2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

# Simulating the Spread of Infectious Disease over Large Realistic Social Networks using Charm++

Keith R. Bisset*, Ashwin M. Aji*†, Eric Bohm‡, Laxmikant V. Kale‡
Tariq Kamal*†, Madhav V. Marathe*†, and Jae-Seung Yeom*†
**Virginia Bioinformatics Institute*
†*Department of Computer Science*
*Virginia Tech, Blacksburg VA 24061, USA*
*Email: {kbisset,aaji,tkamal,mmarathe,jyeom}@vbi.vt.edu*
‡*Department of Computer Science*
*University of Illinois at Urbana-Champaign Urbana, IL 61801, USA*
*{ebohm,kale}@illinois.edu*

***Abstract*—Preventing and controlling outbreaks of infectious diseases such as pandemic influenza is a top public health priority. EpiSimdemics is an implementation of a scalable parallel algorithm to simulate the spread of contagion, including disease, fear and information, in large ($10^8$ individuals), realistic social contact networks using individual-based models. It also has a rich language for describing public policy and agent behavior. We describe CharmSimdemics and evaluate its performance on national scale populations. Charm++ is a machine independent parallel programming system, providing high-level mechanisms and strategies to facilitate the task of developing highly complex parallel applications. Our design includes mapping of application entities to tasks, leveraging the efficient and scalable communication, synchronization and load balancing strategies of Charm++. Our experimental results on a 768 core system show that the Charm++ version achieves up to a 4-fold increase in performance when compared to the MPI version.**

***Keywords*-Computational Epidemiology; Parallel Efficiency and Scalability; Agent Based Simulation; Programming Models; Charm++; MPI**

## I. Introduction

Contagion (or diffusion) models are pervasive in social and physical sciences. Three recent global scale contagions that have received attention in the media as well as academic circles are: current and past financial contagions, failure of the coupled infrastructure system caused by the Northeast blackout of 2003 and, potential pandemics caused by avian influenza. Developing computational models to reason about these systems is complicated and scientifically challenging for at least three reasons. First, often the size and scale of these systems is extremely large (e.g., pandemic planning at a global scale requires models with 6 Billion agents). Second, the contagion, the underlying interaction network (consisting of both human and technical elements), the public policies and the individual agent behaviors co-evolve making it nearly impossible to apply standard model reduction techniques that are successfully used to study physical systems. Finally, in practical situations, multiple contagion processes simultaneously co-evolve. Going back to our example of epidemiology, we are interested in at least two separate contagion processes: spread of disease through the population and spread of fear, influence and information in response to the epidemic. Developing scientific foundations for practical global-scale problems requires one to model systems comprising of multiple interacting behaviors, networks, and contagions.

MPI (message passing interface) [1] is the de-facto standard for writing efficient parallel programs for distributed memory environments such as high performance clusters. As our previous work [2] has shown, EpiSimdemics, our epidemiological simulation application implemented using MPI, achieves acceptable scalability for up to 376 processing elements when simulating a population of 100 million. However, this requires significant programmer effort to perform efficient inter-process communication to effectively hide their latencies with computation. Moreover, good load balancing and data locality remain hard programming challenges. On the other hand, Charm++ [3] is a message-driven parallel programming framework, where high level object-oriented principles can be used to write efficient high performance software. A typical Charm++ program consists of many parallel objects that communicate among each other via asynchronous messages. The programmer specifies the decomposition of the problem only in terms of interacting objects, and the run-time system handles the mapping of these objects to the processors. Charm++ also has several other advantages, such as inbuilt load balancing and communication optimization frameworks, which significantly improves programmability along with performance. We explain some of the main features of Charm++ in Section III.

In this paper, we describe and evaluate *CharmSimdemics*, a novel design of the epidemiology simulation algorithm using Charm++ in section IV. Our design includes mapping of application entities to tasks, leveraging the efficient and scalable communication, synchronization and load balancing strategies of Charm++. We show that our algorithm maps well to the message-driven parallel object programming model of Charm++. Our experimental results on a 768 core system show that the Charm++ version achieves more than a four-fold increase in performance when compared to the MPI version.

## II. The EpiSimdemics Algorithm

A brief description of the EpiSimdemcis algorithm is given below, and summarized in Figure 1. More details can be found in [2], [4]. The EpiSimdemics algorithm is based on information diffusion across a social network. The network is represented as a bi-partite graph, with people and locations as the nodes, and edges between them representing a visit of a location by a person. The processing is separated into iterations that represent simulated days. Each iteration has the basic following steps:

1) Person entities determine the locations that they are going to visit, based on a normative schedule, public policy, and individual behavior and health state. The person sends a message to each visited location with the details of the visit (time, duration and health state).
2) Location entities compute the interactions that occur between occupants. Each interaction may result in an infection, depending on a stochastic model. For the Epidemiological model, disease propagation is modeled by

$$p_{i\to j} = 1 - (1 - r_i s_j \rho)^{\tau} \qquad (1)$$

where $p_{i\to j}$ is the probability infectious individual $i$ infecting susceptible individual $j$, $\tau$ is the duration of exposure, $r_i$ is the infectivity of $i$, $s_j$ is the susceptibility of $j$, and $\rho$ is the transmissibility, a disease specific property defined as the probability of a single completely susceptible person being infected by a single completely infectious person during one minute of exposure. A message is then sent to each infected person with the details of the infection (time of infection, infector and location).
3) Each person who was infected uses all of the infection messages to determine a new health state.

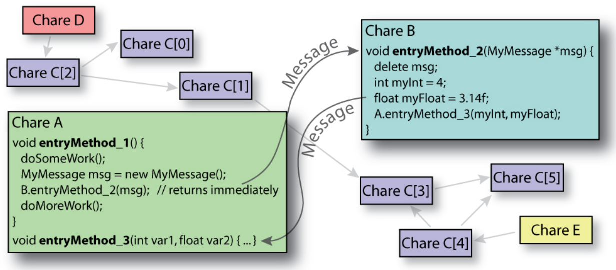
*Object-to-process mapping:* Before the simulation begins, all the person and location objects are first read and processed from the input files. Then, they are randomly distributed among the available MPI processes, i.e. each MPI process will contain many person objects and many location objects. All the messages between any person and location objects are buffered by the corresponding source MPI process, and then explicitly communicated with the target process periodically when the buffer threshold is reached.

```
1:  initialize();                                   ▷ partition data across PEs
2:  partition();
3:  for $t = 0$ to $T$ increasing by $\delta t$ do
4:      foreach individual $p_j \in P_i$ do         ▷ send visits to location PEs
5:          computeVisits($j, t$ to $t + \Delta t$);
6:          sendVisits($\mathrm{MB}_i$);
7:      end for
8:      Visits ← $\mathrm{MB}_i$.retrieveMessages();
9:      synchronize();
10:     foreach location $l_k \in L_i$ do           ▷ compose a serial DES
11:         makeEvents(k, Visits);                  ▷ turn visit data into events
12:         computeInteraction($k$);                ▷ Process Events
13:         sendOutcomes($\mathrm{MB}_i$);
14:     end for
15:     $\mathrm{MB}_i$.retrieveMessages();
16:     synchronize();
17:     foreach $j \in P_i$ do                      ▷ combine outcomes of multiple interactions
18:         updateState(j);
19:     end for
20: end for
```

Figure 1. A pseudocode version of the EpiSimdemics algorithm.

## III. The Charm++ Language and Framework

Charm++ [3] is a distributed object oriented programming environment, where the programmer writes parallel applications in terms of *chare* objects. More importantly, Charm++ is a message driven execution model, where the chare objects interact with each other by message passing via remote method invocation. The programmer's view of the program will be a collection of interacting chare objects, as shown in Figure 2. However, the physical location of the chares, the number of running processes and the details of the physical resources that are being used are all abstracted away by the Charm++ runtime system, as shown in the Figure 3. The programmer may also create regular sequential C++ objects in the same program, which are instantiated and executed locally and not known to the Charm++ runtime.



```
void entryMethod_1() {
  doSomeWork();
  MyMessage msg = new MyMessage();
  B.entryMethod_2(msg);  // returns immediately
  doMoreWork();
}
void entryMethod_3(int var1, float var2) { ... }
```

```
void entryMethod_2(MyMessage *msg) {
  delete msg;
  int myInt = 4;
  float myFloat = 3.14f;
  A.entryMethod_3(myInt, myFloat);
}
```

Figure 2. Charm++: User View.

A message in Charm++ triggers the associated computations in the corresponding remote chare object. These computations in turn can fire off more messages to other (possibly remote) processors that trigger more computations on those processors. The Charm++ runtime system

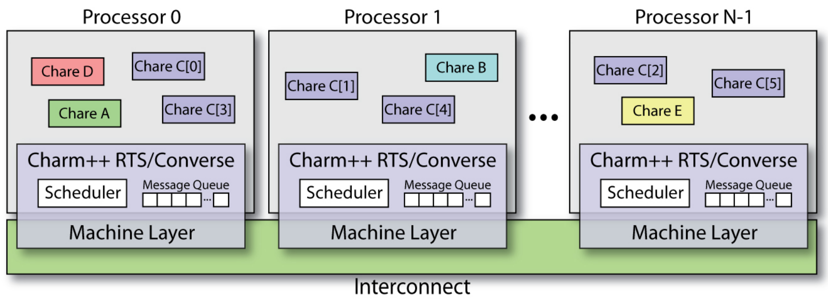


Figure 3. Charm++: System View.

comes with a scheduler, which continuously chooses the next message to be processed from the available pool of messages, and executes the computations associated with that message. Message passing is asynchronous by default, which provides a lot of scope for hiding communication with computation, which is required for any efficient parallel program execution. Messages can be invoked on remote chares by calling their corresponding *entry methods*, which are explicitly registered with the Charm++ runtime system via special interface files.

The resource management in the runtime system allows the programmer to focus on decomposing the problem into chares (or concurrent objects), their associated messages (or communication objects), and the readonly global data of the program. Any Charm++ program begins execution by creating a single instance of a *main chare* on processor 0, and calls its constructor. The main chare is a unique chare object that is reserved by the Charm++ runtime to start the program execution. The main chare can then create a number of other chares, possibly on other processors, which can simultaneously work to solve the target problem.

In addition, the Charm++ framework provides the programmer with several novel inbuilt features for chare management, object migration, dynamic load balancing and synchronization, about which we discuss briefly below.

*Chare arrays:* An indexed collection, in one or more dimensions, of chare elements. The application can invoke entry methods either on the individual array elements by using their globally unique indexes or collectively on all the elements in the chare array (broadcast), or to a subset (section multicast). The Charm++ runtime treats each element as an individual chares, which means that each individual chare element can physically reside on different processors. If the need arises, the application can directly access the local array element using the *ckLocal* method, which returns a C++ pointer to the element if it exists on the local processor, and `NULL` if the element does not exist or is on another processor. The load balancing framework of Charm++ can redistribute the chare array elements across the available processors depending on the degree of load imbalance. Charm++ provides synchronization mechanisms (the *contribute()* reduction routine) for elements within chare arrays. It also provides support for non-blocking asynchronous reduction and gather operations.

*Chare groups:* These are special type of chare collection which guarantees that there will be a single chare element per processor. These are declared as Groups in the interface file, and inherit from the base Group class definition. In this application, we have used chare groups to perform any per-process initializations.

*Quiescence Detection:* The Charm++ definition of quiescence [5] is the global state of the system where no processor is executing an entry point, and no messages are awaiting processing. To detect quiescence in our program, we can use either the blocking wait function or register for a callback that is invoked on quiescence. We use this mechanism to implement a system wide barrier for our application.

*Dynamic Load Balancing:* Charm++ allows elements in a chare array to migrate from heavily loaded processors to lighter ones. The programmer can specify regular time intervals for migration, or can explicitly invoke the migration calls and register a callback upon completion of load balancing. Charm++ maintains a history of the execution time of each chare and processor and applies one of the several inbuilt, or user defined, load balancing strategies to decide the new processor-chare mapping. Measurement based periodic load balancing [6] is suitable for iterative applications such as CharmSimdemics, where different iterations have different load characteristics and migration is done periodically at the end of some iterations. In this paper, we study the effects of the *GreedyLB* and *RefineLB* load balancing strategies.

*Packing/Unpacking framework (PUP):* This is a data serialization framework that is included within the Charm++ framework. It is used extensively to marshal message parameters across processors, or for serializing chare array elements during migration/load balancing. It can also be used to write chare objects to disk if the application state needs to be check-pointed. All the object elements are packed at the source, and unpacked at the destination. Charm++ defines special syntax to write the PUP routines for the chare objects that need to be migrated or serialized.

## IV. CharmSimdemics: EpiSimdemics on the Charm++ Platform

In this section, we first describe the different entities in CharmSimdemics and the features of Charm++ that we used to develop CharmSimdemics. Next, we introduce the CharmSimdemics algorithm itself and describe how the different entities interact with each other. We also explain how Charm++ can be a better platform for our algorithm, in terms of programmer productivity and performance.

### A. Designing the Chares

The EpiSimdemics algorithm involves iterative message exchanges between the set of person objects and the set of location objects, for the desired number of simulation days. In the MPI implementation, we distributed the person and location objects among the available MPI processes, and message passing occurred by explicitly calling the MPI communication primitives. Each MPI process contains many person objects and many location objects. This means that a system-wide collective communication was always required even if only one group of objects required synchronization. With this approach, different object-to-process mappings can potentially result in different communication patterns, where some are more inefficient than the others. On the other hand, in CharmSimdemics, we first create two types of *chare arrays* called LocationManager and PersonManager to handle location and person objects, respectively, as shown in Figure 4. We then distribute the person and location objects among the chare elements of the corresponding chare arrays. The individual chares in both the chare arrays handle the computation and communication of their location or person objects, respectively. The Charm++ runtime intelligently maps the chares to processes and runs the simulation. In summary, we follow a two-level hierarchical data distribution technique – first we assign people/locations to chares, next we let the Charm++ runtime assign chares to processes. We later show that the intelligent assignment of chares to processes, in conjunction with efficient load balancing strategies, can result in a very efficient communication pattern amongst the simulation objects.

Initialization routines for modules such as loggers and file parsers are called at beginning of the simulation. However, it can be optimized and these modules may be initialized just once per process rather than once per chare, without losing correctness. We create a *chare group* called InitManager for this purpose, where a single chare group object is instantiated in every process, as shown in Figure 4. The per-process initializations are invoked at the beginning of the simulation in every chare group object, so that all the other chares of the system can access the required handles and other resources from their local chare group objects.

The stochastic nature of EpiSimdemics *ideally* requires a unique random stream object for each location and person object in the system to be perfectly repeatable. Repeatability

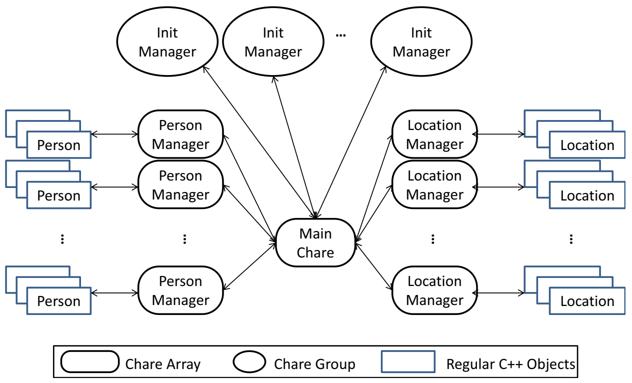


Figure 4. Entities in CharmSimdemics.

is desirable for several reasons: it eases the debugging burden, it ensures that data can be regenerated in the future as long as the configuration files are kept, and it allows branching while keeping the initial portions of the simulation identical. It would take a significant amount of memory to allocate a separate stream for hundreds of millions of objects. To alleviate this problem, the MPI version of EpiSimdemics creates one random stream per process, so that the location and person objects within each process can share random streams. While we can run repeatable simulation when using the same number of MPI processes, the output may change when changing the processor count. Similar limitations apply to CharmSimdemics as well, because we share random number streams within a Chare array element object, which contains multiple people or location objects. With CharmSimdemics, this repeatability is maintained even when a Charm object migrates to different processor. It is a trade-off between the flexibility of simulation and efficiency of memory management. If we maintain the total number of chares in the system a constant, we can potentially have the same number of total random number streams for any number of processors. If the number of processors in the system is changed, the Charm++ runtime will simply create different allocation of chares to processors, but the total chare count (and the total random number streams) remains constant. Thus, we reduce memory overhead yet can produce repeatable simulations.

One of the most useful features of Charm++ is dynamic load balancing via chare migration. Charm++ allows us to specify the phase in the program during which load balancing should occur, and also lets us choose the specific chare arrays that have to be migrated. In CharmSimdemics, we migrate only LocationManager chares at the end of some pre-determined iterations (simulation days). The LocationManager chares send out all the infection messages as soon as they are computed and do not store any state information at the end of each iteration. LocationManager chare elements are the best candidates for migration as they are light-weight and the computation varies dynamically as people change their schedule due to illness or public policy interventions such as school closure. In comparison, the load on PersonManager chares is uniform and static, and, thus, statically distributed. In addition, PersonManager stores quite a large amount of complex state information throughout the simulation process, thus making migration costly. However, even after taking all these points into account, there may still be benefits of migrating PersonManager chares as discussed in Section VII.

## B. The CharmSimdemics Algorithm

The Main chare first creates the LocationManager and PersonManager chare arrays. It then creates the InitManager chare group objects, which are used for per-process initializations. The proxies (or handles) to all the chare array objects are defined as global *readonly* objects (a Charm++ feature), so that any chare can invoke the entry methods on every other chare in the system. Next, the input files (locations, persons and visits) are read by the Main chare and distributed randomly to the appropriate LocationManager or PersonManager chare objects. Once all the input is read in to memory, the simulation process is ready to start. The different stages of a single iteration in the CharmSimdemics algorithm are shown in Figure 5. At the beginning of each simulation day, the person objects compute the visit messages and send them to the location objects, i.e., the PersonManager chare elements send messages to specific LocationManager chare elements (Figure 5a). The location objects compute the infections and send messages to the person objects about the new infections for the current simulation day (Figure 5c). The person objects then decide the locations that have to be visited on the next iteration and the process continues. The Charm++ runtime system implicitly handles queuing, dispatch and processing of messages at the source and destination chares/processes. Moreover all our messages are sent asynchronously, thereby enabling efficient overlap of communication with computation.

The CharmSimdemics algorithm has two global synchronization points per iteration. The first is to ensure that every visit message set from a person object has been received at the destination location before the computation by the location is started. The second is to ensure that every interaction result message sent by a location has been received by the destination person before the persons state is updated. This is done by a combination of a global reduce using the *contribute()* method of Charm++ followed by Quiescence Detection (QD). Using the first synchronization point as an example, each PersonManager chare calls *contribute()* when all of its associated person objects have sent all of their visit messages. When processing resumes after the contribute, it is ensured that all of the visit messages have been sent. This is followed by a QD call. that will finish when there are no messages in flight across the entire system. This ensures that every visit message has been received. The second global synchronization is analogous. It is important to note that both the reduction and the QD can be asynchronous, so additional work can be done during the synchronization, as long as no additional message traffic is generated during the QD. We show in section V that the time taken for QD is less than 5% of the total execution time.

At the end of each iteration, the PersonManager chare array elements resolve all the received infection messages and updates the person state. At the same time, the LocationManager chare elements can be allowed to migrate for load balancing purposes, because they do not have any more computations to perform for the simulation day. The Charm++ runtime can potentially perform the above two actions simultaneously, thereby improving performance. The productivity of the programmer also significantly improves, because the object oriented nature of Charm++ allows the programmer to just focus on the implementations of the individual chare array elements, and leave the communication optimizations to the framework.

## V. Experimental Results

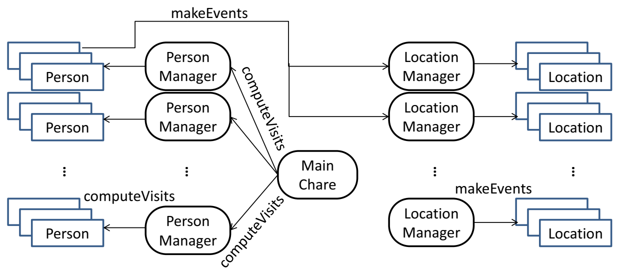
In this section, we first identify the compute-intensive methods in CharmSimdemics. Then, we evaluate the strong and weak scaling performances of CharmSimdemics, and compare them against the MPI implementation of EpiSimdemics. Next, we evaluate the impact of applying load balancing on the performance of CharmSimdemics. We also show the efficacy of the quiescence detection method of synchronization in CharmSimdemics over the regular MPI synchronizations of EpiSimdemics.

### A. Experimental Setup

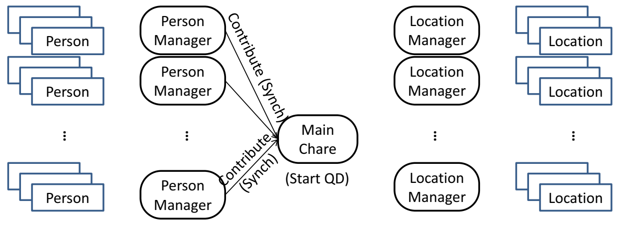
*Hardware:* Our experimental testbed was a high performance computing cluster consisting of 96 compute nodes, where each node had two quad-core Intel Xeon E5440 processors and 16 GB of DDR2 memory. The nodes were connected by 20 Gb/s InfiniBand interconnects.

*Software:* For the MPI implementation of EpiSimdemics, we used the SGI Message Passing Toolkit v1.2 implementation, which is part of the SGI ProPack (version 6) suite of performance optimization libraries and tools. For CharmSimdemics, we use Charm++ v6.3 for most of our experiments, with the exception of the load balancing experiment where Charm++ v6.2 is used (section V-E). This means that we do not enable load balancing in CharmSimdemics by default unless specified otherwise. This is to isolate the effects of multiple optimizations to gain insightful results.

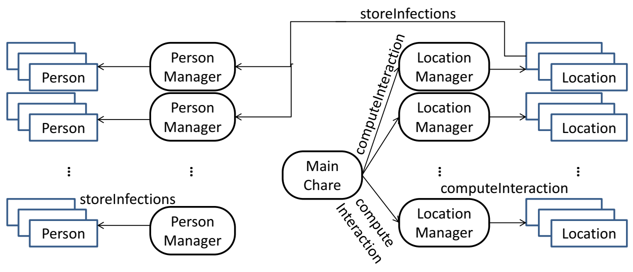
*Data:* The algorithms for both EpiSimdemics and CharmSimdemics simulate the spread of the H5N1 avian influenza virus across social contact networks of various US populations, with sizes between 2.6 million and 244 million. The population data is organized by the states of US. We may choose a single state or a group of states to discuss different



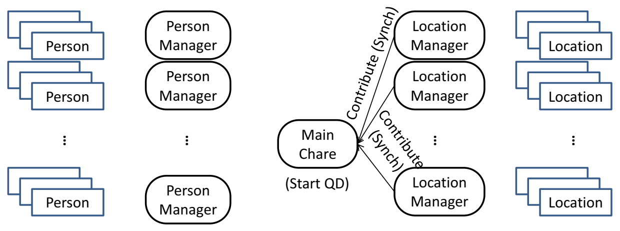
(a) Step 1: Computation and transfer of the visit messages.



(b) Step 2 (Barrier-1): Synchronize with the main chare, wait via *Quiescence Detection* for visit messages to be delivered.



(c) Step 3: Computation and transfer of the infection messages.



(d) Step 4 (Barrier-2): Synchronize with the main chare, wait via *Quiescence Detection* for infection messages to be delivered.

Figure 5. The CharmSimdemics Algorithm: One Simulation Day.

aspects of our experiments. Each test is executed for 120 iterations, where each iteration corresponds to a simulation day. We configure our simulation such that the attack rate, i.e. the ratio of the number of infected people to the total number of people, is roughly 40%.

## B. *Performance Characteristics*

We first investigate which of the CharmSimdemics methods dominate the overall execution time. Although there is a non-trivial amount of communication in CharmSimdemics, it relies on non-blocking communication for exchanging data and overlaps the communication with computation. As a result, the cost of the major data exchange which is to communicate visit schedules is partially hidden except for the cost of creating and handling messages. Such a hidden cost is particularly difficult to estimate when the load is uneven across processors. Here, we focus on analyzing the fully visible costs, the cost of computation, message creation and message handling. In addition, we investigate the cost of synchronization method in Section V-C. In Section V-C and V-D, we further evaluate the scaling performance to understand the impact of the exposed overhead on performance and the benefit of Charm++ framework over MPI. For this comparison, we do not employ the load balancing mechanism provided in Charm++. Finally, we present our case study to show the impact of load balancing using the charm++ framework's built-in methods. Our simulation data for this test was the population of Virginia (6.8 million people), and we used 40 processor cores for execution. We collected the detailed execution time statistics by using the Charm++ *Projections* performance analysis/visualization framework [7]. For this test, we enabled load balancing, but will defer its analysis and discussion to section V-E.

Figure 6 shows the breakdown of the CPU utilization among the selected CharmSimdemics methods from the whole simulation run. Due to the space limitation, we plot the results only for a subset of the processors used. LocationManager's computeInteraction() method takes 46% of the total utilization, and it is the most time consuming. The computeVisits() method sends visit messages to LocationManager Chares and constitute 22% of the total utilization. PersonManager's updateState() method takes 8.2% of the total utilization for updating the person's status at the end of each simulation day. LocationManager's makeEvents() adds the messages received from PersonManager Chares to local data structures and takes 7.8% of the total utilization.

## C. *Effects of Strong Scaling*

In this section we compare the scalability of EpiSimdemics and CharmSimdemics with a fixed problem size. We

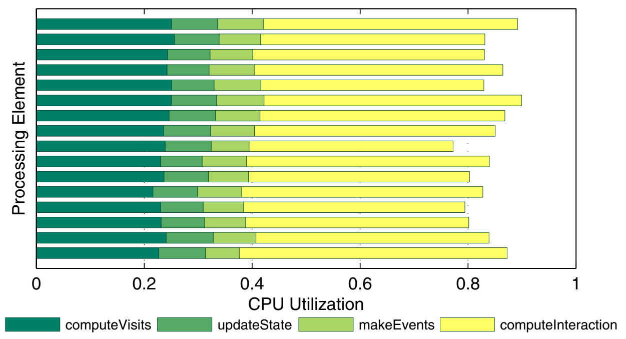


Figure 6. The breakdown of CPU utilization identifies compute-intensive Charm++ methods. On average, computeInteraction() takes 46%, computeVisits() takes 22%, updateState() takes 8.2% and makeEvents() takes 7.8% of CPU utilization.

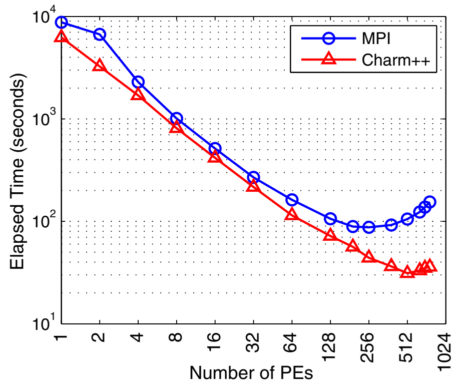


Figure 7. The comparison of strong scaling performance between two different implementations of EpiSimdemics. The MPI version peaks at 192 PEs while the Charm++ version scales up to 512 PEs.

use the Arkansas population of 2.6 million people as it is the largest data that a single compute node can accommodate in our experimental setup. Figures 7 and 8 show the total execution time and the speedup, respectively, for the two implementations. We use the runtime of the serial version of EpiSimdemics as the basis for calculating speedup. For this experiment and the one discussed in V-D, load balancing is not enabled and we assign one chare of each type per processor. For EpiSimdemics, the simulation scales only up to 256 PEs, and gains a speedup of up to 82 which is 32% of the ideal speedup. However, the scalability of CharmSimdemics extends to more PEs (peaks at 512 PEs), and achieves a speedup of 229 which is 45% of the ideal speedup. Beyond these scaling peaks, using more PEs does not achieve better performance. As the data set is split among more and more processors, it comes to a point where each processor will not have enough data to generate a sufficient amount computation to amortize the costs of communication and synchronization [8].

Figure 8 and 9 shows that CharmSimdemics achieves 2.8X speedup compared to the best gained by EpiSimdemics,

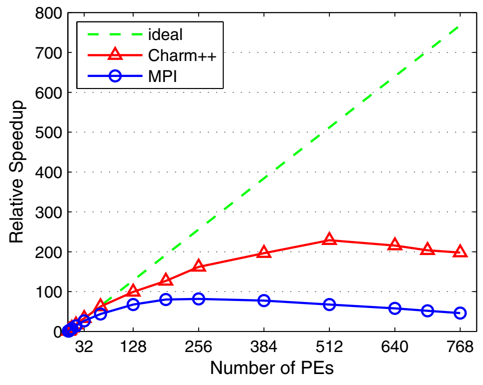


Figure 8. Strong Scaling comparison of EpiSimdemics and CharmSimdemics for Arkansas data. CharmSimdemics achieves 2.8X speedup compared to the best gained by EpiSimdemics, and shows up to 3.9X speedup with 768 PEs used.

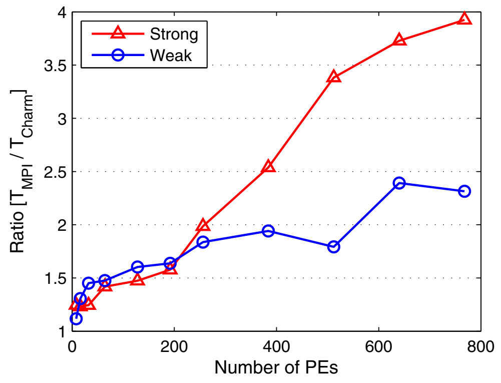


Figure 9. The improvement of CharmSimdemics over EpiSimdemics. CharmSimdemics achieves up to 3.9x performance of EpiSimdemics with Arkansas data of 2.6 millions, and 2.4x performance with 200 million population data.

and shows up to 3.9X speedup with 768 PEs used. This improvement mostly comes from the more efficient synchronization mechanism (contribute() and QD) and the improved communication.

Figure 10 shows the synchronization cost of both implementations relative to the total execution time. As discussed in Section IV, CharmSimdemics relies on contribute() and QD for synchronization. Note that the synchronization cost of CharmSimdemics does not increase as the MPI synchronization cost does with increasing number of PEs. The MPI synchronization cost increases linearly with the number of PEs used while the 'contribute() and QD' method does not. The synchronization overhead using contribute() and QD method takes at most 4.23% of the total execution time while the MPI synchronization cost increases up to 14.5%.

## D. Effects of Weak Scaling

To study the scalability of the Charm++ implementation for large social contact network data, we increase the input data size as we increase the number of PEs such that the

Table I
Data chosen for the weak scaling experiment shown in Figure 11. As we increase the number of PEs, we use the data of larger population such that each PE simulates roughly 314,360 people.

| States | PEs | Individuals | | Locations | Visits | Execution Time (sec) | | | | Charm++ Improvement |
|---|---|---|---|---|---|---|---|---|---|---|
| | | total | per-PE | | | pre-normalized | | normalized | | |
| | | | | | | MPI | Charm | MPI | Charm | |
| ar | 8 | 2,605,954 | 325,744 | 683,978 | 14,376,852 | 1,013 | 908 | 978 | 876 | 1.12 |
| ut,ia | 16 | 5,029,760 | 314,360 | 1,314,623 | 27,627,997 | 1,158 | 887 | 1,158 | 887 | 1.31 |
| tn,mn | 32 | 9,333,984 | 331,524 | 2,362,346 | 51,478,432 | 1,238 | 854 | 1,335 | 920 | 1.45 |
| tx | 64 | 20,345,848 | 317,904 | 4,482,514 | 111,372,662 | 1,412 | 957 | 1,396 | 947 | 1.48 |
| tx,il,nc | 128 | 40,279,841 | 314,686 | 9,664,024 | 220,882,205 | 1,582 | 986 | 1,580 | 985 | 1.60 |
| tx,il,nc,ga,wa,mo | 192 | 59,469,571 | 309,737 | 14,800,992 | 326,457,893 | 1,739 | 1,062 | 1,765 | 1,078 | 1.64 |
| tx,il,nc,ga,wa,mo,mi,nj,ct | 256 | 80,287,853 | 313,624 | 20,012,407 | 439,186,388 | 2,523 | 1,373 | 2,529 | 1,377 | 1.84 |
| tx,il,nc,ga,wa,mo,mi,nj,ct, tn,mn,sc,ky,md,az,wi,in,id | 384 | 119,871,849 | 312,166 | 30,160,072 | 656,422,335 | 2,459 | 1,266 | 2,476 | 1,275 | 1.94 |
| tx,il,nc,ga,wa,mo,mi,nj,ct, tn,mn,sc,ky,md,az,wi,in,id, ut,ks,ms,ma,ia,ok,or,co,al, la,wv,nm | 512 | 159,402,940 | 311,334 | 40,231,107 | 873,877,636 | 2,946 | 1,644 | 2,975 | 1,660 | 1.79 |
| me,tx,il,nc,ga,wa,mo,mi,nj, ct,tn,mn,sc,ky,md,az,wi,in, id,ut,ks,ms,ma,ia,ok,or,co, al,la,wv,nm,ca,va | 640 | 200,633,822 | 313,490 | 49,299,145 | 1,062,782,842 | 4,146 | 1,734 | 4,158 | 1,738 | 2.39 |
| me,tx,il,nc,ga,wa,mo,mi,nj, ct,tn,mn,sc,ky,md,az,wi,in, id,ut,ks,ms,ma,ia,ok,or,co, al,la,wv,nm,ca,va,ny,fl,oh | 768 | 243,425,288 | 316,960 | 60,951,207 | 1,268.675,372 | 4,906 | 2,120 | 4,866 | 2,102 | 2.31 |

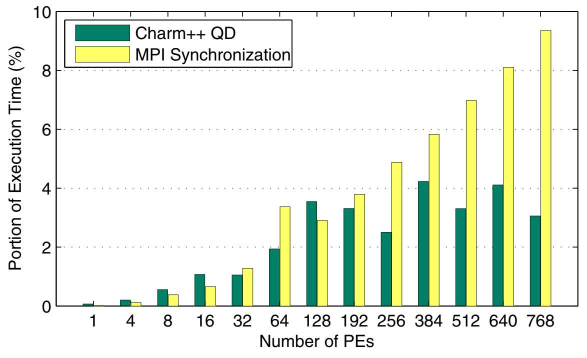


Figure 10. The synchronization cost using contribute() and QD method takes at most 4.23% of the total execution time while the MPI synchronization cost linearly increases up to 14.5% as the number of PEs used increases for simulating Arkansas population.

population per PE remains constant (which we denote as $POP_{target}$). We also want to make sure that $POP_{target}$ to be as large as possible while not exceeding the size of the memory of a single processing element. Since the Arkansas data of 2.6 million fits on a single node and there are 8 cores per node, we choose to use $POP_{target} = 314,360$ which is roughly close to $2.6 \times 10^6/8$. For a given number of PEs, we pick a certain set of states such that the total population from the chosen states is close to $POP_{target} \times N_{PE}$, where $N_{PE}$ is the number of PEs used. Since the size of the data from the different states is not likely a multiple of $POP_{target}$, the population per PE varies within 5.6% as we pick a different number of PEs and different set of data to run. Thus, for comparing the performance with data of different sizes, we normalize an execution time $T_{org}$ taken to run $POP_{actual}$ sized data by calculating the representative execution time as $T_{rep} = T_{org} \times POP_{target}/POP_{actual}$. Further details of data sets are listed in Table I. It lists the total population of data, $POP_{actual}$, $T_{org}$, $T_{rep}$ as well as the ratio of execution time of EpiSimdemics to CharmSimdemics in the last column. Figure 11 shows the weak scaling performance of MPI and Charm++ implementations, and Figure 9 compares them. When we use 768 PEs (96 nodes), the problem size is 96 times larger than when we use 8 PEs (1 node). As the per-PE population is the same, ideally, the execution time should be constant. We calculate this constant ideal time $T_{ideal}$ from the serial execution time $T_{serial}$ of EpiSimdemics for Arkansas data. Since the Arkansas data contains roughly eight times larger population than $POP_{target}$, we calculate $T_{ideal}$ to be 811 seconds by the normalization as $T_{serial} \times POP_{target}/POP_{actual}$. CharmSimdemics executes 2.6 times longer than $T_{ideal}$ using 768 PEs, EpiSimdemics takes 6 times longer. As a result, CharmSimdemics executes 2.3 times faster than EpiSimdemics with 768 PEs.

## E. *Effect of Load Balancing*

The Charm++ framework migrates compute-intensive tasks (chare objects) from heavily loaded processors to less utilized processors. In Charm++, load balancing can be done in a centralized, fully distributed or hierarchical fashion. With the centralized approach, the machine's load and communication structure are accumulated to one processor, followed by a decision process that determines the new distribution of Charm++ objects at given synchronization points. We are using a centralized approach, as it best suits

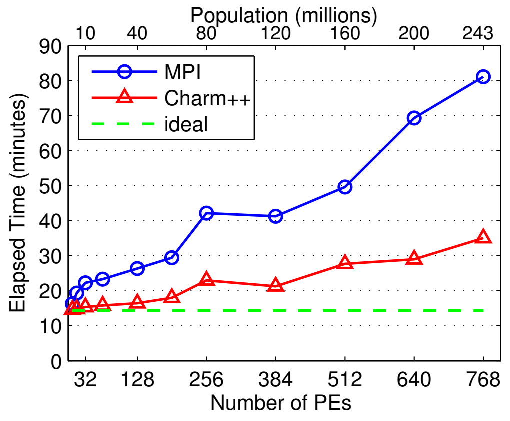


Figure 11. While CharmSimdemics executes 2.4 times longer than the ideal using 768 PEs, it shows 2.3× improvement in execution time when compared to EpiSimdemics (MPI) with 768 PEs.

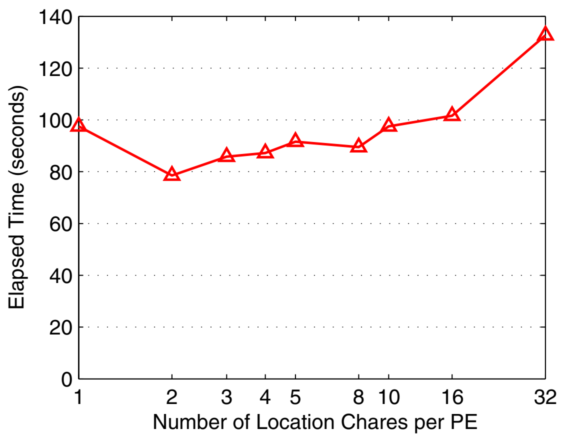


Figure 12. Impact of the number of chares per PE used on load balancing. For simulating the North Carolina population, the execution takes 98 seconds using total 128 PEs and a single chare per PE (without load balancing). With two chares per PE, it takes 78 seconds which is about 81% of that without load balancing.

to applications that iterates over multiple phases. We use the *Greedy* and *Refine* centralized load balancing strategies in our experiments. GreedyLB uses a greedy algorithm to assign chares to processors. It does not attempt to minimize the amount of chare migration that occurs. On the other hand, RefineLB limits the number of chares migrated by shifting chares from highly utilized processors to under utilized processors. As our first step of load migration in our application, we are migrating LocationManager chares only, as they are lightweight and easy to migrate. There should be several chares on one processor for the migration to happen from one processor to another. At the same time, some overhead is incurred with increasing the number of chares on processor. Figure 12 shows the increase in execution time for increased number of chares. For our load balancing experiments we use two LocationManager chares per processor.

Figure 13 shows the effect of load balancing strategies applied when simulating the Arkansas population data. This diagram is again collected using the Charm++ Projection tool [7]. The horizontal bars show PEs, while the colors show methods and idle times. Since there is a large overhead with GreedyLB, we apply it only once after the first iteration and follow it up by a RefineLB step after every third iteration. The dark orange color shows the time taken by computeInteraction() and the white following it show the idle time. Three out of twelve processors in the first iteration have extended idle times (white lines) which mean that the computational loads of computeInteraction() is not even and other PEs are waiting on them to finish their compute step. At the beginning of the second iteration, GreedyLB (shown by red color) is applied, which aggressively migrates objects from heavily loaded processors to less utilized processors. In the second iteration the load is more balanced compared to the first iteration (small wait times after computeInteraction). We further apply RefineLB after every third iteration to reduce load imbalance through the iterations.

Figure 15 shows the CPU utilization by CharmSimdemics methods, the load balancing overhead and idle times over two simulation days. Note that the communication of computeVisits() overlaps the computation of makeEvents() methods. The remote entry method makeEvents() called within computeVisits() to send visit messages from PersonManger to remote LocationManager is a non-blocking call and put into the runtime queue until the CPU resource becomes available on the remote processors. There are also less compute intensive methods, but because of their smaller proportion to total utilization they are not visible. We can see the overhead from GreedyLB step (shown by red color) after the first iteration. We can also notice in the iteration after the GreedyLB is applied, that the computeInteraction() load is more balanced across all the processors. The black color show the overhead associated with GreedyLB and the unaccounted time taken by Charm++ framework. The gaps (white in color) between utilization sections show the idling during synchronizations. Figure 10 shows the time taken by Quiescence detection (synchronization), running Arkansas data with one chare of each type per process. The QD time is maximum (4.3%) when running on 348 processors. This shows that QD is not taking a lot of time even when running large number of processors.

To get a good view of the benefits of load balancing and show the individual contribution of Greedy and Refine step, we show time per iteration for simulation running over 60 iteration days. For this experiment we used GreedyLB and a combination of GreedyLB and RefineLB. Figure 14 shows the time/iteration performance of load balancing strategies compared to not using it. All the results in this figure are collected for North Carolina data (6.8 million population) on 128 processors with two chares per process. 'No LB'

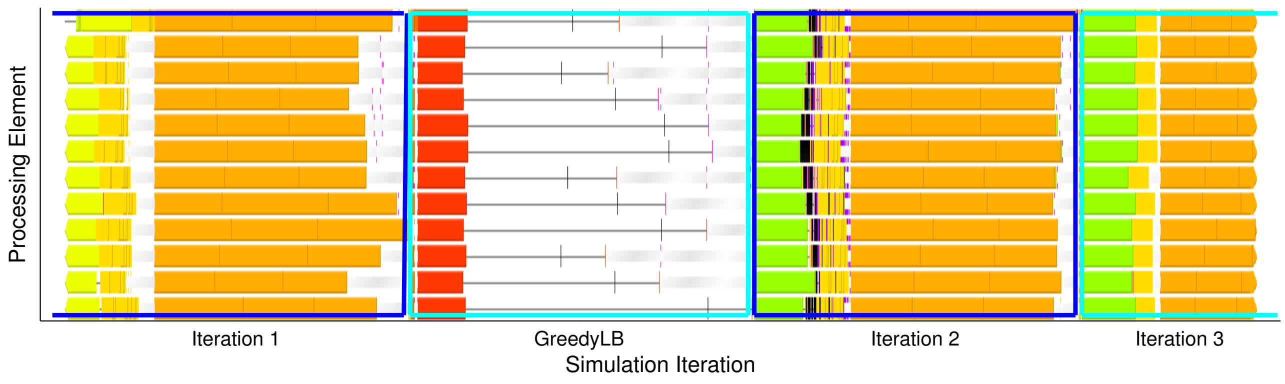


Figure 13. Effect of Load Balancing: heavily imbalanced in Iteration-1 (three PEs have more load than the others). GreedyLB at the start of second iteration reallocates chares to processes with lighter loads. Iteration-2 shows better load distribution.

means that no load balancing is happening. 'Greedy once' means GreedyLB is applied once after the first iteration. 'Greedy+Refine' means that GreedyLB applied after the first iteration followed by RefineLB every 10th iteration. The longer times for the second iteration show the overhead associated with GreedyLB step. Notice the reduction in time per iteration when GreedyLB is applied at the second iteration. In case of GreedyLB, load balancing happens only once in the entire simulation. However, for Greedy+Refine, the GreedyLB in second iteration is followed by refine steps every tenth iteration. This can be seen at the start of 11th iteration when there is visible drop in time per iteration after that. The same refine step effect can be seen in iterations 21, 31, 41 and 51. Cumulative timings show that GreedyLB when used only once after the first iteration gives about 20 percent performance improvement compared to not using the load balancing (No LB). Similarly, the performance improvement of GreedyLB+RefineLB scheme compared to not using load balancing is more than 24 percent.

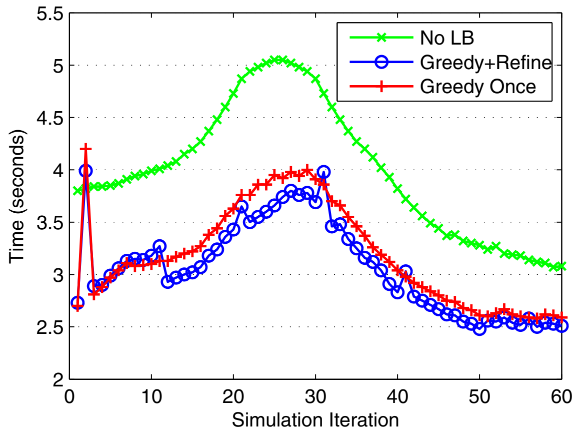


Figure 14. Performance gain for different load balancing strategies for NC data on 128 PEs. Total simulation time when not using load balancing is 240 seconds, GreedyLB applied only once after the 1st iteration is 191 seconds and GreedyLB followed by refine steps every 10th iteration is 181 seconds

The overall contribution of CharmSimdemics load balancing to improvement on the weak scaling performance over that of EpiSimdemics is shown in Figure 16. We use the GreedyLB+RefineLB combination among those supported by the Charm framework. Enabling the load balancing feature of CharmSimdemics further improves the performance by 10% on average and up to 15% compared to that of EpiSimdemics implemented in MPI without load balancing. The load balancing helps especially when load distributed by the initial static method is not even. In our case, it helps especially when we use 256 PEs making additional 15% improvement. On the other hand with 64 PEs, the load balancing makes minor contribution less than 1% additional improvement. When the improvement by CharmSimdemics peaks with 640 PEs, the load balancing brings 8.5% additional improvement.

## VI. Related Work

Among many epidemiological ABS platforms are those developed by Eubank et al. [9], [10], Longini et al. [11], Ferguson [12], and Parker et al. [13]. The system described by Ferguson is implemented to be executed on a shared memory platform and, thus, is limited by the amount of available shared memory. Emulated shared memory machines can be used, but very few machines at present exist that can store very large social networks in such a form. The work of Longini et al. is a parallel simulation that uses a very simple and structured social contact network. The locations in these social contact networks are not real but simply surrogates for simple location types such as school, home, etc. This results in a structured social contact network that is more amenable to efficient parallel computation, but which, arguably, is less representative of real-world social networks. The simulator developed by Parker et. al. is implemented

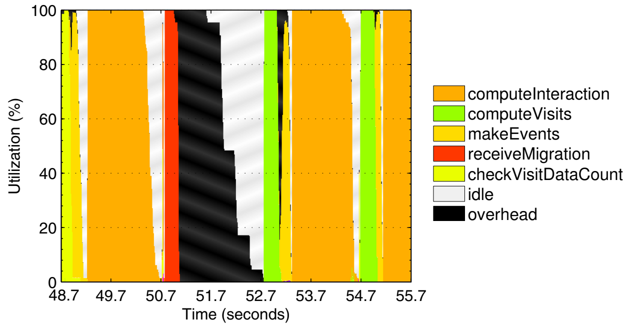


Figure 15. Waterfall graph shows CPU utilization across all processes by methods. Red shows the load migration while the Black shows the overhead from load migration plus the unaccounted time used by the Charm++ framework. The white areas show processor idle times.

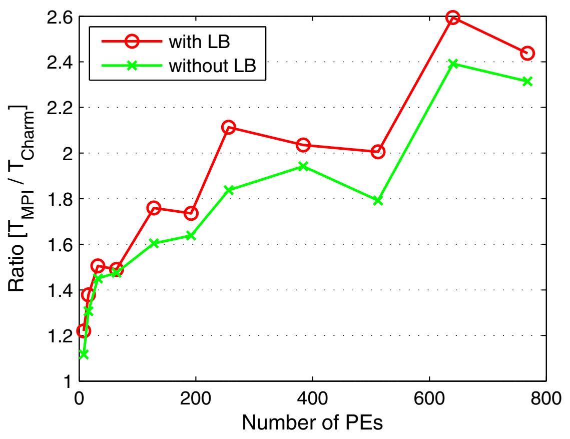


Figure 16. Enabling the load balancing feature of CharmSimdemics further improves the weak scaling by 10% on average and up to 15% compared to that of EpiSimdemics implemented in MPI without load balancing.

in Java with numerous optimizations. It is a combination of spatial model (dividing the region into pixels) and agent model with randomly constructed contact networks. While it quite efficiently simulates very large populations, it currently does not support necessary interventions required in practical public health studies.

There has also been a wealth of recent work on general-purpose simulators that utilize a range of dynamics models. Among these are AnyLogic, Aurora2, BRACE, $\mu$sik, NetLogo, Repast SC++, SASSY, and Swarm [14]–[18]. These simulators span the range of smaller-scale ones with accompanying visual tools (e.g., NetLogo) to some of the largest simulations in terms of computing resources used (as many as 65000 processing cores [19], [20]). Graphics processing units (GPU) have also been used [21]–[23], simulating hundreds of millions of agents.

## VII. Conclusions and Future Work

We have developed a scalable system that models the diffusion of diseases through an evolving social network. The initial version of this system out-performs the MPI based code when the number of cores is increased. Our short-term scaling goal is to efficiently simulate the entire country (300 million people) on 5000 processors. In long term, we plan to intelligently map persons and locations to chares to reduce the communication load, migrate the PersonManager's load for the purpose of load balancing and do dynamic load balancing at run-time that takes advantage of both computation load and communication patterns observed. We also intend to use Charm++ structured daggers to remove global synchronization barriers and overlap iterations.

## Acknowledgment

We thank our external collaborators and members of the Network Dynamics and Simulation Science Laboratory (NDSSL) and Parallel Programming Laboratory (PPL) for their suggestions and comments. This work has been partially supported by NSF Nets Grant CNS-0626964, NSF HSD Grant SES-0729441, NSF PetaApps Grant OCI-0904844, NSF NETS Grant CNS-0831633, NSF REU Supplement Grant CNS-0845700, NSF Netse Grant CNS-1011769, NSF SDCI Grant OCI-1032677, DTRA R&D Grant HDTRA1-0901-0017, DTRA CNIMS Grant HDTRA1-07-C-0113, DOE Grant DE-SC0003957, US Naval Surface Warfare Center Grant N00178-09-D-3017 DEL ORDER 13, NIH MIDAS project 2U01GM070694-7 and NIAID & NIH project HHSN272201000056C.

## References

[1] Message passing interface forum. [Online]. Available: http://www.mpi-forum.org

[2] C. L. Barrett, K. R. Bisset, S. G. Eubank, X. Feng, and M. V. Marathe, "Episimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[3] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[4] K. Bisset, X. Feng, M. Marathe, and S. Yardi, "Modeling interaction between individuals, social networks and public policy to support public health epidemiology," in *Proceedings of the 2009 Winter Simulation Conference*, Dec. 2009, pp. 2020–2031.

[5] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[6] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

[7] Charm++ projections: A performance analysis/visualization framework. [Online]. Available: http://charm.cs.uiuc.edu/manuals/html/projections/manual.html

[8] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, vol. 30, 1967, pp. 483–485.

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

[10] S. Eubank, H. Guclu, M. V. Marathe *et al.*, "Modelling disease outbreaks in realistic urban social networks," *Nature*, vol. 429, no. 6988, pp. 180–184, May 2004.

[11] I. Longini, A. Nizam *et al.*, "Containing pandemic influenza at the source," *Science*, vol. 309, no. 5737, pp. 1083–1087, 2005.

[12] N. M. Ferguson, M. J. Keeling *et al.*, "Planning for smallpox outbreaks," *Nature*, vol. 425, no. 6959, pp. 681–685, 2003.

[13] J. Parker, "A Flexible, Large-Scale, Distributed Agent Based Epidemic Model," in *Proceedings of the 2007 Winter Simulation Conference*, 2007, pp. 1543–1547.

[14] K. Perumalla, "$\mu$sik: A Micro-Kernel for Parallel/Distributed Simulation Systems," in *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005, pp. 185–192.

[15] M. Hybinette, E. Kraemer, Y. Xiong, G. Matthews, and J. Ahmed, "SASSY: A Design for Scalable Agent-Basd Simulation System Using a Distributed Discrete Event Infrastructure," in *Proceedings of the 2006 Winter Simulation Conference*, L. Perrone, F. Wieland, J. Liu, B. Lawson, D. Nicol, and R. Fujimoto, Eds., 2006, pp. 926–933.

[16] A. Park and R. Fujimoto, "Efficient Master/Worker Parallel Discrete Event Simulation," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009, pp. 145–152.

[17] M. North and C. Macal, "Foundations of and Recent Advances in Artificial Life Modeling with Repast 3 and Repast Simphony," in *Artificial Life Models in Software*, A. Adamatzky and M. Komosinski, Eds. Springer, 2009, pp. 37–60.

[18] G. Wang, M. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White, "Behavioral Simulations in MapReduce," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 952–963, 2010.

[19] K. Perumalla and S. Seal, "Reversible Parallel Discrete-Event Execution of Large-Scale Epidemic Outbreak Models," in *Proceedings of the 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2010.

[20] C. Carothers and K. Perumalla, "On Deciding Between Conservative and Optimistic Approaches on Massively Parallel Platforms," in *Proceedings of the 2010 Winter Simulation Conference*, 2010.

[21] R. D'Souza, M. Lysenko, and K. Rahmani, "SugarScape on Steroids: Simulating over a Million Agents at Interactive Rates," in *Proceedings of the Proceedings of Agent2007 Conference*, 2007.

[22] B. Aaby, K. Perumalla, and S. Seal, "Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Core Clusters," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, 2010.

[23] W. Hwu, *GPU Computing Gems*. Elsevier Morgan Kaufmannnce, 2011, see "Chapter 21. Template-Driven Agen-Based Modeling and Simulation with CUDA" by P. Richmond and D. Romano.