Iterative Grid-Based Computing Using Mobile Agents

Hairong Kuang, Lubomir F. Bic, Michael B. Dillencourt
Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
{hkuang, bic, dillenco}@ics.uci.edu

Abstract

We describe an environment for the distributed solution of iterative grid-based applications. The environment is built using the MESSENGERS mobile agent system. The main advantage of paradigm-oriented distributed computing is that the user only needs to specify application-specific sequential code, while the underlying infrastructure takes care of the parallelization and distribution. The two paradigms discussed in this paper are the finite difference method, and individual-based simulation. These paradigms present some interesting challenges, both in terms of performance (because they require frequent synchronized communication between nodes) and in terms of repeatability (because the mapping of the user space onto the network may change due to load balancing or due to changes in the underlying logical network). We describe their use, implementation, and performance within a mobile agent-based environment.

Keywords: mobile agents, programming paradigms, grid-based computing, finite difference methods, individual-based simulations

1. Introduction

Developing distributed application is significantly more difficult than developing sequential applications. Paradigm-oriented distributed computing greatly simplifies the task of distributed computing. It makes use of the computation and communication skeletons, while the application programmer only has to provide the application-specific components. The advantage of this approach is that it hides the details of distributed computing such as task partitioning, communication, and synchronization.

We have built a paradigm-oriented distributed computing environment on top of the MESSENGERS mobile agent infrastructure [2, 3]. The autonomous migration ability of agents makes them capable of utilizing a dynamically changing network. Their inherent portability allows them to handle the distribution of tasks in a heterogeneous environment in a transparent manner. Agent mobility can also be exploited for load balancing and fault tolerance. Because of these features, mobile agents lend themselves naturally to paradigm-oriented distributed computing.

In previous research, we investigated three common paradigms: bag-of-tasks, branch-and-bound, and genetic programming [9, 10]. The reason for choosing those paradigms was that applications that fit those paradigms can be easily divided into multiple, highly independent tasks. In the bag-of-tasks paradigm, there is no communication at all between tasks. In the branch-and-bound and genetic-programming paradigms, exchanges of information between tasks are non-essential communications, in the sense that they do not affect the correctness although they may be quite useful for optimization purposes.

In this paper, we investigate more complicated paradigms in the area of iterative grid-based computing. Iterative grid-based applications are suitable candidates for distributed computing. They usually require large amounts of computation, which makes it worth distributing tasks over multiple computers. Spatially oriented computations make it easy to partition the simulation space. Near neighbor computations demand only near neighbor communications, which reduces the communication cost. The paradigms we investigated are the finite difference method for solving partial differential equations [14] and spatially oriented individual-based simulations in which the behavior of entities is based solely on their interactions with nearby entities [13].

Unlike the three paradigms discussed in [9, 10], iterative grid-based paradigms require frequent synchronized communications that are also essential communications, meaning that they are necessary for correctness. The presence of essential communications increases the challenge of providing a distributed implementation with reasonable performance characteristics. In this paper, we describe an implementation that addresses that challenge. We describe the two iterative paradigms in detail in Section 2, and then present, in Section 3, the underlying implementation using
1. FDM() {
2.   double vals1[XSIZE][YSIZE], vals2[SIZE][YSIZE];
3.   double *oldVals, *newVals;
4.   double ΔX = XLEN/XSIZE, ΔY = YLEN/YSIZE;
5.   // initialization
6.   for ( every element (X, Y) ) {
7.     vals1[X][Y] = Init( X, Y, XSIZE, YSIZE, ΔX, ΔY );
8.   }
9.   oldVals = vals1;
10.  newVals = vals2;
11.  // iterative computation
12.  until (Termination_condition) {
13.    for ( every element (X, Y) ) {
14.      newVals[X][Y] = Compute( X, Y, oldVals, ΔX, ΔY, Δt );
15.    }
16.    newVals ↔ oldVals;
17.  }
18.  // output
19.  for ( every element (X, Y) ) {
20.    WriteResult( outFile, X, Y, oldVals );
21.  }
22. }

Figure 1. The finite difference paradigm specification

MESSENGERS. Performance results are discussed in Section 4.

2. Paradigms

2.1. Finite difference paradigm

The finite difference method [14] is a common approach for solving differential equations. In the finite difference method, we have a discrete d-dimensional grid of element locations, and we want to compute a value \( u_{x,t} \) at each grid location \( x \) and for a regular sequence of times \( t \). The value of \( u_{x,t+Δt} \) is a function of \( u_{x,t} \) and the values of \( u \) at the neighbors of \( x \) at time \( t \).

Figure 1 shows the structure of the finite difference paradigm for a 2D problem. The algorithm starts by initializing the values of all elements (lines 6–8), and pointers to element buffers (lines 9–10). It then repeatedly computes the new values of all elements for each time step until the termination condition is satisfied (lines 12–17). In each iteration of the computation, the value of each element gets updated (lines 13–15), and the pointers to the two element buffers are swapped (line 16). Finally, element values are written into the output file (lines 19–21).

Specifying a finite difference paradigm problem requires four functions.

1. **Init**: this is the initialization function that initializes the value of each element.

2. **Termination_condition**: this specifies the criteria for terminating the computation. A few special termination condition routines are provided, such as terminating after a user-specified number of iteration or terminating when the \( L_1, L_2, \) or \( L_∞ \) difference between successive arrays of grid values is below a user-specified tolerance.

3. **Compute**: this is the main function of the paradigm. Given its own value and its neighbors’ values in the previous time step, it will produce the new value for the current time step.

4. **WriteResult**: this is an output function, which writes the value of an element to an output file.

In addition, users need to specify several parameters. The geometry parameters include the dimension of the grid, the number of nodes (SIZE) in each dimension, the length (LEN) of each dimension, and whether the grid is toroidal or not. The time interval (Δt) is also required, as are parameters required by the termination condition. These parameters are specified through a graphical user interface, which is an extension of the interface described in [9].

2.2. Individual-Based Simulation Paradigm

Individual-based simulation programs are widely used to simulate the behaviors of a collection of entities, for example, the movements of molecule particles [5], the schooling behavior of fish or birds [7, 13], and the evolution of an ecology environment [6]. The simulated entities move in a specified space over a period of time. At each time step, an entity decides its own behavior by interacting with its nearby environment and surrounding entities. Typically, an entity has an associated radius of visibility, and its behavior is affected only by entities within this radius. The state of an environment may include information, such as the temperature, the speed of a current, the amount of food, or the shape and the size of an obstacle. The state of an entity may include its position, moving speed, moving orientation, age, and so on.

The individual-based simulation paradigm is similar in several respects to the finite difference paradigm. First, in both paradigms, the simulated space is partitioned into a grid. Secondly, both are time-dependent iterative computations. Lastly, both are near-neighbor computations: the computation on each grid cell depends on only the states of the neighboring grid cells. Nevertheless, these two
paradigms also have differences. The finite difference computation is static, while the individual-based simulation is dynamic. In a finite difference computation, each grid has only one element, which is not moving. In an individual-based simulation, each grid cell has zero or more entities, moving around and interacting with the environment and with each other.

A typical individual-based simulation program starts by initializing the environment and the entities and then repeatedly executing simulation steps. In each step, the state of each entity and the state of the environment are updated. The state of each entity is updated based on its surrounding environment and its interactions with nearby entities, all of which lie within its radius of visibility. As a result of these interactions, an entity may move to a new position, new entities may be spawned, and old entities may be killed or die.

Variants of the individual-based simulation paradigm arise depending on how the collision problem is handled. This problem arises because entities share resources (including space), and hence must contend for resources. For example, two entities may move to the same position, their paths may intersect, or they may decide to eat the same food. Collision detection and resolution is an important issue in individual-based simulation.

Our individual-based simulation paradigm supports two frameworks of collision detection and resolution: the immediate state update method and the delayed state update method. In the delayed state update method, each entity makes a tentative decision based on the old state of the previous time step. After all the entities are finished, possible collisions are detected and resolved. The advantage of the delayed state update method is that the decision making is easy, but the collision detection and resolution are complicated, and sometimes will cause a domino effect.

An individual-based simulation program using the delayed state update method needs to keep two sets of states. The old state stores the snapshot of the simulated space and the entities at the end of the previous time step, while the current state stores the snapshot in the current time step. The structure of individual-based simulation programs using the delayed state update method is very similar to that of finite difference programs, which can be deduced in a straightforward way. We omit its details here.

In the immediate state update method, each entity makes a collision-free decision based on the current state. After an entity’s own state and its surrounding environment are updated, the old states are thrown away. Because an entity uses the most current states, collision detection and resolution is much easier to conduct. Collision detection and resolution can be combined with the process of state update, so that a global collision detection and resolution stage is no longer needed. The immediate state update method requires less memory compared to the delayed state update method since the old state does not need to be kept.

Figure 2 shows the structure of the individual-based simulation paradigm using the immediate state update method. It solves a 2D problem. First, the environment and all the entities get initialized (line 8–16). The while loop (line 17–31) is the main body of the program, which simulates the behavior of entities for a fixed number of iterations. At each iteration, all the entities update their states separately (line 20–26). The program keeps only the most current state. Before an entity’s state is updated, it is dequeued (line 21). The entity’s state is modified in place and the environment is updated incrementally (line 23). An entity can detect and resolve possible collisions while updating its state. Any updating of the environment that is independent of the effect of the entities is performed on lines 27–30.

The individual-based simulation paradigm is quite complicated and even has many variants. However, the details of the paradigm can be hidden from users. Specifying an individual-based simulation application requires specifying only four functions:

1. newEnv: this initializes the environment in a grid cell.
2. newEntity: this initializes an entity.
3. updateEntity: this updates the state of an entity at a time step.
4. updateEnv: this updates the state of an environment on a grid cell at a time step.

Users also need to specify the geometry of the simulation space. XLEN is the width of the space, while YLEN is the length of the space. XSIZE is the number of grid cells at dimension x, while YSIZE is the number of grid cells at dimension y. Δt is the time interval between time steps. NUM_OF_ENTITIES is the initial number of entities in the space. MAX_TIME_STEPS specifies the number of simulation steps. R is the radius of visibility for an entity; this must be smaller than both XLEN/XSIZE and YLEN/YSIZE.

In addition, two data structure definitions must be provided, one describing an environment (Env), the other describing an entity (Entity). The data structures are defined by the user, but the Entity structure must include the following fields:

```c
{
    ...
    long id;
    double x;
    double y;
    ...
}
```
1. IBS() {
2. Env env[XSIZE][YSIZE];
3. School entityGroup[XSIZE][YSIZE];
4. School *entityList;
5. Entity entities[NUM_OF_ENTITIES];
6. Entity *curEntity;
7. double ΔX=XLEN/XSIZE, ΔY=YLEN/YSIZE;
8. // initialize fish school and environment
9. for ( all the grid cell (i, j) ) {
10.   newEnv( &env[i][j], i, j, XSIZE, YSIZE, ΔX, ΔY );
11.   newEntityList( &entityGroup[i][j] );
12. }
13. for( i=0; i<NUM_OF_ENTITIES; i++ ) {
14.   newEntity( &entities[i], i );
15.   add entity i to the corresponding current grid cell;
16. }
17. // iterative simulation
18. while( t<MAX_TIME_STEPS ) {
19.   t++;
20.   for ( each grid cell (i, j) ) {
21.     while( curEntity = popEntity( &entityGroup[i][j]) != NULL ) {
22.       entityList = getNeighbors( curEntity,
23.                                   entityGroup, R);
24.       (entityList, env) = updateEntity( t, curEntity,
25.                                             env, entityList, ΔX, ΔY, Δt);
26.       add each entity in entityList to entityGroup;
27.     }
28.   }
29.   for ( each grid cell (i, j) ) {
30.     entityList = updateEnv( t, &env[i][j], i, j,
31.                                 ΔX, ΔY, Δt);
32.     add each entity in entityList to entityGroup;
33. }
34. }
35. }

Figure 2. The individual-based simulation paradigm specification

The field id contains an identifier which uniquely identifies an entity. The fields x and y specify the position of the entity in the 2-D simulated space. The visibility of these two fields allows the system to transparently manage entity lists. The entity list management issue is common to all the individual-based simulation applications and independent of specific applications. When an entity adjusts its position, we need to know if it has moved out of the current grid cell. If the entity has moved out of the grid, it will be deleted from the old list and added to a new list. In our individual-based simulation paradigm, we provide an entity list management library, which includes a data structure describing an entity list, and a list of operations to manipulate an entity list. For example, getNeighbors() function returns a list of entities within a specific radius of a given entity, newEntityList() creates an empty entity list with a default size, AddEntity() adds an entity to an existing entity list, and PopEntity() pops the first entity from an entity list. Functions are also provided to navigate an entity list. These entity list operations can be called by the user-defined functions.

3. Implementation of Paradigms Using MESSENGERS

3.1. Finite difference paradigm

Figure 3(a) shows a logical network to support the distributed implementation of the finite difference paradigm. Each node represents a place to which a messenger can hop. The “Meeting room” node is where the tolerance is gathered and computed. If the termination condition is satisfied, a termination notification will be sent to all the “Office” nodes. An “Office” node is where the element values assigned to the node get iteratively updated. At each iteration, a new value for each of the elements is computed, and boundary information is exchanged with its neighbors when
all the boundaries become obsolete. The “Office” nodes can be connected either as a two-dimensional grid (as shown in the figure) or as a ring, depending on how the user grid is partitioned. If the user grid is partitioned into strips, the “Office” nodes are connected as a ring. The advantage of this type partition is that each partition has only two neighboring strips, which simplifies the boundary exchange and synchronization. If the user grid is partitioned into rectangular blocks, the “Office” nodes are connected as a 2D grid. The advantage of this partition technique is that each partition exchanges less boundary message with its neighbors.

Figure 4. The structure of a rectangular partition

The implementation uses four types of messengers as shown in Figure 3(b). The initialization messenger (i) builds the logical network and injects the worker (w) messengers into offices, one per office. Each worker messenger initializes its partition and starts boundary messengers (b). Then each partition is repeatedly updated by the worker messenger and the boundary messengers until the termination condition is satisfied. Figure 4 shows a structure of a partition when the user grid is partitioned into rectangular blocks, in which the light gray area is the ghost boundary, the dark gray area is the boundary of the partition, and the white area is the inner part of the partition. At each iteration, the worker messenger updates the grid cells in the white area, which do not depend on the values of the grid cells in the neighboring partitions. Each partition has eight boundary messengers, one per side and one per corner. Each boundary messenger alternates between two neighboring “Office” nodes. At each iteration, a boundary messenger updates the grid cells on the portion of the boundary it takes care of. It then carries its boundary information, hops to a neighboring “Office”, and deposits the data at the ghost boundary. At the next step, it works at the neighboring “Office” node, updates a portion of its boundary, and then carries the boundary information back to the departure node. Each “Office” node also has a messenger that gathers the tolerance information at the end of each iteration and carries it to the “Meeting room”. These are called “report” messengers (r). Once all the ant messengers have carried their information to the meeting room, the tolerance is computed. If the termination condition is met, the ant messenger will hop back to its “Office” and notify the workers.

The above implementation uses several strategies to improve the performance. In principle, it attempts to update each boundary and send it to its neighbor as early as possible. In this way the communication and computation can be overlapped, and the idle waiting time can be avoided. At the beginning of each iteration, the worker messenger sends out a signal, which wakes up incoming boundary messengers and allows them to update the boundary and carry the boundary data to its neighboring node. However, if some of the neighboring nodes are slower and the boundary messengers have not arrived yet, the worker messenger does not wait for the slower nodes. Instead it goes ahead and updates the inner part of the partition, but it interrupts itself periodically to give late incoming boundary messengers a chance to work. As a result, it eliminates the necessity of a barrier at each step, and therefore squeezes out the idle waiting time.

3.2. Individual-based simulation paradigm

Our distributed implementation of the individual-based simulation paradigm is similar to that of the finite difference paradigm, reflecting the similarities in the paradigms themselves. They have the same possible logical networks and similar types of messengers, and near neighbor boundary exchange is also required for each time step. One major difference arises because of the dynamic migration of the entities in individual-based simulations. Because of this migration, near-neighbor communication in the distributed implementation requires two steps: first the emigrating entities move to the neighboring nodes, then the boundary information is exchanged. Another difference arises because of the more dynamic nature of the individual-based simulation paradigm. In the finite difference paradigm, once the partition of the user grid is fixed, the load on each machine and the message size exchanged remains constant for the duration of the simulation. In the individual-based simulation paradigm, because entities are moving in the space, the load on each machine and the message sizes are dynamically changing.

Our implementation of the individual-based simulation paradigm supports the same logical network structures as in the individual-based simulation paradigm. The “Meeting room” node is where the positions of all the entities are gathered and sent to the client to be visualized. An “Office” node is where a worker iteratively updates the states of the entities and environment assigned to the node. At each time step, a worker updates entities and environments, sends the
entities migrating to the neighboring spaces away, and exchanges boundary information.

The system has three types of messengers. The initialization messenger \( i \) builds the logical network and injects the worker \( w \) messengers into offices, one per office. Each worker messenger initializes its partition. It then repeatedly updates its grid cell until the termination condition is satisfied. At each iteration, it receives its neighbors' boundaries, computes entities' new states, sends emigrating entities to its neighbors, receives immigrating entities, and sends the boundary to its neighbors. A shuttle messenger \( s \) carries emigrating entities to its neighboring node and brings the neighbor's boundary back.

An important issue in mobile agent-based implementations of individual-based systems is the representation of migrating entities. In [2], each entity is represented as a separate agent. This allows an entity to freely migrate to its destination, but it means that each migrating entity is a separate hop of an agent from one node to another. In our implementation, we use shuttle agents that synchronously carry dynamic clusters of entities from one node to another. This is arguably a less natural way to address entity migration, but it decreases network traffic considerably and hence increases performance and scalability considerably. What is really needed is an increase in the number of agents that can hop in a short period of time without degrading system performance. Some work in this direction is described by Fukuda et al. [4].

### 3.3. Repeatability

An important goal of a simulation is repeatability: a user should have the option of rerunning a simulation and obtaining exactly the same results. This can be very important for validating changes made at the application level or for tracking down elusive application bugs. Achieving repeatability in distributed implementations presents some interesting challenges, due to the repartitioning of the user grid (i.e., changes if the mapping of the user grid onto the logical nodes). Repartitioning may occur within a run due to load balancing, and it may also occur from one run to another if the user runs the same simulation but changes the configuration (e.g., changes the logical network or the number of machines).

One issue that must be addressed to achieve repeatability is random number generation. In order for two simulations to achieve the same result, the random choice made during the second run must be exactly the same as the corresponding random choice made during the first run. This can be achieved in various ways: for example, a stream of random numbers can be associated with each entity, or a stream of random numbers can be associated with each user grid cell. The paper [11] contains a comparison of these two approaches and a few others as well.

The second issue that affects repeatability is the order in which entities are processed. This order affects the result of the simulation when the immediate state update method is used. We introduce an odd-even labeling scheme to specify a particular order in which entities are updated. This scheme ensures that entities are processed in the same order, irrespective of the partitioning of the user grid.

The scheme works as follows: we label each user grid cell with its index \((x, y)\), and then label the cell with a number in the range 0–3. The label is 0 if both \(x\) and \(y\) are even, 1 if \(x\) is even and \(y\) is odd, 2 if \(x\) is odd and \(y\) is even, and 3 if both \(x\) and \(y\) are odd. At each time step, we update the states of all entities located at user grid cells with a particular label (starting with 0) before proceeding on the next label. The odd-even scheme is shown in Figure 5.

In order to make this labeling scheme work correctly, the boundary exchange between neighboring partitions needs to be expanded, and a condition must be imposed on the size of the user grid. The reason for the expanded boundary exchange is illustrated in Figure 5. The white area is the collection of user cells that are allocated to the machine, and the gray area is the ghost boundary (i.e., the boundary data obtained from the neighbor as part of the boundary exchange). In order to correctly update the cell with label 3 in the lower left corner of the area allocated to the machine, we must have the updated contents of the cell with label 2 immediately below it, which in turn requires the cell with label 1 below it and to its left, which in turn requires the cell with label 0 below it. To update this last cell, we need all its neighbors. It is not hard to see that this example implies that to correctly update the white area in all cases, the exchanged boundary must consist of a layer of 4 cells around the white area. In other words, this scheme requires expanding the size of the exchanged boundary by a factor of 4.

![Figure 5. Distributed odd-even individual-based simulation paradigm](image-url)
The odd-even labeling scheme represents a coloring of the user grid in the graph-theoretical sense, namely that two cells that touch, either along an edge or at a corner, are assigned different labels. In fact, if two entities are located in two different grid cells that are assigned the same label, the distance between them is at least as large as the length of the shortest side of a grid cell. In a typical individual-based system model, there are parameters $r_v$ and $r_m$, which respectively represent the radius of visibility and the radius of motion. An entity’s behavior in a time step can only be affected by another entity if the second entity is within a distance of $r_v$ of the first, and an entity can move a distance of at most $r_m$ in one time unit. Generally it is assumed $r_m \leq r_v$. It must be true that the length of a user cell must be at least $r_v$ (note that this is enforced as described in Section 2.2). If we strengthen this constraint by requiring that the length of the shortest side a user grid cell is at least $(r_v + r_m)$, then an entity in a grid cell cannot affect an entity in a different grid cell with the same label. Hence, with this strengthened constraint, the odd-even labeling scheme will guarantee repeatability when the immediate-update method is used.

4. Performance Evaluation

4.1. Finite difference paradigm

We tested the finite difference paradigm using Metropolis Monte Carlo algorithm, which solves the Ising model [1, 8]. The Ising model is one of the pillars of statistical mechanics. It consists of an array of spins which can be pointing “up” or “down”, and interact with neighboring spins. Each spin and its neighboring spins have an energetic preference to be the same value. Energy is given by

$$E = -J \sum_{i \neq j} S_i S_j$$

where $S$ is equal to $+1$ or $-1$ as spin state, $(i, j)$ are nearest neighbors, and $J$ is the interaction strength. The Metropolis Monte Carlo algorithm uses Boltzman’s rejector for energy fluctuation.

In our experiments, the simulated space is a 2-D toroidal grid, of which spin states are initialized randomly and changed for 500 steps. At each step, each spin makes a tentative flip and uses Boltzman’s rejector to decide if this change is accepted or not. We varied the grid size to see how the problem size influenced the speedup.

Figure 6 shows the speedup of the distributed finite difference programs running on 9 machines. The figure shows that the speedup of the distributed programs increases as the user grid size increases. This is because a program with a larger user grid has a larger program size, which represents a bigger computation-to-communication ratio. We can also see that programs with a rectangular partition consistently have better speedup than those with a strip partition. This is because programs with a rectangular partition have a smaller boundary than programs with a strip partition. Therefore, programs with a rectangular partition has smaller amount of communication data, which leads to a better speedup.

In our implementation of the distributed finite difference paradigm, the worker messenger interrupt itself periodically to give boundary messengers a chance to work. Another type of experiments has been performed to see how the interrupt rate influences the performance. We use programs with a rectangular partition since they have a better speedup.
Figure 7 shows the performance of the distributed finite difference programs running on 9 machine when the interrupt rate varies. The horizontal axis represents the size of the user grid, while the vertical axis represents the execution time in seconds. Different lines represent the performance of the programs with different interrupt rates. When the interrupt rate is equal to zero, the worker messenger does not interrupt itself during its computation at each iteration, while when the interrupt rate, denoted as $r$, is greater than zero, the worker messenger interrupts itself $r$ times. The figure shows that the performance of the distributed programs improves as the interrupt rate increases no matter the size of the program size. When the interrupt rate increases from zero to one, the performance improves the largest. As the interruption gets more frequent, the performance still improves but not as significant. This is because the interruption of a computation brings extra context switch cost.

4.2. Individual-based simulation paradigm

We tested the individual-based simulation paradigm using a fish schooling model described in [7]. This model assumes a 2-dimensional space where each fish periodically adjusts its position and velocity by coordinating its movement with up to four of its neighbors. We tested the paradigm using both the delayed state method and the odd-even immediate state update method. To make the programs using these two methods comparable, we made a few changes to the fish schooling model. After each fish calculates its new velocity by coordinating with its neighbors, it will discard the effort and adjust its position by moving a random angle. As a result, each fish moves around independently. Whatever method is used, each fish will move at the same trace, which guarantees the amount of computation will be the same. The simulation space is a 2-dimensional 300 by 300 toroid in which fish move as a single school of fish for 500 simulation steps. The positions of all the fish are logged in output files. Because of the uncertainty of distributed programs, we run each program three times. The execution time presented is the average of three runs.

Figure 8 compares the performance of the delayed state update method and the immediate state update method using the odd-even process order. The horizontal axis represents the number of simulated fish, i.e., the problem size, while the vertical axis represents the execution time. The figure shows that the delayed state update method performs better than the odd-even immediate state update method if run in the distributed way. This is because the odd-even immediate state update method has larger communication volume. While in the sequential situation, the odd-even immediate state update method runs faster when the problem size becomes larger. This is because the odd-even immediate state update method uses less memory compared to the delayed state update method.

Figure 9 shows the speedup of distributed fish schooling simulation programs. The horizontal axis represents the number of simulated fish, while the vertical axis represents the speedup, which is the ratio of the execution time of the sequential program to the execution time of distributed programs which run on 9 machines. The figure shows that the speedup of the distributed programs increases as the problem size increases. This is because the computation-to-communication ratio increases with the increase of the problem size. From the figure, we can also see that the distributed delayed state update programs have better speedup than the distributed odd-even immediate state update programs. This is because distributed odd-even immediate state update programs have better speedup than the distributed odd-even immediate state update programs. This is because distributed odd-even immediate state update programs send and receive four times message size as large as distributed delayed state update programs each time when the boundary information is exchanged. Distributed odd-even programs also need to duplicate redundant computations.
Another issue related to the performance of distributed individual-based simulation programs is the load imbalance on each machine. As we discussed in the previous section of the paper, because the simulated entities move around the space, the load on each machine will dynamically change accordingly. In the experiments we performed, as the fish randomly walks in the space, the load statistically should be balanced. However, in a snap shot of a experiment with 5000 fish performed on 9 machines, we found the most heavily loaded machine has 622 fish, while the least loaded machine has 490 fish. The most heavily loaded machine has 25.5% more load than the least loaded machine. Therefore, a load balance scheme which can dynamically balance the load on each machine will increase the performance of distributed individual simulation. Although we did not present the performance of the load-balanced distributed individual-based simulated programs in this paper, our group has investigated the load balance mechanism in [12].

5. Final Remarks

Paradigm-oriented computing simplifies distributed implementation by providing a coordination layer that insulates the application programmer from the details of the distributed computation. We have described the specification and implementation of environments supporting paradigms for solving finite difference equations and simulating individual-based systems. Our performance evaluations have shown that significant speedups can be achieved on a network of workstations, cooperating with each other using a system of mobile agents.

References