to disk. Throughout the implicit indices algorithm, any index is implicitly deduced from the MPI rank of the node that sent the batch and the offset within the batch. With all indices being implicit, there is up to half less data that needs to be written to disk, thus making the algorithm up to twice as fast as the RAM-buckets algorithm. More specifically, the implicit indices algorithm as follows exhibits a *map-reduce*-like control flow, as follows:

- *Map*: The X buckets are the *mapper input data*. Each cluster node scans its own X bucket and saves each $X[i]$ value to a local batch. There are N/Bl such local batches. (N is the length of X , while Bl is the static array-bucket length.) Local batch $X[i]/Bl$ of values of array X is sent to the cluster node with MPI rank $X[i]/Bl$. Upon arrival to the destination node, $X[i]$ is saved to the local D -array batch. The D -array batch consists of appended X -values batches received from all cluster nodes. The append is performed in the order of the MPI ranks of the nodes that sent the X – *values* batches. The D -array batches represent the *reducer input data* batches.

- *Reducer Input Synchronization:* Synchronization is not needed, since each received batch is saved to a separate file.
- *Reduce:* Each cluster node loads its own D -array bucket and its own Y -array bucket into RAM. Local in-RAM permutation multiplication is performed. Note that neither the indices of the X -array values in D , nor the indices of the Y -array values are represented explicitly. The local partial $Y[X[i]]$ permutation uses the implicit indices of array Y . The implicit index of a value in the local bucket of array Y is calculated by: $(Node\ MPI\ Rank * Bl + offset\ of\ value\ in\ bucket)$. The local bucket of array D does not, by itself, have a way of determining implicit indices, but they are not needed in this phase. The results are saved locally to D' -array buckets. If value $X[i]$ is saved at a certain *offset* in the file that represents the local D bucket, then value $Y[X[i]]$ is saved at the same *offset* in the local D' bucket.
- *Inter-Phase Barrier and After-Reduce Barrier:* These synchronization mechanisms are performed with a global barrier.
- *Postprocess:* Implicit indices are calculated by re-scanning array X and determining the MPI rank of the node and the offsets in the D bucket on that node that value $X[i]$ was sent to. Following from *Reduce*, the value $Y[X[i]]$ is on the same node and at the same offset, but in the D' bucket. The permutation multiplication results can be arranged in the i index order.

Bibliography

- [AAR03] M. H. Albert, M. D. Atkinson, and N. Ruškuc. Regular closed sets of permutations. *Theoret. Comput. Sci.*, 306(1-3):85–100, 2003.
- [ABC⁺09] Eduard Ayguade, Rosa M. Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez, Jesus Labarta, Xavier Martorell, Rafael Mayo, Josep M. Perez, and Enrique S. Quintana-Ortí. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag.
- [AL04] M. H. Albert and Steve Linton. A practical algorithm for reducing non-deterministic finite state automata. Technical report, 2004. <http://www.cs.otago.ac.nz/research/publications/oucs-2004-11.pdf>.
- [ALT97] M. D. Atkinson, M. J. Livesey, and D. Tulley. Permutations generated by token passing in graphs. *Theor. Comput. Sci.*, 178:103–118, May 1997.
- [AMB] AMBER. <http://ambermd.org/>.

- [AS99] M.D. Atkinson and J.-R. Sack. Pop-stacks in parallel. *Information Processing Letters*, 70(2):63 – 67, 1999.
- [ATNW11] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.
- [BC05] Jean Berstel and Olivier Carton. On the complexity of Hopcroft’s state minimization algorithm. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *Implementation and Application of Automata*, volume 3317 of *Lecture Notes in Computer Science*, pages 35–44. Springer Berlin / Heidelberg, 2005.
- [BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, Sept. 2010.
- [Big] BigDFT. http://inac.cea.fr/L_Sim/BigDFT.
- [BJK⁺96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37:55–69, August 1996.
- [BM08] Henrik Björklund and Wim Martens. The tractability frontier for NFA minimization. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP ’08, pages 27–38, Berlin, Heidelberg, 2008. Springer-Verlag.

-
- [Bon03] Miklos Bona. A survey of stack-sorting disciplines. *Electron. J. Combin.*, 9:1, 2003.
- [CCA⁺10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.
- [CDHW73] John J. Cannon, Lucian A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.
- [CH97] G. Cooperman and G. Havas. Practical parallel coset enumeration. In *Proc. of Workshop on High Performance Computation and Gigabit Local Area Networks*, volume 226 of *Lecture notes in control and information sciences*, pages 15–27. Springer Verlag, 1997.
- [CKL⁺06] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.
- [CKP02] J.-M. Champarnaud, A. Khorsi, and T. Paranthou. Split and join for minimizing: Brzozowski's algorithm. In *Proceedings of PSC 2002 (Prague Stringology Conference)*, pages 96–104, 2002.

- [CM02] G. Cooperman and X. Ma. Overcoming the memory wall in symbolic algebra: A faster permutation algorithm (formally reviewed communication). *SIGSAM Bulletin*, 36:1–4, December 2002.
- [CR03] Gene Cooperman and Eric Robinson. Memory-based and disk-based algorithms for very high degree permutation groups. In *ISSAC*, pages 66–73, 2003.
- [CSK08] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.
- [CWSB10] Matthew L. Curry, H. Lee Ward, Anthony Skjellum, and Ron Brightwell. A lightweight, GPU-based software RAID system. In *ICPP*, pages 565–572, 2010.
- [DBB] Romain Dolbeau, Stephane Bihan, and Francois Bodin. HMPP: A hybrid multi-core parallel programming environment. Technical report.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [DKT11] Usman Dastgeer, Christoph Kessler, and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. In *International conference on Parallel Computing (ParCo)*, Gent, Belgium, August 2011.

-
- [DLM] Manuel Delgado, Steve Linton, and Jose Morais. GAP Automata Package. <http://www.gap-system.org/Packages/automata.html>.
- [DY08] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.
- [ELcFS11] M. Elteir, Heshan Lin, Wu chun Feng, and T. Scogland. StreamMR: An optimized MapReduce framework for AMD GPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 364 –371, dec. 2011.
- [Eld] Problems and Conjectures presented at the Third International Conference on Permutation Patterns. <http://www.math.ufl.edu/~vatter/publications/pp2005-problems/pp-problems.pdf>.
- [Eld06] Murray Elder. Permutations generated by a stack of depth 2 and an infinite stack in series. *Electron. J. Combin.*, 13(1):Research Paper 68, 12 pp. (electronic), 2006.
- [Eli] Sergi Elizalde. The X -class and almost-increasing permutations. <http://arxiv.org/abs/0710.5168>.
- [ELZ⁺10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM*

- International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.
- [FBHJ04] G. Fann, G. Beylkin, R. J. Harrison, and K. E. Jordan. Singular operators in multiwavelet bases. *IBM Journal of Research and Development*, 48(2):161–171, march 2004.
- [Fel61] H. Felsch. Programmierung der Restklassenabzählung einer Gruppe nach Untergruppen. *Numerische Mathematik*, 3:250–256, 1961.
- [Fog] Agner Fog. Subroutine library. <http://www.agner.org/optimize/#asmlib>.
- [FPH⁺09] G I Fann, J Pei, R J Harrison, J Jia, J Hill, M Ou, W Nazarewicz, W A Shelton, and N Schunck. Fast multiresolution methods for density functional theory in nuclear physics. *Journal of Physics: Conference Series*, 180(1):012080, 2009.
- [GAM] GAMESS. <http://www.msg.ameslab.gov/gamess/>.
- [GAP08] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008.
- [GP08a] Rob Glabbeek and Bas Ploeger. Correcting a space-efficient simulation algorithm. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 517–529, Berlin, Heidelberg, 2008. Springer-Verlag.
- [GP08b] Rob Glabbeek and Bas Ploeger. Five determinisation algorithms. In *Proceedings of the 13th international conference on Implemen-*

-
- tation and Applications of Automata*, CIAA '08, pages 161–170, Berlin, Heidelberg, 2008. Springer-Verlag.
- [GPP03] R. Gentilini, C. Piazza, and A. Policriti. From bisimulation to simulation: Coarsest partition problems. *J. Autom. Reason.*, 31(1):73–103, October 2003.
- [Gri73] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973. 10.1007/BF00264025.
- [GRO] GROMACS. <http://www.gromacs.org/>.
- [GSBS11] Max Grossman, Alina Simion Sbîrlea, Zoran Budimlić, and Vivek Sarkar. CnC-CUDA: declarative programming for GPUs. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 230–245, Berlin, Heidelberg, 2011. Springer-Verlag.
- [Had] Hadoop MapReduce. <http://hadoop.apache.org/mapreduce/>.
- [Har] Robert J. Harrison. Accelerating past the petascale. A case study of GPGPUs in chemistry. http://www.csm.ornl.gov/workshops/SOS14/documents/harrison_pres.pdf.
- [HCC⁺10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.
- [HF] Robert J. Harrison and George I. Fann. MADNESS: Multiresolution Adaptive Numerical Environment for Scientific Simulation.

- [HF07] Robert J. Harrison and George I. Fann. Speed and Precision in Quantum Chemistry. 2007. <http://www.scidacreview.org/0701/pdf/chemistry.pdf>.
- [HFL⁺08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.
- [HFY⁺04] R. J. Harrison, G. I. Fann, T. Yanai, Z. Gan, and G. Beylkin. Multiresolution Quantum Chemistry: Basic Theory and Initial Applications. *Journal of Chemical Physics*, 121:11587–11598, 2004.
- [HFYB03] Robert J. Harrison, George I. Fann, Takeshi Yanai, and Gregory Beylkin. Multiresolution quantum chemistry in multiwavelet bases. In *Proceedings of the 2003 international conference on Computational science*, ICCS'03, pages 103–110, Berlin, Heidelberg, 2003. Springer-Verlag.
- [HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [Hop71] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford University, Stanford, CA, USA, 1971.
- [HS99] G. Havas and C. Sims. A presentation for the Lyons simple group. In *Computational Methods for Representations of Groups and Al-*

-
- gebras*, volume 173 of *Progress in Mathematics*, pages 241–249, 1999.
- [HSW01] G. Havas, L.H. Soicher, and R.A. Wilson. A presentation for the Thompson sporadic simple group. In *Groups and Computation III, Computational Methods for Representations of Groups and Algebras*, volume 8 of *Ohio State University Mathematical Research Institute Publications*, pages 193–200. de Gruyter, 2001.
- [Huf54] D.A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161 – 190, 1954.
- [ICP] Permutation patterns 2012. <http://combinatorics.cis.strath.ac.uk/pp2012/>.
- [Int] Intel. Intel Cilk Plus. <http://software.intel.com/en-us/intel-cilk-plus-archive>.
- [IY03] Lucian Ilie and Sheng Yu. Reducing NFAs by invariant equivalences. *Theor. Comput. Sci.*, 306(1-3):373–390, September 2003.
- [JR93] Joseph F. JáJá and Kwan Woo Ryu. An efficient parallel algorithm for the single function coarsest partition problem. In *Proceedings of the fifth annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 230–239, New York, NY, USA, 1993. ACM.
- [KC09] Daniel Kunkle and Gene Cooperman. Harnessing parallel disks to solve Rubik’s cube. *Journal of Symbolic Computation*, 44:872–890, 2009.

- [Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
- [Kno09] Kathleen Knobe. Ease of use with concurrent collections (CnC). In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar’09, pages 17–17, Berkeley, CA, USA, 2009. USENIX Association.
- [Knu68] Donald E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968.
- [Knu01] Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.*, 250:333–363, January 2001.
- [Kor03] Richard E. Korf. Delayed duplicate detection: extended abstract. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1539–1541, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.
- [Kor04] Richard E. Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the 19th National Conference on Artificial Intelligence*, AAAI’04, pages 650–657. AAAI Press, 2004.
- [Kos96] Kimmo Koskenniemi. Finite state morphology and information retrieval. *Nat. Lang. Eng.*, 2:331–336, December 1996.
- [KRSG95] L.V. Kale, B. Ramkumar, A. B. Sinha, and A. Gürsoy. The Charm parallel programming language and system: Part I - description of language features, 1995.

-
- [KSC10] Daniel Kunkle, Vlad Slavici, and Gene Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 63–72, New York, NY, USA, 2010. ACM.
- [Kun10] Daniel Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. <http://roomy.sourceforge.net/>.
- [LAM] LAMMPS. <http://lammps.sandia.gov/>.
- [LCWM08] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM.
- [LRV10] Steve Linton, Nik Ruškuc, and Vincent Vatter. *Permutation Patterns*. Cambridge University Press, 2010.
- [MAB⁺10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
- [MMK10] Yandong Mao, Robert Morris, and M. Frans Kaashoek. Optimizing MapReduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

- [Moh94] Mehryar Mohri. Syntactic analysis by local grammars automata: an efficient algorithm. In *Proceedings of the International Conference on Computational Lexicography*, COMPLEX '94. Linguistic Institute, Hungarian Academy of Science, 1994.
- [Moh96] Mehryar Mohri. On some applications of finite-state automata theory to natural language processing. *Nat. Lang. Eng.*, 2:61–80, March 1996.
- [Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23:269–311, June 1997.
- [Moo56] Edward F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.
- [MPR02] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69 – 88, 2002.
- [NAM] NAMD. <http://www.ks.uiuc.edu/Research/namd/>.
- [Neu82] J. Neubüser. An elementary introduction to coset table methods in computational group theory. In C.M. Campbell and E.F. Robertson, editors, *Groups – St Andrews 1981*, volume 71 of *London Math. Soc. Lecture Note Ser.*, pages 1–45, Cambridge, 1982. Cambridge University Press.
- [NPT⁺06] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20:203–231, May 2006.

-
- [NVIA] NVIDIA. New GPU applications accelerate search for more effective medicines and higher quality materials. http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09\&version=live\&prid=820175\&releasejsp=release_157\&xhtml=true.
- [NVib] NVIDIA. NVIDIA CUDA C Programming Guide. Technical report. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.
- [Opea] OpenACC. <http://www.openacc-standard.org/>.
- [Opeb] OpenMP. <http://openmp.org/wp>.
- [OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.
- [PRS94] Fernando Pereira, Michael Riley, and Richard Sproat. Weighted rational transductions and their application to human language processing. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 262–267, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.
- [QMC] QMCPACK. <http://code.google.com/p/qmcpack/>.
- [RC06] Eric Robinson and Gene Cooperman. A parallel architecture for disk-based computing over the Baby Monster and other large finite simple groups. In *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, ISSAC '06, pages 298–305, New York, NY, USA, 2006. ACM.

- [RHH12] Matthew G. Reuter, Judith C. Hill, and Robert J. Harrison. Solving PDEs in irregular geometries with multiresolution methods I: Embedded Dirichlet boundary conditions. *Computer Physics Communications*, 183(1):1 – 7, 2012.
- [Rob08] Eric Robinson. *Large implicit state space enumeration: overcoming memory and disk limitations*. PhD thesis, Boston, MA, USA, 2008. Adviser-Cooperman, Gene.
- [Roc96] Emmanuel Roche. Transducer parsing of free and frozen sentences. *Nat. Lang. Eng.*, 2:345–350, December 1996.
- [RX96] B. Ravikumar and X. Xiong. A parallel algorithm for minimization of finite automata. *Parallel Processing Symposium, International*, 0:187, 1996.
- [S⁺12] W.A. Stein et al. *Sage Mathematics Software (Version 5.0.1)*. The Sage Development Team, 2012. <http://www.sagemath.org>.
- [SDKC10] Vlad Slavici, Xin Dong, Daniel Kunkle, and Gene Cooperman. Fast multiplication of large permutations for disk, flash memory and RAM. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 355–362, New York, NY, USA, 2010. ACM.
- [SHM⁺11] Kevin Stock, Tom Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert Harrison. Model-driven SIMD code generation for a multi-resolution tensor kernel. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1058–1067, Washington, DC, USA, 2011. IEEE Computer Society.

-
- [Sil94] Max D. Silberztein. INTEX: a corpus processing system. In *Proceedings of the 15th Conference on Computational Linguistics - Volume 1*, COLING '94, pages 579–583, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.
- [SJ] Gregory Scott Jones. MADNESS named R&D 100 winner. <http://www.olcf.ornl.gov/2011/06/29/madness-named-rd-100-winner>.
- [SKCL11] Vlad Slavici, Daniel Kunkle, Gene Cooperman, and Stephen Linton. Finding the minimal DFA of very large finite state automata with an application to token passing networks. Technical report, 2011. <http://arxiv.org/abs/1103.5736v1>.
- [SKCL12] Vlad Slavici, Daniel Kunkle, Gene Cooperman, and Stephen Linton. An efficient programming model for memory-intensive recursive algorithms using parallel disks. In *Proceedings of the 2012 International Symposium on Symbolic and Algebraic Computation*, ISSAC '12, New York, NY, USA, 2012. ACM. to appear.
- [SKL88] Mark Swanson, Robert Kessler, and Gary Lindstrom. An implementation of portable standard LISP on the BBN butterfly. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, LFP '88, pages 132–142, New York, NY, USA, 1988. ACM.
- [SMYH08] H. Sekino, Y. Maeda, T. Yanai, and R.J. Harrison. Basis set limit Hartree-Fock and density functional theory response property evaluation by multiresolution multiwavelet basis. *J Chem Phys*, 129(3):034111, 2008.

- [SO11] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11*, pages 1068–1079, Washington, DC, USA, 2011. IEEE Computer Society.
- [Spr95] Richard Sproat. A finite-state architecture for tokenization and grapheme-to-phoneme conversion for multilingual text analysis. In *Proceedings of the EACL SIGDAT Workshop*, pages 65–72. Association for Computational Linguistics, 1995.
- [SV09] Rebecca Smith and Vincent Vatter. The enumeration of permutations sortable by pop stacks in parallel. *Inf. Process. Lett.*, 109(12):626–629, May 2009.
- [SVCH12] Vlad Slavici, Raghu Varier, Gene Cooperman, and Robert Harrison. Adapting Irregular Computations to Large CPU-GPU Clusters in the MADNESS Framework. In *Proceedings of the 2012 IEEE Cluster Conference, IEEE Cluster '12*. IEEE Computer Society, 2012.
- [TC36] J.A. Todd and H.S.M. Coxeter. A practical method for enumerating cosets of a finite abstract group. *Proc. Edinburgh Math. Soc., II. Ser. 5*, 5:26–34, 1936.
- [Tena] Bridget Tenner. Database of permutation pattern avoidance. <http://math.depaul.edu/bridget/patterns.html>.
- [Tenb] Bridget Eileen Tenner. Repetition in reduced decompositions. <http://arxiv.org/abs/1106.2839v2>.
- [Ter] TeraChem. <http://www.petachem.com/products.html>.

-
- [Tho11] William Scott Thornton. *Electronic Excitations in YTiO₃ using TDDFT and electronic structure using a multiresolution framework*. PhD thesis, 2011. http://trace.tennessee.edu/utk_graddiss/1134.
- [Topa] Top 500 Green Supercomputers. <http://www.green500.org/lists/2011/11/top/list.php/>.
- [Topb] Top 500 Supercomputers. <http://www.top500.org/>.
- [TSG02] Ambuj Tewari, Utkarsh Srivastava, and P. Gupta. A parallel DFA minimization algorithm. In *Proceedings of the 9th International Conference on High Performance Computing, HiPC '02*, pages 34–40, London, UK, 2002. Springer-Verlag.
- [TV05] Deian Tabakov and Moshe Vardi. Experimental evaluation of classical automata constructions. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *Lecture Notes in Computer Science*, pages 396–411. Springer Berlin / Heidelberg, 2005.
- [VHKac12] Nicholas Vence, Robert Harrison, and Predrag Krstić. Attosecond electron dynamics: A multiresolution approach. *Phys. Rev. A*, 85:033403, Mar 2012.
- [Vik] Daniel Vik. Fast memcpy in C. <http://www.danielvik.com/2010/02/fast-memcpy-in-c.html>.
- [VZC⁺10] A. Verma, N. Zea, B. Cho, I. Gupta, and R.H. Campbell. Breaking the mapreduce stage barrier. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 235–244. IEEE, 2010.

- [Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.
- [Wat07] Stephen D. Waton. *On Permutation Classes Defined by Token Passing Networks, Gridding Matrices and Pictures: Three Flavours of Involvement*. PhD thesis, University of St. Andrews, 2007.
- [WCC⁺07] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07*, pages 156–166, New York, NY, USA, 2007. ACM.
- [YDHP07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07*, pages 1029–1040, New York, NY, USA, 2007. ACM.
- [YRK09] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC '09*, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.