

# AUTOMATIC SCALING ITERATIVE COMPUTATIONS

A Dissertation

Presented to the Faculty of the Graduate School  
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of  
Doctor of Philosophy

by

Guozhang Wang

August 2013

© 2013 Guozhang Wang  
ALL RIGHTS RESERVED

## AUTOMATIC SCALING ITERATIVE COMPUTATIONS

Guozhang Wang, Ph.D.

Cornell University 2013

In this thesis, we address the problem of efficiently and automatically scaling iterative computational applications through parallel programming frameworks. While there has been much progress in designing and developing parallel platforms with high level programming paradigms for batch-oriented applications, these platforms are ill-fitted for iterative computations due to their ignorance of resident data and enforcement of “embarrassingly parallel” batch-style processing of data sets within every computational operators.

To address these challenges we propose a set of methods that leverage certain properties of iterative computations to enhance the performance of the resulting parallel programs for these large-scale iterative applications. More specifically, we (1) leverage data locality to reduce communication overhead within individual iterations due to data transfer, and (2) leverage sparse data dependency to further minimize inter-process synchronization overhead and enable asynchronous executions by relaxing the consistency requirements of iterative computations.

To illustrate (1) we propose a large-scale programming framework for behavioral simulations. Our framework allows developers to script their simulation agent behavior logic using an object-oriented Java-like programming language and parallelize the resulting simulation systems with millions of scripted agents by compiling the per-agent behavior logic as iterative spatial joins and distributing this query plan into a cluster of machines. We use various query

optimization techniques such as query rewrite and indexing to boost the single-machine performance of the program. More importantly, we leverage the spatial locality properties of the scripted agent behavior logic to reduce the inter-machine communication overhead.

To illustrate (2), we present a parallel platform for iterative graph processing applications. Our platform distinguishes itself from previous parallel graph processing systems in that it combines the easy programmability of a synchronous processing model with the high performance of asynchronous executions. This combination is achieved by separating the application's computational logic from the underlying execution policies in our platform: developers only need to code their applications once with a synchronous programming model based on message passing between vertices, where the sparse data dependency is completely captured by the messages. Developers can then customize methods of handling message reception and selection to effectively choose different synchronous or asynchronous execution policies via relaxing of the consistency requirements of the application encoded on the messages.

## BIOGRAPHICAL SKETCH

Guozhang Wang was born in the historic city of Xian, China, where he grew up and spent the first 18 years of his life. Until taking his college at Fudan University in Shanghai, Guozhang had no interactions with computer science and was instead fascinated by concrete mathematics. Then at Fudan he started to enjoy the fun of using computers to model and solve real world problems and completed with a Bachelor of Science degree in Computer Science in 2008. In 2006 and 2007, Guozhang was selected as a visiting student at National University of Singapore and Microsoft Research Asia, which made his first experience in being exposed to the scientific research frontier, especially in the area of data management and machine learning. This experience eventually motivated him to pursue a graduate degree in computer science.

Since 2008, Guozhang Wang has been studying for his doctorate degree in Computer Science at Cornell University.

To Xiaonan.

## ACKNOWLEDGEMENTS

I am deeply grateful to the people that I have met during my graduate study for their help and support over these years.

First, I would like to thank my advisor, Johannes Gehrke, for his vision and guidance on not only specific research projects but also scientific thinking and practical engineering, and for his encouragement to let me always think big along my research. Indeed, it was his enthusiasm and patience in teaching me from how to find and solve interesting and practical problems to how to write papers and give presentations that made me enjoy these five years of research life.

In addition to Johannes, I am also thankful to the other members of my committee, Thorsten Joachims, David Bindel, and Ping Li for the constructive feedback and advice they have provided.

Moreover, I would like to thank my mentors during my internships at Microsoft Research, Microsoft Jim Gray Systems Lab, and LinkedIn. Sanjay Agrawal gave me full confidence to discover new research problems from product experiences. Donghui Zhang treated me like a peer and the conversation with him was inspiring and a lot of fun. Neha Narkhede and Jun Rao introduced me to a brand new area of important real-world problems, which lured me to decide continue exploring in this area after my graduation.

The database group at Cornell provided me a friendly and comfortable learning environment. I learned so much from each one of them about how to do good research and how to stick to it. In particular, I would like to thank Alan Demers, who has been a vice-advisor to me that provided so much valuable advice and generous support. I would also like to thank Ashwin Machanavajjhala, Michaela Götz, and Xiaokui Xiao for mentorship during my first year of

grad school, Marcos Vaz Salles, Wenlei Xie, Tao Zou, Ben Sowell, Tuan Cao, and Walker White for their help and thoughtful suggestions in our research collaborations, Lucja Kot, Nitin Gupta, Yanif Ahmad, Lyublena Antova, Yin Lou, Sudip Roy, Gabriel Bender and Bailu Ding and for all the insightful discussions and wonderful conference trips.

Many thanks to my office mates and friends in the department, who have offered a wealth of advice and enriched my wonderful experience at Cornell: Shuang Zhao, Qi Huang, Haoyuan Li, Hu Fu, Changxi Zheng, Dustin Tseng, Huijia Lin, Danfeng Zhang, Shuo Chen, Haoyan Geng, Chenhao Tan, Hussam Abu-Libdeh, Lu Wang, Jiexun Xu, Ashwinkumar Badanidiyuru Varadaraja, and Renato Paes Leme.

Finally, I am deeply indebted to my family. To my parents and grandparents: Thanks for being my biggest source of support, even though I have to be far away from you during these year. My mom and grandpa has kept writing me a long mail every year chatting with me about life and growth. I just want to say how grateful I am to have you. And to my wife, Xiaonan: Thanks for your love and understanding. While we met back in the high school, most of the time we spent together are at Cornell. The route to a Ph.D. has never been an easy one for anyone, but with your company it has also been an enjoyment. I look forward to continuing our journey together.

## TABLE OF CONTENTS

Biographical Sketch . . . . .	iii
Dedication . . . . .	iv
Acknowledgements . . . . .	v
Table of Contents . . . . .	vii
List of Tables . . . . .	ix
List of Figures . . . . .	x
<b>1 Introduction</b>	<b>1</b>
1.1 Organization of this Thesis . . . . .	6
<b>2 Background</b>	<b>8</b>
2.1 Domain Specific Parallel Implementations . . . . .	8
2.1.1 Parallel Scientific Computation Applications . . . . .	9
2.1.2 Parallel Machine Learning Algorithms . . . . .	12
2.2 Low-level Parallel Programming Abstractions . . . . .	15
2.2.1 Parallel Programming Libraries . . . . .	16
2.2.2 Parallel Programming Languages . . . . .	19
2.3 High-level Parallel Frameworks . . . . .	22
2.3.1 Batch Processing Frameworks . . . . .	23
2.3.2 Iterative Bulk Synchronous Processing Frameworks . . . . .	25
2.3.3 Asynchronous Processing Frameworks . . . . .	27
<b>3 Scale Behavioral Simulations in Extended MapReduce</b>	<b>30</b>
3.1 Introduction . . . . .	30
3.1.1 Requirements for Simulation Platforms . . . . .	31
3.2 Behavioral Simulations in the State-Effect Pattern . . . . .	34
3.3 MapReduce for Simulations . . . . .	37
3.3.1 Simulations as Iterated Spatial Joins . . . . .	37
3.3.2 Iterated Spatial Joins in MapReduce . . . . .	39
3.3.3 The BRACE MapReduce Runtime . . . . .	41
3.4 Programming Agent Behavior . . . . .	45
3.4.1 Overview of BRASIL . . . . .	46
3.4.2 Optimization . . . . .	49
3.5 Experiments . . . . .	50
3.5.1 Setup . . . . .	51
3.5.2 BRASIL Optimizations . . . . .	52
3.5.3 Scalability of the BRACE Runtime . . . . .	54
3.6 Conclusions . . . . .	56

<b>4</b>	<b>Enable Asynchronous Execution for Large-Scale Graph Processing</b>	<b>57</b>
4.1	Introduction . . . . .	57
4.2	Iterative Graph Processing . . . . .	62
4.3	Programming Model . . . . .	69
4.3.1	Graph Model . . . . .	69
4.3.2	Iterative Computation . . . . .	70
4.4	Asynchronous Execution in BSP . . . . .	72
4.4.1	Relaxing Synchrony Properties . . . . .	73
4.4.2	Customizable Execution Interface . . . . .	75
4.4.3	Original and Residual BP: An Example . . . . .	77
4.5	Runtime Implementation . . . . .	78
4.5.1	Shared-Memory Architecture . . . . .	79
4.5.2	Batch Scheduling . . . . .	81
4.5.3	Iterative Processing . . . . .	81
4.5.4	Vertex Updating . . . . .	83
4.6	Experiments . . . . .	84
4.6.1	System . . . . .	85
4.6.2	Workloads . . . . .	86
4.6.3	Experimental Setup . . . . .	90
4.6.4	Results for Custom Execution Policies . . . . .	91
4.6.5	Results for Speedup . . . . .	96
4.7	Conclusions . . . . .	98
<b>5</b>	<b>Related Work</b>	<b>100</b>
5.1	Large-Scale Behavioral Simulations . . . . .	100
5.2	Parallel Systems for Iterative Graph Processing . . . . .	102
<b>A</b>	<b>Appendix for Chapter 3</b>	<b>104</b>
A.1	Spatial Joins in MapReduce . . . . .	104
A.1.1	MapReduce . . . . .	104
A.1.2	Formalizing Simulations in MapReduce . . . . .	105
A.2	Formal Semantics of BRASIL . . . . .	108
A.2.1	Monad Algebra Translation . . . . .	109
A.2.2	Effect Inversion . . . . .	115
A.3	Details of Simulation Models . . . . .	119
<b>B</b>	<b>Appendix for Chapter 4</b>	<b>123</b>
B.1	Convergence Rate Analysis for PageRank . . . . .	123
	<b>Bibliography</b>	<b>126</b>

## LIST OF TABLES

3.1	The state-effect pattern in MapReduce . . . . .	39
4.1	Summary of Applications . . . . .	86
A.1	RMSPE for Traffic Simulation (LookAhead = 200) . . . . .	121

## LIST OF FIGURES

3.1	BRACE Architecture Overview . . . . .	42
3.2	Class for Simple Fish Behavior . . . . .	47
3.3	Traffic - Indexing . . . . .	53
3.4	Fish - Indexing . . . . .	53
3.5	Predator - Effect Inversion . . . . .	54
3.6	Fish - Load Balancing . . . . .	54
3.7	Traffic - Scalability . . . . .	55
3.8	Fish - Scalability . . . . .	55
4.1	Graph Data for BP . . . . .	70
4.2	Update logic for BP . . . . .	72
4.3	Synchronous Original Execution for BP . . . . .	78
4.4	Asynchronous Residual Execution for BP . . . . .	79
4.5	Data Flow in GRACE Runtime . . . . .	80
4.6	Qualitative Evaluation of BP Restoration on Lenna Image. Left: noisy ( $\sigma = 20$ ). Right: restored . . . . .	89
4.7	SfM Running Time . . . . .	91
4.8	SfM Speedup . . . . .	91
4.9	IR Convergence . . . . .	92
4.10	TPR Convergence . . . . .	92
4.11	IR Running Time . . . . .	93
4.12	IR Speedup . . . . .	93
4.13	TPR Running Time . . . . .	93
4.14	TPR Speedup . . . . .	93
4.15	LP Convergence . . . . .	94
4.16	LP Running Time . . . . .	95
4.17	LP Speedup . . . . .	95
4.18	IR Speedup <sub>32core</sub> . . . . .	96
4.19	TPR Speedup <sub>32core</sub> . . . . .	97
4.20	TPR <sub>loaded</sub> Speedup <sub>32core</sub> . . . . .	98
A.1	Map and Reduce Functions with Local Effects Only . . . . .	106
A.2	Map and Reduce Functions with Non-local Effects . . . . .	108
A.3	Translation for Common Commands . . . . .	112

# CHAPTER 1

## INTRODUCTION

The last two decades have witnessed the birth and growth of a new era, the era of *Big Data* where a continuous increase in computational power has produced an overwhelming flow of data. Examples of such large scale datasets come from both industry (e.g., web-data and social-network analysis with billions of webpages and items, stream-based fraud detection with hundreds of input events every second) and the sciences (e.g., massive-scale simulations-based climate modeling, genome sequence analysis, astronomy supported by powerful telescopes and detectors) and the growing demand for large-scale data processing has spurred the development of novel solutions. On the other hand, the trend of the increasing number of transistors on a single chip as predicated by Gordon Moore in 1965, hence the roughly equivalent increasing performance of a single CPU, has already run into physical limits. This scenario calls for a paradigm shift in the computing architecture, and parallelism has become the norm for large scale computations.

However, writing a parallel program has earned its reputation over the years as one of the most difficult tasks for software and system developers. People have to deal with new classes of potential software bugs introduced by concurrency, such as race conditions, deadlocks and livelocks, and non-determinism. Furthermore, parallel overheads such as communication and synchronization for distributed memory environment and interthread coordination through message passing or shared objects for multicore architecture are typically the obstacles to getting good parallel performance, according to Amdahl's Law. As a result, over the past years a long-standing problem has re-attracted

people's attention: how can we make it easier for developers to write efficient parallel and distributed programs for large scale computational applications?

Various approaches have been proposed to solve this problem at different levels of the system architectures and at different levels of abstraction for the programmers. Among these approaches, a promising direction that has emerged recently is to provide parallel systems with high level programming paradigms that can implicitly guide developers themselves to explore data parallelism within their applications. In this way, developers can be provided with the abstraction from the parallel details and instead abstracted with an illusion of writing a sequential program that later can be automatically scaled up. Successful examples of such an approach include parallel databases [65, 64] and data-parallel processing frameworks such as MapReduce [63] and Dryad [94]. Most of these systems are designed to target on batch-oriented data processing tasks which can usually be expressed as data workflows of simple data parallel-operators such as aggregates and sorting. Example applications of such tasks include web services and simple analytics [94]. In addition, these frameworks usually assume an *embarrassingly parallel* property within their application, i.e., the application can be partitioned into parallel tasks such that there exists little or even no dependency between these tasks.

On the other hand, many other large-scale applications in scientific computing, machine learning and graph processing, etc, are iterative or incremental computations in which the same computational functions will be applied repeatedly, using the output from one iteration as the input to the next until some fixpoint is achieved. Due to the dependency of the previous output data on the current iteration's computation, under normal execution, each iteration will not

be triggered until the previous iteration has been completely finished and the computation of the current iteration will only read input data generated from the previous iteration, in order to guarantee *consistency* of the application under parallel processing. Developing iterative applications using such batch-oriented programming frameworks usually results in inferior performance. For example, most parallel databases use a domain-specific language called SQL [55] to process the stored data usually as OnLine Analytical Processing (OLAP) queries, in which it is not natural or even possible to express iterative algorithms that make passes over a data set, and people usually need to make some workarounds for this problem such as using virtual tables or windowed aggregates, or using a driver function written in another script with another language to trigger the SQL query for each iteration. The former approach does not provide either generality or portability (e.g., the level of support for windowed aggregates varies across SQL engines), while the latter approach requires pulling a large amount of data out of the database which becomes a scalability bottleneck since the driver code typically does not parallelize [85]. As another example, the “Map” phase of the MapReduce model is borrowed from the functional programming literature, which requires that the input data can be divided into smaller subsets, and each subset can be distributed and processed independently by a parallel processor [63]. The output tuples will be shuffled by their keys and materialized to stable storage to be read by the next “Reduce” phase. If we execute each iteration of computations of an iterative application as one or multiple Map and Reduce step, we then need to shuffle and materialize the data between iterations [109].

In this thesis, we try to resolve this problem by providing a new class of parallel programming frameworks that leverage some specific iterative compu-

tational properties in order to enhance the performance of the result programs for these iterative applications. We mainly consider two common properties of iterative computations: *data locality* and *sparse data dependency*. Here data locality means that since the output of every iteration will be used as input of the next iteration, we should better treat these intermediate results between iterations not as part of a dataflow process but instead as resident data whose values will be iteratively modified by the same update logic. Therefore, we can try to reduce data transfer by processing distributed data as close as possible to where it resided in the previous iteration. In addition to data locality, although we cannot assume “embarrassingly parallel” computational patterns for iterative computations, we do observe that the computation of each data unit is only dependent on a small subset of the whole data set; we say that the computation has only sparse data dependencies. For example, many scientific simulations expose certain neighborhood properties such that the update logic of each simulation unit is only dependent on its physical neighbors in the simulation space [155]. In many graph processing applications such as PageRank, the computational logic is distributed as the update function for each individual vertex, which only depends on its neighboring vertices [41]. Also in many machine learning models such as Support Vector Machines and Latent Dirichlet Allocation, although the parameter space can be very large, evaluation of individual data points only modifies a small part of the parameters [129, 113]. As a result, we can reduce the inter-process communication overhead and also enhance the convergence rate of such applications by relaxing their consistency requirements.

To demonstrate these ideas we have implemented two parallel programming frameworks as case studies. The first one is designed for large scale be-

behavioral simulation developers to easily code their applications as a sequential program which can then be automatically parallelized across a cluster. We can leverage spatial locality to treat behavioral simulations as iterated spatial joins and greatly reduce the communication between nodes. In our experiments we achieve nearly linear scale-up on several realistic simulations.

Though processing behavioral simulations in parallel as iterated spatial joins can be very efficient, it can be much simpler for the domain scientists to program the behavior of a single agent. Furthermore, many simulations include a considerable amount of complex computation and message passing between agents, which makes it important to optimize the performance of a single node and the communication across nodes. To address both of these challenges, BRACE includes a high-level language that has object-oriented features for programming simulations, but can be compiled to a dataflow representation for automatic parallelization and optimization. We show that by using various optimization techniques, we can achieve both scalability and single-node performance similar to that of a hand-coded simulation.

We then switch to a different class of iterative computations: graph processing. Scaling large-scale iterative graph processing applications through parallel computing is a very important problem. Several graph processing frameworks have been proposed that insulate developers from low-level details of parallel programming. Most of these frameworks are based on the bulk synchronous parallel (BSP) model in order to simplify application development while achieving good scalability [42, 98, 119, 135, 167, 169]. However, in the BSP model, vertices are processed in fixed rounds, which often leads to slow convergence. Asynchronous executions can significantly accelerate convergence

by intelligently ordering vertex updates and incorporating the most recent updates. Unfortunately, asynchronous models do not provide the programming simplicity and scalability advantages of the BSP model.

Our second framework is exactly designed for such large scale graph applications. We have designed a new graph programming platform that separates application logic from execution policies to combine the easy programmability of the BSP model with the high performance of asynchronous executions. This platform provides a synchronous iterative graph programming model for users to easily implement, test, and debug their applications. It also contains a carefully designed and implemented parallel execution engine for both synchronous and user-specified built-in asynchronous execution policies. The framework lets developers explicitly define sparse data dependencies as messages between adjacent vertices, and allows them to specify different ways of relaxing such dependencies in order to enable asynchronous processing features for better convergence rate. Our experiments show that asynchronous execution in our platform can yield convergence rates comparable to fully asynchronous executions, while still achieving the near-linear scalability of a synchronous BSP system.

We present the details of all these methods in the following chapters.

## **1.1 Organization of this Thesis**

The subsequent chapters of this thesis are organized as follows: First, we lay out some background on previous approaches to scale iterative applications in Chapter 2. Then we present our large scale behavioral simulation framework

in Chapter 3 and our parallel framework for graph processing applications in Chapter 4. Chapter 5 discusses existing work on behavioral simulations and iterative graph programming systems. Finally, we provide several appendices on the implementation and theoretical details about the material presented in Chapters 3 and 4.

## CHAPTER 2

### BACKGROUND

Many approaches have been proposed to scale iterative computational applications at different architectural abstraction levels. At the lowermost level, a number of hand-coding parallel implementations have been provided solely for certain domain specific applications (Section 2.1). In order to extend such domain specific implementations across different hardware and operating systems, another category of works captures a core of computation and communication routines and abstracts them within general programming libraries or programming languages (Section 2.2). Most recently, system researchers have focused on providing even higher level parallel programming frameworks that can further free application developers from considering low level parallel processing issues, instead providing them with the illusion of coding a single-node program that will be automatically parallelized (Section 2.3). In the remainder of this chapter, we will survey each one of these three categories of works.

#### **2.1 Domain Specific Parallel Implementations**

In this section we describe a number of domain specific hand-coded parallel implementations for large scale iterative computing applications. Developers of such implementations are usually domain experts who understand their applications very well but may not be familiar with details of data locality and communication, synchronization and concurrency control, etc. Therefore, substantial training in parallel programming and distributed processing is usually required for these domain experts before they can build scalable systems that are optimized for their applications. In addition, their parallel processing op-

timization techniques cannot be shared and reused, even though they have already been commonly encoded to leverage the inherent parallelism in the applications, since each of these parallel implementations are specialized only for certain application and hence developed from scratch.

In this section, we will pay particular attention to scientific computation (Section 2.1.1) and machine learning (Section 2.1.2) applications. However, we should note that in many other fields, such as social network analysis [34], computer vision [60] and information retrieval [107], there also exist a plethora of specialized parallel implementations that are developed by domain experts.

### **2.1.1 Parallel Scientific Computation Applications**

The problem of how to efficiently solve large scale scientific tasks in an iterative manner has been widely studied in the high performance computing literature. There are many areas in which parallel processing can be used to greatly enhance the performance of iterative computations, including financial modeling [141], astrodynamics [56], civil engineering [84] and computational biology [43]. As a concrete example, behavioral simulations are instrumental in understanding large-scale complex systems such as transportation networks, insect swarms, or panicked individuals, which can have tremendous economic and social impacts [43, 59, 84, 35]. Behavioral simulations proceed in a time-stepped model in which time is discretized into “ticks”. Within each tick all simulation agents execute actions concurrently. These actions usually result from interactions with other agents. Hence behavioral simulation applications demonstrate a typical iterative scientific computation pattern. So rather than

enumerate a list of parallel programs, one for each scientific computation application, we attempt to focus on sketching the approaches behavioral simulation scientists have taken to write their large scale simulation applications.

Today most behavioral simulation platforms designed for ease of implementation are still based on single node, since scaling these frameworks is very hard due to heavy and complicated communications between simulated agents. For example, SWARM is a multi-agent simulation platform in which a group of agents is defined as a swarm (e.g., a swarm of rabbits and coyotes), and the swarm events (e.g., rabbits hide from coyotes, coyotes eat rabbits) specify the interaction of agents [124]. Further, agents cannot sense their environment directly (e.g., coyotes ask how many rabbits in certain area), but can only get the information from the swarm object. Although the communication overhead can be largely minimized by simplifying agent interaction to swarm interaction, this structure makes the simulator hard to parallelize, since a swarm of millions of agents must be located on a single machine. Other single node simulation platforms that have similar architecture include MASON [115], TransCAD [21], SUMO [19], TransModeler [22], etc. Another category of single-node simulators tend to not based on time-stepped procedure but control agent interaction by scheduling and transmitting discrete events between agents. In other words, the operation of these simulation systems is represented as a chronological sequence of events. Each event occurs at a simulation agent in time and marks a change of state in the agent. Each agent then communicates with each other via asynchronously receiving events and propagating new ones. The system states are in turn changed by these events. This type of simulators is categorized as “event-based simulations” and it includes SHIFT [31], FLAME [4], REPAST [16], PARAMICS [108], etc. Although there have been attempts to parallelize general

agent-based simulations over multiple nodes [33, 127, 140, 170, 47, 162], they either scale poorly, or they require unrealistic restrictions or simplifications for behavioral simulation models in order to achieve reasonable scalability. For example, in the traffic domain, the MITSIMLab system includes a micro-simulator with highly detailed driver models [162]. However, MITSIMLab relies on task parallelism (i.e., functional decomposition) instead of data parallelism, which means that different modules can run on different computers. For example, the micro-simulation could run on one computer, while an on-line routing module could run on another computer. While task parallelism is somewhat easier to implement and also is less demanding on the hardware to be efficient, the achievable speed-up is limited by the number of different functional modules. Under normal circumstances, one probably does not have more than a handful of these functional modules that can truly benefit from parallel execution. SWAGES [140], on the other hand, applies data parallelism (i.e., domain decomposition) to partition agent data across machines, but can only handle embarrassing parallelism among completely disjoint agent clusters. In other words, agents that need to communicate must be assigned to the same process.

As a result, developers have resorted to hand-coding parallel implementations of specific simulation models [127, 72, 157]. For example, the driving logic of the TRANSIMS traffic micro-simulation driving logic is based on a cellular automata (CA) technique where roads are divided into cells [88]. Each cell is a length that a car uses up in a jam. All the driving logical modules are computed based on only neighbor cell values and meta-data of the road on which the cell is located. Such driving logic is implemented in parallel using graph partitioning algorithms (e.g., orthogonal recursive bisection or more complicated multi-level partitioning algorithms in METIS [9]) to decompose the traffic network and as-

signs each partition to a different processes [127]. Networks are cut by links instead of the intersections, with each divided link fully represented in both CPUs in order to separate the traffic complexity at the intersections from the complexity caused by the parallelization.

As we have noted before, this approach brings a lot of programming burdens to the simulation scientists since the code handling low-level parallel processing details cannot be shared among different programs. In addition, since these parallel implementations are usually specialized to the fixed set of behavioral models, as those models evolve over time, their original parallel implementations may not be able to scale any more, and new programs needs to be designed and implemented. For example, the parallel implementation of the TRANSIMS simulation logic depends on the fact that the logic in based on CA, hence only replicating the cut links on neighbor CPUs is sufficient [127]. This parallel implementation of TRANSIMS may need substantial modifications to accommodate any driving logic changes that require information beyond the current located link.

## **2.1.2 Parallel Machine Learning Algorithms**

Sequential machine learning algorithms are not feasible for many large scale real-world problems [137, 91, 60]. On the other hand, advances in hardware and distributed computing have enabled scientists to develop machine learning algorithms over multi-core and distributed environments at unprecedented scales. Therefore, many approaches that confront the challenges of parallelism for large-scale machine learning algorithms have been proposed in the liter-

ature, and many of these algorithms involve iterative computations, such as distributed Gibbs sampling for Latent Dirichlet Allocation (LDA) [156], parallel Approximate Matrix Factorization for Support Vector Machines (SVM) [171, 50], parallel Spectral Clustering [144] and the EM algorithm for graphical models with latent variables [161]. One key observation shared by those approaches is that for most iterative machine learning algorithms, the computations within a single iteration are independent and hence can be naturally decomposed, allowing subsets of data to be processed in parallel. More specifically, for those models whose parameter learning process is based on sufficient statistics or gradients, the data can be subdivided into distributed nodes. So calculation of the sufficient statistics and gradients will require little communication between nodes [46].

Take the EM algorithm for word alignment in a corpora of parallel sentences as an concrete example, where each parallel sentence pair consists of a source sentence  $S$  and its translation  $T$  into a target language [161]. Using a mixture model, each word of  $T$  is considered to be generated from some word of  $S$  or from a null word  $\emptyset$  prepended to each source sentence. The null word allows words to appear in the target sentence without any evidence in the source. The formal generative model is as follows: (1) Select a length  $n$  for the translated sentence  $T$  based upon  $|S| = m$ ; (2) For each  $i = 1, \dots, n$ , uniformly choose some source alignment position  $a_i \in \{0, 1, \dots, m\}$ ; (3) For each  $i = 1, \dots, n$ , choose a target word  $t_i$  based on source word  $s_{a_i}$  with the probability parameter  $\theta_{s_{a_i}, t_i}$ . Given this model, the expected sufficient statistics are expected alignment counts between each source and target word that appear in a parallel sentence pair. These expectations can be obtained from the posterior probability of each alignment:  $P(a_i = j | S, T, \theta) = \frac{\theta_{s_j, t_i}}{\sum_{j'} \theta_{s_{j'}, t_i}}$ . Note that with the given  $\theta$  parameters,

the above posterior computation in the E(xpectation)-step can be done independently for each parallel pair, while the M(aximization) needs to read the aggregated posterior values as the sufficient statistics to update its  $\theta$  parameters.

This embarrassingly data parallel calculation can naturally fit in the MapReduce framework and therefore be easily distributed over multiple processes [151, 130]. In these works, data is partitioned by training examples, and each partition is associated with a mapper task to collect the intermediate data (e.g., sufficient statistics and gradients) as part of the inference process in parallel. If this inference process is not embarrassingly parallel but based on some irregular structures with certain level of data locality, it can usually be structured in a graph-based computation manner: data is stored on vertices, and the calculation is decomposed into processing of individual vertices. Each vertex's computation only depends on its connected neighbors and hence be still be processed in parallel mapper tasks [119]. However, one major problem for applying the MapReduce framework to iterative computations is that there is a global synchronization barrier at the end of each iteration in order to process the collected intermediate data and update the global constraints or variables. This procedure can not be parallelized and hence must be handled in a single reducer task.

As a result, much research has tried to relax this synchronization barrier by allowing asynchronous data update for various specific algorithms [66, 70, 77, 129]. For example, HOGWILD! is a specialized implementation strategy for Stochastic Gradient Descent (SGD) for multicore platforms. The goal of SGD is to minimize a function  $f : X \in \mathcal{R}^n \rightarrow \mathcal{R}$  of the form  $f(x) = \sum_{e \in E} f_e(x_e)$ , where  $X$  denotes the model parameter vector and  $e$  denotes a small subset of

$\{1, \dots, n\}$  with  $x_e$  denotes the values of the vector  $x$  on the coordinates indexed by  $e$  and  $f_e$  denotes the cost function associated to  $e$ . In practice, the cost functions are *sparse* with respect to parameter vectors in the sense that each individual  $f_e$  only depends on a very small number of components of  $x$ . In HOGWILD!, processors are allowed equal access to shared memory and are able to update individual components of memory in order to eliminate the overhead associated with locking. More specifically, each processor will iteratively sample one  $e$  uniformly from  $E$ , read current state  $x_e$  and evaluate  $G_e(x)$ , a gradient or sub-gradient of the function  $f_e$ , then for each  $v \in e$  update  $x_v$  with  $G_e(x)$ . Note that without locks, each processor can asynchronously access and update parts of  $X$  (hence probably overwrite each other's progress) at will. The authors show that when such synchronous data access is *sparse* they introduce barely any error into the convergence result. However, the precise conditions under which general asynchronous execution will converge to the same result as synchronous execution are still not well understood.

## 2.2 Low-level Parallel Programming Abstractions

In this section we survey low-level parallel programming abstractions, including parallel programming libraries and data parallel languages. Such programming abstractions provide a core of semantics and routines useful to a wide range of users exploiting parallelism and writing portable parallel code. Key properties of the architecture, including hardware and operating systems, are isolated from key aspects of a parallel program, including data distribution, communication, I/O, and synchronization. Such isolation of architecture-sensitive aspects simplifies the task of porting programs to new platforms.

In the remainder of the section we give a sketch on two categories of low-level parallel programming abstractions: parallel programming libraries (Section 2.2.2) and parallel programming languages (Section 2.2.2).

### 2.2.1 Parallel Programming Libraries

Most parallel programming libraries apply the Single Program Multiple Data (SPMD) technique to achieve data parallelism. In other words, programs are coded once and assigned to multiple processes and run simultaneously with split data input. Inter-process communication are usually abstracted as message passing or distributed shared memory access. There are two types of messages: *synchronous* and *asynchronous*. Synchronous message passing requires both the sender and the receiver to wait for each other while transferring the message, while in asynchronous message passing the sender and receiver do not block when sending and getting the message, but can overlap their communication with other computations.

In the message-passing communication paradigm, messages are sent from a sender process to one or more recipient processes, and each process has its own local data that can only be accessed by itself. Remote data values cannot be read or updated directly, but can be learned from the received messages. As a concrete example, the Message Passing Interface (MPI) Standard has dominated distributed-memory high-performance computing since the mid-1990s due to its portability, performance, and simplicity [79]. Although designed for distributed memory systems, MPI programs are also regularly run on shared memory computers. Most MPI implementations consist of a specific set of rou-

tines (e.g., point-to-point and collective, synchronous and asynchronous message passing, etc) directly callable from C/C++, Java and Python, etc. Similar libraries using message passing include Parallel Virtual Machine (PVM) [14] and Charm++ [97].

Another paradigm of inter-process communication is shared memory, where one process can create data in a shared global data space that processes can access directly. Shared memory is an efficient means of passing data between processes without replicating it. Examples of shared memory programming libraries include Unified Parallel C (UPC) [2], OpenMP [20], POSIX Threads (PThreads) [45], Ateji PX [1] and NVIDIA CUDA/OpenACC [12]. Although some of the above libraries such as PThreads and OpenMP can only be used for shared memory multiprocessing programming on the multicore environment, some others such as UPC can also be used for distributed memory environments. Therefore, we take UPC as an illustrative example of shared memory parallel programming libraries<sup>1</sup>.

UPC is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The programmers are presented with a single shared, partitioned address space, while each variable is physically associated with a single processor. In order to expose this fact to the programmers, UPC distinguishes between shared and private variables at each thread, and each physical processor can contain at least one thread. The same address for shared data on each thread refers to the same physical memory location, whereas the same address for private data on each thread refers

---

<sup>1</sup>In fact, UPC and CUDA are designed as parallel programming languages with interoperability to ordinary C/C++, and OpenMP is designed with interoperability to Fortran. We treat them as parallel programming libraries here since they are more focused on explicitly supporting core parallel programming routines such as parallel iterations, barrier synchronization and local/global memory regions.

to distinct physical memory locations. The programmer declares the affinity of each variable to a specific thread. Like synchronous and asynchronous message passing mechanisms, UPC provides two user-controlled consistency models for interactions between memory accesses to shared data: *strict* and *relaxed*. Strict access always appear (to all threads) to have executed in program order with respect to other strict accesses. In a given execution of a program whose only accesses to shared data are strict the program is guaranteed to behave in a sequentially consistent [104] manner. On the other hand, a sequence of purely relaxed shared accesses issued by a given thread may appear to other threads to occur in any order except as constrained by explicit barriers and memory fence operations.

Although parallel programming libraries can help users write portable parallel programs across different architectures, in domains where parallel applications evolve rapidly the relatively low level of parallel libraries is still perceived as a significant drawback. For example, MPI's low-level programming abstraction creates several difficulties for developers of iterative computation applications. In particular, the synchronization barriers at the end of each iteration pose a heavy communication overhead on the application, which can hardly be handled without significant optimizations on the communication layer [82, 117]. As a result, some higher level libraries/packages and specialized implementations of the low-level libraries whose main goal is minimizing communication overheads for high performance computing have been proposed for some specific data parallel computations [99, 73, 13, 18]. For example, PETSc is a suite of sparse linear algebra data structures and routines for the parallel solution of scientific applications modeled by partial different equations (PDEs). It has been provided over many low-level libraries such as MPI, PThreads and NVIDIA

CUDA. Unfortunately, this work does not generalize to the wide class of data parallel (iterative) applications, hence low-level abstractions like MPI still remain the dominant programming paradigm.

## 2.2.2 Parallel Programming Languages

Instead of parallel programming libraries which are usually based on some standard imperative languages such as C/C++ and Java, parallel programming languages are designed to provide higher level programming abstractions. That is, rather than guiding users to explore the sub-routines that can be parallelized within in a sequentially written program, parallel programming languages are designed to either explicitly or implicitly enforce users to consider the parallelism of their applications' computation patterns. We start the discussion with general-purpose languages first, then move on to domain-specific languages.

In contrast with imperative programming, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. The control flow describing "when" and "how" the computation logic should be executed is then left to the language's implementation (e.g., through static analysis on the compiler, or dynamic scheduling mechanism on the runtime). Although declarative programming languages are not originally designed for parallel processing, they have become of particular interest for parallel programming recently, since by applying the declarative programming paradigm one can attempt to minimize or even eliminate side effects that may change the value of program state, and the written programs can be much easily to be parallelized. For example, a well known sub-category of

declarative programming falls in functional programming, which treats computation as the evaluation of user-defined functions and avoids state and mutable data. Since functions cannot change the state of anything, the resulted programs is side effects-free and programmers on longer need to worry about various concurrency control issues such as shared data access and synchronization.

General-purpose declarative programming languages include Lisp [121], Erlang [3], Haskell [5], Prolog [54], OCaml [11] and Scala [17], etc. We take Erlang, a general-purpose concurrent programming language as a concrete example. Processes are the primary means to structure an Erlang application, and the language provides a set of primitives and features for managing processes. Inter-process communication works via shared-nothing asynchronous message passing: every process maintains a queue of messages from other processes that have been received but not consumed yet. A process can retrieve messages from the receiving queue and try to match its desired patterns to a message-handling routine. When a handling routine has been found that matches the message, it will be triggered to handle and consume the message. When the message is consumed and removed from the queue the process resumes execution. Hence Erlang programs have limited synchronization power, and programmers can only partially control the consuming order by the expressing order of the message-handling routines.

The above declarative languages can help users to simplify writing parallel programs by removing or eliminating side effects of the computation. However, concurrency control is still explicit and has to be specified by the users. There are another type of higher-level parallel programming languages that are designed to automate data parallelism over regular data structures [143, 40].

These languages usually integrate ideas from declarative programming languages. Programmers using these data parallel languages are only required to write a single-process code, which will then be automatically distributed and parallelized at the underlying runtime. However, these approaches only support restricted data structures such as sets and arrays, limiting its expressiveness.

Besides general-purpose parallel programming languages, there are also a large number of domain-specific languages (DSLs) that are based on declarative programming and data parallelism. In addition, the underlying compilers and execution runtimes usually leverage application-specific properties to automatically optimize the written program. As a well-known special-purpose programming language, SQL is designed for querying and managing data in relational database management systems (RDBMS). It is based on relational algebra and tuple relational calculus, and was one of the first commercial languages for Edgar F. Codd's relational model [25, 55]. It follows the declarative language paradigm with procedural elements such as control-of-flow constructs and user defined functions (UDFs). SQL queries allow users to describe the desired data, leaving DBMS responsible for planning, optimizing and performing the physical operations necessary to produce that result as it chooses. There are a number of excellent surveys on query optimization techniques for parallel/distributed databases, which are based on relational algebra [93, 27, 118]. Datalog, on the other hand, is another query language for deductive databases based on logic programming [74]. It is similar to Prolog with certain syntax and semantics constraints to guarantee that Datalog queries on finite sets always terminates. Query evaluation with Datalog is based on first order logic and usually done bottom-up. The declarative programming style of both lan-

guages makes it easier to parallel process the written programs on a cluster of distributed database services or parallel databases [75, 65, 78]. In recent years, Datalog has also found new application for parallel processing in other areas such as data integration, information extraction, networking, program analysis, security, and cloud computing [112, 111, 110, 57, 90]. Other examples of such DSLs include R for statistics [15], Matlab/Mathematica for technical computing [8, 24], SGL/BRASIL for computer games and simulations [146, 159, 155], and Bloom/Dedalus for data-centric distributed applications [28, 29].

### **2.3 High-level Parallel Frameworks**

Although low-level parallel programming abstractions can help users write portable parallel programs that are independent of the underlying infrastructure, and hence have been used as ad-hoc parallel implementations for iterative computation applications across various fields, they still require users to consider parallel processing issues such as concurrency control, communication and fault tolerance. In order to mitigate this problem, during the past few years many high-level parallel programming frameworks have emerged to present the programmer an illusion of writing a single process program while still implicitly guiding them to consider the data parallelism of the computation. Once an application is cast into a certain framework, it will be automatically parallelized the system will deal with the low-level parallel details such as data partitioning, task scheduling and inter-process synchronization.

In this section we survey three categories of high-level parallel frameworks in terms of their targeted computation patterns: batch processing (Section 2.3.1),

iterative bulk synchronous processing (Section 2.3.2), and asynchronous processing (Section 2.3.3).

### 2.3.1 Batch Processing Frameworks

Batch processing frameworks have first been proposed for distributed web service applications which involves extensible data analysis. Such data analysis tasks usually contain a set of aggregation operations that have input/output data dependencies between each other. Such data dependencies expose a natural control flow of the computation. As a result, users of these frameworks are only required to provide batch style data processing logic for each operation and dependencies between operations, and the frameworks contain modules for resource management, operation scheduling, and fault tolerance, etc.

Since its introduction in 2004, MapReduce/Hadoop has been one of the most successful programming models for processing large data sets on clusters of distributed computers [81, 63, 133, 51, 168]. This programming model is based on two functional programming primitives that operate on key-value pairs. The *Map* function takes a key-value pair and produces a set of intermediate key-value pairs, while the reduce function collects all of the intermediate pairs with the same key and produces a value. The underlying runtime allows for parallel processing of the Map and Reduce operations given that these operations are independent regarding the input/output data. Although originally proposed for distributed computers, MapReduce/Hadoop has also been recently extended to multicore shared memory architectures [138, 164]. In addition, due to its parallel automatic resource management and fault tolerance,

MapReduce/Hadoop has also been combined with parallel database query optimization techniques to generate large-scale data analysis platforms with SQL language support [26, 147].

DryadLINQ is another programming environment for writing data parallel application running on shared-nothing environments [165, 94, 95]. It follows the MapReduce/Hadoop model but with a higher-level programming interface. Computation are expressed in a declarative language, the .NET Language Integrated Query (LINQ). LINQ enables developers to write and debug their applications in a SQL-like manner, and the written sequential declarative code can be compiled to highly parallel query plans spanning large computer clusters. In addition, DryadLINQ also exploits multi-core parallelism on each machine.

FlumeJava is a Java library for pipelining of MapReduce tasks at Google [48]. A variety of optimization techniques such as operation combination and deferred evaluation are applied to achieve better pipeline latency. Dremel complements the MapReduce model by allowing users to express interactive ad hoc queries over distributed computers [123]. In order to support interactive queries, Dremel uses columnar nested storage and hierarchical query processing engines to read data from disk in parallel and execute queries in a few seconds.

Piccolo [135] and Spark [167] are main-memory based distributed parallel processing frameworks. While still following the MapReduce/Hadoop programming model, these frameworks add support for resident or cached state in order to reduce the overhead of streaming large amounts of data between dependent operations. In other words, data is kept in memory between operators instead of being written to disk. Since data is no longer materialized on

persistent storage, periodic checkpoints are applied for failure recovery.

### 2.3.2 Iterative Bulk Synchronous Processing Frameworks

Batch processing frameworks have limited support for iterative applications. Therefore, other frameworks specifically designed for these types of applications have been proposed. Many of these frameworks are based on one common property of iterative computations: the computation within each iteration can be decomposed over individual data records, and each tuple’s computation only depends on a very small subset of the complete data set. As a result, these frameworks commonly follow the bulk synchronous parallel (BSP) model of parallel processing [153, 166, 154]. The BSP model organizes computation into synchronous “ticks” (or “supersteps”) delimited by a global synchronization barrier. In order to achieve parallelism, data is partitioned and assigned to different processes. Within an iteration, each process will process the partitions it owns independently, and all inter-process communication is done at the synchronization barriers by message passing. Due to sparse data dependencies, the size of the messages communicated at the barriers — and hence the synchronization overheads — will be small. This model ensures determinism and maximizes data parallelism within ticks, making it easy for users to design, program, test, and deploy parallel implementations of iterative computations while achieving excellent parallel speedup and scaleup.

Pregel is a distributed large-scale graph analysis framework developed on top of Google MapReduce [119]. Computation in Pregel is decomposed into the update logic of individual vertices, and vertices are hash partitioned to dis-

tributed machines. Vertices can only communicate with their connected neighbors by sending messages. Within each superstep, each vertex will be updated at most once based on the received messages, and a global synchronization barrier is imposed at the end of each superstep to guarantee consistency across machines. However, since vertices are partitioned based on their hash values, data locality is not explored to reduce communication overhead.

PEGASUS is implemented on top of Hadoop for large-scale graph mining applications [98]. It represents linear graph algorithms such as PageRank, random walks, diameter estimation and connected components in an iterative matrix-vector multiplication (GIMV) pattern so that it can be easily distributed on the Hadoop platform. More specifically, each iteration's computation is a multiplication of the edge matrix with the vertex vector, and the computation will only terminate when the resulted vector does not change any more (i.e., convergence has been achieved). With this representation, block multiplication can be applied to reduce the amount of data that need to be synchronized between iterations.

Twister is an alternative MapReduce runtime for iterative computations [68]. In order to efficiently apply iterative computations, Twister separates static data that will only be read through the computation from variable data that will be updated iteratively. Static data is only loaded once at the beginning and will be kept in memory as status, while variable data will be read from disk, processed, and written back to disk during each iteration. Since variable data usually need to be read by each Reducer operator, communication between Map and Reduce operators is done via publish/subscribe message passing. Similar approaches have also been proposed by Dionysios et al for incremental data analytics [109].

Naiad [122] is an iterative version of the DryadLINQ framework, in which each iteration's computation output is not updated data values but increments. Increments will be applied to the old data values at the synchronization barriers. If no increments are generated within certain iteration, a fixed point is reached and the computation terminates. Similar approaches also include HaLoop, which is a modified version of the Hadoop framework designed to serve iterative large-scale applications [42]. HaLoop not only extends MapReduce with programming support for iterative applications, but also dramatically improves their efficiency by making the task scheduler loop-aware and by adding various caching mechanisms.

### 2.3.3 Asynchronous Processing Frameworks

In contrast to synchronous execution of iterative applications in the BSP model, the main characteristic of an *asynchronous* parallel processing algorithm is that local computation does not have to wait for its dependent data to be available. If the current value of the dependent data updated by some other process is not available, then some outdated value received at some time in the past is used instead. In other words, there is no clear global synchronization barriers during the computation, hence some processes can compute faster and execute more iterations than others.

There are several potential advantages that may be gained from asynchronous execution [38, 39]: First, there is no overhead such as the one associated with the global synchronization method. Second, since data updates are asynchronous, the most recent state can be used as soon as it becomes available.

In addition, updates can be intelligently ordered so that data updates that contribute more to convergence can be incorporated faster. Both of these factors increase the convergence rate. Therefore, although asynchronous implementations are not guaranteed to converge to the same fixpoint as the corresponding synchronous algorithms in general, they are still extensively applied in practice for better performance [70, 129, 169, 102, 136].

A major potential drawback of asynchronous processing, however, is that asynchronous parallel programs are much more difficult to write, debug, and test than synchronous parallel programs. Therefore, several asynchronous parallel processing frameworks have been proposed trying to mitigate this problem.

GraphLab [113] is an asynchronous processing framework for large-scale machine learning applications. GraphLab abstracts local dependencies of the learning model parameters as directed graphs and global constraints as shared data tables. Therefore, local computation can be defined in GraphLab as update procedures over individual vertices, and global aggregation can be defined as the synchronization function over the whole graph. When an update task is scheduled, it computes based on whatever data is available on the vertex itself and possibly its neighbors. As for the synchronization function that aggregates data across all vertices in the graph, it can be set to run either periodically in the background along with local vertex update procedures, or actively triggered which will halt any local computation. However, since adjacent vertices can be scheduled simultaneously, resulting in read and write conflicts, users need to specify consistency models (vertex consistency, edge consistency, etc) for local computation. The multicore parallel runtime provides a number of

dynamic scheduling schemes to update vertices respecting user defined consistency model, such as Jacobi and Gauss-Siedel. Recently GraphLab has been extended for distributed memory environment, which uses distributed locking for inter-partition synchronization [114].

Galois is another asynchronous parallel processing framework designed for irregular applications on shared memory architecture [103]. In Galois, different processes can simultaneously iterate over the shared data set, updating their data in an optimistic parallel manner. Users need to specify which method calls can safely be interleaved without leading to data races; they also need to specify how the effects of each method call can be undone when conflicts are detected.

## CHAPTER 3

### SCALE BEHAVIORAL SIMULATIONS IN EXTENDED MAPREDUCE

In this chapter we present BRACE (Big Red Agent-based Computation Engine), which extends the MapReduce framework to process behavioral simulations efficiently across a cluster by leveraging spatial locality to reduce the communication between nodes. In addition, BRACE includes a high-level language called BRASIL (the Big Red Agent Simulation Language) to allow domain scientists to easily script the behavior of a single agent and scale this behavior logic to millions of agents. BRASIL has object-oriented features for programming simulations, but can be compiled to a dataflow representation for automatic parallelization and optimization. We show that by using various optimization techniques, BRACE can achieve both nearly linear scale-up and single-node performance similar to that of a hand-coded programs on several realistic simulations.

#### 3.1 Introduction

Behavioral simulations, also called agent-based simulations, are instrumental in tackling the ecological and infrastructure challenges of our society. These simulations allow scientists to understand large complex systems such as transportation networks, insect swarms, or fish schools by modeling the behavior of millions of individual agents inside the system [43, 53, 59].

For example, transportation simulations are being used to address traffic congestion by evaluating proposed traffic management systems before implementing them [53]. This is a very important problem as traffic congestion cost

\$87.2 billion and required 2.8 billion gallons of extra fuel and 4.2 billion hours of extra time in the U.S. in 2007 alone [142]. Scientists also use behavioral simulations to model collective animal motion, such as that of locust swarms or fish schools [43, 59]. Understanding these phenomena is crucial, as they directly affect human food security [80].

Despite their huge importance, it remains difficult to develop large-scale behavioral simulations. Current systems either offer high-level programming abstractions, but are not scalable [76, 115, 124], or achieve scalability by hand-coding particular simulation models using low-level parallel frameworks, such as MPI [157].

In this chapter we propose to close this gap by bringing database-style programmability and scalability to agent-based simulations. Our core insight is that behavioral simulations may be regarded as computations driven by large *iterated spatial joins*. We introduce a new simulation engine, called BRACE (Big Red Agent-based Computation Engine), that extends the popular MapReduce dataflow programming model to these iterated computations. BRACE embodies a high-level programming language called BRASIL, which is compiled into an optimized shared-nothing, in-memory MapReduce runtime. The design of BRACE is motivated by the requirements of behavioral simulations, explained below.

### 3.1.1 Requirements for Simulation Platforms

**(1) Support for Complex Agent Interaction.** Behavioral simulations include frequent local interactions between individual entities in the simulation system, or

agents. In particular, agents may affect the behavior decisions of other agents, and multiple agents may issue concurrent writes to the same agent. A simulation framework should support a high degree of agent interaction without excessive synchronization or rollbacks. This precludes discrete event simulation engines or other approaches based on task parallelism and asynchronous message exchange.

**(2) Automatic Scalability.** Scientists need to scale their simulations to millions or billions of agents to accurately model phenomena such as city-wide traffic or swarms of insects [43, 59, 72]. These scales make it essential to use data parallelism to distribute agents across many nodes. This is complicated by the interaction between agents, which may require communication between several nodes. Rather than requiring scientists to write complex and error-prone parallel code, the platform should automatically distribute agents to achieve scalability.

**(3) High Performance.** Behavioral simulations are often extremely complex, involving sophisticated numerical computations and elaborate decision procedures. Much existing work on behavioral simulations is from the high-performance computing community, and they frequently resort to hand-coding specific simulations in a low-level language to achieve acceptable performance [72, 127]. A general purpose framework must be competitive with these hand-coded applications in order to gain acceptance.

**(4) Commodity Hardware.** Historically, many scientists have used large shared-memory supercomputer systems for their simulations. Such machines are tremendously expensive, and cannot scale beyond their original capacity. We believe that the next generation of simulation platforms will target shared-

nothing systems and will be deployed on local clusters or in the cloud on services such as Amazon's EC2 [30].

**(5) Simple Programming Model.** Domain scientists have shown their willingness to try simulation platforms that provide simple, high-level programming abstractions, even at some cost in performance and scalability [76, 115, 124]. Nevertheless, a behavioral simulation framework should provide an expressive and high-level programming model without sacrificing performance.

Our main contributions can be summarized as follows:

- We show how behavioral simulations can be abstracted in the state-effect pattern, a programming pattern we developed for scaling the number of non-player characters in computer games [158, 160]. This pattern allows for a high degree of concurrency among strongly interacting agents (Section 3.2).
- We show how MapReduce can be used to scale behavioral simulations expressed in the state-effect pattern across clusters. We abstract these simulations as iterated spatial joins and introduce a new main memory MapReduce runtime that incorporates optimizations motivated by the spatial properties of simulations (Section 3.3).
- We present a new scripting language for simulations that compiles into our MapReduce framework and allows for algebraic optimizations in mappers and reducers. This language hides all the complexities of modeling computations in MapReduce and parallel programming from domain scientists (Section 3.4).
- We perform an experimental evaluation with two real-world behavioral simulations that shows our system has nearly linear scale-up and single-

node performance that is comparable to a hand-coded simulation. (Section 3.5).

### 3.2 Behavioral Simulations in the State-Effect Pattern

Behavioral simulations model large numbers of individual agents that interact in a complex environment. Unlike scientific simulations that can be modeled as systems of equations, agents in a behavioral simulation can execute complex programs that include non-trivial control flow. In order to illustrate the functioning of behavioral simulations, we use a traffic simulation [163] and a fish school simulation [59] as running examples. Details on these simulations can be found in Appendix A.3.

**Concurrency in Behavioral Simulations.** Logically, all agents in a behavioral simulation execute actions concurrently. These actions usually result from interactions with other agents. In a fish school simulation, for example, a fish continuously updates its direction of movement based on the orientations of other fish within its visibility range. This implies that it is necessary to ensure consistency on multiple concurrent reads and writes among several fish.

Traditional discrete-event simulation platforms handle concurrency by either preempting or avoiding conflicts [49, 128, 62, 120, 170]. These systems implement variants of pessimistic or optimistic concurrency control schemes. The frequency of local interactions among agents in a behavioral simulation, however, introduces many conflicts. This leads to poor scalability due to either excessive synchronization or frequent rollbacks, depending on the technique

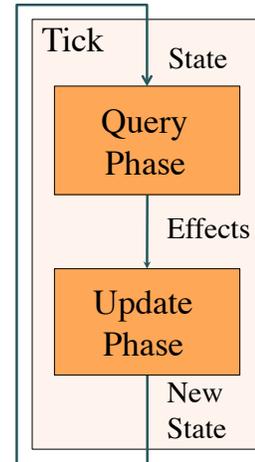
employed. Thus, these previous solutions, while programmable, do not allow us to scale behavioral simulations to large scenarios.

**The State-Effect Pattern.** In order to deal with concurrency in behavioral simulations, we observe that most behavioral simulations can be modeled under a similar structure, which we introduce below. Behavioral simulations use a time-stepped model in which time is discretized into “ticks” that represent the smallest time period of interest. Events that occur during the same tick are treated as *simultaneous* and can be reordered or parallelized. This means that an agent’s decisions cannot be based on previous actions made during the same tick. An agent can only read the state of the world as of the previous tick. For example, in the traffic simulation, each car inspects the positions and velocities of other cars as of the beginning of the tick in order to make lane changing decisions.

In previous work on scaling computer games, a model called the *state-effect pattern* has been proposed for this kind of time-stepped behavior [158, 160]. The basic idea is to separate read and write operations in order to limit the synchronization necessary between agents. In the state-effect pattern, the attributes of an agent are separated into *states* and *effects*, where states are public attributes that are updated only at tick boundaries, and effects are used for intermediate computations as agents interact. Therefore state attributes remain fixed during a tick, and only need to be synchronized at the end of each tick. Furthermore, each effect attribute has an associated decomposable and order-independent *combinator function* for combining multiple assignments, i.e., updates to this attribute, during a tick. This allows us to compute effects in parallel and combine the results without worrying about concurrent writes. For example, in the fish school simulation every fish agent has an orientation vector as an effect attribute. This

effect attribute uses vector addition as its combinator function, which is invoked on assignments of the orientations of nearby fish. Since vector addition is commutative, we can process these assignments in any order.

In the state-effect pattern, each tick is divided into two phases: the *query phase* and the *update phase*, as shown in the figure on the right. In the query phase, each agent queries the state of the world and assigns effect values, which are combined using the appropriate combinator function. To ensure the property that the actions during a tick are conceptually simultaneous, state variables are read-only during the query phase and effect variables are write-only.



In the update phase, each agent can read its state attributes and the effect attributes computed from the query phase; it uses these values to compute the new state attributes for the next tick. In the fish school simulation, the orientation effects computed during the query phase are read during the update phase to compute a fish's new velocity vector, represented as a state attribute. In order to ensure that updates do not conflict, each agent can only read and write its own attributes during the update phase. Hence, the only way that agents can communicate is through effect assignments in the query phase. We classify effect assignments into *local* and *non-local* assignments. In a local assignment, an agent updates one of its own effect attributes; in a non-local assignment, an agent writes to an effect attribute of a different agent.

**The Neighborhood Property.** The state-effect pattern addresses concurrency in a behavioral simulation by limiting the synchronization necessary during a tick. However, the synchronization at tick boundaries may still be very expensive, as

it is possible that every agent needs to query every other agent in the simulated world to compute its effects. We observe that this rarely occurs in practice. Most behavioral simulations are spatial, and simulated agents can only interact with other agents that are close according to a distance metric [96]. For example, a fish can only observe other fish within a limited distance  $\rho$  [59]. We will take advantage of this property of a large class of behavioral simulations to optimize communication within and across ticks.

### **3.3 MapReduce for Simulations**

In this section, we abstract behavioral simulations modeled according to the state-effect pattern and neighborhood property as computations driven by iterated spatial joins (Section 3.3.1). We proceed by showing how these joins can be expressed in the MapReduce framework (Section 3.3.2). We then propose a system called BRACE to process these joins efficiently (Section 3.3.3).

#### **3.3.1 Simulations as Iterated Spatial Joins**

In Section 3.2, we observed that behavioral simulations can be modeled by two important properties: the state-effect pattern and the neighborhood property. The state-effect pattern essentially characterizes behavioral simulations as iterated computations with two phases: a query phase in which agents inspect their environment to compute effects, and an update phase in which agents update their own state.

The neighborhood property introduces two important restrictions on these

phases, called visibility and reachability. We say that the *visible region* of an agent is the region of space containing agents that this agent can read from or assign effects to. An agent needs access to all the agents in its visible region to compute its query phase. Thus a simulation in which agents have small visible regions requires less communication than one with very large or unbounded visible regions. Similarly, we can define an agent's *reachable region* as the region that the agent can move to after the update phase. This is essentially a measure of how much the spatial distribution of agents can change between ticks. When agents have small reachable regions, a spatial partitioning of the agents is likely to remain balanced for several ticks. Frequently an agent's reachable region will be a subset of its visible region (an agent cannot move farther than it can see), but this is not required.

We observe that since agents only query other agents within their visible regions, processing a tick is similar to a *spatial self-join* [116]. We join each agent with the set of agents in its visible region and perform the query phase using only these agents. During the update phase, agents move to new positions within their reachable regions and we perform a new iteration of the join during the next tick. We will use this observation next to parallelize behavioral simulations efficiently in the MapReduce framework. As we will see in Section 3.4, the observation also enables us to apply efficient database indexing techniques, e.g., [67, 125].

effects	map <sub>1</sub> <sup>t</sup>	reduce <sub>1</sub> <sup>t</sup>	map <sub>2</sub> <sup>t</sup>	reduce <sub>2</sub> <sup>t</sup>	map <sub>1</sub> <sup>t+1</sup>
<b>local</b>	update <sup>t-1</sup> distribute <sup>t</sup>	query <sup>t</sup>	—	—	update <sup>t</sup> distribute <sup>t+1</sup>
<b>non-local</b>	update <sup>t-1</sup> distribute <sup>t</sup>	non-local effect <sup>t</sup>	—	effect aggregation <sup>t</sup>	update <sup>t</sup> distribute <sup>t+1</sup>

Table 3.1: The state-effect pattern in MapReduce

### 3.3.2 Iterated Spatial Joins in MapReduce

In this section, we show how to model spatial joins in MapReduce. A formal version of this model appears in Appendix A.1. MapReduce has been criticized for being inefficient at processing joins [51] and also inadequate for iterative computations without modification [69]. However, the spatial properties of simulations allow us to process them effectively without excessive communication. Our basic strategy uses the technique of Zhang et al. to compute a spatial join in MapReduce [168]. Each map task is responsible for spatially partitioning agents into a number of disjoint regions, and the reduce tasks join the agents using their visible regions.

The set of agents assigned to a particular partition is called that partition’s *owned set*. Note that we cannot process partitions completely independently because each agent needs access to its entire visible region, which may intersect several partitions. To address this, we define the *visible region* of a partition as the union of the visible regions of all points in the partition. The map task *replicates* each agent  $a$  to every partition that contains  $a$  in its visible region.

Table 3.1 shows how the phases of the state-effect pattern are assigned to map and reduce tasks. For simulations with only local effect assignments, a tick  $t$  begins when the first map task,  $\text{map}_1^t$ , assigns each agent to a partition

(distribute<sup>t</sup>). Each reducer is assigned a partition and receives every agent in its owned set as well as replicas of agents within its visible region. These are exactly the agents necessary to process the query phase of the owned set (query<sup>t</sup>). The reducer, reduce<sub>1</sub><sup>t</sup>, outputs a copy of each agent it owns after executing the query phase and updating the agent’s effects. The tick ends when the next map task, map<sub>1</sub><sup>t+1</sup>, executes the update phase (update<sup>t</sup>).

This two-step approach works for simulations that have only local effects, but does not handle non-local effect assignments. Recall that in a non-local effect assignment some agent  $a$  updates an effect in some other agent  $b$  within  $a$ ’s visible region. For example, if the fish simulation included predators, we could model a shark attack as a non-local effect assignment by a shark to a nearby fish. Non-local effects require communication during the query phase. We implement this communication using two MapReduce passes, as illustrated in Table 3.1. The first map task, map<sub>1</sub><sup>t</sup>, is the same as before. The first reduce task, reduce<sub>1</sub><sup>t</sup>, performs non-local effect assignments to its replicas (non-local effect<sup>t</sup>). These partially aggregated effect values are then distributed to the partitions that own them, where they are combined by the second reduce task, reduce<sub>2</sub><sup>t</sup>. This computes the final value for each aggregate (effect aggregation<sup>t</sup>). As before, the update phase is processed in the next map task, map<sub>1</sub><sup>t+1</sup>. Note that the second map task, map<sub>2</sub><sup>t</sup>, is only necessary for distribution, but does not perform any computation and can be eliminated in an implementation. We call this model map-reduce-reduce.

Our map-reduce-reduce model relies heavily on the neighborhood property. The number of replicas that each map task must create depends on the size of the agent’s visible regions, and the frequency that partitions change their

owned set depends on the size of their reachable regions. So, as shown in the next section, we rely on the fact that long-distance interactions are uncommon in behavioral simulations to optimize the communication pattern when processing the simulation.

### 3.3.3 The BRACE MapReduce Runtime

In this section we describe a MapReduce implementation that takes advantage of the state-effect pattern and the neighborhood property. We introduce BRACE, the Big Red Agent Computation Engine, our platform for scalable behavioral simulations. BRACE includes a MapReduce runtime especially optimized for the iterated spatial joins discussed in Section 3.3.1. We have developed a new system rather than using an existing MapReduce implementation such as Hadoop [81] because behavioral simulations have considerably different characteristics than traditional MapReduce applications such as search log analysis. The goal of BRACE is to process a very large number of ticks efficiently, and to avoid I/O or communication overhead while providing features such as fault tolerance. As we show below, our design allows us to apply a number of techniques developed in the HPC community [61, 71, 86, 145] and bridge them to a map-reduce model.

**Shared-Nothing, Main-Memory Architecture.** In behavioral simulations, we expect data volumes to be modest, so BRACE executes map and reduce tasks entirely in main memory. For example, a simulation with one million agents whose state and effect fields occupy 1 KB on average requires roughly 1 GB of main memory. Even larger simulations with orders of magnitude more agents

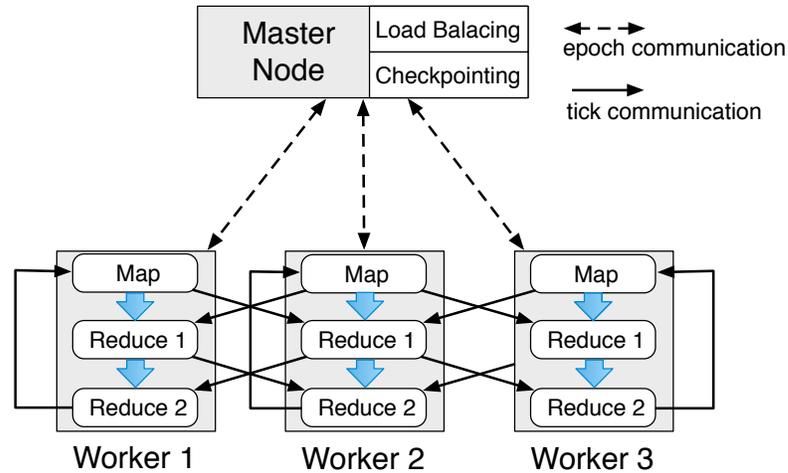


Figure 3.1: BRACE Architecture Overview

will still fit in the aggregate main memory of a cluster. Since the query phase is computationally expensive, partition sizes are limited by CPU cycles rather than main memory size.

Figure 3.1 shows the architecture of BRACE. As in typical MapReduce implementations, a master node is responsible for cluster coordination. However, BRACE’s master node only interacts with worker nodes every *epoch*, which corresponds to a fixed number of ticks. This amortizes overhead related to fault tolerance and load balancing. In addition, we allocate tasks of map-reduce-reduce iterations to workers so as to minimize communication overheads within and across iterations.

**Fault Tolerance.** Traditional MapReduce runtimes provide fault tolerance by storing output to a replicated file system and automatically restarting failed tasks. Since we expect ticks to be quite short and they are processed in main memory, it would be prohibitively expensive to write output to stable storage between every tick. Furthermore, since individual ticks are short, the benefit from restarting a task is likely to be small.

We employ epoch synchronization with the master to trigger *coordinated checkpoints* [71] of the main memory of the workers. As the master determines a pre-defined tick boundary for checkpointing, the workers can write their checkpoints independently without global synchronization. As we expect iterations to be short, failures are handled by re-execution of all iterations since the last checkpoint, a common technique in scientific simulations. In fact, we can leverage previous literature to tune the checkpointing interval to minimize the total expected runtime of the whole computation [61].

**Partitioning and Load Balancing.** As we have observed in Section 3.3.2, bounded reachability implies that a given spatial partitioning will remain effective for a number of map-reduce-reduce iterations. Our runtime uses that observation to keep data partitioning stable over time and re-evaluates it at epoch boundaries.

At the beginning of the simulation, the master computes a partitioning function based on the visible regions of the agents and then broadcasts this partitioning to the worker nodes. Each worker becomes responsible for one region of the partitioning. Agents change partitions slowly, but over time the overall spatial distribution may change quite dramatically. For example, the distribution of traffic on a road network is likely to be very different at morning rush hour than at evening rush hour. This would cause certain nodes to become overloaded if we used the same partitioning in both cases. To address this, the master periodically receives statistics from the workers about the number of agents in the owned region and the communication and processing costs. The master then decides on repartitioning by balancing the cost of redistribution with its expected benefit [86]. If the master decides to modify the partitioning, it

broadcasts the new partitioning to all workers. The workers then switch to the new partitioning at a specified epoch boundary.

**Collocation of Tasks.** Since simulations run for many iterations, it is important to minimize communication between tasks. We accomplish this by collocating the map and reduce tasks for a partition on the same node so that agents that do not switch partitions can be exchanged using shared memory rather than the network. Since agents have limited reachable regions, the owned set of each partition is likely to remain relatively stable across ticks, and so will remain on the same node. Agents still need to be replicated, but their primary copies do not have to be redistributed. This idea was previously explored by the Phoenix project for SMP systems [164] and the Map-Reduce-Merge project for individual joins [51], but it is particularly important for long-running behavioral simulations.

Figure 3.1 shows how collocation works when we allow non-local effect assignments. Solid arrows (both thick and thin) indicate the flow of agents during a tick. Each node processes a map task and two reduce tasks as described in Section 3.3.1. The map task replicates agents as necessary and sends them to the appropriate reduce tasks. The first-level reducers compute local effects and send non-local effects to the second-level reducers, which aggregate all (local and non-local) effects and send them to the map tasks for the next tick. Because of the neighborhood property, only agents that are near partition boundaries need to be replicated, and agents change partitions infrequently. Most messages are between tasks on the same node (as indicated by the thick blue arrows), enabling communication by shared memory rather than the network. This optimization exploits dependencies on the data between tasks, much in line

with schemes for relaxing communication in bulk-synchronous computations, e.g., [145].

When we put together all of the optimizations above, the runtime of BRACE resembles an optimized shared-nothing message-passing environment. However, BRACE still exposes an intuitive map-reduce programming abstraction as an interface to this runtime. As we show in the next section, this interface enables us to compile a high-level language for domain scientists into efficient map-reduce-reduce pipelines over BRACE.

### **3.4 Programming Agent Behavior**

In this section, we show how to offer a simple programming model for a domain scientist, targeting the last requirement of Section 3.1.1. MapReduce is set-based; a program describes how to process all of the elements in a collection. Simulation developers prefer to describe the behavior of their agents individually, and use message-passing techniques to communicate between agents. This type of programming is closer to the scientific models that describe agent behavior.

We introduce a new programming language – BRASIL, the Big Red Agent Simulation Language. BRASIL embodies agent centric programming with explicit support for the state-effect pattern, and performs further algebraic optimizations. It bridges the mental model of simulation developers and our MapReduce processing techniques for behavioral simulations. We provide an overview of the main features of BRASIL (Section 3.4.1) and describe algebraic optimization techniques that can be applied to our scripts (Section 3.4.2). For-

mal semantics for our language as well as the proofs of theorems in this section are provided in Appendix A.2.1.

### 3.4.1 Overview of BRASIL

BRASIL is an object-oriented language in which each object corresponds to an agent in the simulation. Agents in BRASIL are defined in a class file that looks superficially like Java. The programmer can specify fields, methods, and constructors, which can be either public or private. Unlike in Java, each field in a BRASIL class must be tagged as either *state* or *effect*. The BRASIL compiler enforces the read-write restrictions of the state-effect pattern over those fields as described in Section 3.2. Figure 3.2 is an example of a simple two-dimensional fish simulation, in which fish swim about randomly, avoiding each other with imaginary repulsion “forces”.

Recall that the state-effect pattern divides computation into query and update phases. In BRASIL, the query phase for an agent class is expressed by its `run()` method. State fields are read-only and effect assignments are aggregated using the functions specified at the effect field declarations. This is similar to aggregator variables in Sawzall [133]. In our fish simulation example, each fish repels nearby fish with a “force” inversely proportional to distance. The update phase is specified by *update rules* attached to the state field declarations. These rules can only read values of other fields in this agent. In our example, fish velocity vectors are updated based on the avoidance effects plus a random perturbation.

BRASIL has some important restrictions. First, it only supports iteration

```

class Fish {

    // The fish location
    public state float x : (x+vx); #range[-1,1];
    public state float y : (y+vy); #range[-1,1];

    // The latest fish velocity
    public state float vx : vx + rand() + avoidx / count * vx;
    public state float vy : vy + rand() + avoidy / count * vy;

    // Used to update our velocity
    private effect float avoidx : sum;
    private effect float avoidy : sum;
    private effect int count : sum;

    /** The query-phase for this fish. */
    public void run() {
        // Use "forces" to repel fish too close
        foreach(Fish p : Extent<Fish>) {
            p.avoidx <- 1 / abs(x - p.x);
            p.avoidy <- 1 / abs(y - p.y);
            p.count <- 1;
        }
    }
}

```

Figure 3.2: Class for Simple Fish Behavior

over a finite set or list via a `foreach`-loop. This eliminates unbounded looping, which is not available in algebraic database languages. Second, there is an interplay between `foreach`-loops and effects: effect variables can only be read outside of a `foreach`-loop, and all assignments within a `foreach`-loop are aggregated. This powerful restriction allows us to treat the entire program, not just the communication across map and reduce operations, as a dataflow query plan.

BRASIL also has a special programming construct to enforce the neighborhood property outlined in Section 3.2. Every state field that encodes spatial location is tagged with a visibility and reachability constraint. For example, the

constraint  $\text{range}[-1, 1]$  attached to the  $x$  field in Figure 3.2 means that each fish can inspect others whose  $x$  coordinate is at most  $[-1, 1]$  different to its own in the query phase, and can change its  $x$  coordinate by at most  $[-1, 1]$  in the update phase. The language runtime enforces these constraints, as described in Appendix A.2. Note that visibility has an interplay with agent references: it is possible that a reference to another agent is fine initially, but violates the visibility constraint as that other agent moves relative to the one holding the reference. For that reason, BRASIL employs *weak reference semantics* for agent references, similar to weak references in Java. If another agent moves outside of the visible region, then all references to it will resolve to NIL.

Note that this gives a different semantics for visibility than the one present in Section 3.3. BRASIL uses visibility to determine how agent references are resolved, while the BRACE runtime uses visibility to determine agent replication and communication. The BRASIL semantics are preferable for a developer, because they are easy to understand and hide MapReduce details. Fortunately, as we prove formally in Appendix A.2.1, these are equivalent.

**Theorem 1.** *The BRASIL semantics for visibility and the BRACE implementation of visibility are equivalent.*

While programming features in BRASIL may seem unusual, everything in the language follows from the state-effect pattern and neighborhood property. As these are natural properties of behavioral simulations, programming these simulations becomes relatively straightforward. Indeed, a large part of our traffic simulation in Section 3.5.1 was implemented by a domain scientist.

### 3.4.2 Optimization

We compile BRASIL into a well-understood dataflow language. In our previous work on computer games, we used the relational algebra to represent our data flow [158]. However, for distributed simulations, we have found the monad algebra [44, 101, 131, 150] – the theoretical foundation for XQuery [101] – to be a much more appropriate fit. In particular, the monad algebra has a `MAP` primitive for descending into the components of its nested data model; this makes it a much more natural companion to MapReduce than the relational algebra.

We present the formal translation to the monad algebra in Appendix A.2, together with several theorems regarding its usage in optimization. Most of these optimizations are the same as those that would be present in a relational algebra query plan: algebraic rewrites and automatic indexing. In fact, any monad algebra expression on flat tables can be converted to an equivalent relational algebra expression and *vice versa* [131]; rewrites and indexing on the relational form carry back into the monad algebra form. In particular, many of the techniques used by Pathfinder [152] to process XQuery with relational query plans apply to the monad algebra.

**Effect Inversion.** An important optimization that is unique to our framework involves eliminating non-local effects. When non-local effect assignments can be eliminated, we are able to process each tick with a single reduce pass instead of two (Section 3.3). Consider again the program of Figure 3.2. We may rewrite its `foreach`-loop as

```
foreach(Fish p : Extent<Fish>) {  
    avoidx <- 1 / abs(p.x - x);
```

```
    avoidy <- 1 / abs(p.y - y);  
    count <- 1;  
  }
```

This rewritten expression does not change the results of the simulation, but only assigns effects locally. We call this transformation *effect inversion*, and it is always possible at some cost in visibility range. Indeed, in Appendix A.2.2, we formally prove the following:

**Theorem 3.** *If the visibility constraint on a script is distance  $d$ , there is an equivalent script with a visibility constraint at most  $2d$  that uses only local effect assignments.*

Increasing the visibility bound increases the number of replicas required at each node. Hence this optimization eliminates one communication round at the cost of sending more information in the remaining round.

### 3.5 Experiments

In this section, we present experimental results using two distinct real-world behavioral simulation models we have coded using BRACE. We focus on the following: (i) We validate the effectiveness of the BRASIL optimizations introduced in Section 3.4.2. In fact, these optimizations allow us to approach the efficiency of hand-optimized simulation code (Section 3.5.2); (ii) We evaluate BRACE’s MapReduce runtime implementation over a cluster of machines. We measure simulation scale-up via spatial data partitioning as well as load balancing (Section 3.5.3).

### 3.5.1 Setup

**Implementation.** The prototype BRACE MapReduce runtime is implemented in C++ and uses MPI for inter-node communication. Our BRASIL compiler is written in Java and directly generates C++ code that can be compiled with the runtime. Our prototype includes a generic KD-tree based spatial index capability [36]. We use a simple rectilinear grid partitioning scheme, which assigns each grid cell to a separate slave node. A one-dimensional load balancer periodically receives statistics from the slave nodes, including computational load and number of owned agents; from these it heuristically computes a new partition trying to balance improved performance against estimated migration cost. Checkpointing is not yet integrated into BRACE’s implementation at the time of writing.

We plan to integrate more sophisticated algorithms for all these components in future work. But our current prototype already demonstrates good performance and automatic scaling of realistic behavioral simulations written in BRASIL.

**Simulation Workloads.** We have implemented realistic traffic and fish school simulations in BRASIL. The traffic simulation includes the lane-changing and acceleration models of the state-of-the-art, open-source MITSIM traffic simulator [163]. MITSIM is a single-node program, so we compare its performance against our BRASIL reimplementation of its model also running on a single node. We simulate a linear segment of highway, and scale-up the size of the problem by extending the length of the segment.

The fish simulation implements a recent model of information flow in an-

imal groups [59]. In this model the “ocean” is unbounded, and the spatial distribution of fish changes dramatically as “informed individuals” guide the movements of others in the school.

Neither of these simulations uses non-local effect assignments; therefore we need only a single reducer per node. To evaluate our effect inversion optimization, we modified the fish simulation to create a predator simulation that uses non-local assignments. It is similar in spirit to artificial society simulations [96]. Appendix A.3 describes these simulation models in more detail. We measure total simulation time in our single-node experiments and tick throughput (agent ticks per second) when scaling up over multiple nodes. In all measurements we eliminate start-up transients by discarding initial ticks until a stable tick rate is achieved.

**Hardware Setup.** We ran all of our experiments in the Cornell Web Lab cluster [32]. The cluster contains 60 nodes interconnected by a pair of 1 gigabit/sec Port Summit X450a Ethernet Switches. Each node has two Quad Core Intel Xeon, 2.66GHz, processors with 4MB cache each and 16 GB of main memory.

### 3.5.2 BRASIL Optimizations

We first compare the single-node performance of our traffic simulation to the single-node program MITSIM. The main optimization in this case is spatial indexing. For a meaningful comparison, we validate the aggregate traffic statistics produced by our BRASIL reimplementation against those produced by MITSIM. Details of our validation procedure appear in Appendix A.3.

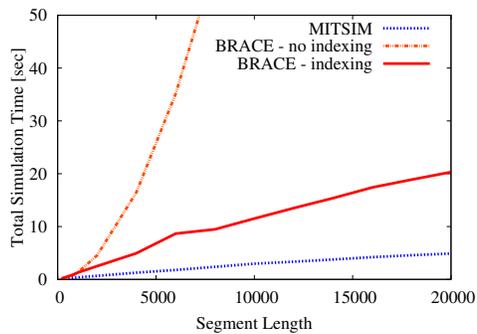


Figure 3.3: Traffic - Indexing

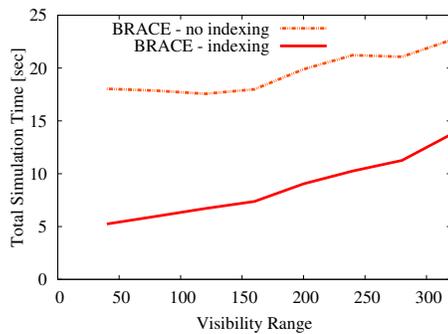


Figure 3.4: Fish - Indexing

Figure 3.3 compares the performance of MITSIM against BRACE using our BRASIL reimplementation of its model. Without spatial indexing, BRACE’s performance degrades quadratically with increasing segment length. This is expected: In this simulation, the number of agents grows linearly with segment length; and without indexing every vehicle enumerates and tests every other vehicle during each tick. With spatial indexing enabled, BRACE converts this behavior to an orthogonal range query, resulting in log-linear growth, as confirmed by Figure 3.3. BRACE’s spatial indexing achieves performance that is comparable, but inferior to MITSIM’s hand-coded nearest-neighbor implementation. Our optimization techniques generalize to nearest-neighbor indexing, and adding this to BRACE is planned future work. With this enhancement, we expect to achieve performance parity with MITSIM.

We observed similar log-linear versus quadratic performance when scaling up the number of agents in the fish simulation in a single node. We thus omit these results. When we increase the visibility range, however, the performance of the KD-tree indexing decreases, since more results are produced for each index probe (Figure 3.4). Still, indexing yields from two to three times improvement over a range of visibility values.

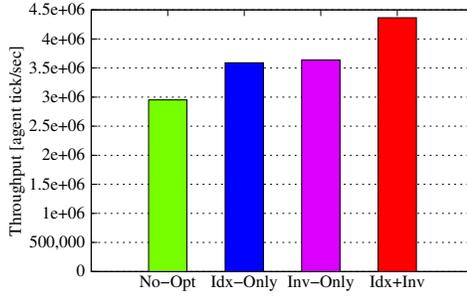


Figure 3.5: Predator - Effect Inversion

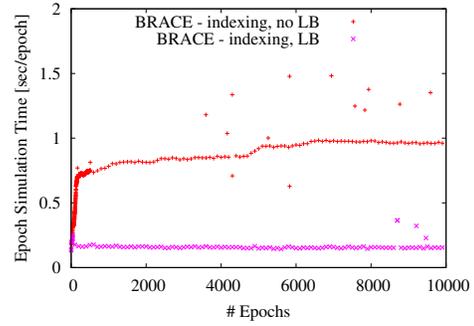


Figure 3.6: Fish - Load Balancing

In addition to indexing, we also measure the performance gain of eliminating non-local effect assignments through effect inversion. Only the predator simulation has non-local effect assignments, so we report results exclusively on this model. We run two versions of the predator simulation, one with non-local assignments and the other with non-local assignments eliminated by effect inversion. We run both scripts with and without KD-tree indexing enabled on 16 slave nodes, and with BRACE configured to have two reduce passes in the first case and only a single reduce pass in the second case. Our results are displayed in Figure 3.5. Effect inversion increases agent tick throughput from 3.59 million (Idx-Only) to 4.36 million (Idx+Inv) with KD-tree indexing enabled, and from 2.95 million (No-Opt) to 3.63 million (Inv-Only) with KD-tree indexing disabled. This represents an improvement of more than 20% in each case, demonstrating the importance of this optimization.

### 3.5.3 Scalability of the BRACE Runtime

We now explore the parallel performance of BRACE’s MapReduce runtime on the traffic and fish school simulations as we scale the number of slave nodes

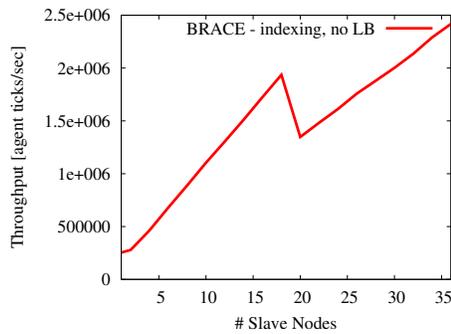


Figure 3.7: Traffic - Scalability

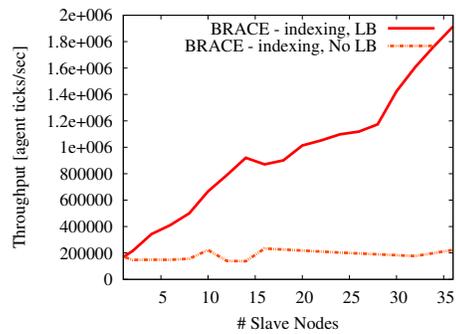


Figure 3.8: Fish - Scalability

from 1 to 36. The size of both simulations is scaled linearly with the number of slaves, so we measure scale-up rather than speed-up.

The traffic simulation represents a linear road segment with constant upstream traffic. As a result, the distribution on the segment is nearly uniform, and load is always balanced among the nodes. Therefore, throughput grows linearly with the number of nodes even if load balancing is disabled (Figure 3.7). The sudden drop around 20 nodes is an artifact of IP routing in the multi-switch configuration of the WebLab cluster on which we ran our experiments.

In the fish simulation, fish move in schools led by informed individuals [59]. In our experiment, there are two classes of informed individuals, trying to move in two different fixed directions. The spatial distribution of fish, and consequently the load on each slave node, changes over time. Figure 3.8 shows the scalability of this simulation with and without load balancing. Without load balancing, two fish schools eventually form in nodes at the extremes of simulated space, while the load at all other nodes falls to zero. With load balancing, partition grids are adjusted periodically to assign roughly the same number of fish to each node, so throughput increases linearly with the number of nodes.

Figure 3.6 confirms this, where still 16 slave nodes are used and we sample the time per epoch every 100 epochs, for readability, but also display all outliers. With load balancing enabled, the time per simulation epoch is essentially flat; with load balancing disabled, the epoch time gradually increases to a value that reflects all agents being simulated by only two nodes.

### 3.6 Conclusions

In this chapter we show how MapReduce can be used to scale behavioral simulations across clusters by abstracting these simulations as iterated spatial joins. To efficiently distribute these joins we leverage several properties of behavioral simulations to get a shared-nothing, in-memory MapReduce framework called BRACE, which exploits collocation of mappers and reducers to bound communication overhead. In addition, we present a new scripting language for our framework called BRASIL, which hides all the complexities of modeling computations in MapReduce and parallel programming from domain scientists. BRASIL scripts can be compiled into our MapReduce framework and allow for algebraic optimizations in mappers and reducers. We perform an experimental evaluation with two real-world behavioral simulations to show that BRACE has nearly linear scalability as well as single-node performance comparable to a hand-coded simulator.

## CHAPTER 4

# ENABLE ASYNCHRONOUS EXECUTION FOR LARGE-SCALE GRAPH PROCESSING

In this chapter we present GRACE, a new graph programming platform that separates application logic from execution policies in order to combine the easy programmability of the synchronous processing model with the high performance of asynchronous execution. Users of GRACE only needs to program their applications once using a synchronous iterative graph programming model, but can switch between various synchronous and asynchronous executions to get different parallel implementations without reimplementing their application logic. Our experiments with four large-scale real-world graph applications show that GRACE can achieve convergence rates and performance similar to that of completely general asynchronous execution engines while still maintaining the advantages of nearly linear parallel scalability inherited from a synchronous processing model.

### 4.1 Introduction

Graphs can capture complex data dependencies, and thus processing of graphs has become a key component in a wide range of applications, such as semi-supervised learning based on random graph walks [107], web search based on link analysis [41, 83], scene reconstruction based on Markov random fields [60] and social community detection based on label propagation [106], to name just a few examples. New applications such as social network analysis or 3D model construction over Internet-scale collections of images have produced graphs of unprecedented size, requiring us to scale graph processing to millions or even

billions of vertices. Due to this explosion in graph size, parallel processing is heavily used; for example, Crandall et al. describe the use of a 200-core Hadoop cluster to solve the structure-from-motion problem for constructing 3D models from large unstructured collections of images [60].

Recently, a number of graph processing frameworks have been proposed that allow domain experts to focus on the logic of their applications while the framework takes care of scaling the processing across many cores or machines [42, 68, 98, 119, 122, 135, 167, 169]. Most of these frameworks are based on two common properties of graph processing applications: first, many of these applications proceed iteratively, updating the graph data in rounds until a fixpoint is reached. Second, the computation within each iteration can be performed independently at the vertex level, and thus vertices can be processed individually in parallel. As a result, these graph processing frameworks commonly follow the bulk synchronous parallel (BSP) model of parallel processing [153]. The BSP model organizes computation into synchronous “ticks” (or “supersteps”) delimited by a global synchronization barrier. Vertices have local state but there is no shared state; all communication is done at the synchronization barriers by message passing. During a tick, each vertex independently receives messages that were sent by neighbor vertices during the previous tick, uses the received values to update its own state, and finally sends messages to adjacent vertices for processing during the following tick. This model ensures determinism and maximizes data parallelism within ticks, making it easy for users to design, program, test, and deploy parallel implementations of domain specific graph applications while achieving excellent parallel speedup and scaleup.

In contrast to the synchronous execution policy of the BSP model, *asyn-*

*chronous* execution policies do not have clean tick and independence properties, and generally communicate using shared state instead of—or in addition to—messages. Vertices can be processed in any order using the latest available values. Thus, there is no guarantee of isolation between updates of two vertices: vertices can read their neighbor’s states at will during their update procedure. Asynchronous execution has a big advantage for iterative graph processing [38, 114]: We can intelligently order the processing sequence of vertices to significantly accelerate convergence of the computation. Consider finding the shortest path between two vertices as an illustrative example: the shortest distance from the source vertex to the destination vertex can be computed iteratively by updating the distances from the source vertex to all other vertices. In this case, vertices with small distances are most likely to lead to the shortest path, and thus selectively expanding these vertices first can significantly accelerate convergence. This idea is more generally referred as the label-setting method for shortest/cheapest path problems on graphs, with Dijkstra’s classical method the most popular algorithm in this category. In addition, in asynchronous execution the most recent state of any vertex can be used directly by the next vertex that is scheduled for processing, instead of only using messages sent during the previous tick as in the BSP model. This can further increase the convergence rate since data updates can be incorporated as soon as they become available. For example, in belief propagation, directly using the most recent updates can significantly improve performance over synchronous update methods that have to wait until the end of each tick [70].

Although asynchronous execution policies can improve the convergence rate for graph processing applications, asynchronous parallel programs are much more difficult to write, debug, and test than synchronous programs. If

an asynchronous implementation does not output the expected result, it is difficult to locate the source of the problem: it could be the algorithm itself, a bug in the asynchronous implementation, or simply that the application does not converge to the same fixpoint under synchronous and asynchronous executions. Although several asynchronous graph processing platforms have been proposed which attempt to mitigate this problem by providing some asynchronous programming abstractions, their abstractions still require users to consider low-level concurrency issues [103, 113]. For example in GraphLab, the unit of calculation is a single update task over a vertex [113]. When an update task is scheduled, it computes based on whatever data is available on the vertex itself and possibly its neighbors. But since adjacent vertices can be scheduled simultaneously, users need to worry about read and write conflicts and choose from different consistency levels to avoid such conflicts themselves. In Galois, different processes can iterate over the vertices simultaneously, updating their data in an optimistic parallel manner [103]. Users then need to specify which method calls can safely be interleaved without leading to data races and how the effects of each method call can be undone when conflicts are detected. Such conflicts arise because general asynchronous execution models allow parallel threads to communicate at any time, not just at the tick boundaries. The resulting concurrent execution is highly dependent on process scheduling and is not deterministic. Thus, asynchronous parallel frameworks have to make concurrency issues explicit to the users.

For these reasons, a synchronous iterative model is clearly the programming model of choice due to its simplicity. Users can focus initially on “getting the application right,” and they can easily debug their code and reason about program correctness without having to worry about low-level concurrency issues.

Then, having gained confidence that their encoded graph application logic is bug-free, users would like to be able to migrate to asynchronous execution for better performance *without reimplementing their applications*; they should just be able to change the underlying execution policy in order to switch between synchronous and asynchronous execution.

Unfortunately, this crucially important development cycle — going from a simple synchronous specification of a graph processing application to a high-performance asynchronous execution — is not supported by existing frameworks. Indeed, it is hard to imagine switching from the message-passing communication style of a synchronous graph program to the shared-variable communication used in an asynchronous one without reimplementing the application. However, in this chapter we show such reimplementations are unnecessary: most of the benefit of asynchronous processing can be achieved in a message-passing setting by allowing users to explicitly relax certain constraints imposed on message delivery by the BSP model.

In this work, we combine synchronous programming with asynchronous execution for large-scale graph processing by cleanly separating application logic from execution policies. We have designed and implemented a large scale parallel iterative graph processing framework named GRACE, which exposes a synchronous iterative graph programming model to the users while enabling both synchronous and user-specified asynchronous execution policies. Our work makes the following three contributions:

- (1) We present GRACE, a general parallel graph processing framework that provides an iterative synchronous programming model for developers. The programming model captures data dependencies using messages passed be-

tween neighboring vertices like the BSP model (Section 4.3).

(2) We describe the parallel runtime of GRACE, which follows the BSP model for executing the coded application. At the same time GRACE allows users to flexibly specify their own (asynchronous) execution policies by explicitly relaxing data dependencies associated with messages in order to achieve fast convergence. By doing so GRACE maintains both fast convergence through customized (asynchronous) execution policies of the application and automatic scalability through the BSP model at run time (Section 4.4).

(3) We experiment with four large-scale real-world graph processing applications written in a shared-memory prototype implementation of GRACE (Section 4.5). Our experiments show that even though programs in GRACE are written synchronously, we can achieve convergence rates and performance similar to that of completely general asynchronous execution engines, while still maintaining nearly linear parallel speedup by following the BSP model to minimize concurrency control overheads (Section 4.6).

We begin our presentation by introducing iterative graph processing applications in Section 4.2 and conclude in Section 4.7. A discussion of related work can be found in Section 5.2

## 4.2 Iterative Graph Processing

*Iterative graph processing applications* are computations over graphs that update data in the graph in iterations or *ticks*. During each tick the data in the graph is updated, and the computation terminates after a fixed number of ticks have

been executed [60] or the computation has converged [77]. We use the belief propagation algorithm on pairwise Markov random fields (MRFs) as a running example to illustrate the computation patterns of an iterative graph processing application [132].

**Running Example: Belief Propagation on Pairwise MRF.** The pairwise MRF is a widely used undirected graphical model which can compactly represent complex probability distributions. Consider  $n$  discrete random variables  $\mathcal{X} = \{X_1, X_2, \dots, X_n\}$  taking on values  $X_i \in \Omega$ , where  $\Omega$  is the sample space.<sup>1</sup> A pairwise MRF is an undirected graph  $G(V, E)$  where vertices represent random variables and edges represent dependencies. Each vertex  $u$  is associated with the potential function  $\phi_u : \Omega \mapsto \mathbb{R}^+$  and each edge  $e_{u,v}$  is associated with the potential function  $\phi_{u,v} : \Omega \times \Omega \mapsto \mathbb{R}^+$ . The joint distribution is proportional to the product of the potential functions:

$$p(x_1, x_2, \dots, x_n) \propto \prod_{u \in V} \phi_u(x_u) \cdot \prod_{(u,v) \in E} \phi_{u,v}(x_u, x_v)$$

Computing the marginal distribution for a random variable (i.e., a vertex) is the core procedure for many learning and inference tasks in MRF. Belief propagation (BP), which works by repeatedly passing messages over the graph to calculate marginal distributions until the computation converges, is one of the most popular algorithms used for this task [70]. The message  $m_{u \rightarrow v}(x_v)$  sent from  $u$  to  $v$  is a distribution which encodes the “belief” about the value of  $X_v$  from  $X_u$ ’s perspective. Note that since MRFs are undirected graphs, there are two messages  $m_{u \rightarrow v}(x_v)$  and  $m_{v \rightarrow u}(x_u)$  for the edge  $e_{u,v}$ . In each tick, each vertex  $u$  first updates its own belief distribution  $b_u(x_u)$  according to its incoming

---

<sup>1</sup>In general, each random variable can have its own sample space. For simplicity of discussion, we assume that all the random variables have the same sample space.

messages  $m_{w \rightarrow u}(x_u)$ :

$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \rightarrow u}(x_u) \quad (4.1)$$

This distribution indicates the current belief about  $X_u$ 's value. The message  $m_{u \rightarrow v}(x_v)$  for its outgoing edge  $e_{u,v}$  can then be computed based on its updated belief distribution:

$$m_{u \rightarrow v}(x_v) \propto \sum_{x_u \in \Omega} \phi_{u,v}(x_u, x_v) \cdot \frac{b_u(x_u)}{m_{v \rightarrow u}(x_u)} \quad (4.2)$$

Each belief distribution can be represented as a vector, residing in some *belief space*  $\mathcal{B} \subset (\mathbb{R}^+)^{|\Omega|}$ ; we denote all the  $|V|$  beliefs as  $\mathbf{b} \in \mathcal{B}^{|V|}$ . Hence the update procedure of Equation (4.2) for a vertex  $v$  can be viewed as a mapping  $f_v : \mathcal{B}^{|V|} \mapsto \mathcal{B}$ , which defines the belief of vertex  $v$  as a function of the beliefs of all the vertices in the graph (though it actually depends only on the vertices adjacent to  $v$ ). Let  $\mathbf{f}(\mathbf{b}) = (f_{v_1}(\mathbf{b}_1), f_{v_2}(\mathbf{b}_2), \dots, f_{v_{|V|}}(\mathbf{b}_{|V|}))$ , the goal is to find the fixpoint  $\mathbf{b}^*$  such that  $\mathbf{f}(\mathbf{b}^*) = \mathbf{b}^*$ . At the fixpoint, the marginal distribution of any vertex  $v$  is the same as its belief. Thus BP can be treated as a way of organizing the “global” computation of marginal beliefs in terms of local computation. For graphs containing cycles, BP is not guaranteed to converge, but it has been applied with extensive empirical success in many applications [126].

In the original BP algorithm, all the belief distributions are updated simultaneously in each tick using the messages sent in the previous tick. Algorithm 1 shows this procedure, which simply updates the belief distribution on each vertex and calculates its outgoing messages. The algorithm terminates when the belief of each vertex  $u$  stops changing, in practice when  $\|b_u^{(t)} - b_u^{(t-1)}\| < \epsilon$  for some small  $\epsilon$ .

Although the original BP algorithm is simple to understand and implement,

---

**Algorithm 1: Original BP Algorithm**

---

```
1 Initialize  $b_u^{(0)}$  as  $\phi_u$  for all  $u \in V$  ;
2 Calculate the message  $m_{u \rightarrow v}^{(0)}$  using  $b_u^{(0)}$  according to Eq. 4.2 for all  $e_{u,v} \in E$  ;
3 Initialize  $t = 0$  ;
4 repeat
5    $t = t + 1$  ;
6   foreach  $u \in V$  do
7     Calculate  $b_u^{(t)}$  using  $m_{w \rightarrow u}^{(t-1)}$  according to Eq. 4.1 ;
8     foreach outgoing edge  $e_{u,v}$  of  $u$  do
9       Calculate  $m_{u \rightarrow v}^{(t)}$  using  $b_u^{(t)}$  according to Eq. 4.2 ;
10    end
11  end
12 until  $\forall u \in V, \|b_u^{(t)} - b_u^{(t-1)}\| \leq \epsilon$  ;
```

---

it can be very inefficient. One important reason is that it is effectively only using messages from the previous tick. A well-known empirical observation is that when vertices are updated sequentially using the latest available messages from their neighbors, the resulting asynchronous BP will generally approach the fixpoint with fewer updates than the original variant [70]. In addition, those vertices whose incoming messages changed drastically will be more “eager” to update their outgoing messages in order to reach the fixpoint. Therefore, as suggested by Gonzalez et al., a good sequential update ordering is to update the vertex that has the largest “residual,” where the residual of a message is defined as the difference between its current value and its last used value, and the residual of a vertex is defined as the maximum residual of its received messages [77]. The resulting residual BP algorithm is illustrated in Algorithm 2, where during

---

**Algorithm 2: Residual BP Algorithm**

---

- 1 Initialize  $b_u^{\text{new}}$  and  $b_u^{\text{old}}$  as  $\phi_u$  for all  $u \in V$  ;
  - 2 Initialize  $m_{u \rightarrow v}^{\text{old}}$  as uniform distribution for all  $e_{u,v} \in E$  ;
  - 3 Calculate message  $m_{u \rightarrow v}^{\text{new}}$  using  $b_u^{\text{new}}$  according to Eq. 4.2 for all  $e_{u,v} \in E$  ;
  - 4 **repeat**
  - 5      $u = \arg \max_v (\max_{(w,v) \in E} \|m_{w \rightarrow v}^{\text{new}} - m_{w \rightarrow v}^{\text{old}}\|)$  ;
  - 6     Set  $b_u^{\text{old}}$  to be  $b_u^{\text{new}}$  ;
  - 7     Calculate  $b_u^{\text{new}}$  using  $m_{w \rightarrow u}^{\text{new}}$  according to Eq. 4.1 ;
  - 8     **foreach** outgoing edge  $e_{u,v}$  of  $u$  **do**
  - 9         Set  $m_{u \rightarrow v}^{\text{old}}$  to be  $m_{u \rightarrow v}^{\text{new}}$  ;
  - 10         Calculate  $m_{u \rightarrow v}^{\text{new}}$  using  $b_u^{\text{new}}$  according to Eq. 4.2 ;
  - 11     **end**
  - 12 **until**  $\forall u \in V, \|b_u^{\text{new}} - b_u^{\text{old}}\| \leq \epsilon$  ;
- 

each tick the vertex with the largest residual is chosen to be processed and then its outgoing messages are updated.

Comparing Algorithm 1 and 2, we find that their core computational logic is actually the same: they are both based on iterative application of Equations 4.1 and 4.2. The only difference lies in how this computational logic is executed. In the original BP algorithm, all vertices simultaneously apply these two equations to update their beliefs and outgoing messages using the messages received from the previous tick. In the residual BP algorithm, however, vertices are updated sequentially using the most recent messages while the order is based on message residuals. The independent nature of the message calculation suggests that the original BP algorithm can be easily parallelized: since in each tick, each ver-

tex only needs to read its incoming messages from the previous tick, the vertex computations will be completely independent as long as all messages from the previous tick are guaranteed to be available at the beginning of each tick. Such a computational pattern can be naturally expressed in the BSP model for parallel processing, where graph vertices are partitioned and distributed to multiple processes for local computation during each tick. These processes do not need to communicate with each other until synchronization at the end of the tick, which is used to make sure every message for the next tick has been computed, sent, and received. Such a parallel program written in the BSP model can automatically avoid non-deterministic access to shared data, thus achieving both programming simplicity and scalability.

On the other hand, in the residual BP algorithm only one vertex at a time is selected and updated based on Equation 4.1 and 4.2. In each tick it selects the vertex  $v$  with the maximum residual:

$$\arg \max_v \max_{(w,v) \in E} \|m_{w \rightarrow v}^{\text{new}} - m_{w \rightarrow v}^{\text{old}}\|$$

which naively would require  $O(|E|)$  time. To find the vertex with maximal residual more efficiently, a priority queue could be employed with the vertex priority defined as the maximum of residuals of the vertex's incoming messages [70]. During each tick, the vertex with the highest priority is selected to update its outgoing messages, which will then update the priorities of the receivers correspondingly. Note that since access to the priority queue has to be serialized across processes, the resulted residual BP algorithm no longer fits in the BSP model and thus cannot be parallelized easily. Sophisticated concurrency control is required to prevent multiple processes from updating neighboring vertices simultaneously. As a result, despite the great success of the serial residual BP algorithm, researchers have reported poor parallel speedup [77].

The above observations can also be found in many other graph processing applications. For instance, both Bellman-Ford algorithm and Dijkstra's algorithm can be used to solve the shortest path problem with non-negative edge weights. These two algorithms share the same essential computational logic: pick a vertex and update its neighbors' distances based on its own distance. The difference lies only in the mechanisms of selecting such vertices: in Bellman-Ford algorithm, all vertices are processed synchronously in each tick to update their neighbors, whereas in Dijkstra's algorithm only the unvisited vertex with the smallest tentative distance is selected. On a single processor, Dijkstra's algorithm is usually preferred since it can converge faster. However, since the algorithm requires processing only one vertex with the smallest distance at a time, it cannot be parallelized as easily as Bellman-Ford. Similar examples also include the push-relabel algorithm and the relabel-to-front algorithm for the network max-flow problem [100].

In general, although asynchronous algorithms can result in faster convergence and hence better performance, they are (1) more difficult to program than synchronous algorithms, and (2) harder to scale up through parallelism due to their inherent sequential ordering of computation. To avoid these difficulties, graph processing application developers want to start with a simple synchronous implementation first; they can thoroughly debug and test this implementation and try it out on sample data. Then once an application developer has gained confidence that her synchronous implementation is correct, she can carefully switch to an asynchronous execution to reap the benefits of better performance from faster convergence while maintaining similar speedup — as long as the asynchronous execution generates identical (or at least acceptable) results compared to the synchronous implementation. To the best of our knowledge,

GRACE is the first effort to make this development cycle a reality.

### 4.3 Programming Model

Unlike most graph query systems or graph databases which tend to apply declarative (SQL or Datalog-like) programming languages to interactively execute graph queries [6, 7, 10], GRACE follows batch-style graph programming frameworks (e.g., PEGASUS [98] and Pregel [119]) to insulate users from low level details by providing a high level representation for graph data and letting users specify an application as a set of individual vertex update procedures. In addition, GRACE lets users explicitly define sparse data dependencies as messages between adjacent vertices. We explain this programming model in detail in this section.

#### 4.3.1 Graph Model

GRACE encodes application data as a directed graph  $G(V, E)$ . Each vertex  $v \in V$  is assigned a unique identifier and each edge  $e_{u \rightarrow v} \in E$  is uniquely identified by its source vertex  $u$  and destination vertex  $v$ . To define an undirected graph such as an MRF, each undirected edge is represented by two edges with opposite directions. Users can define arbitrary attributes on the vertices, whose values are modifiable during the computation. Edges can also have user-defined attributes, but these are read-only and initialized on construction of the edge. We denote the attribute vector associated with the vertex  $v$  at tick  $t$  by  $S_v^t$  and the attribute vector associated with edge  $e_{u \rightarrow v}$  by  $S_{u \rightarrow v}$ . Since  $S_{u \rightarrow v}$  will not be changed

```

class Vertex {
    Distribution potent; // Predefined potential
    Distribution belief; // Current belief
};

class Edge {
    Distribution potent; // Predefined potential
};

class Msg {
    Distribution belief; // Belief of the receiver
                        // from sender's perspective
};

```

Figure 4.1: Graph Data for BP

during the computation, the state of the graph data at tick  $t$  is determined by  $S^t = (S_{v_1}^t, S_{v_2}^t \dots, S_{v_{|V|}}^t)$ . At each tick  $t$ , each vertex  $v$  can send at most one message  $M_{v \rightarrow w}^t$  on each of its outgoing edges. Message attributes can be specified by the users; they are read-only and must be initialized when the message is created.

As shown in Figure 4.1, for Belief Propagation,  $S_v^t$  stores vertex  $v$ 's predefined potential function  $\phi_v(x_v)$ , as well as the belief distribution  $b_v(x_v)$  estimated at tick  $t$ .  $S_{u \rightarrow v}$  stores the predefined potential function  $\phi_{u,v}(x_u, x_v)$ . Note that the belief distributions are the only modifiable graph data in BP. As for messages,  $M_{u \rightarrow v}^t$  stores the belief distribution  $m_{u \rightarrow v}^t$  about the value of  $X_v$  from  $X_u$ 's perspective at tick  $t$ .

### 4.3.2 Iterative Computation

In GRACE, computation proceeds by updating vertices iteratively based on messages. At each tick, vertices receive messages through all incoming edges,

update their local data values, and propagate newly created messages through all outgoing edges. Thus each vertex updates only its own data. This update logic is captured by the `Proceed` function with the following signature:

```
List<Message> Proceed(List<Message> msgs)
```

Whenever this function is triggered for some vertex  $u$ , it will be processed by updating the data of vertex  $u$  based on the received messages `msgs` and by returning a set of newly computed outgoing messages, one for each outgoing edge. Note that at the beginning, the vertices must first send messages before receiving any. Therefore the received messages parameter for the first invocation of this function will be empty. When executed synchronously, in every tick each vertex invokes its `Proceed` function once, and hence expects to receive a message from each of its incoming edges.

As shown in Figure 4.2, we can easily implement the `Proceed` function for BP, which updates the belief following Equation 4.1 (lines 2 - 6) and computes the new message for each outgoing edge based on the updated belief and the edge's potential function (lines 7 - 13); we use `Msg[e]` to denote the message received on edge  $e$ . Similarly, `outMsg[e]` is the message sent out on edge  $e$ . The `computeMsg` function computes the message based on Equation (4.2). If a vertex finds its belief does not change much, it will vote for halt (lines 14 - 15). When all the vertices have voted for halt during a tick, the computation terminates. This voting-for-halt mechanism actually distributes the work of checking the termination criterion to the vertices. Therefore at the tick boundary if we observe that all the vertices have voted for halt, we can terminate the computation.

Note that with this synchronous programming model, data dependencies

```

1 List<Msg> Proceed(List<Msg> msgs) {
2     // Compute new belief from received messages
3     Distribution newBelief = potent;
4     for (Msg m in msgs) {
5         newBelief = times(newBelief, m.belief);
6     }
7
8     // Compute and send out messages
9     List<Msg> outMsgs(outDegree);
10    for (Edge e in outgoingEdges) {
11        Distribution msgBelief;
12        msgBelief = divide(newBelief, Msg[e]);
13        outMsg[e] = computeMsg(msgBelief, e.potent);
14    }
15
16    // Vote to terminate upon convergence
17    if (L1(newBelief, belief) < eps) voteHalt();
18
19    // Update belief and send out messages
20    belief = newBelief;
21    return outMsgs;
22 }

```

Figure 4.2: Update logic for BP

are implicitly encoded in messages: a vertex should only proceed to the next tick when it has received all the incoming messages in the previous tick. As we will discuss in the next section, we can relax this condition in various ways in order to execute graph applications not only synchronously, but also asynchronously.

#### 4.4 Asynchronous Execution in BSP

Given that our iterative graph applications are written in the BSP model, it is natural to process these applications in the same model by executing them in

ticks: within each tick, we process vertices independently in parallel through the user-specified `Proceed` function; at the end of each tick we introduce a synchronization barrier. A key observation in the design of the GRACE runtime is that asynchronous execution policies can also be implemented with a BSP-style runtime: an underlying BSP model in the platform does not necessarily force synchronous BSP-style execution policies for the application.

We first describe in Section 4.4.1 the semantics that require to be specified when relaxing synchronous execution properties in order to enable asynchronous execution policies, and then describe how they should be specified in Section 4.4.2.

#### 4.4.1 Relaxing Synchrony Properties

In the BSP model, the graph computation is organized into ticks. For synchronous execution, each vertex reads all available messages, computes a new state and sends messages that will be available in the next tick. This satisfies the following two properties:

**Isolation.** Computation within each tick is performed independently at the vertex level. In other words, within the same tick newly generated messages from any vertices will not be seen by the `Proceed` functions of other vertices.

**Consistency.** A vertex should be processed if and only if all its incoming messages have been received. In other words, a vertex' `Proceed` function should be triggered at the tick when all its received messages from incoming edges have been made available at the beginning of that tick.

By ensuring both isolation and consistency in the BSP model, all vertices are processed independently in each iteration using their received messages from the previous iteration. Hence data dependencies implicitly encoded in messages are respected. In asynchronous execution, however, we can relax isolation or consistency (or both) in order to achieve faster convergence. By relaxing consistency, we allow a vertex to invoke its `Proceed` function before all the incoming messages have arrived; for example, we can schedule a vertex that has received only a single, but important message. By relaxing isolation, we allow messages generated earlier in an iteration to be seen by later vertex update procedures. Therefore, the invocation order of vertex update procedures can make a substantial difference; for example, we may process vertices with “important” messages early in order to generate their outgoing messages and make these messages visible to other vertices within the same tick, where the message importance is intended to capture the messages contribution to the convergence of the iterative computation. In either case, an edge  $e_{u,v}$  could have multiple unread messages if the destination vertex  $v$  has not been scheduled for some time or it could have no unread message at all if the source vertex  $u$  has not been scheduled since the last time  $v$  was processed. Hence we must decide which message on each of the incoming edges should be used when we are going to update the vertex. For example, an option would be to use the latest message when there are “stale” messages or to use the last consumed message if no new message has been received.

By relaxing isolation and/or consistency properties, we can efficiently simulate asynchronous execution while running with a BSP model underneath. Combining different choices about how to relax these properties of the BSP model results in various execution policies, which may result in different con-

vergence rates. We now describe the customizable execution interface of the GRACE runtime which enables this flexibility.

#### 4.4.2 Customizable Execution Interface

**Vertex Scheduling.** If users decide to relax the consistency property of the BSP model, they can determine the set of vertices to be processed as well as the order in which these vertices are processed within a tick. In order to support such flexible vertex scheduling, each vertex maintains a dynamic priority value. This value, called the *scheduling priority* of the vertex, can be updated upon receiving a new message. Specifically, whenever a vertex receives a message the execution scheduler will trigger the following function:

```
void OnRecvMsg(Edge e, Message msg)
```

In this function, users can update the scheduling priority of the receiver vertex by aggregating the importance of the received message. Then at the start of each tick, the following function will be triggered:

```
void OnPrepare(List<Vertex> vertices)
```

In `OnPrepare`, users can access the complete global vertex list and select a subset (or the whole set) of the vertices to be processed for this tick. We call this subset the *scheduled vertices* of the tick. To do this, users can call `Schedule` on any single vertex to schedule the vertex, or call `ScheduleAll` with a specified predicate to schedule all vertices that satisfy the predicate. They can also specify the order in which the scheduled vertices are going to be processed; currently, GRACE allows scheduled vertices to be processed following either the order determined by their scheduling priorities or by their vertex ids. For example,

users can schedule the set of vertices with high priority values; they can also use the priority just as a boolean indicating whether or not the vertex should be scheduled; they can also simply ignore the scheduling priority and schedule all vertices in order to achieve consistency. The framework does not force them to consider certain prioritized execution policies but provides the flexibility of vertex scheduling through the use of this scheduling priority.

**Message Selection.** If users decide to relax the isolation property of the BSP model, vertices are no longer restricted to using only messages sent in the previous tick. Users can specify which message to use on each of the vertex's incoming edges when processing the vertex in `Proceed`. Since every vertex is processed and hence sends messages at tick 0, every edge will receive at least one message during the computation. Users can specify this message selection logic in the following function:

```
Msg OnSelectMsg(Edge e)
```

This function will be triggered by the scheduler on each incoming edge of a vertex before the vertex gets processed in the `Proceed` function. The return message `msg` will then be put in the parameters of the corresponding edge upon calling `Proceed`. In the `OnSelectMsg` function, users can get either 1) the last received message, 2) the selected message in the last `Proceed` function call, or 3) the most recently received message up to the previous tick. Users have to choose the last option in order to preserve isolation.

In general users may want to get any “unconsumed” messages or “combine” multiple messages into a single message that is passed to the `Proceed` function, we are aware of this requirement and plan to support this feature in future work. For now we restrict users to choosing messages from only the

above three options so that we can effectively garbage collect old messages and hence only maintain a small number of received messages for each edge. Note that during this procedure new messages could arrive simultaneously, and the runtime needs to guarantee that repeated calls to get these messages within a single `OnSelectMsg` function will return the same message objects.

### 4.4.3 Original and Residual BP: An Example

By relaxing the consistency and isolation properties of the BSP model, users can design very flexible execution policies by instantiating the `OnRecvMsg`, `OnSelectMsg` and `OnPrepare` functions. Taking the BP example, if we want to execute the original synchronous BP algorithm, we can implement these three functions as in Figure 4.3. Since every vertex is going to be scheduled at every tick, `OnRecvMsg` does not need to do anything for updating scheduling priority. In addition, since every edge will receive a message at each tick, `OnSelectMsg` can return the message received from the previous tick. Finally, in `OnPrepare` we simply schedule all the vertices. By doing this both consistency and isolation are preserved and we get as a result a synchronous BP program.

If we want to apply the asynchronous residual BP style execution, we need to relax both consistency and isolation. Therefore we can instead implement these three functions as in Figure 4.4. In this case we use the maximum of the residuals of the incoming messages as the scheduling priority of a vertex. In `OnRecvMsg`, we first get the belief distribution of the last received message. Next we compute the residual of the newly received message as the L1 difference between its belief distribution with that of the last received message. This residual

```

1 void OnRecvMsg(Edge e, Message msg) {
2     // Do nothing to update priority
3     // since every vertex will be scheduled
4 }

1 Msg OnSelectMsg(Edge e) {
2     return GetPrevRecvMsg(e);
3 }

1 void OnPrepare(List<Vertex> vertices) {
2     ScheduleAll(Everyone);
3 }

```

Figure 4.3: Synchronous Original Execution for BP

is then used to update the scheduling priority of the vertex via the sum aggregation function. In `OnSelectMsg`, we simply return the most recently received message. For `OnPrepare`, we schedule approximately  $r \cdot |V|$  of the vertices with high scheduling priorities by first picking a threshold from a sorted sample of vertices and then calling `ScheduleAll` to schedule all vertices whose priority values are larger than this threshold.<sup>2</sup>

## 4.5 Runtime Implementation

We now describe the design and implementation of the parallel processing runtime of GRACE. The goal of the runtime is to efficiently support vertex scheduling and message selection as we discussed in Section 4.4. Most asynchronous processing frameworks have to make the scheduling decision of which vertex to

---

<sup>2</sup>To strictly follow the residual BP algorithm, we can only schedule one vertex with the highest scheduling priority for each tick; although this can also be achieved in `OnPrepare` by first sorting the vertices based on their priorities and then choose only the vertex with the highest priority value, we choose not to demonstrate this execution policy since it is not efficient for parallel processing.

```

1 void OnRecvMsg(Edge e, Message msg) {
2     Distn lastBelief = GetLastUsedMsg(e).belief;
3     float residual = L1(newBelief, msg.belief);
4     UpdatePriority(GetRecVtx(e), residual, sum);
5 }

1 Msg OnSelectMsg(Edge e) {
2     return GetLastRecvMsg(e);
3 }

1 void OnPrepare(List<Vertex> vertices) {
2     List<Vertex> samples = Sample(vertices, m);
3     Sort(samples, Vertex.priority, operator >);
4     float threshold = samples[r * m].priority;
5     ScheduleAll(PriorGreaterThan(threshold));
6 }

```

Figure 4.4: Asynchronous Residual Execution for BP

be processed next. However, because GRACE follows the BSP model, it schedules a set of vertices to be processed for the next tick at the barrier instead of repeatedly schedule one vertex at a time. As we will show below, this allows both fast convergence and high scalability.

### 4.5.1 Shared-Memory Architecture

For many large-scale graph processing applications, after appropriate preprocessing, the data necessary for computation consists of a few hundred gigabytes or less and thus fits into a single shared-memory workstation. In addition, although the BSP model was originally designed for distributed-memory systems, it has also been shown useful for shared-memory multicore systems [154, 166]. As a result, while the GRACE programming abstraction and its customizable execution interface are intended for both shared-memory and

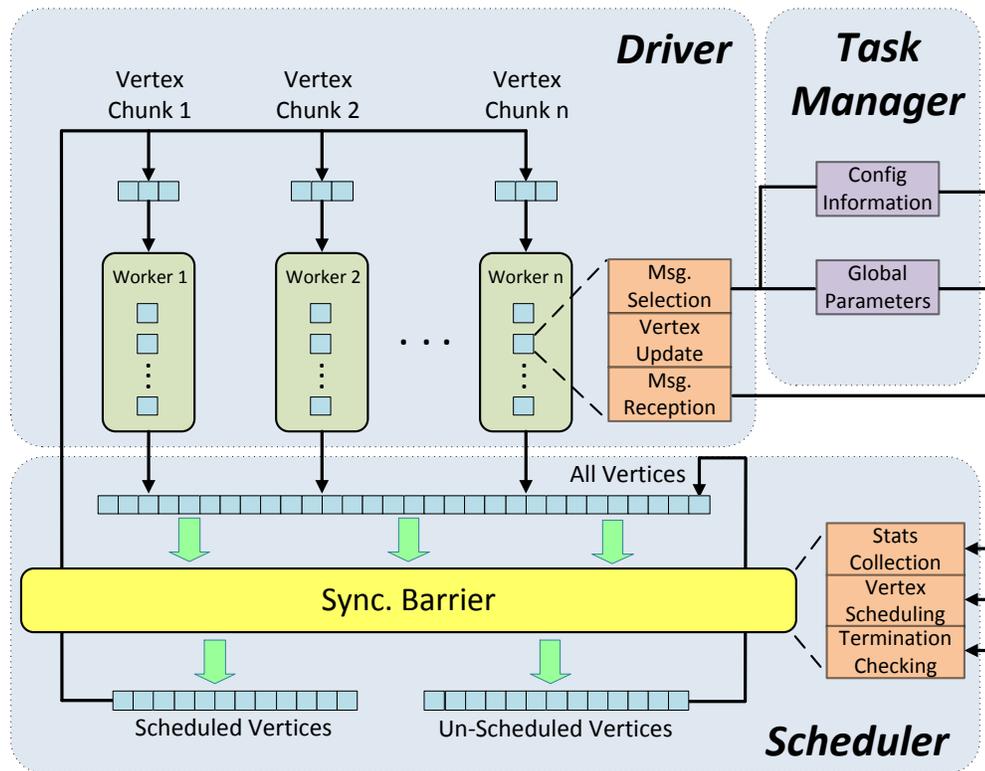


Figure 4.5: Data Flow in GRACE Runtime

distributed memory environments, we decided to build the first prototype of the GRACE runtime on a shared-memory architecture. In the future we plan to extend it to a distributed environment for even larger data sets.

The architecture of the runtime is shown in Figure 4.5. As in a typical BSP implementation, GRACE executes an application as a group of worker threads, and these threads are coordinated by a driver thread. The runtime also contains a scheduler, which is in charge of scheduling the set of vertices to be processed in each tick. Each run of the application is treated as a *task*, with its config information such as the number of worker threads and task stop criterion stored in the task manager. All vertices, together with their edges and the received messages

on each incoming edge, are stored in a global *vertex list* in shared memory. Although messages are logically stored on the edges, in the implementation they are actually located in the data space of the receiving vertex to improve locality.

### 4.5.2 Batch Scheduling

Recall that there is a synchronization barrier at the end of each tick, where the user-specified `OnPrepare` function is triggered. This barrier exists not only for enforcing determinism but also for users to access the global vertex list of the graph. Within this `OnPrepare` function users can read and write any vertex data. For example, they can collect graph statistics by aggregating over the global vertex list, change global variables and parameters of the application, and finalize the set of scheduled vertices for the next tick. In addition to the user-specified data attributes, the runtime maintains a *scheduling bit* for each vertex. When a vertex is scheduled inside `OnPrepare` through the `Schedule` and `ScheduleAll` function calls, its scheduling bit is set. Thus, at the end of the `OnPrepare` function, a subset of vertices have their scheduling bits set, indicating that they should be processed in the next tick.

### 4.5.3 Iterative Processing

As discussed above, at the end of each tick a set of scheduled vertices is selected. During the next tick these scheduled vertices are assigned to worker threads and processed in parallel. In order to decide the assignment in an efficient and load-balanced way, the driver partition the global vertex list into fixed-size chunks.

Those chunks are allocated to worker threads in a round robin manner during the tick. Each worker thread iterates over the vertices of the chunk following either a fixed order or the order specified by vertices' scheduling priority values, selecting and processing vertices whose scheduling bits are set. After a worker thread has finished processing its current chunk, the next free chunk is allocated to it.

When no more free chunks are available to a worker thread, the thread moves on to the tick barrier. Once a worker thread has moved to the tick barrier, there are no more free chunks available to be assigned later to other threads in the current tick. Therefore, the earliest thread to arrive the synchronization barrier will wait for the processing time of a single chunk in the worst case. When every thread has arrived at the tick barrier, we are guaranteed that all the scheduled vertices have been processed. The scheduler then checks if the computation can be stopped and triggers `OnPrepare` to let users collect statistics and schedule vertices for the next tick.

In the current GRACE runtime, computation can be terminated either after a user-specified number of ticks, or when a fixpoint has been reached. A fixpoint is considered to be reached if the set of scheduled vertices is empty at the beginning of a tick or if all the vertices have voted to halt within a tick. To efficiently check if all the vertices have voted to halt in the tick, each worker thread maintains a bit indicating if all the vertices it has processed so far have voted to halt. At the tick barrier, the runtime only needs to check  $M$  bits to see if every vertex has voted to halt, where  $M$  is the number of worker threads.

#### 4.5.4 Vertex Updating

Recall that we use a message passing communication interface to effectively encode the data dependencies for vertices. When the `Proceed` function is triggered on a vertex, it only needs to access its received messages as well as its local data. Compared to a remote read interface where the `Proceed` function triggered on a vertex directly accesses the data of the vertex' neighbor vertices, a message passing interface also avoids potential read and write conflicts due to concurrent operations on the same vertex (e.g., a thread updating vertex  $u$  is reading  $u$ 's neighbor vertex  $v$ 's data while another thread is updating  $v$  concurrently). By separating reads and writes, high-overhead synchronization protocols such as logical locking and data replication can usually be eliminated [134, 155].

In the GRACE runtime we implement such a low-overhead concurrency control mechanism for vertex updates as follows. When a vertex is about to be processed, we must select one message on each of its incoming edges as arguments to the `Proceed` function. Each edge maintains three pointers to received messages: one for the most recently received message; one for the message used for the last call of `Proceed`; and one for the most recently received message up to the previous tick. Some of these pointers can actually refer to the same message. When any of these message pointers are updated, older messages can be garbage collected. For each incoming edge, the `OnSelectMsg` function is invoked and the returned message is selected; if `OnSelectMsg` returns `NULL` the most recently received message is selected by default. The update operations of these message pointers are made atomic and the results of the message pointer de-reference operations are cached such that new message receptions will be logically serialized as either completely before or after the `OnSelectMsg` func-

tion call. Therefore, multiple calls to get a message reference within a single `OnSelectMsg` function will always return the same result.

After the messages have been selected, the `Proceed` function will be called to update the vertex with those messages. As described in Section 4.3, `Proceed` returns a newly computed message for each of the vertex's outgoing edges, which will in turn trigger the corresponding receiver's `OnRecvMsg` function. Since during the `OnRecvMsg` procedure, the receiver vertex's scheduling priority can be updated, and multiple messages reception can trigger the `Proceed` function simultaneously within a tick, we need to serialize the priority update procedure. The current implementation achieves this with a spin lock. Since the the chance of concurrent `Proceed` function calls is small and the priority update logic inside `Proceed` is usually simple, the serialization overhead is usually negligible. Once a scheduled vertex has finished processing, its scheduling bit is reset.

## 4.6 Experiments

Our experimental evaluation of GRACE had two goals. First, we wanted to demonstrate that by enabling customized execution policies GRACE can achieve convergence rates comparable to state-of-the-art asynchronous frameworks such as GraphLab. Second, we wanted to demonstrate that GRACE delivers the good parallel speedup of the BSP model even when asynchronous policies are used.

### 4.6.1 System

Our shared-memory GRACE prototype is implemented in C++, using the PThreads package. To test the flexibility of specifying customized executions in our runtime, we implemented four different policies: (synchronous) Jacobi (S-J), (asynchronous) GaussSeidel (AS-GS), (asynchronous) Eager (AS-E), and (asynchronous) Prior (AS-P).<sup>3</sup>

Jacobi is the simple synchronized execution policy in which all vertices are scheduled in each tick, and each vertex is updated using the messages sent in the previous tick. It guarantees both consistency and isolation and is the default execution policy in the current GRACE prototype.

The remaining three execution policies are asynchronous; all of them use the most recently received messages. For GaussSeidel, all the vertices are scheduled in each tick, while for Eager and Prior only a subset of the vertices are scheduled in each tick. Specifically, for the Eager execution policy, a vertex is scheduled if its scheduling priority exceeds a user-defined threshold. For the Prior execution policy, only  $r \cdot |V|$  of the vertices with the top priorities are scheduled for the next tick, with  $r$  a configurable selection ratio. Accurately selecting the top  $r \cdot |V|$  vertices in Prior could be done by sorting on the vertex priorities in `OnPrepare` at the tick barrier, but this would dramatically increase the synchronization cost and reduce speedup. Therefore, instead of selecting exactly the top  $r \cdot |V|$  vertices we first sample  $n$  of the priority values, sort them to obtain an approximate threshold as the scheduling priority value of the  $\lceil r \cdot n \rceil$ th vertex, and then schedule all vertices whose scheduling priorities are larger than this threshold.

---

<sup>3</sup>The system along with the provided execution policy implementations is available at <http://www.cs.cornell.edu/bigreddata/grace/>

Application	Comp. Type	Data Set	# Vertices	# Edges
SfM for 3D Model	non-linear	Acropolis	2,961	17,628
Image Restoration	non-linear	Lenna-Image	262,144	1,046,528
Topic PageRank	linear	Web-Google	875,713	5,105,039
Community Detection	non-numeric	DBLP	967,535	7,049,736

Table 4.1: Summary of Applications

The implementations of all these execution policies are straightforward, requiring only tens of lines of code in the `OnPrepare`, `OnRecvMsg`, and `OnSelectMsg` functions.

## 4.6.2 Workloads

### Custom Execution Policies

Recall that the first goal of our experiments was to demonstrate the ability of GRACE to support customized execution policies with performance comparable to existing frameworks. We therefore chose four different graph processing applications that have different preferred execution policies as summarized in Table 4.1.

**Structure from Motion.** Our first application is a technique for structure from motion (SfM) to build 3D models from unstructured image collections downloaded from the Internet [60]. The approach is based on a hybrid discrete-continuous optimization procedure which represents a set of photos as a discrete Markov random field (MRF) with geometric constraints between pairs of cameras or between cameras and scene points. Belief Propagation (BP) is used

to estimate the camera parameters as the first step of the optimization procedure. We looked into the manually written multi-threaded program from the original paper, and reimplemented its core BP component in GRACE. The MRF graph is formulated in GRACE with each vertex representing a camera, and each edge representing the relative orientation and the translation direction between a pair of cameras. During each tick  $t$ , vertex  $v$  tries to update its belief  $S_v^t$  about its represented camera's absolute location and orientation based on the relative orientations from its neighbors and possibly its priors specified by a geo-referenced coordinate. The manually written program shared by the authors of the SfM paper consists of more than 2300 lines of code for the BP implementation, while its reimplementaion in GRACE takes only about 300 lines of code, most of them copied directly from the original program. This is because in GRACE issues of parallelism such as concurrency control and synchronization are abstracted away from the application logic, greatly reducing the programming burden. The execution policy of SfM is simply Jacobi.

**Image Restoration.** Our second application is image restoration (IR) for photo analysis [149], which is also used in the experimental evaluation of GraphLab [113]. In this application the color of an image is captured by a large pair-wise Markov random field (MRF) with each vertex representing a pixel in the image. Belief propagation is used to compute the expectation of each pixel iteratively based on the observed dirty "pixel" value and binary weights with its neighbors. A qualitative example that shows the effectiveness of the BP algorithm is shown in Figure 4.6.

In the literature, three different execution policies have been used for this problem, resulting in the following three BP algorithms: the original Syn-

chronous BP, Asynchronous BP, and Residual BP [132, 70]. The execution policy for Synchronous BP is Jacobi, while the Eager policy in GRACE and the FIFO scheduler in GraphLab correspond to the Asynchronous BP algorithm. For the Residual BP algorithm, the Prior policy in GRACE and the priority scheduler in GraphLab are good approximations. Previous studies reported that Residual BP can accelerate the convergence and result in shorter running time compared to other BP algorithms [70, 77].

**Topic-Sensitive PageRank.** Our third application is topic-sensitive PageRank (TPR) for web search engines [83], where each web page has not one importance score but multiple importance scores corresponding to various topics. At query time, these importance scores are combined based on the topics of the query to form a composite score for pages matching the query. By doing this we capture more accurately the notion of importance with respect to a particular topic. In GRACE, each web page is simply a vertex and each link an edge, with the topic relative importance scores stored in the vertices. Like IR, TPR can use the Jacobi, Eager, and Prior execution policies [169].

**Social Community Detection.** Our last application is label propagation, a widely used social community detection (SCD) algorithm [106, 136]. Initially, each person has a unique label. In each iteration, everyone adopts the label that occurs most frequently among herself and her neighbors, with ties broken at random. The algorithm proceeds in iterations and terminates on convergence, i.e., when the labels stop changing. In GRACE, each vertex represents a person in the social network with her current label, and vertices vote to halt if their labels do not change over an update. Label propagation can be executed either synchronously [106] or asynchronously [136], corresponding to the Jacobi



Figure 4.6: Qualitative Evaluation of BP Restoration on Lenna Image.  
Left: noisy ( $\sigma = 20$ ). Right: restored

and GaussSeidel execution policies, respectively. Researchers have reported the Label Propagation algorithm converges much faster and can avoid oscillation when using an asynchronous execution policy [136].

## Speedup

Recall that the second goal of our experiments was to evaluate the parallel performance of GRACE. We distinguish between *light* and *heavy* iterative graph applications. In heavy applications, such as IR, the computation performed by the `Proceed` function on a vertex is relatively expensive compared to the time to retrieve the vertex and its associated data from memory. Heavy applications should exhibit good parallel scaling, as the additional computational power of more cores can be brought to good use. On the other hand, in light applications such as TPR, the computation performed on a vertex is relatively cheap compared to the time to retrieve the vertex's associated data. We anticipate that for light applications access to main memory will quickly become the bottleneck,

and thus we will not scale once the memory bandwidth has been reached.

We explore these tradeoffs by investigating the scalability of both IR and TPR to 32 processors. We also add additional computation to the light applications to confirm that it is the memory bottleneck we are encountering.

### 4.6.3 Experimental Setup

**Machine.** We ran all the experiments using a 32-core computer with 8 AMD Opteron 6220 quad-core processors and quad channel 128GB RAM. The computer is configured with 8 4-core NUMA regions. This machine is running CentOS 5.5.

**Datasets.** Table 4.1 describes the datasets that we used for our four iterative graph processing applications. For SfM, we used the Acropolis dataset, which consists of about 3 thousand geo-tagged photos and 17 thousand camera-camera relative orientations downloaded from Flickr. For IR we used the “Lenna” test image, a standard benchmark in the image restoration literature [23]. For TPR, we used a web graph released by Google [105], which contains about 880 thousand vertices and 5 million edges, and we use 128 topics in our experiments. For SCD, we use a coauthor graph from DBLP collected in Oct 2011; it has about 1 million vertices and 7 million edges.

We used the GraphLab single node multicore version 2.0 downloaded from <http://graphlab.org/downloads/> on March 13th, 2012 for all the conducted experiments.

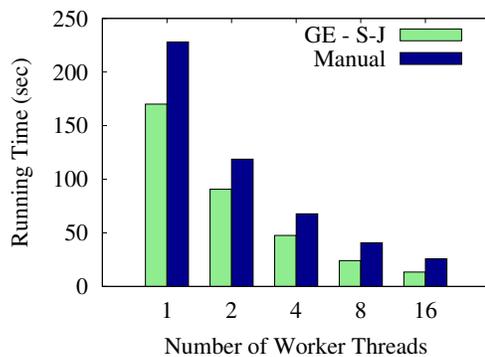


Figure 4.7: SfM Running Time

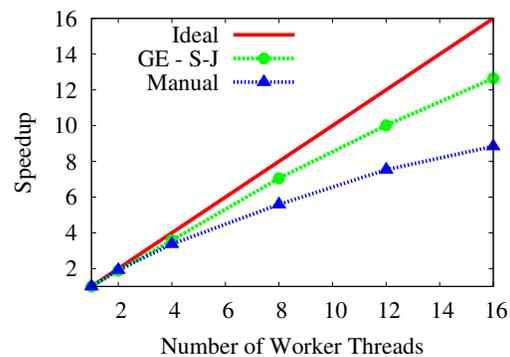


Figure 4.8: SfM Speedup

#### 4.6.4 Results for Custom Execution Policies

##### Structure from Motion

First, we used GRACE to implement the discrete-continuous optimization algorithm for SfM problems to demonstrate that by executing per-vertex update procedures in parallel GRACE is able to obtain good data parallelism. We evaluated the performance of the GRACE implementation against the manually written program. Since no ground truth is available for this application, like the manually written program we just executed the application for a fixed number of ticks. Using the Jacobi execution policy, the GRACE implementation is logically equivalent to the manually written program, and thus generates the same orientation outputs. The performance results up to 16 worker threads are shown in Figure 4.7, and the corresponding speedup results are shown in Figure 4.8. We can see that the algorithm reimplemented in GRACE has less elapsed time on single CPU, illustrating that GRACE does not add significant overhead. The GRACE implementation also has better multicore speedup, in part because the manually written code estimates the absolute camera poses and the labeling error sequentially, while following the GRACE programming model this func-

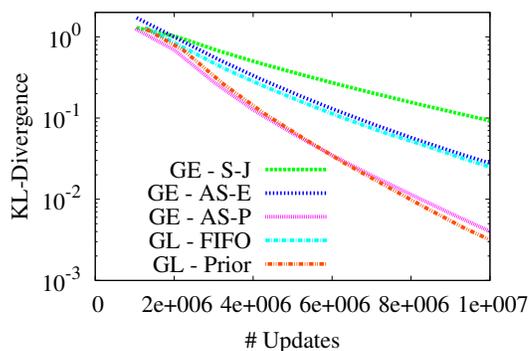


Figure 4.9: IR Convergence

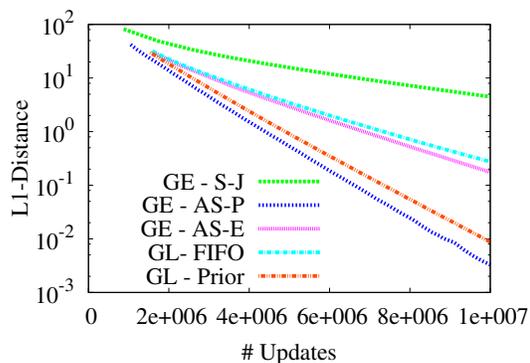


Figure 4.10: TPR Convergence

tionality is part of the `Proceed` logic, and hence is parallelized.

### Image Restoration

To compare the effectiveness of the Prior policy in GRACE with a state of the art asynchronous execution engine, we implemented the image restoration application in both GRACE and GraphLab. We use three different schedulers in GraphLab: the low-overhead FIFO scheduler, the low-overhead splash scheduler, and the higher-overhead prioritized scheduler.

Figure 4.9 examines the convergence rate of various execution policies; the x-axis is the number of updates made (for GRACE this is just the number of `Proceed` function calls) and the y-axis is the KL-divergence between the current distribution of each vertex and the ground truth distribution. Here both the priority scheduler of GraphLab and the Prior execution policy of GRACE approximate the classic residual BP algorithm. By comparing with the Jacobi execution policy of GRACE, we can see that Prior yields much faster convergence. For this application such faster convergence indeed yields better performance than Jacobi, which takes about 6.9 hours on a single processor and about 14.5 minutes

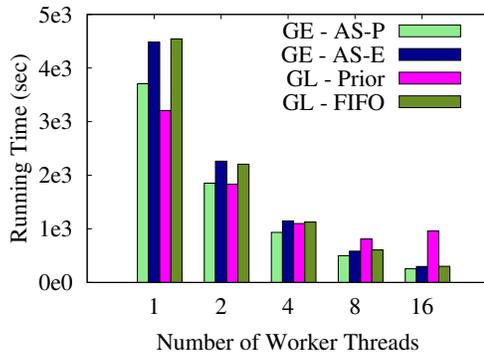


Figure 4.11: IR Running Time

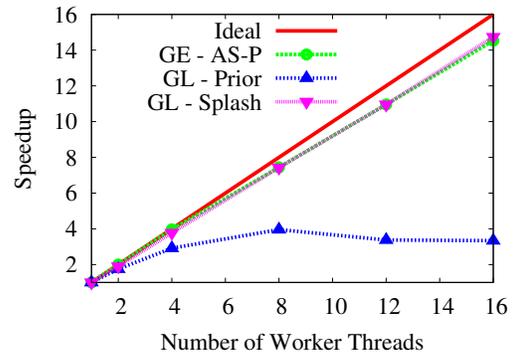


Figure 4.12: IR Speedup

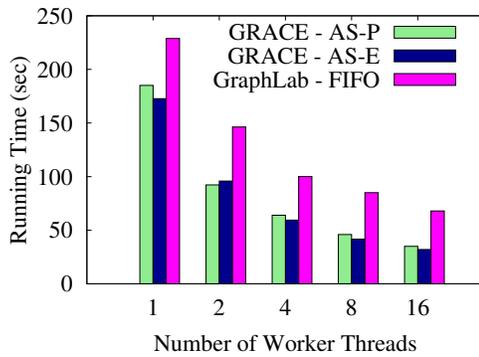


Figure 4.13: TPR Running Time

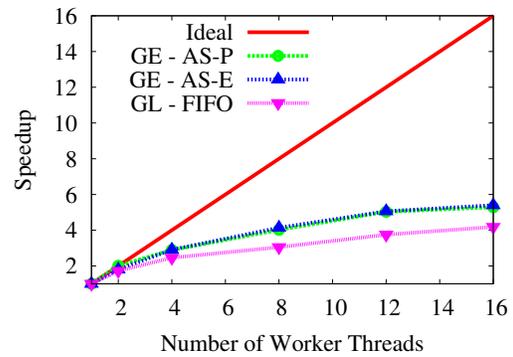


Figure 4.14: TPR Speedup

when using 32 processors.

Figures 4.11 and 4.12 compare the running time and the corresponding speedup of GRACE's Prior execution policy with the priority and splash schedulers from GraphLab. As shown in Figure 4.11, although GraphLab's priority scheduler has the best performance on a single processor, it does not speed up well. The recently introduced ResidualSplash BP was designed specifically for better parallel performance [77]. Implemented in the splash scheduler of GraphLab, this algorithm significantly outperforms the priority scheduling of GraphLab on multiple processors. As shown in Figure 4.12, the Prior policy in GRACE can achieve speedup comparable to ResidualSplash BP implemented in GraphLab. This illustrates that by carefully adding asynchronous features into

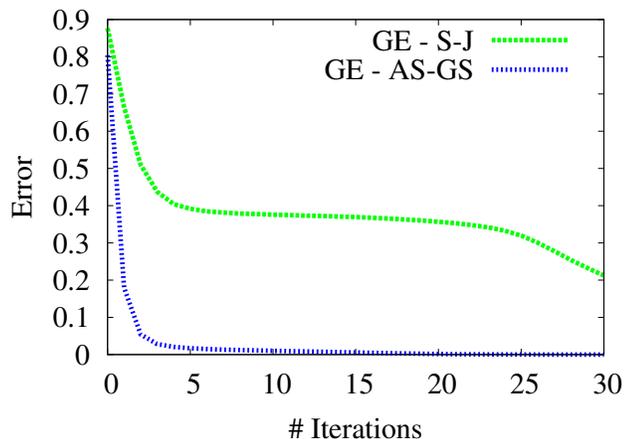


Figure 4.15: LP Convergence

synchronous execution, GRACE can benefit from both fast convergence due to prioritized scheduling of vertices and the improved parallelism resulting from the BSP model.

### Topic-sensitive PageRank

To demonstrate the benefit of the simpler Eager execution policy, we implemented the topic-sensitive PageRank (TPR) algorithm [83] in GRACE.

We also implemented this algorithm in GraphLab [113] with both the FIFO scheduler and the priority scheduler for comparison. Figure 4.10 shows the convergence results of both GRACE and GraphLab. We plot on the y-axis the average L1 distance between the converged graph and the snapshot with respect to the number of vertex updates so far. All asynchronous implementations converge faster than the synchronous implementation, and the high-overhead priority of GraphLab and the Prior execution of GRACE converge faster than the low-overhead FIFO scheduler of GraphLab or the Eager policy of GRACE. In addition, GRACE and GraphLab converge at similar rates with either low-

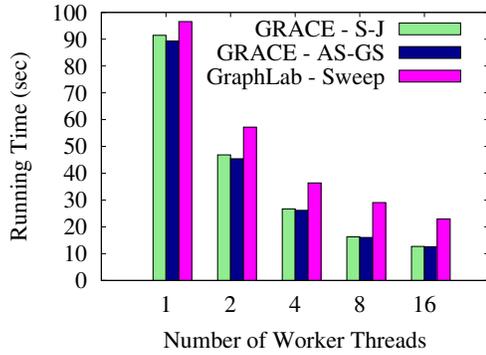


Figure 4.16: LP Running Time

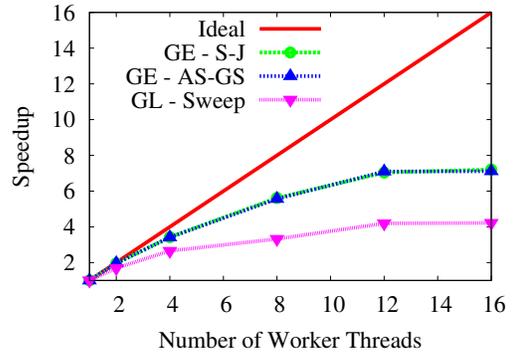


Figure 4.17: LP Speedup

overhead or high-overhead scheduling methods.

Figures 4.13 and 4.14 show the running time and the corresponding speedup of these asynchronous implementations on up to 16 worker threads. The Eager policy outperforms the Prior policy although it has more update function calls. This is because Topic PR is computationally light: although the Prior policy results in fewer vertex updates, this benefit is outweighed by its high scheduling overhead. We omit the result of the priority scheduler in GraphLab since it does not benefit from multiple processors: on 32 worker threads the running time (654.7 seconds) decreases by only less than 7 percent compared to the running time of 1 thread (703.3 seconds).

## Social Community Detection

To illustrate the effectiveness of the GaussSeidel execution policy, we use GRACE to implement a social community detection (SCD) algorithm for large-scale networks [106, 136].

Figure 4.15 shows the convergence of the algorithm using the Jacobi and GaussSeidel execution policies. On the y-axis we plot the ratio of incorrect la-

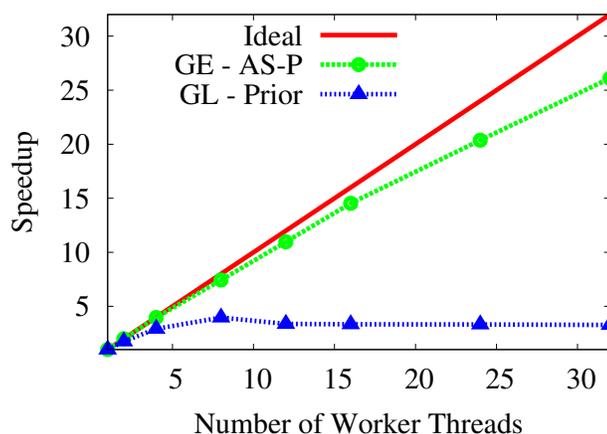


Figure 4.18: IR Speedup<sub>32core</sub>

beled vertices at the end of each iteration compared with the snapshot upon convergence. As we can see, using the Gauss-Seidel method we converge to a fixpoint after only 5 iterations; otherwise convergence is much slower, requiring more than 100 iterations to reach a fixpoint.

Figure 4.16 and 4.17 illustrate the running time and the multicore speedup compared with the sweep scheduler from GraphLab, which is a low-overhead static scheduling method. We observe that by following the BSP model for execution, GRACE achieves slightly better performance.

## 4.6.5 Results for Speedup

As discussed in Section 4.6.2, for light applications, which do little computation on each vertex, we expect limited scaling above a certain number of worker threads due to the memory bandwidth bottleneck. We have already observed this bottleneck for the SfM, topic-sensitive PageRank, and social community detection applications in Section 4.6.4, and we now investigate this issue further.

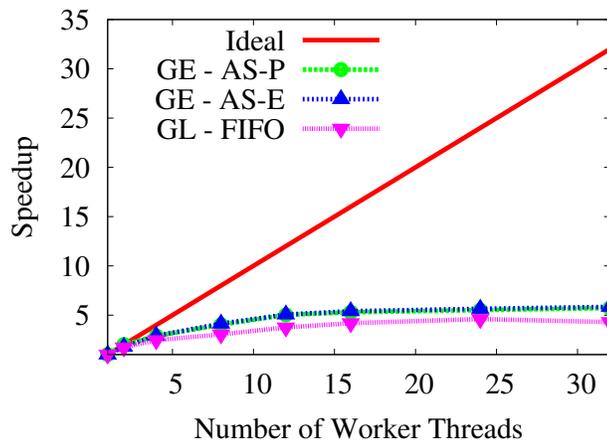


Figure 4.19: TPR Speedup<sub>32core</sub>

We start with image restoration, an application with a high ratio of computation to data access. Figure 4.18 shows the speedup for this application up to 32 worker threads. We observe that both the Eager and the Prior execution policies have nearly linear speedup. In GraphLab, FIFO and splash scheduling achieve similar speedup, but GraphLab’s priority scheduler hits a wall around 7 threads.

Our remaining three applications have a low ratio of computation to data access. We show results here from topic-sensitive PageRank as one representative application; results from the other two graph applications are similar. As shown in Figure 4.19, since the computation for each fetched byte from the memory is very light, the speedup slows down after 12 threads for both GraphLab and GRACE as our execution becomes memory-bound: we cannot supply enough data to the additional processors to achieve further speedup. We illustrate this phenomenon by adding some (unnecessary) extra computation to the update function of TPR to create a special version that we call “Loaded Topic PR.” The speedup of Loaded Topic PR, shown in Figure 4.20 is close to linear up to 32 threads. Since both Eager and Prior have similar speedup, we only show the

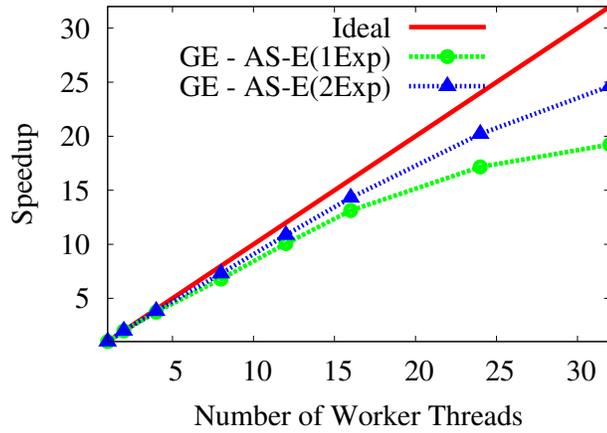


Figure 4.20:  $\text{TPR}_{\text{loaded}} \text{Speedup}_{32\text{core}}$

speedup for Eager in the figure. In Eager(1Exp) we have added one extra `exp` function call for each plus operation when updating the preference vector, and in Eager(2Exp) we added two extra `exp` function calls for each plus operation. We can observe that although Loaded Topic PR with two `exp` function calls is about 6.75 times slower than the original Topic PR on a single core, it only takes 1.6 times longer on 32 processors, further demonstrating that the memory bottleneck is inhibiting speedup.

## 4.7 Conclusions

In this chapter we have presented GRACE, a new parallel graph processing framework. GRACE provides synchronous programming with asynchronous execution while achieving near-linear parallel speedup for applications that are not memory-bound. In particular, GRACE can achieve a convergence rate that is comparable to asynchronous execution while having a comparable or better multi-core speedup compared to state-of-the-art asynchronous engines and

manually written code.

In future work, we would like to scale GRACE beyond a single machine, and we would like to investigate how we can further improve performance on today's NUMA architectures.

## CHAPTER 5

### RELATED WORK

There has been previous work on scaling behavior simulation and graph processing applications in scientific computing, distributed systems, and database literatures. This chapter first summarizes the previous work for behavioral simulations then outlines of the previous work on iterative graph programming systems.

#### 5.1 Large-Scale Behavioral Simulations

Much of the existing work on behavioral simulations has focused on task-parallel discrete event simulation systems, where computations are executed in response to events instead of in time steps [49, 128, 62, 120, 170]. In contrast to our data-parallel solution for behavioral simulations based on the state-effect pattern, these systems implement either conservative or optimistic protocols to detect conflicts and preempt or rollback simulation tasks. The strength of local interactions and the time-stepped model used in behavioral simulations lead to unsatisfactory performance, as shown in attempts to adapt discrete event simulators to agent-based simulations [92, 89].

Platforms specifically targeted at agent-based models have been developed, such as Swarm [124], Mason [115], and Player/Stage [76]. These platforms offer tools to facilitate simulation programming, but most rely on message-passing abstractions with implementations inspired by discrete event simulators, so they suffer in terms of performance and scalability. A few recent systems attempt to distribute agent-based simulations over multiple nodes without ex-

exploiting application properties such as visibility and time-stepping [87, 140]. This leads either to poor scale-up or to unrealistic restrictions on agent interactions.

Regarding join processing with MapReduce, Zhang et al. [168] compute spatial joins by an approach similar to ours when only local effect assignments are allowed. Their mapper partitions are derived using spatial index techniques rather than by reasoning about the application program, and they do not discuss *iterated* joins, an important consideration for our work. Locality optimizations have been studied for MapReduce on SMPs [164] and for MapReduce-Merge [51]; in this thesis we consider the problem in a distributed main memory MapReduce runtime.

Data-driven parallelization techniques have also been studied in parallel databases [64, 78] and data parallel programming languages [40, 143]. However, it is unnatural and inefficient to use either SQL or set-operations exclusively to express flexible computation over individuals as required for behavioral simulations.

Given this situation, behavioral simulation developers have resorted to hand-coding parallel implementations of specific simulation models [72, 127], or trading model accuracy for scalability and ease of implementation [47, 157]. Our work, in contrast, aligns in spirit with recent efforts to bring dataflow programming techniques to complex domains, such as distributed systems [28] and networking [110], with huge benefits in performance and programming productivity. To the best of our knowledge, our approach is the first to bring both programmability and scalability through data parallelism to behavioral simulations.

## 5.2 Parallel Systems for Iterative Graph Processing

Much of the existing work on iterative graph programming systems is based on the Bulk Synchronous Parallel (BSP) model which uses global synchronization between iterations. Twister [68], PEGASUS [98], Pregel [119] and PrIter[169] build on MapReduce [63], which is inspired by the BSP model. Naiad [122] is an iterative incremental version of DryadLINQ [165] with one additional fix-point operator, in which the stop criterion is checked at the end of each tick. HaLoop [42] extends the Hadoop library [81] by adding programming support for iterative computations and making the task scheduler loop-aware. Piccolo [135] and Spark [167] also follow an iterative computation pattern although with a higher-level programming interface than MapReduce/DryadLINQ; they also keep data in memory instead of writing it to disk. All these frameworks provide only synchronous iterative execution for graph processing (e.g., update all vertices in each tick) — except PrIter, which allows selective execution. However, since PrIter is based on MapReduce, it is forced to use an expensive shuffle operation at the end of each tick, and it requires the application to be written in an incremental form, which may not be suitable for all applications (such as a Belief Propagation). In addition, because of the underlying MapReduce framework, PrIter cannot support Gauss-Seidel update methods.

GraphLab [113, 114] was the first approach to use a general asynchronous model for execution. In their execution model, computation is organized into tasks, with each task corresponding to the update function of a scope of a vertex. The scope of a vertex includes itself, its edges and its adjacent vertices. Different scheduling algorithms are provided for ordering updates from tasks. GraphLab forces programmers to think and code in an asynchronous way, hav-

ing to consider the low level issues of parallelism such as memory consistency and concurrency. For example, users have to choose one from the pre-defined consistency models for executing their applications.

Galois [103], which is targeted at irregular computation in general, is based on optimistic set iterations. Users are required to provide commutativity semantics and undo operations between different function calls in order to eliminate data races when multiple iterators access the same data item simultaneously. As a result, Galois also requires users to take care of low level concurrency issues such as data sharing and deadlock avoidance.

Another category of parallel graph processing systems are graph querying systems, also called graph databases. Such systems include HyperGraph DB [6], InfiniteGraph DB [7], Neo4j [10] and Horton [139], just to name a few. Their primary goal is to provide a high level interface for processing queries against a graph. Thus, many database techniques such as indexing and query optimizations can be applied. In order to implement a graph processing application over these systems, users can only interactively call some system provided functions such as BFS/DFS traversals or Dijkstra's algorithms. This restricts programmability and applicability of these systems for iterative graph processing applications.

APPENDIX A  
APPENDIX FOR CHAPTER 3

## A.1 Spatial Joins in MapReduce

In this appendix, we first briefly review the MapReduce model (Appendix A.1.1). We then formalize the spatial joins run at each tick of a behavioral simulation in MapReduce (Appendix A.1.2).

### A.1.1 MapReduce

Since its introduction in 2004, MapReduce has become one of the most successful models for processing long running computations in distributed shared-nothing environments [63]. While it was originally designed for very large batch computations, MapReduce is ideal for behavioral simulations because it provides automatic scalability, which is one of the key requirements for next-generation platforms. By varying the degree of data partitioning and the corresponding number of map and reduce tasks, the same MapReduce program can be run on one machine or one thousand.

The MapReduce programming model is based on two functional programming primitives that operate on key-value pairs. The `map` function takes a key-value pair and produces a set of intermediate key-value pairs,  $\text{map} : (k_1, v_1) \rightarrow [(k_2, v_2)]$ , while the reduce function collects all of the intermediate pairs with the same key and produces a value,  $\text{reduce} : (k_2, [v_2]) \rightarrow [v_3]$ . Since simulations operate in ticks, we use an iterative MapReduce model in which the output of the

reduce step is fed into the next map step. Formally, this means that we change the output of the reduce step to be  $[(k3, v3)]$ .

### A.1.2 Formalizing Simulations in MapReduce

In the following, we formally develop the map and reduce functions for processing a single tick of a behavioral simulation.

**Formalizing Agents and Spatial Partitioning.** We first introduce our notation for agents and their state and effect attributes. We denote an agent  $a$  as  $a = \langle \text{oid}, \mathbf{s}, \mathbf{e} \rangle$ , where  $\mathbf{s}$  is a vector of the agent's state attributes and  $\mathbf{e}$  is a vector of its effects. To refer to an agent or its attributes at a tick  $t$ , we will write  $a^t$ ,  $\mathbf{s}^t$ , or  $\mathbf{e}^t$ . Since effect attributes are aggregated using combinator functions, they need to be reset at the end of every tick. We will use  $\theta$  to refer to the vector of idempotent values for each effect. Finally, we use  $\oplus$  to denote the aggregate operator that combines effect variables according to the appropriate combinator.

The neighborhood property implies that some subset of each agent's state attributes are spatial attributes that determine an agent's position. For an agent  $a = \langle \text{oid}, \mathbf{s}, \mathbf{e} \rangle$ , we denote this spatial location  $\ell(\mathbf{s}) \in \mathcal{L}$ , where  $\mathcal{L}$  is the spatial domain. Given an agent  $a$  at location  $l$ , the visible region of  $a$  is  $VR(l) \subseteq \mathcal{L}$ .

Both the map and reduce tasks in our framework will have access to a spatial partitioning function  $P : \mathcal{L} \rightarrow \mathbf{P}$ , where  $\mathbf{P}$  is a set of partition ids. This partitioning function can be implemented in multiple ways, such as a regular grid or a quadtree. We define the *owned* set of a partition  $p$  as the inverse image of  $p$  under  $P$ , i.e., the set of all locations assigned to  $p$ . Since each location has an

$$\begin{aligned} \text{map}^t(\cdot, a^{t-1}) &= [(p, \langle \text{oid}, \mathbf{s}^t, \theta \rangle) \text{ where } \ell(\mathbf{s}^t) \in VR(p)] \\ \text{reduce}^t(p, [\langle \text{oid}_i, \mathbf{s}_i^t, \theta \rangle]) &= [(p, \langle \text{oid}_i, \mathbf{s}_i^t, \mathbf{e}_i^t \rangle), \forall i \text{ s.t. } P(\ell(\mathbf{s}_i^t)) = p] \end{aligned}$$

Figure A.1: Map and Reduce Functions with Local Effects Only

associated visible region, we can also define the visible region of a partition as  $VR(p) = \bigcup_{l \in \mathcal{L}, P(l)=p} VR(l)$ . This is the set of all locations that might be visible by some agent in  $p$ .

**Simulations with Local Effects Only.** Since the query phase of an agent can only depend on the agents inside its visible region, the visible region of a partition contains all of the data necessary to execute the query phase for its entire owned region. We will take advantage of this by *replicating* all of the agents in this region at  $p$  so that the query phase can be executed without communication.

Figure A.1 shows the map and reduce functions for processing tick  $t$  when there are only local effect assignments. At tick  $t$ , the map function performs the update phase from the previous tick, and the reduce function performs the query phase. The map function takes as input an agent with state and effect variables from the previous tick ( $a^{t-1}$ ), and updates the state variables to  $\mathbf{s}^t$  and the effect attributes to  $\theta$ . During the very first tick of the simulation,  $\mathbf{e}^{t-1}$  is undefined, so  $\mathbf{s}^t$  will be set to a value reflecting the initial simulation state. The map function emits a copy of the updated agent keyed by partition for each partition containing the agent in its visible set ( $\ell(\mathbf{s}^t) \in VR(p)$ ). This has the effect of replicating the agent  $a$  to every partition that might need it for query phase processing. The amount of replication depends on the partitioning function and on the size of each agent's visible region.

The reduce function receives as input all agents that are sent to a particular partition  $p$ . This includes the agents in  $p$ 's owned region, as well as replicas of all the agents that fall in  $p$ 's visible region. The reducer will execute the query phase and compute effect variables for all of the agents in its owned region (agent  $i$  s.t.  $P(\ell(s_i^t)) = p$ ). This requires no communication, since the query phase of an agent in  $p$ 's owned region can only depend on the agents in  $p$ 's visible region, all of which are already replicated at the same reducer. After the query phase the reducer then outputs agents with updated effect attributes that are going to be processed in the next tick.

**Simulations with Non-Local Effects.** The method above only works when all effect assignments are local. If an agent  $a$  makes an effect assignment to some agent  $b$  in its visible region, then it must communicate that effect to the reducer responsible for processing  $b$ . Figure A.2 shows the map and reduce functions to handle simulations with non-local effect assignments. The first map function task is same as in the local effect case. Each agent is partitioned and replicated as necessary. As before, the first reduce function computes the query phase for the agents in  $p$ 's owned set and computes effect values. In this case, however, it can only compute intermediate effect values  $f^t$ , since it does not have the effects assigned at other nodes. This reducer outputs one pair for every agent, including replicas. These agents are keyed with the partition that owns them, so that all replicas of the same agent go to the same node.

The second map function is the identity, and the second reduce function performs the aggregation necessary to complete the query phase. It receives all of the updated replicas of all of the agents in its owned region and applies the  $\oplus$  operation to compute the final effect values and complete the query phase.

$$\begin{aligned}
\text{map}_1^t(\cdot, a^{t-1}) &= [(p, \langle \text{oid}, \mathbf{s}^t, \theta \rangle) \text{ where } \ell(\mathbf{s}^t) \in VR(p)] \\
\text{reduce}_1^t(p, [\langle \text{oid}_i, \mathbf{s}_i^t, \theta \rangle]) &= [(P(\ell(\mathbf{s}^t)), \langle \text{oid}_i, \mathbf{s}_i^t, \mathbf{f}_i^t \rangle)] \\
\text{map}_2^t(k, a) &= (k, a) \\
\text{reduce}_2^t(p, [\langle \text{oid}_i, \mathbf{s}_i^t, \mathbf{f}_i^t \rangle]) &= [(p, \langle \text{oid}_i, \mathbf{s}_i^t, \oplus_j \mathbf{f}_j^t \rangle), \forall j \text{ s.t. } \text{oid}_i = \text{oid}_j]
\end{aligned}$$

Figure A.2: Map and Reduce Functions with Non-local Effects

Each reducer will output an updated copy of each agent in its owned set.

## A.2 Formal Semantics of BRASIL

In this appendix, we provide a more formal presentation of the semantics of BRASIL than the one presented in Section 3.4. In particular, we show how to convert BRASIL expressions into monad algebra expressions for analysis and optimization. We also prove several results regarding effect inversion, introduced in Section 3.4.2, and illustrate the resulting trade-offs between computation and communication.

For the most part, our work will be in the traditional monad algebra. We refer the reader to the original work on this algebra [44, 101, 131, 150] for its basic operators and nested data model. We also use standard definitions for the derived operations like cartesian product and nesting. For example, we define cartesian product as

$$f \times g := \langle 1 : f, 2 : g \rangle \circ \text{PAIRWITH}_1 \circ \text{FLATMAP}(\text{PAIRWITH}_2) \quad (\text{A.1})$$

For the purpose of readability, composition in (A.1) and the rest of our presentation, is read left-to-right; that is,  $(f \circ g)(x) = g(f(x))$ .

We assume that the underlying domain is the real numbers, and that we have associated arithmetic operators. We also add traditional aggregate functions like COUNT and SUM to the algebra; these functions take a set of elements (of the appropriate type) and return a value.

In order to simplify our presentation, we do make several small changes that relax the typing constraints in the classic monad algebra. In particular, we want to allow union to combine sets of tuples with different types. For this end, we introduce a special NIL value. This value is the result of any query that is undefined on the input data, such as projection on a nonexistent attribute. This value has a form of “null-semantics” in that values combined with NIL are NIL, and NIL elements in a set are ignored by aggregates. In addition, we introduce a special aggregate function GET. When given a set, this function returns its contents if it is a singleton, and returns NIL otherwise. Neither this function, nor the presence of NIL significantly affects the expressive power of the monad algebra [148].

### **A.2.1 Monad Algebra Translation**

For the purpose of illustration, we assume that our simulation has only one class of agents, all of which are running the same simulation script. It is relatively easy to generalize our approach to multiple agent classes or multiple scripts. Given this assumption, our simulation data is simply a set of tuples  $\{t_0, \dots, t_n\}$  where each tuple  $t_i$  represents the data inside of an agent. Every agent tuple has a special attribute KEY which is used to uniquely identify the agent; variables which reference another agent make use of this key. The state-effect pattern

requires that all data types other than agents be processed by value, so they can safely be stored inside each agent.

We let  $\tau$  represent the type/schema of an agent. In addition to the key attribute,  $\tau$  has an attribute for each state and effect field. The value of a state attribute is the value of the field. The value of an effect attribute is a pair  $\langle 1 : n, 2 : \text{AGG} \rangle$  where  $n$  is a unique identifier for the field and AGG is the aggregate for this effect.

During the query phase, we represent effects as a tuple  $\langle k : \mathbb{N}, e : \mathbb{N}, v : \sigma \rangle$ , where  $k$  is the key of the object being effected,  $e$  is the effect field identifier, and  $v$  is the value of the effect. As a shorthand, let  $\rho$  be this type. Even though effects may have different types, because of our relaxed typing, this will not harm our formalism.

The syntax of BRASIL forces the programmer to clearly separate the code into a query script (i.e. `run()`) and an update script (the update rules). A query script compiles to an expression whose input and output are the tuple  $\langle 1 : \tau', 2 : \{\tau\}, 3 : \{\rho\} \rangle$ . The first element represents the active agent for this script;  $\tau'$  “extends” type  $\tau$  in that it is guaranteed to have an attribute for the key and each state field, but it may have more attributes. The second element is the set of all other agents with which this agent must communicate. The last element is the set of effects generated by this script.

Let  $Q$  be the monad expression for the query script. Then the effect generation stage is the expression

$$\Omega(Q) = (\text{ID} \times \text{ID}) \circ \text{NEST}_2 \circ \text{MAP}(\widehat{Q}) \quad (\text{A.2})$$

where  $\widehat{Q}$  is defined as

$$\widehat{Q} = \langle 1:\pi_1, 2:\pi_2, 3:\{\} \rangle \circ Q \circ \langle 1:\pi_1, 2:\pi_3 \rangle \quad (\text{A.3})$$

This produces a set of agents and the effects that they have generated (which may or may not be local). In general, we will aggregate aggressively, so each agent will only have one effect for each  $\langle k, e \rangle$  pair. For the effect aggregation stage, we must aggregate the effects for each agent and inline them into the agent tuple. If we only have local effect assignments, then this expression is  $\Omega(Q) \circ \mathfrak{E}$  where

$$\mathfrak{E} = \text{MAP}(\langle \langle \text{KEY}:\pi_{\text{KEY}}, s_i:\pi_{s_i}, e_j:\pi_2 \circ \sigma_{\pi_e=\pi_{e_j} \circ \pi_1} \circ (\pi_{e_j} \circ \pi_2) \rangle \rangle_{i,j}) \quad (\text{A.4})$$

where the  $s_i$  are the state fields and the  $e_j$  are the value of effect fields. However, in the case where we have non-local effects, we must first redistribute them via the expression

$$\mathfrak{R} = (\pi_1 \times \pi_2) \circ \text{MAP}(\langle 1:\pi_1, 2:\text{FLATTEN} \circ \sigma_{\pi_k=\pi_1 \circ \pi_{\text{KEY}}} \rangle) \quad (\text{A.5})$$

So the entire query phase is  $\Omega(Q) \circ \mathfrak{R} \circ \mathfrak{E}$ . Finally, for the update phase, each state  $s_i$  has an update rule which corresponds to an expression  $U_{s_i}$ . These scripts read the output of the expression  $\mathfrak{E}$ . Hence the query for our entire simulation is the expression

$$\Omega(Q) \circ \mathfrak{R} \circ \mathfrak{E} \circ \mathfrak{U}(U_{s_0}, \dots, U_{s_n}) \quad (\text{A.6})$$

where the update phase is defined as

$$\mathfrak{U}(U_{s_0}, \dots, U_{s_n}) = \text{MAP}(\langle \langle \text{KEY}:\pi_{\text{KEY}}, s_0:U_{s_0}, \dots, s_n:U_{s_n} \rangle \rangle) \quad (\text{A.7})$$

The only remaining detail in our formal semantics is to define semantics for the query scripts and update scripts. Update scripts are just simple calculations on a tuple, and are straightforward. The nontrivial part concerns the query

$$\begin{aligned}
\llbracket \text{const } \tau \ x = E \rrbracket_V &= \langle 1 : \chi_x(\llbracket E \rrbracket_V), 2 : \pi_2, 3 : \pi_3 \rangle \\
\llbracket \text{effect } \tau \ x : f \rrbracket_V &= \langle 1 : \chi_x(\langle 1 : \rho(x), 2 : f \rangle), 2 : \pi_2, 3 : \pi_3 \rangle \\
\llbracket x \leftarrow E \rrbracket_V &= \langle 1 : \pi_1, 2 : \pi_2, 3 : \pi_3 \oplus (\langle 1 : \pi_1 \circ \pi_{\text{KEY}}, 2 : \rho(x), 3 : \llbracket E \rrbracket_V \rangle \circ \text{SNG}) \rangle \\
\llbracket R.x \leftarrow E \rrbracket &= \langle 1 : \pi_1, 2 : \pi_2, 3 : \pi_3 \oplus (\langle 1 : \llbracket R \rrbracket_v, 2 : \rho(x), 3 : \llbracket E \rrbracket_V \rangle \circ \text{SNG}) \rangle \\
\llbracket \text{if } (E) \ \{B_1\} \ \text{else } \{B_2\} \rrbracket_V &= \langle 1 : \pi_1, 2 : \pi_2, \\
&\quad 3 : \text{SNG} \circ \sigma_{\llbracket E \rrbracket_V} \circ \text{GET} \circ \llbracket B_1 \rrbracket_V \oplus \\
&\quad \text{SNG} \circ \sigma_{\neg \llbracket E \rrbracket_V} \circ \text{GET} \circ \llbracket B_2 \rrbracket_V \rangle \\
\llbracket \text{foreach } (\tau \ x : E) \ \{B\} \rrbracket_V &= \langle 1 : \pi_1, 2 : \pi_2, \\
&\quad 3 : \langle 1 : \pi_1 \circ \chi_x(\llbracket E \rrbracket_V) \circ \text{PAIRWITH}_x, 2 : \pi_2, 3 : \pi_3 \rangle \\
&\quad \circ \text{FLATMAP}(\llbracket B \rrbracket_V \circ \pi_3) \rangle
\end{aligned}$$

Figure A.3: Translation for Common Commands

scripts. A script is just a sequence of statements  $S_0; \dots; S_n$  where each statement is a variable declaration, assignment, or control structure (e.g. conditional, foreach-loop). See the BRASIL Language manual for more information on the complete grammar [58]. It suffices to define, for each statement  $S$ , a monad algebra expression  $\llbracket S \rrbracket$  whose input and output are the triple  $\langle 1 : \tau', 2 : \{\tau\}, 3 : \{\rho\} \rangle$ ; we handle sequences of commands by composing these expressions.

Our query script semantics in BRASIL also depends upon the visibility constraints in the script. In particular, it is possible that a reference to another agent is fine initially, but violates the visibility constraint as that agent moves relative to the one holding the reference. For that reason, BRASIL employs *weak reference semantics* for agent references, similar to weak references in Java. If another agent moves outside of the visible region, then all references to it will resolve to

NIL. To formally support the notion of weak references, we represent visibility as a predicate  $V(x, y)$  which compares two agents; for any statement  $S$ , we let  $\llbracket S \rrbracket$  be its interpretation with this constraint and  $\llbracket S \rrbracket_V$  be the semantics without.

Before translating statements, we must translate expressions that may appear inside of them. The only nontrivial expressions are references; arithmetic expressions or other complex expressions translate to the monad algebra in the natural way. References return either the variable value, or the key for the agent referenced. Ignoring visibility constraints, for any identifier  $x$ , we define

$$\llbracket x \rrbracket = \begin{cases} \text{PAIRWITH}_3 \circ \sigma_{\pi_1 \circ \pi_x \circ \pi_1 = \pi_3 \circ \pi_e} \circ \sigma_{\pi_1 \circ \pi_{\text{KEY}} = \pi_3 \circ \pi_k} \circ \text{GET} & E \text{ is effect} \\ \pi_1 \circ \pi_x & \text{otherwise} \end{cases} \quad (\text{A.8})$$

In general, for any reference  $E.x$ , we define

$$\llbracket E.x \rrbracket = \langle 1 : \pi_2 \circ \sigma_{\pi_{\text{KEY}} = \llbracket E \rrbracket} \circ \text{GET}, 2 : \pi_2, 3 : \pi_3 \rangle \circ \llbracket x \rrbracket \quad (\text{A.9})$$

If we include visibility constraints,  $\llbracket E \rrbracket_V$  is defined in much the same way as  $\llbracket E \rrbracket$  except when  $E$  is an agent reference. In that case,

$$\begin{aligned} \llbracket x \rrbracket_V = \langle 1 : \text{ID}, 2 : \pi_2 \circ \sigma_{\pi_{\text{KEY}} = \llbracket E \rrbracket} \circ \text{GET} \rangle \circ \langle 1 : V, 2 : \pi_2 \rangle \circ \text{SNG} \\ \circ \sigma_{\pi_1} \circ \text{GET} \circ \pi_2 \circ \pi_k \end{aligned} \quad (\text{A.10})$$

This expression temporarily retrieves the object, tests if it is visible, and returns NIL if not.

To complete our semantics, we introduce the following notation.

- $\chi_a(f)$  is an operation that takes a tuple and extends it with an attribute  $a$  having value  $f$ . It is definable in the monad algebra, but its exact definition depends on its usage context.

- $\oplus$  is an operation that takes two sets of effects and aggregates those with the same key and effect identifier. It is definable on in the monad algebra, but its exact definition depends on the effect fields in the BRASIL script.
- $\rho(x)$  is the effect identifier for a variable  $x$ . In practice, this is the position of the declaration of  $x$  in the BRASIL script.

Given these three expressions, Figure A.3 illustrates the translation of some of the more popular statements in the monad algebra. In general, variable declarations modify the first element of the input triple (i.e. the active agent), while assignments and control structures modify the last element (i.e. the effects).

As we discussed in Section 3.4.2, this formalism allows us to apply standard algebraic rewrites from the monad algebra for optimization. For example, many of the operators in Figure A.3 – particularly the tuple constructions – are often unnecessary. They are there to preserve the input and output format, in order to facilitate composition. There are rewrite rules that function like dead-code elimination, in that they remove tuples that are not being used. One of the consequences is that many `foreach`-loops simplify to the form

$$F(E, B) = \langle 1 : \text{ID}, 2 : E \rangle \circ \text{PAIRWITH}_2 \circ \text{FLATMAP}(B) \quad (\text{A.11})$$

Note that this form is “half” of the cartesian product in (A.1); it joins a single value with a set of values. Thus when we simplify the `foreach`-loop to this form, we can often apply join optimization techniques to the result.

Another advantage of this formalism is that it allows us to prove correctness results. Note that the usage of weak references in BRASIL gives a different semantics for visibility than the one present in Section 3.3; BRASIL uses visibility to determine how agent references are resolved, while the BRACE runtime uses

visibility to determine agent replication and communication. The BRASIL semantics are preferable for a developer, because they are easy to understand and hide MapReduce details. Fortunately, our formalism allows us to establish that these two are equivalent.

**Theorem 1.** *Let  $Q$  be a BRASIL query script whose references are restricted by visibility predicate  $V$ . Then*

$$\text{NEST}_2 \circ \text{MAP}(\widehat{\llbracket Q \rrbracket}_V) = \sigma_V \circ \text{NEST}_2 \circ \text{MAP}(\widehat{\llbracket Q \rrbracket}) \quad (\text{A.12})$$

Furthermore, let

$$O(F) = F \circ (\pi_2 \times \pi_3) \circ \sigma_{\pi_1 \circ \pi_{\text{KEY}} = \pi_2 \circ \pi_k} \circ \text{MAP}(\pi_1) \quad (\text{A.13})$$

be the set of objects affected by an expression  $F$ . Then

$$\text{MAP}(\langle 1 : \pi_1, 2 : O(\llbracket Q \rrbracket_V) \rangle) \circ \sigma_V = \text{MAP}(\langle 1 : \pi_1, 2 : O(\llbracket Q \rrbracket) \rangle) \quad (\text{A.14})$$

The significance of (A.12) is that, instead of implementing the overhead of checking for weak references, we can filter out the agents that are not visible and eliminate any further visibility checking. The significance of (A.14) is that weak references insure one agent can only affect other agents falling in its visible region.

## A.2.2 Effect Inversion

As we saw in Section 3.4.2, there is an advantage to writing a BRASIL script so that all effects assignments are local. It may not always be natural to do so, as the underlying scientific models may be expressed in terms of non-local effects. However, in certain cases, we may be able to automatically rewrite a

BRASIL program to only use local effects. In particular, if there are no visibility constraints, then we can always invert effect assignments to make them local-only.

**Theorem 2.** *Let  $Q$  be a query script with no visibility constraints. There is a script  $Q'$  with only local effects such that  $\llbracket Q \rrbracket = \llbracket Q' \rrbracket$ .*

*Proof Sketch.* Our proof takes advantage of the fact that effect fields (as opposed to effect variables) may not be read during the query phase, and that effects are aggregated independent of order. We start with  $Q$  and create a copy script  $Q_1$ . Within this copy, we remove all syntactically non-local effect assignments (e.g.  $E.x \leftarrow v$ ). Some of these may actually be local in the semantic sense, but this does not effect our proof.

We construct another copy  $Q_2$ . For this copy, we pick a variable  $a$  that does not appear in  $Q$ . We replace every local state reference  $x$  in  $Q$  with  $a.x$ . We also remove all local effect assignments. Finally, we replace each syntactically non-local assignment  $E.x \leftarrow v$  with the conditional assignment `if (E == this) {x ← v}`. We then let  $Q_3$  be the script

$$\text{foreach}(\text{Agent } a : \text{Extent}\langle\text{Agent}\rangle) \{ Q_2(a) \}$$

That is,  $Q_3$  is the act of an agent running the script for each other agent, searching for effects to itself, and then assigning them locally. The script  $Q_1; Q_3$  is our desired script.  $\square$

Note that this conversion comes at the cost of an additional `foreach`-loop, as each agent simulates the actions of all other agents. Thus, this conversion is much more computationally expensive than the original script. However, we can often simplify this to remove the extra loop. As mentioned previously, a

foreach-loop can often be simplified to the form in (A.11). In the case of two nested loops over the same set  $E$ , the merging of these two loops is a type of self-join. That is,

$$\begin{aligned}
F(E, F(E, B)) &= \langle 1 : \text{ID}, 2 : E \rangle \circ \text{PAIRWITH}_2 \circ \\
&\quad \text{FLATMAP}(\langle 1 : \text{ID}, 2 : E \rangle \circ \text{PAIRWITH}_2 \circ \text{FLATMAP}(B)) \\
&= \langle 1 : \text{ID}, 2 : E, 3 : E \rangle \circ \text{PAIRWITH}_2 \circ \\
&\quad \text{FLATMAP}(\text{PAIRWITH}_3 \circ \text{FLATMAP}(B')) \\
&= \langle 1 : \text{ID}, 2 : (E \times E) \rangle \circ \text{PAIRWITH}_2 \circ \text{FLATMAP}(B'')
\end{aligned}$$

where  $B'$  and  $B''$  are  $B$  rewritten to account for the change in tuple positions. As part of this rewrite, one may discover that self-join is redundant in the expression  $B''$  and eliminate it; this is how we get simple effect inversions like the one illustrated in Section 3.4.2.

In the case of visibility constraints, the situation becomes a little more complex. In order to do the inversion that we did the proof of Theorem 2, we must require that any agent  $a_1$  that assigns effects to another agent  $a_2$  must restrict its visibility to agents visible to  $a_2$ ; that way  $a_2$  can get the same results when it reproduces the actions of  $a_1$ . This is fairly restrictive, as it suggests that every agent needs to be visible to every other agent.

We can do better by introducing an information flow analysis. We only require that, for each non-local effect assigned to agent, that effect is computed using only information from agents visible to the one being assigned. However, this property depends on the values of the agents, and cannot (in general) be inferred statically from the script. Thus it is infeasible to exploit this property in general.

However, there is another way to invert scripts in the phase of visibility constraints. Suppose the visibility constraint for a script  $Q$  is a distance bound, such as  $d(x, y) < R$ . If we relax the visibility constraint for the script in the proof of Theorem 2 to  $d(x, y) < 2R$ , then the proof carries through again. We state this modified result as follows:

**Theorem 3.** *Let  $Q$  be a query script with visibility constraint  $V$ . Let  $V'$  be such that  $V'(x, y)$  if and only if  $\exists z V(x, z) \wedge V(z, y)$ . Then there is a script  $Q'$  with only local effects such that  $\llbracket Q \rrbracket_V = \llbracket Q' \rrbracket_{V'}$ .*

*Proof Sketch.* The proof is similar to that of Theorem 2. The only difference is that we have to ensure that the increased visibility for  $Q'$  does not cause the weak references in a script to resolve to agents that would have otherwise evaluated to NIL. In the construction of  $Q_2$ , we use local constants to normalize the expressions so that any agent reference in the original script becomes a local constant. For example, suppose each agent has a field `friend` that is a reference to another agent. If we have a conditional of the form

```
if (friend.x - x < BOUND) { ... }
```

then we normalize this expression as

```
const agent temp = friend;
if (temp.x - x < BOUND) { ... }
```

We then wrap these introduced constants with conditionals that test for visibility with respect to the old constraints. For example, the code above would become

```
const agent temp = (visible(this, friend) ? friend : null);  
if (temp.x - x < BOUND) { ... }
```

where `visible` is a method evaluating the visibility constraint and `evaluates` to `NIL` in the monad algebra. Given the semantics of `NIL`, this translation has the desired result. □

### A.3 Details of Simulation Models

This appendix describes the simulation models we have implemented for BRACE single-node performance and scalability experiments.

**Traffic Simulation.** Traffic simulation is required to provide accurate, high-resolution, and realistic transportation activity for the design and management of transportation systems. MITSIM, a state-of-the-art single-node behavioral traffic simulator has several different models covering different aspects of driver behavior [163]. For example, during each time step, a lane selection model will make the driver inspect the lead and rear vehicles as well as the average velocity of the vehicles in her current, left, and right lanes (within lookahead distance parameter  $\rho$ ) to compute the utility function for each lane. A probabilistic decision of lane selection is then made according to the lane utility. If the driver decides to change her lane, she needs to inspect the gaps from herself to the lead and rear vehicles in the target lane to decide if it is safe to change to the target lane in the next time step. Otherwise, the vehicle following model is used to adapt her velocity based on the lead vehicle. The newly computed velocity will replace the old velocity in the next time step. Note that if the driver can-

not find a lead or rear vehicle within  $\rho$ , she will just assume the distance to the lead or rear vehicle is infinite, and adjust the velocity according to a free-flow submodel.

One note is that since the MITSIM implementation hand-coded nearest neighbor indexing for accessing the lead and rear vehicles for performance reasons, its lookahead distance actually varies for each vehicle. In our reimplementation we fix the lookahead distance to 200 and apply single-node spatial indexing. In order to make sure this implementation difference does not generate drastically different aggregate driving behavior, we validate consistency of the MITSIM model encoded in BRASIL in terms of the simulated traffic conditions. We compare lane changing frequencies, average lane velocity and average lane density with the segment length 20,000 on both simulators. The statistical difference is measured by RMSPE (Relative Mean Square Percentage Error), which is often used as a goodness-of-fit measure in the traffic simulation literature [52]. The results for all these three statistics are shown in Table A.1. We can see that except for Lane 4's average density and changing frequency, all the other statistics demonstrate strong agreement between the two simulators. This exception is due to the fact that in the MITSIM lane changing model drivers have a reluctance factor to change to the right most lane (i.e., Lane 4). As a result there are only a few vehicles on that lane (56.33 vehicles on average compared to 351.42 on other lanes), and small lane changing record deviations due to the fixed lookahead distance approximation can contribute significantly to the error measurement.

**Fish School Simulation.** Couzin et al. have built a behavioral fish school simulation model to study information transfer in groups of fish when group mem-

Lane	Change Frequency	Avg. Density	Avg. Velocity
1	8.93%	7.42%	0.007%
2	5.57%	10.38%	0.007%
3	7.67%	9.38%	0.007%
4	21.37%	19.72%	0.007%

Table A.1: RMSPE for Traffic Simulation (LookAhead = 200)

bers cannot recognize which companions are informed individuals who know about a food source [59]. This computational model proceeds in time steps, i.e., at each time period each fish inspects its environment to decide on the direction which it will take during the next time period. Two basic behaviors of a single fish are avoidance and attraction. Avoidance has the higher priority: Whenever a fish is too close to others (i.e., distance less than a parameter  $\alpha$ ), it tries to turn away from them. If there is no other fish within distance  $\alpha$ , then the fish will be attracted to other fish within distance  $\rho > \alpha$ . The influence will be summed and normalized with respect to the current fish. Therefore, any other individuals out of the visibility range  $\rho$  of the current individual will not influence its movement decision. In addition, informed individuals have a preferred direction, e.g., the direction to the food source or the direction of migration. These individuals will balance the strength of their social interactions (attraction and avoidance) with their preferred direction according to a weight parameter  $\omega$ .

**Predator Simulation.** Since both the traffic and the fish school simulations only use local effect assignments, we designed a new predator simulation, inspired by simulations of artificial societies [96]. In this simulation, a fish can “spawn” new fish and “bite” other fish, possibly killing them, so density naturally approaches an equilibrium value at which births and deaths are balanced. Since effect inversion is not yet implemented in the BRASIL Compiler, we program biting behavior either as a non-local effect assignment (fish assign “hurt” effects

to others) or as a local one (fish collect “hurt” effects from others) in otherwise identical BRASIL scripts.

APPENDIX B  
APPENDIX FOR CHAPTER 4

## B.1 Convergence Rate Analysis for PageRank

In this section we discuss the convergence rate of the algorithms for the Topic-Sensitive PageRank in Section 4.6.4. For simplicity, we will consider Topic-Sensitive PageRank with one topic in undirected graph.

Consider the graph  $G(V, E)$  with  $n$  vertices, we denote its adjacency matrix as  $A$ , i.e.  $A_{i,j} = 1$  if  $(i, j) \in E$  otherwise 0. Let  $d_i$  be the degree of vertex  $i$  and  $D = \text{diag}(d_1, d_2, \dots, d_n)$ . Thus the goal of PageRank algorithm is to find the fixpoint  $x^*$  of the following iteration:

$$x^{(t+1)} = dD^{-1}Ax^{(t)} + b \tag{B.1}$$

where  $d$  is the damping factor, set to 0.85 in our experiments, and  $b$  is the preference vector [83].

**Theorem 4.** *Both Jacobi method and Gauss-Seidel method will converge to the unique fixpoint specified by Eq.(B.1). And Gauss-Seidel method converges faster.*

*Proof Sketch.* Since  $D^{-1}A$  is a stochastic matrix, by the Perron-Frobenius theorem its spectral radius  $\rho(D^{-1}A)$  is equal to 1. Thus  $\rho(dD^{-1}A) = d < 1$ , therefore both Gauss-Seidel method and Jacobi method converge to the unique fixpoint, while Gauss-Seidel method converges faster according to the Stein-Rosenberg Theorem [38]. □

**Theorem 5.** *Prioritized scheduling on regular undirected graphs converges to a unique fixed point, and converges faster in the energy norm than the Gauss-Seidel methods.*

*Proof Sketch.* Notice that the fixpoint is also the solution for the equation  $Cx = b$ , where  $C = I - dD^{-1}Ax$ . To measure the progress of the asynchronous algorithm, we introduce the cost function  $F(x) = \frac{1}{2}x^T Cx - x^T b$ . It is not difficult to verify that  $C$  is a symmetric positive definite matrix, which makes  $F$  a strictly convex function. Thus a vector  $x$  minimizes  $F$  if and only if  $\nabla F(x) = 0$ . Since  $\nabla F(x) = Cx - b$ , the vector  $x^*$  is the fixpoint specified by Equation (B.1) if and only if  $x^*$  minimizes  $F$ .

Therefore, the update on vertex  $i$  is equivalent to changing vector  $x$  only on coordinate  $x_i$  so that  $\nabla_i F(x) = 0$  [38]. The normal Gauss-Seidel method picks up each coordinate sequentially, while the prioritized scheduling picks up the coordinate with the largest  $\nabla_i F(x)$  each time. Both of the two methods can be viewed as coordinate descent over the objective function  $F$ . And both of them will converge to the minimum of  $F$  since  $F$  is strictly convex [37]. Correspondingly, both of the two methods will converge to the solution of  $Cx = b$ .

By lemma 1, choosing the coordinate  $i$  with larger  $\nabla_i F(x)$  decreases the value of function  $F$  more, and results in faster convergence. Therefore prioritized scheduling will converge faster than Gauss-Seidel method.  $\square$

**Lemma 1.** *Updating coordinate  $x_i$  will decrease the value of function  $F$  by  $\frac{1}{2}(\nabla_i F(x))^2$ .*

*Proof Sketch.* Suppose we have the current vector  $x$  and after the update it will become  $x'$ , where  $x'$  only differs with  $x$  on the  $i^{\text{th}}$  coordinate, and  $\nabla_i F(x') = 0$ . Notice that  $\nabla_i F(x) - \nabla_i F(x') = x_i - x'_i$ , we have  $x_i - x'_i = -\nabla_i F(x)$ . Now

consider  $F(x) - F(x')$ , we have

$$\begin{aligned}
F(x) - F(x') &= \left(\frac{1}{2}x^\top Cx - x^\top b\right) - \left(\frac{1}{2}x'^\top Cx - x'^\top b\right) \\
&= \frac{1}{2} \sum_{j,k} (x_j x_k - x'_j x'_k) \cdot C_{j,k} + (x'^\top - x^\top)b \\
&= \frac{1}{2} \left( \sum_{j \neq i} x_j (x'_i - x_i) \cdot C_{j,i} + \sum_{k \neq i} (x_i - x'_i) x_k \cdot C_{i,k} \right. \\
&\quad \left. + (x_i'^2 - x_i^2) \right) + (x_i - x'_i) b_i \\
&= \nabla_i F(x) \left( \sum_{j \neq i} C_{i,j} \cdot x_j + b_i \right) + \frac{1}{2} (x_i'^2 - x_i^2) \\
&= \nabla_i F(x) \left( \sum_j C_{i,j} \cdot x_j + b_i \right) - \nabla_i F(x) x_i + \frac{1}{2} \nabla_i F(x) (x'_i + x_i) \\
&= \nabla_i F(x) (Cx - b)_i + \frac{1}{2} \nabla_i F(x) (x'_i - x_i) \\
&= (\nabla_i F(x))^2 - \frac{1}{2} (\nabla_i F(x))^2 = \frac{1}{2} (\nabla_i F(x))^2
\end{aligned}$$

□

## BIBLIOGRAPHY

- [1] Ateji PX: Java parallel programming made simple.  
<http://http://www.ateji.com/px/index.html>.
- [2] Bekkeley UPC - Unified Parallel C. <http://upc.lbl.gov/>.
- [3] Erlang programming language. <http://www.erlang.org/>.
- [4] FLAME: Flexible large-scale agent modeling environment.  
<http://www.flame.ac.uk/joomla/>.
- [5] The Haskell programming language. <http://www.haskell.org/haskellwiki/Haskell>.
- [6] Hypergraphdb. <http://www.hypergraphdb.org/>.
- [7] Infinitegraph: The distributed graph database.  
<http://www.infinitegraph.com/>.
- [8] MATLAB - the language of technical computing.  
<http://www.mathworks.com/products/matlab/>.
- [9] METIS library. <http://www-users.cs.umn.edu/karypis/metis/metis.html>.
- [10] Neo4j: Nosql for the enterprise. <http://neo4j.org/>.
- [11] OCaml. <http://caml.inria.fr/ocaml/>.
- [12] OpenACC: Directives for accelerators. <http://http://www.openacc-standard.org/>.
- [13] PETSc: Portable, extensible toolkit for scientific computation.  
<http://www.mcs.anl.gov/petsc/petsc-as/>.
- [14] PVM: Parallel virtual machine. <http://http://www.csm.ornl.gov/pvm/>.
- [15] The R project for statistical computing. <http://www.r-project.org/>.
- [16] Repast Symphony. <http://repast.sourceforge.net/>.
- [17] The Scala programming language. <http://www.scala-lang.org/>.

- [18] ScaLAPACK – scalable linear algebra package.  
<http://www.netlib.org/scalapack/>.
- [19] SUMO: Simulation of urban mobility. <http://sumo.sourceforge.net/>.
- [20] The OpenMP API specification for parallel programming.  
<http://openmp.org/wp/>.
- [21] TransCAD. <http://www.caliper.com/tcovu.htm>.
- [22] TransModeler: Traffic simulation software.  
<http://www.caliper.com/transmodeler/default.htm>.
- [23] USC-SIPI image database. [sipi.usc.edu/database/](http://sipi.usc.edu/database/).
- [24] Wolfram Mathematica: Technical computing software.  
<http://www.wolfram.com/mathematica/>.
- [25] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [26] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.
- [27] Alaa Aljanaby, Emad Abuelrub, and Mohammed Odeh. A survey of distributed query optimization. *Int. Arab J. Inf. Technol.*, 2(1):48–57, 2005.
- [28] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. BOOM analytics: Exploring data-centric, declarative programming for the cloud. In *Proc. EuroSys*, pages 223–236, 2010.
- [29] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Proc. CIDR*, 2011.
- [30] Amazon Elastic Compute Cloud (Amazon EC2).  
<http://aws.amazon.com/ec2/>.

- [31] M. Antoniotti and A. Göllü. SHIFT and SMART-AHS: A language for hybrid system engineering modeling and simulation. In *Proc. DSL*, 1997.
- [32] William Y. Arms, Selcuk Aya, Pavel Dmitriev, Blazej J. Kot, Ruth Mitchell, and Lucia Walle. Building a research library for the history of the web. In *Proc. JCDL*, pages 95–102, 2006.
- [33] A. Vollmy B. Raney, N. Cetin and K. Nagel. Large scale multi-agent transportation simulations. In *Proc. ERSA*, pages 27–31, 2002.
- [34] David A. Bader and Kamesh Madduri. SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS*, pages 1–12, 2008.
- [35] T. Balch. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, Georgia Institute of Technology, 1998.
- [36] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proc. SOCG*, pages 187–197, 1990.
- [37] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, 1999.
- [38] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.
- [39] Dimitri P. Bertsekas and John N. Tsitsiklis. Some aspects of parallel and distributed iterative algorithms - a survey. *Automatica*, 27(1):3–21, 1991.
- [40] G. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.
- [41] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
- [42] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.
- [43] J. Buhl, D. Sumpter, I. Couzin, J. Hale, E. Despland, E. Miller, and S. Simpson. From disorder to order in marching locusts. *Science*, 312(5778):1402–1406, 2006.

- [44] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.
- [45] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [46] Doina Caragea, Adrian Silvescu, and Vasant Honavar. A framework for learning from distributed data using sufficient statistics and its application to learning decision trees. *Int. J. Hybrid Intell. Syst.*, 1(2):80–89, 2004.
- [47] N. Cetin, A. Burri, and K. Nagel. A large-scale agent-based traffic microsimulation based on queue model. In *Proc. STRC*, 2003.
- [48] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proc. PLDI*, pages 363–375, 2010.
- [49] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, 1978.
- [50] Edward Y. Chang, Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, and Hang Cui. PSVM: Parallelizing support vector machines on distributed computers. In *Proc. NIPS*, 2007.
- [51] Hung Chih Yang, Ali Dasdan, Ruey Lung Hsiao, and Douglas Stott Parker Jr. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *Proc. SIGMOD*, pages 1029–1040, 2007.
- [52] C. Choudhury, M. Ben-Akiva, T. Toledo, A. Rao, and G. Lee. NGSIM cooperative lane changing and forced merging model. Technical report, Federal Highway Administration, 2006. FHWA-HOP-07-096.
- [53] C. Choudhury, T. Toledo, and M. Ben-Akiva. NGSIM freeway lane selection model. Technical report, Federal Highway Administration, 2004. FHWA-HOP-06-103.
- [54] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1994.

- [55] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [56] Shannon Coffey, Liam Healy, and Harold Neal. Applications of parallel processing to astrodynamics. *Celestial Mechanics and Dynamical Astronomy*, 66:61–70, 1996.
- [57] T. Condie, D. Chu, J. Hellerstein, and P. Maniatis. Evita Raced: Metacom-pilation for declarative networks. In *Proc. VLDB*, 2008.
- [58] Cornell Database Group. The BRASIL language specification, 2010. <http://www.cs.cornell.edu/bigreddata/games/simulations.php>.
- [59] I. Couzin, J. Krause, N. Franks, and S. Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433(7025):513–516, 2005.
- [60] David Crandall, Andrew Owens, Noah Snavely, and Daniel P. Huttenlocher. Discrete-Continuous optimization for large-scale structure from motion. In *Proc. CVPR*, 2011.
- [61] J.T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.
- [62] Samir Ranjan Das, Richard Fujimoto, Kiran S. Panesar, Don Allison, and Maria Hybinette. GTW: A time warp system for shared memory multi-processors. In *Winter Simulation Conference*, pages 1332–1339, 1994.
- [63] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.
- [64] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.
- [65] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
- [66] Inderjit S. Dhillon, Pradeep D. Ravikumar, and Ambuj Tewari. Nearest neighbor based greedy coordinate descent. In *Proc. NIPS*, pages 2160–2168, 2011.

- [67] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *Proc. SSTD*, pages 189–207, 2009.
- [68] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proc. HPDC*, pages 810–818, 2010.
- [69] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *Proc. eScience*, pages 277–284, 2008.
- [70] Gal Elidan, Ian McGraw, and Daphne Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proc. UAI*, 2006.
- [71] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
- [72] Ugo Erra, Bernardino Frola, Vittorio Scarano, and Iain Couzin. An efficient GPU implementation for large scale individual-based simulation of collective behavior. In *Proc. HIBI*, pages 51–58, 2009.
- [73] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):216–231, 2005.
- [74] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Comput. Surv.*, 16(2):153–185, June 1984.
- [75] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. A framework for the parallel processing of datalog queries. In *Proc. SIGMOD*, pages 143–152, 1990.
- [76] B. Gerkey, R. Vaughan, and A. Howard. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proc. ICAR*, 2003.
- [77] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *Proc. AISTATS*, 2009.

- [78] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. SIGMOD*, pages 102–111, 1990.
- [79] William Gropp. Learning from the success of MPI. In *Proc. HiPC*, pages 81–94, 2001.
- [80] Daniel Grünbaum. BEHAVIOR: Align in the sand. *Science*, 312(5778):1320–1322, 2006.
- [81] Apache Hadoop. <http://hadoop.apache.org/>.
- [82] Per Brinch Hansen. An evaluation of the message-passing interface. *SIGPLAN Notices*, 33(3):65–72, 1998.
- [83] Taher H. Haveliwala. Topic-sensitive pagerank. In *WWW*, pages 517–526, 2002.
- [84] D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, 2000.
- [85] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.
- [86] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2-4):485–500, 2000.
- [87] Dac Phuong Ho, The Duy Bui, and Nguyen Luong Do. Dividing agents on the grid for large scale simulation. pages 222–230, 2008.
- [88] A. Hobeika and Y. Gu. TRANSIMS: The next generation planning/simulation model. In *Proc. Int. Conf. on Urban Transport and the Environment*, pages 143–151, 2004.
- [89] Bryan Horling, Roger Mailler, and Victor R. Lesser. FARM: A scalable environment for multi-agent development and evaluation. In *Proc. SELMAS*, pages 225–242, 2003.
- [90] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and

- emerging applications: an interactive tutorial. In *SIGMOD Conference*, pages 1213–1216, 2011.
- [91] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M.J. Franklin, P. Abbeel, and A.M. Bayen. Scaling the mobile millennium system in the cloud. In *Proc. SOCC*, 2011.
- [92] Maria Hybinette, Eileen Kraemer, Yin Xiong, Glenn Matthews, and Jaim Ahmed. SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In *Winter Simulation Conference*, pages 926–933, 2006.
- [93] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
- [94] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, pages 59–72, 2007.
- [95] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proc. SIGMOD*, pages 987–994, 2009.
- [96] Joshua Epstein and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. The MIT Press, 1996.
- [97] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel programming with message-driven objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
- [98] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. In *Proc. ICDM*, pages 229–238, 2009.
- [99] Nicholas T. Karonis, Brian R. Toonen, and Ian T. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.
- [100] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.
- [101] Christoph Koch. On the complexity of nonrecursive xquery and func-

- tional query languages on complex values. In *Proc. PODS*, pages 84–97, 2005.
- [102] Giorgios Kollias, Efstratios Gallopoulos, and Daniel B. Szyld. Asynchronous iterative computations with web information retrieval structures: The pagerank case. In *Proc. PARCO*, pages 309–316, 2005.
- [103] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proc. PLDI*, pages 211–222, 2007.
- [104] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
- [105] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1), 2009.
- [106] Ian X. Y. Leung, Pan Hui, Pietro Liò, and Jon Crowcroft. Towards real-time community detection in large networks. *Physical Review E*, 79:066107, 2009.
- [107] Frank Lin and William W. Cohen. Semi-Supervised classification of network data using very few labels. In *Proc. ASONAM*, pages 192–199, 2010.
- [108] H. Liu, W. Ma, R. Jayakrishnan, and W. Recker. Large-scale traffic simulation through distributed computing of paramics. Technical Report UCB-ITS-PRR-2004-42, University of California, Berkeley, 2004.
- [109] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proc. SOCC*, pages 51–62, 2010.
- [110] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proc. SIGMOD*, pages 97–108, 2006.
- [111] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proc. SOSR*, pages 75–90, 2005.

- [112] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *Proc. SIGCOMM*, pages 289–300, 2005.
- [113] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proc. UAI*, pages 340–349, 2010.
- [114] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
- [115] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: A multiagent simulation environment. *Simulation*, 81, 2005.
- [116] Gang Luo, Jeffrey F. Naughton, and Curt J. Ellmann. A non-blocking parallel spatial join algorithm. In *Proc. ICDE*, pages 697–705, 2002.
- [117] Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: the darpa hpcs language project. *Parallel Processing Letters*, 17(1):89–102, 2007.
- [118] S. M. Mahajan and V. P. Jadhav. A survey of issues of query optimization in parallel databases. In *Proc. ICWET*, pages 553–554, 2011.
- [119] 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 *Proc. SIGMOD*, pages 135–146, 2010.
- [120] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.*, 18(4):423–434, 1993.
- [121] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.
- [122] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proc. CIDR*, 2013.
- [123] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

- [124] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Working Papers 96-06-042, Santa Fe Institute, 1996.
- [125] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proc. SIGMOD*, pages 634–645, 2005.
- [126] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proc. UAI*, pages 467–475, 1999.
- [127] K. Nagel and M. Rickert. Parallel implementation of the TRANSIMS micro-simulation. *Parallel Computing*, 27(12):1611–1639, 2001.
- [128] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM*, 40(2):304–333, 1993.
- [129] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. HOG-WILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. NIPS*, 2011.
- [130] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. *PVLDB*, 2(2):1426–1437, 2009.
- [131] Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proc. PODS*, pages 29–38, 1988.
- [132] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., 1988.
- [133] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.
- [134] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of parallelism in algorithms. In *Proc. PLDI*, pages 12–25, 2011.

- [135] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. OSDI*, pages 293–306, 2010.
- [136] Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76:036106.
- [137] Lavanya Ramakrishnan, Keith R. Jackson, Shane Canon, Shreyas Cholia, and John Shalf. Defining future platform requirements for e-Science clouds. In *Proc. SOCC*, pages 101–106, 2010.
- [138] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. HPCA*, pages 13–24, 2007.
- [139] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. Horton: Online query execution engine for large distributed graphs. In *Proc. ICDE*, pages 1289–1292, 2012.
- [140] M. Scheutz, P. Schermerhorn, R. Connaughton, and A. Dingler. SWAGES—an extendable parallel grid experimentation system for large-scale agent-based alife simulations. In *Proc. Artificial Life X*, pages 412 – 418, 2006.
- [141] D. Schoenwald, D. Barton, and M. Ehlen. An agent-based simulation laboratory for economics and infrastructure interdependency. In *Proc. American Control Conference*, volume 2, pages 1295–1300, 2004.
- [142] D. Schrank and T. Lomax. The 2009 urban mobility report. Technical report, Texas Transportation Institute, 2009.
- [143] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with Sets; an Introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
- [144] Yangqiu Song, WenYen Chen, Hongjie Bai, Chih Jen Lin, and Edward Y. Chang. Parallel spectral clustering. In *Proc. ECML/PKDD (2)*, pages 374–389, 2008.
- [145] Jin soo Kim, Soonhoi Ha, and Chu Shik Jhon. Relaxed barrier synchronization for the BSP model of computation on message-passing architectures. *Information Processing Letters*, 66, 1998.

- [146] Benjamin Sowell, Alan J. Demers, Johannes Gehrke, Nitin Gupta, Haoyuan Li, and Walker M. White. From declarative languages to declarative processing in computer games. In *Proc. CIDR*, 2009.
- [147] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbms: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
- [148] Dan Suciu. *Parallel programming languages for collections*. PhD thesis, University of Pennsylvania, 1995.
- [149] Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen, and Carsten Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30, 2008.
- [150] Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *Proc. ICDT*, pages 140–154, 1992.
- [151] Cheng Tao Chu, Sang Kyun Kim, Yi An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for machine learning on multicore. In *Conf. NIPS*, pages 281–288, 2006.
- [152] Jens Teubner. *Pathfinder: XQuery Compilation Techniques for Relational Database Targets*. PhD thesis, Technische Universität München, 2006.
- [153] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [154] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.
- [155] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.
- [156] Y. Wang, H. Bai, M. Stanton, W. Chen, and E. Chang. PLDA: Parallel latent dirichlet allocation for large-scale applications. In *Proc. AAIM*, pages 301–314, 2009.

- [157] Y. Wen. *Scalability of Dynamic Traffic Assignment*. PhD thesis, Massachusetts Institute of Technology, 2008.
- [158] Walker M. White, Alan J. Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportion. In *Proc. SIGMOD*, pages 31–42, 2007.
- [159] Walker M. White, Christoph Koch, Johannes Gehrke, and Alan J. Demers. Better scripts, better games. *Commun. ACM*, 52(3):42–47, March 2009.
- [160] Walker M. White, Benjamin Sowell, Johannes Gehrke, and Alan J. Demers. Declarative processing for computer games. In *Proc. SIGGRAPH Sandbox Symposium*, 2008.
- [161] Jason Wolfe, Aria Haghighi, and Dan Klein. Fully distributed EM for very large datasets. In *Proc. ICML*, pages 1184–1191, 2008.
- [162] Q. Yang and H. Koutsopoulos. A microscopic traffic simulator for evaluation of dynamic traffic management systems. *Transportation Research Part C: Emerging Technologies*, 4(3):113–129, 1996.
- [163] Q. Yang, H. Koutsopoulos, and M. Ben-Akiva. A simulation laboratory for evaluating dynamic traffic management systems. In *Annual Meeting of Transportation Research Board*, 1999. TRB Paper No. 00-1688.
- [164] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proc. IISWC*, pages 198–207, 2009.
- [165] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. OSDI*, pages 1–14, 2008.
- [166] A.N. Yzelman and Rob H. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience*, 24(5):533–553, 2012.
- [167] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. HotCloud*, 2010.

- [168] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, pages 1–8, 2009.
- [169] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: A distributed framework for prioritized iterative computations. In *Proc. SOCC*, 2011.
- [170] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *Proc. IPDPS*, 2004.
- [171] Kaihua Zhu, Hang Cui, Hongjie Bai, Jian Li, Zhihuan Qiu, Hao Wang, Hui Xu, and Edward Y. Chang. Parallel approximate matrix factorization for kernel methods. In *Proc. ICME*, pages 1275–1278, 2007.