Communication papers of the Federated Conference on Computer Science and Information Systems, pp. 65–73 DOI: 10.15439/2017F172 ISSN 2300-5963 ACSIS, Vol. 13

Collision-Free Agent Migration in Spatial Simulation

Christopher Bowzer Benjamin Phan Kasey Cohen Munehiro Fukuda
Computing and Software Systems
University of Washington Bothell
18115 NE Campus Way, Bothell, WA 98011
Email: {cjbowzer, firstbbp, kcparker, mfukuda}@uw.edu

Abstract— **Parallelization of agent-based models (ABMs) is one solution for scaling up their simulation size sufficiently covering more realistic problems. In order to break through memory limitation, some ABM simulators such as RepastHPC and FLAME enabled parallel simulation over a cluster system, (i.e. distributed memory). They visualize to agents remote processors' boundary data as ghost space or facilitate message broadcast among agents, so that agents can still share a full or partial view of their simulation space. Yet, ABMs encounter a parallelization problem where multiple agents may migrate to and thus collide with each other on the same logical coordinates, which should not occur in some applications, (e.g., traffic simulation where two vehicles cannot change to the same lane). Although such collision problems have been addressed algorithmically at a user level where an agent stops before or hops over another agent, moves faster or slower, ticks over time, or cuts coordinates finer, they yet require inter-agent synchronization such as serializing agent migration over all collision-inducing sub-spaces or cells, using a single thread. To facilitate collision-free agent migration more efficiently, we considered two migration algorithms named location-ordered and direction-ordered migration, and implemented them over three ABM simulators: Multi Agent Spatial Simulation (MASS), RepastHPC, and FLAME. This paper discusses about programmability and execution performance among these three simulators in collision-free agent migration.**

I. Introduction

SCALABILITY of simulation size is quite important for agent-based models (ABMs) including transport simulation [1], neural network simulation [2], ecological simulation [3], and immune system simulation [4], all requiring millions of agents to predict practical phenomena. Obviously, one solution is parallelization of underlying simulators or applications themselves. However, the biggest challenge is that most ABM applications have been based on a shared-memory paradigm where agents interact with each other on a global simulation space. Although multithreading or even GPU computing has been applied to ABMs such as MATSim [1] and TB simulation [5] to reserve their share-memory-based implementations, some simulators such as RepastHPC [6], FLAME [7], and FluTE [3] enabled parallel simulation over a cluster system, (i.e., distributed memory) in support with MPI. Their implementation facilitates message broadcast among agents or visualizes to agents remote processors' boundary data as ghost space, so that agents can still share a full or partial view of their simulation space.

Yet, departed from a shared-memory paradigm, parallelized ABMs encounter another problem where multiple agents may migrate to and thus collide with each other on the same logical coordinates, which should not occur in some applications, (e.g. transport simulation where two vehicles cannot change to the same lane). In pedestrian simulation [8], each pedestrian agent avoids a collision by calculating its repulsive force with others, which results in $O(N^2)$ complexity. In traffic simulation [9], collision-inducing subspaces are serialized by a single thread, which requires careful space partitioning. Other simulators [10], [11] leave this collision problem on user-level solutions where an agent stops before or hops over another agent, moves faster or slower, ticks over time, or cuts coordinates. These algorithms and implementations burden model designers with more programming complexity.

We particularly focus on agent migration[1] over cellular-based or logical network space in transport simulation [12], [9] and artificial life [13], [14]. For this type of simulation, we consider three different algorithms that enforce collision-free agent migration, each named trial-and-error, location-ordered, and direction-ordered migration. Trial and error considers all such agent actions including a stop, a hop, or a faster/slower move upon encountering a collision. On the other hand, the location and direction-ordered algorithms move only agents at a time, which are located on the same group of coordinates, (e.g. those with their coordinates[i,j] where i and j are divisible by three) or migrate to the same direction, (e.g. moving to the north). We implemented the location and direction-oriented algorithms over our own ABM simulator named the MASS (Multi-Agent Spacial Simulation) library [15] as well as RepastHPC and FLAME as our benchmark platforms, and compared their programmability and execution performance.

The contribution of this paper is two-fold: (1) comparing three approaches to collision-free agent migration in parallel simulation and (2) demonstrating the programmability and performance superiority of the MASS library over RepastHPC and FLAME in collision-free migration. The rest of the paper is organized as follows: Section II surveys the conventional user-level and system-level collision-free migrations, and compares them with our location/direction-ordered algorithms in

[1]As mobile agents frequently use "migration", we use this terminology rather than "move".

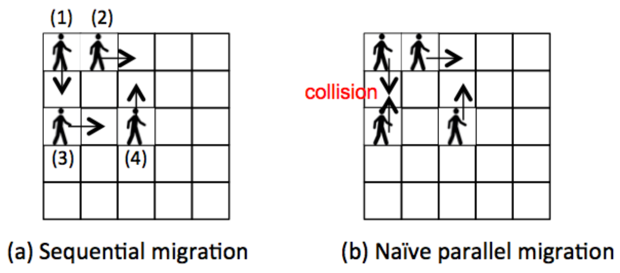

Fig. 1. Agent migration over a meshed space

parallel simulation; Section III analyzes implementations of collision-free agent migration, each on MASS, RepastHPC, and FLAME; Section IV compares execution performance of these three simulation systems; and Section V summarizes the MASS-based collision-free agent migration.

II. Agent Migration Algorithms

Some ABM applications such as TRANSIMS [12], AIMSUN2 [9], Wa-Tor [14], and Sugarscape [13] simulate agent migration over a cellular-based or logical network space where at most one agent should reside at a given cell, node, or link, which however needs special cares in parallel simulation. This section gets started with a sequential execution of such agent migration over a space, thereafter raises an agent-collision problem in parallel execution, and examines three solutions to address this problem.

A. Agent Migration in Spatial Simulation

For simplicity, let us consider agent migration over a two-dimensional (or 2D) cellular space as shown in Figure 1-(a). Sequential execution scans the space from upper left to lower right as moving an agent at each cell at a time. This gives a higher priority to an agent residing on an upper-left cell so that another at a lower-right cell can safely find its next destination cell without considering any potential collisions.

However, parallel execution removes such agent priorities that all agents in a space are allowed to migrate to their next destinations in any orders (see Figure 1-(b)), which causes an agent collision on a cell. Therefore, we need to consider parallel implementations to facilitate collision-free agent migration.

B. Related Work

Several collision avoidance algorithms have been proposed in both conceptual models and underlying parallel implementations. In conceptual models, Kirchner et al. [10] gave four resolutions for conflicts of pedestrians that move over a cellular automata model: (1) having an agent hop over or stop before another agent, (2) allowing an agent to move as far as possible, (3) dividing simulation time into sub-time steps, and (4) keeping a pedestrian from crossing the trajectory of another pedestrian that has already moved. Bandini et al. [11] gave another set of four methods to avoid agent collisions in pedestrian simulation: (1) changing their walking speeds, (2) modifying the space discretization towards a finer grain, (3) modifying the current time scale, and (4) combining methods 2 and 3. However, these conceptual solutions still need careful implementation techniques in parallel simulation. If a simulation space is partitioned and mapped over a distributed-memory system, the simulator must facilitate so-called *ghost space* that visualizes a remote computing node's boundary data to the local node, so that each agent can observe others even on remote nodes. Furthermore, an agent must move into its destination cell exclusively no matter how finely simulation time and space are sliced. Otherwise "trial and error" will be repeated where multiple agents may end up moving to the same cell, in which case all except one that can stay there must follow one of the above resolutions to change their destination.

In parallel implementations, AIMSUN2 [9], a cellular-based transport simulator serializes agent migration over collision-induing adjacent cells with a single thread, which handles these cells as a critical section and guarantees exclusive agent migration. However, users are burdened with grouping such collision-inducing cells into a non-interruptible block. In pedestrian simulation, Wagoum et al. [8] avoided a conflict of pedestrian agents by calculating each agent's repulsive force with others. Their simulator was parallelized with MPI where each rank maintains a neighborhood list of agents and computes their repulsive forces in parallel. However, the simulation results in $O(N^2)$ complexity yet within each MPI rank.

Our goal is to mitigate these user burdens incurred by conceptual models and to reduce complexity of parallel implementations and their computation.

C. Parallel Algorithms

We consider the following three collision-free migration algorithms: (1) trial-and-error migration, (2) location-ordered migration, and (3) direction-ordered migration.

1) Trial-and-Error Collision-Free Migration: As illustrated in Figure 2, this algorithm allows agents to collide with each other on the same destination cell. However, upon a collision, only one agent (with the highest identifier in most cases) can keep residing there while all the others must back off to their source cells and thereafter choose another available cell to move. This trial-and-error migration needs to be repeated until all agents find their next destination or conclude no more cells to go. It belongs to a so-called distributed-termination detection problem. If simulation wants to avoid solving this problem, it will end up examining every single direction of four or eight destinations in the von Neumann neighborhood, (i.e., north, east, south, and west) or the Moore neighborhood, (i.e., the former four directions plus north east, south east, south west, and north west). Unless parallel implementations guarantee serialization of agent migration as seen in AIMSUN2 [9], collision avoidance in conceptual models may result in trial and error.

2) Location-Ordered Collision-Free Migration: Let us consider a 2D space tiled with 3-by-3 cells, each numbered 0 through to 8. As shown in Figure 3, if we pick up only

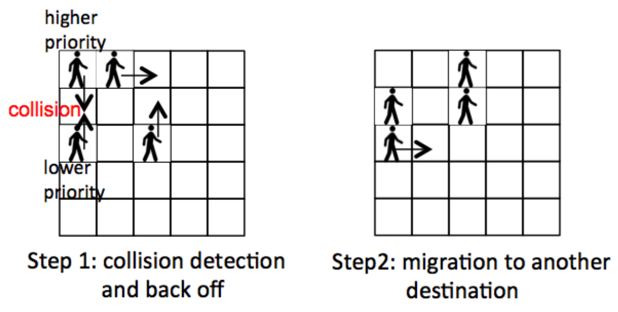
higher priority
collision
lower priority
Step 1: collision detection and back off
Step2: migration to another destination

Fig. 2. Trial-and-error collision-free migration

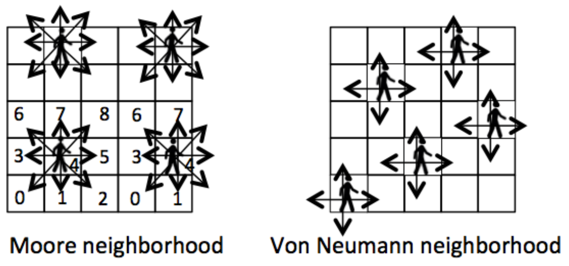

Fig. 3. Location-ordered collision-free migration

agents residing on cells with the same number, (e.g., those with #4), their migration based on the Moore neighborhood will cause no collision. Upon moving these agents, each cell must make its agent occupancy readable to its neighbors (as ghost space over distributed memory), so that the other agents will no longer choose the occupied cells. An iteration of agent migration and cell communication needs to be repeated nine times. The von Neumann neighborhood can reduce this repetition to five times. We call this algorithm location-ordered collision-free migration in the following discussions.

3) Direction-Ordered Collision-Free Migration: The third collision-free migration is direction-ordered. As shown in Figure 4, we will choose only agents that migrate toward the same direction. This ordered migration must be repeated eight times in the Moore and four times in the von Neumann neighborhood, (e.g., north first, east second, south third, and finally west in Figure 4). Note that, similar to the location-ordered migration, all cells must inform their neighbors of their agent occupancy each turn of agent migration.

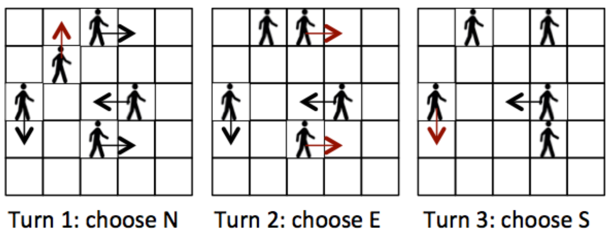

Fig. 4. Direction-ordered collision-free migration

D. Pros and Cons

Among the three collision-free algorithms, trial-and-error migration is potential to move agents in the least iterations, (e.g., ≤ 4 in the von Neumann neighborhood). It prioritizes agents with a higher ID, which allows an agent with the highest ID to stay at a new destination and thus can reproduce a deterministic migration over multiple simulation runs. On the other hand, trial-and-error migration needs to have agents back off to their source cell upon a collision as well as to implement a distributed voting algorithm among all processes to see if they have to conduct the next trial. The vote needs three phases of collective communication: (1) a vote initiated from the master to all the slave processes, (2) ballots collected to the master, and (3) the result broadcast from the master. Although these three phases can be described in one *MPI_Allreduce()* function, the complexity of the underlying implementation doesn't change. Therefore, trial-and-error migration costs $p \times (5c + a)$ where $p = \#$trials, $5c =$ agent sending, agent retracting, and three phases of communication needed for a distributed vote, and $a =$ agent synchronization overheads. We may avoid these distributed votes and therefore repeat four trials as default, in which case the cost will be $4(2c + a)$.

Figure 5 shows how quickly the number of collisions grows and how many trial iterations are required as increasing the number of agents randomly distributed and migrating over a 1000×1000 simulation space. Even 10,000 agents, (i.e., only 1% population density) results in 103 collisions in average. This in turn means that trial-and-error migration in most cases can't complete in the first trial phase. With 50,000 agents, (i.e., 5% population density), the algorithm needs three trial phases. Finally it repeats four phases beyond 200,000 agents, (i.e., 20% population density). Considering practical applications that populate agents non-uniformly and move them toward the same direction, (e.g., toward an exit door in an evacuation), we may estimate $3(5c + a)$ as the cost of trials and errors or end up $4(2c + a)$ with four default trials.

The location-ordered migration is the easiest to implement at an application level by scheduling multiple events, (e.g., 9 in the Moore neighborhood), each moving agents that reside on cells with the same number: 0 through to 8. However, unless an application regroups all agents into these nine events before migration, it must inevitably scan all agents at each event for the purpose of identifying which agent resides on which cell. Therefore, the location-ordered algorithm is the most expensive from the viewpoints of the number of iterations in migration and agent-scanning cost per iteration: $9(2c + s)$ where $2c =$ghost-space updating and agent sending overheads and $s =$ memory access overheads for scanning agents.

The direction-ordered migration can complete less iterations of migration, (e.g., four in the von Neumann) than the location-ordered algorithm although it has the same agent regrouping or scanning problem as the location-ordered migration. Therefore, the algorithm costs $4(2c + s)$ where $2c$ and s are the same parameters as location-order migration. If this algorithm should be implemented at a system level, the system needs to

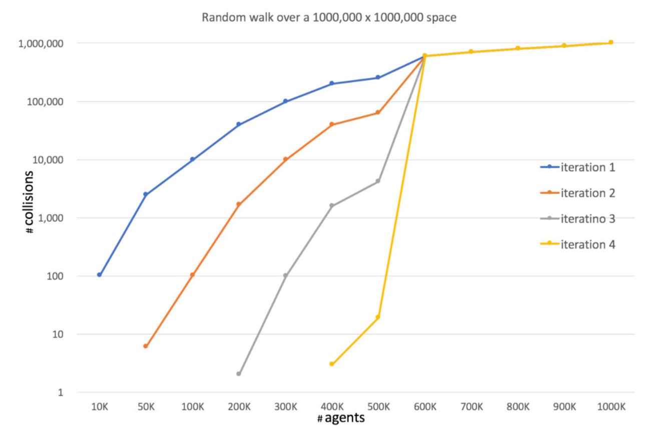

Fig. 5. #Collisions in each trial phase when increasing agent population over a 1000 × 1000 space

directly examine each agent's data members to check its next destination.

In summary, the trial-and-error migration apparently burdens model designers with implementing inter-agent synchronization as well as a distributed voting algorithm, and it has little performance superiority as discussed above. Therefore, we will focus on location- and direction-ordered migration algorithms in our programmability analysis of three agent-based systems: our MASS library, RepastHPC, and FLAME.

III. Implementations of Parallel Migration

This section compares MASS, RepastHPC, and FLAME systems in coding agents' random walk over a 2D space, (named RandomWalk). Our analysis covers location-ordered collision-free migration at an application level as well as direction-ordered migration at a system level.

A. MASS

We have developed the MASS (Multi-Agent Spatial Simulation) library in Java and C++. *Places* and *Agents* are keys to the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element is called a *place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other *places*. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *places* with array indices (thus as duplicating themselves), and indirectly interact with other *agents* through variables local to the current *place*.

Parallelization with the MASS library uses a set of multi-threaded communicating processes that are forked over a cluster of multi-core computing nodes with JSCH in Java or libssh2 in C++ and are connected to each other through TCP sockets. Multi-threads take charge of method call and information exchange among *places* and *agents* in parallel. A user designs a behavior of a *place* and an *agent* by extending the *Place* and *Agent* base classes respectively. They are populated through the *Places* and *Agents* classes. Actual computation is performed between *MASS.init()* and *MASS.finish()*, using the following major C++ methods, each performed in parallel [15].

Places Class

- ***public Places(int handle, String className, int boundary, void *argument, int argSize int dim, int size[])*** instantiates a shared array with *size* and a ghost space with *boundary* from *className* as passing an *argument* to the *className* constructor.
- ***public void* callAll(int functionId, void *arguments, int argSize)*** calls the method specified with *functionId* of all elements as passing *arguments[i]* to element[i], and receives a return value into *(void *)[i]*.
- ***public void exchangeAll(int handle, int functionId, vector<int*> *destinations)*** calls from each element to a given method of all destination elements, each indexed with a *vector* element, and exchanges data among the elements.
- ***public void exchangeBoundary( )*** exchanges boundary data with neighboring cluster nodes as a ghost space.

Place Class

- ***private vector<int> size, index*** maintain the size of the shared array that each element belongs to and the index of each array element.
- ***public void* callMethod(int functionId, void *arguments)*** is invoked from *Places.callAll()* or *exchange-All()* so as to call a function specified with *functionId*.

Agents Class

- ***public Agents(int handle, String className, void *arguments, int argSize, Places *places, int population)*** instantiates agents from *className*, passes *arguments* to their constructor, and populates them over a given *Places*, based on *Agent.map()*.
- ***public void* callAll(int functionId, void *arguments, int argSize, int retSize)*** is the same as *Places.callAll()*.
- ***public void manageAll()*** updates each agent's status, based on its latest calls of *migrate()*, *spawn()*, and *kill()*. These methods are invoked within *callAll()*.

Agent Class

- ***migrate(int[] index...)*** allows a calling *Agent* to migrate or propagate itself to one or more *Places* specified with *index* upon *Agents.manageAll()*.
- ***spawn(int nChildren, Object arguments)*** spawns children as passing *arguments* to them.
- ***kill()*** terminates a calling *Agent*.
- ***public void* callMethod(int functionId, Object argument)*** is the same as *Place.callMethod()*.

Figure 6 shows abstract C++ code for parallelizing RandomWalk with MASS. We use location-ordered agent migration, focusing on the von Neumann neighborhood, (i.e., including four N, E, S, and W neighbors) while numbering places from 0 to 8 and scheduling nine turns of migration as in the Moore neighborhood for simplicity. The *main()* function (lines 26-38) serves as a simulation scenario. It creates an $sz \times sz$ virtual *land* over which the n number of *nomad* agents

```
1  // Place logic
2  void *Land::checkOccupancy(void* arg) {   //each place
3    outMessage=new int((int)agents.size());//counts #agents
4    return NULL;
5  }
6  // Agent logic
7  void *Nomad::randomWalk(int *turn) {
8    int myTurn = index[0] % 3 + 3 * (index[1] % 3);//get 0-8
9    if (myTurn != *turn) return NULL;
10   vector<int> destination;        //choose next destination
11   random = rand() % 4;
12   for (int i = 0; i < 4; i++) {//check N,E,S,W vacancies
13     int dirX = (random%2==0) ? 0 : ((random==1) ? 1:-1);
14     int dirY = (random%2==1) ? 0 : ((random==0) ? 1:-1);
15     destination.push_back( index[0] + dirX );
16     destination.push_back( index[1] + dirY );
17     if ((int *)(place->getOutMessage(destination)) > 0)
18       random = ( random + 1 ) % 4;//occupied, check next
19     else {
20       migrate( destination );       //vacant, migrate there
21       break;
22   } }
23   return NULL;
24 }
25 // simulation scenario
26 int main(int argc, char** argv){
27   MASS::init(argv, nProc, NThr);//populate agents on land
28   Places *land=new Places(1,"Land", 1, NULL,0, 2,sz,sz);
29   Agents *nomad=new Agents(2, "Nomad", NULL,0, land, n);
30   for (int time=0; time<maxTime; time++){//cycle simulation
31     for (int turn=0; turn<9; turn++) {//9 migrations/time
32       land->callAll(Land::checkOccupancy_);
33       land->exchangeBoundary(); //Places exchange #agents.
34       nomad->callAll(Nomad::randomWalk, &turn,
35                                                sizeof(int));
36       nomad->manageAll();         //actual migration
37   } }
38   MASS::finish()
39 }
```

Fig. 6. Application-level agent migration in MASS

are populated (lines 28-29). Thereafter, a cycle simulation gets started with $time = 0$ through to $maxTime - 1$ (line 30). For each cycle, we schedule nine turns of agent migration (line 31). For each turn, we need to update each place's ghost space (stored in *outMessage*) (lines 32-33), for whose purpose each *land* updates its agent population in its *outMessage* (lines 2-5). Then, each agent invokes its *randomWalk()* function (lines 34-35 and thereafter 7) that checks if it is this agent's turn (lines 8-9), randomly chooses one of its four neighboring places (lines 10-16), checks its occupancy (line 17), and decides to migrate there or to choose another place (line 18 or 20). The actual agent migration is committed at once by *manageAll()* (line 36).

Needless to say, model designers do not want to be aware of agent collisions that are not the essence of their simulation and thus should be supported by a system. Therefore, we have implemented direction-ordered collision-free migration in MASS. The implementation used the 2D von Neumann neighborhood only. As shown in Figure 7, RandomWalk can remove all the code related to collision avoidance and therefore focus on agents' *randomWalk()* function.

B. RepastHPC

RepastHPC is an MPI-supported parallel simulation platform for agent-based modeling, which was developed by the Argonne National Laboratory. In RepastHPC, *Context* is an

```
1  // Agent logic
2  void *Nomad::randomWalk(int *turn) {
3    vector<int> destination; //choose next destination
4    random = rand() % 4;
5    int dirX = (random%2==0) ? 0 : ((random==1) ? 1:-1);
6    int dirY = (random%2==1) ? 0 : ((random==0) ? 1:-1);
7    destination.push_back( index[0] + dirX );
8    destination.push_back( index[1] + dirY );
9    migrate( destination );  //collision-free migration
10   return NULL;
11 }
12 // simulation scenario
13 int main(int argc, char** argv){
14   MASS::init(argv, nProc, NThr);//populate agents on land
15   Places *land=new Places(1,"Land", 1, NULL,0, 2,sz,sz);
16   Agents *nomad=new Agents(2, "Nomad", NULL,0, land, n);
17   for (int time=0; time<maxTime; time++){//cycle simulation
18     nomad->callAll(Nomad::randomWalk,&turn,sizeof(int));
19     nomad->manageAll();          //actual migration
20   }
21   MASS::finish()
22 }
```

Fig. 7. System-supported agent migration in MASS

execution environment that populates agents over a given *Projection* instance such as a shared network, gird, and space. We can regard RepastHPC's contexts, agents, and projections as MASS processes, agents, and places respectively. RepastHPC has the following similarities to MASS:

1) ***Ghost space*** views adjacent MPI ranks' simulation boundary so that agents in each rank can see their neighborhoods' information including sub-space occupancy by other agents.
2) ***Agent migration*** are all committed at once by *projection.balance()* (similar to MASS *agents.manageAll()*) and are carried out physically by moving agents to a remote rank.

On the other hand, RepastHPC is different from MASS in the following aspects:

1) ***Simulation events*** are scheduled in a context separately from *main()*, whereas MASS schedules events as for-loop iterations in *main()*.
2) ***agent collisions*** are not supported at the system level unlike MASS.
3) ***spatial operations*** are passively invoked from a context or each agent, whereas MASS allows each place to manipulate agents through its *agents* vector.

In a similar way to MASS, we parallelized RandomWalk with RepastHPC. Figure 8 shows the abstract code in C++. The *main()* function initializes RepastHPC (line 35), instantiates a context (line 36), populates agents (line 38), schedules nine turns or events of agent migration (line 39), and starts a cycle simulation (line 40). In RepastHPC, *Context* schedules actual events and controls agents. Its *schedule()* function (lines 2-6) needs to declared these nine migration events separately, each calling the *move()* function (lines 7-16). This actually means that, upon each invocation, *move()* must scan all agents from the top in the context (lines 8-9) for examining each agent's current location (lines 10-11). It invokes *Nomad::randomWalk()* if a given agent resides on a cell with the current turn. The *randomWalk()* logic is the same as that of MASS: repeating a selection of the next destination up to four times until finding a vacant neighbor.

As compared to MASS, RepastHPC facilitates to its users more generic and various types of simulation space and gives them more freedom of agent management, which however tends to burden model designers with a much steeper learning curve and to increase their code size.

```
// Context logic
void MyContext::schedule(repast::ScheduleRunner& r){
 r.scheduleEvent(1.1,Schedule::FunctorPtr(this, move));
    ...;//schedule 9 migration events, each for cell #0-8
 r.scheduleEvent(1.9,Schedule::FunctorPtr(this, move));
}
void MyContext::move(){//scan all agents and move only those
  it = agents.begin(); //residing on cells with the same #
  while (it != agents.end()) {
     discreteSpace->getLocation( (*it)->getId(), loc );
     if (partitionCounter % 9 == loc)  //pick up this agent
      (*it)->randomWalk(discreteSpace);//let it plan on move
    i++;
  }
  discreteSpace->balance();//actual migration at once
}
// Agent logic
void Nomad::randomWalk(repast::SharedDiscreteSpace* sp){
  for (int i=0; i<4; i++) dst.push_back(i);
  while (!moveset.empty()) {//check N,E,S,W neighbors.
    int r=rand() % dst.size();//randomly choose one
    newLoc.push_back(loc[0]+cardinals[dst[r]][0]);
    newLoc.push_back(loc[1]+cardinals[dst[r]][1]);
    repast::Point<int> center(newLoc);//get space info
    repast::Moore2DGridQuery<Nomad> moore2DQuery(sp);
    moore2DQuery.query(center, neighbor);
    if (neighbor.empty()) //is a selected space occupied?
      sp->moveTo(id, newLoc);                  //if not, move there
    else                                       //otherwise,
      dst.erase(dst.begin()+r);//try another
} }
// Simulation Scenario
int main(int argc, char** argv) {
  boost::mpi::communicator world;
  repast::RepastProcess:init(argv[1]);// initialize Repast
  MyContext* context=new repast::Properties(&world);
  repast::ScheduleRunner& runner=repast::RepastProcess;
  context->init();              //populate agents.
  context->schedule(runner); //schedule 9 migration events.
  runner.run();                 //run a simulation.
  delete model;                 //finish all.
  repast::RepastPRocess::instance()->done;
}
```

Fig. 8. Agent migration in RepastHPC

C. FLAME

FLAME is another MPI-based ABM system, originally developed by University of Sheffield, UK. Since FLAME users write their simulation in C, for object-based programming purposes, they need to declare all agents, their data and method members, and environment variables in XML, in a similar way to C++ header files. Although FLAME uses environment variable such as *env_north_x, env_south_y, env_min_x, and env_max_y* to shape a simulation space, it does not instantiate any actual space on memory beyond making these variables accessible to all agents. Instead, agents are capable of broadcasting their messages among one another through message boards, each launched at a different MPI rank. Contrary to MASS and RepastHPC, both moving agents over a distributed simulation space, FLAME is considered as a collection of communicating, state-transitting agents statically mapped over MPI ranks. FLAME's other notable differences include:

1) ***Simulation events*** are scheduled in the model XML separately from auto-generated *main()* and C-described agents.
2) ***agent collisions*** are impossible to support at the system that cannot keep track of agent locations except looking into their initial coordinates to uniformly map agents over the system.
3) ***spatial information*** is captured by and maintained inside each agent that informs the others of its migration through a broadcast message.

Figure 9 describes RandomWalk in FLAME. The source code consists of two files: (1) an XML file for declaring events and agent functions and (2) a C program for describing each agent's random-walk logics. More specifically, XML schedules 10 events for each simulation cycle (lines 2-6) where the first event at time 0 calls each *nomad* agent's *new_turn* function (lines 11-15) and the rest nine events, each at time 1-9 invoke *random_walk()* (lines 16-20). The *new_turn()* function computes each agent's current cell number: 0-8 (lines 24-26). Only agents whose *MY_TURN* equals the current time 1 to 9 can invoke *random_walk()* that allows the calling agent to examine four potential neighbors to migrate to (lines 29-46). Although FLAME's *random_walk()* is almost similar to the logics of MASS and Repast, the notable difference is that a FLAME agent needs to read all the others' current locations to examine a potential collision (lines 33-37).

As compared to MASS and RepastHPC, FLAME is purely agent-based. There is no actual implementation of simulation spaces. Functions including main() necessary to start and stop a simulation, to allocate and deallocate memory space, and to facilitate inter-agent communication are all automated, which allows model designers to focus on their own agent descriptions.

D. Programmability Comparisons

Given the RandomWalk code above in MASS, RepastHPC, and FLAME, we compare their programmability in collision-free migration from the following four aspects.

1) Agent Handling and Description: For scheduling nine migration events, (each corresponding to moving agents on cells with the same number 0-8), MASS and FLAME can describe them in a for-loop or a simple XML statement, whereas RepastHPC needs to declare nine events separately. For each migration event, all these three systems must scan an entire list of agents whether or not they need to migrate at the current event. However, decision on moving a given agent can be made by each agent itself in MASS and FLAME while RepastHPC needs to describe such an agent selection separately in its *Context* logic, (i.e., in *MyContext::move()*), which implies that MASS and FLAME agents tend to be more self-descriptive. This is RepastHPC's inherent problem in using a separate *Context* class as an agent controller. Contrary to that, MASS decouples agents from places and FLAME is purely agent-based with no space concept.

```
<name>random walk</name>
<!− 1 location calculation & 9 move events per cycle − >
<environment> <timeUnits> <timeUnit>
  <name>turn</name><unit>iteration</unit>
  <period>10</period>
<timeUnit> </timeUnits> </environment>
<!− Agent declaration − >
<agents><xagent>
  <name>nomad</name>
   <functions>
    <function>
      <name>new_turn</name>    <!− scheduled at time 0 − >
      <condition><time><phase>0
      </phase></time></condition>
    </function>
    <function>
      <name>random_walk</name> <!− scheduled at 1 - 9 − >
      <condition><time><phase>nomad.my_turn
      </phase></time></condition>
    </function>
  </functions>
</xagent><agents>
// Agent logic
int new_turn() {
  MY_TURN = (myX % 3) + 3 * (myY % 3) + 1;// get 0-8
}
int random_walk() {
  bool unoccupied[] = {true, true, true, true};
  for (int i=0; i<4; i++) {//for each of N, E, S, and W
    int newX = computeCoord(myX, i);get its coordinates.
    int newY = computeCoord(myY, i);

    // START_MESSAGE_AGENTLOC_MESSAGE_LOOP
      if (message_agentloc_message->x == newX &&
          message_agentloc_message->y == newY)
         unoccupied[i] = false;
    // FINISH_MESSAGE_AGENTLOC_MESSAGE_LOOP
  }
  random = moveset.array[rand() % 4];choose a neighbor
  for ( int i = 0; i <
    if (unoccupied[random] == true) {if not occupied
      myX = newX;                     migrate there
      myy = newY;
      return 0;
    }
    random += (random + 1) % 4;       choose another
} }
```

Fig. 9. Agent Migration in FLAME

2) Collision Detection: Agent migration needs a space concept to detect agent collisions on given coordinates. Both MASS and RepastHPC facilitate such a space as the *Places* or the *SharedDescreteSpace* class, both visualizing remote processor boundary as ghost space to local agents. For this purpose, logically neighboring processors have to exchange their ghost space with each other at each migration event. On the other hand, due to lack of space concepts, FLAME must have each agent broadcast its current location to all the others and compare its coordinates with others to avoid collisions. This results in $N \times N$ message exchanges and comparisons by each migration event.

3) System-level Collision-free Migration: System-level location-ordered migration is feasible as far as a space concept is supported by a system. From this standpoint, MASS and RepastHPC have potential to implement this migration algorithm. However, direction-ordered migration is more challenging unless a system can access each agent's next destination. RepastHPC can capture such agent information only when an agent calls *projection.balance()* to migrate over a space, which is too late for collision detection. Needless to say, FLAME cannot access each agent's private data. The reason why MASS was able to support direction-ordered migration at the system level is that MASS can forecast each agent's next destination by comparing a difference in coordinates between the agent and its current place, (i.e., *Agent.index[]* and *Place.index[]*).

4) Quantitative Measures: For each of MASS, RepastHPC, and FLAME, we measured the number of files and lines of code (LOC) necessary to implement RandomWalk and embed an application-level location-ordered migration algorithm in it.

Systems	#Files	Lines of Code (LOC)
MASS	5	727
RepastHPC	5	921
FLAME	5 (16)	783

All of these three systems need five different files. Both MASS and RepastHPC requires (1) a simulation scenario in main.cpp, (2) a simulation space definition in Land.h, (3) its implementation in Land.cpp, (4) an agent definition in Nomad.h, and (5) its implementation in Nomad.cpp. On the other hand, FLAME needs one XML definition, three C-based agent-controlling functions, and one more C program that describes simulation rules. Notable is that, since FLAME automatically generates C template and stub files from the XML definition, the nominal number of files is 16. For the code size comparison, MASS was slightly the smallest. More specifically, it was 7% smaller than FLAME. Although we gave RepastHPC a few advantages by re-engineering the source code and subtracting space and comment lines from it, its LOC could not beat out neither MASS or FLAME. This was resulted from RepastHPC's coding style as discussed above: nine events must be declared independently and agent management need to be implemented in *Context* separately.

From these four observations, we feel that MASS facilitates collision-free migration more efficiently than the other two.

IV. Performance Evaluation

We coded all the four RandomWalk programs with MASS both at the application and system levels, with RepastHPC, and with FLAME. They respectively correspond to Figures 6, 7, 8, and 9 in Section III. To compare their execution performance, we implemented system-level direction-ordered collision-free agent migration in MASS C++ and installed RepastHPC 2.1.0, and FLAME 2.1.3 on a cluster of 16 3.4GHz Intel Core i7 desktop machines, each equipped with 16GB memory and running Ubuntu 14.04. Figure 10 shows an execution of RandomWalk with the MASS library.

A. Performance Comparison of MASS, Repast-HPC, and FLAME

We first compared MASS at both application- and system-level implementations (called MASS App and MASS Lib respectively in the following discussions) with RepastHPC and FLAME for their parallel performance.

Figure 11 shows their performance with four computing nodes as increasing the number of threads from one to four.

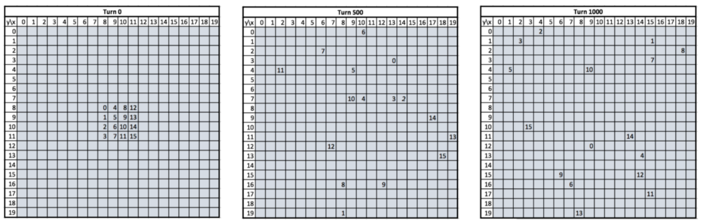

Fig. 10. A random walk execution with the MASS library

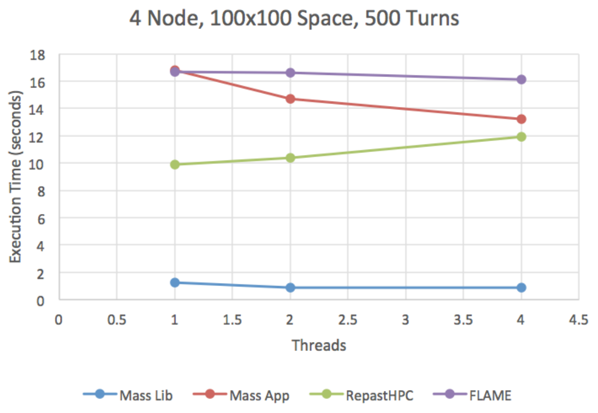

Fig. 11. Performance of MASS, RepastHPC, and FLAME running over 4 computing nodes

In this evaluation, 400 agents walked over a 100×100 space. With a single thread, RepastHPC performed 1.7 times better than MASS App and FLAME. This is because RepastHPC's *Context* object quickly scans a list of agents to invoke each agent's *randomWalk* function, whereas MASS needs two phases of operations: (1) updating ghost space among neighboring places and (2) calling each agent's *randomWalk()*; and FLAME needs $N \times N$ inter-agent communication over four computing nodes. However, using four threads per each node, RepastHPC slowed down due to its serialized accesses to the same agent list by multithreaded *Context* objects. FLAME could not improve its parallel performance because of its inter-agent communication overheads. On the other hand, MASS App improved 1.27 times faster than its single-threaded execution. This is because MASS controls fine-grained synchronizations among threads.

Notable is MASS Lib's performance that showed 7.9 to 13.6 times faster than RepastHPC. The biggest factor of this improvement is that MASS Lib groups agents into north, east, south, and west directions, thus scans the agent list only once for this grouping work, and completes migration in four turns. On the other hand, RepastHPC needs nine turns of migration where each turn must scan the entire agent list.

Figure 12 measures execution performance with 16 computing nodes. The evaluation was able to extend its simulation size to 3600 agents over a 300×300 space. The reason was that both RepastHPC and FLAME needed a large space of memory

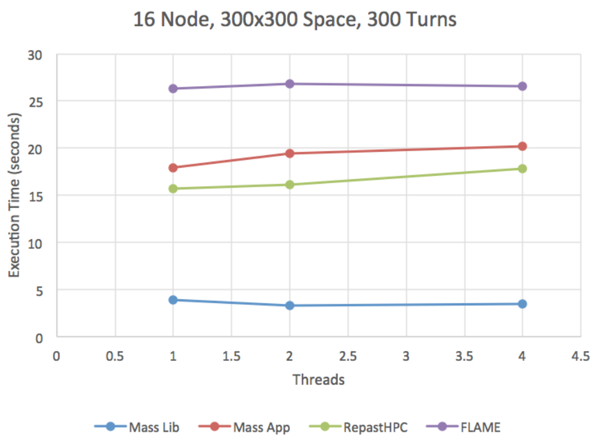

Fig. 12. Performance of MASS, RepastHPC, and FLAME running over 16 computing nodes

that couldn't fit a smaller number of computing nodes. Similar to the four-way parallelization, FLAME was the slowest and unable to improve its performance with multithreading due to its communication overheads. RepastHPC ran faster than MASS App whose performance loss was however mitigated to 14%-21% slow-down as we used a larger problem size. Again, MASS Lib performed fastest among the four test cases, more specifically 4.0 to 5.1 times better than RepastHPC.

B. *Performance of MASS Library*

We focused on the MASS library's CPU scalability for its collision-free agent migration. Our performance measurement walked 3,600 agents over a 300×300 space, (i.e., 4% population density) as increasing the number of nodes from one to 16 and the number of threads from one to four.

Figures 13 and 14 demonstrate MASS App's and Lib's CPU-scalable execution respectively. Despite RandomWalk's fine-grained pallalelization where each agent computes only the next destination to visit, MASS App showed that 16 single-threaded computing nodes performed 2.2 times faster than a sequential execution. MASS Lib scaled up its parallelization to 8 ways, (i.e., four computing nodes, each with two threads) and ran 2.9 times faster than a sequential execution.

These results confirm three advantages of the MASS library: (1) both MASS App and Lib are CPU scalable; (2) MASS Lib is at least four times faster than the other ABM simulators; and (3) MASS saves memory space efficiently to run a larger simulation with a fewer computing nodes.

V. Conclusions

Needless to say, there are many applications that need no agent migration, which FLAME could benefit well with static agents that communicate with each other. On the other hand, a number of application domains need to observe agent migration over a space: vehicle flow in traffic simulation, pedestrian flow in emergency evacuation, and immune cells' attack to bacteria, etc. Model designers used to address such

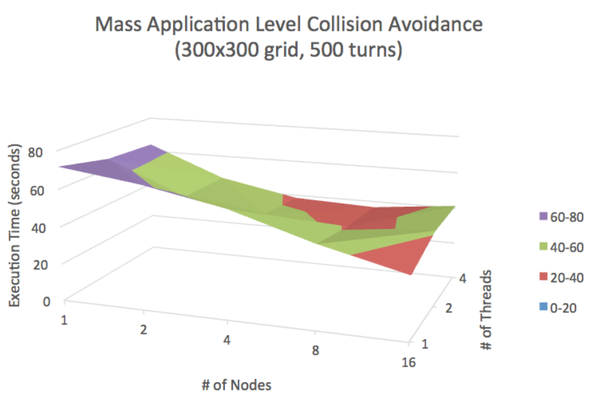

Fig. 13. Performance of MASS application-level collision-free migration

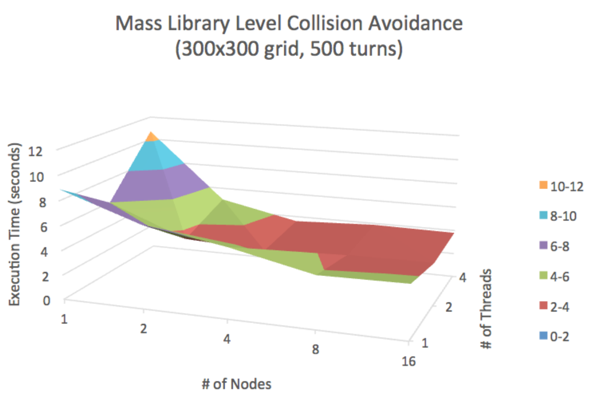

Fig. 14. Performance of MASS system-level collision-free migration

agent collisions in conceptual models [11], [10] by controlling agent migration or slicing simulation time and space finely or to implement collision avoidance mechanisms in their application-specific parallel execution [8], [9]. Therefore, from the programmability viewpoint, it is worthwhile supporting system-level collision-free agent migration in general-purpose simulators.

As shown in Figure 5, agent collisions increase in proportional to the growth of agents' population density until the growth reaches 50% of the entire simulation space. As observed in Figure 3, if agents are uniformly distributed to occupy less than only 1/9 or 1/5 of an entire simulation space in the Moore or the von Neumann neighborhood, the probability of agent collisions remains negligible (but does not result in zero even with the 1% population density). In such scenarios, the trial-and-error migration could most effectively reduce the turns of agent migration. However, the population of agents is dynamic in most cases, (e.g., in Sugarscape [13]), and agents may gather around (as observed in Wa-Tor [14]). Once a collision is detected, the trail-and-error algorithm may cost more than the other two migrations.

Therefore, we developed and focused on location and direction-ordered collision-free migration. Since the MASS library has access to each agent's index[] variable, (i.e., the current and the next locations), it was able to implement direction-ordered migration at the system level. This paper demonstrated its simple programmability in and faster execution of system-supported agent migration.

Our future plan is to continue our verification work on the MASS library's collision-free migration, using actual applications such as Sugarscape and Wa-Tor. We recently made the MASS library available to the public through: `http://depts.washington.edu/dslab/MASS`.

References

[1] MATSim Homepage, "http://www.matsim.org," 2012.
[2] F. Kawasaki, "Accelerating large-scale simulations of coortical neuronal network development," Master's thesis, Master of Science in Computing and Software Systems, University of Washington, 2012.
[3] D. L. Chao, M. E. Halloran, V. J. Obenchain, and I. M. Longini Jr, "FluTE, a Publicly Available Stochastic Influenza Epidemic Simulation Model," *PLoS Computational Biology*, vol. Vol.6, no. No.1, pp. 517–527, January 2010.
[4] M. Oryani, "Applying agent-based modeling to studying emergent behaviros of the immune system cells," Master's thesis, KTH Electrical Engineering, Stockholm, Sweden, May 2014.
[5] R. M. D'Souza, S. Marino, and D. Kirschner, "Data-parallel algorithms for agent-based model simulations of tuberculosis on graphics processing units," in *Proc. of Agent-Directed Symposium - ADS09*. San Diego, CA: SCS, March 2009.
[6] Argonne National Laboratory, "Repast for High Performance Computing, http://repast.gifhub.io/repast_hpc.html."
[7] C. Geenough and M. Holcombe, "FLAME Flexible Large-scale Agent Modeling Environment, http://www.flame.ac.uk."
[8] A. U. K. Wagoum, B. Stefen, A. Seyfried, and M. Chraibi, "Parallel real time computation of large scale pedestrian evacuations," *Advances in Engineering Software*, vol. Vol.60-61, pp. 98–103, 2013.
[9] J. Barceló, J. Ferrer, D. Garcća, and R. Grau, "Microscopic traffic simulation for att systems analysis. a parallel computing version," in *25th Anniversary of CRT*, August 1998.
[10] A. Kirchner, H. Klüpfel, K. Nishinari, A. Schadschneider, and M. Schreckenberg, "Discretization effects and the influence of walking speed in cellular automata models for pedestrian dynamics," *Journal of Statistical Mechanics: Theory and Experiment*, vol. Vol.2004, no. No.10, p. P10011, October 2004.
[11] S. Bandini, L. Crociani, and G. Vizzari, "Towards a more comprehensive estimation of social costs in pedestrian facilities," in *Proc. of the Workshop on The Challenge of Ageing Society: Technological Roles and Opportunities for Artificial Intelligence in conjunction with the 13th Conference of the Italian Association for Artificial Intelligence, (AI*IA 2013)*, Turin, Italy, December 2013, p. paper 7.
[12] C. L. Barrett *et al.*, "TRANSSIMS(TRansportation ANalysis SIMulation System) Volume 0 - Overview," Los Alamos National Laboratory," LA-UR-99-1658, May 28 1999.
[13] J. Epstein and R. Axtell, *Growing artificial societies: social science from the bottom up*. Brookings Institution Press, October 1996, p. 224.
[14] A. K. Dewdney, "Computer recreations sharks and fish wage an ecological war on the toroidal planet wa-tor," *Scientific American*, pp. 14–22, December 1984.
[15] M. Fukuda, "MASS: A Parallelizing Library for Multi-Agent Spatial Simulation, http://depts.washington.edu/dslab/MASS."