Large-Scale Agent-Based Modeling with Repast HPC: A Case Study in Parallelizing an Agent-Based Model

Nicholson Collier[1,2(✉)], Jonathan Ozik[1,2], and Charles M. Macal[1]

[1] Global Security Sciences, Argonne National Laboratory, Argonne, IL, USA
{ncollier, jozik, macal}@anl.gov
[2] Computation Institute, University of Chicago, Chicago, IL, USA

Abstract. We present a case study for parallelizing a large-scale epidemiologic ABM developed with Repast HPC, the Chicago Social Interaction Model (chiSIM). The original serial model is a CA-MRSA model which tracks CA-MRSA transmission dynamics and infection in Chicago, and represents the spread of CA-MRSA in the population of Chicago. We utilize both within compute node parallelization using the OpenMP toolkit and distributed parallelization across multiple processes using MPI. The combined approach yields a 1350 % increase in run time performance utilizing 128 compute nodes.

Keywords: Agent-based modeling · ABMS · Repast HPC · Parallel simulation · Distributed simulation · High performance computing

1 Introduction

Agent-based models with large populations of agents can be prohibitively slow to execute. The threshold where slow becomes too slow is dependent on the types of questions that the model is intended to answer and the overall experimental design needed to provide those answers. Adequate results may be achieved within a limited period of time. However, to serve as useful *electronic laboratories*, computational models often require a battery of experimental analyses, such as adaptive parametric studies; large scale sensitivity analyses and scaling studies; optimization and meta-heuristics; inverse modeling; uncertainty quantification; and data assimilation. Sophisticated statistical techniques are increasingly being used for some of these types of analyses with ABMs [1], but due to the large number of simulations that these analyses require, efforts have been hampered by performance considerations. Some researchers have gone so far as to suggest that: "the way for ABM to become an accepted research method is not to make the models as realistic and complex as possible. An important, if not decisive, design criterion for ABMs, as well as any other simulation models, is that they must run quickly enough to be amenable to comprehensive and systematic analysis." [1] However, such comments also suggest that, if adequate performance gains can be achieved, these statistical techniques can be used for "realistic and complex" ABMs.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 454–465, 2015.
DOI: 10.1007/978-3-319-27308-2_37

Given this necessity of running many simulations in a reasonable amount of time, the problem then is how to keep execution time manageable while scaling up to larger numbers of agents. The execution time of a model run, indeed its ability to run at all, is dependent on the design of the model itself, that is, on the "richness" of the agents, the computational complexity of their behavior, and the number of agents executing that behavior. A typical serial ABM iterates through all the agents at each time step and has each agent execute some behavior. The time to execute this loop is obviously dependent on the number of agents the loop has to iterate through and the complexity of the agent behavior. Consequently, as an ABM scales up to populations of increasing size and complexity, it is more susceptible to prohibitive increases in execution time as result of the time it takes to complete this behavior loop. Additionally, it is also susceptible to more mundane memory issues, ultimately limiting the number of possible agents in a model.

The solution that is utilized in this paper is to parallelize the ABM. In parallelizing the behavior loop, the agents are divided into n groups. Each group is concurrently iterated over and each agent executes its behavior. This type of parallelization can be performed both within a compute node via threading and also by distributing the agents among compute nodes, each of which has its own memory capacity. By distributing the agents among compute nodes, we also reduce any potential memory pressure as the number of agents per compute node is now lower. The remainder of this paper is a case study of how a large-scale non-parallelized ABM, a model of community associated methicillin-resistant *Staphylococcus aureus* (CA-MRSA), was parallelized and distributed to become the scalable general epidemiological Chicago Social Interaction model. Section 2 details some specifics of the CA-MRSA model. Section 3 presents the details of the parallelization process. We conclude in Sect. 4. The source code referenced in the paper is presented as a set of Github gists [2].

2 The CA-MRSA Model

The CA-MRSA Model [3] is an agent-based model of community associated methicillin-resistant *Staphylococcus aureus* (CA-MRSA) transmission. *Staphylococcus aureus* is a common cause of human bacterial infections. It is estimated that 25–40 % of the population are colonized in the nasopharynx at any given time. While colonization is asymptomatic, colonized individuals may develop an active infection [3]. Colonization may be of short or long duration, and often clears without causing an infection. Transmission is believed to be largely via skin-to-skin contact with either a colonized or infected individual. In the 1960s new antibiotic resistant isolates of *S. aureus* were identified among hospitalized patients. Called methicillin-resistant *Staphylococcus aureus* (MRSA), these isolates rapidly became an important cause of infections within healthcare settings. These are referred to as health-care associated MRSA (HA-MRSA) [4].

In the 1990s new MRSA strains that were distinct from HA-MRSA were identified in individuals who did not have healthcare exposures. These are referred to as CA-MRSA. The problem of CA-MRSA has been particular severe in the United States where incidence increased exponentially in the early 2000s [5]. By 2004, one of the

CA-MRSA strain types, USA300, became the most frequent cause of skin and soft tissue infections seen in U.S. emergency departments [6]. While uncomplicated skin infections are the most common manifestation, CA-MRSA infections may be invasive and even fatal. In 2005, the Centers for Disease Control and Prevention (CDC) estimated that there were more deaths from invasive MRSA infections than from HIV-AIDS in the U.S. [7].

Epidemiologic studies can be used to estimate the incidence and risk factors for CA-MRSA infections, but it is difficult to perform a study that examines the actual transmission dynamics within a population. The CA-MRSA model [3] was created to aid in the understanding of CA-MRSA transmission dynamics and infection in Chicago, and represents the spread of CA-MRSA in the population of Chicago based on clinical data from 2001 to 2010. The model was used to determine the types of places most important for the transmission of CA-MRSA, such as households, the County jail, hospitals and schools and the relative contributions of the colonized and infected states to transmission.

2.1 Model Structure

The full Chicago CA-MRSA model represents every person in the City of Chicago as an agent. The baseline data for this agent population is adapted from a synthetic population derived from U.S. Census files [8]. Each of the approximately 2.9 million agents resides in a household, dormitory or retirement home/long term care facility and moves among other places such as schools, workplaces, hospitals, jails and sports facilities. Each simulated hour about 2.9 million agents move to and from 1.2 million of these places, where the places are further characterized by a level of risk-of-transmission, primarily differentiated by levels of physical contact. The risk of transmission between agents is determined by the disease status (uncolonized, colonized, or infected) of other agents co-located in the same places modified by the place's level of risk.

Agents move hourly between places according to their shared activity profiles. Each agent has a profile that determines what times throughout the day he or she occupies a particular location. The profile may be overridden for two reasons: (1) an agent may seek care in a hospital for MRSA treatment, or (2) an agent may be jailed. For further information on how the agents' activity profiles, health seeking behaviors and jailing probabilities were derived from various empirical data sources, see [3].

Once in the appropriate place determined by the activity profile, the agent has contact with the other persons co-located in that place and disease transmission from one agent to another can occur. In the full Chicago model for a typical simulated 10 year run, these model dynamics yield approximately 3 trillion individual contacts.

Disease transmission consists of an uncolonized individual becoming colonized with a base probability modified by the place and activity risk as well as the co-location of colonized or infected persons. In the absence of any colonized or infected persons, transmission is not possible. More information on the CA-MRSA model disease state dynamics, estimates of population level colonization, decolonization and infection rates, and household infection rates can be found in [3].

2.2 Implementation

The CA-MRSA model was originally prototyped using Repast Simphony ReLogo [9] for a small region of Chicago and then ported to C++ using the Repast HPC [10] toolkit. The major components of the model, persons and places, are implemented as C++ classes with additional classes to represent model components such as the shared activity profiles, and activities. The base input data consists of a set of files that define persons, places and activity profiles together with a properties file that defines the model's parameters.

Model execution consists of two phases: an initialization phase and the repeated hourly time step. The initialization phase consists of reading in the various input files and creating the appropriate C++ objects (Persons, Places, etc.) from them. The hourly time step executes the code for the next simulated hour and is run repeatedly until a specified number of hours have been completed. Pseudo-code for the CA-MRSA model hourly time step is shown in Fig. 1.

```
For each person:
  Move to the place for current hour's activity
For each place:
  Run the transmission and transition algorithms
```

Fig. 1. Pseudo-code for the CA-MRSA model hourly time step.

The CA-MRSA model is a serial (non-parallelized) model. The two loops in Fig. 1 are executed serially and parallelizing them is the focus of the work described in the next section.

3 From the CA-MRSA Model to the Social Interaction Model

The CA-MRSA model was a success and yielded useful results, producing temporal and geographic trends that match the increase in CA-MRSA incidence similar to Chicago from 2001 to 2010 [3], and providing insights into the role of households, schools, and other places in the spread of the disease [3]. It was and is a useful platform for exploring alternate treatment strategies such as treating children in schools where infections or colonizations have been detected, work on which is currently on-going. However, the model is slow to execute, taking approximately 60 h to simulate 10 years with 1 h time steps, making it time consuming and difficult to do any involved computational experimentation such as the sensitivity analyses and parameter estimation methods mentioned in the introduction. The majority of the model runs were done on the Blues and Fusion clusters (310 nodes with 4960 available compute cores and 320 nodes with 2560 available compute cores, respectively) hosted at Argonne National Laboratory. As with most cluster systems, a user submits compute jobs to the cluster and then waits for the results. Given the nature of the job scheduler, the longer the job,

the longer it may wait in the queue and thus the actual time to get results back is in fact longer than the 60 h of simulation run time.

Regardless of the lengthy execution time, we recognized that the model contained the core of what could become a more general epidemiological model. From the CA-MRSA model, we then derived the Chicago Social Interaction model (chiSIM). chiSIM retains the synthetic population, activity profiles and shared places of the CA-MRSA model and removes the MRSA specific disease transmission, hospitalization and treatment protocols. In this way, chiSIM simulates the hourly mixing and interactions of a population and can easily be extended to model the epidemiological consequences of such activities. Implementation-wise the hourly time step remains the two loops presented in Fig. 1, but the second loop is now a placeholder for the epidemiological consequences of co-location in a place (Fig. 2).

```
For each person:
  Move to the place for current hour's activity
For each place:
  Compute the epidemiological consequences of co-location
```

Fig. 2. Pseudo-code for the simplified chiSIM time step.

We began the parallelization work by profiling chiSIM to find the "hotspots" where the majority of the execution time was taking place. This initial profiling was performed on a desktop machine running Mac OS X using Apple's Instruments profiler [11] with a 3 zip code (post code) smaller version of the model. It indicated, not entirely unexpectedly, that the person loop in the time step consumed the majority of the execution time. This was linear with respect to the number of persons in the loop. What was unexpected was the amount of time spent in the place loop. Despite being essentially an empty placeholder for any disease specific code, the place loop did remove any person objects in each of the places as a final clean up step. This step resolved into multiple `clear()` calls on C++ standard library vectors that accounted for the time consumed.

3.1 OpenMP

Given the profiling data, the obvious candidates for optimization through parallelization were the person and place loops. We began by using OpenMP, a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs [12]. In particular, we used the OpenMP, *parallel for loop* directive to parallelize both the person and place loops. The resulting source code (Loops_OMP.cpp) can be seen in [2]. The OpenMP compiler directives begin with `#pragma omp` (lines 3, 7, and 14). The parallel loop directive `#pragma omp for` (line 14) parallelizes the loop that follows the directive (lines 15–17). Following this directive the compiler parallelizes the loop by spawning *n* number of threads and dividing the loop iterations among those threads, executing the iterations in parallel.

The person loop (lines 4–11) is more complicated due to the typical constraints of multi-threaded programming, concurrent access of shared memory, and so forth. In the person loop each person object selects its next activity and then moves to the place indicated by that activity. As with the place loop, the person loop is parallelized using the compiler directive (line 3). In selecting the next activity each person requires a `NextAct` object that can be re-used serially but not concurrently. The `firstprivate (next_act)` phrase in the compiler directive (line 3) directs the compiler to create one `NextAct` per thread, preventing concurrent access. In the loop, only the activity selection can be performed in parallel. The selection of the next activity is essentially a thread-safe read on shared activity profiles together with writes on the now thread safe `NextAct` object. However, the actual movement of the person to the place associated with the selected activity (line 9) is not thread safe given that the places are shared among all the persons and a concurrent write to that place would corrupt it. Consequently, the code is surrounded by a `#pragma omp critical` directive (lines 7–10) that insures that only a single thread executes that section at one time.

Parallelizing with OpenMP in this way did result in a faster execution, the speed increase was not linear: running with 16 threads did not result in a 16-fold speed up. Profiling, as might be expected, revealed that the presence of the critical section did tend to mitigate the speed increases provided by parallelizing the person loop. The place loop, however, did show a linear speed increase.

3.2 Parallelizing the Model with MPI

The next phase in parallelizing chiSIM was to distribute the model among multiple processes. Commodity CPUs, at the time of writing, may have up to 16 hardware threads, meaning that OpenMP parallelization could under the best conditions parallelize our loops into 16 concurrent iterations. By distributing the model across processes we can leverage the power of clusters and other high performance computing resources to parallelize the loops into thousands, and potentially 10 s of thousands, of concurrent iterations, each taking on a fraction of the required computations and, hence, running in less time. By distributing the total number of persons and places among multiple processes, each loop on each process would have a concomitantly smaller number of persons and places to iterate through, hopefully resulting in shorter run times.

chiSIM, like its CA-MRSA model precursor, is written using Repast HPC [10]. Although the original CA-MRSA model and the initial OpenMP version of chiSIM run as a single process, the ABM components provided by Repast HPC, such as the event scheduler and data collection, are designed to work in a multi-process non-shared memory environments using MPI [13]. Consequently, very little of the ABM-specific infrastructural parts of the model (e.g. event scheduling) had to be changed. The majority of the work focused on refactoring the model itself to work in the multi-process non-shared memory environment found in clusters and high performance computers. In the multi-process parallel environment, each process is responsible for agents (in this case, persons and places) *local* to that process. That process executes the

code that represents the agents' behavior. Data (e.g. agents) can be shared across processes using MPI.

In parallelizing the model over multiple processes, the core concept remains the same: people move between places and contact occurs in those places. However, given that we are distributing people and places among different processes, a person's next place may not share the same process as the person. Disease transmission is dependent on the characteristics of the place in which persons are co-located, that is, on the risk associated with the place, and with the disease state of the persons in that place. Consequently, if disease transmission is to occur, then either (1) persons need to move to the process on which their next place exists, and the epidemiological consequences of the contact in that place can then be computed in that place; or (2) places need to be shared across processes such that the epidemiological consequences can be computed for any person on any process.

In refactoring the model, we chose the first option. Places are created on a process and remain there. Persons move among the processes according to their activity profiles. Implementation-wise this option was simpler than the second option, in that it was similar to the CA-MRSA model and only additional code to coordinate the movement of persons among processes was required. We also suspected and hoped that it would be possible to cluster persons around groups of places in such a way as to minimize the cross-process movement of the persons. In this chosen scheme, when a person selects its next place to move to, it may stay on its current process or it may have to move to another process if the place is not on the person's current process. The second case is the one that we hoped to minimize. The updated pseudo code for the distributed chiSIM time step incorporating cross-process person movement can be seen in Fig. 3.

```
for each person:
  place = select the place for the next hour's activity
  if place's process rank[1] == person's process rank:
    move to the place
  else:
    add person to list of persons to move place's rank

move persons to other processes
receive persons from other processes

for each person received from other processes:
  move to the current place

for each place:
  compute the epidemiological consequences of co-location
```

Fig. 3. Pseudo-code for the chiSIM time step including cross-process movement (A process rank is an integer id used to identify a process.).

In distributing the model is this way, the number of persons and places in each process is smaller and thus the number of loop iterations is smaller as well. However, cross-process movement is expensive and, in fact, a previous naïve attempt at parallelizing the CA-MRSA model ran slower than the single process serial version due to the communication cost (i.e. the time spent) of sending and receiving the persons. This inter-process communication cost has two factors: the amount of data sent and the frequency with which it is sent. To minimizing the cross-process communication we needed to send less data, and do so less frequently.

3.2.1 Minimizing the Data Cost

chiSIM is a C++ model and persons and places are implemented as C++ classes. Sending a person object from one process to another consists of sending the current state of the person object, that is, its current field values, via MPI and creating a new person object on the other end with these sent field values. When a person is moved between processes, it needs enough of its state available on the target process to execute its place selection and any disease related behavior. That required information consists of the activity profile used to select the next place and the activity risk, enough place data to select the next place from the places on the current process, and the person's current disease state.

Much of this can be cached, given sufficient memory of which we had plenty (16 to 32 GB of RAM per compute node). Rather than moving a person's entire activity profile between processes when it moves, all the shared activity profiles are read and constructed on each process during initialization. These are placed in a standard library map with a unique activity profile id as the key. When a person is moved between processes only this activity profile id needs to be transferred rather than the entire activity profile. A similar strategy was used for the required place data. All the processes read all the place data. Each place has a unique id and a process rank id where the rank id identifies the process on which the place resides. Those places that will reside on the reading process are constructed as fully functioning place objects able to compute the epidemiological consequences of contact. Those places that are not local to the reading process are constructed as only the simple *id, rank* pair. When a person is transferred between processes only the *id, rank* pair is transferred.

A person's activity profile structure does not change, although it can be overridden, and the id and rank pairs of the places used in that profile do not change. Given that the persons are essentially constituted by these data, they can also be cached. When a person arrives at a target process for the first time, the transferred data (activity profile ids, etc.) is re-assembled into a new person object. That object is then cached in a map using a unique person id as the key. The next time the same person is moved to that target process the pre-assembled person is retrieved from the map and updated with any dynamic data (i.e. the disease state) associated with the person. This pre-assembled person is then used in the person loop on the target process. This pre-assembly also avoids any overhead associated with the frequent creation of new C++ objects and the concomitant memory allocation. It also allows us to potentially eliminate the transfer of any data that can be cached within the pre-assembled person. The gist, PersonCachingExample.cpp, in [2] illustrates how persons are cached when first received from another process and subsequently retrieved from the cache.

3.2.2 Minimizing the Sending Frequency

As described earlier, persons move between processes when the place for their next activity is on a different process from the one on which they currently reside. If all of a person's possible places are on the same process, that person will never have to move between processes. To the extent that we can assign places to processes in a way that maximizes this within process movement, the frequency of cross process movement will be minimized. Fewer persons will need to be sent to other processes as part of moving to their next place.

This problem essentially maps to a *graph partitioning problem*. In this graph, each place is a vertex and each edge represents person movement between two places, that is, between two vertices. The edge weight represents the number of persons that travel between the two places. The goal, then, is to partition the graph into groups of vertices (places), such that the edge *weights* connecting these groups are minimized. That is, by putting places that are highly connected to each other on the same process, we can minimize cross-process movement.

To perform the partitioning, we created the places graph. This involved creating a vertex for each place and then, following each person's activity profile, an edge between two activities when the prior activity's place differed from the subsequent activity's place. If an edge already existed, the edge's weight count was incremented by 1. For the full Chicago scenario, the graph consisted of approximately 1.2 million vertices (places) and approximately 1.9 million edges (trips between places). To partition the graph, we used the METIS toolkit [14, 15]. METIS is a set of applications for partitioning graphs and finite element meshes. It is used in a variety of domains such as linear programming and load balancing, the latter making it a good fit for balancing our places among the distributed processes. Having partitioned the existing place graph with METIS, the places data were then updated by adding a rank attribute for each place that corresponded to the graph group as determined by METIS.

In order to establish a baseline against which the METIS partitioned places could be compared, we also created a random partitioning. The random partitioning assigned ranks to places based on their location in the input file. Given a total number of processes n, and a total number of places p, the first p/n number of places in the places file were assigned to process 0, the next p/n number of places in the file were assigned to process 1, and so on. If p is not a divisor of n, any remaining processes were assigned in round robin fashion to the n processes. When the METIS partitioning was initially benchmarked against this random partitioning, it was in fact slower. A simulated week run on 128 processes on Argonne National Laboratory's Cetus BG/Q high performance computer took 43 s for the METIS partitioning versus 41 s for the random partition. We suspected that the good performance of the so-called random partitioning had revealed a pattern in the input data that resulted in improved performance, but we were more interested in why the METIS partitioning was slower.

To investigate this further we used the TAU [16, 17] profiler, and the instrumentation API in particular. Profiling in a parallel and distributed environment can be challenging. Per-process, inter-process, and application level performance data needs to be gathered in an efficient manner, often necessitating low-level access to the underlying hardware. Thus, mature cross-platform profiling tools such as TAU are invaluable when investigating parallel distributed application performance. We used TAU to

record the time spent in the person loop, the time spent in the sending and receiving persons between processes, and the amount of time to re-assemble the moved persons data into Person objects. (An example of using the TAU API can be seen in the gist, TAUProfiling.cpp, in [2].) Of these timing data sets, the first and last proved to be informative. Repast HPC, and thus chiSIM, use a conservative schedule synchronization approach where all processes execute a time step in parallel and then *wait for all processes to finish* before executing the next time step. As a consequence, the overall execution time is only as fast as the slowest process. Other processes may execute more quickly but will have to wait for the slowest process to finish. Therefore speeding up the slowest process will improve the overall run time. TAU provides per process timing data for each profiled section. The timings for the worst performing processes of a test run[1] are presented in Table 1.

Table 1. Profiling timing values for the worst performing processes using the random and METIS partitionings.

Partition type	Time in loop	Time for re-assembly
Random	28.99 s	6.95 s
METIS	32.75 s	6.09 s

We can see in Table 1 that the slowest process in the METIS partitioned case takes roughly 4 s longer than the random partitioned case to iterate over all the persons in the person loop. However, the METIS partitioned case is nearly a second faster in re-assembling cross-process person data into Persons. The latter is a good measure of how much data is being passed between processes and thus indicates that the partitioning actually works, that is, there is less inter-process communication in the METIS partitioned case. The loop timing, however, is dependent on the number of persons that have to be iterated through and indicates that there is perhaps a larger number of persons on the slowest process in the METIS partitioned case. Subsequent runs that recorded the number of persons per process at each time step and confirmed that the slowest METIS partitioned process contained more persons than the slowest random partitioned process.

The problem was that any potential speed increase gained by moving persons less frequently was being lost by unbalanced numbers of persons across processes. The METIS partitioning, while minimizing the cross-process communication, had inadvertently placed too many persons on some processes by assigning too many heavily populated places to those processes. The solution to this issue was to find a way to balance the number of persons on each process during the partitioning such that each process retained a roughly equal number of persons, while still attempting to minimize the movement of persons between processes. Fortunately, METIS can be used to achieve such a multi-objective partition by providing it with weights for the vertices in the network. We re-created the place graph but this time assigned a weight to each vertex that represented the total number of persons that visited that place. METIS was

[1] The test runs showed very little variation in run time between runs, with the profiling results staying constant throughout.

then rerun using both the edge and vertex weights. The model run time with this new partitioning was now faster than the baseline random partitioning. Run times for all three cases for a simulated week on 128 processes can be seen in Table 2.

Table 2. Profiling timing values for the worst performing processes using the random, METIS and METIS with vertex weights partitionings.

Partition type	Total run times
Random	41 s
METIS	43 s
METIS w/Vertex Weights	33 s

Comparing the original CA-MRSA model, which for a simulated 10 year run running on a single thread on a single process took approximately 60 h to complete, the parallelized and multi-threaded chiSIM, running over 128 processes and using 16 threads per process, took approximately 4 h. Thus, by applying multi-threading and parallelizing the model across processes, we achieved a 1350 % speed up.

4 Conclusion

If ABMs are to serve as validated and trusted electronic laboratories, they must undergo a set of experimental analyses, including adaptive parametric studies, sensitivity analyses, optimization and uncertainty quantification. For these computational experiments, which involve the execution of large numbers of simulation runs, to complete in reasonable amounts of time, the run time performance of each individual simulation needs to be improved. Here we have presented a case study for refactoring a useful, albeit slow running, model into a scalable, parallelized and distributed model whose improved run time performance opens the door for such studies. We began by parallelizing the model via multi-threading using the OpenMP toolkit and subsequently enabled it to be distributed across multiple processes over MPI. In doing the latter, the cost of cross-process communication was mitigated by moving less data and moving data less frequently. To move less data, we took advantage of a memory-rich environment by caching the required person data on each process and only moving the minimum amount necessary. To move data less frequently, we optimized the assignment of places on processes to minimize the amount of person movement between them. The realization that the movement of the persons between places could be mapped to a graph partitioning problem proved the key to the place assignment, and allowed the model to efficiently scale up to larger populations. Throughout the refactoring, we continually profiled, grounding our intuitions in hard data.

Acknowledgements. This work is supported by the U.S. Department of Energy under contract number DE-AC02-06CH11357 and National Science Foundation (NSF) RAPID Award DEB-1516428.This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.The Chicago MRSA ABM was developed with support from the MIDAS Program, NIH/National Institute of General Medical Sciences, grant U01GM087729.

References

1. Thiele, J.C., Kurth, W., Grimm, V.: Facilitating parameter estimation and sensitivity analysis of agent-based models: a cookbook using NetLogo and R. J. Artif. Soc. Soc. Simul. **17**(3), 11 (2014)
2. https://gist.github.com/ncollier/8030c144c902cf803584
3. Macal, C.M., North, M.J., Collier, N.T., Dukic, V.M., Wegener, D.T., David, M.Z., Daum, R.S., Schumm, P., Evans, J.A., Wilder, J.R., Miller, L.G., Eells, S.J., Lauderdale, D.S.: Modeling the transmission of community-associated methicillin-resistant *Staphylococcus aureus*: a dynamic agent-based simulation. J. Transl. Med. **12**, 124 (2014)
4. Chambers, H.F., Deleo, F.R.: Waves of resistance: Staphylococcus aureus in the antibiotic era. Nat. Rev. Microbiol. **7**(9), 629–641 (2009)
5. Dukic, V.M., Lauderdale, D.S., Wilder, J., Daum, R.S., David, M.Z.: Epidemics of community-associated methicillin-resistant *Staphylococcus aureus* in the United States: a meta-analysis. PLoS One. **8**(1), E52722 (2013)
6. Moran, G.J., Krishnadasan, A., Gorwitz, R.J., Fosheim, G.E., McDougal, L.K., Carey, R.B., Talan, D.A.: Methicillin-resistant *S. aureus* infections among patients in the emergency department. N. Engl. J. Med. **355**(7), 666–674 (2006)
7. Klevens, R.M., Morrison, M.A., Nadle, J., Petit, S., Gershman, K., Ray, S., Harrison, L.H., Lynfield, R., Dumyati, G., Townes, J.M., Craig, A.S., Zell, E.R., Fosheim, G.E., McDougal, L.K., Carey, R.B., Fridkin, S.K.: Invasive methicillin-resistant *Staphylococcus aureus* infections in the United States. JAMA **298**(15), 1763–1771 (2007)
8. Wheaton, W.D., Cajka, J.C., Chasteen, B.M., Wagener, D.K., Cooley, P.C., Ganapathi, L., Roberts, D.J., Allpress, J.L.: Synthesized Population Databases: A US Geospatial Database for Agent-Based Models. RTI Press, Durham (2009). Publication No MR-0010-0905
9. Ozik, J., Collier, N.T., Murphy, J.T., North, M.J.: The ReLogo agent-based modeling language. In: Proceedings Of The WSC 2013, Washington, D.C. (2013
10. Collier, N., North, M.: Parallel agent-based simulation with repast for high performance computing. Simulation **89**(10), 1215–1235 (2012)
11. Instruments User Guide. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html. Accessed 1 Jun 2015
12. OpenMP specification home page. http://openmp.org. Accessed 1 Jun 2015
13. MPI standard official website. http://www.mcs.anl.gov/research/projects/mpi/index.htm. Accessed 1 Jun 2015
14. Karypis, G., Kumar, V.: A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1999)
15. METIS home page. http://glaros.dtc.umn.edu/gkhome/views/metis. Accessed 1 Jun 2015
16. Shende, S., Malony, A.D.: TAU: the TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
17. TAU home page, https://www.cs.uoregon.edu/research/tau/home.php. Accessed 1 Jun 2015