A Distributed Approach to EpiFast using Apache Spark

Vijayasarathy Kannan

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Madhav V. Marathe, Chair
Jiangzhuo Chen
Anil Kumar S. Vullikanti
Achla Marathe

June 1, 2015
Blacksburg, Virginia

Keywords: computational epidemiology, parallel programming, distributed computing

A Distributed Approach to EpiFast using Apache Spark

Vijayasarathy Kannan

(ABSTRACT)

EpiFast is a parallel algorithm for large-scale epidemic simulations, based on an interpretation of the stochastic disease propagation in a contact network. The original EpiFast implementation is based on a master-slave computation model with a focus on distributed memory using message-passing-interface (MPI). However, it suffers from few shortcomings with respect to scale of networks being studied. This thesis addresses these shortcomings and provides two different implementations: *Spark-EpiFast* based on the Apache Spark big data processing engine and *Charm-EpiFast* based on the Charm++ parallel programming framework. The study focuses on exploiting features of both systems that we believe could potentially benefit in terms of performance and scalability. We present models of EpiFast specific to each system and relate algorithm specifics to several optimization techniques. We also provide a detailed analysis of these optimizations through a range of experiments that consider scale of networks and environment settings we used. Our analysis shows that the Spark-based version is more efficient than the Charm++ and MPI-based counterparts. To the best of our knowledge, ours is one of the preliminary efforts of using Apache Spark for epidemic simulations. We believe that our proposed model could act as a reference for similar large-scale epidemiological simulations exploring non-MPI or MapReduce-like approaches.

Acknowledgements

I would never have been able to complete my thesis without the contributions of many outstanding people.

Foremost, I would like to express my sincerest gratitude to my advisor, Dr. Madhav V. Marathe for his continuous support of my program of study and for giving me an opportunity to work with him which made my experience at Virginia Tech complete.

I am grateful to Dr. Jiangzhuo Chen, who mentored me throughout my thesis. By closely working with him on several other projects, I learnt a lot through his insight and knowledge. This thesis would not have been possible without his expertise and encouragement.

I would also like to thank the rest of my committee members, Dr. Anil Kumar S. Vullikanti and Dr. Achla Marathe, for their valuable comments about my thesis.

I greatly appreciate the time and support of my colleagues at NDSSL and all my friends at Virginia Tech who helped me through the last two years.

I would also like to acknowledge the funding received for my research from NSF NEtSE, CNIMS, V&V, DIBBS and CINET.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Epidemiological simulation involves the mathematical modeling of how infectious diseases progress during an epidemic among the population under study [1]. Such simulations help understand the rate of disease spread and inform public health interventions. They also help understand the dynamics of the spread in various contexts such as estimating the load on medical services, risk assessment, intervention policies, etc. Thus, such studies typically focus on understanding the spread of disease through a given population.

Computational epidemiology refers to the development and use of computer-based models to understand the diffusion of diseases through contact networks. Contact networks denote the population under study and encompass the following: individuals in the population and their attributes (such as demographics, disease states, etc.); the interactions between individuals and attributes of these interactions (such as duration, location, etc.). Computational epidemiology considers spatio-temporal spread of an infectious disease in a population. One of the important factors that influences disease spread is the underlying structure of the interactions between individuals. In the past, there have been computational epidemiology models (such as compartmental models [2, 3, 4]), that assume partitioned sub-populations (for example, by age) and a regular interaction structure within and between those sub-populations. This is often restrictive and is not completely representative of real-world contact networks. It has been identified that an interaction based computational model plays a major role in developing public health policies. This presents the need for *interaction-based computational epidemiology* models. Such models consider the detailed representation of the underlying structure of the contact networks, within and across disease progression and transmission models. Moreover, these models are computational representations of policies that can be implemented in real-world scenarios.

The literature has accounted for several efforts in developing computational models for epi-

demiological simulations. Efforts in networked epidemiology include analytical techniques over graphs [5, 6], developing individual-based models using statistics [7] and realistic representations of socail contact networks such as EpiSims [8], EpiSimdemics [9], EpiFast [10] are few of the simulation tools that have approached the problem of epidemiological simulations from different perspectives and provided plausible and effective solutions.

This thesis focuses on one of the simulation tools, *EpiFast*. EpiFast is a parallel algorithm based on a novel interpretation of the stochastic disease propagation in a contact network. A typical epidemiological study poses several challenges such as scale, heterogeneity and dynamics of the network. These factors justify the need for a fast, high-performance computing based solution. EpiFast aims to address these challenges and furthermore represent complex interventions and public health policies in a practical setting.

1.2 Contribution

In this thesis, we present *Charm-EpiFast* and *Spark-EpiFast*, two distributed approaches to EpiFast. This work builds upon the existing MPI-based implementation of EpiFast with focus on scale of the contact networks being studied. Below is a summary of our contributions.

- **Charm-EpiFast**: a prototype of EpiFast based on the *Charm++* parallel programming framework. This is a preliminary effort on the study of parallel programming frameworks that can be used for large-scale epidemic simulations. We provide a Charm++ model for EpiFast and an intervention-less implementation of the same. We also include details of an experimental study of some of the optimizations provided by Charm++ with respect to EpiFast. Results and observations from these experiments are also provided.
- **Spark-EpiFast**: the major component of the thesis. Spark-EpiFast is based on a distributed setting built over *Apache Spark*. We provide a MapReduce-like model for simulations in EpiFast. We present a detailed study of Apache Spark. We discuss our considerations of Spark for EpiFast and present details of our approach including design and implementation, and, features that have been identified as possible optimizations with respect to EpiFast. Unlike Charm-EpiFast, this is a more sophisticated version with support for a majority of configuration options that original EpiFast provides. A detailed experimental analysis of Spark-EpiFast is also included.

With respect to the above contributions, we also identify their limitations and possible strategies that can further enhance performance.

Thesis organization

The rest of the thesis is organized as follows.

- Chapter 2 provides a survey of existing epidemiological simulation tools (specifically focusing on parallel and distributed simulations) and applications of the two major programming systems that we use: *Charm++* and *Apache Spark*.
- Chapter 3 provides a comprehensive description of the epidemic simulation problem and focuses on the approach of *EpiFast* to solving it.
- Chapter 4 describes the Charm++ parallel programming system and our first development effort in using Charm++ for EpiFast, *Charm-EpiFast*.
- Chapter 5 follows a similar outlook except that it focuses on the Apache Spark version of *EpiFast*, *Spark-EpiFast*.
- Chapter 6 presents our experimental analysis and performance evaluation of both *Charm-EpiFast* and *Spark-EpiFast*.
- Chapter 7 discusses avenues for improvements of both tools from functional and performance perspectives.
- Chapter 8 provides a brief discussion of the three parallel programming frameworks we have studied in this thesis: MPI, Charm++ and Apache Spark, and the potential of using Charm++ and Apache Spark for epidemic simulations in general.
- Chapter 9 provides concluding remarks about the thesis.

Chapter 2

Related Work

2.1 Epidemic Simulations

Epidemic simulations have been extensively used in the study of the evolution of epidemics. This chapter highlights some of the efforts on epidemic models with a focus on parallel simulations of such models. In this section, specifically, we focus on some of the efforts in parallel and distributed epidemiological simulations. *Agent-based models* represent entities of the population as *agents* and account for the interactions between the agents to determine the dynamics of the epidemic under study simulate epidemics. Epidemiological research in recent years has seen few mathematical models to study the evolution of epidemics. Among them are *compartmental* models [2, 3, 4] that divide the population into compartments and use differential equations to study the evolution of the disease spread. They assume a perfect mixing among the population and are often unrealistic. Other models that assume a particular structure within the contact network [5, 11, 12] model disease spreads as stochastic propagation processes and derive results analytically. These models suffer from their inability to handle irregularities in the contact networks. *EpiFast* is very closely related to *EpiSims* [8] and *EpiSimdemics* [9]. While they differ in some aspects (for example, EpiSims is a discrete event simulation while EpiFast is a discrete time simulation), all these models are *individual*-based network models and are capable of handling large networks. From a modeling perspective, [9] uses a person-location representation as opposed to a person-person representation adopted by EpiFast.

In the last few years, there have been several attempts to efficiently perform large-scale epidemic simulations on compute clusters. These can categorized based on few distinct parameters such as memory model, programming model, etc. The memory model of such a simulation refers to the usage of either *shared* or *distributed* memory. The literature has seen several attempts on both approaches.

Shared-memory programming environments typically involve multithreading, OpenMP and

CUDA. *FluTE* [13] is an influenza epidemic simulation tool based on *OpenMP*. *FluTE* provides a parallel simulation of large populations on a shared memory multiprocessor system. It has the ability to simulate the entire US continental population. There have been attempts on *graphics processing units* (GPUs). Such models typically use *CUDA* programming framework and take advantage of reduced latency in shared memory of GPUs. Instances of such attempts include [14, 15, 16, 17]. All these simulation tools are based on the shared-memory model. Though they take advantage of the reduced latency, they are limited by the amount of shared memory available, thereby restricting them for extremely large-scale real-world processing. Other shared-memory based models include [18, 19] that are able to scale to millions of individuals but are limited by computing environments and language specific constraints.

On the other end of the memory spectrum are the distributed memory simulation models. *Distributed arrays* are commonplace in such models. Unified Parallel C [20], GlobalArray [21] and *message-passing interface* (MPI) are some examples of systems that support *distributed shared* arrays. Such systems typically allocate a global memory space and share them among multiple threads or processes. Intuitively, such systems suffer from problems such as memory consistency and scalability (due to the amount of memory available). A popular example of an epidemic simulation based on this model is MATSim [22]. Other models such as PDES-MAS [23] and MACE3J [24] focus on parallel execution of coarse-grained simulation agents on distributed memory. In these models, threads migrate over a distributed array to make all data accesses local to them for realizing data-locality. Repast HPC [25] is another large-scale model implemented over MPI. EpiFast [10], EpiSims [8] and EpiSimdemics [9] fall under this category as well. Implementation of these algorithms are developed over the *Simdemics* framework [26] using a distributed memory model. MASS [27, 28] is another model that uses a composition of distributed arrays and multi-agents through a parallel-computing library. Other simulation tools that rely on distributed systems are Swarm [29] and NetLogo [30].

There have been other simulation tools that are based on unique programming models such as: [31] is an attempt at large-scale behavioral simulation on the MapReduce framework; it provides an engine called *BRACE* (Big Red Agent-based Computation Engine) that extends the MapReduce framework for efficiently simulating across clusters and the *Indemics* [32] modeling environment which uses a relational database management system for supporting adaptive behavioral interventions and for supporting interactions between HPC simulations and decision making processes.

Table 2.1 provides a summary of the models discussed in this section.

2.2 Distributed Programming Frameworks

In this section, we highlight the potential of *Charm++* and *Apache Spark* for epidemic simulations and provide a survey of applications based on these systems. We focus only on these

Memory Model	Simulation Tool	Programming Model
Shared	*FluTE* [13]	OpenMP
	Lysenko et. al [14]	Stream Programming Model: GPU
	Perumalla et. al [15]	Multi-GPU and PThreads
	Parker et. al [18]	Multi-threading in Java
Distributed	MatSim [22]	Java, distributed arrays
	PDES-MAS [23]	Communication Logical Processes
	MACE3J [24]	Java MAS simulation
	Repast HPC [25]	C++ and MPI
	EpiSims [8]	C++ *mpich* library
	Indemics [32]	Java and Relational Database
	EpiSimdemics [9]	C++/MPI and Charm++ porting
	EpiFast [10]	C++/MPI

Table 2.1: Summary of parallel/distributed epidemic simulation tools

two programming frameworks as they form the core of the contributions of this thesis.

Charm++ parallel programming system

Charm++ [33] is a parallel programming system that comes with an adaptive runtime system enhancing developer productivity and providing capabilities for developing large-scale applications on clusters of systems. Below are some features that make Charm++ suitable for parallel applications:

1. overlap of communication and computation due to the parallel-objects representation;
2. modularized control flow with independent modules interacting with each other allowing interleaved execution;
3. load balancing modules that support dynamic re-allocation of objects to resources allowing applications to adapt to changing loads;
4. capability to support MPI at the machine layer which makes porting of MPI-based applications much easier.

There have been several applications developed in recent years that use the Charm++ framework across a variety of domains such as molecular biology, epidemic simulations, integer programming, etc. We describe some of them here.

- *NAMD* [34] is a parallel high-performing molecular dynamics application that directly takes advantage of features 1 and 3 mentioned above.
- *Charm++ N-Body Gravity Simulator* (ChaNGa) [35, 36] is a cosmological simulator that studies formation of large structures in the universe.
- *Parallel Stochastic Integer Programming* (PSTIP) [37] is stochastic technique used in branch-and-bound integer programming. This uses Charm++ to explore branch-and-bound tree vertices and is scalable to hundreds of physical cores.
- *Contagion in Social Networks - EpiSimdemics* [38] is an agent-based simulation of contagions in large social networks. This implements several graph partitioning techniques. It leverages Charm++'s ability to migrate objects at runtime to achieve high performance on extremely large amounts of data. It also uses a novel topology aggregation methodology called *TRAM* [39]. TRAM supports a fine-grained communication pattern and aggregation of messages that is aware of the topological structure of the parallel environment. This application scales to thousands of cores and hundreds of millions of individuals. The performance benefits realized in this particular effort has spawned further research into using Charm++ for epidemiological simulations. One of contributions of this thesis, *Charm-EpiFast* is a direct spin-off of such an attempt.

Apache Spark distributed processing engine

Apache Spark [40] is a fast engine for big data processing. Since its inception it has found itself the crux of a variety of application domains. We outline some of them here.

- *Real-time processing* involving applications such as *Stratio* [41] that stream real-time data and apply machine-learning models to analyze them.
- *Data integration and data processing* that form the core of several data-fusion systems including engines that process noisy data [42], geospatial applications [43], and a distributed image processing platform [44].
- *Analytics* using Spark's *MLlib* library including contributions of efficient algorithms for random forest [45] and matrix factorization [46].
- *Genomics* applications including a recent attempt to analyze genomics data to identify cancer through *ADAM* [47] by using distributed data processing and pipe-lining stages of the application.

Apache Spark provides in-memory abstractions to realize performance benefits. Iterative algorithms that rely heavily on reusing data across iterations can take advantage of Spark's persistence capabilities. There are claims that Spark is faster than Hadoop by a factor

of 100× in-memory and 10× on disk [40]. The major contribution of this thesis, *Spark-EpiFast*, to the best of our knowledge, is one of the preliminary efforts of using Spark for large-scale epidemiological simulations. Specifically, for *EpiFast* which is inherently iterative and relies on preserving partitions of contact networks across iterations, we believe that Spark's memory abstractions and persistence capabilities are well suited.

Chapter 3

Epidemic Simulation Using EpiFast

In this chapter we describe the epidemic simulation problem in detail and how EpiFast approaches it.

3.1 The Epidemic Simulation Problem

In general, an epidemic simulation consists of simulating the dynamics of disease spread in a population. In this thesis, we focus on an agent based model where the population is modeled as a social contact network of individuals. In such a setting, an epidemic simulation computes

- number of infected individuals on a given day,
- list of infected individuals,
- the day a given individual got exposed or infected, and,
- set of individuals who infected a given individual.

Epidemic simulations typically require the following types of information.

- **Population**
  - a contact graph - basically the population under study including unique identifiers to individuals, their contacts (set of individuals who come in contact with this individual)
  - a disease and its characteristics such as transmissibility

  - characteristics of each individual with respect to the disease, includes details such as
    * likelihood of the individual getting infected,
    * likelihood of the individual infecting his contacts,
    * amount of time the individual would remain infected or equivalently the amount of time he takes to recover

- **Interventions**

  - self-interventions
  - public health interventions

  Interventions affect the vulnerability/infectivity of the individuals or their activities (thus their local network structures) in order to mitigate the spread of the disease.

- **Initial conditions** representing the health state of each individual at the beginning of the epidemic.

3.2 Problem Formalization

Stochastic graphical dynamical systems are used to study problems related to the dynamic behavior of a contact network. In [48], the authors propose a formal and generic model for social networks. Similarly, [32] provides a formalization of an interactive epidemic simulation based on *Graph Dynamical Systems (GDS)* [49, 50].

Let D be the domain of possible states of an individual and L be the domain of possible labels for contacts. In a typical epidemic simulation problem, D includes individuals' demographic attributes, health states and behavioral states. The basis for formalizing any GDS includes the following components as put forth in [51]. We reiterate the components here for clarity.

- $G(V, E)$ - the **underlying dependency graph**,
- $F = \{f_1, f_2, ..., f_n\}$ - a set of **local transition functions**, where f_i is a function for node $v_i \in V$,
- π - a specific **order** for applying f_i on the nodes,
- **global transition function** - determines the dynamic behavior of the system,
- $S = (c_1, c_2, ..., c_n)$ - a **configuration** of the system, where $c_i \in D$ denotes the value of state of node v_i, and,

- **phase space** - a directed graph (nodes are configurations) which indicates the movement from configuration C to configuration C' in a single transition (edges in the graph).

During a simulation of such a system, functions f_i are applied to nodes v_i in the order specified by π. This is equivalent of moving the system from one configuration to another. An important property of these systems is that at some point the transition functions *cycle* through a finite sequence of configurations. They always end at a **fixed point** meaning the global transition functions applied on a given configuration produces the same configuration.

Tools such as EpiSims [8], EpiSimdemics [9] and EpiFast [10] are distinct ways to model stochastic graphical dynamical systems. We now formalize EpiFast using the above mentioned components.

- **D** is composed of the four health states - S (susceptible), E (exposed), I (infectious) and R (recovered), vulnerability/infectivity and diagnosis of individuals.
- **L** includes the type and duration of the contacts.
- The underlying ***social contact network***, $\mathbf{G}(\mathbf{V}, \mathbf{E})$ where,

$$V = \{v_1, v_2, ..., v_n\},$$

$$E = \{e_1, e_2, ..., e_m\}, E \subseteq (V \times V)$$

are the vertex and edge sets respectively, and, n is the number of individuals in the population and m is the number of contacts among the individuals. G is directed and edge-weighted (each edge has an associated weight - usually the duration of the contact). Figure 3.1 shows a simple contact network with $V = \{1, 2, 3, 4, 5, 6\}$ and the edges weighted with the duration of contact (in seconds) between adjacent nodes. Edge (u, v) with weight $w(u, v)$ denotes that nodes u and v have been in contact for a duration of $w(u, v)$ units of time. $w(u, v)$ represents the period of time during which the disease may transmit from u to v. Edges in G are considered bi-directional - given an edge (u, v), the disease can transmit either from u to v or from v to u. $p(w(u, v))$ denotes the probability that the disease transmits from u to v.

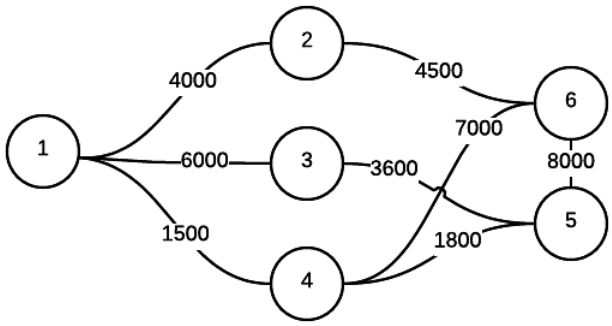

Figure 3.1: An example contact network with 6 individuals and 7 contacts each labeled with duration of contact.

- ***local state transitions*** governed by *SEIR* disease model. More specifically, the local state transitions are defined by the ***within-host transition*** functions. The *within-host transitions* in SEIR model accounts for changes in health states of individuals without the influence of external factors such as contacts with other individuals.

  The following information is associated with an individual v.

  - health state on day t, $\theta(v,t)$
  - incubation period $\triangle t_E(v)$ - the number of days v remains exposed. During this period, v is not infectious.
  - infectious period $\triangle t_I(v)$ - the number of days v remains infectious before moving to the recovered state.

  The basic transitions of an individual v are as follows.

  - v is in *susceptible* state until he becomes *exposed*
  - when v becomes exposed, he remains so for $\triangle t_E(v)$ days
  - then v moves to *infectious* state. He remains infectious for $\triangle t_I(v)$ days
  - finally, v moves to *recovered* state and remains so permanently.

  Figure 3.2 illustrates the within-host disease progression. We represent the health state of an individual v during the epidemic simulation by,

$$\theta(\mathbf{v}) = (\mathbf{t_{S->E}}(\mathbf{v}), \mathbf{t_{E->I}}(\mathbf{v}), \mathbf{t_{I->R}}(\mathbf{v}))$$

  Note that,

  - $t_{S->E}$: the day v becomes *exposed*,

- $t_{E->I}(v)$: $t_{S->E}(v) + \triangle t_E(v)$, the day v becomes *infectious*, and,
- $t_{R->R}(v)$: $t_{E->I}(v) + \triangle t_I(v)$, the day v becomes *recovered*

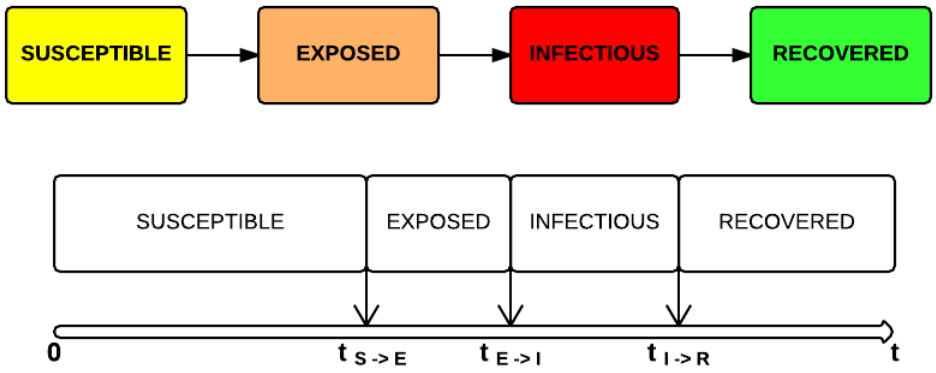

Figure 3.2: SEIR model: Within-host disease progression depicting the four health states - Susceptible, Exposed, Infectious and Recovered, and incubation and infectious periods.

- ***global transition function*** governed by ***between-host transmission*** functions that determine transmission from an infectious to a susceptible individual. The *between-host transmission* accounts for the changes in health states of individuals due to their contacts with other individuals. This takes into account the stochastic nature of the disease model, i.e., the disease propagates probabilistically along the edges of the network. The between-host transmission is characterized as follows.

  - disease only transmits from an *infectious* node to a *susceptible* node,
  - on any day, disease transmission from node u to node v occurs with probability

$$\mathbf{p(w(u,v)) = 1 - (1 - r)^{w(u,v)}}$$

    where,

    * node u is infectious,
    * node v is susceptible,
    * $w(u, v)$ is the duration of contact between u and v, and,
    * r is the probability of disease transmission for a contact of one unit time.

  It is important to note that the SEIR model assumes *independence* as mentioned in [10]. This means that,

– disease transmission from node u to node v is completely independent of the transmission from a node u' to node v, and

– an infected node u transmits the disease to each neighbor v, independent of the other neighbors of u.

Figure 3.3 illustrates between-host disease progression for a period of 5 days. In this example, there are *5* nodes and *4* edges (represented by dashed lines) in the contact graph. Also, for all nodes, $\triangle t_E = 0$ and $\triangle t_I = 2$. Solid edges denote that the disease transmits along that edge.

– Initially (day-0), node *1* is *infectious*.

– On day-1, *1* transmits the disease to node *2*; so, both *1* and *2* are infectious.

– On day-2, *1* and *2* transmit the disease to *3* and *4* respectively and *1* moves to *recovered* state.

– On day-3, *3* transmits the disease to *5*. Nodes *3, 4, 5* are *infectious*.

– Day-4 sees nodes *3* and *4* move to *recovered* state.

– On day-5, node *5* moves to *recovered* state.

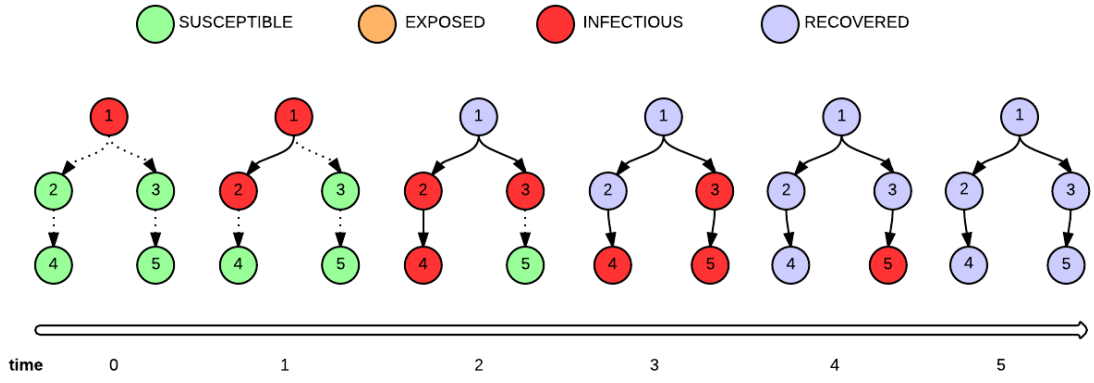

Figure 3.3: SEIR model: Between-host disease progression. Illustrates state changes of individuals over a duration of 6 days.

- **W**, representing a pre-determined schedule (such as when individuals come into contact) at each time step (typically, a day) in the simulation.

A typical simulation of EpiFast starts with an initial configuration - a set of infectious nodes. During the simulation, local transition functions (within-host transitions) are applied on the nodes, global transition function (between-host transmission) probabilistically determines

state transitions leading to new configurations. Like any graph dynamical system, EpiFast also ends at a fixed point where the transition functions no more produce new configurations, i.e., no node will remain *infectious*.

Interventions

Interventions are a major component of EpiFast. Interventions supported by EpiFast can be categorized into

1. pharmaceutical (PI) and non-pharmaceutical (NPI)
2. adaptive and non-adaptive.

EpiFast [10] provides a description of interventions as follows. *Pharmaceutical* interventions typically refer to administering antivirals, vaccines and antibiotics. *Non-pharmaceutical* interventions involve actions that change the dynamics of the social contact network. They do not administer drugs but typically enforce social distancing - closure of different locations/activities such as schools, separation of individuals (otherwise known as *quarantine*) and *sequestration*. *Adaptive* interventions depend on the progress of the epidemic. They are not predetermined, i.e., whether they will be applied, when they will be applied and, whom they are applied to is determined during the simulation. *Non-adaptive* interventions on the other hand, are applied at pre-determined times during the epidemic.

From an implementation point of view, interventions are specified as either of the following.

- changing the infectivity/transmissibility of individuals: no changes in the people-people interactions which means that edges are still preserved and could still potentially transmit the disease. However, nodes are administered vaccines (and other pharmaceutical measures) to mitigate disease transmission to their contacts.
- changing the behaviors of the individuals: these change the underlying social contact network by removing (or reducing) contacts.

EpiFast [10] mentions that non-adaptive interventions are implemented as changes to node or edge attributes before the simulation begins. Adaptive interventions are implemented through *trigger* conditions determined by the health states of the individuals on the current simulation day.

3.3 EpiFast Algorithm

In this section, we formally describe the *EpiFast* algorithm. First, we describe the sequential version of the algorithm to help understand the approach and then move on to the parallelized

version.

Both the sequential and parallel versions take as input the following.

- G: contact network,
- $\triangle\tau$: incubation and infectious period durations of each individual,
- A: interventions, and,
- Λ: set of initial conditions

The versions return the following as output.

- τ: health state transition times of all people, and,
- Π: set of infectors who transmit the disease to the infected people.

3.3.1 Sequential EpiFast Algorithm

The sequential EpiFast algorithm proceeds in two phases: within-host transition and between-host transmission. The simulation runs for T days. On each day, first, the within-host transition phase occurs. During this phase, the health state $\theta(v)$ of each node v in the contact network is updated according to $\tau(v)$. The between-host transmission phase occurs next. During this phase, for each *infectious* node u we consider each of its neighbors. If a neighbor v is *susceptible*, then transmission $u \rightarrow v$ occurs with $p(w(u, v))$. If transmission $u \rightarrow v$ does occur, the health state of v is updated accordingly, i.e., marked as *exposed*. The simulation ends after day-T and outputs the health state transition times of all individuals and the infectors who transmit the disease to the infected individuals. EpiFast [10] provides the complete sequential version of the algorithm.

3.3.2 Parallel EpiFast Algorithm

This version attempts to parallelize computation wherever possible. Here, we describe few key characteristics of the algorithm.

Computational model

The main characteristic of this version of the algorithm is that it is based on a *master-slave* computational model. This means that there is a dedicated process, designated as the

master and several other processes designated as *slaves*. Each type of process has its own responsibilities. The *master* coordinates the communication between *slaves*. Each of the *slaves* work on a small subset of the nodes in the contact network: maintain the neighbors of the nodes, perform disease progression and relay state updates to the *master*.

Network partitioning

Another key characteristic of this version is the partitioning strategy which aims to divide data and computations. The contact network is identified to be the major data for partitioning. EpiFast [10] discusses few partitioning strategies. We outline them here for quick reference. The first strategy is to consider a list of undirected edges, divide them into groups, and assign them to processors. This makes it difficult to keep track of the status of each individual. Since the neighbors of a given node can reside on any processor (we do not have this information readily), it becomes expensive to synchronize the location information among all processors. Another approach discussed is to partition according to the geographic locations of the individuals. With this approach it becomes extremely difficult to balance load among processors as a uniform distribution of the population is not guaranteed.

EpiFast [10] proposes an efficient partitioning strategy. It considers an adjacency list representation of the contact network with edges grouped by their *tail* node. Intuitively, there are $|V|$ partitions: one corresponding to each node and containing the neighbors of that node or equivalently the set of outgoing edges from that node. These partitions are arbitrarily assigned to processors. In order to account for a balanced load among processors, each processor is assigned approximately the same number of edges. Each node u in $|V|$ has an *owner* processor which maintains all information related to u: its health state and its contacts. Since, the contacts of u may or may not be on the same processor, the co-ordination is done with the help of the master.

Communication

Given that the network is now partitioned among processors, it is obvious that state updates require communication between *slaves*. Let's consider an example where node u is owned by processor p_u. p_u maintains the list of contacts of u. Let's assume, during the computation, that u transmits the disease to one of its contact node v. If v is owned by p_u, then there is no need for communication; p_u can update the health state of v locally. In a case where v is not owned by p_u, it has to relay the transmission to the owner of v (say p_v). This is done via the master. p_u send the transmission to the master and the master relays this to p_v and p_v updates the health state of v after receiving the transmission. EpiFast [10] provides the complete parallel version of the algorithm.

A simple analysis of the parallel algorithm shows that the communication complexity is

quadratic in the number of processors which means that the algorithm is not scalable to large number of processors. Computation complexity is linear in the number of infected nodes as shown in [10]. It is also important to understand the scalability of this implementation. This version exhibits strong scaling and weak scaling. As mentioned in [10], for a fixed problem size, it achieves 3.1-4.1 times speedup when using 6× as many processors; and by doubling the number of processors with doubling the network sizes, the execution times remains constant.

3.4 Challenges with EpiFast

The implementation provided in [10] is based on the *message-passing-interface* (aka *MPI*) parallel programming framework. MPI has long been the standard for parallel applications. Though this implementation has accounted for the scale and performance to an extent, it has few challenges that restrict its application to real-world networks. Using MPI means that the onus is on the developer to

- maintain the identity and status of processors,
- manage partitioning of the network among processors,
- coordinate communication among processors,

These tasks are bound to be tedious when large real-world networks are considered. With the advent of several parallel programming frameworks that aim to make these tasks easier, it is rather evident that a new, more sophisticated implementation is required.

Any newer implementation should focus on abstracting the complexity of the above mentioned tasks. We have identified two parallel programming frameworks: **Charm++** and **Apache Spark** that are potentially beneficial with respect to improving the MPI-based version of EpiFast. The next few chapters provide a comprehensive description of these frameworks and discuss our efforts on using them to improve EpiFast.

Chapter 4

Efforts On Charm++

In this chapter we describe the first parallel programming framework, Charm++, that we considered for developing EpiFast. We will first describe what Charm++ is, features it has to offer for parallel applications, and then describe our efforts in adopting them for developing *EpiFast*.

4.1 Charm++ Architecture

In simple terms, Charm++ is a C++-based parallel programming system. The primary objective of Charm++ was to support an adaptive runtime system that facilitates the development of large-scale, complex parallel applications [33]. Below is a brief list of the mechanisms and/or features it offers for developing parallel applications.

- suites of *load-balancing* strategies,
- *fault-tolerance* through checkpointing,
- *re-usability* through efficient modularization,
- *communication latency*, and,
- *portability* through machine independence.

It is not uncommon among parallel and distributed application developers to expect a programming framework to account for one or more of the above mentioned characteristics. However, Charm++ provides a unique experience in that it also considers the scale of data modern applications have to handle.

To understand how Charm++ works, it is important to be aware of these perspectives: *programming model*, *execution model* and *machine model*.

4.1.1 Programming Model

The key feature of Charm++ is decoupling data from computations. It adopts an *over-decomposition* approach where the application is decomposed into a large number of *data* units and *work* units. Moreover, given that the application is agnostic of locations of these units, developers only have to worry about creating these units and defining a way to make them interact. The absence of references to the physical resources these units reside in makes it possible for the system to change their assignment to processors at runtime as and when needed. Figure 4.1 shows how units are assigned to processors by the runtime system to achieve load balancing. This model also makes developers work with a clear distinction between sequential and parallel units in their applications; and abstract distributed data structures and interfaces from the user.

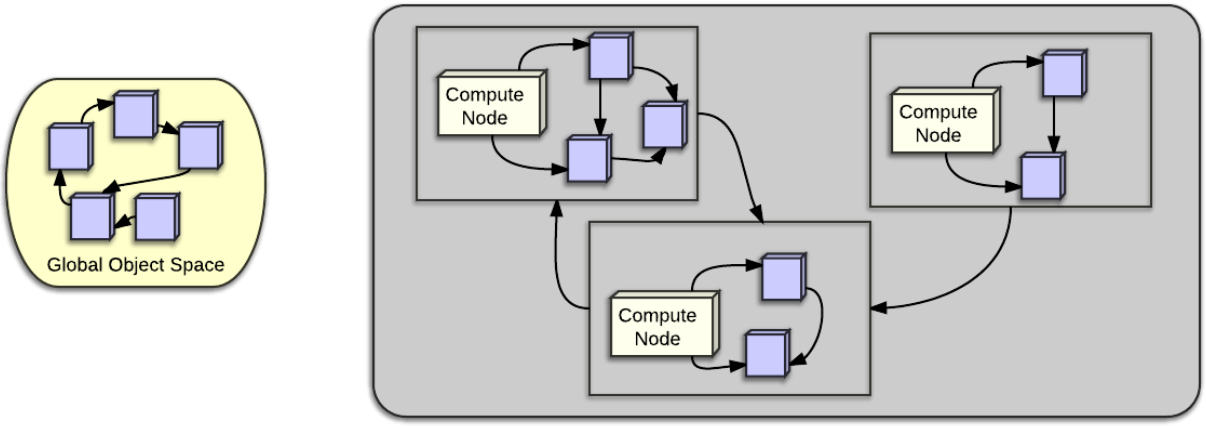

Figure 4.1: Mapping of work units to processors. On the left is the *user* view of the chare collection and on the right is the *system* view with the actual mapping of chares to processors.

This programming model also enables regular (arithmetic operations, manipulating linear data structures, etc.) and irregular (operations of complex data structures such as graphs, trees, etc.) computations through portability, load balancing and latency tolerance, and, managing processes, support for prioritization and handling of dynamic data-structures respectively. This model accounts for an efficient data locality that makes Charm++ applications be able to run on MIMD (multiple instruction, multiple data) machines with or without shared memory [33].

4.1.2 Execution Model

Another perspective of Charm++ is its execution model the basis of which is a *chare*. A chare is a medium-grained process and is the basic unit of parallel computation in a Charm++

application. At a basic level, a chare can be considered a C++ object. Developers identify units of work in their application as chares. The runtime system takes the responsibility of distributing these chares across available processors. Developers also define how these chares interact with each other: basically the communication part of the application and the system takes care of handling this communication through asynchronous method invocations. Similar to MPI, method invocations in Charm++ are essentially sending *messages*, i.e., two chares communicate by exchanging messages which are done through method invocations. This model makes it possible for developers to create and remove chares arbitrarily at runtime without having to worry about mapping them to physical resources. The ability to manage assignment of chares to processors at runtime enables dynamic load balancing in Charm++ applications.

A Charm++ system consists of a pool of chares and queue of messages of these chares. Depending on the availability of physical resources, the Charm++ runtime system (RTS) chooses one or more of these chares in a non-deterministic manner and executes them. The system guarantees that two different methods on the same chare object do not execute simultaneously.

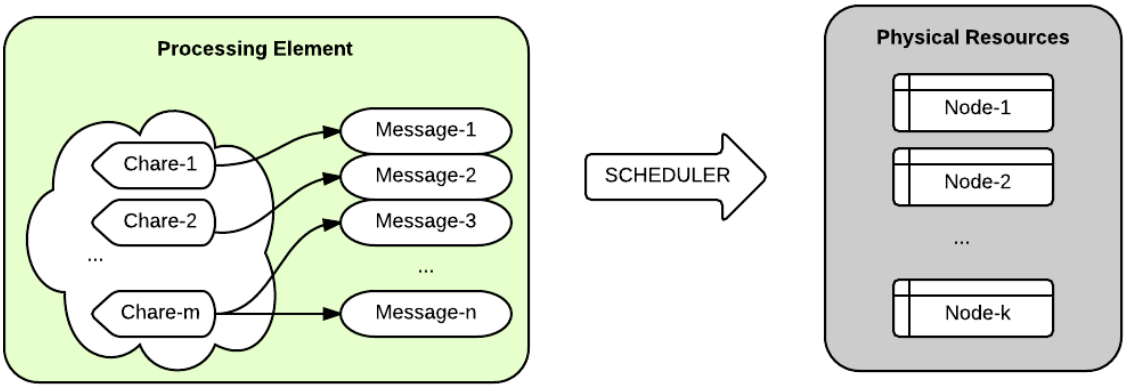

Figure 4.2: Message scheduling in Charm++: Charm scheduler assigning messages to actual physical resources (nodes), on-demand.

Figure 4.2 illustrates the execution model of Charm++. A *processing element* (PE) can be considered as a physical processor core; each chare in the application resides on a single PE. One or more chares may reside on a single PE. Each PE has its own message-pool. These messages may be intended for chares local to the PE or a remote PE. Execution of a message (implicitly a method invocation) may spawn other messages. The job of the scheduler is to select one of these messages, determine the intended recipient of the message and deliver it. *Node-i* in Figure 4.2 denote actual physical machines that available in the programming environment.

4.1.3 Machine Model

From the perspective of physical resources, a chare can be considered a separate processor. Each chare has access to

- private variables,
- *read-only* global variables, and
- proxies to other chares

This is analogous to a C++ object with private members (both variables and methods) and public members (only identifiers to other chares). It is the responsibility of Charm++ RTS to keep frequently communicating objects as close as possible (physically) and optimizing message exchanges between chares on the same physical resource taking advantage of the available shared memory. From an implementation perspective, a separate process exists for each node and all elements within a node share the same address space.

Typically, the number of physical cores and the number of hardware threads available on each core determine the number of PEs for a Charm++ application. The example below helps us understand the components and their relationship.

Consider a parallel environment with 16-core nodes, where each core has two hardware threads and one or multiple logical nodes per physical node. We can develop a Charm++ application with either of the following configurations.

- *32 PEs per (logical) node, and 1 logical node per physical node,*
- *12 PEs per (logical) node, and 1 logical node per physical node,*
- *partition the physical node, and launch it with 4 logical nodes per physical node and 4 PEs per node.*

4.2 Developing Charm-EpiFast

We presented Charm++'s architecture in the previous section. We will now describe our development efforts of *Charm-EpiFast* including design and implementation details.

Programming components

First, we define the components of a Charm++ application to make it easier to understand our approach to designing Charm-EpiFast.

1. **Main chare:** This is a C++ object defined by the developer. This chare is responsible for creating other chares, maintaining read-only global data-structures, maintaining proxies to other chares, etc. Every Charm++ application should have exactly one main chare.

2. **Slave chares:** These are chares that the main chare spawns. These represent work units of the application. The number of such chares, their responsibilities are determined by the application. Assignment of these chares to physical resources is determined by the Charm++ RTS.

3. **Interface modules:** These are Charm++ intrinsics that a developer specifies in the application to make Charm++ RTS recognize the main chare and slave chares, their definitions (as C++ objects), shared variables, methods that are defined for chares to enable communication (entry methods), etc. These are typically in a *.ci* file which is written in a specific interface description language provided by Charm++.

4. **Entry methods:** These form the crux of the communication between chares. These are essentially public methods that chares can invoke to emulate message passing. They are in may ways similar to ordinary C++ methods except that Charm++ optimizes them to make messaging efficient.

5. **Charm++ library:** Charm++ provides an assortment of modules or application programming interfaces to aid developers in several tasks such as object serialization, message aggregation, communication patterns, etc.

Figure 4.3 shows a simplistic view of a Charm++ application. In this view, the application is composed of the *main* chare and *slave* chares. The Charm++ RTS provides the necessary framework for communicating with the underlying environment. This includes an *indexed* collection of chares the assignment (to processors) of which can be determined using the APIs. This processor-aware collection can be considered as the *user-view*. This component also includes optimization libraries that developers can utilize to simplify development or improve performance.

The final component is the parallel environment itself. This is a collection of the actual compute nodes. Each of them have a subset of the chares, a scheduler provided by the Charm++ RTS. The *system-view* is the actual mapping of chares to physical processors which is abstracted from the application by the Charm++ RTS.

4.2.1 Design Overview

In this section, we will describe the design of *Charm-EpiFast*. We consider an object-oriented approach which is typical of Charm++ applications. We identify the objects that compose our application. Then we define how they interact with respect to the actual EpiFast algorithm.

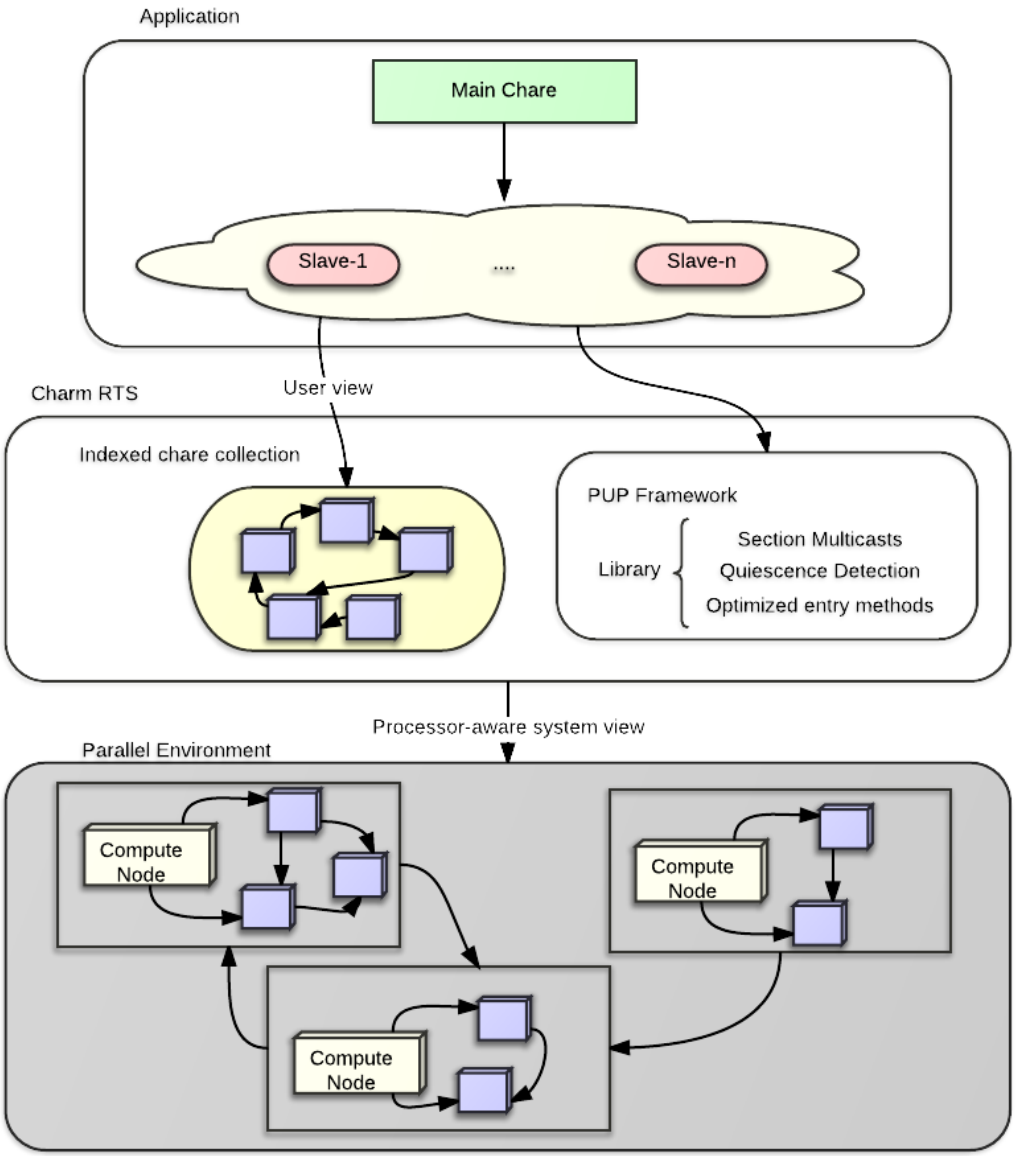

Figure 4.3: Anatomy of a Charm++ application. Shown here are three major components: application, Charm RTS and Parallel environment and two distinct views: user-view and system-view.

Scheme	# *main* chares	# *person* chares	# *contact* chares	# of total chares
1	1	$\|V\|$	$d\,\|V\|$	$d(\|V\|+1)+1$
2	1	$\|V\|$	$\|V\|$	$2\,\|V\|+1$
3	1	$\|V\|$	0	$\|V\|+1$

Table 4.1: Charm-EpiFast design schemes

The first step in the design is to determine the number and functionality of the types of chares. Our design, as any other Charm**++** application, consists of only one ***main*** chare. Next, we determine the number of *slave* chares. We considered the following possible schemes. All the schemes are based on the idea of partitioning the contact network either by nodes (equivalently persons) or edges (equivalently contacts between persons). We denote the first type of chares as ***person*** chares and the second type as ***contact*** chares.

1. **Scheme-1**: In this scheme, we consider,
   - 1 *person* chare for each *node* - maintains the health states of each person,
   - 1 *contact* chare for each *edge* - responsible for a single contact; processes the edge and sends updates to corresponding *person* chare, , in case of disease transmission.
2. **Scheme-2**: In this scheme, we consider,
   - 1 *person* chare for each *node* - maintains the health states of each person,
   - 1 *contact* chare for each *node* - responsible for all contacts associated with a given node; processes the edges and sends updates to corresponding *person* chare, in case of disease transmission.
3. **Scheme-3**: In this scheme, we consider,
   - 1 *person* chare for each *node* - maintains the health states of each person; responsible for all contacts associated with a given node; processes the edges and sends updates to corresponding *person* chare, in case of disease transmission.
   - No *contact* chares.

Given a contact network with $|V|$ nodes and an average degree of d, Table 4.1 provides a count for the number of chares of each type for each of the three schemes. Through experimentation we determined that the lesser the total number of chares, the lesser the overhead of communication and better the performance. It is also argued in [33] that managing larger number of chares as a single component poses a significant overhead in terms of performance. Among our design schemes, it is evident that *Scheme-3* is efficient in the total number of chares in that it uses only $(|V|+1)$ chares. Consequently, we base our implementation on *Scheme-3*.

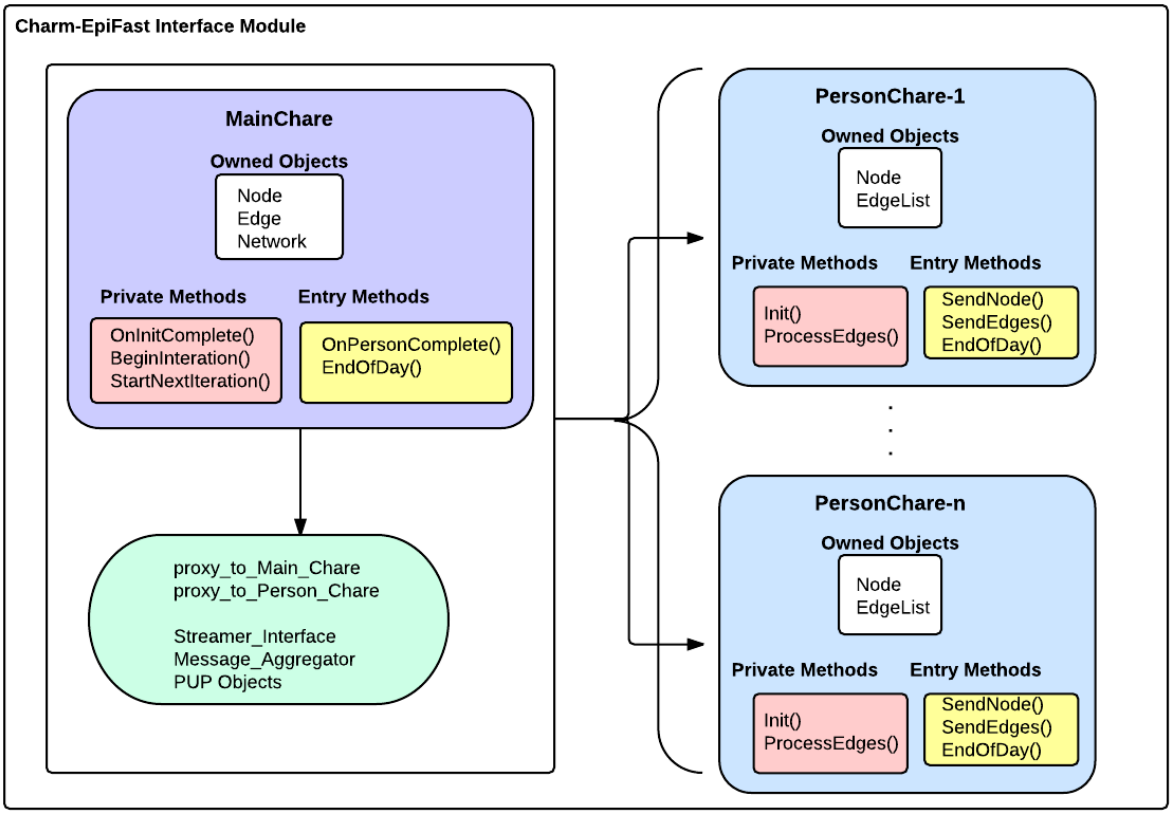

Figure 4.4: Interface and Charm++ objects in Charm-EpiFast. Shows local objects and, private and entry methods of *main* chare on the left and those of *person* chares on the right.

4.2.2 Implementation Overview

The implementation of *Charm-EpiFast* is based on extending *Scheme-3* design by providing a definition of each type of chares and defining *entry* methods that control the flow of communication among chares. Figure 4.4 shows Charm-EpiFast's *interface-module*. The interface-module is a composition of *main* chare and *person* chares, and a definition of methods provided by Charm++ RTS interface for object serialization and optimizations.

The *main* chare and *person* chares are essentially C++ objects with their own private variables including references to other objects, private methods that operate on private variables and public methods including specialized *entry* methods that emulate communication among chares through message passing. However, these two types of chares are inherently different in that they serve different purposes in the application.

The *main* chare is a collection of the following items.

- *Network* object that is a reference to the actual contact network,
- *Node* and *Edge* objects which are references to actual nodes and edges in the contact network,
- entry methods: OnPersonComplete() and EndOfDay() which are invoked by person chares during the simulation

Similarly, a *person* chare is a collection of the following items.

- *Node* object which is a reference to the node this chare is responsible for,
- *EdgeList* object which is a reference to the set of neighbors for this node,
- entry methods: SendNode(), SendEdges() and EndOfDay() which are invoked by the main chare for initialization and broadcast purposes.

Figure 4.5 illustrates the message flow in Charm-EpiFast. Entry methods are bold and italicized and private methods are in normal font. The columns denote the type of the chare the message originates from and the color schemes denote the flow of execution; a change in color marks the beginning of a new execution flow.

- The *main* chare begins an iteration during which it initiates a call to *CallSendData()* which is a wrapper to distribute the node and edge information to *person* chares. The distribution is done via calls to *SendNode()* and *SendEdges()* entry methods.
- The *CallProcessEdges()* is a broadcast function that notifies all *person* chares to begin processing their edges. This spawns a new execution flow in the *person* chares.
- Once a *person* chare completes processing of its edges, it synchronizes with the *main* chare by using the *contribute()* Charm++ routine. The synchronization is done via by invoking *SendUpdatedNodesList()* entry-method on the *main* chare.
- This marks the end of the iteration at the *person* chares. A synchronization is performed by invoking *OnPersonComplete()* at the *main* chare.
- *EndOfDay()* method is private to a *person* chare and is invoked to update any bookkeeping records before synchronization. *EndOfDay()* private method at the *main* chare serves a similar purpose.
- This synchronization spawns a new execution flow at the *main* chare for the next iteration.

Figure 4.6 shows a more detailed view of the flow of control through the different components.

MAIN	PERSON
StartNextIter()	
CallSendData()	
	SendNode()
	SendEdges()
CallProcessEdges()	
	ProcessEdges()
	contribute(Main::SendUpdatedNodesList())
SendUpdatedNodesList()	***EndOfDay()***
	contribute(Main::OnPersonComplete())
OnPersonComplete()	
EndOfDay()	
StartNextIter()	

Figure 4.5: Message flow in Charm-EpiFast. Entry methods are bold and italicized and private methods are in normal font. The columns denote the type of the chare the message originates from and the color schemes denote the flow of execution; a change in color marks the beginning of a new execution flow.

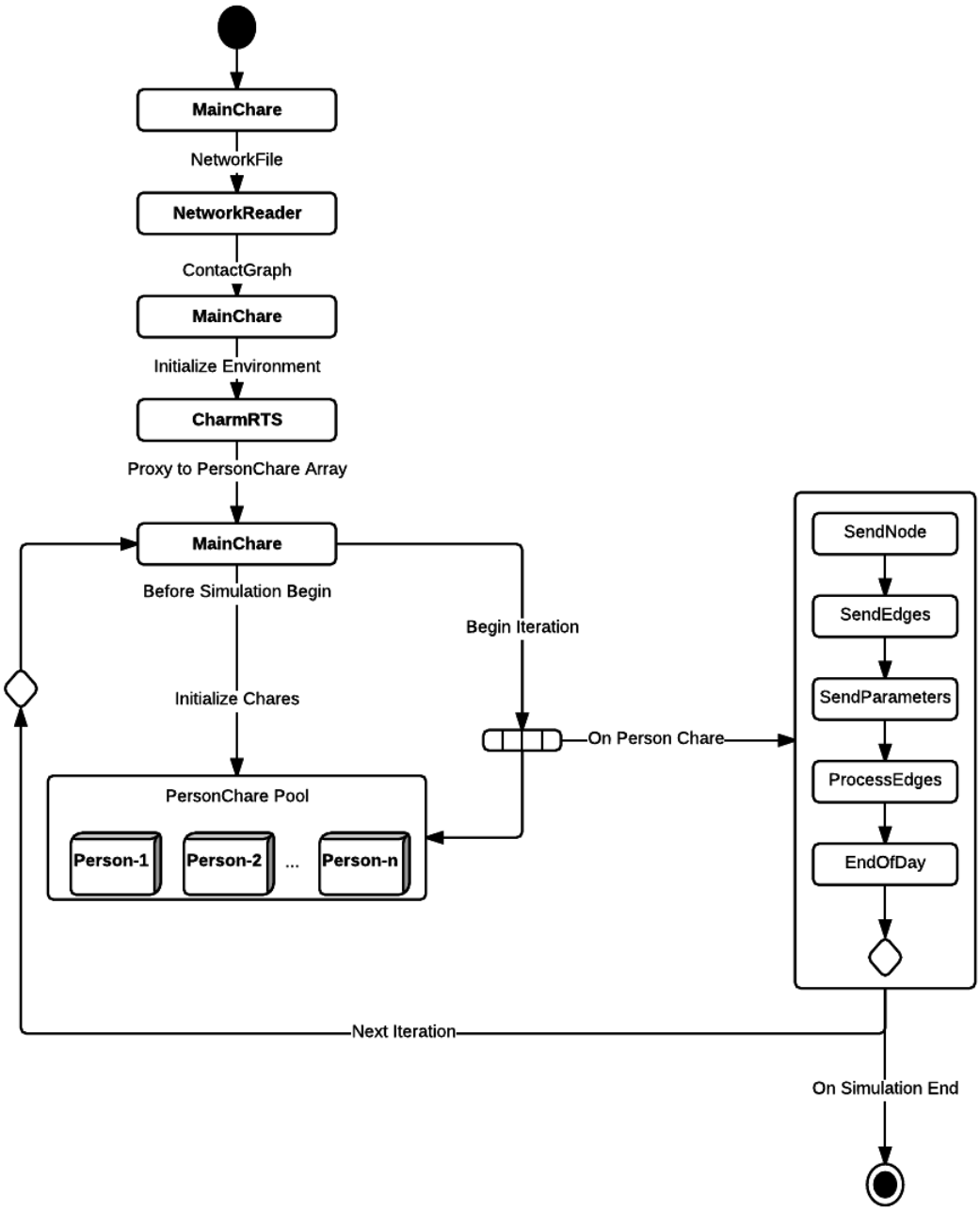

Figure 4.6: Control flow in Charm-EpiFast

4.2.3 Charm++ Optimizations

Our implementation of *Charm-EpiFast* adopts several optimizations that are provided by Charm++. We describe these optimizations in this section. An analysis of our implementation with these optimizations are provided in Chapter 6.

Person chares as Charm++ chare arrays

As discussed in Section 4.2.2, chares in *Charm-EpiFast* are of two types: *main* and *person*. Since there is a one-to-one mapping between *person* chares and individuals in the population, we have $|V|$ chares for a contact network with $|V|$ nodes. An obvious way to represent *person* chares is by using a normal C++ array. This array could be a collection of *person* objects. This makes it easier to maintain the chares and refer them in the application. Moreover, it is sensible to use arrays as all *person* chares are bound to perform the same set of operations.

Charm++ provides an abstraction called *chare arrays* to work with arrays of chare objects. Chare arrays are arbitrarily-sized collection of chares that are distributed across processors. A chare array has a unique identifier (analogous to the array-variable's name) and each element in a chare array has a unique index which can be used to refer the element anywhere in the application. Chare arrays are beneficial in the following ways.

- dynamically create/remove elements: useful when the simulation has to add or remove nodes in the contact network,
- migrate chare elements at runtime to achieve load balancing,
- employ efficient broadcast and/or multicast messaging: useful when one or more subsets of the nodes are required to receive messages, and,
- synchronization and reduction operations: typical of iterative simulations.

We specifically define this array in our implementation as:

$$CProxy\_Person\ personArray = CProxy\_Person::ckNew(numberOfNodes)$$

CProxy_Person is a Charm++ generated type for our *person* chare class and *personArray* is the identifier for our chare array which is initially given a fixed size of *numberOfNodes*.

Using *personArray*, the *main* chare can invoke an entry method on all the chare elements in an optimized way. For example, let us assume all *person* chares expose an entry method called *Initialize()* which is responsible for performing some initialization operations on each chare element. The *main* chare can emulate the broadcast through *personArray.Initialize()*. Charm++ uses an optimization at runtime to perform this broadcast efficiently using a reference to the indexed collection of chares.

PUP framework for serializing objects

An important aspect of method invocations in *Charm-EpiFast* is that they involve exchanging C++ objects. It is critical that objects are stored and moved efficiently, given their size. Charm++ provides the *Pack-Unpack* (PUP) framework to describe contents of an object to Charm++ RTS and use this description while transferring objects. PUP methods are implicitly associated with a virtual function and the actual packing or unpacking of data is done through a memory-to-memory binary copy leading to negligible overheads. To make objects PUP serializable, the classes that define the objects provide a *pup* method that specifies which of their members should follow PUP serialization and how to serialize them. Below is a simple example from our implementation. We have an *Edge* object (as shown in Listing 4.1 with three members: *tailId*, *headId* and *contactDuration*. The PUP method for this object is shown in Listing 4.2 where '|' is the PUP operator overloaded by Charm++. This operator informs Charm RTS to serialize associated variables during message-passing and perform memory-to-memory copy as mentioned above.

Listing 4.1: *Edge* object

```
class Edge {
    long tailId;
    long headId;
    int  contactDuration;
}
```

Listing 4.2: PUP method

```
void pup(PUP::er &p) {
    p | tailId;
    p | headId;
    p | contactDuration;
}
```

Synchronization through quiescence detection

Synchronization is a common task in any parallel application. As discussed in Section 4.2.2, there are several synchronization points. Charm++ provides a feature called *quiescence detection* to achieve synchronization. In Charm++ terms, *quiescence* is defined as a state in which no processor is executing an entry method and no messages are in-flight or awaiting processing. Charm++ provides the *CkStartQD* method which registers a callback with the RTS that is to be invoked when a quiescence is detected. For example, the *main* chare uses the following to begin iteration once all *person* chares are initialized.

Listing 4.3: Quiescence detection after initialization

```
void Main::OnInitComplete(void) {
  CkStartQD(CkCallback(CkIndex_Main::BeginIteration(), mainProxy));
}
```

The *main* chare waits for all the *person* chares to invoke the *OnInitComplete()* entry method. When all *person* chares have made this call, the *main* chare continues with the *BeginIteration()* method since it is the registered callback when quiescence is detected after initial-

ization. Other points of synchronization in *Charm-EpiFast*, such as when an iteration ends, also use quiescence detection.

Multicasting using array-sections

Message-passing is an important task in *Charm-EpiFast*. At each iteration, a *person* chare may send health state updates to one or more *person* chares. Similar to broadcasts, Charm**++** provides a way to efficiently perform multicasts. Multicasts to arbitrary subsets of chare arrays are supported. A subset of a chare array is called a *section*. Elements of a section are referred using a *section proxy*. Multicasts to sections are essentially a broadcast to all members of a section. We illustrate section multicasting using a simple example.

Let u be a node in the contact network and P_u be the chare responsible for u. Also, let $\{v_1, v_2, v_3, v_4\}$ be the neighbors of node u and $P_{v_1}, P_{v_2}, P_{v_3}$ and P_{v_4} be the chares responsible for v_1, v_2, v_3 and v_4 respectively. Further, let us assume that during an iteration, u infects all of its neighbors resulting in a transmission to v_1, v_2, v_3 and v_4, so a multicast is needed to $P_{v_1}, P_{v_2}, P_{v_3}$ and P_{v_4}. We define a chare array section S_u which is the collection of $P_{v_1}, P_{v_2}, P_{v_3}$ and P_{v_4}, i.e., $S_u = \{P_{v_1}, P_{v_2}, P_{v_3}, P_{v_4}\}$. Now, P_u can multicast to its neighbors using a single call as $S_u.SendTransmission()$. Once a P_{v_i} receives this message, it updates the health state of v_i accordingly. Thus, we achieve an optimized multicasting using sections of chare arrays.

Chapter 5

EpiFast using Apache Spark

5.1 Apache Spark

In this section, we outline *Apache Spark*, the programming framework for *Spark-EpiFast*, our second version of EpiFast. We first provide a comprehensive overview of Spark itself and then elaborate our design and implementation of *Spark-EpiFast*.

In simple terms, *Apache Spark* is an engine for large-scale data processing. It boasts of some key features such as *speedup* of 100× over in-memory Hadoop MapReduce [40], *generality*, in that it combines analytics with SQL and streaming, and a *run-everywhere* notion, i.e., either standalone or in the cloud. Spark was originally designed for a particular class of applications that reuse the working data across iterations. Iterative algorithms that work on the same data set across multiple iterations can take advantage of Spark's primitives while still maintaining fault tolerance and being able to scale of large amounts of data.

5.1.1 Revisiting MapReduce Model

The original publication of Spark [40] points out that *MapReduce* pioneered a model that allows data-parallel computations to be efficiently executed on clusters of machines. The programming model behind such a framework usually works on an acyclic data flow graph and is capable of scaling to massive amounts of data. It is important to understand what data parallelism is in order to understand Spark's model. *Data parallelism* is a general parallelization paradigm which focuses on distributing data across multiple physical nodes and perform operations on the data in parallel. MapReduce is a general framework for data parallelism. Spark is based on this paradigm as well except that it focuses on a class of problems that traditional MapReduce fails to handle efficiently.

Typical MapReduce based applications are composed of two stages: *map* and *reduce*. In the

map stage, the data is filtered, sorted and distributed to machines available on the cluster. In the *reduce* stage, an aggregation operation is performed on the data and the result is collected back at the master. Figure 5.1 illustrates a MapReduce job.

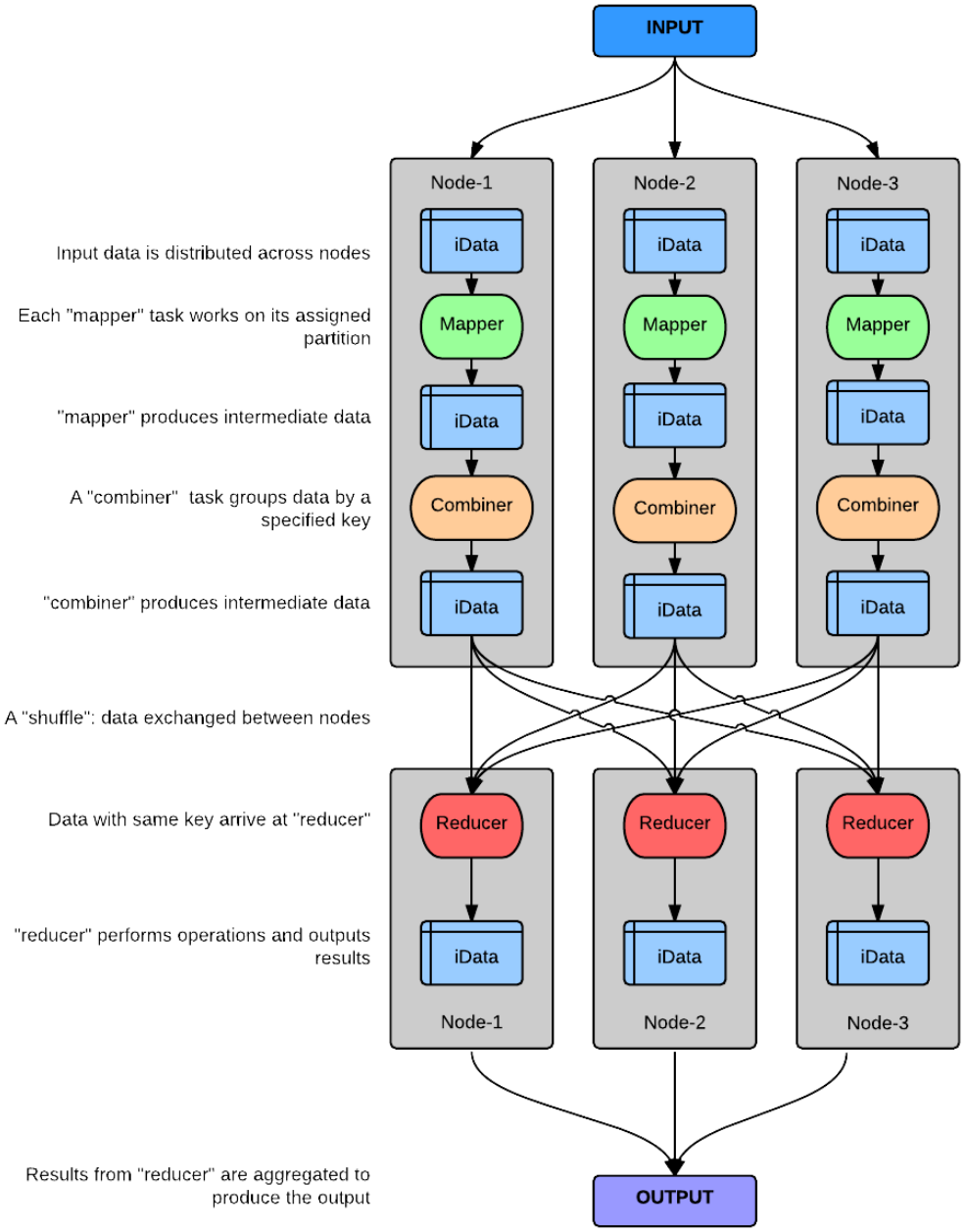

Figure 5.1: Anatomy of a simple MapReduce job. Shows the flow of execution through various stages and the tasks performed in each stage.

The execution flow in this model is as follows.

- input data is split into small subsets and distributed across participating nodes
- *map* tasks work on these subsets of data
- an optional *combiner* task groups data based on a user-defined key
- grouped data are passed as inputs to *reduce* tasks
- *reduce* tasks operate on the data and produce intermediate results
- results from *reduce* tasks are aggregated to produce the overall result

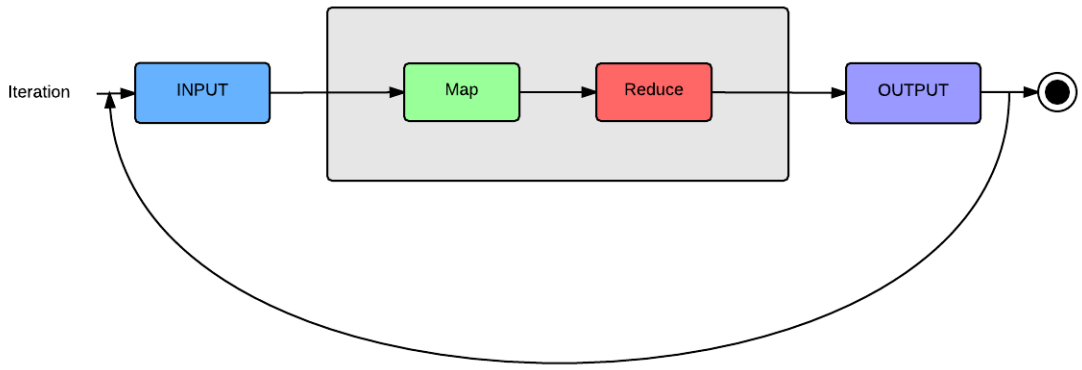

Figure 5.2: An iterative MapReduce program with multiple iterations each involving *map* and *reduce* tasks.

Algorithms that adopt a *MapReduce* model involve a sequence of *map* and *reduce* tasks as shown in Figure 5.2. As seen earlier, iterative algorithms are typical of generating sequences of operations that converge to an optimal solution at some point. Such algorithms are also usually characteristic of starting with a data set and retaining them across iterations. This means that data distribution is done at the beginning of every iteration as iterations are considered to be independent of each other in this model. The repeated data distribution incurs a major overhead leading to degraded performance in large-scale applications. Spark attempts to address this particular issue common to large-scale iterative algorithms. In the following section we describe Spark's approach to this problem.

5.1.2 Resilient Distributed Dataset (RDD)

Spark provides a major abstraction called *resilient distributed dataset* (RDD) to enable efficient data reuse. Per [52], RDDs are defined as fault-tolerant, parallel data structures that enable users to explicitly persist results from an iteration in memory to allow efficient re-usability, control data partitioning to optimize placement across nodes, and manipulate them using a set of operators. This essentially means that RDDs can be operated upon in parallel once they are partitioned across nodes in the cluster. A formal description of an RDD as *a read-only, partitioned collection of records* is provided in [52]. We reiterate on some of the important characteristics of RDDs mentioned in [52].

- creation of RDDs is done through deterministic operations called *transformations* such as *map*, *filter*, etc., from
  - data in stable storage such as HDFS, or,
  - collections provided by the programming language
- capable of recomputing its state with information on how it was derived,
- *persisted* in memory to allow re-usability, and,
- *partitioned* across nodes based on a user-defined key in each record.

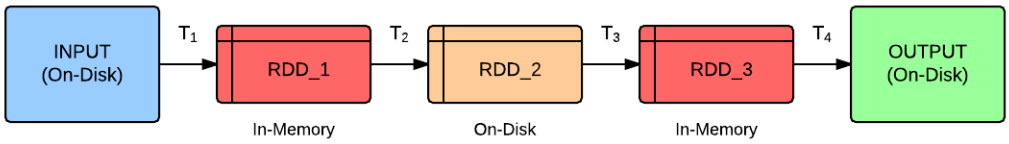

Figure 5.3: A sequence of transformations involving RDDs with RDDs after transformation being written to disk or persisted in-memory.

An illustration of this is shown in Figure 5.3. Here, RDD_1 is constructed from data on the disk through transformation T_1. RDD_1 is persisted in memory and reused efficiently. T_2 produces RDD_2 which is written to disk. This is further operated on by T_3 to produce RDD_3 which is again persisted in memory for T_4 which finally writes the output to disk. Considering several iterations of this sequence, it makes more sense to leverage *persistence* and eliminate expensive disk read/writes to achieve better performance.

5.1.3 Programming with RDDs

As discussed in the previous section, RDDs are essentially containers of data. Spark provides language-integrated APIs that users can leverage to work with RDDs. APIs are available for *Scala*, *Java* and *Python* programming languages. These APIs consider RDDs as objects of the supported language and emulate RDD transformations via methods on the objects. It is important to understand the following components in order to program with RDDs.

Transformations

In Spark, RDDs are defined through *transformations* on data from stable storage. Spark's API provides several transformations common to practical uses of containers. *map(func)*: returns a new RDD by applying function *func* to each element in the source RDD and *filter(func)*: returns a new RDD selecting only those elements of the source that return *true* for function *func*, are two of the basic transformations provided by the APIs. A comprehensive list of transformations available on RDDs is provided in [53].

Transformations have the following characteristics.

- RDDs are *immutable*; so any transformation on an existing RDD creates a new RDD,
- transformations are *lazy*: results are not computed right away. Spark remembers what transformation has to be applied on which RDD. The results are computed only when *actions* are performed on the RDD.

Actions

Actions are operations that perform computation on an RDD. Actions return a value to the application or export data to an external storage. Actions are also triggers for the actual computations and determine when RDDs change. *reduce(func)*: aggregates elements using function *func* and *collect()*: returns a collection of elements of the RDD to the driver program; these are examples of actions provided by the APIs. A comprehensive list of actions available on RDDs in provided in [54].

Persistence

Persistence is one of the key features of Spark that makes in-memory computation possible and efficient. Through persistence, Spark preserves elements around the cluster or disk or replicate across machines. Applying *persist()* on an RDD informs Spark to retain partitions on nodes thereby allowing future actions to reuse data. This makes interactions

faster in iterative algorithms. Spark provides different level of storage for persisting RDDs. *MEMORY_ONLY* and *MEMORY_ONLY_SER* are two common persistence levels supported by the APIs. A list of storage levels is provided in [55].

Programmers have the liberty to choose storage levels for RDDs in their application. Some of the basic characteristics of the data determine their storage levels. For example, RDDs that fit completely in memory can use *MEMORY_ONLY* or *MEMORY_ONLY_SER* to make objects more space-efficient.

5.2 GraphX API For Graph Processing

In the previous section, we discussed Apache Spark and its programming in general usage terms. In this section, we focus on a specific use case and discuss how Spark can help in developing applications related to that use case.

Large scale data processing applications involving graphs are typically classified as either: *data-parallel* or *graph-parallel* applications. Systems such as *Giraph* [56] and *GraphLab* [57] are graph-parallel systems dedicated to graph processing. Graph-parallel systems impose a restricted domain of computations on graphs to achieve performance improvements. On the other hand are data-parallel applications that view data as distributed across machines and try to express as many operations on data as possible. Consider a simple graph for example for distributed processing. Graph-parallel systems consider partitioning the graph based on a partitioning strategy (such as source vertex of edges, sub-graphs with distinct properties, etc.) and distribute these partitions across machines while data parallel systems could simply distribute edges across machines ignoring the structure of the graph. While both distribution strategies are useful, they can be restrictive in terms of modifying or expressing specific operations on components of the graph. Thus far, these two types of systems have mostly been isolated from each other serving different purposes and data movement between these two views has significantly affected application performance. Spark aims to unify these two systems through a more simplified view of data and a programming interface to develop applications to reduce data movement and duplication.

Property Graph

GraphX [58, 59] is Spark's API for graph-processing. It is simply an extension of the Spark RDD based on a different view of data. The view of data inherent to GraphX is the *Resilient Distributed Property Graph*. This *property graph* is a *directed multigraph* with user-defined attributes associated with vertices and edges. Note that in a multigraph, there can be multiple edges between a pair of vertices. A property graph has *vertex* (VD) and *edge* (ED) object types associated with it and each object type can have an *attribute* object attached

to it. Property graphs provide a simple yet efficient representation of objects that compose a graph.

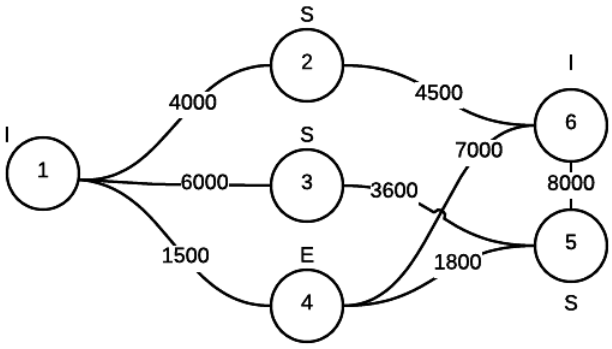

Figure 5.4: A simple property graph with 6 vertices and 7 edges depicting a contact network. Vertices and edges are labeled with health states and contact durations respectively.

Figure 5.4 shows a simple graph with 6 nodes and 7 edges. Each node is attributed with its state (like health states in EpiFast) and each edge is attributed with an integer (like the duration of contact in EpiFast). Note that the graph shown here is un-directed, however. Figure 5.5 shows the vertex and edge tables corresponding to the graph in Figure 5.4. *Vertex* and *Edge* tables are internal representations of the property graph in Spark.

Vertex Table

Id	Property (V)
1	Infectious
2	Susceptible
3	Susceptible
4	Exposed
5	Susceptible
6	Infectious

Edge Table

SrcId	DstId	Property (E)
1	2	4000
1	3	6000
1	4	1500
2	6	4500
3	5	3600
4	5	1800
4	6	7000

Figure 5.5: Vertex and edge tables for the property graph in Figure 5.4.

Property graphs, like RDDs, are,

- *distributed*: split across machines using vertex indices,
- *immutable*: changes to an existing property graph produces a new graph, and,
- *fault-tolerant*: can be reconstructed from existing information on machines.

As with RDDs, property graphs have a collection of operators that modify vertex and edge tables through user-defined functions and produce new graphs. These operators have optimized implementation interfaces exposed through the APIs. A list of available graph operators is available in [60]. Graph operators in Spark extend to *property-operators*, *structural-operators* and *join-operators* that manipulate the property of graphs, structure of graphs, and merge multiple graphs together respectively.

Graphs in Spark, like RDDs, are not by default persisted in memory. Typical GraphX applications cache graphs in order to enable efficient reuse. Furthermore, iterative computations that produce intermediate results, when persisted may lead to cache overflow. So, it is important to determine whether or not to cache graphs in applications especially when graphs are large.

Vertex and Edge RDDs

GraphX provides views of the vertices and edges associated with the graph through *VertexRDD* and *EdgeRDD*.

- *VertexRDD[VD]* represents a set of vertices each with an associated attribute *VD*. This set of vertices is stored in a reusable hash-map data structure.
- *EdgeRDD[ED]* represents the collection of edges partitioned across machines each with an associated attribute *ED*.

These objects extend the basic RDDs and provide additional functionality specific to vertices and edges such as modifying attributes, adding or removing vertices and/or edges, etc. Below is a Scala snippet of a class representing a graph in GraphX.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

5.3 Developing Spark-EpiFast

In Section 5.1, we discussed about Spark's abstraction and programming, in general, with RDDs. Now, we elaborate our efforts on *Spark-EpiFast* which is our implementation of *EpiFast* on *Spark*. We will discuss our approach with respect to the design and implementation of Spark-EpiFast. Chapter 6 provides an analysis of our implementation.

Let's first understand the different components of the Spark environment from a developer's perspective. This is important to understand the complexities behind a Spark application. Figure 5.6 shows the different components. There are 4 nodes (or physical machines) in the setting. Every Spark application has a *SparkMaster*. In our example, it is loaded in Node-1. *SparkMaster* is responsible for managing the application. This is analogous to the *master* in a *master-slave* model. *SparkMaster* is commonly referred to as the *driver* program in the developer community.

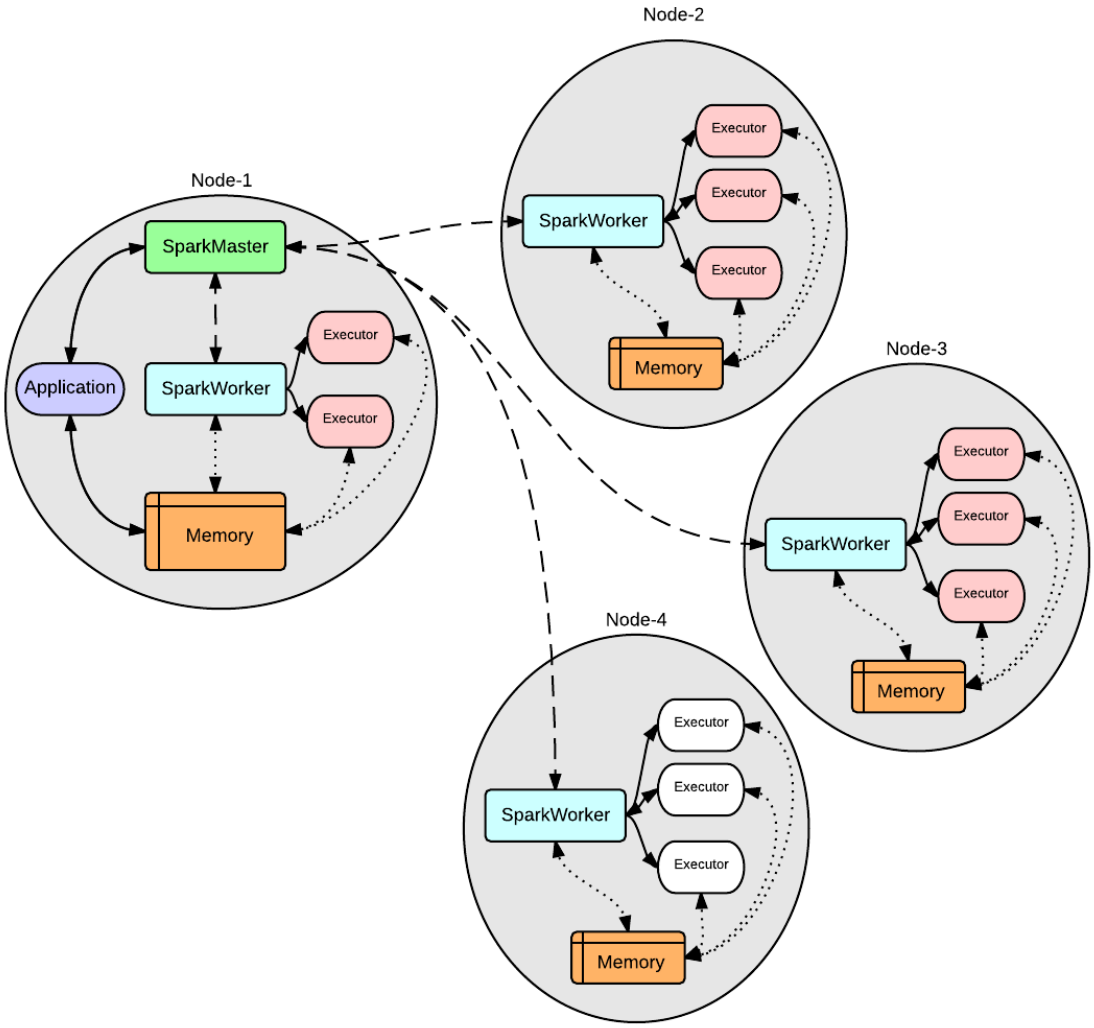

Figure 5.6: Spark integration - shows 4 physical nodes each with various Spark components, and their interactions with the application residing on the master node.

Spark's environment creates a process called *Worker* in each of the nodes in the cluster, referred to as *SparkWorker* in the figure. *SparkWorker* processes can be considered as managers of the respective nodes. Spark also spawns processes called *Executor* for each *Worker*. The number of executor processes depends on the application load and is determined at runtime by Spark's environment. *Executors* are the actual worker processes that perform computations on data.

The application is itself represented as a component in Figure 5.6. However, internally, an application is just a *Java Archive (JAR)* file representing the collection of classes in the

application. Spark distributes the application JAR to all nodes in the cluster. Also, any distribution of data triggered by the application is handled by *SparkMaster* which identifies the number of partitions (if not specified by the application) and distributes the partitions to individual nodes. The distribution is fault-tolerant in the sense that Spark will be able to reconstruct the data in the partition from its lineage information.

The other nodes in our example, *Node-2*, *Node-3* and *Node-4* are similar to the master node except that they do not have *SparkMaster*. They all participate in computations and any communications (if needed) and are coordinated by respective *SparkWorkers*.

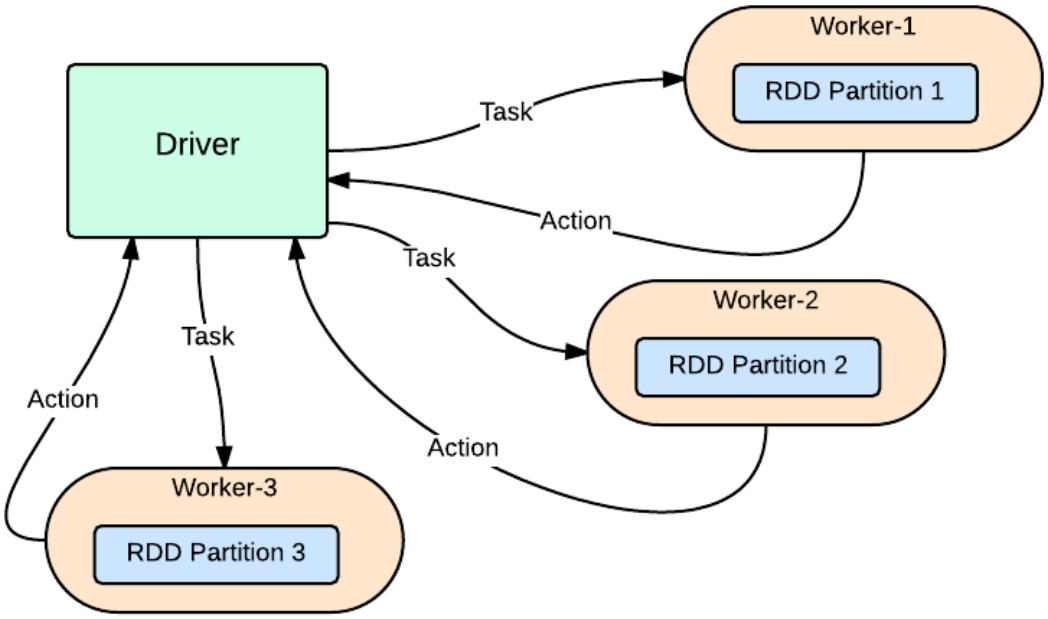

Figure 5.7: Tasks and actions on RDDs in Spark with the driver program delegating tasks to worker processes and results being collected at the driver through actions.

Figure 5.7 shows a group of *worker* processes each with a partition of an RDD. A *task* in Spark triggers the distribution. This is equivalent to *transformations* discussed in previous sections. Similarly, an *action* triggers the computation on the partitioned data and a rendezvous at the *driver*.

Design and Implementation

In this section, we discuss the design and implementation details of *Spark-EpiFast*.

Spark-EpiFast Components

We have identified the following as major components of the *Spark-EpiFast*.

- **APPLICATION**: encompasses various sub-components such as contact graph, I/O handlers, containers for RDD, etc.
- **SPARK_ENGINE**: constitutes the application *driver*, Spark's configurators, task handlers, etc.
- **ENVIRONMENT**: represents the collection of physical nodes, application JAR, scripts to instantiate master and worker processes, etc.

Figure 5.8 shows a view of the components and their interactions. We provide only a minimalistic view of the interactions to help understand the application easier. Table 5.1 summarizes the sub-components.

- **EpiFastConfig**: responsible for loading EpiFast's configuration and setting appropriate parameters in the application metadata such as simulation parameters (input network, transmissibility, incubation and infectious periods of individuals, etc.)
- **FileParser**: reads the input file and loads the contact graph into the application; supports EFIG5 and EFIG6Bb formats currently.
- **ContactGraph**: a representation of the input social contact network; loaded by *FileParser*.
- **GraphContainer**: language-specific containers for vertices (individuals) and edges (contacts) of the contact network.
- **(VertexRDD, EdgeRDD)**: a pair of RDDs containing the graph's vertices and edges; the application works on this pair of RDDs during the simulation.
- **SparkConfigurator**: encompasses *SparkContext* and settings for running the application on Spark.
- **ApplicationDriver** and **ApplicationExecutor**: represent the *driver* and *executor* processes; *ApplicationDriver* is responsible for identifying tasks, partitioning, distribution and collection of data. *ApplicationExecutor* takes care of the actual computations on RDDs.
- **StorageHandler**: key component that handles memory interactions such as caching and persisting RDDs.
- **JobHandler**: responsible for handling *transformations* and *actions* on RDDs.

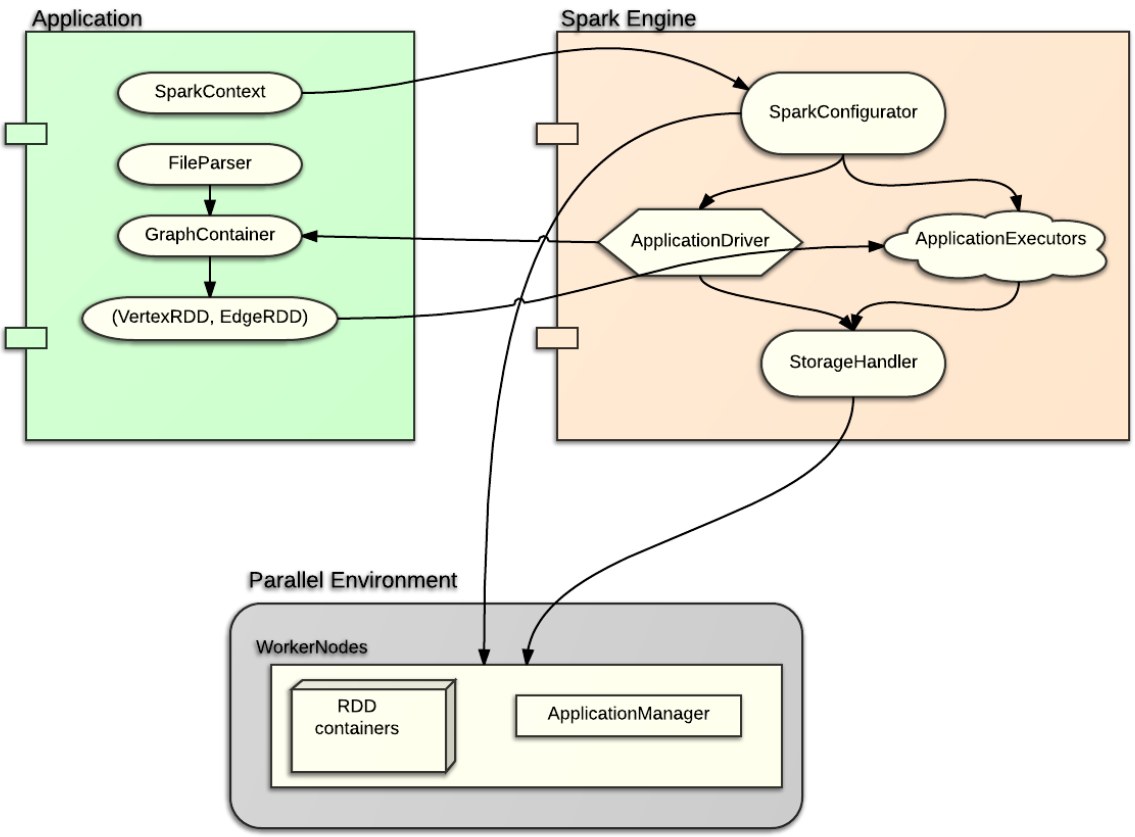

Figure 5.8: A simplistic view of interactions between Spark-EpiFast components

- **SparkApplicationManager**: used by *SPARK_ENGINE* internally to instantiate worker and executor processes. This is analogous to *SparkMaster* discussed earlier.
- **ApplicationJAR**: collection of classes and libraries in the application that is sent to each node by the driver program at runtime.

Control Flow in *Spark-EpiFast*

Control flow in *Spark-EpiFast* preserves *low coupling* and *high cohesion*: the major components are less interdependent and majority of the interactions are among sub-components within the same parent component. We believe this supports reasonable levels of readability and maintainability.

Figure 5.9 illustrates the flow of control during application execution. We outline this below.

Component	Parent Component
EpiFastConfig	APPLICATION
ContactGraph	APPLICATION
FileParser	APPLICATION
GraphContainer	APPLICATION
(VertexRDD, EdgeRDD)	APPLICATION
SparkConfigurator	SPARK_ENGINE
ApplicationDriver	SPARK_ENGINE
ApplicationExecutor	SPARK_ENGINE
StorageHandler	SPARK_ENGINE
JobHandler	SPARK_ENGINE
SparkApplicationManager	ENVIRONMENT
ApplicationJAR	ENVIRONMENT

Table 5.1: Summary of Spark-EpiFast components

- *FileParser* parses the input file of the contact network and creates immutable *GraphContainer* for vertices and edges internally represented as RDDs.
- *SparkConfigurator* loads configuration parameters through *SparkContext* for Spark and initializes the environment including *ApplicationDriver*.
- The control at the *ApplicationDriver* branches into two different flows depending on what stage the application currently is.
- Before the simulation begins,
  - *TaskContext* determines the number of executors required for a particular worker
  - *StorageHandler* determines the number of partitions and distributes the RDDs to the *ApplicationExecutors*. It also applies *persistence* to the partitioned data if specified by the application
- During an iteration,
  - State updates due to *within-host* disease progression are applied at the *ApplicationDriver*. Note that this step also constitutes a transformation on the vertex RDD composed of a *map* operation.
  - a **flatMap()** transformation is applied on the RDDs; this causes *betweenHostTransmission* function to be applied on the partitioned data residing at the executors.
  - once the executors have applied the *flatMap()* transformation function, they return results back to *JobHandler* and rendezvous at the *ApplicationDriver* when the **collect()** action is triggered.

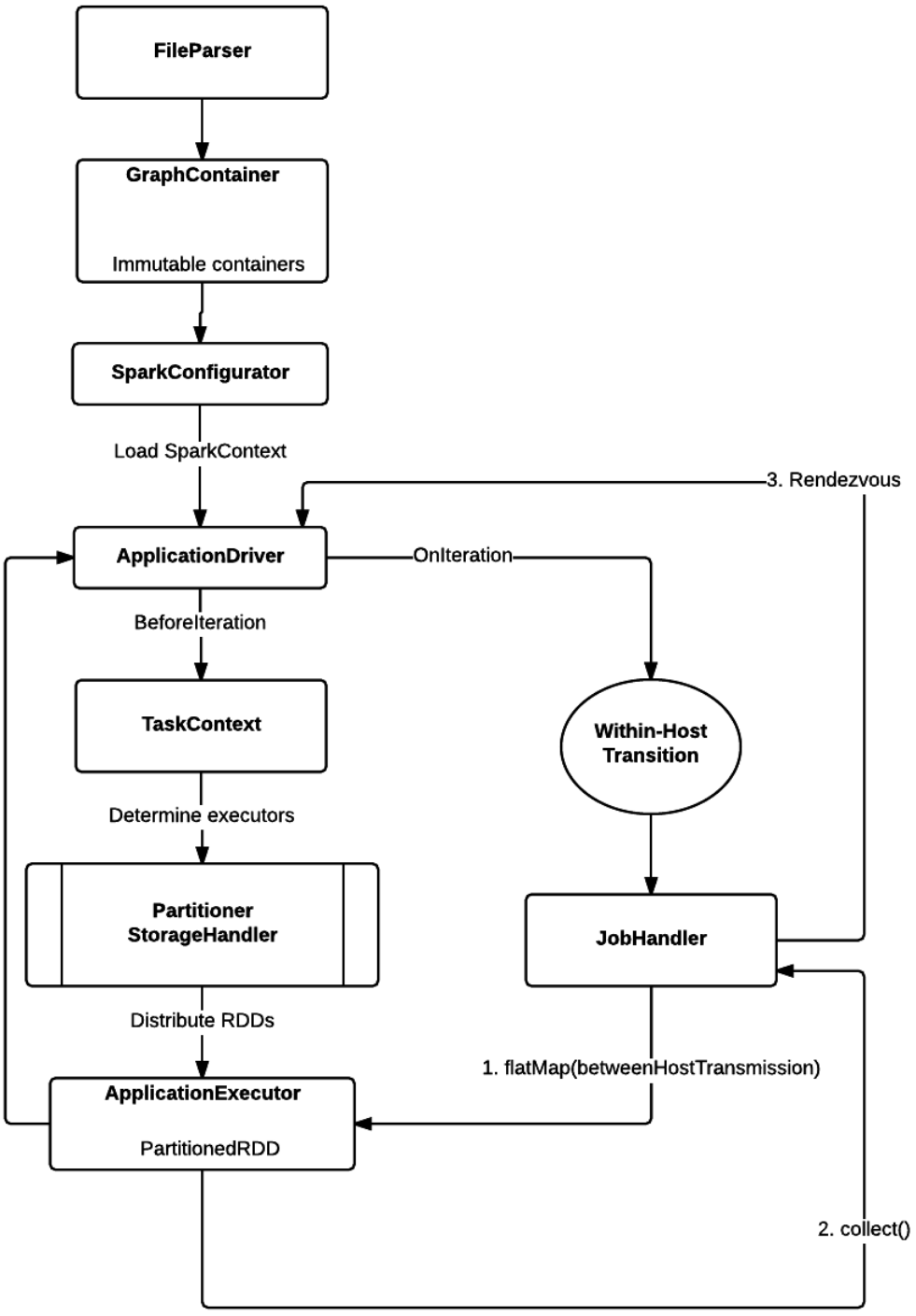

Figure 5.9: Control flow in Spark-EpiFast

Spark-EpiFast algorithm

In this section, we answer an important question. *"How do we interpret EpiFast transmissions in terms of Spark's abstractions?"*. Our solution is to ***map transition functions in EpiFast to transformations in Spark***.

Formally, let's consider a transmission in EpiFast to be of the form $y = f(x)$ where

- x: nodes/edges
- y: nodes/edges with modified attributes
- f: transition/transmission function

We formulate an equivalent representation of $y = f(x)$ as $y = x.t(f)$ where

- x: RDD (vertex or edge)
- y: modified RDD
- f: user-defined function defining transition/transmission in EpiFast
- t: Spark's transformation function

Now, we consider the two EpiFast functions separately and relate them to the above formulation.

- *Within-host transition:*

  Original representation,

  Listing 5.1: Within-host transition in EpiFast

```
modifiedVertices = withinHost(vertices)
```

  Equivalent representation can be given as,

  Listing 5.2: Within-host transition in Spark

```
modifiedVertexRDD = vertexRDD.map(withinHost)
```

- *Between-host transmission:*

  Original representation,

  Listing 5.3: Between-host transmission in EpiFast

```
edgesWithTransmission = betweenHost(edges)
```

Equivalent representation can be given as,

Listing 5.4: Between-host transmission in Spark

```
edgeRDD.groupBy(sourceVertex).persist()
intermediateRDD = edgeRDD.flatMap(betweenHost)
edgesWithTransmission = intermediateRDD.collect()
edgesWithTransmission.map(updateStates)
```

After identifying the representations, we provide the actual algorithm for *Spark-EpiFast* in Algorithm 1.

Algorithm 1: Spark-EpiFast Algorithm

Input: $(G, \triangle\tau, \Lambda)$ where G is the contact network, $\triangle\tau$ is the incubation and infectious period durations, and Λ is the set of initial conditions.

Output: (τ, Π) where τ is the health state transition times of all people, and Π is the infectors who transmit the disease to the infected people.

load *Spark* configurations;
initialize *SparkContext*;
load *EpiFast* configurations;
prepare *vertex* and *edge* RDDs from *ContactGraph* container using G;
group EdgeRDD by source vertex and *persist* in-memory;

for $t = 0$ *to* T **do**
    update health-states of $v \in$ *vertexRDD* based on $\triangle\tau$;
    apply *betweenHostTransmission* transformation on *EdgeRDD*;
    collectedRDD $\leftarrow$ *collected* list of vertices requiring state-updates;
    update health-states of $v \in$ *collectedRDD*;
end

output (τ, Π)

Complexity Analysis

"Embarrassingly parallel" problems are those that are trivially separable into a number of parallel tasks and that there exists no communication between those tasks. It is apparent that EpiFast is not "embarrassingly parallel" since transmissions that form the core of the algorithm, are required to be communicated. Our version of Spark-EpiFast follows a MapReduce-like paradigm. In previous sections we showed how to interpret transmissions in EpiFast to transformations in Spark. We analyze the cost of each transformation here. Algorithm 1 involves three basic transformations, *groupBy()*, *flatMap()* and *collect()*.

The *groupBy()* transformation in Spark uses an implicit hashing mechanism. For instance, a VertexRDD construction induces BTREE-based indexing on the vertex ID. A grouping on

the EdgeRDD causes the edges to be grouped into $|V|$ groups. Note that this causes a *shuffle* on the already partitioned data. The cost of *shuffle* can be determined using *replication rate,* r as discussed in [61]. r is defined as the average number of key-value pairs produced by mapper tasks. In our case, $r = |V|$, same as the number of groups. However, the number of edges $|E|$ also impacts the overall complexity as *groupBy()* traverses the list of edges once to determine the group for each edge.

The *flatMap()* transformation represents the bulk of the computation in Spark-EpiFast. This deals with the computations involved in the *between-host* transmission function. This is a sequential computation bounded by the average degree d of a node in the contact network.

Finally, the *collect()* action which triggers a traversal over an iterator of the infected nodes. Let's assume that the number of infected nodes on any given simulation day is k. A collection of all these infected nodes can be transformed to an RDD and applied health updates in parallel using k groups which can be done in parallel using k reducers.

The comparison of theoretical cost of communication and computation between MPI-EpiFast and Spark-EpiFast is not straightforward. We believe experimental analysis will provide more insight into the performance of Spark-EpiFast compared to MPI-EpiFast and Charm-EpiFast. We dedicate Chapter 6 for this purpose.

Operations on *EdgeRDD*

The important functions in *Spark-EpiFast* are the *transformation* and *action* operations involved during *between-host* disease propagation. We will describe the working of these operations behind the scenes with an example. Consider the graph in Figure 5.10. This graph has 6 vertices and 9 edges. Further, note that this graph is un-directed which makes the corresponding *EdgeRDD* to have a record for both *(srcId, dstId)* and *(dstId, srcId)* pairs. Table 5.2 shows the list of edges associated with each vertex in our example graph.

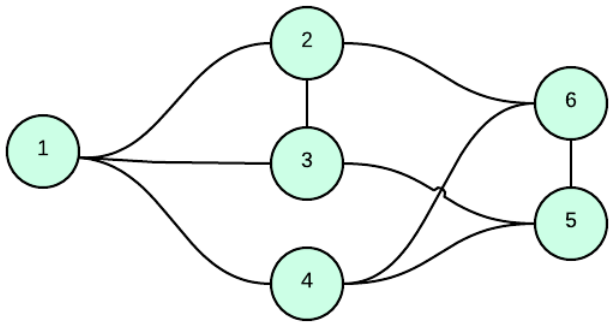

Figure 5.10: An example graph for *groupBy* operator

Source Vertex	Edge List
1	{(1, 2), (1, 3), (1, 4)}
2	{(2, 1), (2, 3), (2, 6)}
3	{(3, 1), (3, 2), (3, 5)}
4	{(4, 1), (4, 5), (4, 6)}
5	{(5, 3), (5, 4), (5, 6)}
6	{(6, 2), (6, 4), (6, 5)}

Table 5.2: *EdgeRDD* for graph in Figure 5.10

Figure 5.11 illustrates the *groupBy* and *collect* operations on the *EdgeRDD*.

- We start with the RDD *edgeRDD*.
- *groupBy(groupBySourceVertex)* is a *transformation* applied on *edgeRDD*. This groups the edges with source vertex as the key. Here, *sourceVertexId* is a user-defined function that specifies the grouping criteria. *groupBy()* returns *groupedEdgeRDD* container. Below is our definition of *groupBySourceVertex* in Scala.

  Listing 5.5: *groupBySourceVertex* function

  ```
  val groupBySourceVertex = (edge: Edge[(Int, Int, Int)]) => {
    edge.srcId
  }
  ```

  This function specifies that the group for an edge *edge* is determined by its source vertex, *edge.srcId*.

- Next, we apply the between-host transmission function *betweenHostTransmission* on *groupedEdgeRDD* resulting in partitions of the grouped edges. These are arbitrarily assigned to available executors for further computations. In our example, *Executor-1* gets the edges for source vertices $\{1, 3\}$, *Executor-2* gets the edges for source vertices $\{2, 6\}$, and, *Executor-3* gets the edges for source vertices $\{4, 5\}$.
- Within the *betweenHostTransmission* function, each executor computes and samples the transmission probability distribution associated with each of the edges and determines the set of vertices that require state updates. They maintain this set in *localResultRDD* which is an immutable collection of vertices.
- Finally, *collect()* is applied on the local results on each executor which are combined and communicated to the *driver* as *collectedRDD*.

It is important to note that *groupBy()* triggers the distribution of edges each time it is invoked. So it not advisable to use it in every iteration. We eliminate this concern using

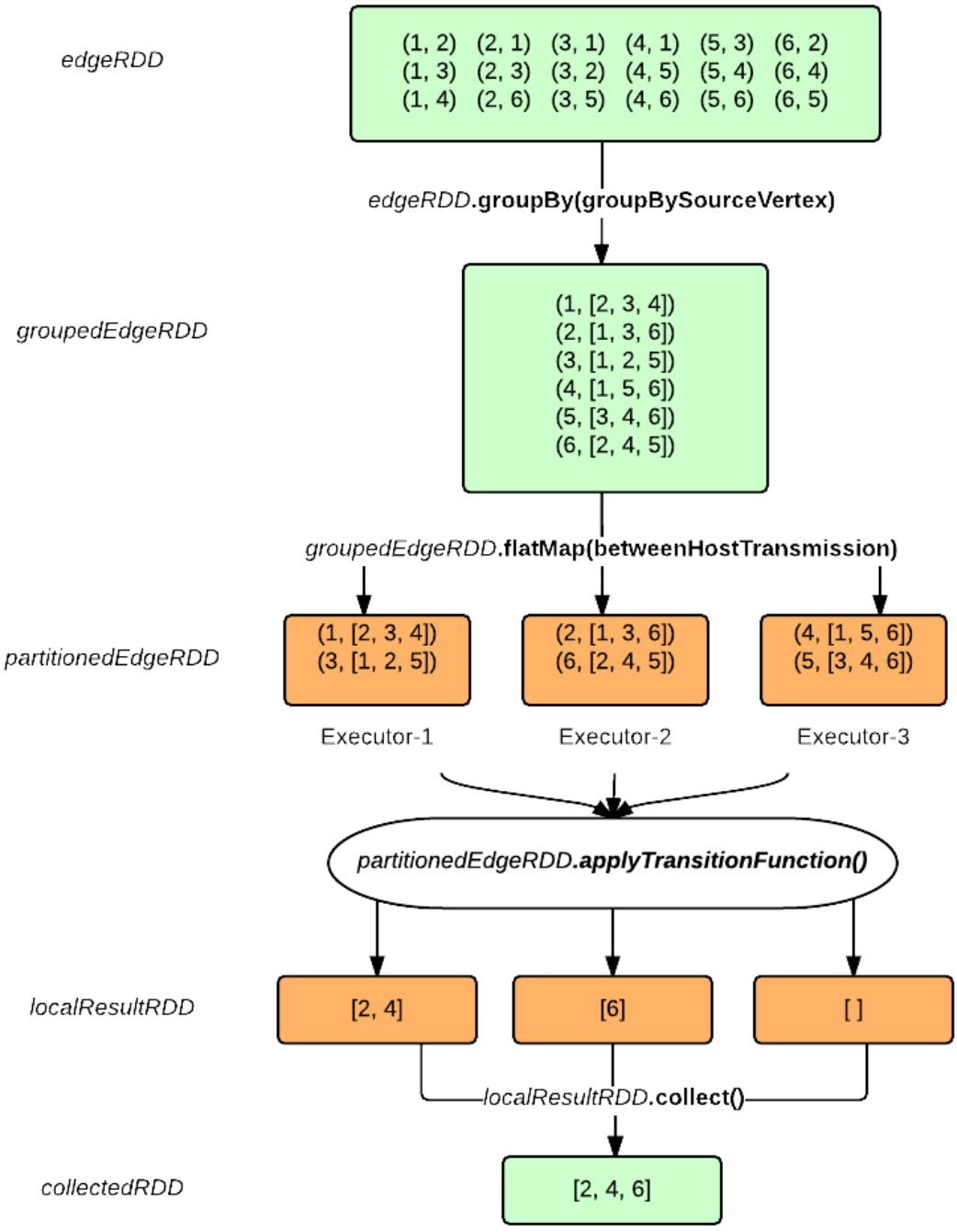

Figure 5.11: *Between-Host* transmission. Shown here is the sequence of steps involved in the between-host transmission process. Starting with *edgeRDD*, groupBy() produces *groupedEdgeRDD*, followed by *localResultRDD* after applying flatMap(). Finally the collect() operation results in *collectedRDD*.

Spark's *persistence*. Our approach is to determine the groups, distribute them to executors and persist them on target nodes so that groups are reused across iterations. Below is a Scala snippet that performs this.

Listing 5.6: *groupBy()* and *persist()* on *edgeRDD*

```
var groupedEdgeRDD = edgeRDD
                     .groupBy[VertexId](groupBySourceVertex)
                     .persist(StorageLevel.MEMORY_ONLY)
```

Chapter 6 provides an analysis of using different persistence levels for edge RDDs.

Modeling Interventions in Spark-EpiFast

We now discuss our design for handling *interventions* in Spark-EpiFast. Interventions in EpiFast are composed of the following components.

- *Sub-population*: represents subsets of individuals in the contact network; usually defined by demographic and/or geographic criteria, e.g. school age people in a particular county.
- *Threshold*: representation of condition(s) that must be satisfied for an intervention action to be applied; possibly associated with one or more sub-populations.
- *Total*: represents total available supply of pharmaceutical interventions such as vaccines, antivirals, etc.
- *Action*: represents the action to be taken as a result of an intervention; typically defines how the intervention is applied - add/remove contacts, increase/decrease contact durations, scale infectivity (or vulnerability) of participating nodes, etc.
- *Intervention*: represents a composition of all the above components

Figure 5.12 illustrates a conceptualization of flow of control with interventions. This is similar to Figure 5.9 except that here we have an *InterventionHandler* which performs some pre-processing (as shown in Figure 5.13) before the start of the simulation and handles interventions during the simulation (as shown in Figure 5.14).

The objective of the pre-processing step is to parse the intervention file and create a representation of the interventions which can be used later during the simulation. First, the *InterventionFileParser* parses the intervention file and identifies the various components and registers them with *InterventionRegistry*. Another important responsibility of the *InterventionRegistry* is to create RDDs for each sub-population component. The purpose of this is to directly manipulate these RDDs during the simulation without having to reconstruct them

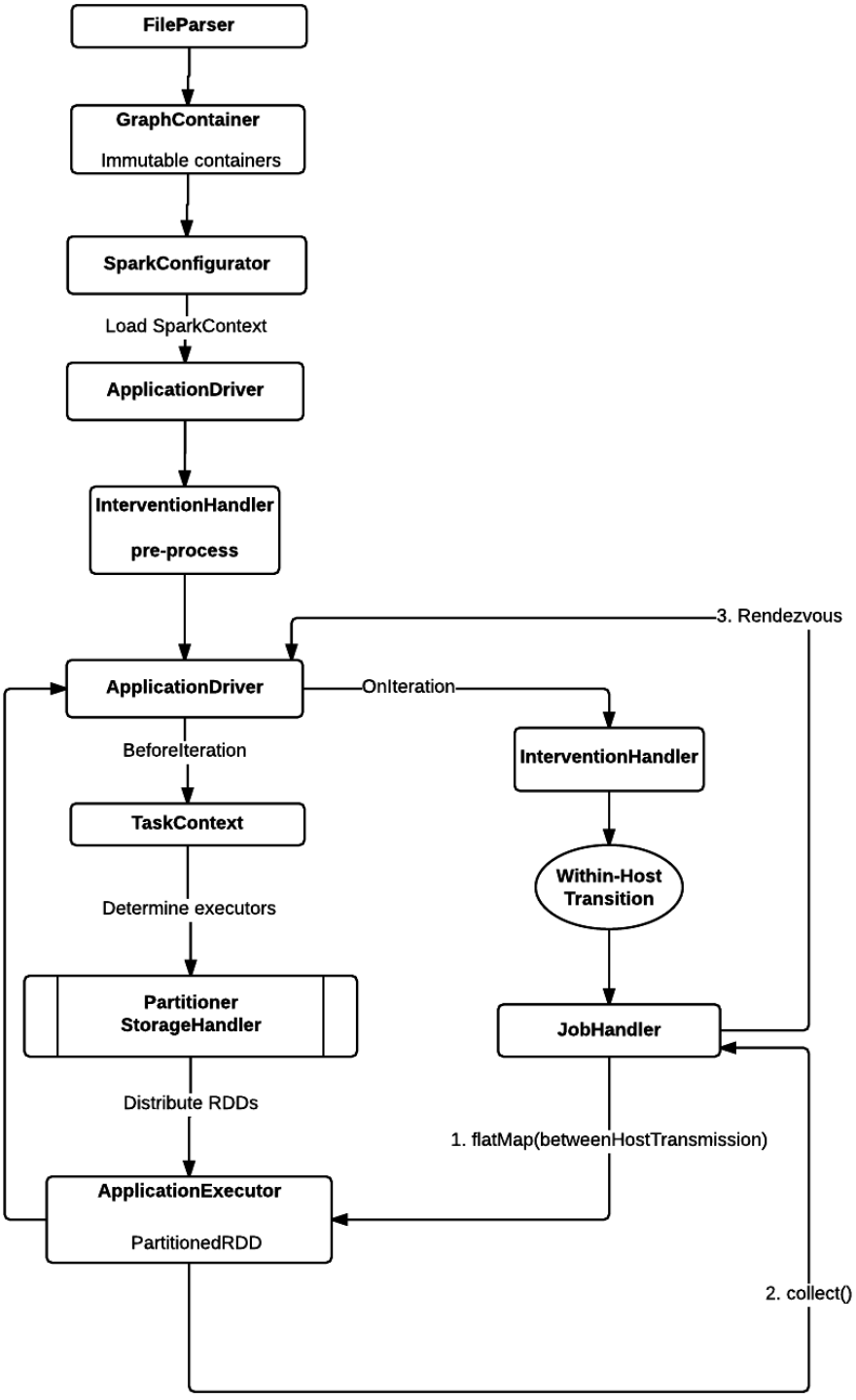

Figure 5.12: Control flow for interventions in Spark-EpiFast

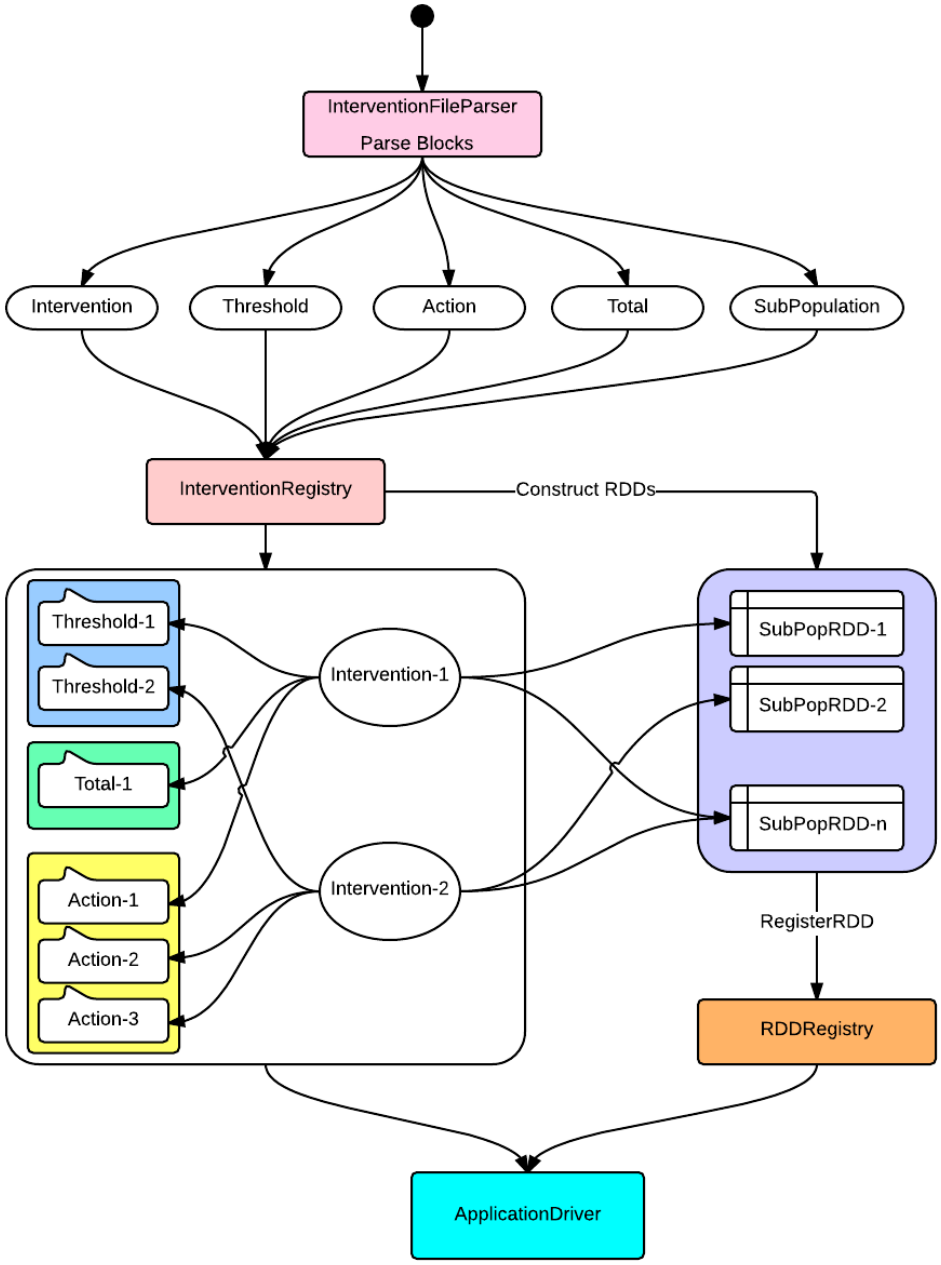

Figure 5.13: InterventionHandler - Pre-processing. Shows how the interventions file is parsed for identifying components and constructing the *registry* for later reference by InterventionHandler.

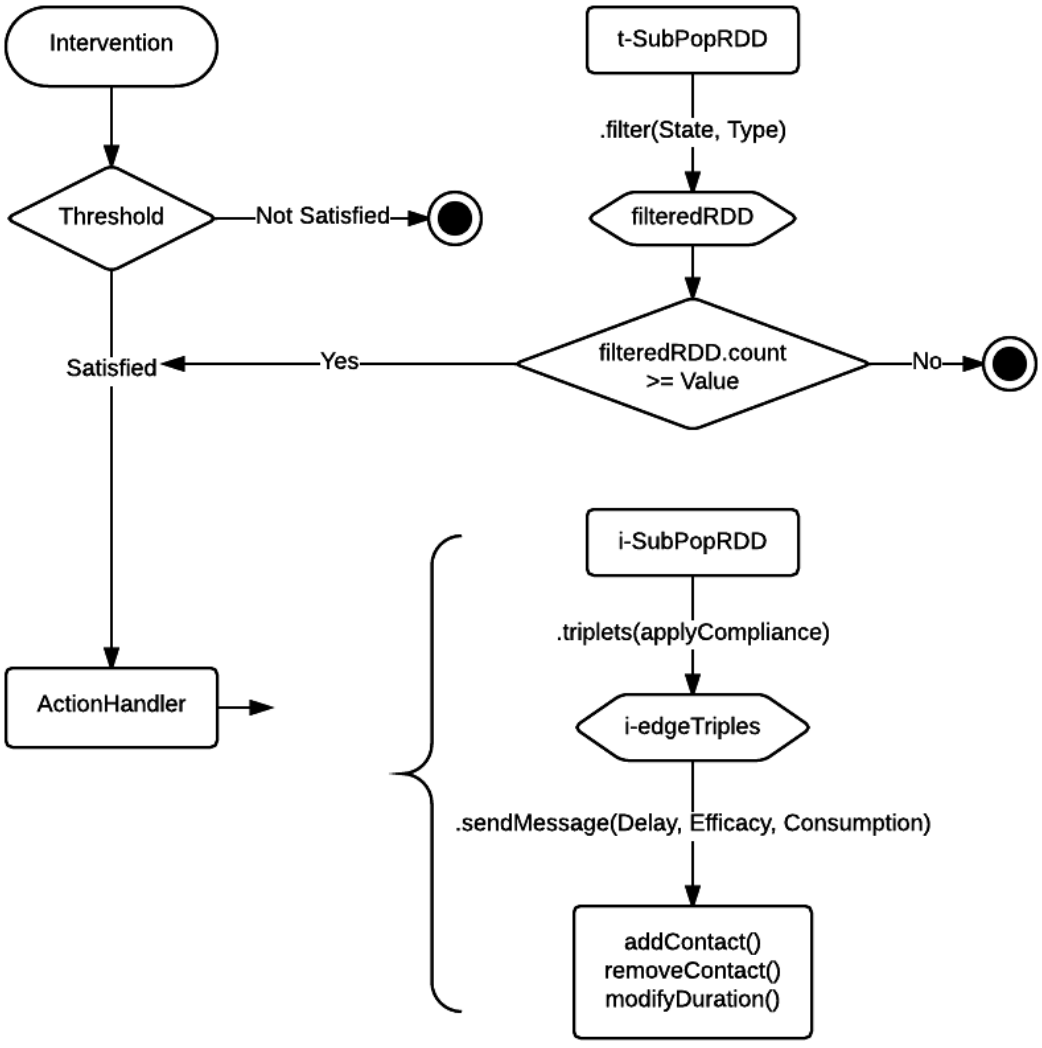

Figure 5.14: InterventionHandler - Iteration. Shows how threshold condition is checked for on sub-population RDDs and consequently how actions are handled.

every iteration. It also prepares data containers for each component and creates a mapping between them for quick references during the simulation.

During an iteration, the *InterventionHandler* first checks whether the *threshold* condition is satisfied. This composes the following operations on the threshold sub-population RDD *t-SubPopRDD*.

- apply a transformation to filter individuals based on the *state* and *type* of the threshold, indicated as *.filter(State, Type)* in the figure.
- on the filtered RDD, i.e., the subset of individuals satisfying the threshold condition, we apply another action *count()* to determine the number of such individuals.
- if the count is smaller than the threshold value, *Value*, we continue with the simulation. Otherwise, we perform the corresponding action through the *ActionHandler*.

The *ActionHandler* also works on the sub-population RDDs denoted as *i-SubPopRDD* in the figure. The following operations are applied on these RDDs.

- complying individuals are first identified by setting their *compliance* according to the *ConditionCompliance* parameter.
- we obtain triplets representing edges, *i-edgeTriples*. These are essentially RDDs with the compliance attribute set.
- the corresponding action is taken by applying a Spark *action* called *sendMessage()* which applies the action on each edge in the RDD based on the parameters such as *Delay*, *Efficacy* and *Consumption*. The messages communicate the corresponding actions such as modifying edge attributes, adding/removing edges, etc.

We provide a brief discussion on the generalizability of Spark-EpiFast. It is apparent that our setup makes it easier to provide different definitions of *withinHost* and *betweenHost* functions. For example, a user may provide his own definition of the these functions without having to change the entire code. However, a scenario where a user wants to provide a completely different transmission model (such as a combination of several transmission functions) may not be trivial. With respect to interventions also, our design is generic to all types of interventions that EpiFast supports. We considered all types of interventions that EpiFast supports and came up with a design that accommodates all of them in a generic way with focus on re-usability of modules. However, we suspect that incorporating any intervention that original EpiFast does not support is non-trivial and that this will consume significant development effort. However, in Chapter 7, we discuss few other features of Apache Spark that can benefit when implementing interventions. The details of those features are out of scope of this thesis. But we believe that further research of such features can help us identify ways to quickly and easily incorporate all types of interventions to our current design.

Chapter 6

Experimental Analysis

In this chapter, we present a detailed analysis of *Charm-EpiFast* and *Spark-EpiFast*. We describe the experiments we performed and analyze their results to understand the behavior of the implementations.

Table 6.1 summarizes our experimental setup.

- **Processing environment:** NDSSL ShadowFax cluster of 60 nodes. Each node has a 12-Core Westmere-EP X5670 2.93 GHz dual processor, 48 GB memory, and 1 TB hard drive.
- **Charm++:** Version-6.6.0 for MPI-based 64-bit Linux architecture with *Production* and *tracing* mode enabled for less runtime overhead and in-built optimizations.
- **Apache Spark:** Manual installation of standalone Spark version-1.2.1 (built for non-Hadoop infrastructure) on a subset of nodes in the cluster to emulate distributed environment. APIs (such as for reading network files, managing application, etc.) are minimal.
- **Parameters (metrics) for comparison:** *Average execution times* (averaged over 25 runs) and *relative parallel speedup* to analyze scalability.
- **Contact networks:** Population range of 77k to 4M individuals and 3M to 228M contacts as summarized in Table 6.3.

Table 6.1: Summary of experimental setup: processing environment, setup of programming frameworks: Charm**++** and Apache Spark, and, metrics used in our analysis.

Table 6.2 provides a summary of experiments we performed and key observations.

- **Charm-EpiFast:** Primary objective is to identify optimizations (and combination of optimizations) among **quiescence detection** and **section multicasting** and other strategies.

  The following strategies are considered.

  - Unoptimized version (a plain version with none of the optimizations incorporated).
  - Version with quiescence detection.
  - Quiescence detection and reduction at master chare.
  - Quiescence detection and slaves messaging directly without intervening master.
  - Quiescence detection and section multicasting for many-to-many messaging.

  Version with **quiescence detection** and **section multicasting** provides better performance among all strategies - Figure 6.5.

- **Spark-EpiFast:** The following is a list of parameters considered for evaluation.

  - **RDD partitions:** number of partitions of RDDs of input contact network - Figure 6.10.
  - **RDD persistence:** storage levels for persisting RDDs (along with serialization) to preserve grouping of edges across iterations - Figure 6.11.
  - **Transformations:** two major transformations on RDDs - *flatMap()* and *collect()* - Figure 6.15.

  A single groupBy() with *MEMORY_ONLY_SER* provides better performance - Figure 6.16.

- Scalability (as relative parallel speedup) and efficiency of three versions (MPI-EpiFast, Charm-EpiFast and Spark-EpiFast) are compared in Figure 6.20 and Figure 6.21 respectively. I/O overhead is compared in Figure 6.22.

  Spark-EpiFast is more scalable and efficient than other two versions.

- **Spark-EpiFast** improves developer productivity as illustrated in Figure 6.23.

Table 6.2: Summary of experiments considered for analysis for Charm-EpiFast, Spark-EpiFast and comparison of these with original MPI-based EpiFast.

Table 6.3 provides details of the networks we considered for our analysis. The *Memory (MB)* column in this table denotes the disk storage space required for the input networks.

Network	**# nodes**	**# edges**	**Memory (MB)**
Montgomery County (MC)	77,569	3,936,774	46
NRV County (NRV)	152,653	8,093,740	94
Washington D.C (DC)	497,034	19,984,180	350
New Hampshire (NH)	1,229,661	72,076,468	700
Miami	2,167,915	110,375,174	1300
Boston	4,184,498	228,851,784	2600

Table 6.3: Networks considered for experimental analysis

We provide the details of our analysis in the following sections - Section 6.1 for Charm-EpiFast and Section 6.2 for Spark-EpiFast and summarize the observations in Table 6.6.

6.1 Analysis of *Charm-EpiFast*

We first analyze the performance of *Charm-EpiFast* incrementally with respect to optimization techniques we discussed in Section 4.2.3 in Chapter 4. We used a contact network with 4.18 million individuals and 228.85 million contacts as the base graph for our analysis. We show the execution times of *Charm-EpiFast* against the number of cores we used. The execution times here represent the average execution times (in seconds) per simulation day.

Figure 6.1 shows execution times of plain Charm-EpiFast, one without any optimizations. This is a basic implementation which only uses chare arrays to represent person chares.

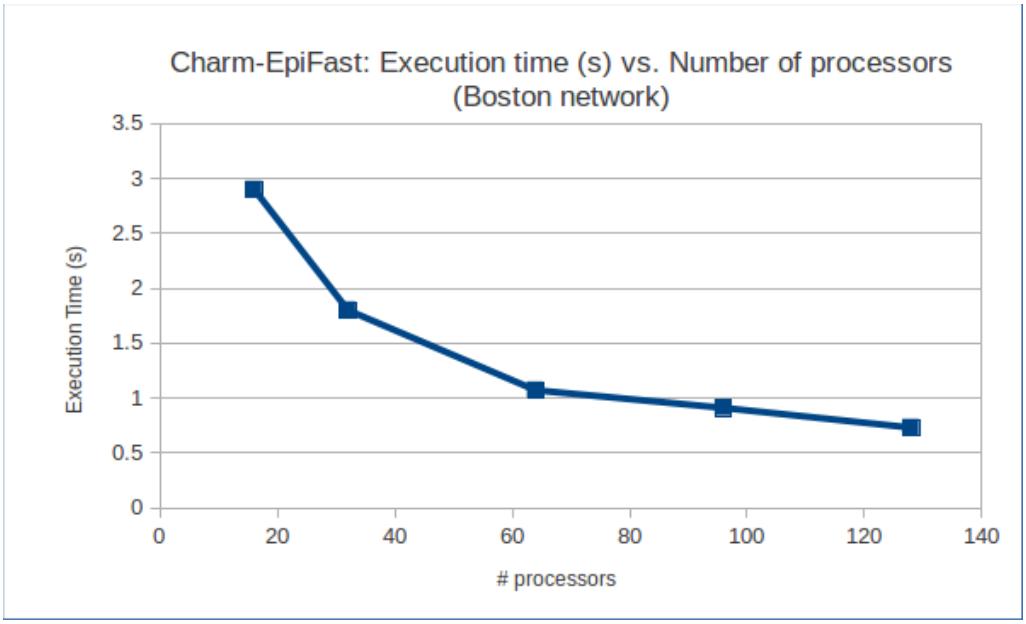

Figure 6.1: A simple analysis of plain Charm-EpiFast (without any optimizations discussed in Chapter 4). Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates against number of processors for Boston network. Further analysis uses this plot as a base for comparisons.

We now provide analysis of the Charm-EpiFast incrementally with each optimization technique we considered and compare them against versions from which they were built and also against other optimization techniques. An outline of Charm-EpiFast versions that the plots signify is given below.

- Unoptimized Charm-EpiFast (Figure 6.1): a plain version with none of the optimizations incorporated.
- Quiescence detection (QD) (Figure 6.2): version with quiescence detection included.
- QD and reduction at master (Figure 6.3): version with quiescence detection and global reduction at master.
- QD and messaging slaves (Figure 6.4): version with quiescence detection and slaves messaging directly without intervention from master.
- QD and section multicasting (Figure 6.5): version with quiescence detection and section multicasting for many-to-many messaging.

Figure 6.2 compares the plain version with *quiescence detection* (refer Section 4.2.3). The version with quiescence detection consistently outperforms the plain version.

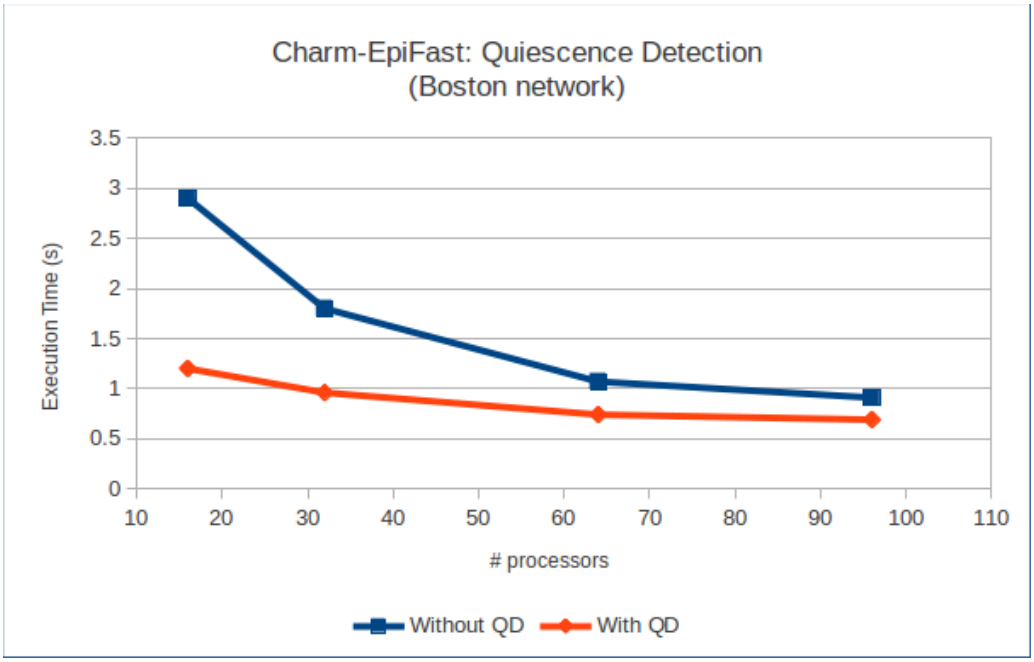

Figure 6.2: Comparison of (a) Unoptimized version and (b) version with quiescence detection. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates against number of processors for Boston network. Quiescence detection provides performance improvements as indicated by the red curve.

Figure 6.3 considers another version with synchronization at the master where each slave sends its updates to the master which then forwards the transmission to actual recipients. Intuitively, this approach consumes a significant amount of messaging (because of the relay via the main chare) resulting in a degraded performance compared to other previous versions.

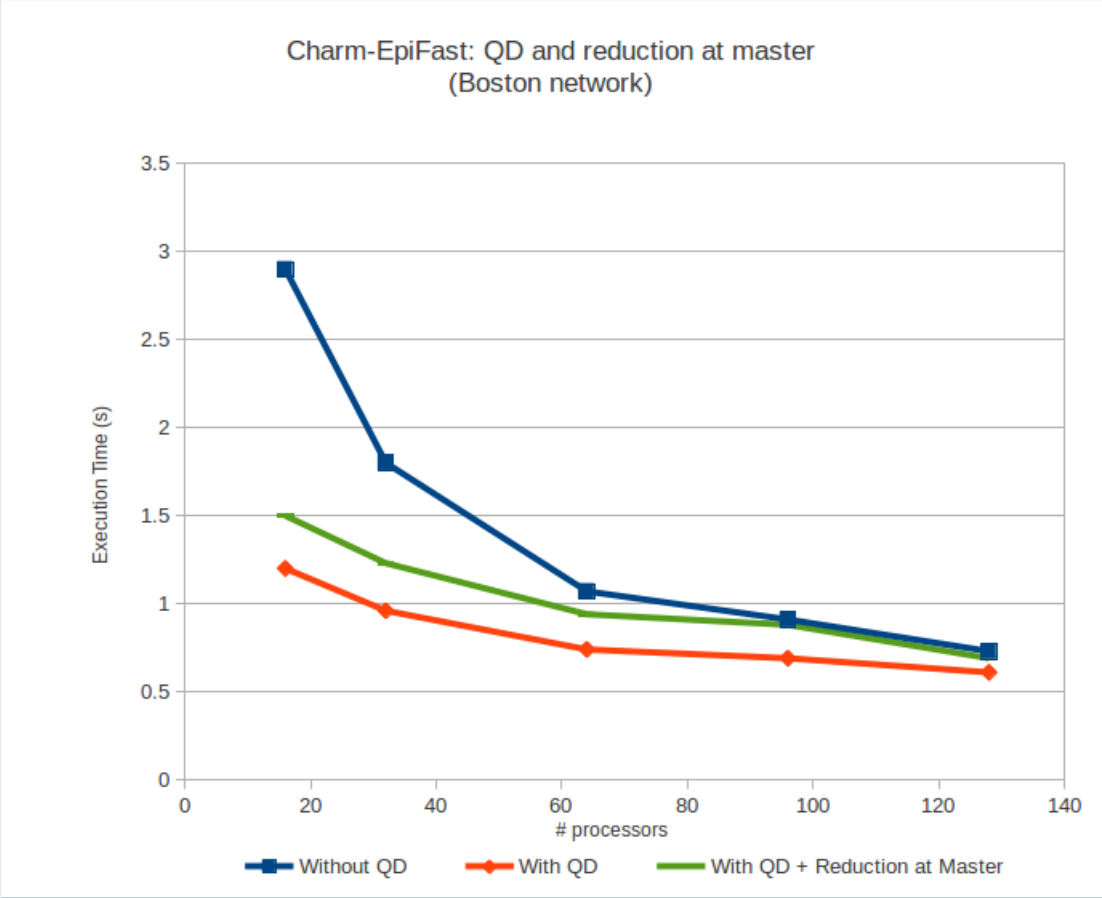

Figure 6.3: Comparison of (a) Unoptimized version, (b) version with quiescence detection and (c) version with quiescence detection and reduction at master chare. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates against number of processors for Boston network. Master chare becomes a bottleneck due to global reduction at the end of every day.

Figure 6.4 shows another version where slaves directly communicate, i.e., they send the transmissions directly to the receiving person chares with the main chare playing no role. This seems to perform much better than the case when master intervenes.

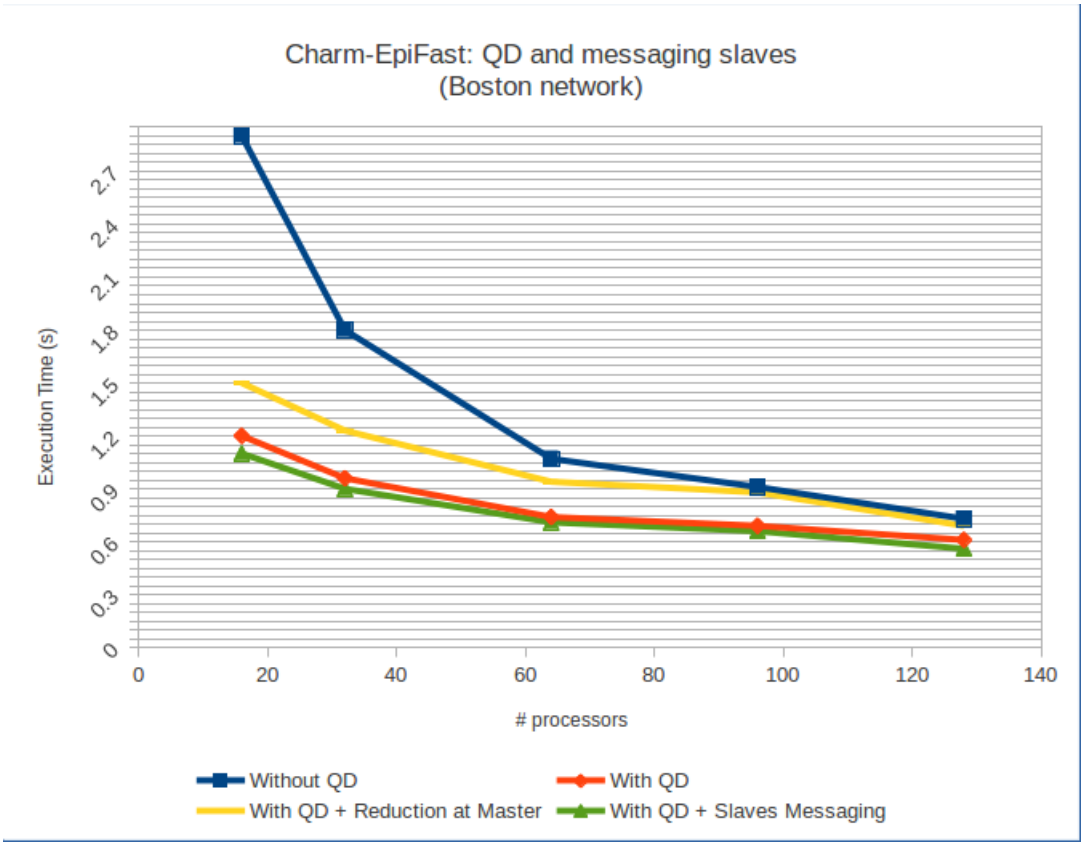

Figure 6.4: Comparison of (a) Unoptimized version, (b) version with quiescence detection, (c) version with quiescence detection and reduction at master chare and (d) version with quiescence detection and slaves communicating transmission directly. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates against number of processors for Boston network. Having slave chares communicate directly eliminates the master chare as a bottleneck leading to improved performance.

Figure 6.5 shows execution times with *section-multicasting* implemented. This version is similar to the one where slaves chares directly communicate but use Charm++'s section multicasting optimization technique. This performs slightly better than the version with directly communicating slaves. It is also shown to be the most efficient version among all other versions.

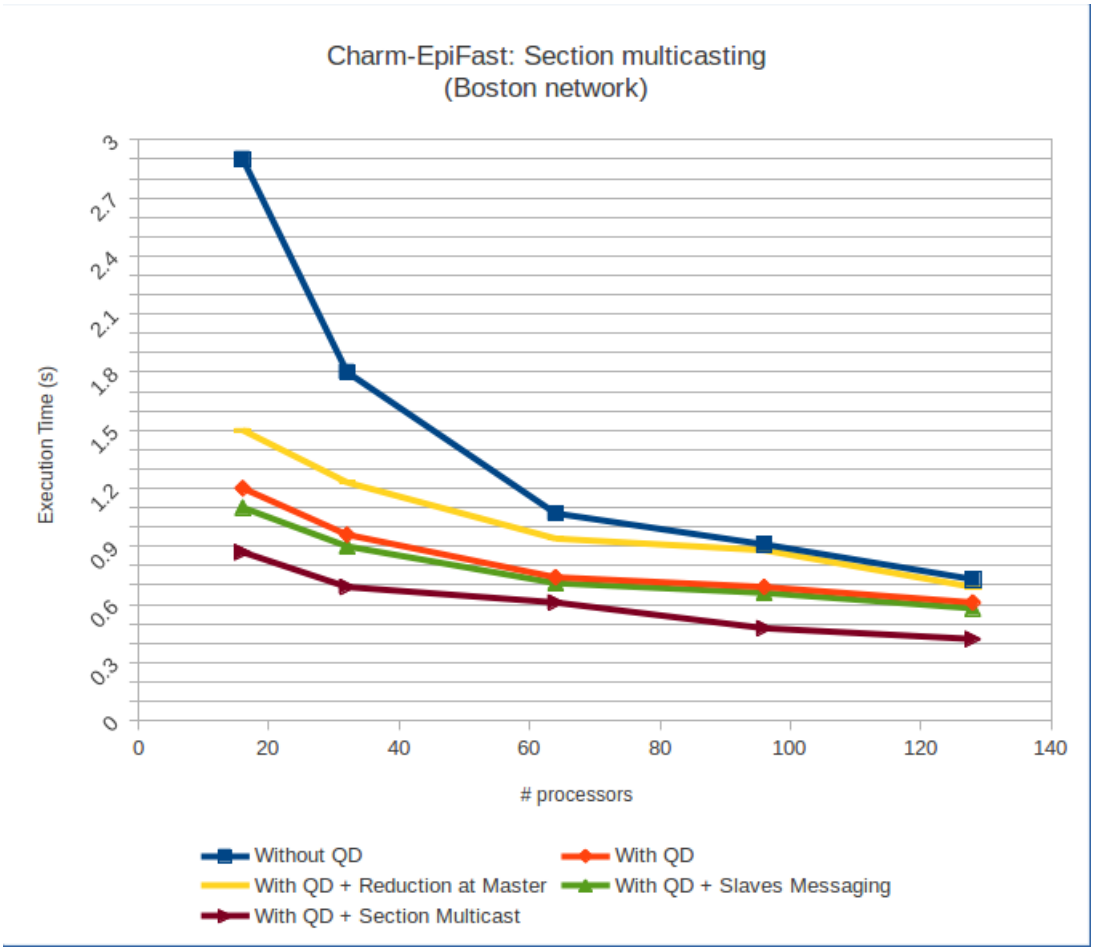

Figure 6.5: Comparison of (a) Unoptimized version, (b) version with quiescence detection, (c) version with quiescence detection and reduction at master chare, (d) version with quiescence detection and slaves communicating transmission directly and (e) version with quiescence detection and section multicasting. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates against number of processors for Boston network. Section multicasting provides better performance when slaves are communicating directly. This version incorporates the two major optimizations (quiescence detection and section multicasting) and is empirically shown to be the best performing among all versions considered.

Figure 6.6 shows quartiles of execution times (in seconds, for 25 replicates) varying number of processors for Boston network, for Charm-EpiFast version with quiescence detection and section multicasting. Table 6.4 shows the mean and variance of the recorded execution times for the same network over 25 replicates.

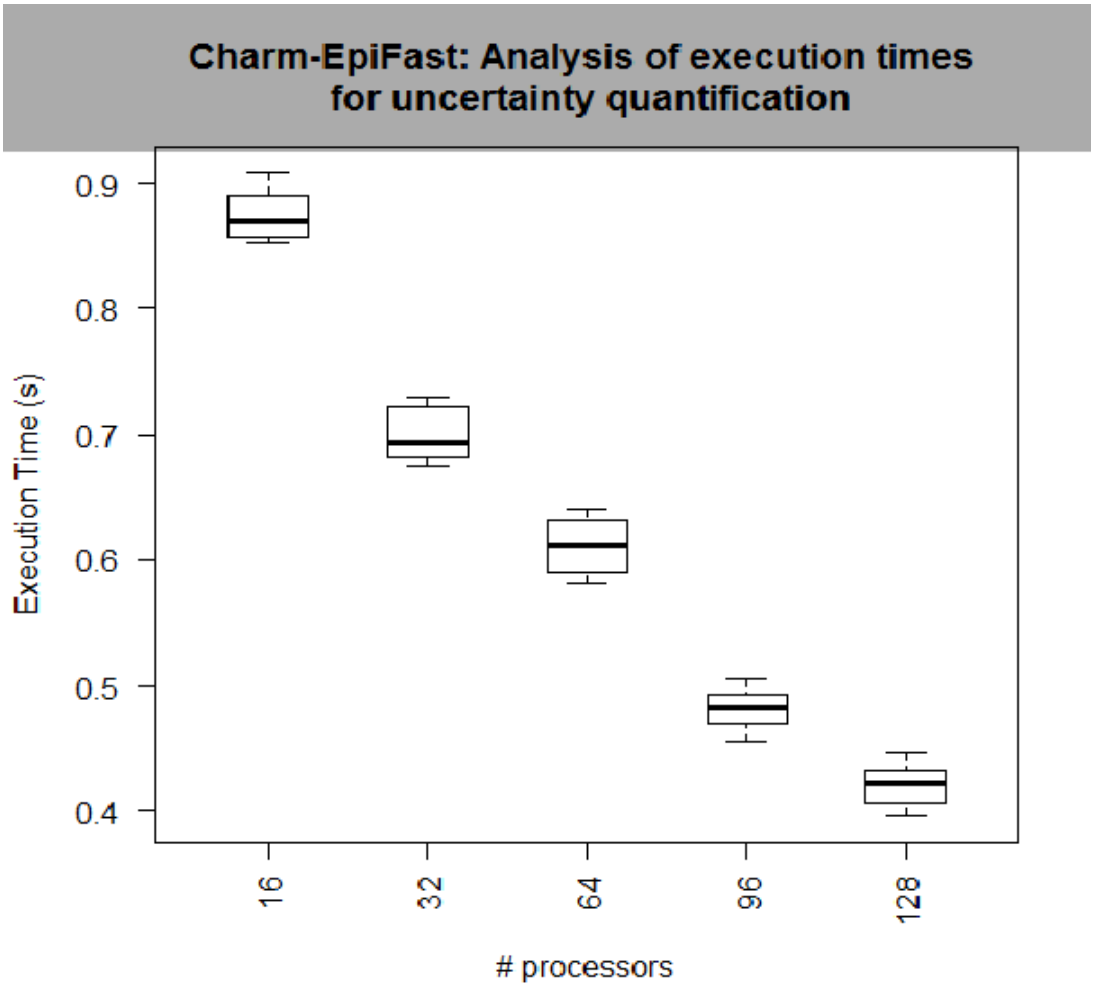

Figure 6.6: Understanding uncertainty in execution times for Charm-EpiFast version with quiescence detection and section multicasting. Shows quartiles of per-day execution times (in seconds, for 25 replicates) varying number of processors for Boston network.

# processors	Mean (s)	Variance
16	0.8745	0.000357
32	0.7011	0.000394
64	0.6122	0.000374
96	0.4819	0.000216
128	0.4211	0.000241

Table 6.4: Uncertainty bounds for execution times for Charm-EpiFast

From this analysis, *quiescence detection* and *section multicasting* appear to be obvious choices for Charm-EpiFast as this combination is efficient than other techniques discussed. We

use the version with *quiescence detection* and *section multicasting* as the basis for further analysis.

We now study the scalability of *Charm-EpiFast* for varying network sizes and number of processor cores. We performed experiments for 25 replicates and 365 simulation days on different network sizes to understand how Charm-EpiFast scales. Figure 6.7 illustrates this for all the networks shown in Table 6.3. It behaves as expected showcasing increase in execution times with increase in network sizes (keeping the number of processor cores constant) and decrease in execution times with increasing processor cores (for constant network size). Figure 6.8 shows that Charm-EpiFast achieves a speedup of up to 1.9× using 16× the number of processor cores.

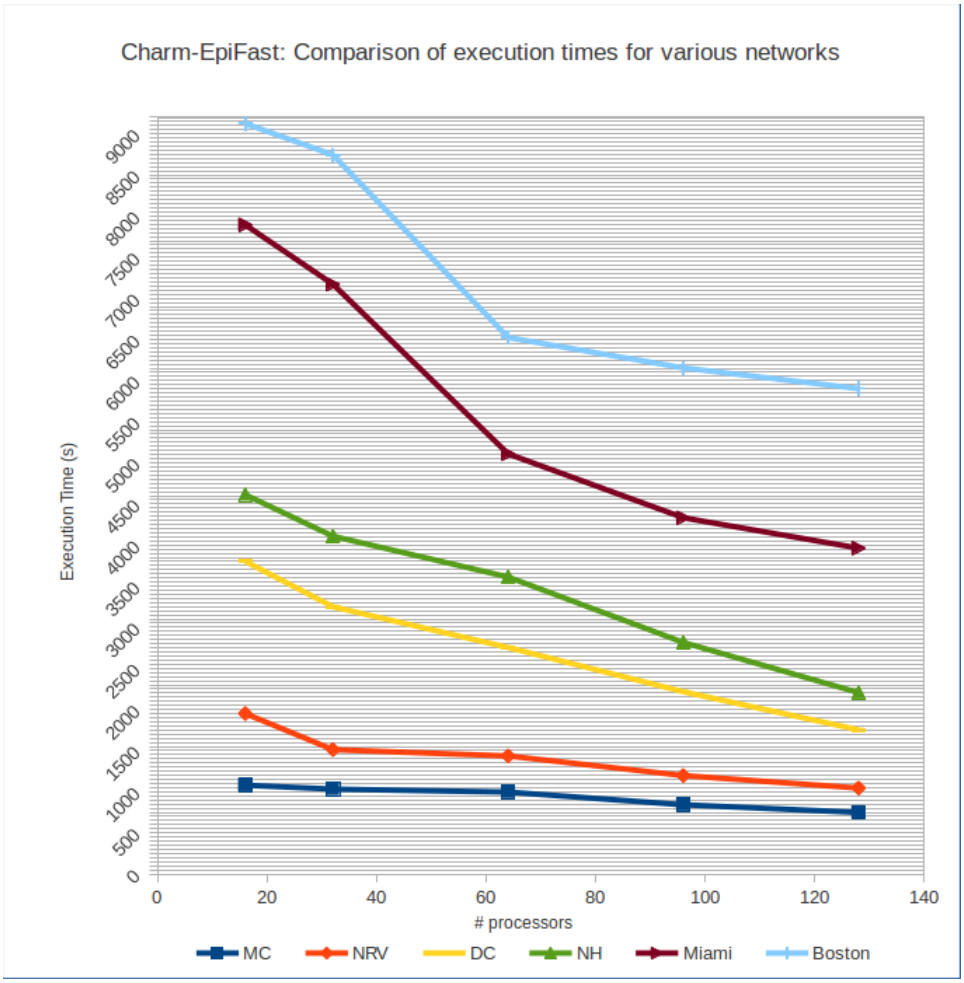

Figure 6.7: Total execution time (in seconds) for a single simulation of 365 days averaged over 25 replicates against number of processors, for all six networks considered. The trend appears to be similar for all networks with execution times increasing with network size.

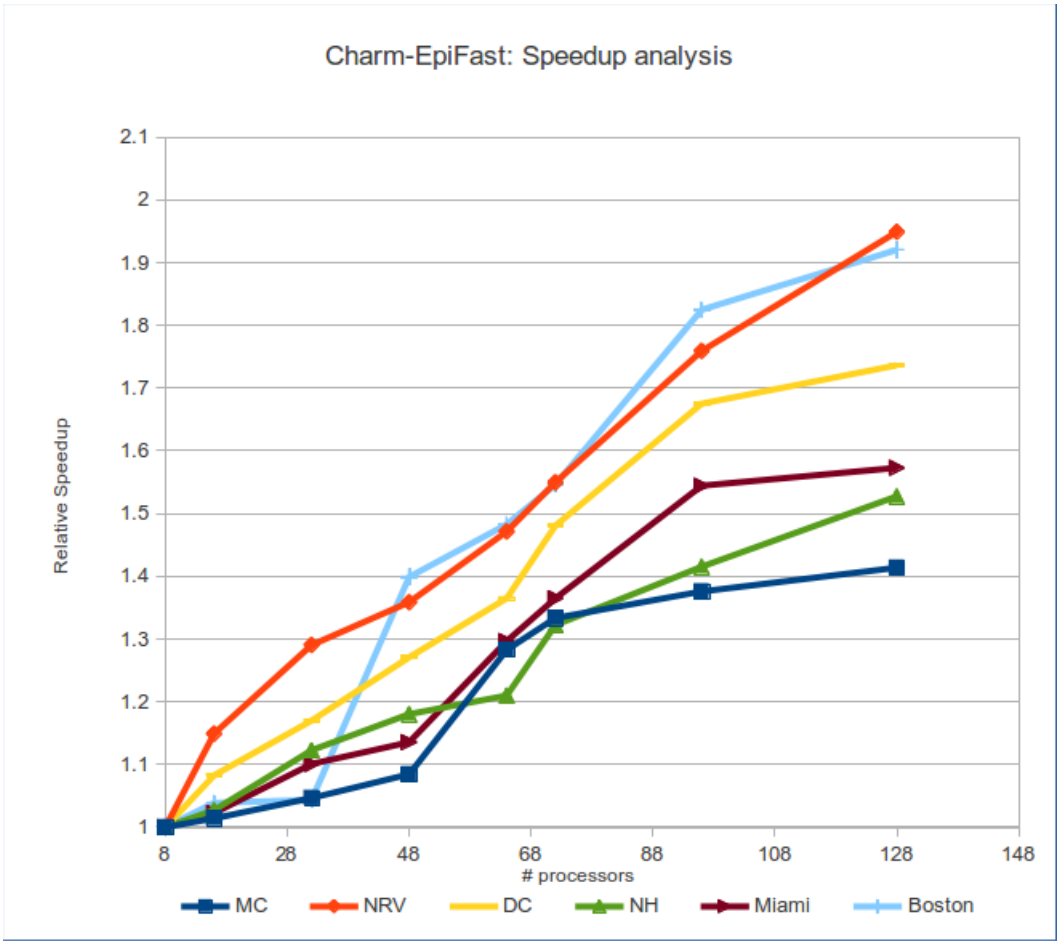

Figure 6.8: Analysis of scalability of Charm-EpiFast. Speedup is relative to 8 processors. Different networks (so different network sizes) are considered with increasing number of processors. Speedup of 1.9× using 16× the number of processors is observed.

Another aspect of Charm-EpiFast is that the communication pattern it follows during a simulation is similar to that of the pure MPI-based version, i.e., execution time tends to increase with increasing number of infected individuals. This behavior can be attributed to the amount of communication due to transmission. Based on the SEIR model, on a given simulation day, the number of transmissions depends on the number of infectious people on that day. Figure 6.9 shows the trend of execution times during a simulation (1 replicate). Like EpiFast, it appears to emulate an epicurve. Though the execution time includes the cost of computations, it is the communication of transmissions that dominates overall execution times. The purpose of this plot is to understand the behavior and cost of communication.

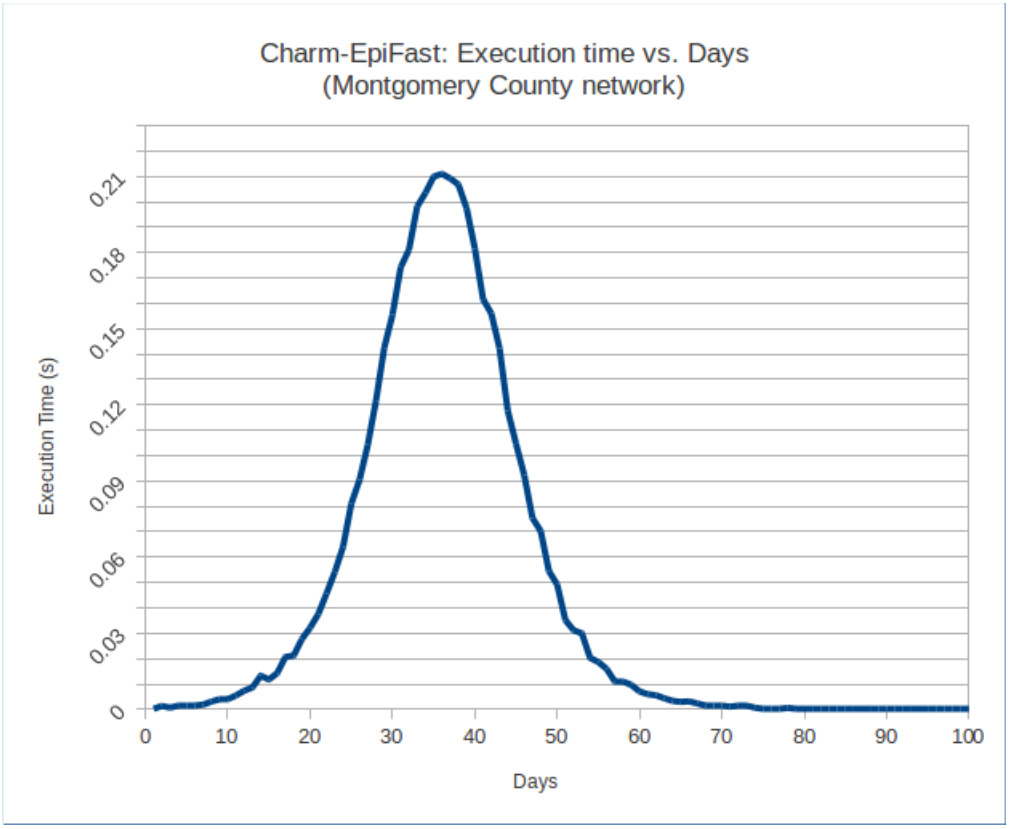

Figure 6.9: Communication pattern in Charm-EpiFast. Shows per-day execution time (in seconds) averaged over 25 replicates, against number of days, for Montgomery County network. This plot suggests that communication is significant - execution time tends to increase as the number of transmissions increase emulating an epicurve (like MPI-EpiFast).

6.2 Analysis of *Spark-EpiFast*

In this section, we provide analysis of our implementation of EpiFast on Apache Spark, i.e., *Spark-EpiFast*. With Spark-EpiFast, we take a similar evaluation approach as with Charm-EpiFast. We consider several important parameters administering the implementation and analyze the performance with respect to each of them.

We have identified the following parameters as key attributes of Spark-EpiFast with respect to performance and efficiency of memory usage.

- RDD partitions: the number of partitions for our VertexRDD and EdgeRDD.
- RDD persistence: storage levels such as MEMORY_ONLY, MEMORY_ONLY_SER, etc. for VertexRDD and EdgeRDD.

- Broadcast variables: sending global shared values from driver to executors through broadcast variables.
- Object serialization: communicating RDDs to executors through serialized objects.
- Transformations on RDDs: basically, *flatMap()* and *collect()* operations on VertexRDD and EdgeRDD.

We first analyze Spark-EpiFast with respect to each of the above parameters individually. We use the same networks as listed in Table 6.3 for the analysis. Later, we provide an analysis of these parameters combined in order to tune Spark-EpiFast for optimal performance.

Figure 6.10 shows the execution times (per day simulation time in seconds) for all the networks against the number of partitions or the level of parallelism of the edge RDDs from a simulation run with 64 cores. Intuitively, parallelism increases with increase in number of partitions. This appears to be the case for all networks initially. However, as the number of partitions increases over a particular threshold, the performance begins to degrade. This decrease is significant for larger networks compared to smaller networks because of the size of the partitions. Partitions of larger networks are larger than partitions of smaller networks which makes the performance to decrease significantly. This plot suggests that the optimal number of partitions of the vertex and edge RDDs is twice the number of available physical cores.

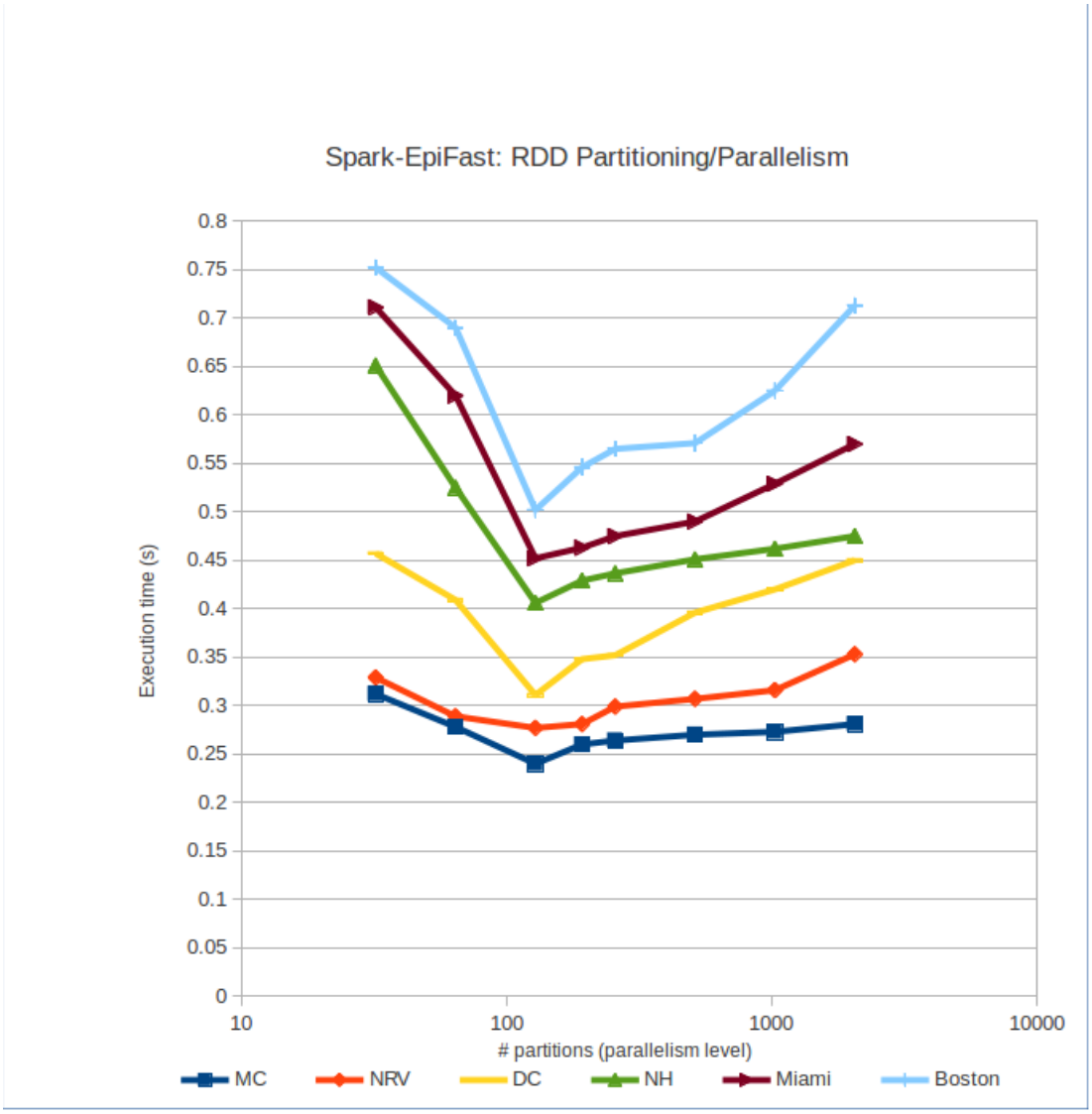

Figure 6.10: Analysis of number of partitions of RDDs. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates against number of partitions for all six networks. The number of processors is kept constant to 64. This plot appears to support the claim that optimal number of partitions is $2\times$ the number of processors (with minimum execution times around 128, in this case.)

Figure 6.11 shows execution times (per day simulation time in seconds) against network sizes (total amount of memory required to store the contact network for a simulation run with 64 cores. We compare four different storage levels we experimented with: DISK_ONLY, MEMORY_ONLY, MEMORY_ONLY_SER and MEMORY_AND_DISK. To understand the behaviour, we restrict the total amount of memory on the driver program to $1GB$. We derive some key observations. DISK_ONLY clearly doesn't appear to help. It is always

useful to include one of the other memory options. For smaller networks (where the entire RDD fits in the memory), all other persistence levels appear to be the same. Networks that require large memory (whose RDDs do not entirely fit in the memory) show some characteristics. In this case, MEMORY_ONLY_SER appears to outperform MEMORY_ONLY and MEMORY_AND_DISK. This is because, once the RDDs do not fit in the memory, Spark automatically switches to using disk space, swapping RDDs whenever required. However, the MEMORY_ONLY_SER stores RDDs as serialized objects consuming less space than the other storage levels leading to an improved performance.

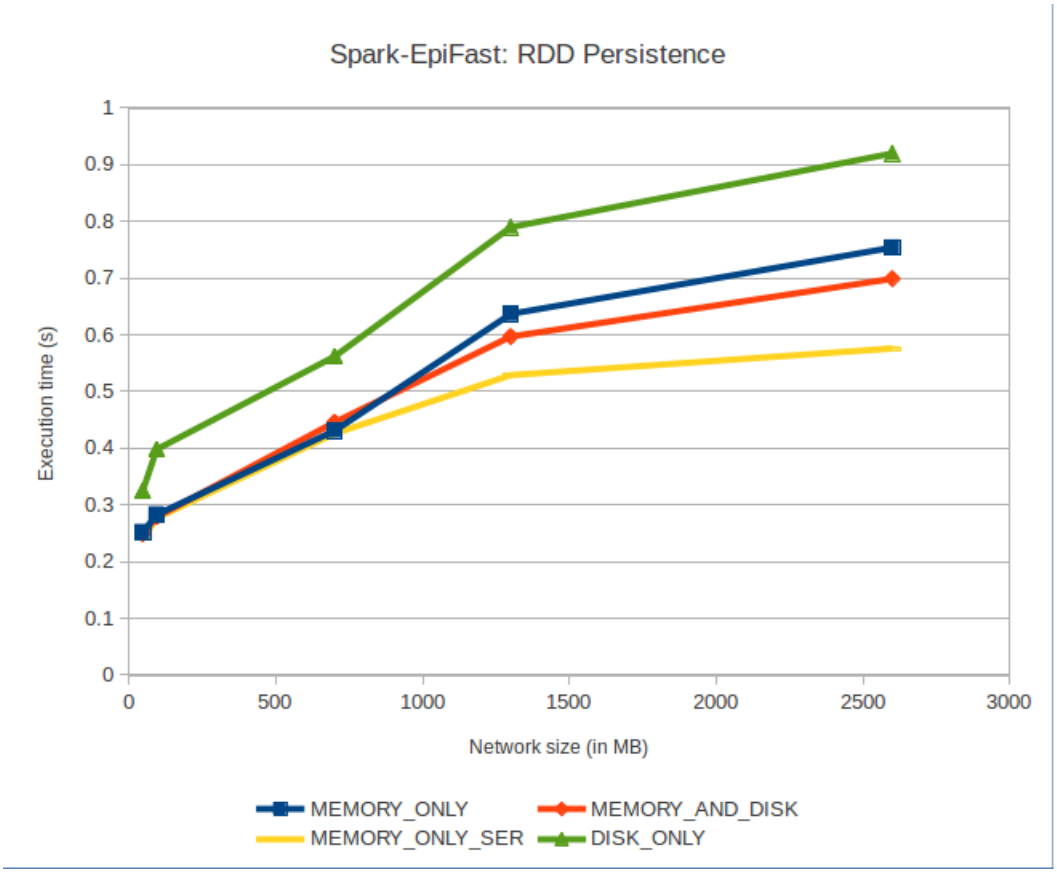

Figure 6.11: Analysis of RDD storage levels. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates for all six networks. DISK_ONLY storage level clearly doesn't help. MEMORY_ONLY_SER appears to have an edge over other options due to implicit serialization.

We also observe the execution times of Spark-EpiFast against simulation days. Figure 6.12 illustrates this for the Montgomery county network. The execution time is higher for a particular day and then drops significantly thereafter. This is because of the persistence of RDDs. This includes the time taken to group the edges based on their source vertices, distribute them to executors and collect results from executors. The initial distribution

consumes time marked by the spike in the plot. Further iterations reuse the persisted data leading to significantly reduced time usage. The drop around day 80 can be explained by the fact that the number of transmissions reduces even further with less to no communication and only by the flatMap() (computation part) contribute to the overall cost.

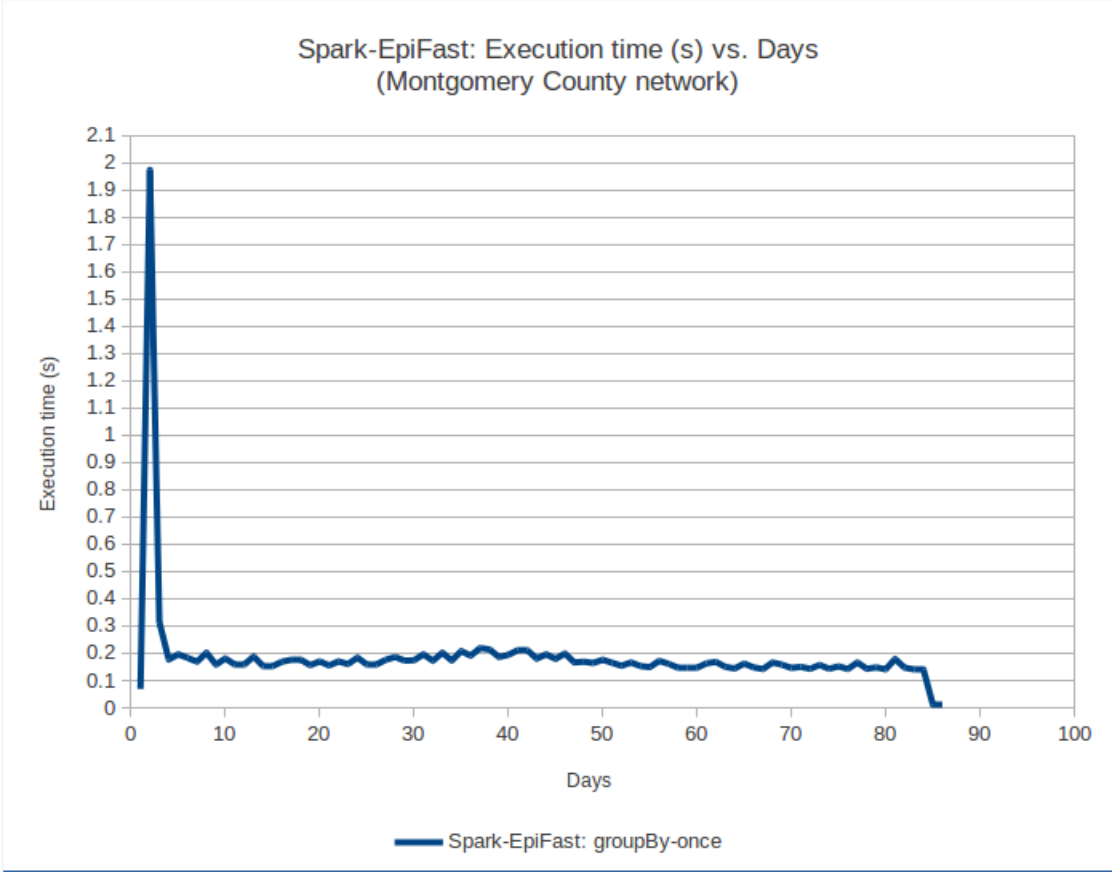

Figure 6.12: Communication pattern in Spark-EpiFast. Shows per-day execution time (in seconds) averaged over 25 replicates, against number of days, for Montgomery County network. This plot suggests that communication is handled efficiently by Spark - execution time tends to depend on the number of transmissions (dominated only by flatMap() and collect() operations for computations) but the initial groupBy() has significantly reduced communication costs.

Figure 6.13 shows a comparison of two different combinations of transformations: (i) *map()* and *coalesce()* and (ii) *flatMap()* and *collect()*. These two combinations essentially serve the same purpose but use different implementation techniques. map() and flatMap() are equivalents for applying a function to all elements in an RDD. In case of using map() which returns a list for each element, we have to explicitly flatten the individual lists to get a single list with all values. Whereas flatMap() performs this implicitly. Similarly, coalesce() and collect() are equivalents of combining results from executors. This comparison helps

us identify the least expensive pair. It is clear from this plot that *flatMap()* and *collect()* combination has a less overhead than *map()* and *coalesce()*.

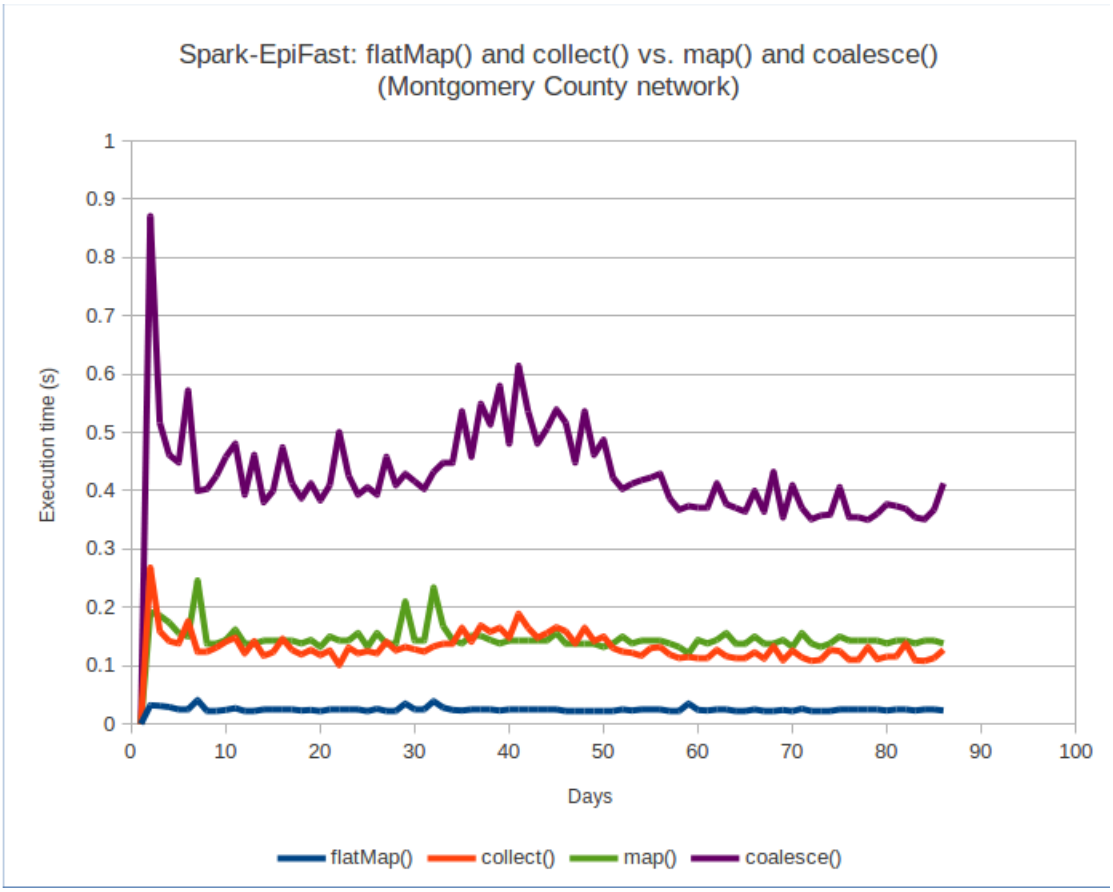

Figure 6.13: Comparison of flatMap()-collect() combination against map()-coalesce() combination. Shows per-day execution time (in seconds) averaged over 25 replicates for Montgomery County network. flatMap()-collect() combination appears to be better than map()-coalesce() combination.

Figure 6.14 shows the overhead of per-iteration groupBy() which is what would be resultant of a traditional MapReduce implementation. However, we show that a single groupBy() along with persistence significantly reduces the overhead and provides much better performance.

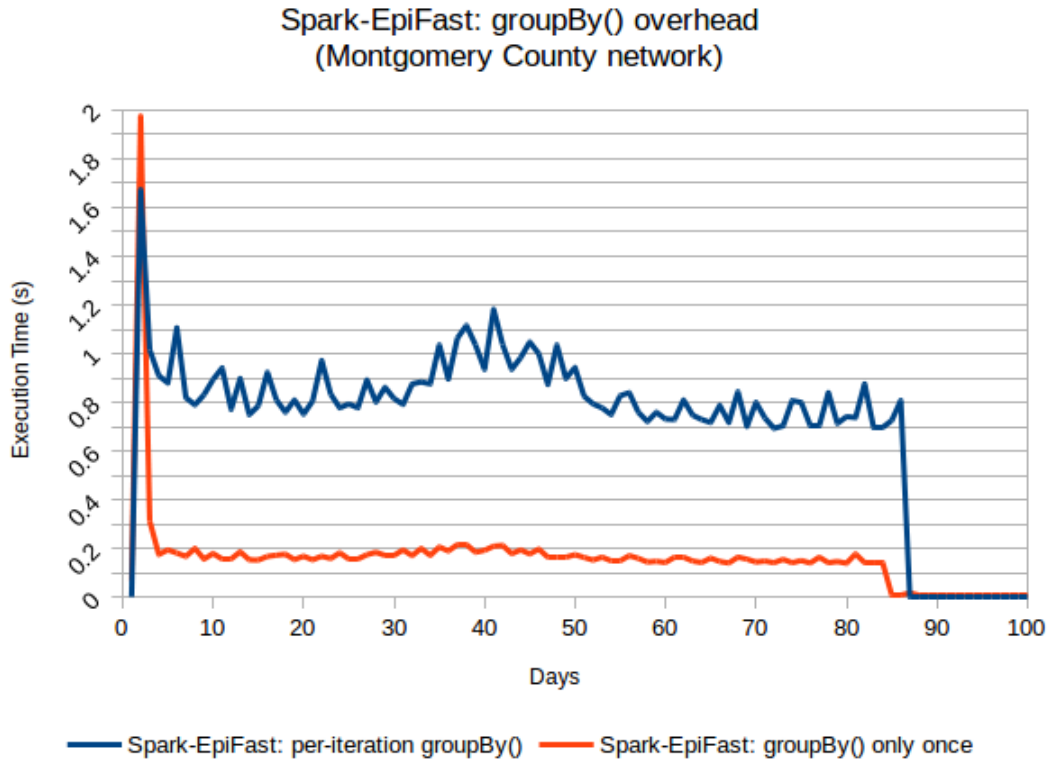

Figure 6.14: Understanding overhead of groupBy(). Shows per-day execution time (in seconds) averaged over 25 replicates for Montgomery County network. This plot compares per-iteration groupBy() against groupBy() only-once. It is apparent that the per-iteration groupBy() (what we would get from traditional MapReduce) has a significant overhead. So we adopt only-once groupBy() which is suitable for EpiFast algorithm.

Figure 6.15 shows the complexity of three basic transformations involved in Spark-EpiFast: *groupBy()*, *flatMap()* and *collect()*. It shows execution time (in milliseconds) against simulation days for the Montgomery County network. Since the RDDs are already distributed among the executors, *flatMap()* and *collect()* do not pose an overhead. The plot shows per-iteration *groupBy()* costs. flatMap() corresponds to the between-host transmission, i.e., identifying the transmissions and collect() corresponds to reducing this list at the driver program for state updates. It is clear that *groupBy()* is an expensive transformation, especially when applied every iteration (i.e., without persistence).

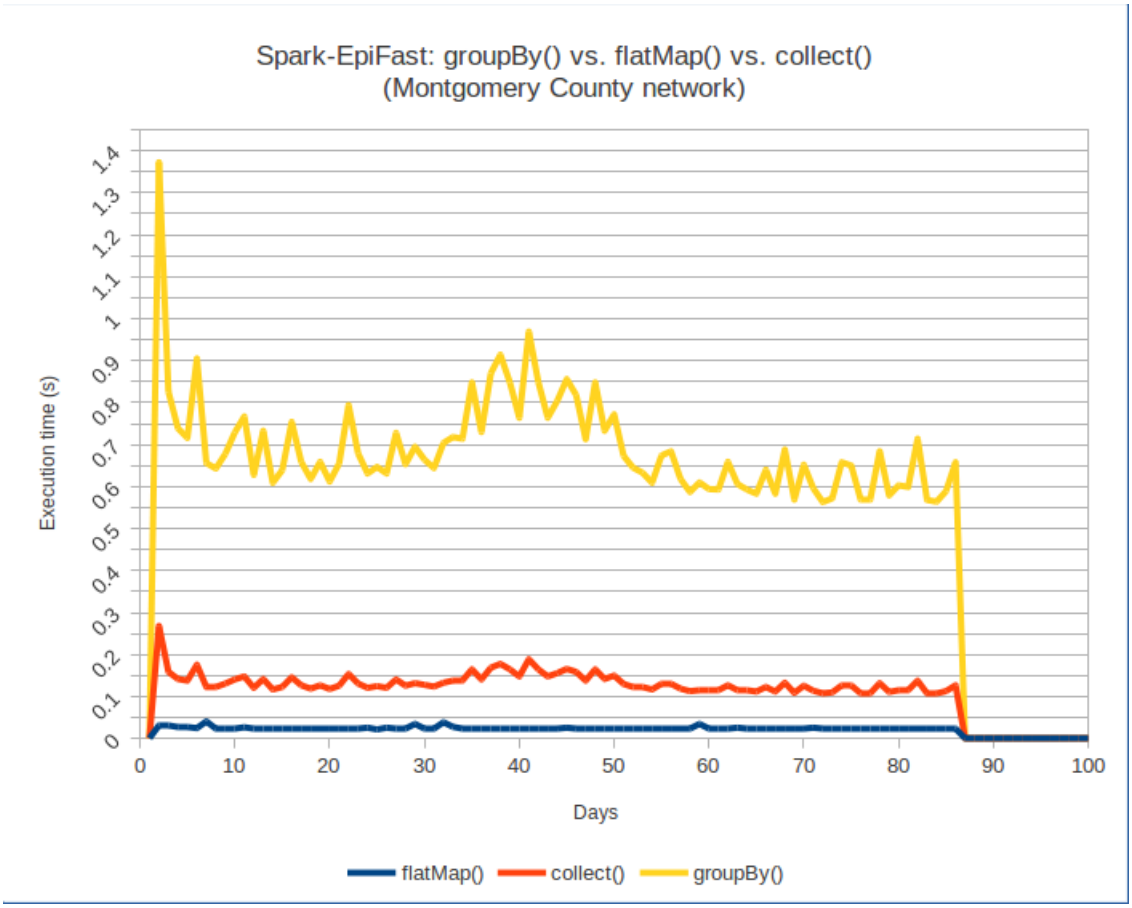

Figure 6.15: Analysis of three major transformations - groupBy(), flatMap() and collect(). Shows per-day execution time (in seconds) averaged over 25 replicates for Montgomery County network. This plot re-iterates the overhead of per-iteration groupBy(). Costs of flatMap() and collect() are much less in comparison.

It is important to note that the cost of transformations such as *flatMap()*, *collect()* and *groupBy()* can be bounded by the size of the RDDs being operated upon and number of partitions of the RDDs.

Figure 6.16 shows a comparison of different versions of Spark-EpiFast each with a different strategy. The experiment considers the Montgomery County network with varying number of partitions. This plot compares

- groupBy() every iteration without persistence,
- groupBy() only once at the beginning and then persisting with MEMORY_ONLY,
- MEMORY_ONLY with Kryo serialization [60] of RDD objects,

- MEMORY_ONLY with Kryo serialization of RDDs and the driver multicasting partitions to executors explicitly.

It is apparent that MEMORY_ONLY with serialization achieves better performance compared to other strategies. Applying groupBy() every iteration seems expensive with increasing partitions.

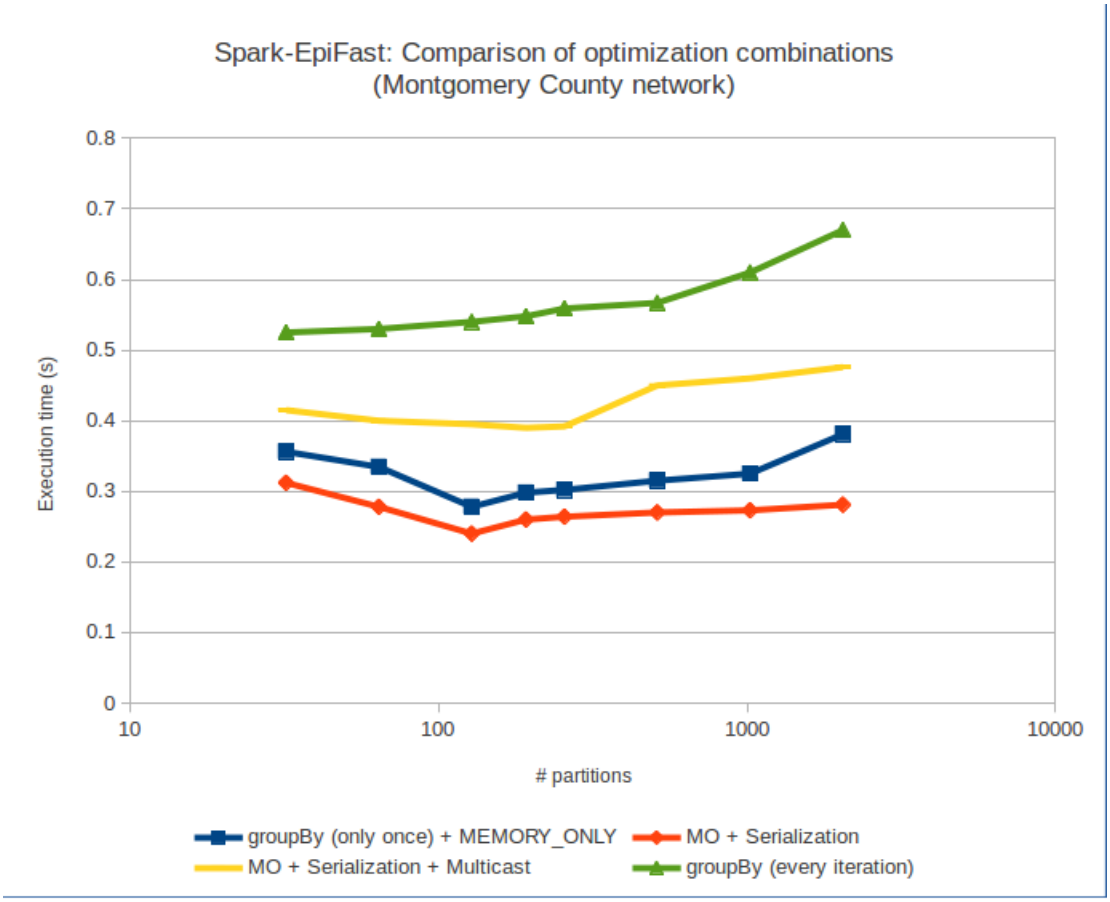

Figure 6.16: Comparison of optimization combinations for Spark-EpiFast. Shows per-day execution time (in seconds) averaged over 365 days with 25 replicates for Montgomery County network. groupBy() only-once (along with MEMORY_ONLY_SER persistence and Kryo serialization) provides better performance compared to other strategies.

Figure 6.17 shows quartiles of execution times (in seconds, for 25 replicates) varying number of partitions for Montgomery County network, for Charm-EpiFast version with quiescence detection and section multicasting. Table 6.5 shows the mean and variance of the recorded execution times for the same network over 25 replicates.

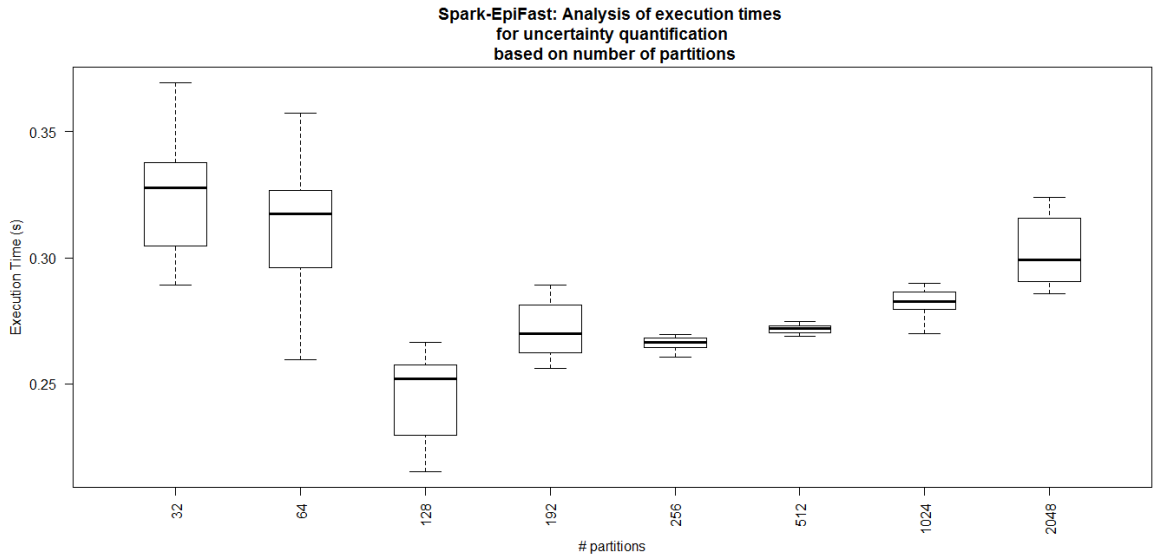

Figure 6.17: Understanding uncertainty in execution times for Spark-EpiFast version with groupBy() only-once, MEMORY_ONLY_SER persistence and Kryo serialization. Shows quartiles of per-day execution times (in seconds, for 25 replicates) varying number of processors for Montgomery County network.

# partitions	Mean (s)	Variance
32	0.3231	0.00036
64	0.3126	0.00071
128	0.2439	0.00028
192	0.2715	0.00012
256	0.2658	0.00007
512	0.2717	0.00002
1024	0.2821	0.00003
2048	0.3023	0.00017

Table 6.5: Uncertainty bounds for execution times for Spark-EpiFast

Figure 6.18 compares speedup of Spark-EpiFast relative to 8 processors for all the networks. This shows the *strong-scaling* aspect of Spark-EpiFast. It is able to scale well with number of processors achieving a speedup of more than 3×.

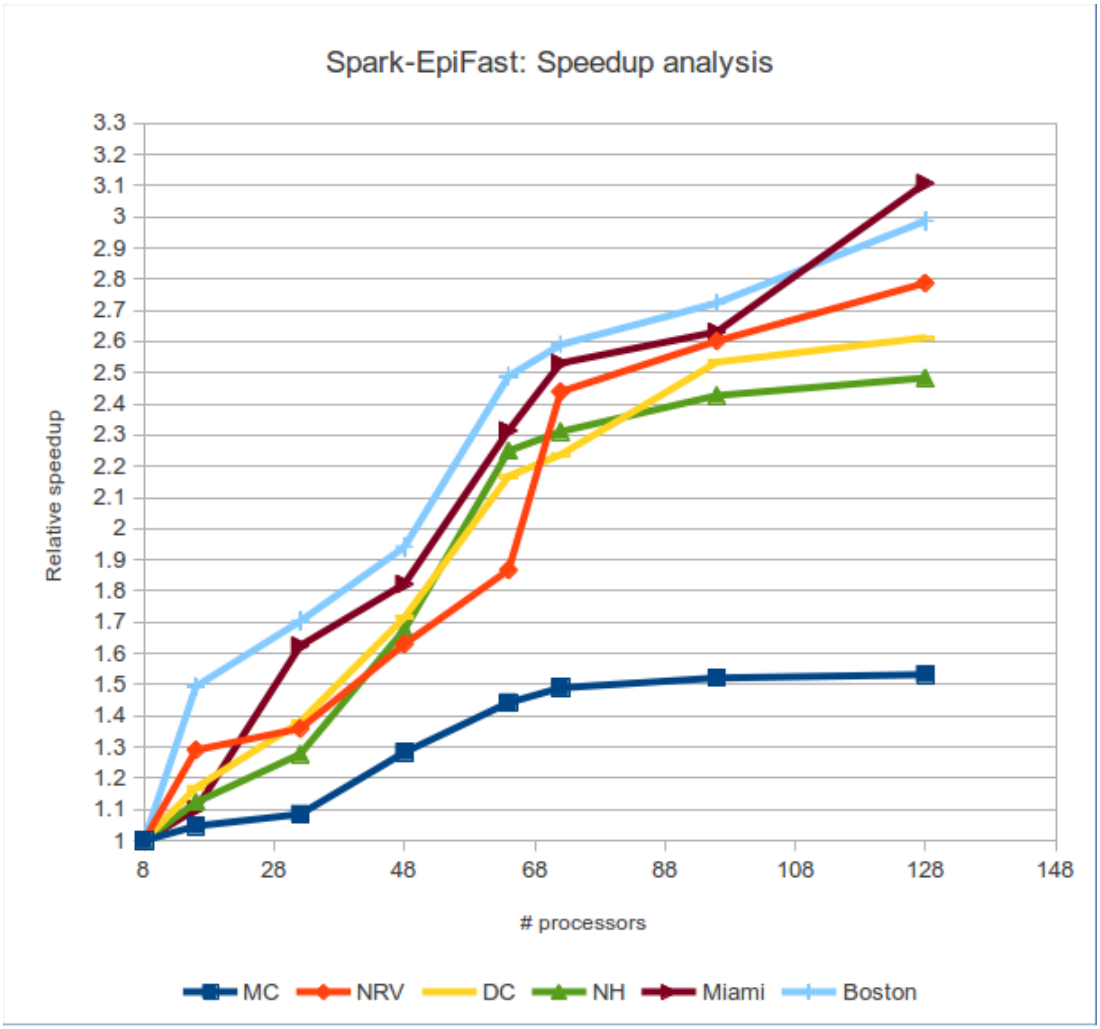

Figure 6.18: Analysis of scalability of Spark-EpiFast. Speedup is relative to 8 processors. Different networks (so different network sizes) are considered with increasing number of processors. Speedup of 3× using 16× the number of processors is observed.

6.3 Comparison with original MPI-based EpiFast

Figure 6.19 compares the execution times of Spark-EpiFast and Charm-EpiFast with original MPI-EpiFast for the Montgomery County network. The purpose of this plot is to understand the differences in communication pattern among the different versions. The curves for Charm-EpiFast and MPI-EpiFast appear to follow epicurve as previously discussed and that of Spark-EpiFast is significantly different due to persistence of RDDs. The number of infections over the course of the simulation do not appear to affect the performance once the RDDs are persisted at the executors.

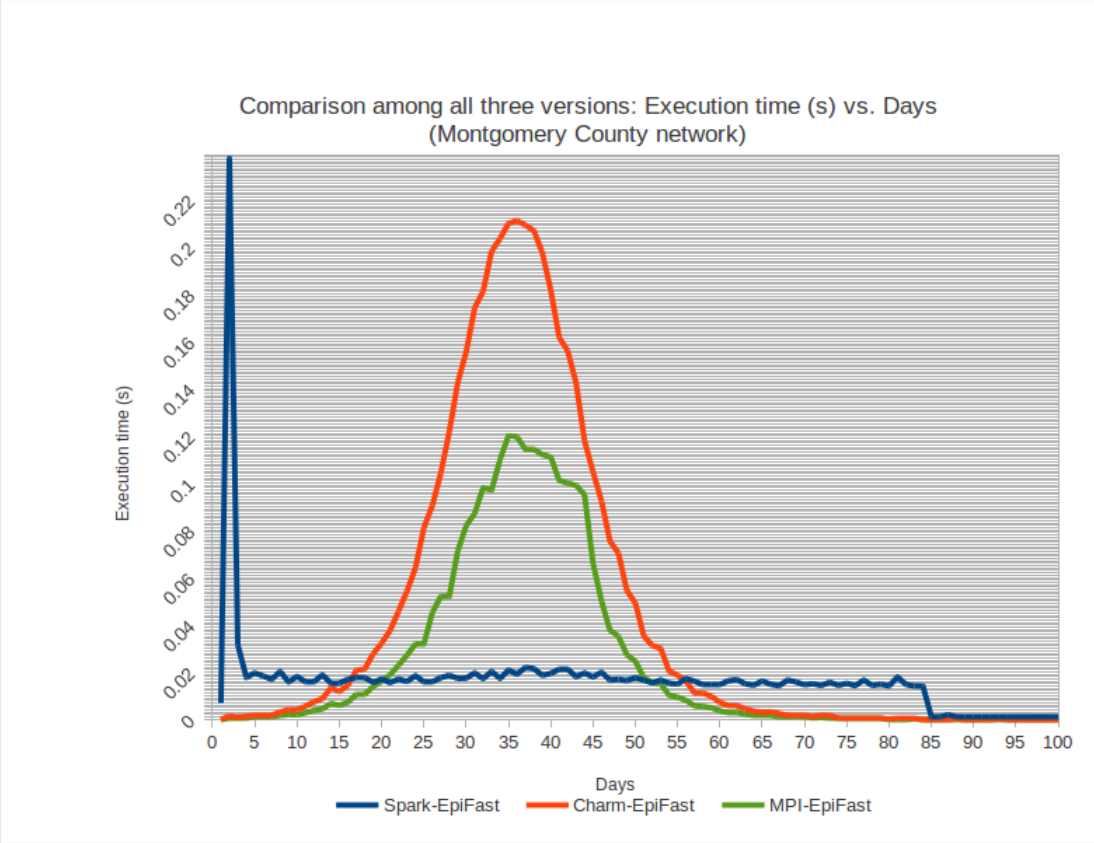

Figure 6.19: Comparison of communication pattern among the versions. Shows per-day execution time (in seconds) averaged over 25 replicates, against number of days, for Montgomery County network. Reiterates the epicurve-like behavior of MPI-EpiFast and Charm-EpiFast. For Spark-EpiFast, the execution times are dependent on the flatMap() and collect() transformations with a high initial cost for the groupBy() transformation.

Figure 6.20 and Figure 6.21 show a comparison of speedup and efficiency (relative to 8 processors) of all three versions, for the Boston network. Spark-EpiFast clearly appears to have an edge over the other two versions. On the other hand, Charm-EpiFast achieves relatively less speedup factors. From this, we can conclude that Spark helps realize some of the performance benefits over other parallel programming systems for EpiFast.

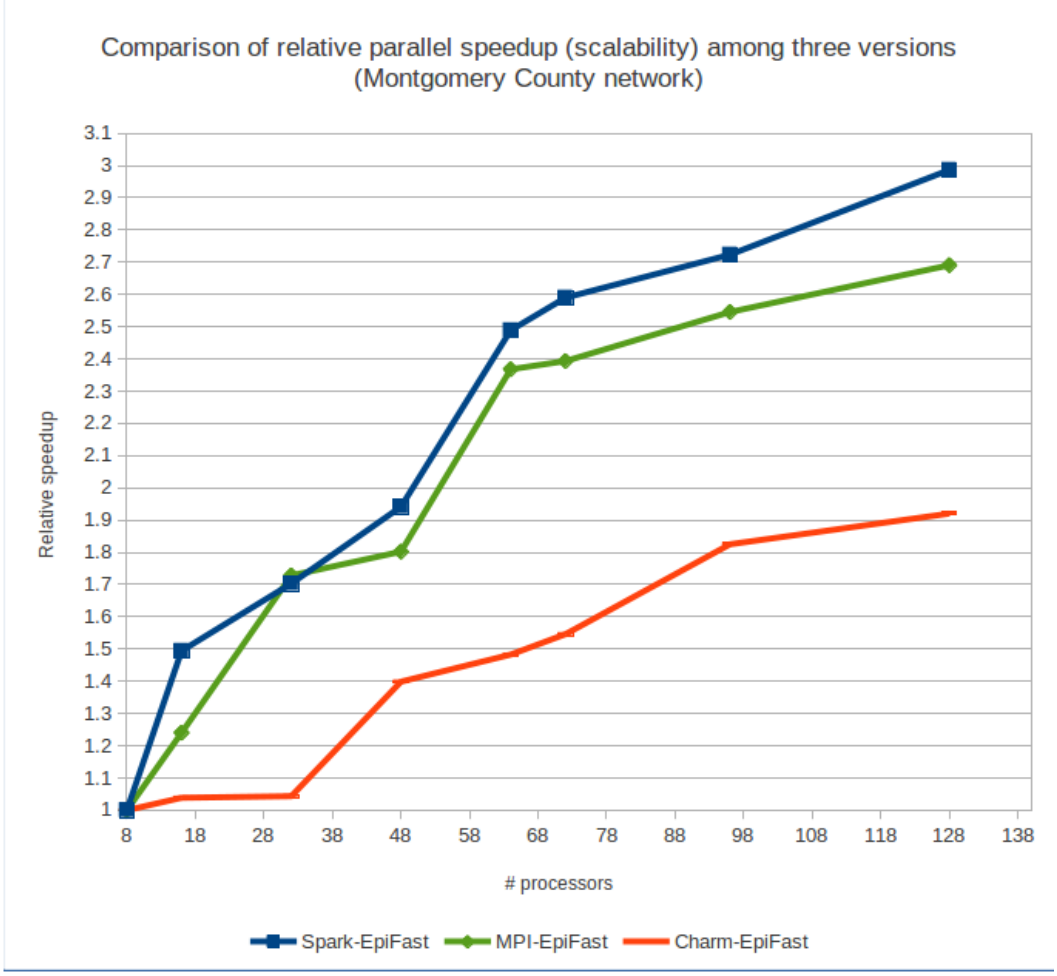

Figure 6.20: Comparison of speedup among the versions for Montgomery County network. Speedup is relative to 8 processors. Similar trends are observed. However, Spark-EpiFast achieves 3× speedup compared to 2.7× and 1.9× for MPI-EpiFast and Charm-EpiFast respectively, for 16× the number of processors.

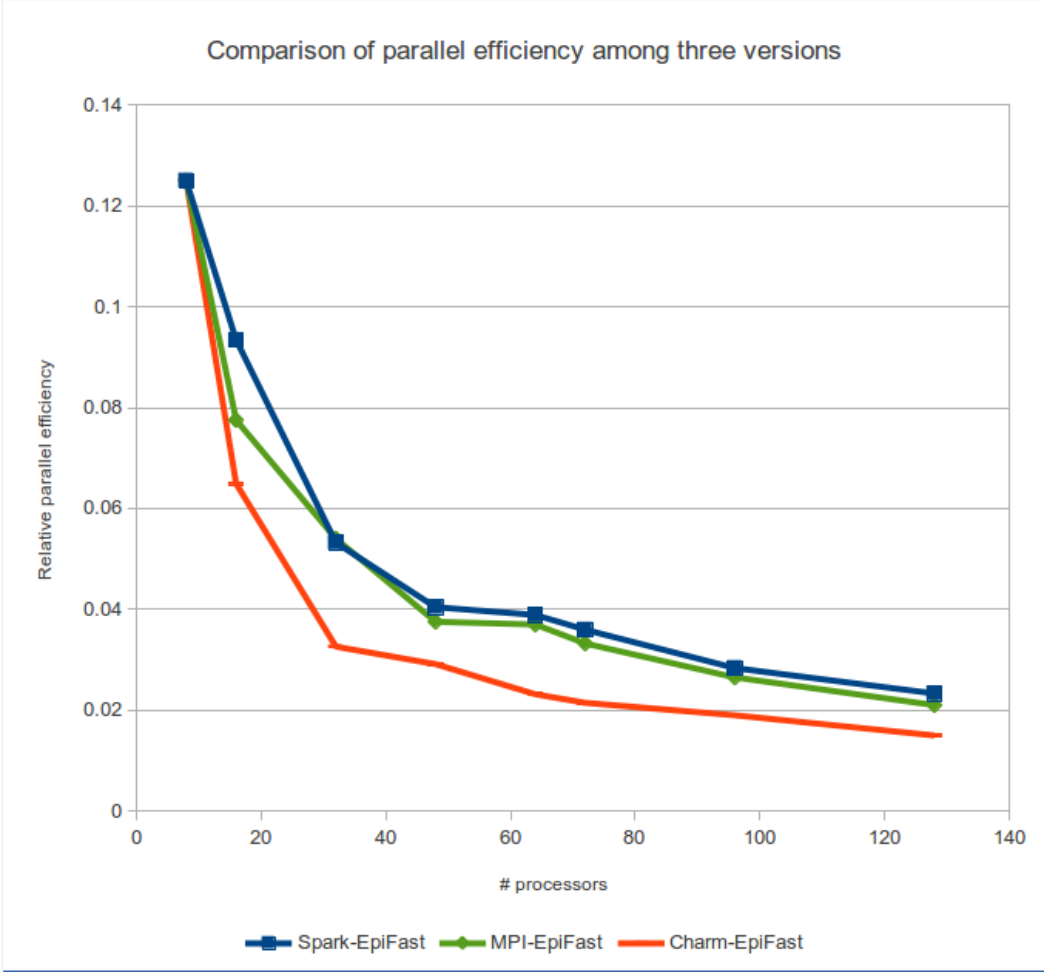

Figure 6.21: Comparison of efficiency among the versions for Montgomery County network. Efficiency is relative to 8 processors. Similar trends are observed. However, Spark-EpiFast achieves higher efficiency compared to other two versions.

Another aspect of the implementations we compare is the overhead of I/O. Large graphs potentially impose a large I/O overhead in our implementations. MPI-EpiFast has the ability to perform distributed read/writes due to the finer level of control over individual processes. Charm-EpiFast, on the other hand, currently does not support parallel read/writes. Spark-EpiFast suffers significantly when compared to the other two versions. The major reason behind this is the lack of support of distributed read/writes of binary data in Spark. Spark has dedicated APIs for distributed read/writes but are specific to the Hadoop platform such as HDFS. In the emulated cluster setup we used, memory constraints along with the lack of APIs significantly incapacitates Spark for large scale I/O of binary data. Figure 6.22 illustrates this comparison. This plot shows the percentage of total execution time consumed by I/O (averaged over 10 replicates).

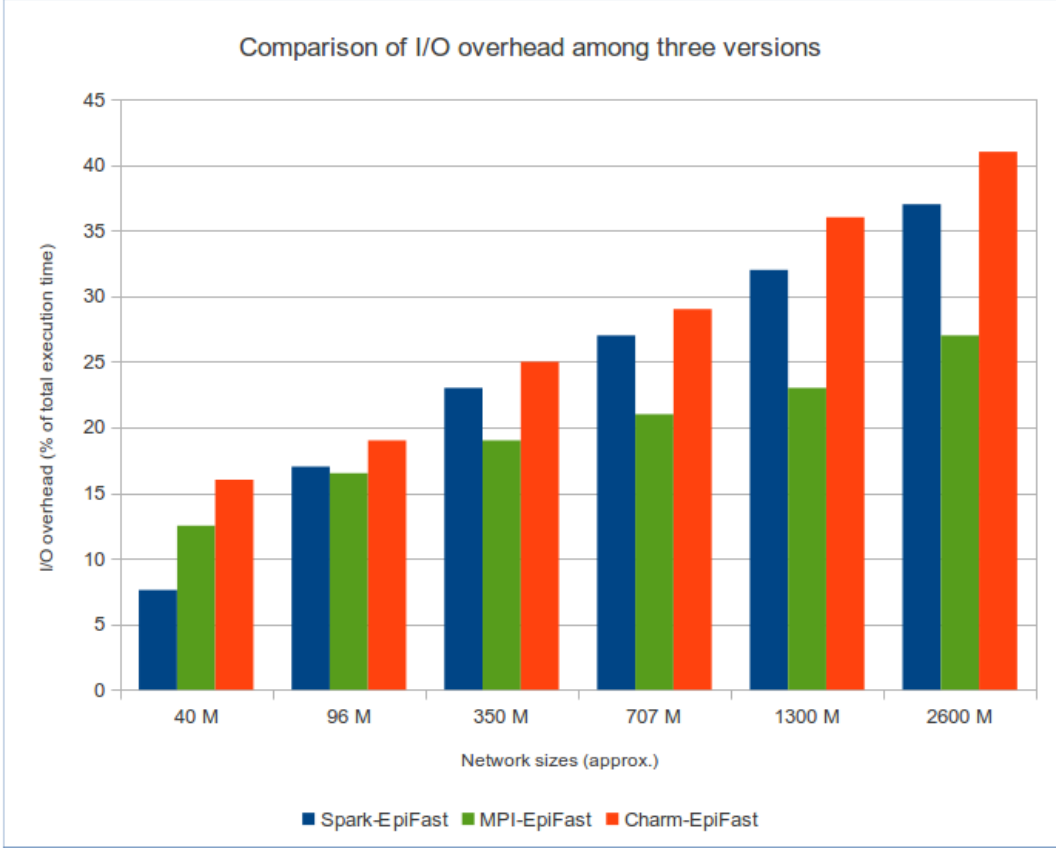

Figure 6.22: Comparison of I/O overhead among all versions. Shows cost of I/O as a percentage of total execution time for all networks considered (averaged over 10 replicates). I/O appears expensive in Charm-EpiFast and Spark-EpiFast (due to lack of support for distributed read/writes).

We would also like to point out the fault-tolerant capabilities of the implementations. Computations using traditional libraries such as MPI are much less fault-tolerant. They fail completely when one component fails and it becomes the responsibility of the developer to manually account for fault-tolerance at the application level. This is characteristic of MPI-EpiFast. On the other hand are systems such as Charm++ and Spark that automatically provide fault-tolerance to applications. Charm++ provides APIs for check-pointing portions of the applications to register modules for quick recovery. Spark provides this by default through its abstraction of RDDs. RDDs are automatically check-pointed at the executors and are capable of reconstructing data from lineage information available at the executors. Thus, we believe Spark has an edge over the other two systems discussed with respect to fault-tolerance.

Another aspect of Apache Spark is its ease-of-use. While MPI's C/C++ based libraries take significant levels of effort for a simple task, Spark's expressive constructs consume very

little effort. We illustrate this using Figure 6.23. We consider different tasks inherent to EpiFast and compare estimates of the number of lines of code required by the MPI-version against those of Spark's APIs. With the level of abstractions provided by Spark, it is apparent that it saves a lot of developer effort and has high productivity. For example, in addition to implementing algorithm specifics, developers have to explicitly provide definitions of several low-level tasks (such as instantiating processes, message packing/unpacking, etc.) which require significant amount of efforts. On the other hand, Spark-like engines provide frameworks for easier programming, usually hiding complexities from developers, making it a viable option for developing large applications.

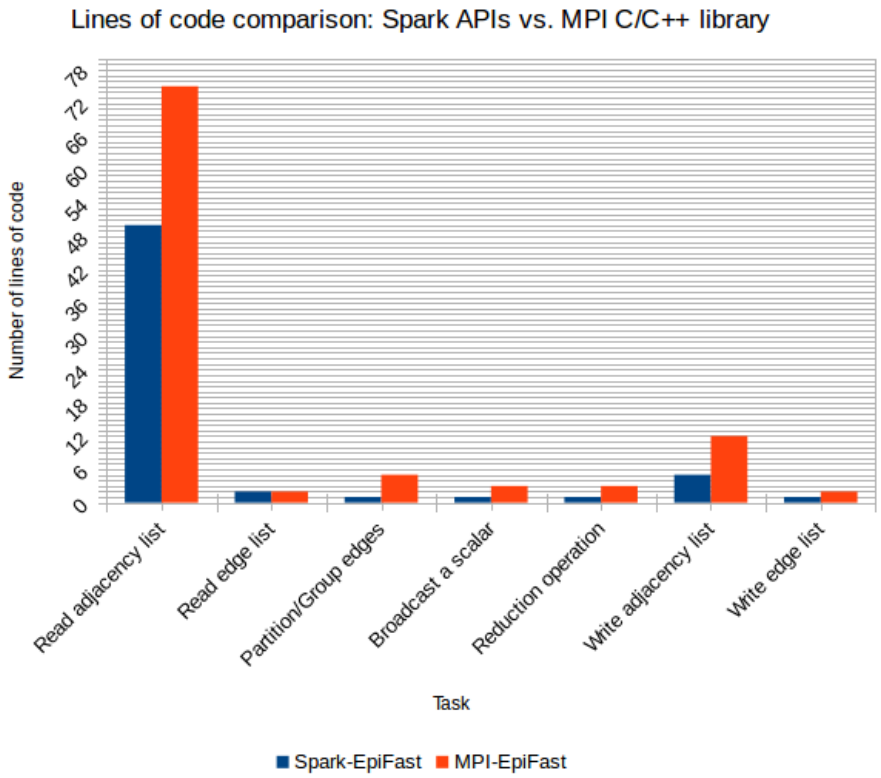

Figure 6.23: Comparison of Spark's API vs. MPI C/C++ library. Shows number of lines of code for various tasks such as reading and writing adjacency lists and edge lists, broadcasting scalar variables, and, partitioning edges. Spark's concise constructs reduces development time and improves productivity.

We provide few snippets of Scala code for few basic operations to illustrate how expressive Spark's constructs are. It is clear from these examples, that Spark provides simplicity and ease-of-use, thereby consuming significantly less amount of developer effort.

Listing 6.1: Partition/Group edges by source vertex

```
val groupedEdges = graph.edges.groupBy[VertexId](groupBySourceVertex)
```

Listing 6.2: Broadcast a scalar to all processes

```
val broadcastValue = sparkContext.broadcast(scalarVariable)
```

Listing 6.3: Writing EFIG6Bb format

```
graph.edges.map(e => e.srcId).foreach(tailId => print(tailId + " "))
graph.edges.map(e => e.dstId).foreach(headId => print(headId + " "))
graph.edges.map(e => e.attr._1).foreach(dur => print(dur + " "))
graph.edges.map(e => e.attr._2).foreach(tType => print(tType + " "))
graph.edges.map(e => e.attr._3).foreach(hType => print(hType + " "))
```

Listing 6.4: Writing EFIG5 format

```
graph.edges.foreach(println)
```

Table 6.6 summarize the observations from our experimental analysis.

1. Charm-EpiFast

- Reduction of results (relaying transmission through the master) at *main* chare is expensive. **Section multicasting** improves performance significantly.
- Automatic synchronization through **quiescence detection** improves performance.
- Relative parallel speedup of up to 2× are achievable.

2. Spark-EpiFast

- **RDD partitions:** Optimal value of number of partitions of vertex and edge RDDs is twice the number of available physical cores.
- **RDD persistence:**
  - *DISK_ ONLY* storage level has the worst performance.
  - Storage levels with ability to persist RDDs in-memory improve performance. More specifically, ***MEMORY_ ONLY_ SER*** appears to outperform MEMORY_ONLY and MEMORY_AND_DISK options due to efficient use of memory through Kryo serialization.
- **Transformations:**
  - *flatMap()* and *collect()* operations are identified as suitable equivalents of map and reduce operations.
  - The natural *groupBy() only-once* combined with persistence achieves better performance.
- Relative parallel speedup of up to 3.2× are achievable.

3. Comparison among all three models

- Charm-EpiFast achieves relatively less speedup factors. Spark-EpiFast appears to have an edge over the other two versions.
- On a non-Hadoop setup, I/O overhead of Spark-EpiFast is significant due to the lack of ability to perform distributed reads.
- Use of RDDs makes fault-tolerance implicit in Spark-EpiFast leading to increased reliability compared to other two versions.
- Concise and expressive constructs of Spark improves developer productivity of Spark-EpiFast compared to other two versions.

Table 6.6: Summary of observations from experimental analysis

Chapter 7

Future Work

In this chapter, we discuss some of the opportunities for improvements, from functional and performance perspectives, of both the tools.

7.1 Potential improvements for *Charm-EpiFast*

Our version of *Charm-EpiFast* has a few limitations. We outline them here.

- Support for interventions: The lack of support for interventions reduces the potential of Charm-EpiFast. We believe that incorporating interventions into the existing implementation will certainly make the tool more realistic. It would also be interesting to see how interventions affect the performance of the tool. We would like to perform a variety of experiments to analyze the performance and simulation results after incorporating interventions.
- The performance section mentioned that the overhead of I/O is significant in Charm-EpiFast. This is mainly attributed to the inability to perform parallel reads. We propose to use a similar technique as in MPI-based EpiFast to have each chare read a portion of the input binary file. We plan to design a new file format that makes distributed reads possible. For example, we can have a file format that ensures all the information related to a single vertex are together. With such a format, we would be able to have each chare read only the portion of the file that it will be responsible for, totally eliminating the network distribution phase. We believe that this approach will significantly reduce the I/O overhead of the tool.
- Scalability to very large networks: The current version is limited to network sizes of 5M (approx.) in a reasonably sized cluster, due to the lack of several other possible optimizations. An ideal situation is where Charm-EpiFast can support networks of

the entire US population. We believe the following optimization techniques can help achieve scalability and performance gains on such large scales.

- Topology Aware Routing - [39] which is a Charm++ library for optimizing multicasts (many-to-many or all-to-all communications) in applications. Since, messaging constitutes the majority of execution times, it is important that we limit the number of messages during the simulation by taking a processor-chare-mapping-aware approach. The knowledge of the locations of chares can significantly contribute to reducing execution times.
- Message Aggregation - Charm++ also provides libraries [39] for aggregating messages intended for the same recipient. In Charm-EpiFast, we can aggregate messages to receiver chares located on the same physical processor to cut down the number of messages in-flight simultaneously. This is known to significantly reduce communication overhead in message-intensive applications.

7.2 Potential improvements for *Spark-EpiFast*

As with *Spark-EpiFast*, we have identified the following as potential improvements.

- Performing distributed reads: Similar to Char-EpiFast, the overhead of I/O is significantly higher. We believe that either a streaming binary file or a distributed file format could reduce the overhead of reading contact networks. With a distributed file format, it could be easier to have all executors read their own partitions before the start of the simulation. This will reduce the load on the ApplicationDriver to load the contact network and distribute the partitions to the executors.
- Integrating with an Hadoop ecosystem: Our current experimental setup emulates a cluster. We installed standalone Spark on several machines and designated a chosen machine to be the master. We specified the list of machines that can be used as slaves while configuring Spark. It would be interesting to see what an actual cluster setup running Spark built for Hadoop provides in terms of management and performance compared to the emulated cluster setup we used. We believe that this option would help manage Spark applications easier.
- We also plan to understand Spark's execution model at a more granular level with respect to the various tasks involved in our application. One of the important objectives in this aspect is to understand the number of mapper and/or reducer tasks that Spark uses to handle a particular task. This will also help us identify bottlenecks (with respect to transformations) in the application and explore other options. With the current setup, it could be possible to parse and analyze Spark's logs to get a better

insight of application execution such as the number of stages a task requires, the number of mapper and reducer tasks at each stage, etc.

- In Chapter 6, we provided a brief analysis of uncertainties in recorded execution times using quartile plots and mean/variance values. It would be useful to perform thorough statistical analysis of all the parameters and understand better the behavior of the application.

- Exploring *Spark SQL*: There have been previous attempts such as [62] to use distributed databases and query-based approaches to work with interventions in epidemic simulations. Given that Spark has support for query-based processing due to its programming model and abstractions, we believe that *Spark SQL* could make incorporating interventions seamless. We believe it would be worthwhile to explore Spark SQL for this purpose.

- GraphX streaming for interventions: Spark's developer community has offered to incorporate capabilities to the GraphX APIs to handle streaming data. These APIs are envisioned to accommodate dynamic changes to graph structures efficiently. For example, operations on graphs such as addition or removal of nodes and/or edges are to be included in the APIs. With respect to EpiFast, there is support for interventions that change the structure of the underlying network. This often involves adding and/or removing vertices and edges. We believe that such intervention actions are ideal use-cases for the streaming GraphX APIs. The current design of spark-EpiFast makes it difficult to incorporate such interventions. Adding or removing vertices or edges often require redistribution of the partitions thereby negating the usefulness of persistence. However, with streaming GraphX it could be possible to handle such interventions efficiently.

Chapter 8

Discussion

8.1 Parallel and Distributed Processing

With the advent of multiprocessor machines and clusters of systems, parallel and distributed processing is fast becoming default models for a wide variety of applications. The scale of data involved in such applications implicitly drive the use of parallel and distributed programming models. High-performance computing research has seen numerous systems suitable for parallel applications. They usually focus on a subset of properties of programming models and provide features that make parallel processing efficient. In this thesis, we focused on three such particular systems namely *MPI*, *Charm++* and *Apache Spark*. We would like to outline their characteristics with respect to selected attributes. We compare the three systems on several aspects such as programming model, execution model, class of applications suitable for the systems, etc. We also provide our ratings on few characteristics we consider important for parallel applications, as a guide for application developers. Table 8.1 summarizes our observations.

- *Communication*: MPI is significantly incapacitated due to its high communication overhead due to its direct message-passing paradigm. Charm++ due to its asynchronous messaging model, and Spark due to its *just-in-time* transformation operations do not suffer from this drawback.
- *Latency tolerance*: Charm++ provides high tolerance to latency as it follows an on-demand allocation of resources strategy. With Spark being a distributed engine, "latency is zero" becomes a fallacy. However, programming languages supported by Spark can eliminate this shortcoming (for example, Scala's *futures* provides ways to asynchronous method invocations). In MPI, it is still possible to hide latency through its APIs but the onus falls on the developer.
- *Fault tolerance*: Charm++ check-pointing capabilities and Spark's resiliency to failures

Characteristics	**MPI**	**Charm++**	**Apache Spark**
What is it?	Low-level parallel library	Abstracted system over MPI	Distributed data processing engine
Programming model	Processes/Threads	Parallel objects (chares)	Resilient Distributed Datasets (RDDs)
Execution model	Processor-to-processor messaging via APIs	chare-to-chare messaging through entry method invocation	Transformations and Actions applied on distributed data by executors
Communication overhead	High (even though asynchronous messaging is possible)	Low (all method invocations are asynchronous). Often overlap with computation.	Low (through just-in-time transformations)
Latency tolerance	Low (due to explicit message-passing)	High (on-demand allocation of resources to message handlers)	"Zero-latency" fallacy. Latency can be hidden through language-specific features (such as futures in Scala)
Fault tolerance	Low (explicitly controlled by developer)	Medium (checkpointing APIs)	High (data sets are resilient)
Level of abstraction	Low (almost everything is explicitly specified by the developer)	Medium (abstracts actual messaging overhead but developers can still control flows)	High (control only through provided APIs)
Applications	Traditional option for parallel applications	Has overridden MPI recently	MapReduce models, data-dependent iterative algorithms
Scale of data	Low to medium (not readily scalable)	High (justified by applications involving extremely large scale)	High (primary focus is on big-data applications)

Table 8.1: Comparison of parallel/distributed programming systems

through RDDs can significantly benefit applications with respect to failure recovery. With MPI, it becomes the responsibility of developers to support fault-tolerance in applications.

- *Abstraction levels*: APIs provided by Charm++ and Spark provide high levels of abstractions hiding several complications in communication, control over resources, etc., from the application level. This becomes an important quality of parallel systems that makes them user-friendly and highly maintainable code bases.
- *Scale of data*: Lastly, scale of data that Charm++ and Spark applications can handle are much larger compared to pure MPI-based systems. Systems such as Spark primarily focus of extremely large amounts of data by providing abstractions of memory, execution flows, etc.

8.2 Considerations of Graph Processing Engines for Epidemic Simulations

As seen earlier, *Charm++* and *Apache Spark* are parallel/distributed programming systems that are becoming favorable for large-scale parallel applications. Though the systems focus on specific classes of applications, use unique programming and execution models, provide other distinct capabilities (such as fault-tolerance, memory abstractions, etc.), they serve a similar purpose, i.e., developing large-scale parallel and/or distributed applications.

With respect to epidemic simulations, we have seen that Charm++ has already set its mark with an implementation of *EpiSimdemics*. This particular application has taken epidemic simulations to unprecedented levels in terms of amounts of data. Charm-EpiSimdemics claims to have capabilities of processing contact networks representative of the entire US population. The key to using Charm++ for epidemic simulations is to comply with its programming and execution models. Simulations in which we can come up with mappings of *parallel objects* (chares) to *entities* in the simulation model, and, represent *communication* among entities as *entry-method invocations* naturally become suitable for Charm++. Charm-EpiSimdemics and Charm-EpiFast are examples of applications that were able to provide such a mapping.

On the other hand, we have Apache Spark. We believe that ours is one of the preliminary attempts of using Spark for epidemic simulations. We considered a simulation tool called *EpiFast*, provided details of our experience in developing *Spark-EpiFast* and showed how we can realize performance benefits from using Spark. Unlike Charm++, the key here is to be able to represent data as *distributed collections* (RDDs) and defining computations as *transformations* on RDDs. Spark-EpiFast provided such a representation for EpiFast. We believe that our efforts has opened avenues for using Spark for epidemic simulations such as EpiSims [8] or EpiSimdemics [9]. For example, a conceptualization of EpiSimdemics using

Spark would be to represent the people-location graph as a combination of two RDDs: one as a collection of individuals, and the other for locations. We can translate computations (such as calculation of probabilities for transmission) to transformations in Spark (similar to Spark-EpiFast). Though it might not be trivial for all epidemic simulation algorithms to determine an ideal representation of transformations equivalent to Spark, we believe that those that do can benefit. Further studies in this direction could help us answer these questions and we believe Spark-EpiFast is one step in that direction.

Apache Giraph [56] is a graph processing engine focused on iterative graph processing with an implementation of Google's Pregel [63] system It is based on the Bulk Synchronous Parallel computation model [64]. Giraph is scalable to trillions of edges and is efficient in terms of performance. With support for message passing, topology mutation and aggregators [63] through C++ APIs, Giraph is also a potential framework for general graph processing. Given that we have a MapReduce-like model for EpiFast through this thesis, it could be worthwhile to explore Giraph and compare against our Spark-based version. *GraphLab* [57] and *Neo4j* [65] are other popular graph processing tools that can be explored to identify a MapReduce-based framework for large-scale epidemic simulations. A comparison of some of these graph processing systems for various classes of applications is provided in [66]. We believe that these systems, like Apache Spark, could be suitable for epidemic simulations. Although each system has a unique programming and execution model, we believe that exploring them further can help us identify a suitable system through interpretations similar to ours, of transmissions in EpiFast as basic map-reduce transformation operations.

Chapter 9

Conclusion

Computational epidemiology is fast becoming an important area of research among epidemiologists. The increasing scale of underlying social contact networks poses continuous challenges to researchers forcing them to adopt parallel or distributed strategies for computations. We have looked into several previous efforts from the literature related to parallel epidemic simulations. This thesis focuses one particular simulation algorithm called *EpiFast*. Here, I have presented two novel implementations of EpiFast: (i) ***Charm-EpiFast*** based on the *Charm++* parallel programming framework and (ii) ***Spark-EpiFast*** based on the *Apache Spark* distributed processing framework. I have presented the detailed architecture of both versions highlighting key optimization techniques. With extensive experiments, I have analysed the behavior of both versions, identified combination of framework-specific parameters befitting EpiFast and helped realize performance benefits claimed by the respective programming systems. I believe this thesis has helped identify Spark as a suitable engine for iterative epidemic simulations and that other similar algorithms can leverage the distributed processing capabilities of Spark.

Bibliography

[1] S. Eubank, H. Guclu, V. S. A. Kumar, M. V. Marathe, T. Z. A. Srinivasan, and N. Wang, "Modelling disease outbreaks in realistic urban social networks," *Nature, 429(6988)*, pp. 180–184, May, 2004.

[2] R. M. Anderson and R. M. May, "Population biology of infectious diseases," *Part I. Nature 280*, p. 361–367, August, 1979.

[3] W. O. Kermack and A. G. McKendrick, "A contribution to the mathematical theory of epidemics," *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, pp. 115(772): 700–721, August, 1927.

[4] J. D. Murray, *Mathematical Biology I. An Introduction*, vol. volume 19 of Biomathematics, 3rd edition. Springer Verlag, 2002.

[5] L. A. Meyers, "Contact network epidemiology: Bond percolation applied to infectious disease prediction and control," *Bulletin of The American Mathematical Society*, p. 44:63–86, 2007.

[6] A. Barrat, M. Barthelemy, and A. Vespignani, "Dynamical processes in complex networks," *Cambridge University Press*, 2008.

[7] N. Ferguson, D. Cummings, C. F. J. Cajka, P. Cooley, and D. Burke, "Strategies for mitigating an influenza pandemic," *Nature, 2006*, p. 442:448–452.

[8] S. Eubank, "Scalable, efficient epidemiological simulation," *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing, New York, NY, USA*, pp. 139–145, ACM, 2002.

[9] C. L. Barrett, K. R. Bisset, S. Eubank, X. Feng, and M. V. Marathe, "EpiSimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks," *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*, p. 37, 2008.

[10] K. R. Bisset, J. Chen, X. Feng, V. S. A. Kumar, and M. V. Marathe, "EpiFast: A fast algorithm for large scale realistic epidemic simulations on distributed memory systems," *International Conference on Supercomputing - ICS*, pp. 430–439, 2009.

[11] L. A. Meyers, M. E. J. Newman, and B. Pourbohloul, "Predicting epidemics on directed contact networks.," *Journal of Theoretical Biology*, p. 240(3):400–418, 2006.

[12] L. A. Meyers, B. Pourbohloul, M. E. J. Newman, D. M. Skowronskic, and R. C. Brunham, "Network theory and sars: predicting outbreak diversity.," *Journal of Theoretical Biology*, pp. 232(1):71–81, 2005.

[13] "FluTE, an influenza epidemic simulation model." http://www.cs.unm.edu/~dlchao/flute/. Accessed in May, 2015.

[14] M. Lysenko and R. M. DŚouza, "A framework for megascale agent based model simulations on graphics processing units," *Journal of Artificial Societies and Social Simulation*, pp. vol. Vol.11, no. No.4 10, October 2008.

[15] B. Aaby, K. Perumalla, and S. Seal, "Efficient simulation of agent-based models on multi-GPU and multi-core clusters," *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, 2010.

[16] N. Ferguson, *Strategies for mitigating an influenza pandemic.* Nature, 2006.

[17] N. Ferguson, D. Cummings, S. Cauchemez, C. Fraser, S. Riley, A. Meeyai, S. Iamsirithaworn, and D. Burke., "Strategies for containing an emerging influenza pandemic in southeast asia," p. 437:209–214, 2005.

[18] J. Parker, "A flexible, large-scale, distributed agent based epidemic model," *In Winter Simulation Conference*, 2007.

[19] I. M. Longini, A. Nizam, S. Xu, K. Ungchusak, W. Hanshaoworakul, D. A. T. Cummings, and M. E. Halloran, "Containing pandemic influenza at the source," *Science, 309(5737)*, pp. 1083–1087, 2005.

[20] "UPC consortium. UPC language. language specifications 1.2, the high performance computing laboratory, george washington university." http://upc.gwu.edu/. Accessed in May, 2015.

[21] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications, vol. Vol.20, no. No.2*, pp. 203–231, 2006.

[22] "Multi-agent transport simulation - MATSim." http://www.matsim.org/. Accessed in May, 2015.

[23] B. Logan and G. Theodoropoulos, "The distributed simulation of multiagent systems," *Proceedings of the IEEE, vol. Vol.89, no. No.2*, pp. 174–186, January, 2001.

[24] L. Gasser and K. Kakugawa, "MACE3J: fast flexible distributed simulation of large, large-grain multi-agent systems," *Proceedings of the first international joint conference on Autonomous agents and multiagent systems - AAMAS-2002. Bologna, Italy: ACM*, November, 2001.

[25] "Repast HPC tutorial." http://repast.sourceforge.net/hpctutorial/toc.html. Accessed in May, 2015.

[26] C. L. Barrett, S. Eubank, and M. Marathe, "An interaction based approach to computational epidemics," *AAAI'08: Proceedings of the Annual Conference of AAAI, Chicago USA. AAAI Press*, 2008.

[27] T. Chuang and M. Fukuda, "A parallel multi-agent spatial simulation environment for cluster systems," *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference*, pp. 143–150, 2013.

[28] M. Fukuda, "Mass: Parallel-computing library for multi-agent spatial simulation," *Distributed Systems Laboratory, Computing & Software Systems, University of Washington Bothell, Bothell, WA*, May, 2010.

[29] K. Perumalla, "μsik: A micro-kernel for parallel/distributed simulation systems," *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pp. 185–192, 2005.

[30] K. Perumalla and S. Seal, "Reversible parallel discrete- event execution of large-scale epidemic outbreak models," *Proceedings of the 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2010.

[31] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White, "Behavioral simulations in mapreduce," *Proceedings of the VLDB Endowment*, pp. , v.3 n.1–2, September 2010.

[32] K. R. Bisset, J. Chen, S. Deodhar, X. Feng, Y. Ma, and M. V. Marathe, *Indemics: An Interactive High-Performance Computing Framework for Data Intensive Epidemic Modeling.* ACM Trans Model Computational Simulations, January, 2014.

[33] "Charm++." http://charm.cs.uiuc.edu/. Accessed in May, 2015.

[34] J. Phillips, K. Schulten, A. Bhatele, C. Mei, Y. Sun, E. Bohm, and L. Kale, *Parallel Science and Engineering Applications: The Charm++ Approach, Scalable Molecular Dynamics with NAMD.* CRC Press, 2013.

[35] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with changa," *Proceedings of IEEE International Parallel and Distributed Processing Symposium.*

[36] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kale, and T. R. Quinn, "Scaling hierarchical n-body simulations on GPU clusters," *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, Washington, DC, USA, 2010*, 2010.

[37] A. Langer, R. Venkataraman, U. Palekar, and L. V. Kale, "In search of a scalable, parallel branch-and-bound for two-stage stochastic integer optimization," *High Performance Computing (HiPC), 2013 20th International Conference*, pp. 266–275, December, 2013.

[38] K. Bisset, A. Aji, T. Kamal, J.-S. Yeom, M. Marathe, E. Bohm, and A. Gupta, *Parallel Science and Engineering Applications: The Charm++ Approach, Contagion Diffusion with EpiSimdemics*. CRC Press, 2013.

[39] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. Quinn, and L. Kale, "Tram: Optimizing fine-grained communication with topological routing and aggregation of messages," *ICPP*, 2014.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10, 2010.

[41] D. Morales and O. Mendez, "Stratio streaming: a new approach to spark streaming." http://spark-summit.org/2014/talk/stratio-streaming-a-new-approach-to-spark-streaming. Accessed in May, 2015.

[42] S. Goyal, "Fuzzy matching with spark." http://spark-summit.org/2014/talk/fuzzy-matching-with-spark. Accessed in May, 2015.

[43] A. Kini and R. Emanuele, "Geotrellis: Adding geospatial capabilities to spark." http://spark-summit.org/2014/talk/geotrellis-adding-geospatial-capabilities-to-spark. Accessed in May, 2015.

[44] K. Mader, "Scaling up fast: Real-time image processing and analytics using spark." http://spark-summit.org/2014/talk/scaling-up-fast-real-time-image-processing-and-analytics-using-spark. Accessed in May, 2015.

[45] S. Chung, "Sequoia forest : Random forest of humongous trees." http://spark-summit.org/2014/talk/sequoia-forest-random-forest-of-humongous-trees. Accessed in May, 2015.

[46] D. Das and S. Das, "Quadratic programing solver for non-negative matrix factorization with spark." http://spark-summit.org/2014/talk/quadratic-programing-solver-for-non-negative-matrix-factorization-with-spark. Accessed in May, 2015.

[47] U. B. AMPLab, "Scalable genomes clustering with adam and spark." http://bdgenomics.org/. Accessed in May, 2015.

[48] C. Barrett, H. H. III, M. Marathe, S. Ravi, D. Rosenkrantz, and R. Stearns, "Modeling and analyzing social network dynamics using stochastic discrete graphical dynamical systems," *Journal in Theoretical Computer Science*, pp. vol. 412, 3932–3946, July, 2011.

[49] H. Mortveit, D. Murrugarra, C. Kuhlman, and V. A. Kumar, "Bifurcations in boolean networks," *Discrete Mathematics and Theoretical Computer Science, Santiago, Chile*, pp. 29–46, 2011.

[50] C. Kuhlman, V. A. Kumar, M. Marathe, H. Mortveit, S. Swarup, S. R. G. Tuli, and D. Rosenkrantz, "A general-purpose graph dynamical system modeling framework," *Proceedings of the 2011 Winter Simulation Conference. Phoenix, Arizona*, December 11-14, 2011.

[51] C. Barrett, H. H. III, M. Marathe, S. Ravi, D. Rosenkrantz, and R. Stearns, "Complexity of reachability problems for finite discrete dynamical systems," *Proceedings of International Symposium on Mathematical Foundations of Computer Science, MFCS*, pp. 159–172, 2004.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NDI*, pp. 2–2, 2012.

[53] "Spark programming guide: Transformations." http://spark.apache.org/docs/1.2.1/programming-guide.html#transformations. Accessed in May, 2015.

[54] "Spark programming guide: Actions." http://spark.apache.org/docs/1.2.1/programming-guide.html#actions. Accessed in May, 2015.

[55] "Spark programming guide: Rdd persistence." http://spark.apache.org/docs/1.2.1/programming-guide.html#transformations. Accessed in May, 2015.

[56] "Apache giraph." http://giraph.apache.org/. Accessed in May, 2015.

[57] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, pp. vol. 5, No. 8, 716–727, 2012.

[58] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: a resilient distributed graph system on spark," *GRADES'13 First International Workshop on Graph Data Management Experiences and Systems*, p. Article No. 2.

[59] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," *GRADES'13 First International Workshop on Graph Data Management Experiences and Systems*.

[60] "Graph operators: Graphx." http://spark.apache.org/docs/1.2.1/graphx-programming-guide.html#summary-list-of-operators. Accessed in May, 2015.

[61] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman, "Upper and lower bounds on the cost of a map-reduce computation," *Proceedings of the VLDB Endowment*, pp. vol. 6, 277–288, 2013.

[62] R. Kaw, "Modeling and computation of complex interventions to control disease outbreak in large-scale epidemiological simulations using sql and distributed database," *Master's Thesis, Virginia Tech*, July, 2014.

[63] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Nwe York, USA*, pp. 135–146, 2010.

[64] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, pp. vol. 33, no. 8, August, 1990.

[65] "Neo4j: The fastest and most scalable native graph database." http://neo4j.com/. Accessed in May, 2015.

[66] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, pp. vol. 7, No. 12, September, 2014.