A High Performance C++ Benchmark for Computational Epidemiology

Aniket Pugaonkar

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
Sandeep Gupta
Keith R. Bisset
Calvin J. Ribbens

December 10, 2014
Blacksburg, Virginia

Keywords: Benchmark, Epidemiology, EpiSimdemics, EpiFast,
C++11, BOOST C++ libraries, Intel® TBB, Intel® Cilk Plus, Intel® MIC
Copyright 2014, Aniket Pugaonkar

A High Performance C++ Benchmark for Computational Epidemiology

Aniket Pugaonkar

ABSTRACT

An effective tool used by planners and policy makers in public health, such as Center for Disease Control (CDC), to curtail spread of infectious diseases over a given population is contagion diffusion simulations. These simulations model the relevant characteristics of the population (age, gender, income etc.) and the disease (attack rate, etc.) and compute the spread under various configuration and plausible intervention strategies (such as vaccinations, school closure, etc.). Hence, the model and the computation form a complex agent based system and are highly compute and resource intensive.

In this work, we design a benchmark consisting of several kernels which capture the essential compute, communication, and data access patterns for such applications. For each kernel, the benchmark provides different evaluation strategies. The goal is to (a) derive alternative implementations for computing the contagion by combining different implementation of the kernels, and (b) evaluate which combination of implementation, runtime, and hardware is most effective in running large scale contagion diffusion simulations. Our proposed benchmark is designed using C++ generic programming primitives and lifting sequential strategies for parallel computations. Together, these lead to a succinct description of the benchmark and significant code reuse when deriving strategies for new hardware. For the benchmark to be effective, this aspect is crucial, because the potential combination of hardware and runtime are growing rapidly thereby making infeasible to write optimized strategy for the complete contagion diffusion from ground up for each compute system.

This work has been partially supported by DTRA CNIMS, DTRA V&V, NSF NetSE with grant numbers HDTRA1-11-D-0016-0001, HDTRA1-11-1-0016, CNS-1011769 respectively. We thank our external collaborators and members of the Network Dynamics and Simulation Science Laboratory (NDSSL) for their suggestions and comments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DTRA, NSF or U.S. Government.

Dedication

Dedicated To My Parents

Acknowledgments

First and foremost, I express my deepest gratitude to my Chair and Committee members for their guidance, motivation and support. I sincerely thank Dr. Sandeep Gupta for giving me the opportunity for this work that is a first step towards this research. Dr. Gupta has been very inspirational and patient and I am very much obliged for his assistance throughout this work. His constant support and guidance during the implementation of the benchmark has helped boost my coding skills and knowledge. I look forward to contribute further to the best of my capabilities. I am highly thankful to Dr. Keith Bisset for his contributions; his suggestions and ideas have been very fruitful towards shaping this work.

Besides Dr. Gupta and Dr. Bisset, I thank Dr. Madhav Marathe for giving me the opportunity to work at NDSSL and also complete my Master's Thesis out of this work. It has been an extraordinary experience, professionally as well as personally and I will always cherish the time spent here at NDSSL and Virginia Tech. I am also thankful to Dr. Cal Ribbens for being on my committee and providing valuable inputs on my work.

Finally, I express my deepest gratitude to my family and friends for their unconditional love, support and encouragement. My parents, sisters and brothers-in-law were always supportive and inspired me to work towards my goals.

Contents

List of Figures **xi**

List of Tables **xiv**

1 Introduction **1**

2 Related Work, Motivation and Contributions **4**

2.1 Related Work 4

2.2 Motivation 5

2.2.1 Challenges 6

2.2.2 Goals 6

2.3 Computational Dwarfs 6

2.4 Contributions 7

3 Preliminaries **9**

3.1 Strategy 9

3.2 Traits 10

3.2.1 Iterator Traits - An example 10

3.2.2 Kernel Traits 11

3.3 Notion of Concept 11

3.4 Benchmark Concepts 12
3.4.1 Alphabet 13
3.4.2 Collection 14
3.4.3 Iterator 14
3.4.4 Maps or Property Maps 15
3.5 Graph 16
3.5.1 Graph Elements 17
3.5.2 Graph Iterators 18
3.6 Contagion Model 20
3.6.1 Motivation 20
3.6.2 Requirements 21
3.6.3 Mathematics behind Disease Model 21
3.7 Functors – Background 23
3.8 Intervention 24
3.8.1 Motivation and Background 24
3.8.2 Intervention Functor 24
3.8.3 Intervention Requirements 25
3.9 Contagion Functor 27
3.10 Algorithms 28
3.10.1 Algorithm Traits 29
3.11 Concepts Combined Together 29
4 Benchmark Specifications **31**
4.1 Epidemiology Specifications 31
4.1.1 Synopsis 31
4.1.2 Kernel 0 : Activity List Generator 32

4.1.3 Kernel 1 : Person-Location Graph Generator 32
4.1.4 Kernel 2 : Person-Person Social Network Generator 32
4.1.5 Kernel 3 : Hierarchical Person-Location Graph 33
4.1.6 Kernel 4 : Activity based Contagion Simulator 33
4.1.7 Kernel 5 : Contact based Contagion Simulator 33
4.2 Concepts based Specifications 33
4.2.1 Synopsis 34
4.2.2 Kernel 0 34
4.2.3 Kernel 1 35
4.2.4 Kernel 2 36
4.2.5 Kernel 3 37
4.2.6 Kernel 4 40
4.2.7 Kernel 5 41
4.3 Computational Expressions 43
4.3.1 Kernel 0 43
4.3.2 Kernel 1 44
4.3.3 Kernel 2 45
4.3.4 Kernel 3 46
4.3.5 Disease Model 46
4.3.6 Update Functor 46
4.3.7 Intervention Functor - Vertex 47
4.3.8 Intervention Functor - Edge 47
4.3.9 Contagion Functor - Location based 47
4.3.10 Contagion Functor - Contact based 48
4.3.11 Kernel 4 49

4.3.12 Kernel 5 50

5 Lifting Sequential Algorithms for Shared and Distributed Memory Machines 51

5.1 Lifting 51

5.2 Lifting for Shared Memory Parallel Computing 52

5.2.1 Generalizing the concept 55

5.2.2 Performance Implications of Mapping Threads 56

5.3 Ensuring Correctness for Parallel Lifting 58

5.4 Lifting for Distributed Memory 59

6 Implementation 61

6.1 C++11 Features 61

6.1.1 auto 61

6.1.2 decltype 62

6.1.3 Range-based For loops 62

6.1.4 Function Objects a.k.a Functors 63

6.1.5 Lambdas a.k.a Anonymous Functors 64

6.1.6 Other Features 65

6.2 Boost Graph Library 66

6.3 Containers 66

6.4 Compilers - GCC and ICPC 67

6.5 Intel TBB 67

6.6 Intel Cilk Plus 69

6.6.1 STL Container Support 70

6.6.2 Intel MIC and Cilk 72

6.7 Intel MIC 72

7 Performance Metrics and Analysis **73**
7.1 TIPS 73
7.1.1 Total Interactions Per Second 73
7.2 Experiment Setup 75
7.2.1 Data Generation and Other Parameters 75
7.2.2 Effect of Input Parameters 75
7.3 Preliminary Results 76
7.3.1 Kernel 4 implementations 76
7.3.2 Run Statistics 78
7.3.3 Epi-curves for Kernel 4 and Kernel 5 79
7.3.4 Cilk vs. TBB for Kernels 4 and 5 81
7.4 Strong Scaling 82
7.4.1 Relative Performance of Simulators 1, 2, 3 83
7.4.2 Scalability Results for Kernel 4: Strong Scalability 84
7.5 Weak Scaling 88
7.5.1 Scalability Results for Kernels 4 and 5: Weak Scalability 88
7.6 Intel MIC Performance for Kernel 4 92

8 Conclusions **94**

Bibliography **95**

Appendices **101**

A Intel Compiler **102**
A.1 Compiling for Intel MIC 102

B Practical Challenges **104**

B.1 Type compatibility 104
B.2 Type Deduction 104

List of Figures

3.1 (i) **SEIR** Model, (ii) a four node and five edge contact network with $\Delta t_E = 0$, $\Delta t_I = 2$ (in days) for each node and transmission probability 0.5 for each day. Node A is infectious at start. (i) **Day 0**: A transmits the disease to B but not D. (ii) **Day 1**: both A and B are infectious. A transmits the disease to D; B infects C but not D. (iii) **Day 2**: A is removed, all others are infectious with no susceptible nodes. (iv) Nodes are removed gradually and on **day 4**, the system enters a fixed point and stops evolving. The state transitions are one-way (from susceptible to exposed to infected to recovered) with no other possible transitions. 22

3.2 Node Intervention: Alter Vertex Properties of Persons (nodes). Note that a change in probability from 1 to 0.5 in both cases (infected or susceptible) will reduce the probabilities of infecting others or getting infected 26

3.3 Edge Intervention: Alter Edge Properties (Activities or Contact period). Location L_1 closure results in redirection of P_1's and P_2's edges to location L_2. 27

4.1 Location Hierarchy – City, Blocks, Location Groups, Locations 39

5.1 Lifting a concrete algorithm to generic algorithm: (i) enlist the trivial or concrete algorithm, (ii) remove the unnecessary requirement on *type*, (iii) after iterations, make the algorithm work for a maximal family of types. 52

5.2 Thread Pool to Location Group mapping. Every thread T_i is assigned a location group LG_i 53

5.3 Shared Memory: Location Groups (LG_i) are mapped to tasks T_i to form a Task Pool. A Worker (W_i) (i.e., a thread) is assigned a task T_i from the pool (blue arrows). A worker can further create additional tasks T_{ij} (by mapping Locations L_i onto the task pool (read arrows)). These tasks are also executed by workers. . . 54

5.4 Shared Memory: The Thread Pool (T_i) to Infected Group (P_i) mapping. Every thread handles one infected person (task) and computes contagion for those people in contact with that person. 55

5.5 Location Hierarchy – City, Blocks, Location Groups, Locations 56

5.6 Distributed Memory: A Worker (MPI) (W_i) process is created on every compute node and is assigned a Location Group (LG_i) (typically a sub-graph of main graph) for the task. The MPI Process spawns a local Thread Pool (T_i) (per compute node) where each thread handles a local task as assigned. 60

7.1 Input Parameters in the Experiments 77

7.2 Output Parameters in the Experiments 78

7.3 Epi-curves for Kernels 4 and 5 for varying graph sizes 80

7.4 Cilk vs. TBB for Kernels 4 and 5. Plot (b) shows the results for Kernel 4 running Sim 2 algorithm on the graph size 2^{18} (on SFX). (c) shows the results for Kernel 5 the graph size 2^{20} (on BlueRidge). 81

7.5 Relative performance of Sim 1, Sim 2, Sim 3 implementations for kernel 4 83

7.6 Strong scalability of Kernel 4 for implementation Sim 1, with varying graph sizes and on three different architectures - SFX, BlueRudge and Hokieone. Plot (a) shows the strong scalability results for graph size 2^{15}. Plot (b) shows the strong scalability results for graph size 2^{16}. Plot (c) shows the strong scalability results for graph size 2^{17}. Plot (d) shows the strong scalability results for graph size 2^{18}. . 85

7.7 Strong scalability of Kernel 4 for implementation Sim 2, with varying graph sizes and on three different architectures - SFX, BlueRudge and Hokieone. Plot (a) shows the strong scalability results for graph size 2^{15}. Plot (b) shows the strong scalability results for graph size 2^{16}. Plot (c) shows the strong scalability results for graph size 2^{17}. Plot (d) shows the strong scalability results for graph size 2^{18}. . 86

7.8 Strong scalability of Kernel 4 for implementation Sim 3, with varying graph sizes and on three different architectures - SFX, BlueRudge and Hokieone. Plot (a) shows the strong scalability results for graph size 2^{15}. Plot (b) shows the strong scalability results for graph size 2^{16}. Plot (c) shows the strong scalability results for graph size 2^{17}. Plot (d) shows the strong scalability results for graph size 2^{18}. . 87

7.9 Weak scalability of Kernel 4 for the three implementations, each with varying graph sizes and on three different architectures - SFX, BlueRidge and Hokieone. Plot (a) shows the weak scalability results for implementation 1. Plot (b) shows the weak scalability results for implementation 2. Plot (c) shows the weak scalability results for implementation 3. 90

7.10 Weak scalability of Kernel 5 with varying graph sizes and on two different architectures - SFX and BlueRidge. 91

7.11 Strong scaling of Kernel 4 for the three implementations on Intel MIC card in BlueRidge System. Plot (a) shows the relative performance of TBB and Cilk implementations. Plot (b) shows comparison for `cilk_for` and `cilk_sync` implementations. 92

List of Tables

7.1 Total Interactions for each Implementation 78

7.2 Total Contagions during 100 day simulation period (70% infection rate) 79

Chapter 1

Introduction

Epidemiology is a discipline of public health science that focuses on studying the space-time patterns, causes, and effects of disease conditions in a demographic region. It plays a crucial role in understanding the processes that lead to an outbreak of disease in a population as well as evaluating strategies designed to prevent or control the outbreak. Networked Epidemiology is the study of spread of any contagion over a social contact network. The contagion may represent an actual infectious disease, or it may represent a more general reaction-diffusion process, such as the diffusion of innovation or any viral advertisement. The populations of interest depend on the contagion and other factors. Similarly, the interactions represented must depend on the disease and the populations, including physical proximity, day-to-day visits, duration of visits, etc. Inherently, Networked Epidemiology is computational in nature and using this concept, we study three basic components of computational epidemiology: (i) individual behaviors of agents, (ii) generation of large multi-scale networks, and (iii) dynamical processes on these networks. In the last couple of decades, epidemiologists have increasingly turned to models and computer simulations to help solve some of these challenges. Specifically, they deploy large machines to understand and control spatio-temporal diffusion of diseases in populations. The results of these simulations are used in implementing public policies for disease containment. Such simulations are compute and resource intensive due to large volumes of raw data at hand from which meaningful results can be obtained.

Interactions amongst the individuals in a population is one of the factors for between-host disease propagation. Over the last few years, interaction based modeling approach has become increasingly important in developing public health policies [24–26]. Interaction based computational epidemiology, thus attempts to develop detailed representation of the underlying social contact networks, within and between host disease progression and transmission models, and

computational representations of implementable interventions. These models are then composed accordingly to create detailed computer simulations to guide and inform public health authorities to draft appropriate measures [1, 7].

Our goal is to create bench work for "interaction based computing" by developing a generic benchmark for such interaction based contagion reaction-diffusion network models that capture the essential data access patterns and computational complexities (and not the semantics) of widely used simulation algorithms in computational epidemiology. Our work lies on intersection of high performance computing and its application in developing tools that simulate contagion over a dynamic social network. The need to benchmark such tools is therefore very important to (1) choose the most appropriate strategy (a choice of implementation, runtime and hardware) for a given set of parameters and (2) select or narrow down the procurement of compute systems, which, overall perform well for a wide range of epidemiological problems.

Benchmarking results generally yield comparative results indicating that some class of machines are better than others. The results usually capture typical machine behavior for a wide class of algorithms. An ideal benchmark must therefore be fairly simple to understand, easy to run, map to real world problems (in addition to spanning common algorithms), and most importantly, must be correct. By far, the High Performance Linpack (HPL) or Top 500 benchmark [17] is the most widely recognized and discussed metric for ranking high performance computing system. However, with the new trends in hardware, it is not merely everything about computational power but also about the interconnect and data access patterns of the underlying application [12]. Hence, there arises a need for application specific benchmarking in order to capture their essential compute, communication and data access patterns. This is particularly a challenging process as one needs to select an appropriate benchmark metric that addresses the problem at hand (which is to analyze and report the behavior of the tool to be benchmarked) and also ensure its extendibility and applicability to other problems in that domain.

Interaction based simulations provide a realistic insight into disease spreading mechanisms which are modeled on social networks and daily activities of people in the network. These models rely on high performance computing systems to provide fast and accurate results for making quick and sensible decisions during an outbreak. Current large scale simulation for contagion diffusion exists for Blue Waters machine using Charm++ runtime [22]. However, modeling dynamics and irregular structures is a challenging problem and is a nascent topic of research. Social networks are modeled as co-evolving discrete dynamical systems that involve repeating computations after every event/phase. Study of Graph dynamical systems is therefore necessary as they involve repetitive computationally intensive tasks. Very few groups have scaled this to networks of our

size. Our benchmark attempts to address this problem and thus is aimed to capture the dynamics and irregularities in social networks and to the best of our knowledge, we are not aware of any benchmark that is specifically designed and dedicated to serve this purpose. We propose our benchmark to be designed on these principles with only one exception - our benchmark is application specific to the domain of computational epidemiology. In this report, we propose an application specific generic C++ benchmark to evaluate high performance computing systems using a suite of six generic kernels with different implementation strategies and determine which combination of implementation, runtime, and hardware is most effective in simulating contagion over large social contact networks.

The rest of the report is organized as follows. Chapter 2 is dedicated to related work, motivation and our contributions. We discuss about the similarities and differences in our proposed benchmark with respect to the Graph 500 benchmark [17]. The preliminaries and concepts required for the benchmark are briefly described in chapter 3. It also provides specifications for interventions and the ability of our benchmark to model epidemiological interventions that are applied in simulations. Such capabilities allows the benchmark to model different interventions so as make it realistic and capture the behavior of epidemiological applications correctly. In chapter 4, we develop specifications for our kernels. These specifications are first explained from epidemiology perspective and then transformed into technical specifications. We provide strategies for parallelization of our sequential benchmark for shared and distributed memory machines in chapter 5. The C++11 features and generic implementation details of our benchmark are discussed briefly in chapter 6 and the preliminary results based upon shared memory implementation of the benchmark using Intel TBB and Intel Cilk Plus are presented in chapter 7. We conclude this report with our interpretation of results and future work to enhance the benchmark in chapter 8.

Chapter 2

Related Work, Motivation and Contributions

2.1 Related Work

The Top 500 or High Performance Linpack (HPL) [14] is the most widely recognized and discussed metric for high performance computing systems. Apart from HPL, other important benchmarks such as NPB [18], HPCC [15], SPEC [19], EuroBen [20], Green 500 [16] are also widely used which examine and rate the performance of existing as well as emerging architectures using kernels which capture various aspects of well known computational problems viz. algorithms (barring the semantics), memory access patterns, data transfer (interconnect), latency, load balancing capabilities, etc. The implementation of such benchmarks exist in most of the performance affine programming languages and environments with a wide array of optimization techniques utilizing close to 100% CPU time and reducing memory accesses to bare minimum.

However, such stress testing methods for leadership class high performance systems are not strongly correlated to real application performance [12] because of which the authors envisioned and proposed a new High Performance Conjugate Gradient (HPCG) benchmark as an apt replacement to HPL. Many important applications have lower computation-to-data access ratios, irregular memory access patterns, fine grained recursive computations (such as distributed bread-first search). Traditional benchmarks by nature capture the type 1 computational patterns of real applications but somewhat fail to address the type 2 patterns. One approach to solve such a problem is to design an application specific benchmark which can truly capture all aspects of real applications - from computational patterns to data access, memory access, latency, interconnect

and load balancing.

2.2 Motivation

We compare the features of our benchmark with the Graph 500 benchmark and discuss the similarities and differences for the same. Our challenges and goals in developing such a benchmark and extracting the necessary specifications for it are also discussed here.

Graph 500 *vs.* **our Benchmark**

(a) **Similarities**: The impetus behind the Graph 500 benchmark and our benchmark is a set of kernels designed to measure the suitability of systems for data intensive applications such as informatics problems which have graphs as their main workloads. By graphs we mean any type of data structures which can be represented in form of graphs and their algorithms reference the graph structure. Another similarity is about the set of kernels provided. While the Graph 500 benchmark provides two kernels (construction and algorithm) and their reference implementations, our benchmark provides similar kernels for generating social networks and simulating contagion over them. The performance of systems which would run both these benchmarks is dominated by their ability to sustain large number of small and random data access patterns across their memory, interconnect and parallelism mechanism available. These factors largely creates differences between the machines which run Linpack benchmark and machines which would run our benchmark (and Graph 500) as our focus is more than the performance of a single instruction stream and the ability for fine-grained synchronization which is the case with Linpack.

(b) **Differences**: The main difference between the Graph 500 and our benchmark is that of the metrics on which the high performance systems are analyzed for superiority. While the former emphasizes on total number of traversed edges per second, our benchmark focuses on total number of interactions in the social network. An edge traversed is different from an interaction between two individuals because of the end results they produce. An interaction could lead to propagation of infection in a network and hence decide the overall performance of the benchmark while traversing edges would only yield the metric about how fast the graph can be processed for a particular problem. Our benchmark kernels depend on the input parameters and the results can vary substantially.

While the Graph 500 kernels are developed for computational domains such as cybersecurity, medical informatics, data enrichment, social networks, and symbolic networks, our benchmark is confined to social networks and interactions within such networks. The interactions could be anything such as disease propagation, viral ad propagation etc.

2.2.1 Challenges

(a) Complex algorithms used in real applications cannot be used as benchmarks because of intricate application parameters.

(b) Designing and implementing strategies for new hardware limits code reuse and generic programming.

2.2.2 Goals

(a) Design kernels which capture essential computation, communication and data access patterns in tools used to simulate spread of infectious disease through contagion models.

(b) Develop and implement different evaluation strategies for kernels for existing and emerging hardware.

(c) Evaluate the most effective combination of implementation, runtime and hardware for a given contagion.

2.3 Computational Dwarfs

Due to proliferation of micro architectures (Haswell, Ivybridge, Sandybridge, IBM Power and Cell, etc.), accelerators (Xeon Phi, Tesla, Firepro), memory technologies (synchronous vs. asynchronous), interconnects(2D-, 3D-, and 5D-torus), and runtimes (OpenMP, MPI, Cilk, TBB, CnC) finding the right stack of technologies most suited for such engines is challenging. In addition to the right choice of hardware and runtime technology, the choice of appropriate types of data-structures, compute expressions, and associated algorithms is necessary. These aspects also play a key role in performance. To address this recent switch to parallel microprocessors and industry laid road map for multi-core designs that preserve programming paradigms, a new approach is required to best express parallel computations and hence there is a need for a higher

level of abstractions about parallel application requirements [53, 54]. This approach led to frame the parallel landscape with seven important question and 13 computational *dwarfs* or *motifs*. This work is famously recognized as Berkeley's Dwarfs in Scientific Computing [53]. A dwarf is an algorithmic method that captures a pattern of computation and communication that is common to a class of important applications. The idea is that, instead of evaluating these new and emerging platforms using traditional benchmarks and specific optimizations, enumerate the application requirements in a manner that is not specific to individual applications or hardware platforms and draw broader conclusions about hardware and runtime requirements. The seven important questions posed for 21st century parallel computing are: (i) What are the applications? (ii) What are their common kernels? (iii) How are they described? (iv) How to program the hardware? (v) What are the hardware building blocks? (vi) How to connect them? and (vii) How to measure success?

Our work is step towards answering some of the questions for Agent Based Contagion Models that are model of Graph Traversal, Graphical Models, and Finite State Machines in Berkeley's 13 dwarfs. For each of these models, we attempt to derive computational expressions for the kernels that define our Agent Based Simulation Applications. These kernels primarily capture the following aspects: graph construction for social networks, probabilistic and timed finite state machine for individuals and overall graph dynamical system including its state update pattern which is derived based upon the disease model and graph dynamical system framework, and the infection (or contagion) spread mechanism (characterized by a generalized reaction-diffusion process) that captures the computational and communication pattern for such simulation applications.

When we say the term "contagion diffusion process" or phrases related to it, we actually mean contagion **reaction-diffusion** process, since spreading of a disease in a social network triggers events or actions (reactions) as well as disease propagation (diffusion) in the the network. This phenomena is captured by a reaction-diffusion process and hence the name. However, we will use the former term for brevity.

2.4 Contributions

Our contributions are as follows: We develop specifications for our kernels and provide a set of metrics to evaluate a computational platform. Our implementation of kernels is based on C++ generic programming and *type* based principles to make it compatible with any graph library which implements such interface. The simulation kernels, which are comprised of input graph,

a contagion model, an intervention strategy along with fast state update mechanism is aimed to capture the computational complexities (and not the actual semantics) of the class of algorithms and models used for studying agent based contagion diffusion. Such kernels can easily be extended to work with other contagion models such as viral information propagation in a social network or even how stock prices affect the behavior (buy/sell/wait) of individuals and their related contacts. Our benchmark is there therefore limited to agent based contagion models but caters to different models in this research domain. In this thesis, we develop shared memory generic implementations of kernels using task based parallelism and propose a scalable distributed memory task based parallelism methodology using the message passing interface which is beyond the scope of this thesis.

Chapter 3

Preliminaries

3.1 Strategy

We broadly define what we mean by a strategy for our benchmark. A strategy is a choice of implementation, runtime and hardware we choose for our benchmark. Thus, a strategy $\mathcal{S}$, is a 3-tuple (ι, ρ, η) where ι is the implementation, ρ is the runtime, and η is the hardware used for studying performance. The hardware consists of the HPC system, its interconnect and other features. Runtime consists of compilers, programming languages, design patterns, dynamic libraries and other libraries required to develop our benchmark. Finally, an implementation is an instance of specialized kernels to be executed. For example, to simulate contagion over person-location contact network, we need kernels 0, 1, 3, and 4. By specialized we mean the kernels and their desired parameters are chosen by the user during compile time and then combined with runtime and hardware to provide a unified strategy. An implementation may have two types of interventions (Node and Edge based) or either of them or even have them turned off for test runs.

In this report, we develop specifications for our kernels and their input/output requirements and also provide sample implementations for the same. Our implementation gives users the flexibility to choose the *fsm-based* input disease model, data structure for representing contact network (graphs), intervention types, and algorithms which capture the computational complexities of contagion-diffusion simulation algorithms. These parameters complete our benchmark suite. We provide our C++ based implementation for the benchmark and discuss the details in chapter 6.

3.2 Traits

The term *traits* is a very important technique in generic programming, specially in C++. As our core benchmark structure is developed in C++ (in conjunction with C++11 standards), we would be using the *traits* methodology to make compile time decisions based on *types*, thereby deriving a new strategy based upon the *traits*-defined behavior of the benchmark. A *Traits* class adds an extra level of indirection on *types* and thus enable us to take compile-time decisions out of the immediate context where they are made. The advantage here is that we would get a cleaner, maintainable and extensible code. Also because it is a compile time decision based policy tool, we do not have to worry about performance as compared to the runtime decision based dispatching. Using traits, we are able to choose exact implementation strategy for a given compute and runtime. We will present a short example of a C++ `iterator_traits` class to explain the concept. Our benchmark uses the `kernel_traits` class to obtain the same compile-time behavior obtained for any other traits classes in other generic programs.

3.2.1 Iterator Traits - An example

In this example taken from [35] and also freely available in standard technical C++ resources, the class template `std::iterator_traits<T>` looks something like this:

```
1  template <class Iterator>
2  struct iterator_traits {
3  typedef typename Iterator::iterator_category iterator_category;
4  typedef typename Iterator::value_type        value_type;
5  typedef typename Iterator::difference_type   difference_type;
6  typedef typename Iterator::pointer           pointer;
7  typedef typename Iterator::reference         reference;
8  };
```

The `value_type` typedef in the traits' class gives the type which the iterator is "pointing at", while the `iterator_category` is used to select more efficient algorithms depending on the underlying iterator's capabilities. The `std::advance` function template is a very good example of choosing efficient implementation of advancing the iterator by n steps. If the iterator's `category` is of type *random-access iterator*, the function uses just once `operator+` or `operator-`. If not, the function uses the increase or decrease operator (`operator++` or `operator--`) repeatedly until n elements have been advanced.

Traits templates allow us to associate information with arbitrary types. We can also specify traits for a particular type using template partial specialization technique. An example supporting this argument is to specialize the `iterator_traits` for arbitrary (generic) pointer types (of type T*). This can be done as follows:

```
1  template <typename T>
2  struct iterator_traits<T*> {
3  typedef T                          value_type;
4  typedef ptrdiff_t                  difference_type;
5  typedef random_access_iterator_tag iterator_category;
6  typedef T*                         pointer;
7  typedef T&                         reference;
8  };
```

3.2.2 Kernel Traits

We develop our kernel traits class `kernel_traits` based on this technique. This class helps us select arbitrary types (alphabet generator, containers, graphs, algorithms) to choose a given implementation strategy (amongst several strategies). Think of this as a plug-n-play behavior, where a combination of given *types* of data structures yields an all together new implementation! Of course we must take into account the compatibility amongst different types, but that is left to implementation detail. Here we develop concepts essential for our benchmark.

3.3 Notion of Concept

The notion of *Concept* is fundamental in generic programming as *Concepts* bundle together coherent sets of requirements into a single entity. *Concepts* were introduced to express the syntactic and semantic behavior of *types* and also to constrain the type parameters in a C++ template. Concepts thus describe a family of related abstractions based on what those abstractions can do and hence can be defined as follows: A *concept* is the formalization of an abstraction as a set of requirements on a type (or a set of types, integers, operations, etc.) [33,34]. A *type* that implements the requirements of a concept is said to *model* the concept. Such requirements may be syntactic, semantic, or performance related. For instance, an Iterator concept would describe abstractions that iterate over a sequence of values (such as a pointer), a Socket concept would describe abstractions

that communicate data over a network (such as an IPv6 socket), and a Polygon concept would describe abstractions that are closed plane figures (such as triangles and octagons) [32].

Concepts are discovered through the process of lifting many algorithms within the same domain and the result of such lifting process yields a generic algorithm and a set of requirements. A concept however must have enough requirements so that it gives a common identity to the abstractions it describes. We can bundle related requirements to form a concept so as to simplify the expression of the requirements on algorithms and also assigning identity to abstractions. There are Nested Requirements and Associated types that further aid in reuse of prior concepts in definition of other concepts and allowing some types to be stored inside the concept definition. Concept Refinement helps understand and create hierarchical relationships between two concepts. A bidirectional iterator is a refined concept in the sense that it refines the Iterator concept and also adds its own set of requirements with it. [32, 35] provides a concise yet precise description about *concept* and its role in generic programming. Often, *Concepts* are documented as set of requirements consisting of valid expressions, associated types, invariants and complexity guarantees.

In this chapter, we present the *Concepts* crucial to build our generic benchmark and then specifications based upon it. These concepts form basic building blocks of the benchmark and any implementation satisfying these concepts forms the implementation component of strategy. The concepts are itemized here and described in the next section: (i) Alphabet, (ii) Collection, (iii) Iterator, (iv) Map or Property Map, (v) Graph, (vi) Disease Model or Contagion Model (vii) Intervention, and (viii) Algorithm. A concept may depend or can be built upon other concepts. For example, Collection concept consists of grouping together a set of similar objects derived from a common Alphabet string. More details are described in further sections.

3.4 Benchmark Concepts

We describe here the benchmark concepts which were summarized in the previous section. These concepts form the basic building blocks of the benchmark and any such implementation conforming to the requirements of the concepts would manifest in *plug-n-play* behavior of implementation and generate the corresponding strategy. This makes the benchmark generic for arbitrary types which model such concepts but is however application specific for contagion models. Further extensions may be developed by replacing the disease model with any other arbitrary model which conforms to the Contagion Model Concept (see 3.6). For example, we can change the input disease model to a stock-price model, where an information propagation (disease) leads to flow of information (event) and triggers the appropriate actions for stocks (short

selling, buying etc.). Another example could be viral information flow in social networks. An advertisement can go viral and affect people to take certain actions or decisions. The contagion model is generic in this sense - an action or an event leads to a transition from existing state to some other state defined by the event.

3.4.1 Alphabet

The first and foremost concept we present is the alphabet. By definition [37,38], an alphabet is a set of symbols, letters or tokens from which finite length strings could be formed. In formal language, we use the symbol Σ to describe the language L over which set of words are formed, where the set of words is denoted by the letter Σ^*. For simplicity, we use the notation $\mathcal{A}$ for the alphabet and parameterize it over a generator (a parameter) τ from which the set of strings or words are derived. Thus, $\mathcal{A}(\tau)$ is an alphabet over the parameter τ. For our benchmark, we allow τ to be defined over a set integers (preferably non-negative), strings or any *type* which is language specific. For example, τ can be an `unsigned int`, `int`, `long` or `string` or any *type* from which finite length words may be formed.

Model of

The Alphabet concept models primitive data types (i.e `int,double,string,` etc.) or derived data types such as `struct,class,` etc.

Motivation

Our benchmark focuses on simulating interactions within a social network, where the entities affected are essentially persons or humans(say) and the region or place where interactions occur is characterized by locations. Thus we might want to label the persons and locations with some tags (or say names or id's) so that every entity is unique in its own kind. A simplest example would be a person's name or a location's co-ordinate. Another example could be a social network comprising of web-pages where user profiles are entities that are affected, the location is the website and the underlying contagion model can be anything such as a viral advertisement which drives the corresponding action over user profiles. Each of the entities is assigned a label. It is noteworthy to consider that similar entities must share a common alphabet type.

3.4.2 Collection

Having described the alphabet, we now define the Collection concept. A collection $\mathcal{C}$ consists of a finite set of Alphabets $\mathcal{A}(\tau)$. We define a collection $\mathcal{C}$ as an abstract data type consisting of a group of elements having some common characteristics. For example, a collection of persons defined over an alphabet of `strings`, a collection of locations defined over an alphabet of type `int`. Every item in a collection is uniquely identified by its index or any other identifier.

Model of

Collection concepts must model the *Container* concept in STL. A collection can be a `map, vector, array, list` or any other *type* capable of containing the alphabet strings in it. It must provide access to the elements and also be able to modify (add/remove) items in it. Also any collection $\mathcal{C}$ must be Iterable i.e., must provide a mechanism or formally, an interface for accessing elements contained in it. The Iterator concept is described in the next section.

3.4.3 Iterator

In C++, the concept of **iterator** is fundamental in understanding the Standard Template Library [49] mainly because iterators provide a means for accessing data stored in container classes such as `vector,list,map` or any other *type* that models the Container concept. Basically, iterators are generalization of pointers used in common programming languages. The iterator points to an element that is part of larger collection of items. Our definition of the iterator concept closely resembles to the STL iterator concept, but instead of container classes we generalize our iterator operator $\mathcal{I}$ over any Collection $\mathcal{C}$ defined over and alphabet $\mathcal{A}$. An iterator $\mathcal{I}$ over any collection $\mathcal{C}$, in principle, traverses over the alphabet over which $\mathcal{C}$ is defined. Without loss of generality and for consistency, we make a requirement that that $\mathcal{I}$, if pointing to the last element of collection, is actually pointing to the address location **past** the last element.

Mathematically, $\mathcal{I}$ is defined as function (or operator) $\mathcal{I} : \mathcal{C}(\mathcal{A}) \rightarrow \mathcal{I}(\mathcal{A})$. i.e., an Iterator over a collection $\mathcal{C}$ is a mapping from every element in collection to an entity pointing to that element and when dereferenced, gives the desired content of that *type*. The iterator operator outputs an iterator range $\mathcal{I}(\mathcal{A})$ facilitating access to all the elements of collection by returning an iterator pair which marks the beginning and iterator past-the-end of the collection.

Two important iterator types required accessing elements of any collection viz. *read* and *write*

iterators or rather *output* or *input* iterators (respectively). Iterators are identified by their associated types and functionalities they provide when defined over a collection. For example, in above definition, the iterator used to traverse a collection does not modify its elements using any add/remove operation. Such an iterator is a *read* iterator or just a traversing iterator. A *write* iterator on the other hand can insert or delete any element from the collection. If we compare this with the C++ iterator concept, a read iterator is an *output* iterator of type *const_iterator*<·> while a write iterator is an *input* iterator which adds or inserts an element into the collection. It can modify the elements in the collections and hence **not** a *const_iterator*<·>.

Model Of

The iterators concept in our benchmark is the model of STL Iterator Concept. Iterators can be categorized according to their functionalities and the language under consideration. In C++, the important categories on which our iterator concept is based upon are: **Bidirectional, Forward, Input, Output, and Random Access**.

3.4.4 Maps or Property Maps

Maps (or property maps) are special type of collections in which, every element is associated with an attribute (i.e. a property). Mathematically, maps are functions that associate elements from one set (domain) to another set (co-domain). However, all the elements are bound by the same mapping function.

A map element is a (key,value) 2-tuple governed by a one-to-one mapping function f. This function associates every element from the *key* set to a particular(or specific) *value* set given by: $\mathcal{M}_f$: key $\rightarrow$ value

The *key* and *value* depend on the alphabet $\mathcal{A}$ over which they are defined. We require that a map $\mathcal{M}$ contains unique *keys*. We use property maps to associate an entity's attributes with that entity. For example, an Address map would store a person's residential address, a Location map would store location's demographic attributes such as co-ordinates etc. To identify one map from another, we use the *tags* to label our maps. A *tag* is just an identifier attached to the map. Denote the notation for tagging the map as `attribute_map`, then the map accessors `get` and `put` are defined to access elements of $\mathcal{M}$ as follows.

- A **`get`** operation on the attribute map fetches or returns the **value** associated with a desired

key. For a given key k, the value v is returned as:

$$v = \texttt{get(attribute_map, k)}$$

- A **put** operation inserts (or updates if already present) a *key-value* pair into the attribute map. Given a key k, insert (or update if already present) into $\mathcal{M}$ using:

```
put(attribute_map, k, v)
```

Since we deal with graphs in our benchmark where the graph elements represent any physical entity (person, locations, contacts, activities, etc.) with some attributes (such as name, age, co-ordinates, duration, etc.), we use the *tag* based property maps to associate attributes with the graph elements (vertices and edges). We also require that the graph concept (explained in the next section) must be able to provide a mechanism to initialize, store and access such information in property maps. The requirement is also present in the Graph Concept where a graph must satisfy the requirements of property map concept.

3.5 Graph

Our goal in developing this benchmark is to derive a standardized interface for all the data structures involved. Mathematically, social networks are best represented as graphs consisting of edges and vertices. The vertices in the graph represent the entities in social network while the edges represent the contact or any form or relation or connection between them. For example, a person (an entity) visits a location (again an entity) at a given time. This visit can be interpreted as an edge in a person-location graph.

However, for our benchmark to be generic, our underlying Graph data structure must meet some basic requirements in terms of accessing and storing the graph elements and their associated properties (such as time stamp, location id, etc.) Thus deriving a generic interface for data access is a key requirement to obtain a generic benchmark. Since graphs are data structures composed of several other data types and containers which store these types, the most suitable design pattern to extract the generic component would be the iterator pattern. Iterators provide a standard method for accessing elements of particular type for common containers. Here we provide specifications for graph data structure which much provide necessary iterators over its elements (vertices, edges, adjacent elements, etc.) for accessing them. The interface requirements we enlist here makes it possible for any graph library to be easily usable with our kernels by just implementing such

interface. Thus by changing the underlying *type* of Graph and abstracting standard interfaces of it, our benchmark provides a generic interface for input data structures that represent social networks. Our kernels are adaptive and provide the same consistency throughout which makes it generic.

Our specifications for the Graph interfaces are very similar to and are adapted from the Boost.Graph interface [8] which provides a rich interface for many graph operations and algorithms without compromising the generic programming principles. The input Graph to the benchmark must model the graph concepts which we describe here. The concepts are categorized into three components: (i) Members (Vertices, Edges, Properties), (ii) Iterators, (iii) Property Map Accessors. The Members consist of graph components such as vertex and edge identifiers. To access the data members of graph, we must abstract the interface to model the Iterator concept as described before. The Graph Iterators are specialized constructs that provide range based access to all vertices, and edges of the graph. Finally, the Graph must also provide mechanism for storing and accessing the properties associated with every vertex and edge in the graph. Thus, our input graph is also a model of Property Maps.

3.5.1 Graph Elements

The input graph must define the following members for graph elements. We call them the associated types required to satisfy our Member requirements. They are enumerated as follows:

- **vertex_identifier**: A vertex identifier `vertex_id_t` identifies every vertex v in the vertex set uniquely. The identified must model an Alphabet $\mathcal{A}(\tau)$ over the parameter τ. For example, assigning names to vertices by generating unique strings over the alphabet $\mathcal{A}(\texttt{string})$.

- **edge_identifier**: An edge identifier `edge_id_t` identifies every edge v in the edge set of the graph. Edge Identifiers must also model the Alphabet concept in the same manner as described for Vertices. The reason for this requirement is because, if there are multiple edges between two common vertices, then they are essentially different from each other.

  An edge in a graph connects any two vertices in the graph and is represented by a vertex tuple (vd_t, vd_t). The alphabet for edge descriptor depends upon the vertex descriptor pair. Given two vertices with vertex descriptors as v_1 and v_2, there could be multiple edges between them.

  For example, in a graph where every edge contains some specific weight, let $e_1 = (v_1, v_2)$ and $e_2 = (v_1, v_2)$ be two edges between the two vertices. Also let weight of e_1 be $w_{e_1} = 3$

and weight of e_2 be $w_{e_2} = 4$. Clearly, the two edges are not the same as $w_1 \neq w_2$. Hence their edge descriptors won't be same as well i.e., $e_1 \neq e_2$.

Overall, edge and vertex descriptors are abstract objects and are implementation dependent. We could ensure non-repetition of the edges by allowing only one edge between two vertices. It is not hard to follow the fact that a graph will always contain unique vertices, but may not edges.

- **Implicit Requirements**:

  1. The graph can contain multiple edges with same start and end vertices. Hence we require edge identifiers to be unique and not merely just vertex-vertex pairs.
  2. The graph must be bidirected or undirected.
  3. The graph must have a null vertex and edge as an empty vertex. This is a trivial requirement in graph algorithms. The algorithm must be able to return null vertex or edge if it does not find one in the graph.
  4. The graph must be non-mutable. Vertices and edges may not be deleted from the graph, however logical deletion may be possible using filtering methods over a predicate (see filter iterator in [40])

3.5.2 Graph Iterators

Iterators defined over graph elements facilitate efficient access by providing a uniform and standardized interface. Graph Iterators are used to iterate over vertices and edges and also can be specialized for adaptor like behavior such as iterating over out-edges of a vertex etc. Graph Iterators must model our Iterator concept and must be designed based upon the data structures chosen for storage of graph elements. The following graph iterators must be defined that provide graph element access.

- **`vertex_iterator`** : A vertex iterator $\mathcal{I}(\mathcal{C}_{vid_t})$ provides access to all the vertices in a graph. The value type of the vertex iterator must be the vertex identifier `vertex_id_t` of the graph. The operator $\mathcal{I}$ over collection of vertices of type `vertex_id_t` provides the iterator range to access all vertices in the graph.

- **`edge_iterator`** : An edge iterator $\mathcal{I}(\mathcal{C}_{eid_t})$ provides access to all the edges in a graph. Thus the value type of the edge iterator must be the edge identifier `edge_id_t` of the graph. The operator $\mathcal{I}$ over the edge collection with edge type as edge identifier `edge_id_t` provides the iterator range to access all the edges in the graph.

- **out_edge_iterator** : An out-edge of a vertex is edge containing the given vertex as the source (or start) vertex and the other vertex as the target (or edge) vertex. The out-edge iterator of a vertex provides access to all such out-edges (if any) of that vertex in the graph. As the iterator provides access to out-edges of the graph, the value type of the iterator must also be the edge identifier `edge_id_t` of the graph. The operator $\mathcal{I}$ must be defined for a given vertex identifier `vertex_id_t`, but returns the iterator range whose value type is edge identifier of the graph.

  The type of edges in the graph are assumed to be bidirected or undirected. In either case, the out-edge iterator for a given vertex will return the desired iterator range for all the out-edges of that vertex.

- **adjacency_iterator** : An adjacent vertex of any vertex is the one which is connected to the current vertex with an edge. The adjacency iterator provides access to all such vertices that are adjacent to a given vertex. The value type of the adjacency iterator must be the vertex identifier `vertex_id_t` of the graph. The operator $\mathcal{I}$ must be defined for a given vertex in the graph in order and must return an iterator range providing with value type as that of the vertex identifier of the graph.

Standard Functions

Access to members of a graph is generally facilitated by means of functions. These functions return the desired object (by value or reference) for use in generic algorithms. The following functions must be defined to facilitate access to vertices and edges of the graph. The functions may take an input parameter and return some output parameter depending upon the functionality. As we require our graph to be non-mutable, we do not require any graph mutating functions to be defined for our benchmark. Some notations which are used are given below.

Denote `Graph` as a type which models our non-mutable graph and G as a graph object of that type. Let e be any edge in the graph G of type `edge_id_t` and u, v be any vertex in graph G of type `edge_id_t`.

- **add_edge(u, v, G)** : Must insert a given edge (u, v) into the graph G and return an edge identifier pointing to the new edge. For edges already present in the graph with that identifier, the returned edge identifier must point to that edge.

- **add_vertex(G)** : Must add a new vertex in G and return an identifier (of type `vertex_id_t`) for that vertex.

- **`num_vertices(G)`** : Must return the total number of vertices in G.
- **`num_edges(G)`** : Must return the total number of edges in G.
- **`vertices(G)`** : Must return an iterator range $\mathcal{I}_v$ over the collection of all vertices in G. $\mathcal{I}$ is the iterator operator and $Graph$ is the type of the graph provided. The operator should return an iterator pair of the type $vertex_iterator$.
- **`edges(G)`** : Must return an iterator range $\mathcal{I}_e$ over the collection of all edges in G. The operator must return an iterator range of the type $edge_iterator$.
- **`adjacent_vertices(v,G)`** : Must return an iterator range $\mathcal{I}_{adj}$ over the collection of all the adjacent vertices of the given vertex v in G. The operator must return an iterator range of the type $adjacency_iterator$. The value type of the iterator however should be $vertex_identifier$ of the graph.
- **`out_edges(v,G)`** : Must return an iterator range $\mathcal{I}_{out}$ over the collection of all out-edges of vertex v in G. $\mathcal{I}_{out}$ must return an iterator range of type type $out_edge_iterator$. The value type of the iterator however should be $edge_identifier$ of the graph.
- **`source(e,G)`** : Must return the vertex identifier of the source (or start) vertex of any edge $e = (u, v)$. Return type should be $vertex_identifier$. In the above example, the operation will result in returning the vertex identifier for u
- **`target(e,G)`** : Must return the vertex identifier of the target (or end) vertex of the edge $e = (u, v)$. Return type should be $vertex_identifier$ of graph. In the above example, the operation will result in returning the vertex identifier for v

3.6 Contagion Model

3.6.1 Motivation

To study how a disease or any contagion spreads across a social network, we must understand and model its underlying mechanism or operation. There are several methods to model diseases for example, using differential equations or compartmental models or finite state machines. We enlist important parameters for modeling disease (or any other contagion for that matter) and state their constraints in order to derive a functional benchmark.

3.6.2 Requirements

A disease is characterized by a state machine (FSM) consisting of a set of states (called stages) and a transition from one state to another is triggered by means of an event. In short, a transition (or jump) from one state to other happens if the evaluation of event yields a positive result. Our benchmark requires that the disease model is an event driven process for state transitions. Since we are dealing with diseases the disease model parameter of our benchmark (`DiseaseModelT`) must be capable of the following requirements as stated below:

1. Any realization of DiseaseModelT must be defined by a set of disease states and incubation periods i.e., the time period a vertex (contagion element) remains in that state before transitioning to next state.
2. The model should provide methods to define transitions from one state to other.
3. The model must provide initialization of vertex (person nodes) state depending on the input criteria defined by user (this criteria is basically the initial fraction of people infected, susceptible etc.)
4. The model must also provide an update mechanism for updating current state of vertex (person) based on the disease model and other related parameters.

3.6.3 Mathematics behind Disease Model

Without reinventing the wheel, our benchmark uses Co-evolving Graph Dynamical System described in [27–30]. Since interaction based social networks are inherently dynamic in nature, our disease model is based upon those used in the discrete graph dynamical systems [31]. The authors in [27–30] describe CGDDS as a triple $(G(V, E), \mathscr{F}, \mathscr{W})$ consisting of three basic components. We use only those components which we require for building our dynamic disease model. The $G(V, E)$ is the underlying social network with V as set of vertices, and associated with each vertex v_i is an edge modification function g_i. Applying g_i for every vertex v_i results in an indexed sequence of social networks. For each vertex v_i, there is a set of local transition functions. The function used to map the state of vertex v_i at time t to its state at time $t+1$ is $f_{v_i,dt}$, and the input to this function is the state sub-configuration induced by the vertex and its neighbors in the contact network N(i,t). The final component is a string W (a schedule) over the alphabet $v_1(s), v_2(s), ..., v_n(s), v_1(g), ..., v_n(g)$. W represents an order in which the state of a vertex or the possible edges incident on the vertex will be updated. Both f_i and g_i are assumed probabilistic. More details can be found in [27–30].

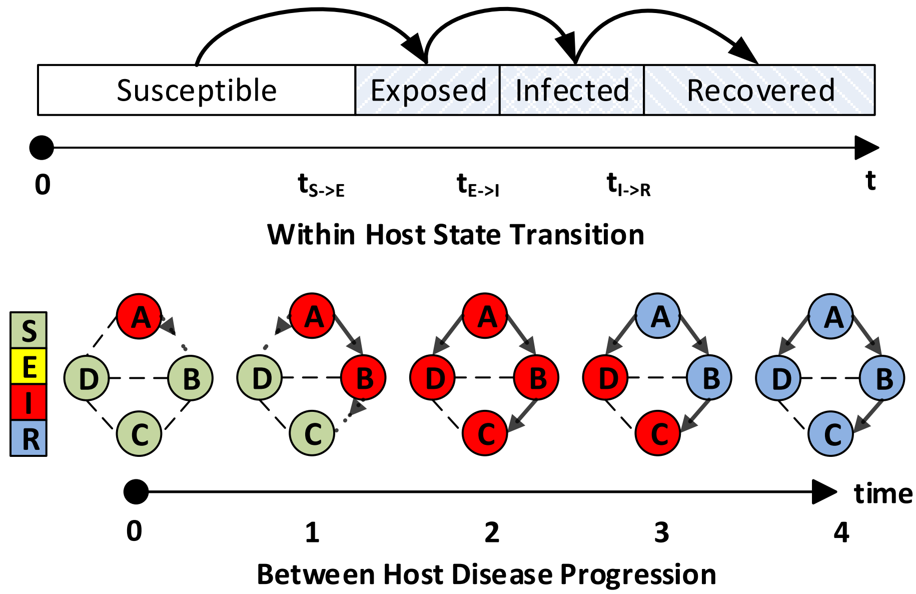


Figure 3.1: (i) **SEIR** Model, (ii) a four node and five edge contact network with $\Delta t_E = 0$, $\Delta t_I = 2$ (in days) for each node and transmission probability 0.5 for each day. Node A is infectious at start. (i) **Day 0**: A transmits the disease to B but not D. (ii) **Day 1**: both A and B are infectious. A transmits the disease to D; B infects C but not D. (iii) **Day 2**: A is removed, all others are infectious with no susceptible nodes. (iv) Nodes are removed gradually and on **day 4**, the system enters a fixed point and stops evolving. The state transitions are one-way (from susceptible to exposed to infected to recovered) with no other possible transitions.

Thus our input Disease model `DiseaseModelT` should take as input: (i) Set of states which defines a state s_v (taken from a finite set K) for each vertex v. (ii) Set of transition functions f_v for every vertex which maps the state of vertex v at time t to the vertex state at time $t+1$. The disease model must adhere to the requirements as stated above. Figure 3.1 depicts an SEIR model [9] with four disease states and a four node and five contact edge contact network (Images are adapted from [7])

3.7 Functors – Background

Before we discuss about the Interventions and Algorithms in detail it is important to understand the concept of **Functor**. Implementation wise, functors and lambdas (anonymous functors) are discussed in chapter 6. Here we attempt to provide a functional programming approach for functor. It is important to note that functor as a whole is not a concept requirement for our benchmark, but is a *type* which our Interventions and Algorithms concepts are based upon.

In mathematics, a *functor* is structure preserving mapping between categories, i.e., A function between categories which maps objects to objects and morphisms to morphisms [41–44]. A category [45–47] consists of a class of objects and a class of morphisms or maps (say) between objects. A morphism f is unique in the sense that the map $f : a \rightarrow b$ is unique for every source and target object. A category must satisfy the associativity and identity axioms. These axioms are very crucial in developing mathematical foundations for Intervention and Algorithm functors in our benchmark Mathematically, *functor* is defined as follows:

If X and Y are two categories and $\mathcal{F}$ is a functor from X to Y then,

- $\mathcal{F}$ maps to each object $x \in X$, an object $\mathcal{F}(x) \in Y$,
- $\mathcal{F}$ maps to each morphism (or arrow) $f : a \rightarrow b \in X$ to a morphism (or arrow) $\mathcal{F}(f) : \mathcal{F}(a) \rightarrow \mathcal{F}(b)$, such that the two axioms hold:
  1. Identity preserving: $\forall x \in X$, we have $\mathcal{F}(I_x) = I_x$ where, I is the identity operator
  2. Associativity preserving:
     for morphisms $m, n \in X$, we have $\mathcal{F}(m \circ n) = \mathcal{F}(m) \cdot \mathcal{F}(n)$

From functional programming perspective, languages such as Haskell have a class called *Functor* that define f as a **polytypic** function used to map functions (i.e morphisms) on an existing type to functions on some new type. For our benchmark, our *f* maps or morphs the input vertex (with attributes) to the same vertex (but with modified attributes). A simple example is that of any intervention (anti-viral, vaccination, etc.). When we apply a particular type of intervention to a group of persons in a network, we do not change their actual type (person-type), but we apply the intervention functor (antiviral,vaccines) and change their attributes. The mapping is an identity mapping of some sort. For our benchmark, we only require the identity preserving behavior of functors and define the same. We now proceed to specify requirements for Intervention and Algorithm Functors as below.

3.8 Intervention

3.8.1 Motivation and Background

Interventions in real life are the measures or necessary course of action taken by public health authorities (such as CDC etc.) to control and reduce the outbreak of any disease. Depending upon the disease, the measures could range from public location closures, vaccination camps, anti-viral administration to absolute quarantine in some areas. Thus interventions are one of the factors affecting the spread of disease and hence the kernels which we use to simulate the spread of contagion must have the ability to select and apply different interventions in order to carefully study disease dynamics and converge to the apt intervention strategy in case of any real event.

Since social networks are represented as graphs with persons and locations as vertices and the edges as activities or contacts, simulation based interventions are applied by altering the graph structure such as addition or removal of edges or redirecting an edge to other location vertex. Such edge-based interventions model those which belong to closure of locations, while node based interventions alter the vertex properties such as infectivity or susceptibility probabilities. The type of intervention and its parameters for a particular strategy affects the benchmark results considerably. For example, two strategies with different infectivities would yield different rates of infections and hence would affect overall time. Similarly, closure of multiple locations or any single random location would alter the total number of active visits in the social network and hence produce varying results. Therefore, intervention type is an important parameter in deciding a strategy.

3.8.2 Intervention Functor

The Intervention operator is a functor $\mathcal{F}_i$ that takes input as a set of entity-property pair (i.e., property map), and transform into new entity-property pair. The functor can model any type of interventions that we discussed. For example, Node Intervention Functor can model antiviral or vaccination type of functor. The input is a set of person vertices that need to be intervened along with their properties and output is the resultant set of new or updated properties by application of intervention functor map f over each person vertex. Since we use property maps to associate properties with every vertex and edge in the graph, the functor takes input the property set and outputs new/updated properties.

The input to the benchmark must be the intervention functors which we would like to model. Each

functor $\mathcal{F}_i$, models a particular intervention type. The functor can be applied over a set of vertices and edges as per design. Mathematically, the intervention functor's operation is similar to the following pseudo code.

```
1 InterventionFunctor   F(attribute_map);      //Functor
2 List<PersonVertices>  VertexList;            //List of vertices for functor
3 foreach (v in VertexList)
4   attribute t = get(attribute_map, v)        //fetch the property
5   put(attribute_map, v, F(t))                //apply the functor and
6                                              //update the property
```

We provide two intervention functors for our benchmark. Depending upon the type of intervention we can choose either the Node or Edge Intervention. For Edge intervention, the input is the set of edges (i.e. contacts or activities) and the functor modifies (logical delete, or alter activity duration etc.) the properties and updates the Edge Property Map. The input can be all the edges or a set of specific edges.

Node Interventions (NI)

Node Interventions alter the vertex properties of persons. Examples of such interventions include anti-viral and vaccinations. Physically, they alter the health of a person in real world but logically, these interventions change the probability of infecting others (i.e., infectivity) or probability of getting infected (i.e. susceptibility) of the person represented as nodes in the graph. (See Fig 3.2).

Edge Interventions (EI)

Edge Interventions alter edge properties or activities. An example of edge intervention is location closure in which the intervened edges are redirected to alternate locations. In a real situation, a person may stop visiting a location (say office or mall). But in a simulation over graphs that represent social networks, it results in activity modification or alteration. A simple alteration would be to redirect the edges to some other location or change the activity duration (see Fig 3.3).

3.8.3 Intervention Requirements

We now summarize the intervention functor requirements:

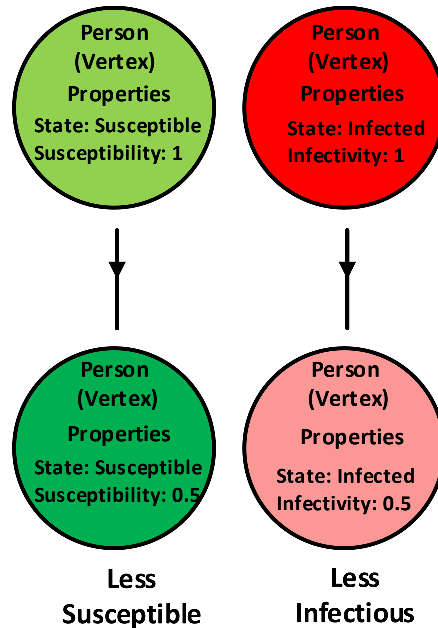


Figure 3.2: Node Intervention: Alter Vertex Properties of Persons (nodes). Note that a change in probability from 1 to 0.5 in both cases (infected or susceptible) will reduce the probabilities of infecting others or getting infected

1. The intervention functor which models the Functor concept must specialize in either of the two types of interventions – Node or Edge Interventions (as explained above)

2. The vertex or node intervention functor must be able to modify/alter the attributes for person like entities (not location): some attributes for interventions may be health states, specific incubation parameters etc.

   - Input: An intervention functor $\mathcal{F}$ and a list of persons (vertices) and their health states (i.e. attributes) $P = \{(p_1, h_1), (p_2, h_2), \ldots, (p_n, h_n)\}$.
   - Output: Updated health states (attributes) of persons.
     $P' = \{(p_1, h'_1), (p_2, h'_2), \ldots, (p_n, h'_n)\}$. such that $\mathcal{F} : P \to P'$, where $\mathcal{F}$ satisfies the functor concept.

3. The edge interventions alter the activity or contact attributes for persons. An activity is represented by an edge and hence the edge intervention functor must be able to modify the activity attribute for edges in the graph.

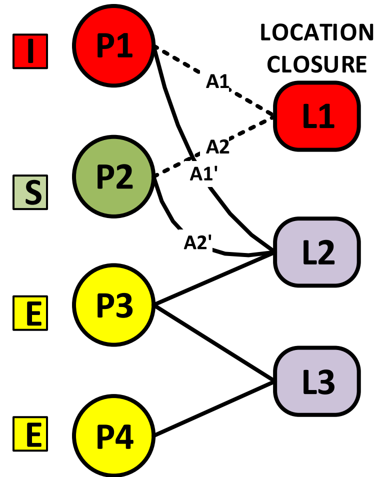


Figure 3.3: Edge Intervention: Alter Edge Properties (Activities or Contact period). Location L_1 closure results in redirection of P_1's and P_2's edges to location L_2.

- Input: An intervention functor $\mathcal{F}$ and a list of activities (edges) and their attributes (such as duration etc.). $A = \{(e_1, a_1), (e_2, a_2), \ldots, (e_n, a_n)\}$.
- Output: Updated activities of persons.
  $A' = \{(e_1, a'_1), (e_2, a'_2), \ldots, (e_n, a'_n)\}$. such that $\mathcal{F} : A \to A'$, where $\mathcal{F}$ satisfies the functor concept.

3.9 Contagion Functor

A contagion functor $\mathcal{F}_c$ facilitates transmission of a disease between two individuals. Transmission of disease is defined by the disease model. The contagion functor must be able to alter the health states of two interacting individuals and thus requires the input disease model to be its core component. $\mathcal{F}_c$ is the functor that maps an individual's state based upon his interaction with other individual(s). The mapping in this case is probabilistic and is disease model specific in general.

- Input:
  1. Functor $\mathcal{F}_c$ with disease model as parameter input: $\mathcal{F}_c\langle M \rangle$
  2. List of individuals and their current states.
     $L = \{(p_1, s_1), (p_2, s_2), \ldots, (p_1, s_1)\}$

- Output:

  1. List of individuals and their updated states (after the application of functor to every other element in associative manner). $L = \{(p_1, s'_1), (p_2, s'_2), \dots, (p_1, s'_1)\}$.
  2. The mapping $\mathcal{F}_c : L \to L'$ is such that for any two elements $p_1, p_2 \in L$, we have $\mathcal{F}_c\langle D\rangle(p_1, p_2) \to s'_1$ (state update for person p_1)

The primary motive of our benchmark is to study the effect of contagion spread in a social network. Since social networks are modeled as graphs and simulations model the real time behavior of any disease mechanism, the contagion functor must take a disease model as its input parameter. Thus, when a functor is applied over two interacting individuals, it will update the individual's current state to next state as per the transitions defined in disease model.

3.10 Algorithms

The Algorithm concept is the most important component of our benchmark. An algorithm is the underlying method or strategy for contagion diffusion. Algorithms help unleash the data access patterns and computational complexities for a particular implementation strategy with chosen data structures. Prior selection of algorithm is necessary for deriving a sound compile time strategy. This is also advantageous in reducing the run-time overhead during the simulation.

Algorithms help visualize how a contagion in a network takes place. For example, the EpiSimdemics algorithm [1] is based upon people interacting at locations where disease propagation occurs at the common meeting place, i.e., the location under consideration. On the other hand, EpiFast [7] relies on person-person contacts for disease propagation with an implicit underlying location present between two individuals. Common to both these mechanisms are the contagion functor and disease model (Note: intervention functors are common as well, but they are not mandatory).

Thus we are successful in separating our benchmark into individual components, whose different implementations (of concepts) when plugged in together, form different strategies at disposal. All this decision is taken at compile time which makes it efficient and deterministic in nature.

3.10.1 Algorithm Traits

Algorithms, by nature are **repetitive**. For example, in matrix multiplication, a row and column are multiplied (element wise) and summed together to form the corresponding entry at the row-column intersection. This process is repeated for all such rows and columns until the final matrix is formed. In the same way contagion simulations are processed over a simulation time which is divided into individual subintervals. The disease spread mechanism (i.e, the contagion) followed by individual state updates and interventions are repeated for every simulation subinterval.

In our work for benchmarking the simulation of contagion spread in social networks, the algorithms used are repetitive in nature in the sense of following:

- Time wise repetition (in every simulation phase, do a bunch of things: spread contagion in network, update person states, apply interventions)
- Space wise (at every time subinterval, do the following : list all locations; list all persons at a given location; for every such location, gather all persons, etc.)

3.11 Concepts Combined Together

We are now ready to define an implementation strategy for our benchmark and then proceed to kernel specifications in the next chapter.

Let $G(V, E)$ be a graph of type $\mathbb{G}$ representing any social or contact network where elements of G satisfy the following concepts as itemized

- Vertices and Edges are defined over an Alphabet that identifies every vertex and edge uniquely in the graph.
- Every vertex and edge have properties attached to it which are modeled by the Property Map concept. The `get` and `put` operations are used to access these properties.
- The data structures used to store the vertices, edges and their associated properties must model the Collection and Iterator Concepts. This provides a uniform and standardized approach for accessing data. Any implementation conforming to such requirements would be easily compatible with the benchmark.

Define $\mathcal{D}$ as a contagion model (or disease model) that satisfies the Contagion model concept in section 3.6. Denote the disease model defined over a finite domain $\mathbb{D}$. This includes providing an interface to initialize a persons health state prior to contagion process and also, within-host and between-host disease propagation mechanisms.

We use the notation $\mathbb{F}$ for any Functor with subscripts defining its *type*. A contagion functor $\mathcal{F}_c$ takes as input the disease model and provides a mechanism for transmission of contagion between two individuals based on the input contact network (can be location based or contacts based). $\mathcal{F}_c$ is the model of functor $\mathbb{F}$ and $\mathbb{F}_{\mathcal{F}_c}$ is the notation used to denote the contagion functor. Similarly, define a set of intervention functors $\mathcal{N} = \{\mathcal{F}_i\}_{i=1}^{n}$ where each $\mathcal{F}_i$ is an intervention type which either models Node Intervention or Edge Intervention concept. Thus, with respect to our functor notations, we have $\mathbb{F}_{\mathcal{N}}$ as the functor for interventions.

And finally, let $\mathbb{A}$ be an algorithm defining the time and space related parameters that facilitate the entire contagion simulation process on the input graph. This includes the local and global parameters, state update mechanisms for person-like entities, applying interventions on a population subgroup and also maintaining a global contagion count. The algorithm can be serial, parallel or any hybrid method. For parallel and hybrid methods (which we will study in the chapter 5), we must ensure the correctness and consistency of Collections and Properties so that they don't end up in undefined state due to race conditions between multiple threads or processes or other similar problems. Selecting an algorithm may also require choosing a hardware platform (distributed memory, shared memory or standalone, or hybrid CPU-GPU, co-processor based etc.).

Thus, an Implementation Strategy $\mathbb{S}$ is a quintuple $\{\mathbb{G}, \mathbb{D}, \mathbb{F}_{\mathcal{F}_c}, \mathbb{F}_{\mathcal{N}}, \mathbb{A}\}$ where each tuple element is minimally dependent on other. The design of benchmark is to create components that are loosely coupled so that a perturbation in one component does not affect others. A particular instance (i.e, selection of these parameters) results in a unique implementation strategy such that any change in one of the parameters would yield an all-together different strategy. Such is the flexibility and genericness demonstrated by our benchmark. In the next chapter, we provide specifications for our kernels based upon our the concepts we discussed here.

Chapter 4

Benchmark Specifications

In this chapter, we first provide Epidemiology based specifications for kernels and then build their corresponding ***Concept*** based specifications. The benchmark concepts as described in chapter 3 are applied to develop specific kernel related requirements. Each kernel performs a set of tasks (or operations) by taking an input and provide output for the next kernel.

4.1 Epidemiology Specifications

These specifications merely help understand the high level functionality every kernel must offer and satisfy.

4.1.1 Synopsis

(a) **kernel 0** : Construct an activity based Person-Location social network with a given number of persons and locations. An activity is a visit of a person (entity that take part in contagion) to a location (place where contagion happens) at a given time.

(b) **kernel 1** : Construct a Person-Location Graph from the Person-Location activity list generated in kernel 1.

(c) **kernel 2** : Derive Person-Person network from Person-Location network and construct a Person-Person Contact Graph.

(d) **kernel 3** : Create an abstract hierarchical Person-Location graph by iteratively grouping locations into location groups.

(e) **kernel 4** : Simulate a contagion diffusion process over the Person-Location Activity Graph.

(f) **kernel 5** : Simulate a contagion diffusion process over the Person-Person Contact Graph.

4.1.2 Kernel 0 : Activity List Generator

In a social contact network comprising persons and locations, an activity is an event which is said to occur when a person visits a location at any given time. A person can visit more than one location in a day, but cannot be present at two locations during same or overlapping time intervals. The kernel simply creates a list of persons and locations and pairs them together using a generating function. The generator function is chosen to derive a specific network with common parameters such as degree distribution factors, clustering co-efficient, etc. Our desired output from the kernel is to obtain a person-location social network.

4.1.3 Kernel 1 : Person-Location Graph Generator

The person-location social network is transformed into a Person-Location Graph with vertices and edges. The vertex set of the graph consists of persons and locations whereas edges represent the activities. The Graph is a labeled graph in which demographic attributes are assigned to person and location vertices and their activities (edges). For example, person vertices may be labeled with names, health states, etc. while activities could be assigned a time duration which is the timestamp of the visit.

Using the person-location network, a person-person contact network is derived as follows: for every location, accumulate all persons that are supposed to visit the location on a given day and then form person-person pairs in that group. By removing the redundant pairs (i.e., (p_1, p_2) is same as (p_2, p_2)), we get a person-person contact network.

4.1.4 Kernel 2 : Person-Person Social Network Generator

The person-person contact network is transformed into a Person-Person contact graph with vertices as the person entities and edges as the contacts. Similar to kernel 1, the vertices and edges in the graph are assigned demographic attributes and have their health states initialized.

4.1.5 Kernel 3 : Hierarchical Person-Location Graph

Just as a group of neighborhood forms a town, several towns form a city and many cities together form a state. Kernel 3 abstracts the person-location graph by creating a location hierarchy amongst locations. Some locations are grouped to form a small location sub-group. Sub-groups form a location group (as a whole), and location groups when grouped again form a super-group. Overall we obtain a hierarchical person-location-group graph. The internal graph is not modified in any case, the kernel facilitates access to the abstracted location hierarchy to be used in the next kernel.

4.1.6 Kernel 4 : Activity based Contagion Simulator

This kernel simulates disease propagation in the person-location graph. Since graphs model social networks very efficiently, the study of dynamics of any infectious disease in social networks are best interpreted by creating similar conditions in a simulated environment. That is, produce a computerized simulation of an outbreak using graphs (that model our social network). The kernel would provide statistics such as the number of contagions per day, total number of contagions, etc. This kernel derives and captures computational aspects of common contagion simulation algorithms applied over person-location graphs.

4.1.7 Kernel 5 : Contact based Contagion Simulator

Kernel 5 aims to capture the computational aspects of the algorithms that simulate disease spread in person-person contact networks. Essential statistics may be provided by the kernel as noted in kernel 4 above.

4.2 Concepts based Specifications

Having developed high level design specifications of the kernels, we now focus towards building concrete specifications that are crucial in developing, implementing and choosing a particular strategy for our benchmark. A strategy is evaluated on multiple platforms and is then characterized by low level results it generates (such as load-store operations, % CPU utilization, compute vs data intensive behavior, etc.)

4.2.1 Synopsis

(a) **kernel 0** : Takes as input, the degree distribution for persons and locations (the generator function). The output is a list of person and location pairs. Each pair (p, l) represents an activity by person p at location l.

(b) **kernel 1** : Constructs a Person-Location activity graph that models the **Graph** Concept. Other concepts are implied while satisfying the Graph concept (for e.g. Disease functor, Property Maps, Alphabets etc.) (see chapter 3, section 3.11).

(c) **kernel 2** : Constructs a Person-Person contact graph. (Other specifications are similar as that described for kernel 1).

(d) **kernel 3** : Forms a Location Group hierarchy by assigning group ids to all location vertices in the Person-Location graph and forms a set of such location groups.

(e) **kernel 4** : Implements a Strategy $\mathbb{S}$ for Person-Location Graph.

(f) **kernel 5** : Implements a Strategy $\mathbb{S}$ for Person-Person Graph.

4.2.2 Kernel 0

Motive

Generate a list of *Person-Location* activities and provide as input to kernel 1.

Requirements

1. Persons and Locations must satisfy the Alphabet concept i.e., unique and finite length strings must be generated.
2. Output must satisfy collection concept and must be iterable i.e., provide iterators to the collection of persons, locations and person-location activity pairs (P-L activities).

Desirable Features

1. Randomization function can be used to form P-L activity pairs. A randomizer function $\chi(P, L)$ randomly forms (p, l) pair and adds to the collection of activities.

2. The collection may be generated in parallel.

3. Abstraction over containers (to hold the P-L activity) must be present i.e., collection must satisfy iterator requirements.

4. The kernel must not depend on any particular type of data structure (collection) or specific alphabet type (`int`) and must support a range of containers and alphabets by abstracting the implementation by defining appropriate iterators.

5. Degree distribution parameter that serves as both generator and randomizer function may affect number of persons and locations generated. Desirable ratio of persons to locations must be approximately 4:1. We arrived at this number using statistical and modeling techniques applied to geographical and temporal data for US population available in our lab. However this ratio can be varied to further study its effect on kernels.

4.2.3 Kernel 1

Motive

Construct a Person-Location Graph (G_{PL}) using $P - L$ activity collection (from kernel 0). Assign properties (attributes) to edges and vertices. Initialize specific properties for edges and vertices such as initial health states, time of visit, etc. and construct property maps for every property.

(a) **Input**:

- An iterable collection of $(P - L)$ activities.
- A disease model parameter `DiseaseModelT` for initializing health states for person vertices.
- The *type* of graph (say `GraphT`).

(b) **Output**: Initialized Person-Location graph G_{PL} of type `GraphT`.

Requirements

(a) `GraphT` must model the Graph Concept.

(b) The collection of person-location activities must be iterable, i.e., the iterators defined over the collection are required to model the Iterator Concept.

(c) `DiseaseModelT` must conform to the Contagion Model concept (for health state initialization).

(d) Property Maps constructed for every vertex or edge attribute must conform to the Property Map Concept by defining appropriate attribute tags for every property and provide access using `get` and `put` operations.

Implications

(a) Internal data structures for storing vertices, edges and their properties in `GraphT` must also satisfy the Collection and Iterator concept.

(b) The task of initializing health states for person vertices may be done in parallel since initial state of any individual is independent of others. However the number of vertices that need to be initialized to a particular state must be defined in the `DiseaseModelT` parameter.

(c) Since every attribute of the vertex or an edge in the graph is a property type, `get` and `put` operations must be overloaded using attribute tags to resolve appropriate function call.

4.2.4 Kernel 2

Motive

Construct a Person-Person Graph G_{PP} using the person-person contact collection. Assign properties to vertices and edges. Initialize specific properties for vertices and edges such as initial states, time of visit, etc. and construct property maps for every property.

(a) **Input**:

- An iterable collection of $(P - P)$ activities.
- A disease model parameter `DiseaseModelT` for initializing health states for person vertices.
- The *type* of the graph (say `GraphT`).

(b) **Output**: Initialized Person-Person graph G_{PP} of type `GraphT`.

Requirements

(a) `GraphT` must model the Graph Concept.

(b) The collection of person-person activities must be iterable. i.e., the iterators defined over the collection are required to model the Iterator Concept.

(c) `DiseaseModelT` must conform to the Contagion Model concept (for health state initialization).

(d) Property Maps constructed for every vertex or edge attribute must conform to the Property Map Concept by defining appropriate attribute tags for every property and provide access using `get` and `put` operations.

Implications

(a) Internal data structures for storing vertices, edges and their properties in `GraphT` must also satisfy the Collection and Iterator concept.

(b) The task of initializing health states for person vertices may be done in parallel since initial state of any individual is independent of others. However the number of vertices that need to be initialized to a particular state must be defined in the `DiseaseModelT` parameter.

(c) Since every attribute of the vertex or an edge in the graph is a property type, `get` and `put` operations must be overloaded using attribute tags to resolve appropriate function call.

4.2.5 Kernel 3

Motive

Distribute the locations in the activity graph G_{PL} into location groups and construct an abstract location hierarchy.

(a) **Input**:

- Iterator over location vertices of the person-location graph G_{PL} from kernel 1.
- A `LocationGroupT` parameter that distributes locations into groups with exactly one location belonging to a location group.

(b) **Output**:

- A collection of location groups with each group containing one or more location vertices.
- An Iterator over the location group collection (Location Group Iterator).

Requirements

1. The location group collection must satisfy the Collection and Iterator Concept. This means providing necessary iterators for traversing over the collection.
2. Dereferencing any location group iterator results in a location group which must also be iterable. For example, a location group iterator on dereferencing results in a range of location iterators that belong to the same group.
3. It is required that a location vertex belongs to only one location group.
4. The `LocationGroupT` parameter to the kernel must be generic enough for extending the hierarchy. For example, a collection of places forms a sub-area in a city. Here, the city represents a location group container and sub-areas are location groups. Every sub-area contains locations (that are unique to the sub-area). Several cities combine to form a city group container. A group of cities combine to form a state and so on. Figure 4.1 below illustrates the requirement for extending the location group concept. However, it it is to be noted that, the location group doesn't necessarily have to follow spatial constraint. It can be done based on load balancing as well.

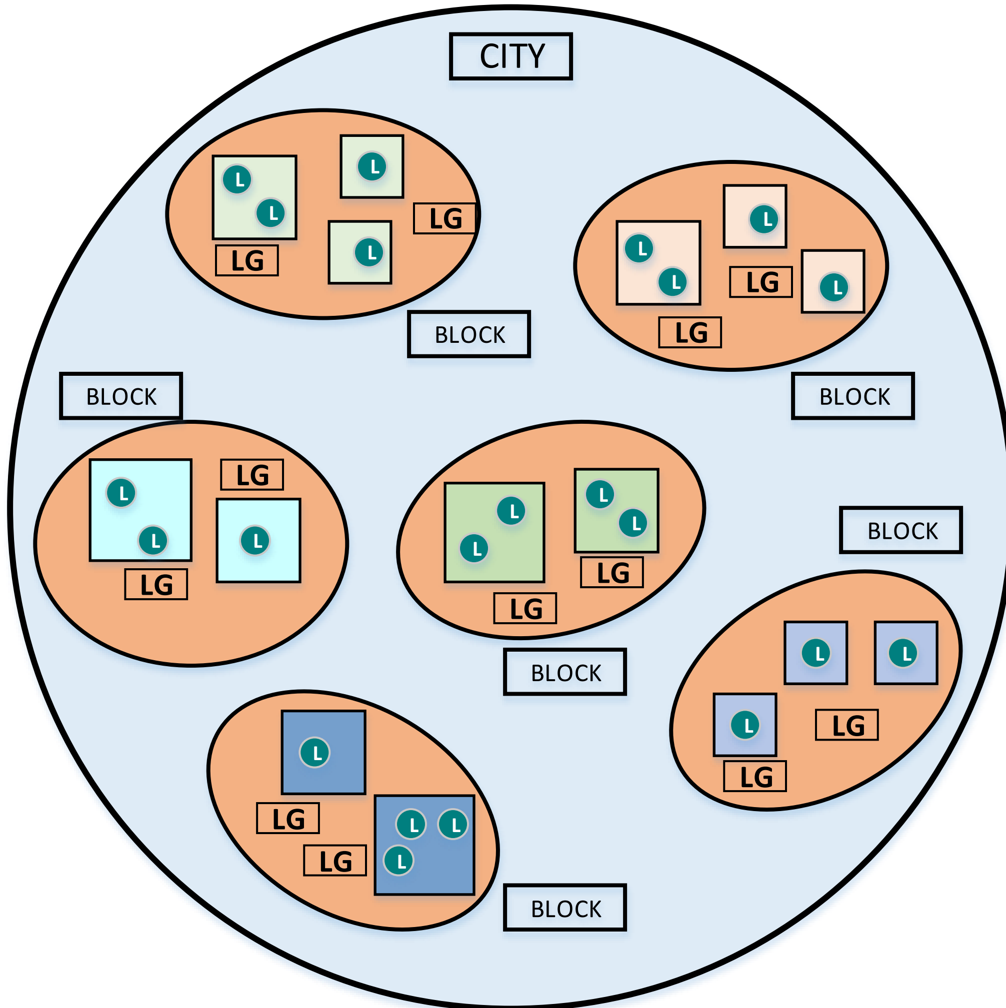


Figure 4.1: Location Hierarchy – City, Blocks, Location Groups, Locations

4.2.6 Kernel 4

Motive

Simulate a contagion process on a social network a.k.a. Person-Location graph G_{PL}. Report the metric. (see section 7.1 for metric details)

(a) **Input**:

- A strategy $\mathbb{S}$ comprising of the following parameters.
  (a) `GraphT` : The input person-location graph that models the Graph Concept.
  (b) `DiseaseModelT` : A model of Contagion Concept. The state update mechanism is part of the model and not a separate type.
  (c) `ContagionFunctorT` : A model of Contagion Functor Concept that defines the mechanism of contagion between two individuals (i.e., location based contagion or activity based contagion or any other type).
  (d) `InterventionFunctorT` : A set of intervention functors that model the Intervention Concept and implements any of the interventions (Node or Edge). (A separate functor is required for every type of intervention). The functors can be chained together by using any mechanism such as tuples, etc.
  (e) `Algorithm` : A particular computer implementation of contagion diffusion that captures the computational aspects of standard location based simulation algorithms such as EpiSimdemics [1] etc. where contagion takes place at common meeting place (location).

(b) **Output**:

- A sequence of system states over time $S(\Delta t), S(2\Delta t), \ldots, S(n\Delta t)$. Each state $S(\Delta t)$ represent the current health state of every individual in the contagion.
- Total interactions per second `TIPS` metric (see 7.1 for more details). The `TIPS` metric characterizes $\mathbb{S}$. We will touch upon characterization in chapter 7.

Requirements

1. **`GraphT`** : Must be a model of Graph concept and provide iterator style access to vertices, edges and their properties. Graph must not be mutable as we do not modify the underlying

physical memory structure of the graph. However, the entire system being dynamic in nature, where the contagions are probabilistic and are affected by intervention strategies, logical deletion of vertices and edges (such as filtering out those vertices and edges that do not take part in contagion process) is possible.

2. **`DiseaseModelT`** : The state update mechanism must be defined by the input disease model where a person's health state is transitioned to a new state, based upon the current state and other factors such as interventions, etc. The sequence of update and low level details is defined by the Algorithm. Thus, we separate update mechanism from the implementation (or algorithm). It is responsible for simulating within-host disease progression mechanism.

3. **`ContagionFunctorT`** : The contagion functor is required for simulating between-host disease propagation mechanism between two individuals.

4. **`InterventionFunctorT`** : Intervention strategies are parameters of benchmark and must be predefined. For example, randomly choose a location and apply edge interventions – so that all the activities to that locations are disabled in the next simulation phase unless they are turned on by a switch per se.

5. **`AlgorithmT`** : The Algorithm is a concept that is required to run the contagion on multiple platforms such as shared memory, distributed memory, hybrid (CPU-GPU), offloading to co-processors (Intel MIC). They aim to capture the computational aspects and data access patterns such as load-store, peak CPU performance, CPU throughput, etc. The benchmark must also provide a characterization of the algorithm in this sense.

4.2.7 Kernel 5

Motive

Simulate a contagion process on a contact network a.k.a. Person-Person graph G_{PP}. Report the metric. (see section 7.1 for details).

(a) **Input**:

- A strategy $\mathbb{S}$ comprising of the following parameters.
  (a) `GraphT` : The input person-person graph that models the Graph Concept.

(b) `DiseaseModelT` : A model of Contagion Concept. The state update mechanism is a part of the model and not a separate type.

(c) `ContagionFunctorT` : A model of Contagion Functor Concept that defines the mechanism of contagion between two individuals (i.e., location based contagion or activity based contagion or any other type).

(d) `InterventionFunctorT` : A set of intervention functors that model the Intervention Concept and implements any of the interventions (Node or Edge). (A separate functor is required for every type of intervention). The functors can be chained together by using any mechanism such as tuples, etc.

(e) `Algorithm` : A particular computer implementation of contagion diffusion that captures the computational aspects of standard contacts based contagion simulation algorithms such as EpiFast [7], etc., where contagion takes place when two individuals interact for some duration in a simulation phase.

(b) **Output**:

- A sequence of system states over time $S(\Delta t), S(2\Delta t), \ldots, S(n\Delta t)$. Each state $S(\Delta t)$ represent the current health state of every individual in the contagion.
- The total interactions per second `TIPS` metric (see 7.1 for more details). The `TIPS` metric characterizes $\mathbb{S}$. We will touch upon characterization in chapter 7.

Requirements

1. **`GraphT`** : Must be a model of Graph concept and provide iterator style access to vertices, edges and their properties. Graph must not be mutable as we do not modify the underlying physical memory structure of the graph. However, the entire system being dynamic in nature, where the contagions are probabilistic and are affected by intervention strategies, logical deletion of vertices and edges (such as filtering out those vertices and edges that do not take part in contagion process) is possible.

2. **`DiseaseModelT`** : The state update mechanism must be defined by the input disease model where a person's health state is transitioned to a new state, based upon the current state and other factors such as interventions, etc. The sequence of update and low level details is defined by the Algorithm. Thus, we separate update mechanism from the implementation (or algorithm). It is responsible for simulating within-host disease progression mechanism.

3. **`ContagionFunctorT`** : The contagion functor is required for simulating between-host disease propagation mechanism between two individuals.

4. **`InterventionFunctorT`** : Intervention strategies are parameters of benchmark and must be predefined. For example, randomly choose any infected person and disable all its contacts (edges) for some simulation intervals unless activated again by a switch.

5. **`AlgorithmT`** : The Algorithm is a concept that is required to run the contagion on multiple platforms such as shared memory, distributed memory, hybrid (CPU-GPU), offloading to co-processors (Intel MIC). They aim to capture the computation aspects and data access patterns such as load-store, peak CPU performance, CPU throughput, etc. The benchmark must also provide a characterization of the algorithm in this sense.

4.3 Computational Expressions

We present basic computational expressions for each kernel using set based notations and concepts described in the previous chapter.

4.3.1 Kernel 0

Input

$P := \{p \mid p \in \mathcal{A}_P\}$ *set of persons*
$L := \{l \mid l \in \mathcal{A}_L\}$ *set of locations*

Output

$E_{PL} := \{(p, l) \mid (p \in P), (l \in L)\}$ *set of person-location pairs*

Expressions

$E_{PL} := \phi$
$E_{PL} \leftarrow E_{PL} \bigcup (p, l) \text{ where } (p \in P), \ (l \in L)$ *randomly pair a person with a location*

4.3.2 Kernel 1

Input

(1) $P := \{p \mid p \in \mathcal{A}_P\}$ *set of persons*
(2) $L := \{l \mid l \in \mathcal{A}_L\}$ *set of locations*
(3) $E_{PL} := \{(p, l) \mid (p \in P), (l \in L)\}$ *set of person-location pairs*

Output

(1) $G_{PL} := (V, E)$ *person location graph*
(2) $V := \{v \mid v \in V\}$ *vertex set*
(3) $E := \{e \mid e = (v_p, v_l) \in E \text{ where } p \in P, l \in L\}$ *person location edge set*

Expressions

(1) $V := V_P \bigcup V_L$ *union of person and location vertex set*
(2) for each $p \in P$ do
(3) $\quad v_p :=$ initialize person vertex
(4) $\quad V_P := \{v_p \mid p \in P\}$ *add person vertex in vertex set*
(5) od
(6) for each $l \in L$ do
(7) $\quad v_l :=$ initialize location vertex
(8) $\quad V_L := \{v_l \mid l \in L\}$ *add location vertex in vertex set*
(9) od
(10) for each $e \in E_{PL}$ do
(11) $\quad e_{(p,l)} :=$ initialize edge
(12) $\quad E := \{e_{(p,l)} = (v_p, v_l) \mid v_p \in V_P, v_l \in V_L\}$ *add person-location edge in edge set*
(13) od

4.3.3 Kernel 2

Input

(1)	$P := \{p \mid p \in \mathcal{A}_P\}$	*set of persons*
(2)	$E_{PP} := \{(p_i, p_j) \mid (p_i, p_j \in P)\}$	*set of person-person pairs*

Output

(1)	$G_{PP} := (V, E)$	*person-person graph*
(2)	$V := \{v \mid v \in V\}$	*vertex set*
(3)	$E := \{e \mid e = (v_{p_i}, v_{p_j}) \in E$ where $p_i, p_j \in P\}$	*person-person edge set*

Expressions

(1)	$V := V_P$	*person vertex set*
(2)	for each $p \in P$ do	
(3)	$v_p :=$ initialize person vertex	
(4)	$V_P := \{v_p \mid p \in P\}$	*add person vertex in vertex set*
(5)	od	
(6)	for each $e \in E_{PP}$ do	
(7)	$e_{(p_i,p_j)} :=$ initialize edge	
(8)	$E := \{e_{(p_i,p_j)} = (v_{p_i}, v_{p_j}) \mid v_{p_i}, v_{p_j} \in V\}$	*add person-person edge in edge set*
(9)	od	

4.3.4 Kernel 3

Input

(1) $V_L := \{v_l \mid (l \in L), (v \in V)\}$ *set of location vertices in person-location graph*

Output

(1) $LG := \{g_i \mid g_i \subseteq V_L\}$ *Set of loc. groups, each group contains at least one location*

Expressions

(1) $LG := \{g_i \mid i \leq n, n \in \mathbb{N}\}$
(3) for each $v_l \in V_L$ do *for each location vertex in location vertex set*
(4) $g_i := \{v_{l_i} \mid v_{l_i} \in V_L, i \in \mathbb{N}\}$ *assign a location to group i*
(5) od

4.3.5 Disease Model

(1) $D_m := \{S, F, W\}$ *Disease Model*
(2) $S := \{s_i, i \in \mathbb{N}\} = \{\texttt{sus, exp, inf, rec}\}$ *finite set of disease states (SEIR Model)*
(3) $F := \{f_i \mid \forall\, v \in V, f_i(s_v(t)) \rightarrow s_v(t+1)\}$
(4) *set of local transition functions that map current state to next state for every person vertex*
(5) $W :=$ string $\{v_1, v_2, ..., v_n\}$ *schedule representing order of state update for each vertex*

4.3.6 Update Functor

(1) $P_s := \{(v, s) \mid v \in V, s \in S\}$ *set of vertices and their current states*
(2) $P_{s'} := \{(v, s') \mid v \in V, s' \in S\}$ *set of vertices and their new (update) states*
(3) $U_f := V_s \rightarrow V_{s'}$ a mapping where
(4) $\forall\, (v, s) \in P_s,\ \exists\, (v, s') \mid s' = f_v(s)$ *for each vertex, state pair*
(5) *new state of vertex is derived based upon local transition function f*

4.3.7 Intervention Functor - Vertex

(1) $R := \{(v_i, r_i) : i = 1, 2, ..., n\}$ *set of person vertices and their health attributes*
(2) $R' := \{(v_i, r'_i) : i = 1, 2, ..., n\}$ *set of person vertices and with updated health attributes*
(3) $I_f := R \to R'$ mapping where
(4) $\quad \forall\, r \in R,\ \exists\, r' \in R' \mid r' = I_f(r)$ *for each person, attribute pair*
(5) *there is an updated (or intervened) attribute for each person defined by functor*

4.3.8 Intervention Functor - Edge

(1) $R := \{(v_i, l_j, r_{ij}) \mid (v_i, l_j) \in E\}$ *set of person-location edges and their edge attributes*
(3) $R' := \{(v_i, l_k, r_{ik}) \mid (v_i, l_k) \in E\}$ *set of new person-location edges with updated attributes*
(4) $I_f := R \to R'$ mapping where
(5) $\quad \forall\, r \in R,\ \exists\, r' \in R' \mid r' = I_f(r)$ *for each person-location, attribute pair*
(6) *redirect the edge to other location*

4.3.9 Contagion Functor - Location based

Input

(1) $D_m := \{S, F, W\} = \texttt{SEIR}$ *Disease Model for Contagion spread*

Expression

(1) for $l \in g_i \mid g_i \subseteq LG$ do
(2) $\quad A := \{(v_p, l) \mid (v_p, l) \in E, t_{\text{visit}} = (t_{\text{start}}, t_{\text{end}})\}$ *find the person-location activity set*
(4) $\quad$ for $(a_i, a_j) \in A$ do *for **any** two activities in activity set*
(5) $\quad\quad t_{\text{overlap}} := (t_{i_{\text{start}}}, t_{i_{\text{end}}}) \bowtie (t_{j_{\text{start}}}, t_{j_{\text{end}}})$ *find duration of overlap*
(6) $\quad\quad$ Propagate$(\texttt{SEIR}, v_{p_i}, v_{p_j}, t_{\text{overlap}})$ propagate the disease
(7) $\quad$ od
(8) od

4.3.10 Contagion Functor - Contact based

Input

(1) $D_m := \{S, F, W\} = \texttt{SEIR}$ *Disease Model for Contagion spread*

Expression

(2) for $v_p \in V \mid p \in P$ do — for each person in person set

(3) $N = \{(v_{p_i}, v_{p_{\text{inf}}}) \mid (v_{p_i}, v_{p_{\text{inf}}}) \in E, t_{\text{contact}} = (t_{\text{start}}, t_{\text{end}})\}$ *find person-person contact set*

(5) for $(v_{p_i}, v_{p_{\text{inf}}}) \in N$ do *for each contact*

(6) $t_{\text{duration}} := (t_{i_{\text{end}}} - t_{i_{\text{start}}})$ compute duration of contact

(7) Propagate$(\texttt{SEIR}, v_{p_i}, v_{p_{\text{inf}}}, t_{\text{overlap}})$ propagate the disease

(8) od

(9) od

4.3.11 Kernel 4

Input

(1) $P, L, LG, G_{PL}, D_m, U_f, C_f, I_f$
(2) $(t, n) :=$ simulation period t divided into n phases

Output

(1) $\mathcal{S} := \{S(t_0), S(t_1), S(t_2), ..., S(t_n)\}$ *overall system state for each phase*

Expressions

(1)	for $i := 1$ to n step 1 do	*for **each** simulation phase in equal step size*
(2)	for $g_i \in LG$ do	*for **each** location group in location group set*
(3)	for $l \in g_i$ do	*for **each** location in the location group*
(4)	Apply C_f	*apply contagion over **all** persons at the location*
(5)	od	
(6)	od	
(8)	for $v_p \in V : v \in V, p \in P$ do	*for **each** person vertex*
(9)	$U_f(v_p)$	*update its current state based upon transition function*
(10)	od	
(12)	for $v_p \in V : v \in V, p \in P$ do	*for **any** person vertex*
(13)	$I_f(v_p)$	*apply node interventions*
(14)	od	
(16)	for $(v_p, v_l) \in E : p \in P, l \in L, v_p, v_l \in V$ do	*for **any** person-location activity edge*
(17)	$I_f(v_p, v_l)$	*apply edge interventions*
(18)	od	
(19)	od	*repeat for every simulation phase till the end*

4.3.12 Kernel 5

Input

(1) $P, G_{PP}, D_m, U_f, C_f, I_f$

(2) $(t, n) :=$ simulation period t divided into n phases

Output

(1) $\mathcal{S} := \{S(t_0), S(t_1), S(t_2), ..., S(t_n)\}$ *overall system state for each phase*

Expressions

(1) for $i := 1$ to n step 1 do *for **each** simulation phase in equal step size*

(3) $\texttt{INF} = \{v_i \in V \mid s_{v_i} = \texttt{inf}\}$ *compute infected set of person vertices*

(5) for $v_i \in \texttt{INF}$ do *for **each** infected person*

(6) if $\exists(v_i, v_j) \in E, j \neq i, s_{vj} \neq \texttt{inf}$

(7) *if there is a contact edge with other person who is not infected*

(8) then $C_f(v_j)$ *apply cntagion functor to vertex*

(9) fi

(10) od

(12) for $v_p \in V : v \in V, p \in P$ do *for **each** person vertex*

(13) $U_f(v_p)$ *update its current state based upon transition function*

(14) od

(16) for $v_p \in V : v \in V, p \in P$ do *for **any** person vertex*

(17) $I_f(v_p)$ *apply node interventions*

(18) od

(20) for $(v_{p_i}, v_{p_j}) \in E : p_i, p_j \in P, v_{p_i}, v_{p_j} \in V$ do *for **any** person-person contact edge*

(21) $I_f(v_{p_i}, v_{p_j})$ *apply edge interventions*

(22) od

(23) od *repeat for every simulation phase till the end*

Chapter 5

Lifting Sequential Algorithms for Shared and Distributed Memory Machines

In this chapter we describe how we *lift* away the implicit requirements of sequential execution to parallelizing the contagion process in context of shared and distributed memory architectures. For shared memory, where the data is common to threads on a multiprocessor, we would only require avoidance of race conditions and also demand consistent state of the underlying data structure used in the benchmark. Concurrent data structures are necessary in this case, whereas for distributed memory, we would require introduction of external (i.e., distributed) data structures for parallel execution. The lifting process would retain the interface and semantics of the generic algorithm such that distributed and shared memory algorithm can be built upon it. The end goal is to develop an efficient and scalable generic algorithm.

5.1 Lifting

Generic programming is a systematic approach for classifying entities within a problem domain according to their underlying behavior and semantics [13]. Emphasis on semantic analysis leads to collecting the essential or minimal properties of components and their interactions so that by defining interfaces for these minimal requirements, we are able to achieve maximal code reuse. The process of *lifting* aims to discover a generic algorithm by enlisting the minimal requirements that our data types need to fulfill so that our algorithm operates correctly and efficiently. In [13,35], the authors describe a basic example of *lifting* using the `sum` algorithm. By separating out the *storage* of values (i.e., containers) from **access** to these values (i.e., iterators), one can write algorithms that

operate on any container. The process is summarized in Figure 5.1.

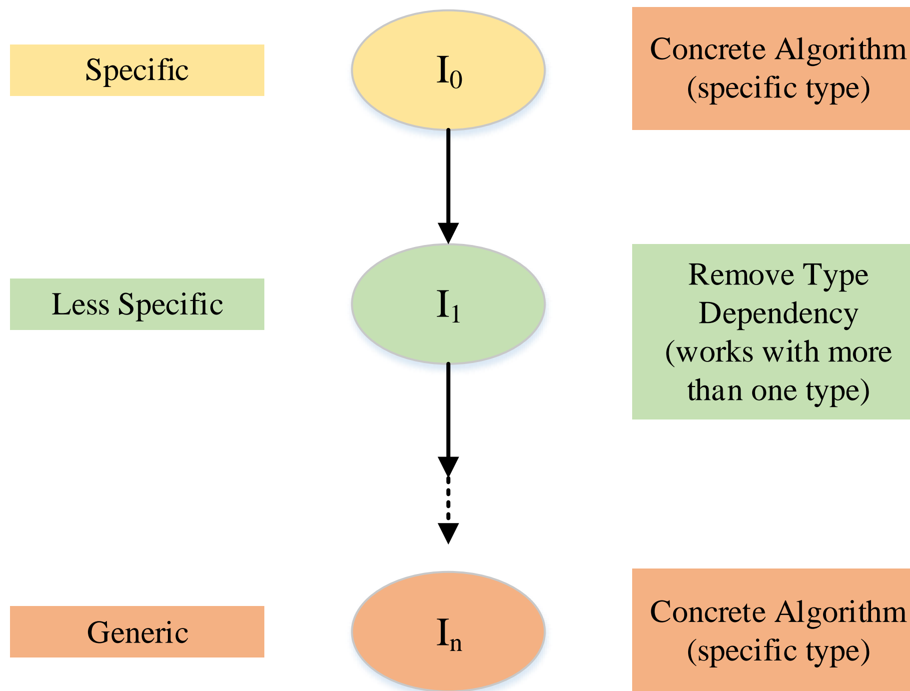


Figure 5.1: Lifting a concrete algorithm to generic algorithm: (i) enlist the trivial or concrete algorithm, (ii) remove the unnecessary requirement on *type*, (iii) after iterations, make the algorithm work for a maximal family of types.

5.2 Lifting for Shared Memory Parallel Computing

Our process of lifting the contagion diffusion functor for shared memory parallel algorithm is fairly straightforward. For parallelizing any algorithm, we must ensure its correctness. A parallel algorithm must yield the same expected result (or rather manifest the same behavior) as compared with its original sequential counterpart.

Consider a SEIR disease model [9] $\mathcal{D}$ with four disease states: S : Susceptible, E : Exposed, I : Infected, R : Recovered. The model has fixed transition paths from susceptible to recovered in

the stated order. At any instant of time, an individual's health state is one amongst the four states. Also, assume a Person-Location interaction network where individuals meet other individuals at common location such as home, office, etc. An individual's health state changes are deterministic in the sense that, based upon the individual's current state, we are able to compute its next state using local transition functions f_i that are probabilistic and timed [1,7]. Also assume that there is a latent period $d_{\min}$ which is the amount of time that must pass between a person becoming infected and a person being able to infect others. If the time period to be simulated is divided into n phases, and the length of single simulation phase is less than $d_{\min}$, then all locations can be *concurrently* processed and interactions between individuals at these locations can be simulated in parallel [1]. This forms the basis for parallelization of our simulation algorithm in kernels 4 and 5. Typically, length of simulation interval is set to one day and $d_{\min}$ is greater than one day. This would be required for our parallel algorithm to be correct. In [1], the authors demonstrate the correctness of this approach with rigorous proof. We, however, use this argument as our basis for the lifting process.

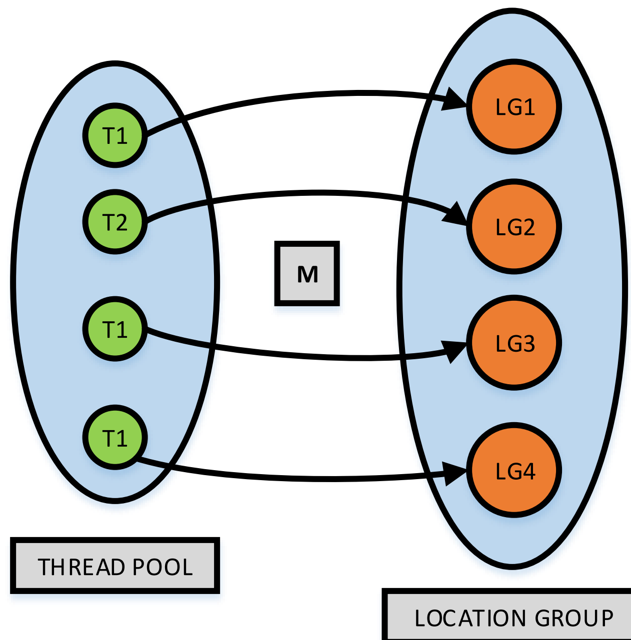


Figure 5.2: Thread Pool to Location Group mapping. Every thread T_i is assigned a location group LG_i

Let us assume that we have n threads at our disposal. For any Person-Location graph $G(V, E)$ with the location vertices grouped into m location groups (as described in kernel 3), define a mapping

function $\mathbb{M}$ from n threads to m location groups which maps a thread to one or more location groups i.e. a thread simulates contagion over one or more location groups. $\mathbb{M}$ is a *one-to-many* function from set of threads $\mathcal{T}$ to set of location groups $\mathcal{LG}$ defined as $\mathbb{M} : \mathcal{T} \rightarrow \mathcal{LG}$, where every thread $t_i \in \mathcal{T}$ maps to one or more location groups lg_j as explained in Figure 5.2.

Assuming number of threads would be far less than number of location groups, each thread would have more than one location group assigned to it for simulating the contagion process (see Figure 5.3 for kernel 4). Figure 5.4 explains the thread mapping for kernel 5. Since we do not have locations in kernel 5, we group all the infected people present in a simulation interval into infected groups and then map every thread to one or more groups. A task is created for every infected person in the group and then assigned to the threads (similar to that explained in 5.3).

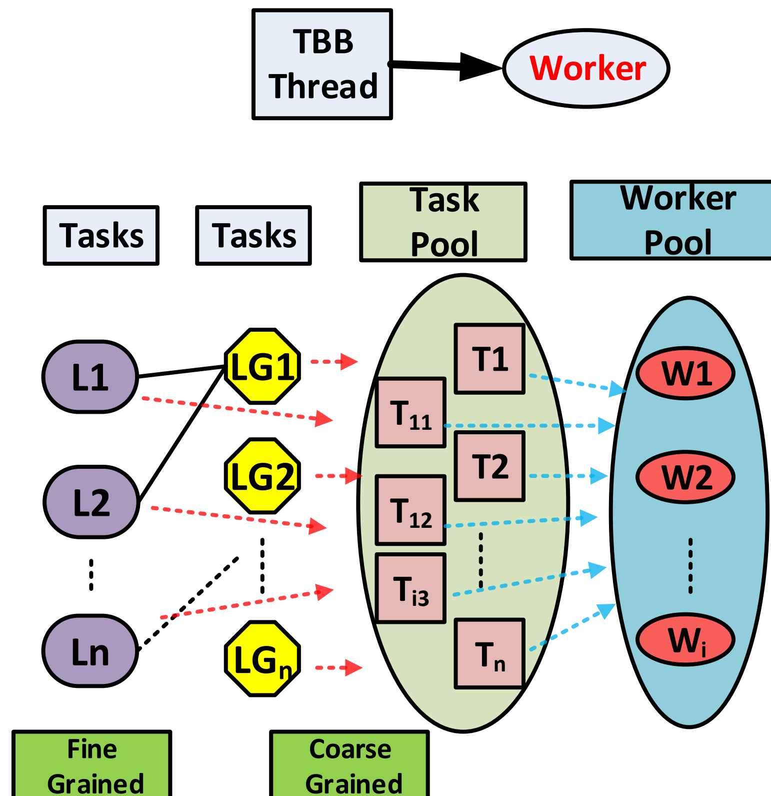


Figure 5.3: Shared Memory: Location Groups (LG_i) are mapped to tasks T_i to form a Task Pool. A Worker (W_i) (i.e., a thread) is assigned a task T_i from the pool (blue arrows). A worker can further create additional tasks T_{ij} (by mapping Locations L_i onto the task pool (read arrows)). These tasks are also executed by workers.

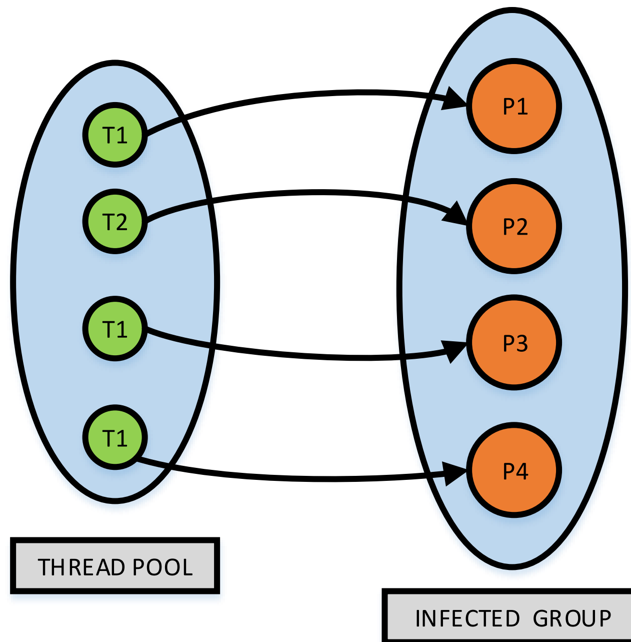


Figure 5.4: Shared Memory: The Thread Pool (T_i) to Infected Group (P_i) mapping. Every thread handles one infected person (task) and computes contagion for those people in contact with that person.

5.2.1 Generalizing the concept

To generalize the group concept, we abstract the requirement of assigning threads to location groups or infected groups and present a generic methodology for mapping. This can be useful when there is a hierarchy of groups involved in the simulation. Figure 5.5 explains our generalization of group concept which is prime requirement for the *lifting* process.

For example, a city consists of several blocks and every block contains several other locations such as offices, schools, homes etc. We can group similar locations to form a small sub-group, then group these into a block sub-group and further up the chain, group the blocks to form a city. We can assign a thread for one or more blocks and repeat the process down the hierarchy. For the time being, we ignore the intricacies of the compute architecture such as max threads spawned and focus on the idea of *lifting*.

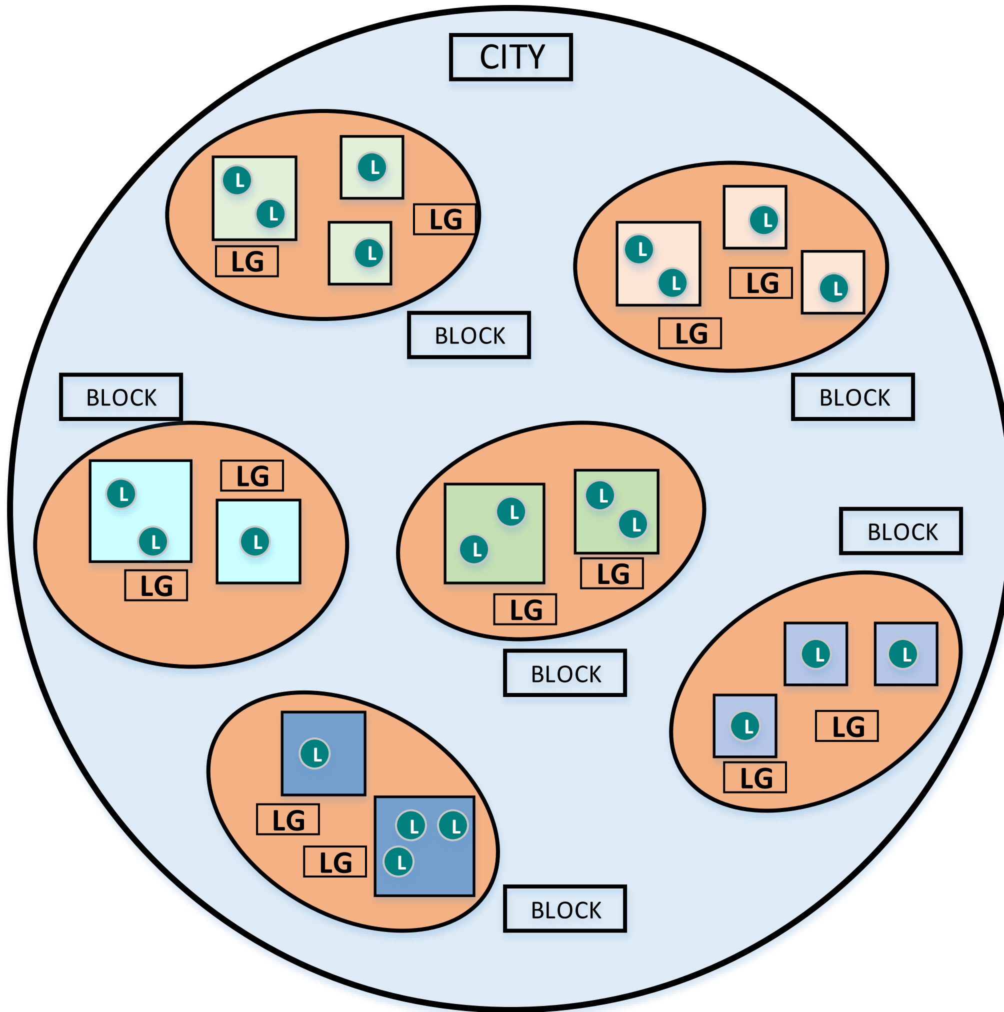


Figure 5.5: Location Hierarchy – City, Blocks, Location Groups, Locations

5.2.2 Performance Implications of Mapping Threads

Here we use the term group as a generic group (location or infected or hierarchical) i.e., assume that a thread is assigned one or more *group(s)*. One might want to argue about how efficient or ideal the mapping of threads to groups must be so as to achieve optimal and maximal performance from the compute system. It may happen that some groups may have less entities while some have more, thereby putting more workload on some threads while others remain idle.

Our proposed lifting concept takes into account *efficiency* as one of the requirements. It requires

a common thread pool pattern where (predefined number of) threads are created to perform a certain number of tasks, typically organized in a queue like data structure (thread safety is implied). Priority of tasks is not a requirement as all tasks are similar in nature i.e., to simulate contagion process over a group. As soon as a thread completes its assigned task, it will request the next task from the queue until all tasks have been completed. The thread can be then terminated, or made to sleep until new tasks are available in the task queue.

The number of threads can be tuned (typically one-to-one mapping between a thread and a processor core) to achieve optimal performance. We also do not require dynamic number of threads as a requirement in lifting. The algorithm chosen will depend on the problem and the expected usage patterns. Based on this we now present requirements for lifting our contagion process for shared memory parallel computing platform.

1 For an n phase simulation, the length of single simulation phase must be less than a latent period. Latent period is the time when an individual's state changes from one state to another.

2 Identify and extract total contagion tasks for every simulation phase (such as contagion at every location group).

3 Initialize a thread pool with required number of threads (usually less than number of tasks)

3 Assign the contagion tasks to idle threads in a thread pool until all tasks get executed.

4 Repeat the step for every simulation phase.

5 Ensure container and data consistency (such as concurrent accesses and data modification).

The last point emphasizes about container and data consistency. Often we need to count the number of exposed persons in every simulation interval. An array `arr` of size equal to the number of intervals would store the desired values. For every simulation phase i, the `arr[i]` would give us the count for that phase. In sequential case, this number is updated in a serial fashion. However, in shared memory where several threads may update this value, we need to ensure the safety of this container. So either we need to make this container thread-safe or we must modify our algorithm in such a way that the sequential and parallel implementation remain the same. Either of the two would suffice.

In our implementation (see chapter 6), we make clever use of STL's `count_if` and `count` algorithm to count such values. This is advantageous in two ways:

1 The task of counting and task of simulating contagion are mutually exclusive (Single Responsibility Principle [52]).

2 We can replace the `count_if` algorithm by a `parallel_count_if` algorithm (say) and the lifting would still valid in this case.

5.3 Ensuring Correctness for Parallel Lifting

Parallelizing an algorithm for shared memory address space involves lifting the requirements for single thread of execution. In the case of embarrassingly parallel algorithms, lifting this requirement is sufficient since the address space is still shared across multiple processors (or cores). However for vast majority of practical algorithms we need to introduce new requirements to ensure correctness for parallel lifting. Some of these requirements for shared memory are, (i) avoiding data-races and (ii) preserving the invariant for the given algorithm.

Consider the case for lifting sequential kernel 4 algorithm. The kernel simulates the spread of contagion in a person-location network where the locations are the prime medium of propagation of the disease. Two or more individuals come in contact at a common location and interaction takes place. Thus the kernel 4 will propagate contagion at each location sequentially, with a requirement that, after all the locations are processed in a given simulation phase, each of the person vertices must undergo a state update mechanism that is characteristic of the disease. This requirement was justified in section 5.2. It can be easily noted that there is no requirement that mandates a fixed ordering for processing location vertices. A person visiting more that one location in a simulation phase may get contagion from multiple locations, but his current state is updated only at the end of simulation phase (when all the locations are processed). Taking the advantage of this, we can process all the locations in parallel. Even though there can be data race on the memory address of state variable of the person, we ensure that threads do not update the address instantly, but delay this until all contagions are completed. In a similar manner, if the state update mechanisms defined for every individual are independent of each other, the we can achieve parallel update. In a similar fashion interventions can also be processed in parallel depending upon what type of intervention we want to use. A `for` loop over location groups is replaced by a `parallel_for` and the parallel implementation retains the behavior of sequential implementation. The following tasks must be serialized in a given simulation phase in the order as stated, but each of the individual tasks may be processed in parallel: (i) contagion of locations, (ii) state updates of individuals (if mechanism is independent), (iii) Interventions (if they are independent). It can be noted that based upon the Disease model and contagion requirements, the parallelization point or the invariant may

change accordingly and hence one may not be able to achieve complete parallelization if there is interdependence between entities. But in our case, we assume that every individual's sequence of state updates are known and are independent of other's states, also we assume that at a given location, the contagion process involves only two interacting individuals. Hence this strategy for disease model gives us a good opportunity of extracting parallelism as much as as possible.

We state the following requirements for kernel 4. The requirements for kernel 5 can be easily derived based upon these requirements.

1. The contagion process on all locations must be completed before a state update mechanism is run over all the persons.
2. If a person's state can be updated without the dependence on other persons and their states, then update mechanism can also be run in parallel.
3. For Interventions, the same case applies as per (2).
4. One cannot run a contagion over a network by parallelizing the simulation phases (time) since these processes are spatio-temporal and evolve over time, input for one simulation phase depends upon output of its penultimate phase.

5.4 Lifting for Distributed Memory

Here we present only an idea about how we can achieve lifting our sequential algorithm for distributed memory parallel computing. Implementation details and performance results are beyond the scope of this report and would be included in the future work to extend this benchmark for distributed memory architectures. We assume MPI [36] as de-facto interface for distributed memory parallel computing. The idea can me extended to other parallel computing platforms such as Charm++ [50].

For our distributed memory parallel computing, the data for computation is stored on different machines and thus it is evident that we require distributed data structures for running our algorithm. Our algorithm implementation however does not change in this case, but it operates over a subset of data which is stored on a compute node. We only require two modifications in data structure types viz. Graphs and Property Maps. Rest of the requirements are implicit and can be taken care of. [13] provides more details on how a generic sequential *BFS* algorithm is lifted for distributed memory

parallel computing merely by introduction of distributed data structures such as Distributed Graph (adjacency_list) and distributed property maps. The algorithm remains the same.

In our case, the contagion algorithm runs only on the graph and other essential data which is present on that compute node. A message passing interface helps facilitate efficient exchange of messages between two MPI processes. Rest of the idea can be interpreted from Figure 5.6. A single MPI process runs on a multi-core compute node. The process constructs its own list of local tasks and also creates a local thread pool, where every thread will be assigned a task. Message passing between processes for state updates, local infection count, etc. is handled by the individual MPI process, while the contagion simulation is handled by the threads. Rest of the figure is self explanatory.

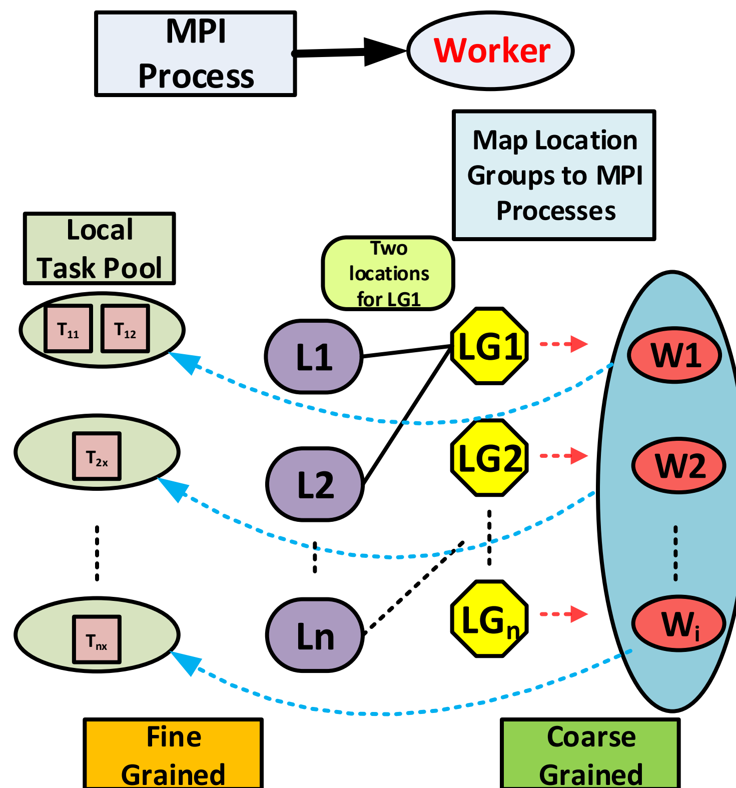


Figure 5.6: Distributed Memory: A Worker (MPI) (W_i) process is created on every compute node and is assigned a Location Group (LG_i) (typically a sub-graph of main graph) for the task. The MPI Process spawns a local Thread Pool (T_i) (per compute node) where each thread handles a local task as assigned.

Chapter 6

Implementation

6.1 C++11 Features

In our present implementation of generic kernels, several C++11 language and standard library features have been used. Some of them have been briefly described below.

6.1.1 auto

In the C++11 standard, *auto* keyword is used for type inference i.e. deducing the type of a declared variable from its initializer. This has been changed from previous ISO standard where auto was used as a storage class specifier. It can be used when declaring variables in block scope, namespace scope or initialization statements of for loops. In template functions returning arbitrary values, auto can be used to specify that the return type is a trailing return type. Given below is an example showing the usage of auto.

```
// Determines the variable type
auto i = 1.2;              // i is a float
auto i = "Test String";    // i is a string
auto i = new func();       // i is a func*

// Determines the type based on the type of first element of
// a vector
vector<int> V;
for (auto it = begin(V); it != end(V); ++it) // it is a vector<int>::iterator
```

```
10 {}
```

6.1.2 decltype

In the C++11 standard, *decltype* is used for either deducing the type of an expression or to make a typedef for the type of an expression. The latter is used when instead of declaring a variable whose type is that of an expression, we want to make a typedef for that type. Given below is an example showing the usage of decltype in both the instances.

```
1 //Deducing type of an expression
2 decltype(10) myInt= 10;
3 decltype("Test String") myString= "Test String";
4
5 // Typedef the return type whose type is the product of the
6 // two arguments.
7 template<typename A, typename B>
8 void func(A a, B b) {
9   typedef decltype(a * b) prod_type;
10   ...
11 }
```

6.1.3 Range-based For loops

The *For* statement in C++11 has the additional support to iterate over a range of values such as all elements of a collection/array, initialization lists, or any other thing for which begin() and end() functions are overloaded. Using a range-based for loop, we can either iterate over a container or modify the container elements. For modifying the values in a collection, & needs to be supplied to auto as shown in the example below.

```
1 //Simple iteration over a vector
2 vector<int> V={1,2,3,4,5};
3 for (auto v: V) cout << v << ","; // 1,2,3,4,5
4
```

```
//Modifying the elements of a vector
for (auto& v: V) v *= 2;
for (auto v: V) cout << v << ","; // 2,4,6,8,10
```

6.1.4 Function Objects a.k.a Functors

Function object or *functor* is an object that behaves like a function and was introduced even before C++11. We can define a function object as a class instance that provides a function call operator. To be more specific, in function objects the "()" operator is overloaded. A function object can be defined as:

```
Class func_obj {
    Public:
        // define "function call" operator:
        <return-type> operator() (args) const;
          ....
};
```

We can create objects of the class *func_obj* which can be called like normal functions as:

```
func_obj f1; // create function object
f1(arg);    // call of operator () for function object f1
```

There are some advantages of using function objects such as:

1. Function objects have a state because they may contain member functions and variables. Hence, two function objects, which represent the same functionality, may have different states at the same time.

2. Each function object has its own type which proves helpful in generic programming. This behavior enables containers of different types to use the same kind of function object as, for example, a sorting criteria.

3. Function objects allow better optimization and hence are usually faster than ordinary functions.

```
1  //Myfunc is a function object
2  template <class Graph_t>
3  struct ContagionFunctor {
4   public:
5    ContagionFunctor(const Graph_t& G) : m_G(G) { }
6    template <typename vd_t>
7    void operator()(const vd_t&  vd) const {
8        //process contagion for every non-infected vertex
9        //which is in contact with infected vertex
10       //i.e. person-person edge
11   }
12  private:
13   Graph_t& m_G;
14 };
15
16 //get vertex identifier
17 typedef typename boost::graph_traits<Graph_t>::vertex_descriptor vd_t;
18
19 //collection of infected persons
20 VertexContainer<vd_t> InfVertices;
21
22 //apply the functor for each infected vertex
23 std::for_each(begin(InfVertices), end(InfVertices),      //apply the functor to
24                ContagionFunctor<Graph_t> (G);            //each infected vertex
25
26 //Using C++11 Lambdas
27 std::for_each(begin(InfVertices), end(InfVertices),
28            [&G](vd_t  vertex) -> void {               //Capture G by reference
29                //process contagion for every
30                //non-infected vertex which is
31                //in contact with infected vertex
32                //i.e. person-person edge
33            }
```

6.1.5 Lambdas a.k.a Anonymous Functors

Lambdas are *anonymous* functions that have body but no name. Lambdas define a function object and constructs a function object class of that type but with no name. The C++11 standard supports

Lambdas and thus are crucial in reducing number of lines in the code. They also make the code look clean and understandable due their inline nature. Refer the example in 6.1.4 where it can be seen that C++11 Lambda can considerably reduce the code and make it cleaner and precise.

6.1.6 Other Features

We have used other C++11 features such as:

- **`std::move`** : move constructor for passing `rvalue` references from one kernel to other. For example, a graph object constructed in kernel 1 is passed to kernel 4 for contagion simulation using C++11 move semantics.
- **`std::bind`** : To bind a function with any number of arguments by providing placeholders _1, _2, ... This is important particularly for computing the probability of infection between two individuals where the **placeholder** elements are person vertices, while the function remains the same. A simple example of bind is given below.

```
template<typename vd_t>
double ComputeProbability(const vd_t& v1, const vd_t& v2) {
  //compute probability
  return prob;
}

auto func = std::bind(ComputeProbability, _1, _2)    //returns a function

//use the function for in contagion
for (const auto& v1 : infected_set) {
  for (const auto& v2 : susceptible_set) {
    auto inf_prob = func(v1,v2);                     //get probability
```

- **Algorithms** : We use few algorithms included in the C++11 standard: `std::copy_if`, `std::shuffle` and `any_of`
  - **`std::copy_if`** : Copies an element into a container if it satisfies a predicate.
  - **`std::shuffle`** : Shuffles the elements in a container. Particularly it is useful in kernel 0 while pairing person and location labels.

- **std::any_of** : Returns a boolean if any of the elements in a container satisfies a predicate. This is very important when we want to know if there is any infected person present at a location. If there are no infected persons at that location (meaning no such person-location edge in the graph), then we can safely omit the location from contagion process as there is no medium for contagion at that location.

- **Containers** : We used `std::unordered_set` (or a hash-set) and `unordered_map` (or hash-map) which are new types of C++11 containers. These containers have constant time element access which makes them superior over their old counter parts which have $O(\log n)$ complexity. We use such containers for intermediate storage for vertices and edges. We can tweak the container's default hashing template by our custom hash function to achieve performance gain.

6.2 Boost Graph Library

Our implementation of the `GraphT` concept is based upon Boost Graph Library [8] interface. The BGL graph interface is generic in the same sense as the Standard Template Library [49]. Our implementation of `boost::adjacency_list` in the `Boost.Graph` satisfies all the Graph Concepts we stated in chapter 3.5. We also use the Boost Property Map library for storing vertex and edge properties. The Property Map library supports generic `get` and `put` operations and uses *tag dispatching* for resolving overloaded function calls. For more details, please refer [8].

6.3 Containers

For our parallel implementation, we extended the data structures used for storing vertices and edges by adding concurrent data structures and extending the boost graph container selector structure. The `Boost.Graph` has interface for selecting the container types for storing vertices and edges (such as list, vector, set etc.). Using the concurrent and thread safe data structures from the Intel TBB library [11], we modified the selection criteria for choosing data structures for storing vertices and edges. This is crucial in shared memory parallel computing because STL containers are not thread safe and hence concurrent read/write access by threads might ruin the data. The following container selectors were added to the boost library namespace:

- **boost::concurrent_vectorS** : A selector to choose Intel TBB's concurrent vector container.

- **boost::concurrent_unordered_mapS** : A selector to choose Intel TBB's concurrent hash map container.
- **boost::concurrent_setS** : A selector to choose Intel TBB's concurrent hash set container.
- **boost::concurrent_queueS** : A selector to choose Intel TBB's concurrent queue container.

6.4 Compilers - GCC and ICPC

We use GNU GCC 4.7.2 and Intel C++ Compiler 15 (2015) (with C++11 support) for our implementation. Here, we discuss two runtimes used for our parallelization strategy – Intel Threading Building Blocks (TBB) and Intel Cilk Plus (Cilk) and also about Intel MIC features.

6.5 Intel TBB

The Intel Threading Building Blocks [11] version 4.2 was used for implementing our task based parallel implementation of kernel 4 and 5. In TBB, the task-based parallelism is implemented in which multiple threads work in parallel without having to manage underlying threading details. Using TBB, we no longer have to think about creating and deleting threads but only the task that needs to get done. TBB provides a robust set of components for parallelism including templates based on common programming algorithms, data-structures and synchronization primitives. It also includes an optional scalable memory allocator to prepare our programs for parallel demands on memory. When we use TBB its run-time library runs in the background, managing all trading details such as scheduling and load balancing.

Here, a simple example of TBB syntax which shows two versions of a simple for loop that goes through an array and applies a function to each array item.

```
1 void serial_funct() {
2    // Serial Version
3    for (int i=first; i<last; i++) {
4       temp[i] = func(temp[i]);
5    }
6 }
```

```
void parallel_funct() {
   // Parallel Version
   parallel_for (blocked_range<int >(first, last),
                   example);
}
```

The *serial_funct()* function in the first code snippet is for the serial version of code while *parallel_funct()* function in the second code snippet is a parallel version in which TBB **parallel_for** template has been applied. *Parallel For* is one of the easiest and most commonly used templates in TBB. It takes the work of a serial for loop and breaks it into tasks. At runtime, the TBB runtime library will distribute these tasks to the available threads insuring that the work stays as balanced as possible. The code in the second snippet is all that would be needed to implement TBB `parallel_for`. It may be observed that the parallel version is very similar to the serial one. This is because the `parallel_for` template is implemented using the lambda format. TBB `parallel_for` can be implemented in two ways: the lambda format or the operator format. The lambda format requires a lambda expression which are on-the-fly functions that are supported by the C++11 standard.

In addition to `parallel_for`, we also use a `parallel_for_each` template function for task based parallelism. The `parallel_for_each` is a variant of `std::for_each` STL algorithm with an added functionality of task based parallelism by creating a task for every iterator element in the iterator range. TBB supports a thread pool where each thread blocks until it is assigned a task by the runtime scheduler. The following is the syntax for `tbb::parallel_for_each)` template function.

```
template<typename InputIterator, typename Body>
void parallel_for_each( InputIterator first, InputIterator last,
                        Body body[,task_group_context& group] );
```

The `parallel_for_each` applies the function object **body** of type **Body** over a sequence [first,last). The arguments to the function object must not be mutable, since the same function object is shared amongst all the threads. The compiler would produce an error if this is not taken care of. Thus the functor is not allowed to modify its private members. Alternatively, we can use an inline anonymous lambda in place of a function object. This makes the code shorter and easier to understand. The following code snippet explains the lambda method for `parallel_for_each`.

```
template <class Graph_t>
struct ContagionFunctor {
  public:
    ContagionFunctor(const Graph_t& G) : m_G(G) { }        //constructor
    template <typename edge_t>
    void operator()(const edge_t& edge) const     //Operator must be 'const'
    {                                             //otherwise compiler error
        //do contagion
    }
 private:
   Graph_t& m_G;
};

EdgeIterator  begin, end;
boost::tie(begin,end) = edges(G)           //Get Edge iterators for graph G
tbb::parallel_for_each(begin, end, ContagionFunctor<Graph_t> (G);

//Using C++11 Lambdas and decltype

//get the type of edge iterator if we don't know the type
using value_type = typename std::decay<decltype(*begin)>::type;
tbb::parallel_for_each(begin, end, [&G](value_type  edge) const {
          //do contagion
}
```

6.6 Intel Cilk Plus

Intel Cilk Plus is an extension to C and C++ that offers a quick and easy way to harness the power of both multicore and vector processing [55]. The three Intel Cilk Plus keywords – `cilk_for`, `cilk_spawn`, `cilk_sync`, provide a simple yet surprisingly powerful model for parallel programming, while runtime and template libraries offer a well-tuned environment for building parallel applications. Cilk provides language extensions supporting fork/join and SIMD (Single Instruction Multiple Data) parallelism (like OpenMP). The `cilk_spawn` keyword causes the compiler to generate code to add an entry to the back of a queue of spawned tasks maintained for each processor. The entry is added before the call to the spawned function is made. When each processor has work to do, a `spawn` is roughly the cost of a call. When a spawned function

returns, the back entry is popped off the queue. The `cilk_for` however, converts a simple for loop into a parallel for loop where iterations of the `for` loop body can be executed in parallel. The `cilk_sync` statement specifies that all child functions spawned from a function must complete before execution continues past this statement. There is always an implied `cilk_sync` at the end of every function that contains a `cilk_spawn`. The Intel Cilk Plus compiler and runtime cooperate to divide the work of the loop in half, and then divide it in half again, until there are enough pieces to keep the cores busy, but at the same time minimize the overhead imposed by `cilk_spawn`. Environment variables such as `CILK_NWORKERS` help the runtime to set the number of workers for execution.

6.6.1 STL Container Support

Since Cilk is based upon expressing or exposing parallelism to the compiler by providing keywords as hints, inherently, the loops require a constant time increment operations on iterators for containers. This implies that for associative containers or tree structured containers such as `set` or `list` that do not have random-access iterators, and thus does not have an `operator-` or an `operator+=`, we cannot advance the iterator in constant time and hence cannot compute the distance between two iterators in constant time. A `cilk_for` loop is simply not intended to traverse linear or tree-structured containers like list or set. A solution for this is to iterate using the container index in `cilk_for` and then use the `operator[]` to access the element in the container. Another solution would be to use `cilk_spawn` in place of `cilk_for`, but we must ensure than the spawned function have significant work to keep the processor busy so that we might be able to amortize the cost of the linear traversal with a simple serial loop containing a spawn.

The following code snippets explain the use of Cilk in our implementations, some problems and their workarounds. We use `std::bind` to bind all of our functors with arguments and placeholders to obtain functions that can be used easily in cilk loops.

```cpp
template <class Graph_t>
struct ContagionFunctor {
 public:
  ContagionFunctor(const Graph_t& G) : m_G(G) { }
  template <typename edge_t>
  void operator()(const edge_t&  edge) const {
    //do contagion
  }
 private:
  Graph_t& m_G;
};

//bind the functor
auto function = std::bind(ContagionFunctor<Graph_t>(G),
                          std::placeholders::_1);

std::vector<vertex_id>  vertices; //container stores vertex ids

//cilk_for
cilk_for(auto beg=vertices.begin(), beg!=vertices.end(), ++beg) {
  function(*beg);  //works only with random access iterators
}

//cilk for - compiler error for vertex_set = std::set<>
cilk_for(auto beg=vertex_set.begin(), beg!=vertex_set.end(), ++beg) {
  function(*beg);  //std::set does not have random access iterators
}

//solution 1 : use container index
cilk_for(size_t i = 0; i < vertex_set.size(); ++i) {
  function(vertices[i]);  //use [] for set or list
}

//solution 2 : cilk_spawn and cilk_sync
for(auto beg=vertex_set.begin(), beg!=vertex_set.end(), ++beg) {
  cilk_spawn function(*beg);  //spawn the function
}
cilk_sync;
```

6.6.2 Intel MIC and Cilk

For compiling code for Intel MIC architecture, we cannot parallelize the loops even if the containers provide random access iterators because the co-processor does not support complex iterator operations and the Cilk compiler throws an error. Hence, all our parallel loops strictly require that we provide the container range using indices in the `cilk_for` and `cilk_spawn` loops and not using iterators (See solution 1 in above snippet).

6.7 Intel MIC

The Intel MIC architecture combines many Intel processor cores onto a single chip, called the co-processor. The co-processors (referred as MIC cards) reside on every compute node as add-on card on high bandwidth PCIe slots. Intel MIC, also known as Xeon Phi, provide GPU-style acceleration for highly-parallel tasks but offer more integration with CPUs and compatibility with existing CPU programming paradigms (C/C++, Fortran, etc) than traditional GPUs.

Xeon MIC cards can be used in a number of different ways. They run a version of Linux, so users can log into them and run tasks directly on them; this is known as "native" execution. Users can also run jobs on the CPUs on a node (the "host") and then use Intel's Math Kernel Library (MKL) or compiler directives to push highly parallel portions of the job to the MIC cards. This is known as "offload". MIC cards can also run a version of Intel MPI on them; however this capability needs to be enabled. MIC supports multiple runtimes such as OpenMP, TBB, Cilk that help programmers focus on exposing parallelism to the co-processor and the rest is taken care by the runtime.

Using Cilk and TBB runtimes, we were able to compile and run our benchmark kernels on Intel MIC architecture in native mode [48]. Preliminary results for Intel MIC are presented in the next chapter. An issue regarding compiling and running our code on MIC architecture using Intel Compilers has also been addressed in the Appendix A.

Chapter 7

Performance Metrics and Analysis

For evaluating a particular strategy (i.e a given combination of implementation, runtime and hardware) we propose the ***TIPS*** metric on the basis of which the simulation kernels are assessed. We will also present our experiment setup and preliminary results for kernels 4 and 5 for a given strategy.

7.1 TIPS

7.1.1 Total Interactions Per Second

The number of interactions in a given social network depends upon several factors. We will enlist them here, but before that we will define what an interaction means in our simulation environment. An interaction is said to occur between two individuals when they meet at a common place or location. Diseases spread across a social network through interactions amongst the individuals which are considered the prime or causal parameters that help in propagation. Obviously there are other factors such as the disease model, interventions, etc. However biological interactions amongst individuals is considered the prime cause. We therefore propose our core metric for evaluating the performance of our simulation kernels based upon the number of interactions the kernel is able to process per unit of time. The Total Interactions Per Second metric thus is calculated as follows:

$$TIPS = \frac{\text{Total no. of Interactions in graph}}{\text{Total Time required (for i threads)}} \tag{7.1}$$

In person-location graph, an edge is an activity that represents a visit of a person p to a location l and hence there is no direct interaction (or contact) between two individuals as it is in case of person-person graph. Thus the locations are the medium where individuals visit and interact with other individuals during a common interval (overlapping time phase).

We enlist several other factors that affect the total number of interactions in the simulation kernels:

- **Degree distribution** : If the locations have fewer people visiting (low degree distribution for locations), the number of interactions (and hence number of people getting contagion) is likely to decrease.
- **Interventions** : Applying edge interventions, lead to logical deletion (or deactivation) of all the edges for a location and thus the number of interactions would decrease considerably in the next simulation phase. For example, while modeling a school closure intervention, all the edges (children-school activities) would be deactivated and do not take part in the next simulation phase unless explicitly turned on (say after 5 days).
- **Algorithm** : Poor choice of algorithm implementation for simulating the contagion may result in superfluous results since it would unnecessarily increase the computations and result in misleading results.

To study the benchmark in depth, we also need to characterize the performance of kernels based upon the following factors. Our ongoing work is focused on evaluating these factors so that we may be able to achieve our desired goal – the kernels must capture the data access patterns and computational complexities of contagion diffusion and can easily lead to choosing the most appropriate data structure and a hardware-software-runtime stack for a given system.

- Bus Contention : In shared memory multiprocessor, all of the memory is accessible to all the cores at same time. This setup can cause tremendous trouble when the number of cores increase beyond certain point which is called the bus saturation. We expect to have maximum performance when the bus is fully utilized.
- Core Saturation : We need to compare the performance of our benchmark with theoretical peak performance of the cores in our test system in order to characterize and remodel the kernels which can help choose the right strategy for target architecture.
- There must be a characterization of data vs. compute intensive nature of the algorithm by means of using some profiling tools.

7.2 Experiment Setup

7.2.1 Data Generation and Other Parameters

Kernel 0 generates and outputs the list of person, locations and person-location activity pairs. We generate out-degrees for persons and pair with the generated indegrees of locations. We use Poisson Generator and Exponential generators for generating indegree (for locations) and outdegree (for persons) respectively. The kernel 1 and kernel 2 construct the person-location and person-person graphs (of type `boost::adjacency_list`). We use SEIR model as our Disease model parameter with parameters as explained in Figure 7.1. Also we use a filter iterator based Edge Intervention functor which filters out the deactivated edges (i.e. intervened edges). Node Interventions change the infectivity and susceptibilities as per set values.

Kernel 3 creates a location group container and adds each of the locations to every location group, and it returns an iterator over location groups. The kernel 4 simulates contagion over Person-Location graph (from kernel 1) while kernel 5 simulates contagion over Person-Person graph.

7.2.2 Effect of Input Parameters

We take a note of the effect of changing input parameters and data structures in our benchmark. As we know, any change in the implementation strategy $\mathbb{S}$ would yield all-together different performance results. Some of the factors affecting the performance are summarized below:

- `DegreeDistribution` : The number of person and locations generated and their degree distributions (outdegree/indegree) play a crucial role in deciding number of interactions in the contact or activity graph. Any parameter change would result in a new contact graph with different interaction characteristics.
- `DiseaseModelT` : Parameters in disease model include the incubation periods, infectivities, susceptibilities, transition functions (FSM) and state update mechanisms.
- `InterventionT` : Interventions affect the logical structure of the graph. Since the contagion diffusion (disease propagation) affects the social network by dynamically altering its structure (active edges), a change in intervention strategy would affect the number of contagions in a simulation intervals and hence logically alter the graph structure. This would affect performance depending on how effective was the intervention.

- `Algorithm` : A given algorithm can be implemented in more than one way and can thus affect the overall simulation time. The total number of interactions are not affected by 2 different implementations of the same algorithm.
- `GraphT` : The generation of graph depends upon the degree distribution parameters in kernel 0. If locations have large degree distribution (large number of out-edges) then number of interactions would increase considerably. Also, the graph data structures play very important role in accessing and storing data. Poor choice of data structure would yield disastrous results.

Figure 7.1 provides the list of input parameters used in our experiments. It also provides information on HPC systems and runtimes used in our benchmark. We measure the performance of our benchmark on three different high performance systems in terms of parameters given in Figure 7.2.

7.3 Preliminary Results

In this section, we present some preliminary results of our benchmark on three high-performance compute systems: Shadowfax (SFX), BlueRidge, and HokieOne (see Figure 7.1 for reference). Specifically, we present scalability results for kernels 4 and 5 that simulate the contagion-diffusion. For kernels 0-3, we do not perform any analysis as the time required for generation of the graphs and assigning locations to location groups is quite insignificant to that compared with the simulation kernels (4 and 5).

7.3.1 Kernel 4 implementations

We evaluate 3 different implementations of kernel 4 and name them as Simulation 1 (or Sim 1), Simulation 2 (or Sim 2), and Simulation 3 (or Sim 3). Each implementation is different from the other in terms of vertex traversal or edge traversal methods. The Simulators are briefly explained below. Some statistics related to every simulator is also provided in section 7.3.2.

1. **Simulator 1 - Edge Based** : For each location, we find the set E of all the edges (i.e., activities) that are incident on the location vertex. For any two edges e_1, e_2 (two persons interacting at the location) at the common location, we assert that if the durations overlap and either of the person vertex is infected while the other is in susceptible state, a potential

1. **Simulation kernels:** Contact-based kernel 5 (models EpiFast) and activity-based kernel 4 (models EpiSimdemics).

2. **Compute Platform:** Single node of two high-performance, shared-nothing linux clusters and one shared memory linux cluster.

Platform	**Shadowfax (SFX)**	**BlueRidge**	**Hokieone**
Processor	Xeon E7-4860	Xeon E5-2670	Xeon X7542
L3 Cache	24 MB	20 MB	18MB
# Cores	10	8	6
# Threads	20	16	6
Clock	2.26 GHz	2.6 GHz	2.67 GZ
QPI Speed	6.4 GT/s (1 link)	8 GT/s (2 links)	5.86 GT/s
# Sockets per node	4	2	82
Total Cores	40 (4x10)	16 (2x8)	492 (6x82)

3. **Compiler:** GCC 4.8.3

4. **Intel Threading Building Blocks:** 4.2

5. **Intel Cilk Plus - GCC 4.8 Binaries**

6. **Boost Library:** 1.55

7. **Disease Parameters:**

   - Model: SEIR Model [9]
   - Transmissiblity: 0.00003
   - Incubation period: 2 days
   - Infectious period: 4 days
   - Interventions: NI, EI
   - Location Groups: 100
   - Infectivity: 1
   - Susceptibility: 1

Figure 7.1: Input Parameters in the Experiments

1. **TIPS** - Total interactions per second.
2. **Scalability** - both weak and strong scaling. In weak scaling, we increase the number of threads as we increase the graph size. In strong scaling, while we go on increasing the number of threads, the graph size remains constant.

Figure 7.2: Output Parameters in the Experiments

contagion event occurs based upon disease model parameters and duration of overlap (interaction period).

2. **Simulator 2 - Edge Based** : This implementation is a refinement over Simulator 1 in the sense that, instead of computing a single set of all the edges at a given location, we compute 2 different edges sets - one for those edges which contain infected vertices E_i, and other for those which contain susceptible vertices E_s. The for any infected edge from infected set ($e_i \in E_i$) and any susceptible edge from susceptible set ($e_s \in E_s$), we compute probability of contagion based upon the disease model parameters and interaction period.
3. **Simulator 3 - Node Based** : This method is altogether a different approach for kernel 4 simulation. At each location we compute the infected vertex set (V_i) and susceptible vertex set (V_s) first. Then for any infected vertex ($v_i \in V_i$) and an susceptible vertex ($v_s \in V_s$), we find all the edges that have common location amongst them and then proceed with contagion diffusion.

7.3.2 Run Statistics

The run statistics for our kernels are provided below in Table 7.1 and 7.2 below:

Algorithm	**Graph 1** (In billions)	**Graph 2** (In billions)	**Graph 3** (In billions)	**Graph 4** (In billions)
Sim 1	671	1364	2734	5479
Sim 2	9.05	18.21	36.51	73.09
Sim 3	9.24	18.74	37.56	75.2

Table 7.1: Total Interactions for each Implementation

Algorithm	Graph 1	Graph 2	Graph 3	Graph 4
Sim 1	23228	46542	93334	186950
Sim 2	23750	47967	95883	191324
Sim 3	23894	48303	96541	193577

Table 7.2: Total Contagions during 100 day simulation period (70% infection rate)

7.3.3 Epi-curves for Kernel 4 and Kernel 5

Figure 7.3 shows the epicurves for kernels 4 and 5. Epicurves provide information about how the contagion process takes place in the social network by plotting the number of contagions (or successful disease transmissions) against simulation interval (days). The graph resembles similarity to those generated by EpiSimdemics and EpiFast algorithms with similar disease parameters but with large population graphs. Depending upon the disease parameters, the simulation can shift towards the right or left and eventually die out. Thus, our benchmark is able to capture the computational and communication patterns of EpiSimdemics and EpiFast.

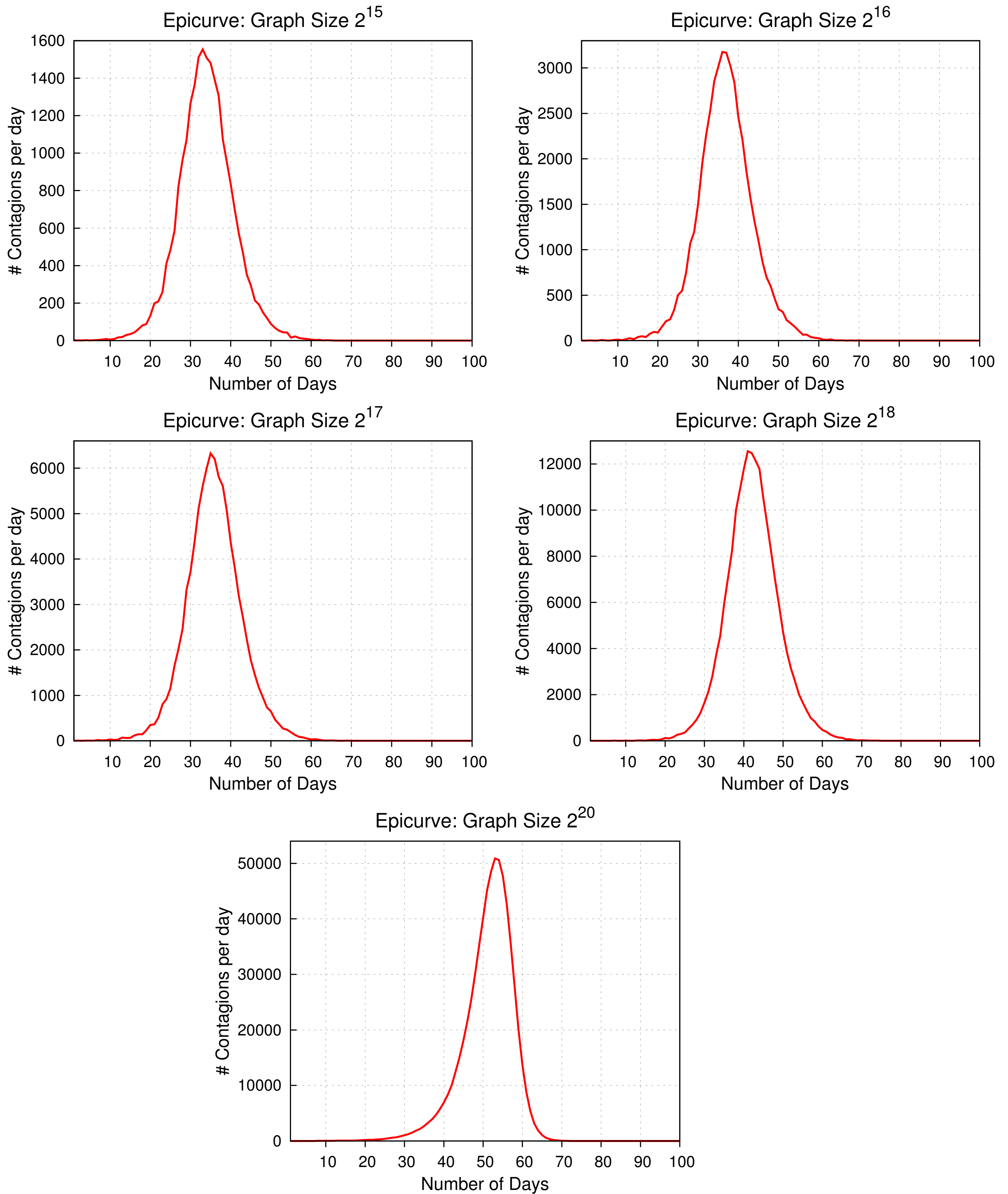


Figure 7.3: Epi-curves for Kernels 4 and 5 for varying graph sizes

7.3.4 Cilk vs. TBB for Kernels 4 and 5

We evaluate our benchmark for two runtimes CILK and TBB on two different machines for kernels 4 and 5. Kernel 4 - Simulator 1 and 2 is run on Shadowfax, while Kernel 5 is run on BlueRidge. Figure 7.4 shows the relative performance of the two runtimes for kernel 4 and 5.

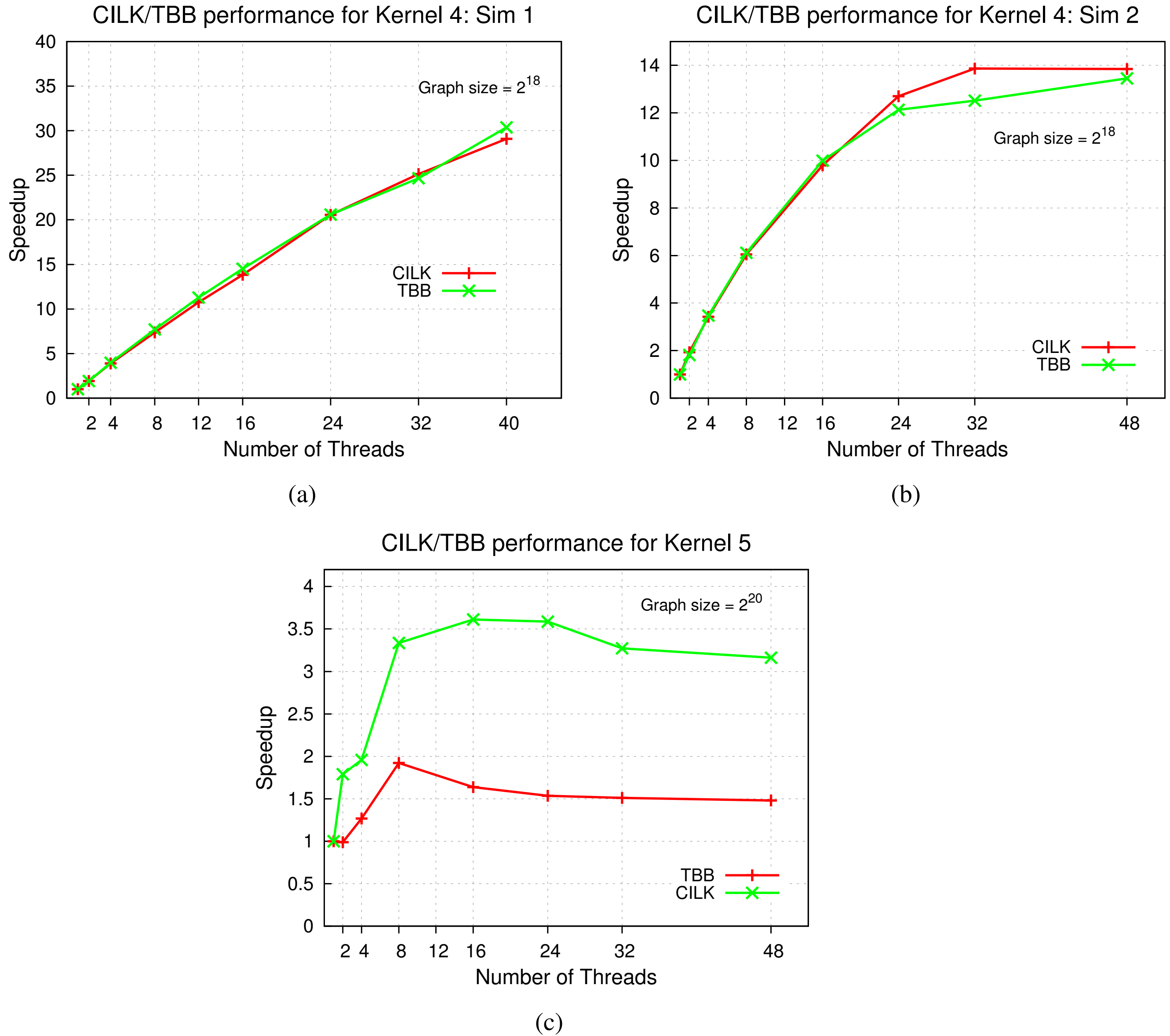


Figure 7.4: Cilk vs. TBB for Kernels 4 and 5. Plot (b) shows the results for Kernel 4 running Sim 2 algorithm on the graph size 2^{18} (on SFX). (c) shows the results for Kernel 5 the graph size 2^{20} (on BlueRidge).

Observations:

1. For kernel 4, on Shadowfax, Cilk and TBB both perform almost equally. This can be seen from Figure 7.4 above for Sim 1 and Sim 2.

2. For kernel 5, on BlueRidge, TBB performance degrades due to large number of tasks created that change dynamically. In kernel 5, tasks are directly mapped onto the number of infected persons in a given simulation phase. As this varies considerably (see epicurves) as the simulation progresses, number of tasks created dynamically by the runtime also increase and decrease accordingly. As a result TBB does not perform well. However, Cilk performs better that TBB because work stealing mechanism of its runtime is based on per worker queue. In case of TBB, the task pool is common and shared amongst all threads.

Conclusion:

1. For Kernel 4, we are able to achieve decent speedup on SFX machine. This is mainly because the task size in case of both the runtimes is fixed - i.e., number of location groups is fixed in kernel 4. This gives both runtimes the opportunity for task/work stealing in case of idle threads.

2. It can be seen that even though both Cilk and TBB do not perform well for Kernel 5 since (it is not scalable), the performance of TBB degrades due to large number of tasks that are spawned by the runtime. The job of creating the tasks for TBB threads becomes the bottleneck. In terms of speedup, Cilk performs almost twice as better than TBB, mainly because every worker (thread) has its own queue from which other workers (threads) can steal the work, rather than a common task pool as in case of TBB. This gives Cilk a minor performance gain.

7.4 Strong Scaling

Strong Scaling is the crude or default speedup obtained by increasing the number of threads (or cores) in a parallel program for a problem of fixed size. In strong scaling, a program is considered to scale linearly if the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used (n). Here, our graph size (number of vertices and edges) stays fixed but the number of processing elements (threads) are increased. Strong scaling justifies the compute bound characteristic of kernel 4 and 5.

Calculating Strong Scaling Efficiency

Let t_1 be the amount of time to complete a work with *one* processing element and, t_n the amount of time to complete the same work with n processing elements. Then the strong scaling efficiency (in %) is given as:

$$\text{SSE} = \frac{t_1}{n \times t_n} \times 100\%$$

7.4.1 Relative Performance of Simulators 1, 2, 3

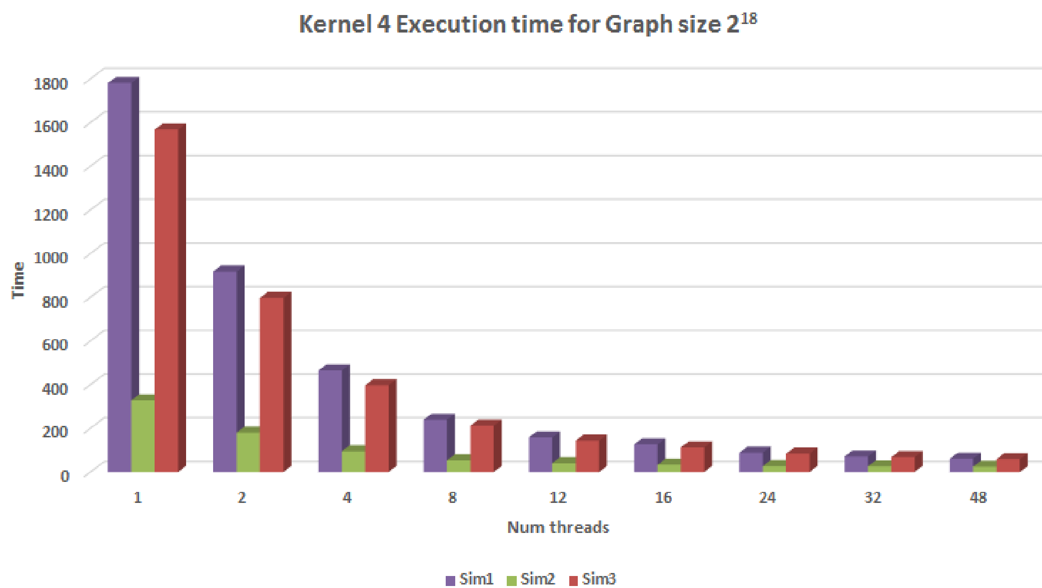


Figure 7.5: Relative performance of Sim 1, Sim 2, Sim 3 implementations for kernel 4

Figure 7.5 shows the relative performance of three implementations for kernel 4 in terms of their running times for serial and parallel execution on Shadowfax for Person-Location graph with 2^{18} person nodes. It can be seen that Sim 2 is the fastest followed by Sim 3 and Sim 1 respectively. In section 7.5.1, we will see that even though Sim 2 is fastest, it is less scalable than Sim 1 and Sim 3. The reason for this behavior is justified in the same section.

7.4.2 Scalability Results for Kernel 4: Strong Scalability

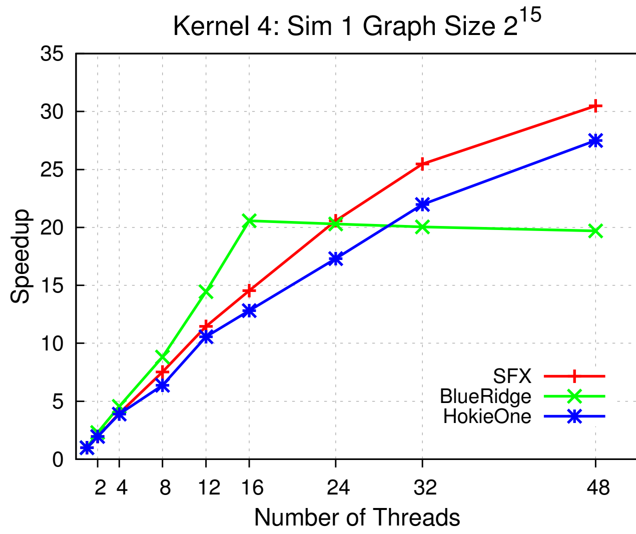
Figures 7.6, 7.7, and 7.8 show the strong scalability for kernel 4 implementations.

Observations:

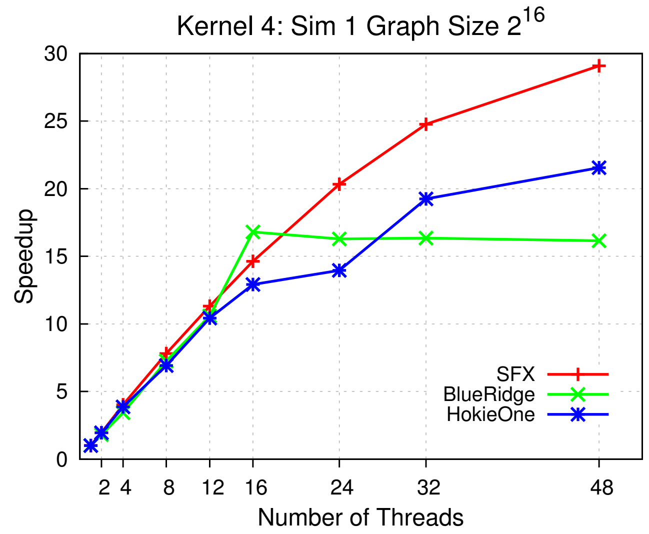
1. **Sim 1:** For BlueRidge (with 16 cores) it can be seen that speedup does not seem to increase after 16 threads. However, HokieOne and SFX yield a decent speed up for Sim 1 implementation.
2. **Sim 2:** Since Sim 2 is based upon refinement of Sim 1 algorithm, its simulation time is considerably less than that of Sim 1. Since this implementation takes very less time as compared with Sim 1, we do not get good speedup for Sim 2 on all the machines.
3. **Sim 3:** For Sim 3 implementation, we observe similar characteristics as that for Sim 2.

Conclusions:

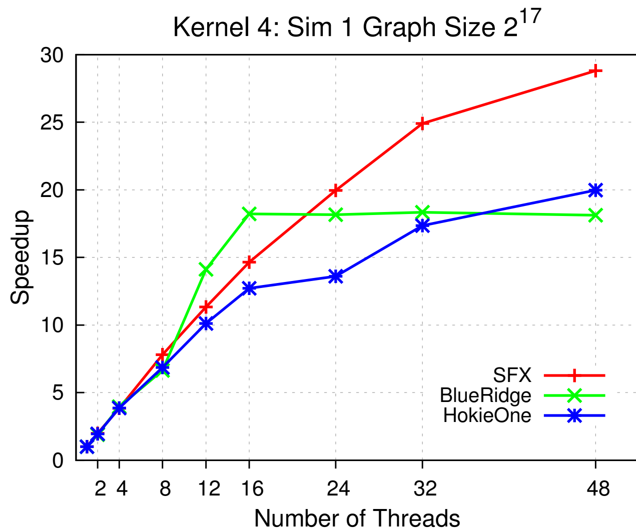
1. Increasing number of threads with limited number of processors at hand will not achieve a good speedup.
2. Sim 2 would achieve good speed up for larger graph sizes because execution time for small graph size (2^{19} or less) is very less as that compared with Sim 1. Similar argument for Sim 3.
3. Since the number of location groups are constant for a given simulation, we expect that with all these kernels would be fairly scalable as problem size increase (weak scalability)



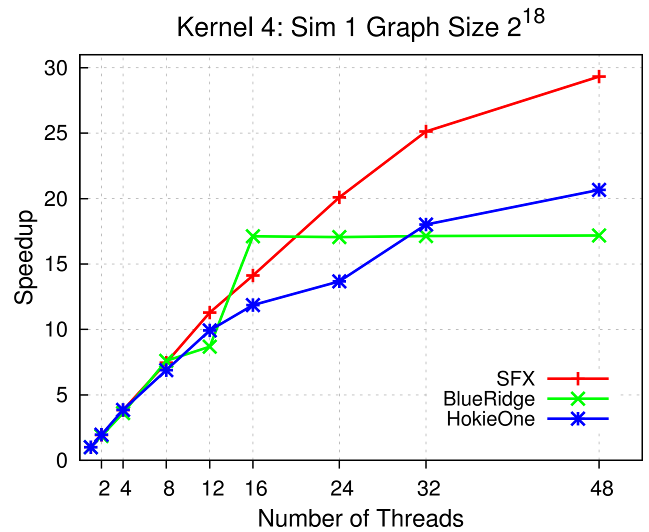
(a)



(b)

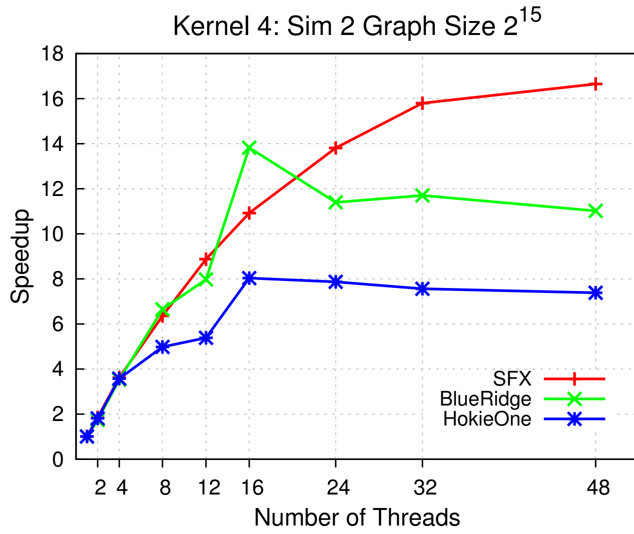


(c)

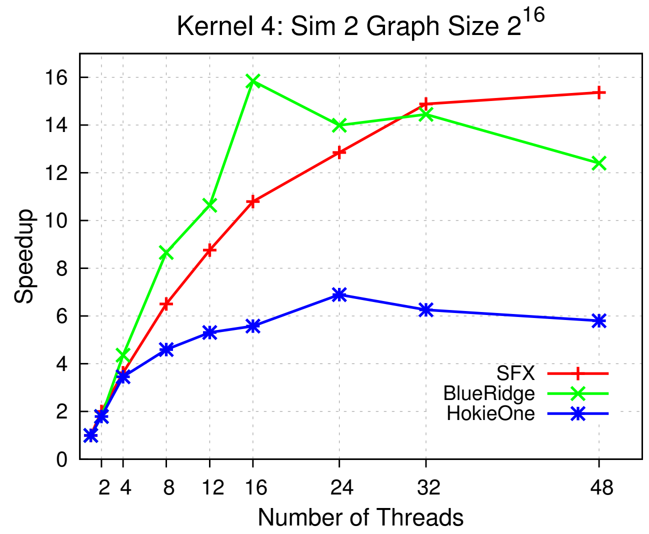


(d)

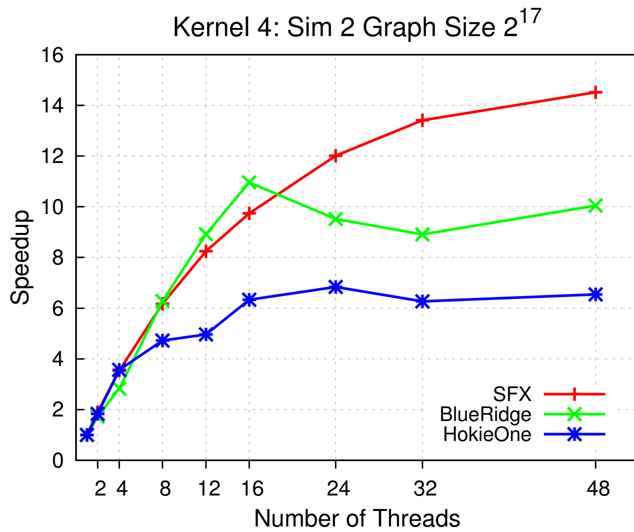
Figure 7.6: Strong scalability of Kernel 4 for implementation Sim 1, with varying graph sizes and on three different architectures - SFX, BlueRudge and Hokieone. Plot (a) shows the strong scalability results for graph size 2^{15}. Plot (b) shows the strong scalability results for graph size 2^{16}. Plot (c) shows the strong scalability results for graph size 2^{17}. Plot (d) shows the strong scalability results for graph size 2^{18}.



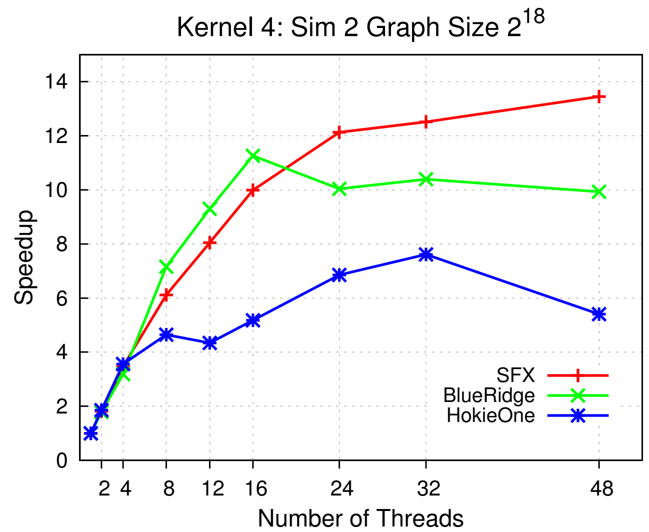
(a)



(b)

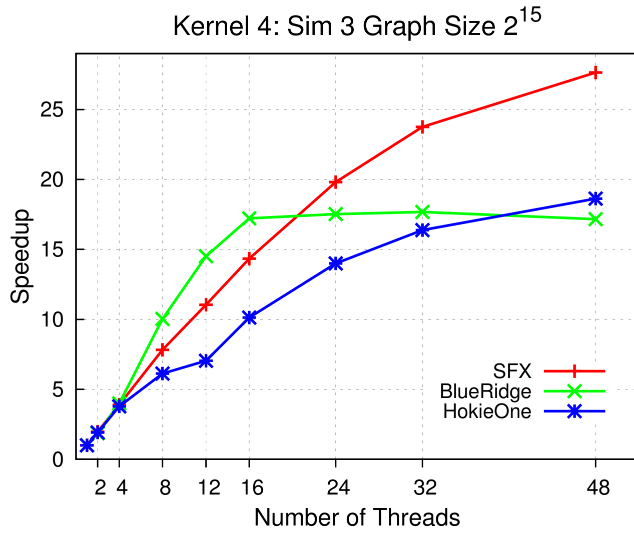


(c)

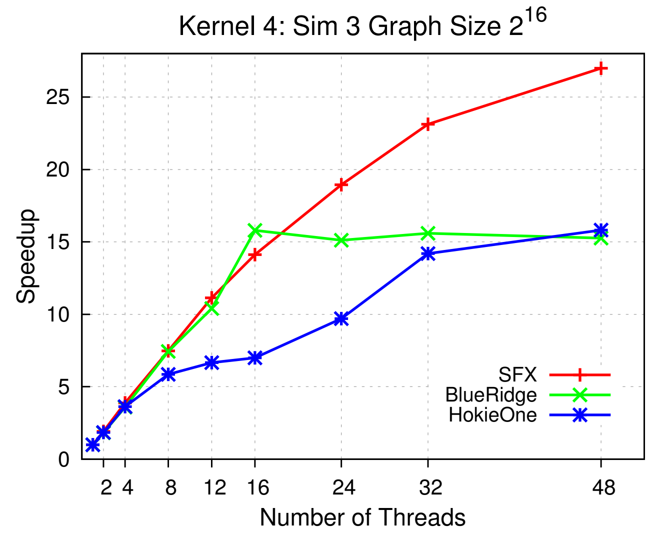


(d)

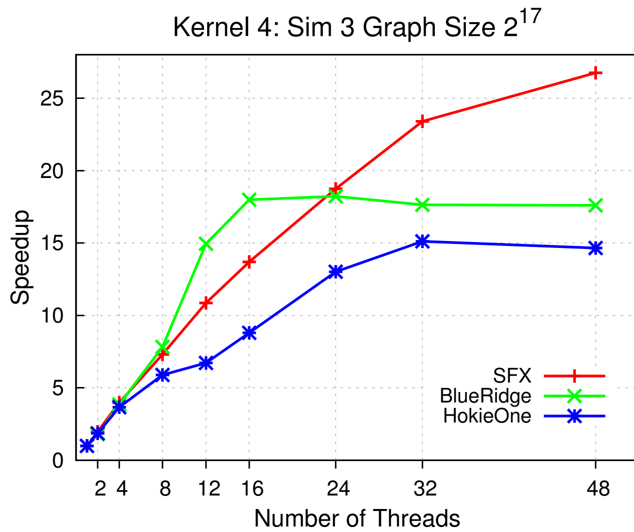
Figure 7.7: Strong scalability of Kernel 4 for implementation Sim 2, with varying graph sizes and on three different architectures - SFX, BlueRudge and Hokieone. Plot (a) shows the strong scalability results for graph size 2^{15}. Plot (b) shows the strong scalability results for graph size 2^{16}. Plot (c) shows the strong scalability results for graph size 2^{17}. Plot (d) shows the strong scalability results for graph size 2^{18}.



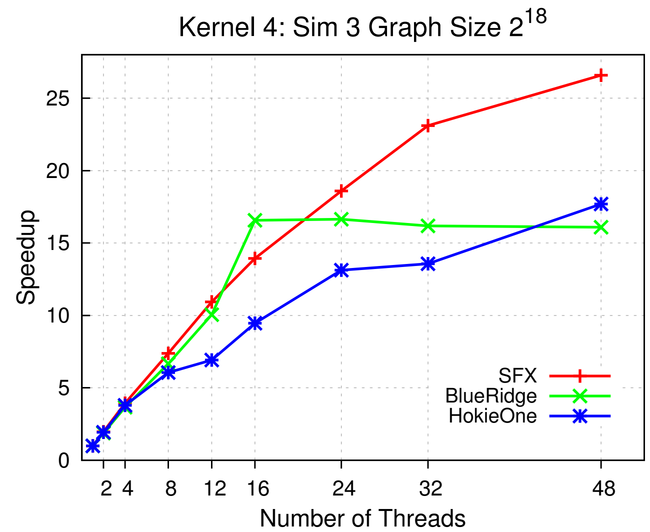
(a)



(b)



(c)



(d)

Figure 7.8: Strong scalability of Kernel 4 for implementation Sim 3, with varying graph sizes and on three different architectures - SFX, BlueRudge and Hokieone. Plot (a) shows the strong scalability results for graph size 2^{15}. Plot (b) shows the strong scalability results for graph size 2^{16}. Plot (c) shows the strong scalability results for graph size 2^{17}. Plot (d) shows the strong scalability results for graph size 2^{18}.

7.5 Weak Scaling

In weak scaling, the work (or problem size) that is assigned to every processing element (a thread) is constant, but more processing elements are used to solve a problem of larger size. Weak Scaling justifies the memory-bound characteristic of the programs (in our case, kernel 4 and 5). Linear speed up is achieved if total running time of the program remains constant when the work is increased in direct proportion to the number of processing elements. Weak scaling signifies how the solution size varies with fixed problem size per processing element (core or thread).

Calculating Weak Scaling Efficiency

If t_1 is the amount of time required to complete a work with *one* processing element (thread, core, etc.) and t_n is the time required to complete n units of the same work with n processing elements, then the weak scaling efficiency is given by:

$$\text{WSE} = \frac{t_1}{t_n} \times 100\%$$

7.5.1 Scalability Results for Kernels 4 and 5: Weak Scalability

Figures 7.9 and 7.10 show the weak scalability graphs for kernel 4 and 5 respectively.

Observations:

1. **Sim 1** : Scalable as graph size and number of processors increase by a constant factor.
2. **Sim 2** : Not scalable for HokieOne. This is because as the number of threads increase, the number of tasks per threads becomes less. Also, this implementation is based upon finding the infected set of edges instead of finding entire edge list at every location. Hence not scalable because number of edges in infected set change every simulation phase.
3. **Sim 3** : Not scalable for HokieOne. To achieve scalability for Hokieone, we need to increase the number of tasks per thread - this can be achieved by increasing the number of location groups as the graph size increases. Also, Sim 3 is based upon finding infected vertices instead of edges and hence similar argument applies as that justified for Sim 2 above.

4. **Kernel 5** : Not Scalable for any of the systems. This is because dynamic number of tasks created during each simulation phase. When the number of tasks are very large, creating them (being serial) inherently becomes the bottleneck. To solve this, we can dynamically increase/decrease worker threads depending upon the number of infected population in a simulation phase.

Conclusions:

1. For Sim 2, we can achieve scalability by increasing the graph size. Sim 2 being an optimization over Sim 1 implementation, can be used for larger graph sizes with large number of location group size.
2. For kernel 5, we must provide a runtime mechanism to dynamically increase or decrease the number of worker threads spawn in order to keep the per-thread-work constant depending upon the number of infected persons in the group.

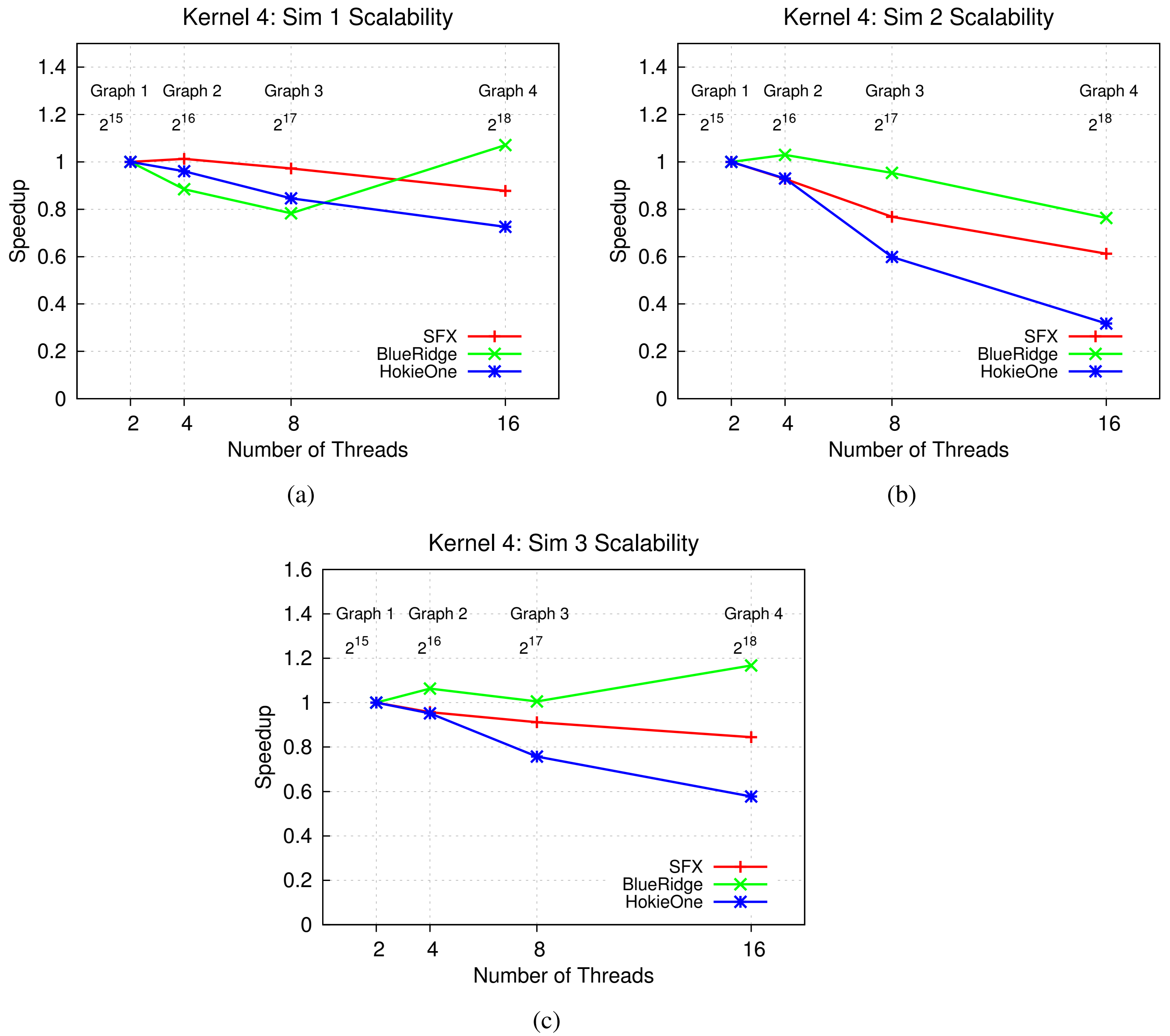


Figure 7.9: Weak scalability of Kernel 4 for the three implementations, each with varying graph sizes and on three different architectures - SFX, BlueRidge and Hokieone. Plot (a) shows the weak scalability results for implementation 1. Plot (b) shows the weak scalability results for implementation 2. Plot (c) shows the weak scalability results for implementation 3.

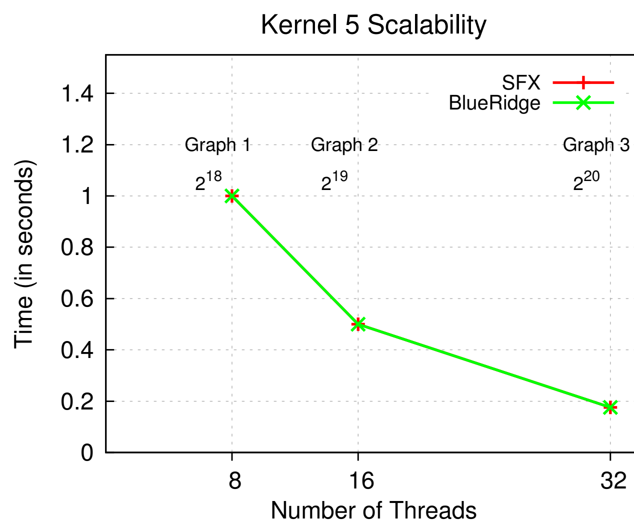


Figure 7.10: Weak scalability of Kernel 5 with varying graph sizes and on two different architectures - SFX and BlueRidge.

7.6 Intel MIC Performance for Kernel 4

We use Simulator 1 for kernel 4 and run two implementations for Cilk and one implementation of TBB for Intel MIC cards. The BlueRidge cluster has 120 nodes each with 2 Intel Xeon Phi 5110P (MIC) cards, where every coprocessor has 60 cores with 1.05 GHz clock speed. The cards do not shared address space in native modes. We evaluate the performance of kernel 4 for Cilk and TBB and summarize the strong scaling results in Figure 7.11.

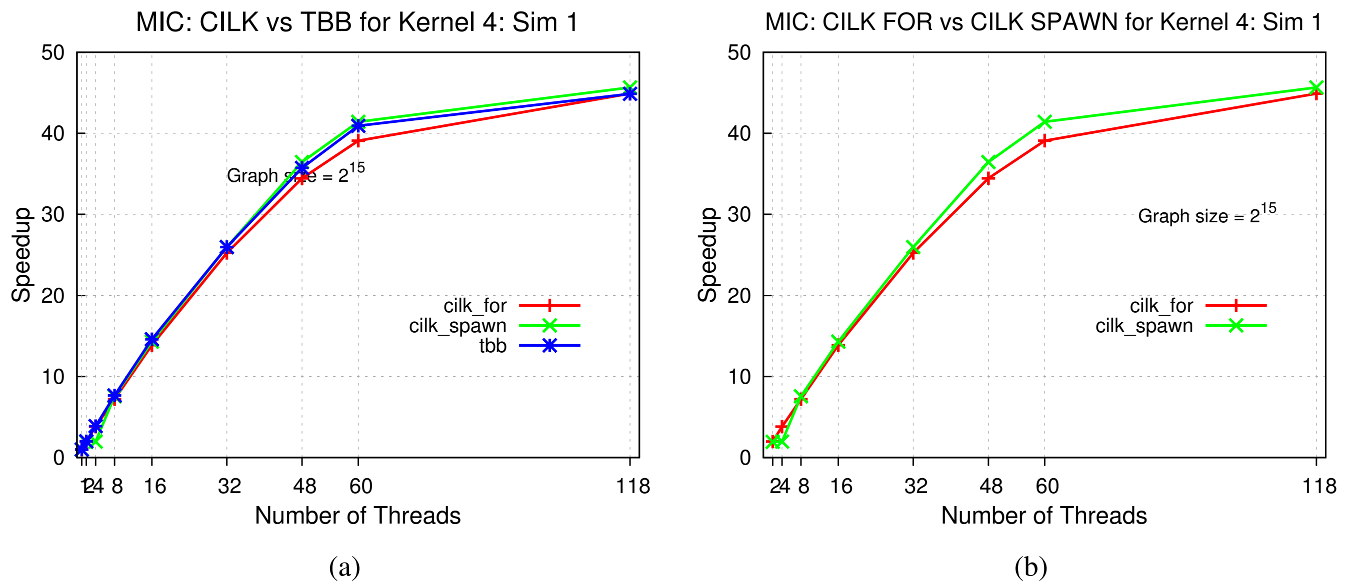


Figure 7.11: Strong scaling of Kernel 4 for the three implementations on Intel MIC card in BlueRidge System. Plot (a) shows the relative performance of TBB and Cilk implementations. Plot (b) shows comparison for `cilk_for` and `cilk_sync` implementations.

Observations:

1. **Cilk:** In Figure 7.11 (b), it can be seen that both `cilk_for` and `cilk_spawn` perform almost the equal and also give decent strong scaling speedup.

2. **TBB:** Since the number of location groups is fixed for each run, TBB also performs at par with the Cilk runtime as it does not suffer from the problem of dynamica task creation explained earlier.

3. **Cilk and TBB:** After we increase the number of threads to 118 (max supported), both Cilk and TBB do not scale well as the processor to thread ratio becomes greater than one. In

most of the cases, for parallel processing applications, we keep this ratio to 1 as increasing it would affect strong speedup due additional thread scheduling.

Conclusions:

1. For Runtimes perform well upto the max cores supported on the card.
2. We expect that kernel 4 will also be scalable (weak scaling) as graph size increases.
3. However we must take note that, the co-processor (MIC card) and the host machine (compute node that hold this card) have different memory addressing. While host machines can have large memory address space ($>$ 64 GB), for native mode execution of our kernels, we are restricted to only 8 GB memory space. Thus we may not be able to run a standalone native benchmark kernel for very large graph sizes.
4. Each core in the MIC card is 1.05 GHz and is quite slow if run for serial code. Thus our sequential kernels would considerably take more time for initializing data (graphs, etc.). But by hiding this latency (computational power) by adding more number of cores, we can achieve a GPU style acceleration for highly parallel tasks.

Chapter 8

Conclusions

In this work, we present a suite of kernels that together form benchmark for contagion diffusion simulation. We provide an encoding of the benchmark specifications using C++11 templates and iterators that is generic (as in implementation agnostic) and composable, i.e., different implementations of kernels can be composed together to arrive at alternative implementation of the benchmark. We have used generic programming primitives such as Concepts, Traits, Tag Dispatch, Arbitrary Overloading, Policy Based Class Designs for our kernels that provide a generic and simple interface to make it compatible with any other graph library apart from BGL.

The benchmark is used to evaluate performance of three class of machines based upon the TIPS metric. Preliminary results indicate that kernel 4 is more scalable than kernel 5 and Shadowfax cluster is more scalable for kernel 5 than BlueRidge and Hokieone. Kernel 5 being non-scalable yield similar results for all machines. Further low level analysis of kernels is under progress to extend our current work with the help of profiling tools.

Ongoing work:
Our current work is focused on two major aspects:
(1) Standardization – developing codes for our benchmark to make it compatible to any standard graph library which implements its basic specifications.
(2) Distributed/Shared memory implementation – Using the concepts and methodology we described in chapter 5 we aim to lift the sequential implementation for large scale distributed memory implementations using MPI [36] and Charm++ [50] without affecting the genericness and simplicity of the benchmark.

Bibliography

[1] Barrett, C. L., Bisset, K. R., Eubank, S. G., Feng, X., & Marathe, M. V. (2008, November). EpiSimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (p. 37). IEEE Press.

[2] Bisset, K. R., Feng, X., Marathe, M., & Yardi, S. (2009, December). Modeling interaction between individuals, social networks and public policy to support public health epidemiology. In Simulation Conference (WSC), Proceedings of the 2009 Winter (pp. 2020-2031). IEEE.

[3] Bisset, K. R., Aji, A. M., Bohm, E., Kale, L. V., Kamal, T., Marathe, M. V., & Yeom, J. S. (2012, May). Simulating the spread of infectious disease over large realistic social networks using Charm++. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International (pp. 507-518). IEEE.

[4] Kamal, T., Bisset, K. R., Butt, A. R., Chungbaek, Y., & Marathe, M. (2013, June). Load balancing in large-scale epidemiological simulations. In Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (pp. 123-124). ACM.

[5] Aji, A. M., Dinan, J., Buntinas, D., Balaji, P., Feng, W. C., Bisset, K. R., & Thakur, R. (2012, June). MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems. In High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on (pp. 647-654). IEEE.

[6] Bisset, K. R., Aji, A. M., Marathe, M. V., & Feng, W. C. (2012). High-performance biocomputing for simulating the spread of contagion over large contact networks. BMC genomics, 13(Suppl 2), S3.

[7] Bisset, K. R., Chen, J., Feng, X., Kumar, V. S., & Marathe, M. V. (2009, June). EpiFast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems. In Proceedings of the 23rd international conference on Supercomputing (pp. 430-439). ACM.

[8] Siek, J. G., Lee, L. Q., & Lumsdaine, A. (2001). Boost Graph Library: User Guide and Reference Manual, The. Pearson Education.

[9] Bailey, N. T. (1975). The mathematical theory of infectious diseases and its applications. Charles Griffin & Company Ltd, 5a Crendon Street, High Wycombe, Bucks HP13 6LE..

[10] Gregor, D., & Lumsdaine, A. (2005). The parallel BGL: A generic library for distributed graph computations. Parallel Object-Oriented Scientific Computing (POOSC), 2.

[11] Reinders, J. (2007). Intel threading building blocks: outfitting C++ for multi-core processor parallelism. " O'Reilly Media, Inc.".

[12] Heroux, M. & Dongarra, J. Towards a New Metric for Ranking High Performance Computing Systems, UTK EECS and Sandia National Labs Report SAND2013-4744, June 2013

[13] Gregor, D., & Lumsdaine, A. (2005). Lifting sequential graph algorithms for distributed-memory parallel computation. ACM SIGPLAN Notices, 40(10), 423-437.

[14] Dongarra, J. J., Meuer, H. W., & Strohmaier, E. (1997). TOP500 supercomputer sites. Supercomputer, 13, 89-111.

[15] Luszczek, P. R., Bailey, D. H., Dongarra, J. J., Kepner, J., Lucas, R. F., Rabenseifner, R., & Takahashi, D. (2006, November). The HPC Challenge (HPCC) benchmark suite. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing (p. 213). ACM.

[16] Feng, W. C., & Cameron, K. W. (2007). The green500 list: Encouraging sustainable supercomputing. Computer, 40(12), 50-55.

[17] Murphy, R. C., Wheeler, K. B., Barrett, B. W., & Ang, J. A. (2010). Introducing the graph 500. Cray UserâĂŹs Group (CUG).

[18] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., ... & Weeratunga, S. K. (1991). The NAS parallel benchmarks. International Journal of High Performance Computing Applications, 5(3), 63-73.

[19] Dixit, K. M. (1991). The SPEC benchmarks. Parallel computing, 17(10), 1195-1209.

[20] Van Der Steen, A. J. (1991). The benchmark of the EuroBen Group. Parallel Computing, 17(10), 1211-1221.

[21] Marathe, M., & Vullikanti, A. K. S. (2013). Computational epidemiology. Communications of the ACM, 56(7), 88-96.

[22] Yeom, J. S., Bhatele, A., Bisset, K., Bohm, E., Gupta, A., Kale, L. V., ... & Wesolowski, L. (2014). Overcoming the Scalability Challenges of Epidemic Simulations on Blue Waters. In Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear).

[23] Seltzer, M., Krinsky, D., Smith, K., & Zhang, X. (1999). The case for application-specific benchmarking. In Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on (pp. 102-107). IEEE.

[24] Meyers, L. (2007). Contact network epidemiology: Bond percolation applied to infectious disease prediction and control. Bulletin of the American Mathematical Society, 44(1), 63-86.

[25] Meyers, L. A., Newman, M. E. J., & Pourbohloul, B. (2006). Predicting epidemics on directed contact networks. Journal of theoretical biology, 240(3), 400-418.

[26] Meyers, L. A., Pourbohloul, B., Newman, M. E., Skowronski, D. M., & Brunham, R. C. (2005). Network theory and SARS: predicting outbreak diversity. Journal of theoretical biology, 232(1), 71-81.

[27] Bisset, K. R., Feng, X., Marathe, M., & Yardi, S. (2009, December). Modeling interaction between individuals, social networks and public policy to support public health epidemiology. In Simulation Conference (WSC), Proceedings of the 2009 Winter (pp. 2020-2031). IEEE.

[28] Barrett, C., Hunt III, H. B., Marathe, M. V., Ravi, S. S., Rosenkrantz, D. J., & Stearns, R. E. (2011). Modeling and analyzing social network dynamics using stochastic discrete graphical dynamical systems. Theoretical Computer Science, 412(30), 3932-3946.

[29] Barrett, C. L., Hunt III, H. B., Marathe, M. V., Ravi, S. S., Rosenkrantz, D. J., & Stearns, R. E. (2006). Complexity of reachability problems for finite discrete dynamical systems. Journal of Computer and System Sciences, 72(8), 1317-1345.

[30] Bisset, K. R., Chen, J., Feng, X., Ma, Y., & Marathe, M. V. (2010, June). Indemics: an interactive data intensive framework for high performance epidemic simulation. In Proceedings of the 24th ACM International Conference on Supercomputing (pp. 233-242). ACM.

[31] Mortveit, H., & Reidys, C. (2007). An introduction to sequential dynamical systems. Springer.

[32] Gregor, D., Jarvi, J., Siek, J., Stroustrup, B., Dos Reis, G., & Lumsdaine, A. (2006, October). Concepts: linguistic support for generic programming in C++. In ACM SIGPLAN Notices (Vol. 41, No. 10, pp. 291-310). ACM.

[33] Austern, M. H. (1998). Generic programming and the STL: Using and extending the C++ Standard Template Library. Professional Computing Series.

[34] Sutton, A., & Stroustrup, B. (2012). Design of concept libraries for C++. In Software Language Engineering (pp. 97-118). Springer Berlin Heidelberg.

[35] Gregor, D. Concepts: An Introduction to Generic Programming. http://www.generic-programming.org/about/intro/concepts.php

[36] Gropp, W., Lusk, E., & Skjellum, A. (1999). Using MPI: portable parallel programming with the message-passing interface (Vol. 1). MIT press.

[37] Hazewinkel, M. (2001). Formal language, Encyclopedia of Mathematics. Springer. ISBN 978-1-55608-010-4.

[38] Hopcroft, J. E. (1979). Introduction to automata theory, languages, and computation. Pearson Education India.

[39] Pointers and Memory Introduction to pointers – Stanford Computer Science Education Library

[40] Abrahams, D., Siek, J., & Broadband, A. (2001, October). Policy adaptors and the Boost iterator adaptor library. In Second Workshop on C++ Template Programming.(October 2001).

[41] Weisstein, E. Functor. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Functor.html

[42] Mac Lane, S. (1998). Categories for the working mathematician (Vol. 5). springer.

[43] Carnap, R. (2002). The logical syntax of language. Open Court Publishing.

[44] Hazewinkel, M., Gubareni, N. M., & Kirichenko, V. V. (2010). Algebras, Rings, and Modules: Lie Algebras and Hopf Algebras (Vol. 168). American Mathematical Soc..

[45] Adamek, J., Herrlich, H., & Strecker, G. E. (1990). Abstract and Concrete Categories: The Joy of Cats. A Wiley-Interscience Publication. Pure Appl. Math.(NY), John Wiley & Sons Inc., New York.

[46] Lawvere, F. W., & Schanuel, S. H. (2009). Conceptual mathematics: a first introduction to categories. Cambridge University Press.

[47] Zalta, E. N. (2004). The Stanford Encyclopedia of Philosophy.

[48] Duran, A., & Klemm, M. (2012, July). The Intel many integrated core architecture. In High Performance Computing and Simulation (HPCS), 2012 International Conference on (pp. 365-366). IEEE.

[49] Plauger, P. J., Lee, M., Musser, D., & Stepanov, A. A. (2000). C++ Standard Template Library. Prentice Hall PTR.

[50] Kale, L. V., & Krishnan, S. (1993). CHARM++: a portable concurrent object oriented system based on C++ (Vol. 28, No. 10, pp. 91-108). ACM.

[51] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., & Zhou, Y. (1996). Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing, 37(1), 55-69.

[52] DeMarco, T. (1979). Structured analysis and system specification (pp. 409-424). Yourdon Press.

[53] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., ... & Yelick, K. A. (2006). The landscape of parallel computing research: A view from Berkeley (Vol. 2). Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[54] Feng, W. C., Lin, H., Scogland, T., & Zhang, J. (2012, April). OpenCL and the 13 dwarfs: a work in progress. In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (pp. 291-294). ACM.

[55] Robison, A. D. (2013). Composable Parallel Patterns with Intel Cilk Plus. Computing in Science & Engineering, 15(2), 0066-71.

[56] Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE, 5(1), 46-55.

[57] Alexandrescu, A. (2001). Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley. Reading.

Appendices

Intel Compiler

A.1 Compiling for Intel MIC

We found an issue in the Intel Compose 13 SP1 Update3 (install on BlueRidge HPC system) while compiling the code for Intel MIC Co-processors using -std=c++11 flags. More details can be found at this link *https://software.intel.com/en-us/forums/topic/517612*. (Internal tracking id: DPD200358195)

Summary: The functor takes 2 arguments in its operator()() by using a boost::bind() method. The operator()() method contains a function which uses comparisons ($<$ or $>$) using if-else logic. The error pops up here. Here is the following code.

```
1  #include <iostream>
2  #include <boost/bind.hpp>
3  using namespace std;
4  struct Info_t {
5    Info_t(int s = 0, int e = 0) : start(s), end(e) {   }
6    int start;
7    int end;
8  };
9
10 template <typename T, typename time>
11 void process_info(const T& info1, const T& info2, const time& t) {
12
13     //the below gives compiler error
14     if ((info1.start < info2.end) &&
15         (info2.end > info1.end) &&
16         (info1.end < info2.start))
17        std::cout << "error with mic" << std::endl;
18
```

```
19     //the below if works fine
20     if ((info1.start < info2.end) &&
21         (info2.end > info1.end))
22       std::cout << "no error at all" << std::endl;
23 }
24
25 //Functor
26 template<typename Time>
27 struct my_functor {
28  public:
29   my_functor() {  }
30   my_functor(Time&  time_) : time(time_) {   }
31
32   typedef void result_type;
33
34   template<typename info_t>
35   void operator()(const info_t& info1, const info_t& info2) {
36     const auto& i1 = info1;
37     const auto& i2 = info2;
38     process_info<info_t, Time> (i1,i2, time);
39   }
40  private:
41   Time&  time;
42 };
43
44 int main() {
45 Info_t  i1(10,2);
46 Info_t  i2(2,10);
47 int t = 1;
48 auto func = boost::bind( my_functor<int>(t), _1, _2 );
49 func(i1,i2);
50 return 0;
51 }
```

Reason for Error: The name **"end"** is being brought in via the using declaration from the **std** namespace. Since **std::end** is a template and **info1** is a dependent name, the compiler thinks that **"end < ..."** is the start of a call to **std::end** (note that std::end is a template so having the start of a template argument list be the token < instead of less than makes sense) – hence the error.

Fix: This issue has been fixed in recent compilers (v14+) and has been tested on Intel Composer 2015 Cluster Edition (Trial) and for Intel Xeon Phi 5110P Co-processor on BlueRidge HPC system.

Practical Challenges

B.1 Type compatibility

The `type` compatibility between the benchmark parameters is a very crucial aspect to derive a generic benchmark. Modern C++ is a *strongly typed* and *statically-typed* language. In order to compile any code, every variable, function's arguments, function's return value must have a `type`. For example, the `type` **`int`** stores integral values, while a **`class`** or a **`struct`** is an user defined type. Common types are built-in types (int, float, double, etc.), void type, const type, string type, pointer type and user-defined type. The C++ compiler assigns a type implicitly to any expression before any evaluation. For our kernels, theoretically we can achieve generic-ness by separating the functionalities but while implementing, it is practically very difficult. For example, if we want a state update functor consisting of a `queue` based implementation, it wont simply work with our contagion functors for this would require additional task of inserting elements in a queue and also process the queue based upon the parameters defined for disease model. This would be a specific implementation and hence we may have to provide appropriate matching contagion functors. We have addressed this issue partially by selecting the right implementation based on an integer constant or a boolean and determine if one type supports other type or not.

B.2 Type Deduction

Deducing an object's or a variable's `type` is trivial for simple `types` such as integers, structures, objects. But when we declare a variable with complicated type such as a C++11 Lambda, function pointers, pointers to members, templates, etc. deducing such `type` is extremely difficult in such cases. For example, we can use `auto` to declare and initialize a variable to a lambda expression. We simply cannot declare the type of the variable ourselves because the `type` of a lambda expression is known only to the compiler! In our case, the `auto` keyword has been extensively

used to bind functors with arguments, in trailing return types, automatic type deduction using `decltype`. This enables programmers to focus more on high level programming without going into the intricate complexity of `types` and leave this task to compiler.

The C++11/14 standard emphasizes on this aspect of programming and therefore feels like a new language as quoted by its create - Bjarne Stroustrup. The compilers perform all the `type`-checking magic in the background to free programmers from different domains from this added burden.

Without C++11/14 support, it would be extremely difficult and we may have to depend upon external libraries such as `Boost` and `Loki` [57] for generic programming.