

PODC: Paradigm-oriented distributed computing

Hairong Kuang*, Lubomir F. Bic, Michael B. Dillencourt

Information and Computer Science, University of California, Irvine, CA 92697-3425, USA

Received 4 February 2002; received in revised form 27 October 2004; accepted 26 November 2004

Abstract

We describe an environment for distributed computing that uses the concept of well-known paradigms. The main advantage of paradigm-oriented distributed computing (PODC) is that the user only needs to specify application-specific sequential code, while the underlying infrastructure takes care of the parallelization and distribution. The main features of the proposed approach, called PODC, are the following: (1) It is intended for loosely coupled network environments, not specialized multiprocessors; (2) it is based on an infrastructure of mobile agents; (3) it supports programming in C, rather than a functional or special-purpose language, and (4) it provides an interactive graphics interface through which programs are constructed, invoked, and monitored.

We discuss five paradigms presently supported in PODC: the bag-of-tasks, branch-and-bound search, genetic programming, finite difference, and individual-based simulation. We demonstrate their use, implementation, and performance within the mobile agent-based PODC environment.

© 2005 Elsevier Inc. All rights reserved.

Keywords: Distributed computing; Paradigm-oriented computing; Mobile agents; Bag of tasks; Branch and bound; Genetic programming; Finite difference; Individual-based simulation

1. Introduction

Increasingly powerful workstations and PCs, interconnected through various networks, continue to proliferate throughout the world. Most are greatly underutilized and thus represent a significant computational resource, which could be tapped for running applications requiring large amounts of computations. Unfortunately, developing distributed applications is significantly more difficult than developing sequential applications. To make the distributed computational resources available to a broader class of users, a number of tools and environments have been proposed, which attempt to simplify the development of distributed programs. One of the pioneering ideas was to provide programming skeletons [3], which embody specific well-known paradigms, such as divide-and-conquer or a bag-of-tasks.

Each skeleton is a high-level template describing the essential coordination structure of the algorithm. Only the problem-specific data structures and functions need to be supplied by the user, and are used as parameters for the given programming skeleton. This idea has been explored mostly in the context of sequential program development, but can also be applied to distributed computing [2,4,18,20,22].

One of the main limitations of programming skeletons is that they are not easily portable, but need to be re-implemented for each new architecture. This makes the approach unsuitable for exploiting available clusters of workstations and PCs, since their numbers, their individual characteristics, and their network topology are known only at runtime and may even change dynamically with the changing availability of individual nodes and/or links.

To address these problems and make paradigm-based computing feasible in dynamic heterogeneous computing environments, we exploit the benefits of a mobile agents infrastructure. The autonomous migration ability of agents makes them capable of utilizing a dynamically changing network. Their inherent portability allows them to handle

* Corresponding author.

E-mail addresses: hkuang@ics.uci.edu (H. Kuang), bic@ics.uci.edu (L.F. Bic), dillenco@ics.uci.edu (M.B. Dillencourt).

the distribution of tasks in a heterogeneous environment in a transparent manner. Agent mobility can also be exploited for load balancing. Because of these features, mobile agents lend themselves naturally to paradigm-oriented distributed computing.

Using a mobile-agent system called MESSENGERS [5,6], which was developed by our group in previous research efforts, we have developed an environment for paradigm-oriented distributed computing, called PODC. This environment currently supports five common computing paradigms: (1) bag-of-tasks, (2) branch-and-bound, (3) genetic programming, (4) finite difference, and (5) individual-based simulation. After presenting these paradigms in detail in Section 2, Section 3 illustrates the user's point of view when employing the paradigms and presents the overall system architecture. Section 4 presents the underlying implementation using MESSENGERS. Performance evaluations are discussed in Section 5, followed by general conclusions.

2. Paradigms

2.1. Choice of paradigms

Currently PODC supports five paradigms: bag-of-tasks, branch-and-bound search, genetic programming, finite difference, and individual-based simulation. There are three main reasons for choosing these paradigms: (1) many applications that fit those paradigms are highly computationally intensive and thus can benefit from multiple computers to improve their performance through parallelism; (2) an application following these paradigms can easily be divided into large numbers of coarse-grained task and (3) there is limited communication among the tasks. These three properties make the chosen paradigms suitable for execution in a network environment, where the cost of communication is high, and must be offset by large numbers of coarse-grained tasks.

We have investigated three categories of paradigms, classified by their inter-task communications. The first category, which requires no communication at all between running tasks, is typified by the bag-of-tasks paradigm in which tasks are highly independent and do not share any information. The second category requires only *non-essential* communications, which do not affect the correctness although they may have great impact on the performance. The paradigms in this category are branch-and-bound and genetic programming. In these paradigms, the tasks are highly asynchronous and self-contained, and they exchange information only for the purpose of optimization. The third category requires *essential* communications, meaning that they are necessary for correctness. Examples of this type of paradigms are iterative grid-based paradigms typified by the finite difference and the individual-based simulation, in which near-neighbor communications must be synchronized at each time step. The difficulty of providing a distributed implementation with

```

1. BagOfTasks(){
2.   D = InitData();
3.   Until ( Termination_condition ){
4.     T = GenNextTask( D );
5.     add T to the bag of tasks BT;
6.   }
7.   while (BT is not empty) {
8.     T = remove a task from BT;
9.     R = Compute ( T );
10.    Write Result ( outFile, R );
11.  }}

```

Fig. 1. The bag-of-tasks paradigm specification.

reasonable performance increases with each type of communications.

To make the paradigm-oriented approach to distributed computing practical, we must be able to clearly differentiate between application-specific program components, which must be provided by the user, and paradigm-specific components, which are provided by the system. In the remainder of this section, we describe each paradigm in pseudo-code that makes it clear which application-specific components are required. The identifiers in bold face indicate the application-specific functions, while the identifiers in *italic* indicate the variables whose type information are required.

2.2. Bag-of-tasks paradigm

The bag-of-tasks paradigm applies to the situation when the same function is to be executed a large number of times for a range of different parameters. Applying the function to a set of parameters constitutes a task, and the collection of all tasks to be solved is called the bag of tasks.

Fig. 1 shows the structure of the paradigm. The problem data and task-generation state are initialized at line 2, and the bag of tasks (BT) is created by the loop at lines 3–6. The termination condition either represents a fixed number of iterations or is given implicitly by reading input values from a file until the end of file. The while loop at lines 7–11 represents the actual computation, which is repeated until the bag of tasks is empty. Multiple workers may execute the loop independently. All workers have shared access to the task bag and the output data. Each worker repeatedly removes a task (line 8), solves it by applying the main function **Compute** to it (line 9), and writes the result into a file (line 10).

2.3. Branch-and-bound paradigm

Branch-and-bound is applicable to various combinatorial optimization problems, and is generally applied when the goal is to find the exact optimum. The branch-and-bound

```

1. BranchAndBound(){
2.   B = +∞;
3.   D = GenProblemData();
4.   R = GenRootNode( D );
5.   if ( an improve dinitial bound is to be used ) {
6.     S = GenInitSol( D );
7.     B = GenBound( S,D );
8.   }
9.   L = { R };
10.  while( L is not empty ) {
11.    N = SelectNode( L );
12.    SN = NextBranch( N, D );
13.    if( SN is Nil ) L = L - {N};
14.    else {
15.      SB = GenBound( SN,D );
16.      if ( !IsSol( SN ) )
17.        if ( SB < B ) L = L + {SN};
18.      else if( SB < B ) {
19.        B = SB;
20.        S = SN;
21.      } }
22.  WriteSol( S );
23. }

```

Fig. 2. The branch-and-bound paradigm specification.

paradigm dynamically constructs and searches a tree, in which the root node represents the original problem to be solved, each descendant node represents a subproblem of the original problem, and each leaf represents a feasible solution. To speed up the search, a subtree is pruned if it can be determined that it will not yield a solution better than the best currently known solution.

Fig. 2 presents the basic structure of the branch-and-bound paradigm. Without loss of generality, we assume the goal is to find the minimum value. We assume D is the initial problem data, which is passed as a parameter to certain functions; R is the initial root node; L is a pool of tree nodes that have yet to be explored; S is the current best solution; and B is the bounding value corresponding to S .

The algorithm starts by setting the bound B to infinity (line 2), initializing the problem data (line 3), and generating the root node R of the branch-and-bound tree (line 4). Lines 5–8 are optional: the starting bound B may be improved by generating an initial feasible solution S and the corresponding bound B . The pool L of problems to be solved is initially set to R (line 9). Lines 10–21 constitute the main loop. A node N is selected from the pool L (line 11) and one of the N 's subnodes is generated (line 12). If all subnodes of N have been explored, then the node N has been completely solved and so is removed from the list (line 13). Otherwise,

```

1. Genetic() {
2.   D = GenProblemData();
3.   P = GenInitPop( S, D );
4.   until ( Termination_condition )
5.     P = CreateNextGen( P ,D );
6.     I = BestIndividual( P );
7.     WriteSol( I );
8. }

```

Fig. 3. The genetic programming paradigm specification.

a new bound is computed for the subproblem. Lines 16–21 then accomplish the pruning. After the pool of subproblems to be solved is drained, the solution to the problem is written (line 22) and the program terminates.

2.4. Genetic programming paradigm

The genetic programming paradigm also solves optimization problems, using the Darwinian principles of survival and reproduction of the fittest and genetic inheritance [8]. Unlike branch-and-bound, genetic programming is generally applied to find a good but not necessarily optimal solution. Fig. 3 shows the basic structure of a sequential genetic programming paradigm. The static problem data is generated and an initial population P of size S (lines 2 and 3) is created. The while loop on lines 4 and 5 represents the evolution process. At each iteration, a new generation is created by applying genetic operations such as crossover, mutation, and reproduction. The termination condition at line 4 is typically based either on the number of iterations completed or the quality of the best solution obtained.

In our distributed version of the genetic programming paradigm, the population is divided into multiple subpopulations that evolve independently and occasionally exchange individuals. This scheme requires specifying two other components: **SelectEmigrant**, which selects an emigrant from a population; and **ProcessImmigrant**, which decides whether an arriving immigrant should be discarded or kept and, in the latter case, selects the individual to be replaced by the immigrant. Several controlling parameters are also required: the number of generations to be created between sending out a wave of emigrants (*EmigrationInterval*) and the number of emigrants in a wave (*EmigrationRate*).

2.5. Finite difference paradigm

The finite difference method [23] is used to solve differential equations. The method uses a discrete d -dimensional grid of element locations and computes a value $u_{x,t}$ at each grid location x and for a regular sequence of times t . The value of $u_{x,t+\Delta t}$ is a function of $u_{x,t}$ and the values of u at the neighbors of x at time t .

```

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.   oldVals = vals1; newVals = vals2;
9.   // iterative computation
10.  until ( Terminatio_ncondition ) {
11.    for (every element ⟨X, Y⟩ )
12.      newVals[X][Y] = Compute( X, Y, oldVals, ΔX, ΔY, Δt );
13.    newVals ↔ oldVals;
14.  }
15.  // output
16.  for ( every element ⟨X, Y⟩ )
17.    WriteResult( outFile, X, Y, oldVals );
18. }

```

Fig. 4. The finite difference paradigm specification.

Fig. 4 shows the structure of the finite difference paradigm for a 2D problem. The algorithm starts by initializing the values of all elements (lines 6 and 7) and pointers to element buffers (line 8). It then repeatedly computes the new values of all elements for each time step until the termination condition is satisfied (lines 10–14). Finally, element values are written to the output file (lines 16 and 17).

2.6. Individual-based simulation paradigm

Individual-based simulation programs are used to simulate the behaviors of a collection of entities such as the movement of particles [9], the schooling behavior of fish or birds [12,21], and the evolution of an ecology environment [10]. The simulated entities move in a specified space over a period of time. At each time step, an entity decides its 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 of it.

Variants of the 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. Our individual-based simulation paradigm supports two frameworks of collision detection and resolution: *delayed state update* and *immediate state update*.

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. Thus, the simulation program needs

to keep two sets of states, the old state and the current state. Because the structure of individual-based simulation programs using the delayed state update method is very similar to that of finite different programs, 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.

Fig. 5 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 are initialized (lines 8–16). The main loop (lines 16–29) simulates the behavior of entities for a fixed number of iterations. At each iteration, all the entities update their states separately (lines 20–25). 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 26–29. Notice that the environment may get updated in two places. The updates on line 23 are caused by entities. The updates on lines 26–29 reflect spontaneous changes to the environment.

3. PODC system

PODC is a graphics-based environment that supports paradigm-oriented distributed programming. To develop an application and execute it in a distributed manner, the user must first choose one of the paradigms supported by the system, and develop the necessary application-specific functions that perform the corresponding domain computations. These functions are all written using conventional unrestricted C, but must obey certain conventions regarding input and output. The user is then guided through a series of form-based queries, as illustrated in Section 3.1, during which s/he supplies the necessary information. Using this information the system automatically generates and compiles distributed code, starts the MESSENGERS system on the set of available nodes, and monitors the execution of the application. In this section, we first present how users interact with PODC, then we show the architecture of the PODC system.

3.1. Problem specification and the user interface

Users interact with PODC through a graphical interface. There are two phases to the interaction. In the first phase, the

```

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<NUMOFENTITIES; i++ ) {
14.    new Entity(&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, entityGroup, R);
23.      ⟨entityList, env⟩ = updateEntity ( t, curEntity, env, entityList, ΔX, ΔY, Δt);
24.      add each entity in entityList to entityGroup;
25.     }}
26.    for ( each grid cell ⟨i,j⟩ ) {
27.     entityList = updateEnv( t,&env[i][j], i, j, ΔX, ΔY, Δt);
28.     add each entity in entity List to entityGroup;
29.    }} }

```

Fig. 5. The individual-based simulation paradigm specification.

user specifies and starts the application through a *submission window*. In this window, the user specifies application-specific program components. In the second phase, the user can monitor and interact with the running application through a *feedback window*, which shows the current status of the system. The details of the submission and feedback windows depend on the specific paradigm. In this section, we illustrate the process of specification and interaction in the context of the branch-and-bound paradigm.

Fig. 6 shows the graphic interface used to specify a branch-and-bound problem. At the top of the window, the user provides the location of the sequential source programs, header file(s), program file(s), and the direction of optimization. Next, the user specifies the application-specific functions and data structures, as identified in bold font in Section 2.3. The procedure to generate an initial feasible solution is optional; if it is not provided, the pruning bound is set to a default value of $+\infty$ for a minimization problem or $-\infty$ for a maximization problem. Finally, the user can

specify a recommended number of machines to use in the computation.

Fig. 7 shows the feedback window of a branch-and-bound application. An icon at the very top indicates that the application is still running. Below that a graph shows the history of generated optimal solutions: the x -coordinate represents elapsed time, and the y -coordinate represents the best solution achieved at the corresponding point in time. The feedback window also displays the number of machines currently being used by the distributed application, a bar showing the estimated fraction of the total work that has been currently completed (computed by estimating the number of leaf nodes in the branch-and-bound tree that have currently either been visited or pruned away), and error/status messages.

3.2. System architecture

PODC has a 3-tier client/server architecture. The top level is the client, which interacts with the users through a

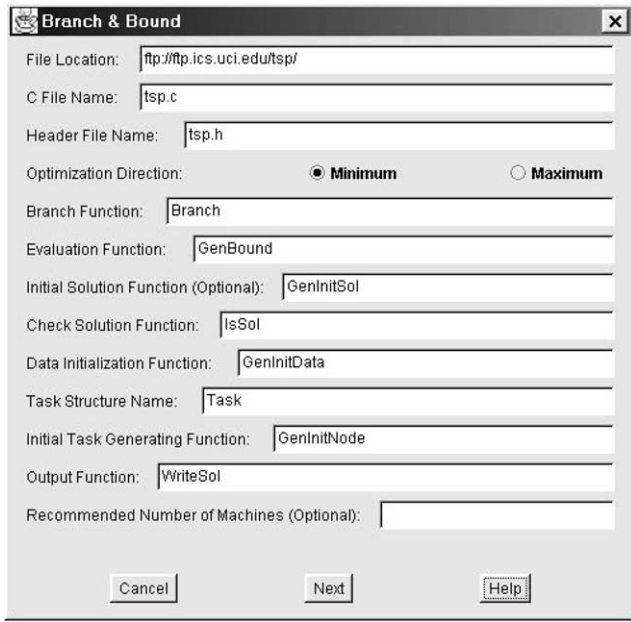


Fig. 6. Submission window for the branch-and-bound paradigm.

graphics interface and permits the submission and monitoring of applications. The middle level is the server, which builds and supervises the execution of the application.

The third level represents the underlying MESSENGERS system, in which the application is implemented as multiple autonomous mobile agents, running on a given computer network [14].

Fig. 8 illustrates the different stages of developing and running an application in PODC. Fig. 8(a) shows the submission of an application through a submission window in the client. By selecting a paradigm from a list, the user implicitly specifies the parallel algorithm for the generated distributed program. The problem-specific information is then used to instantiate the program skeleton.

When the user instructs the system to start the application, the server transparently chooses a set of workstations and automatically generates and compiles the distributed application codes. The system then starts up the MESSENGERS system on the set of workstations and begins to execute the distributed programs. The application is implemented using multiple Messengers, each of which hops within a network of workstations where it works on the task. The underlying MESSENGERS system performs the necessary distribution, load balancing, and fault-recovery automatically and transparently [7]. During execution of the distributed application, the underlying network might change, for example, as a results of failure or a change in the system workload. The MESSENGERS can seamlessly adapt to such changes.

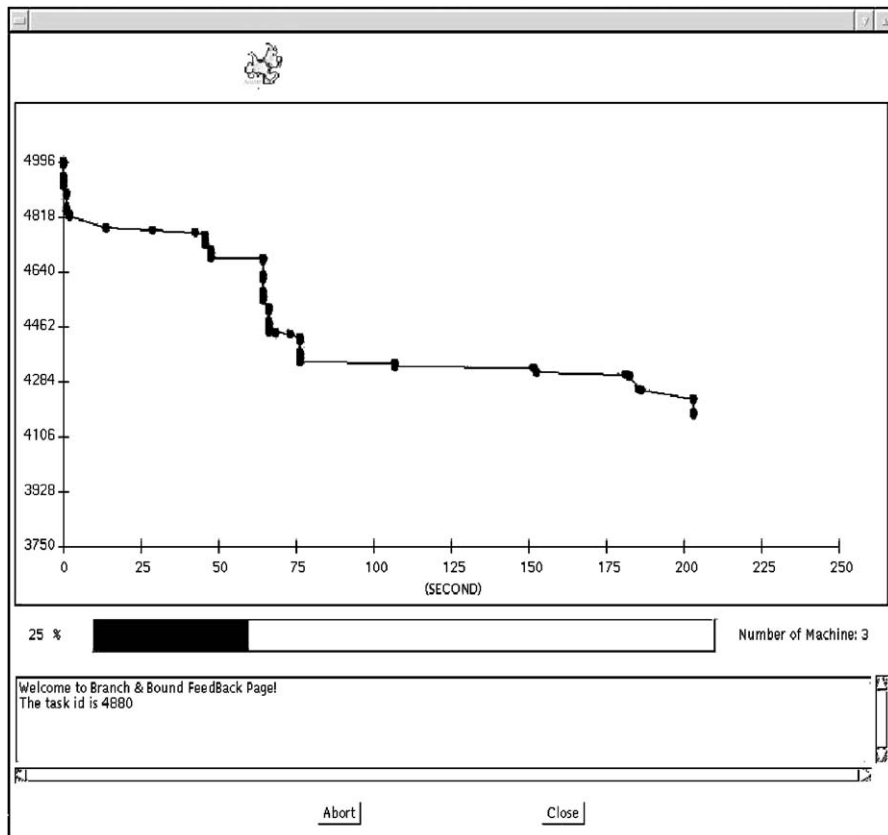


Fig. 7. Feedback window for the branch-and-bound paradigm.

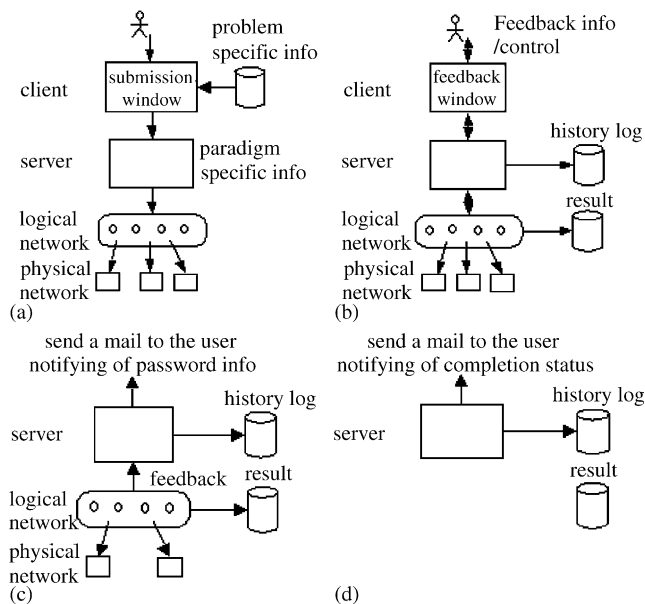


Fig. 8. System architecture.

Fig. 8(b) shows the system while the application is running. During this time, the user is able to monitor the progress using the feedback window. The feedback information is generated by the distributed application and sent to the server, which passes it to the client and to be displayed on the feedback window. The server also saves this information in a history log for later processing.

The user is not required to remain on-line while the application is running. Fig. 8(c) shows the situation where the system continues operating while the user is off-line. During this time, the server continues receiving feedback information from the distributed application and recording it in the history log. A user can later reconnect to the server and check the status of the running application; recreating the on-line monitoring situation shown in Fig. 8(b).

When the application terminates, the MESSENGERS system is shut down and the user is sent a notification via e-mails as illustrated in Fig. 8(d). At that time, the user is able to retrieve the results through a Web browser or ftp tool.

4. Distributed implementation of paradigms

4.1. Bag-of-tasks paradigm

The logical network used to implement the bag-of-tasks paradigm using MESSENGERS is a star topology as shown in Fig. 9(a). The “Meeting Room” node is a central node where the bag of tasks is stored. The “Office” nodes are where workers solves tasks. Since each task can be executed independently, no information needs to be exchanged between workers, and therefore no links exist between office nodes.

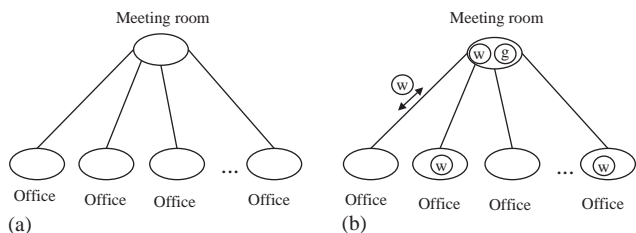


Fig. 9. Logical network for the bag-of-tasks paradigm.

Because the number of tasks may be large, it is useful to allow the tasks to be generated “on-the-fly,” rather than explicitly at the beginning. A task-generation messenger, denoted as *g* in Fig. 9(b), exists for this purpose and stays at the central node. When the application starts, the task-generation messenger generates an initial number of tasks and injects multiple worker Messengers (*w*), each of which is assigned an office. Each of them hops back and forth between the meeting room and its office, as shown in Fig. 9(b). On each trip, it brings a new task to its office to work on. After it finishes executing the task, it carries the result back to the meeting room, and pulls another task from the bag of tasks.

When the number of tasks in the task pool falls below a certain threshold, the task-generation messenger generates additional new tasks. A worker terminates when the bag of tasks becomes empty. The task-generation messenger terminates when all the results have been written to the output file, thus effecting global termination.

4.2. Branch-and-bound paradigm

The logical network supporting the distributed implementation of the branch-and-bound paradigm is shown in Fig. 10(a). The “Meeting room” node is where an initial pool of tasks is stored, and the “Office” nodes are where workers explore a portion of the search space. When a worker finds a solution that is better than the best previously known solution, other workers are notified of the new pruning bound, and the office nodes are fully connected to facilitate this. This exchange of new pruning bounds is an example of *non-essential communication* as defined in Section 2: each task would successfully complete without this information exchange, but using improved pruning bounds discovered by other tasks can improve performance significantly.

Three types of Messengers, shown in Fig. 10(b), are used to implement the branch-and-bound paradigm. An initialization Messenger (*g*), which stays in the meeting room, generates the initial data—static problem data, the initial pruning bound, and the initial task pool (by a partial breadth-first expansion of the root node)—and also injects the worker messengers. Multiple worker messengers (*w*) exist in the system, one per office node. When a worker messenger finds a better solution, runner messengers (*r*) are created. These

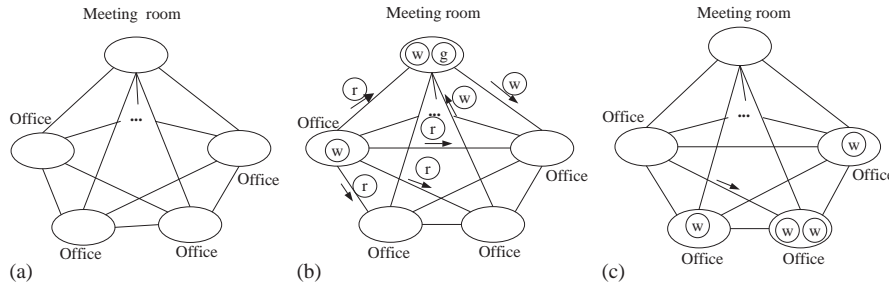


Fig. 10. Logical network for the branch-and-bound paradigm.

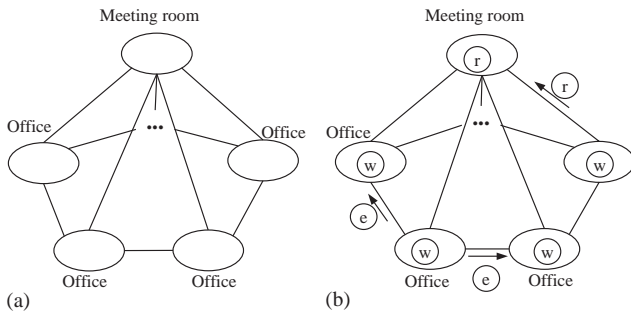


Fig. 11. Logical network for the genetic programming paradigm.

runner messengers hop to the other office nodes to update the pruning bound.

After a worker carries a node from the initial task pool to its office, it explores the subtree rooted at that node in depth-first order. Whenever it expands a node, it works on one of the children and places the remaining children in a local task pool. It repeatedly explores nodes in its local task pool. When the local task pool is exhausted, the worker messenger hops to the meeting room. If the initial task pool has not been drained, the worker carries one of the nodes from the pool back to its office and continues as before.

When a worker hops to the meeting room and finds that the initial task pool has been drained, it attempts to balance the system load by “stealing” work from a randomly chosen office of another worker. It hops to the chosen office as shown in Fig. 10(c). If the chosen office has a nonempty task pool, the worker “steals” one of the tasks by carrying it back to the worker’s own office and executing the stolen task. If the chosen office has an empty task pool, the worker randomly chooses another office to steal a task from. After a certain number of unsuccessful attempts (currently set as half of the number of active workers), the worker terminates. Global termination occurs when the last worker terminates.

4.3. Genetic programming paradigm

Fig. 11(a) illustrates a logical network for the implementation of the genetic programming paradigm in the

MESSENGERS system. The “Meeting room” node is where workers exchange global information, while an “Office” node is where a worker executes the evolution process with a distinct subpopulation pool.

Three types of messengers exist in the system, as illustrated in Fig. 11(b). The worker messengers (*w*) execute the basic genetic algorithm, one per office. Each worker is passed a different seed for its random number generator when it is injected. Each worker randomly generates an initial population pool and then repeatedly applies genetic operations to create subsequent generations until the termination condition is satisfied. When the best individual in the current generation is better than the previous best in the entire system, a runner (*r*) messenger sends it to the meeting room node and reports it to the user. When all the worker messengers finish the specified number of generations or one of them satisfies the termination predicate, the entire application is terminated.

Workers periodically exchange individuals with workers at neighboring nodes, as discussed in Section 2.4. They do this by generating exporter messengers (*e*) that carry emigrants to the neighbors. The frequency and rate of these exchanges are determined by the user-specified parameters *EmigrationInterval* and *EmigrationRate*, which may be modified at runtime through the feedback window. Increasing the emigration rate or decreasing the emigration interval causes more population mixing but increases the communication cost.

4.4. Finite difference paradigm

Fig. 12(a) shows a logical network to support the distributed implementation of the finite difference paradigm on a 2D grid. The “Office” nodes are where element values are updated. Each “Office” node is assigned a portion of the grid. At each iteration, new values are computed. Boundary values are exchanged between neighbors when boundary information becomes obsolete. The “Meeting room” node is where the termination condition is evaluated.

The implementation uses four types of messengers as shown in Fig. 12(b). The initialization messenger (*i*) builds the logical network and injects the worker messengers (*w*)

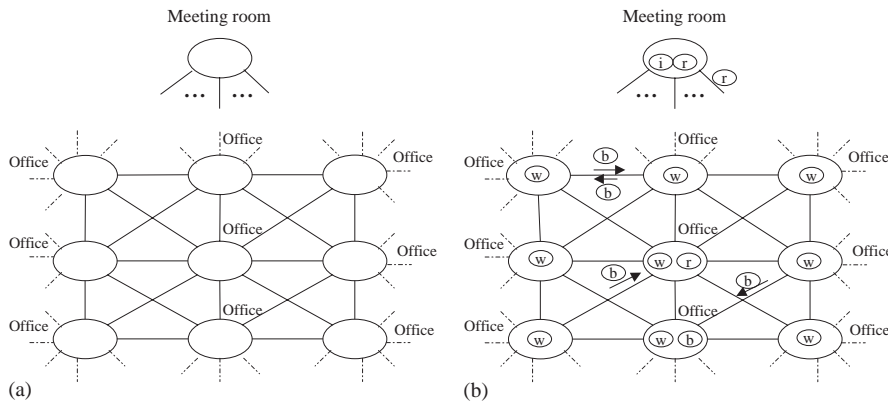


Fig. 12. Logical network for the finite difference paradigm in MESSENGERS System.

into offices, one per office. Each worker messenger initializes its partition and starts boundary messengers (*b*). The function values in each partition are repeatedly updated by worker and boundary messengers. At each iteration, the worker messenger updates the “inner” grid cells (i.e., the cells whose values do not depend directly on the values of grid cells in neighboring partitions). Each partition has eight boundary messengers, one per side and one per corner. Each boundary messenger shuttles between two neighboring “Office” nodes. At each iteration, a boundary messenger updates the grid cells on the portion of the boundary for which it is responsible. It then hops to the other “Office” node, carrying boundary data, and updates the neighboring office’s ghost boundary (i.e., a shadow of its neighbor’s boundary). At the next step, it works at the neighboring office, updates a portion of its boundary, and carries the boundary information back to the previous node. Each “Office” node also has a “report” messenger (*r*) that gathers information necessary to evaluate the global termination condition (e.g., the tolerance) and carries it to the “Meeting room”. When the termination condition is satisfied, the report messengers hop back to the “Office” nodes and set termination flags.

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 updates the inner part of the partition, interrupting itself periodically to give late incoming boundary messengers a chance to work. This eliminates the necessity of a barrier at each step, hence reducing the idle waiting time.

4.5. Individual-based simulation paradigm

The distributed implementation of the individual-based simulation paradigm is similar to that of the finite difference paradigm. Both use the same logical networks, the types of messengers, and the near neighbor boundary exchange at each time step. There are three important differences where the individual-based simulation paradigm requires additional mechanisms. 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 proceeds in two phases: first the emigrating entities move to the neighboring nodes, and then the boundary information is exchanged.

A second 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.

A third difference is the importance of *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 cor-

2	0	2	0	2	0	2	0	2	0	2	0
3	1	3	1	3	1	3	1	3	1	3	1
2	0	2	0	2	0	2	0	2	0	2	0
3	1	3	1	3	1	3	1	3	1	3	1
2	0	2	0	2	0	2	0	2	0	2	0
3	1	3	1	3	1	3	1	3	1	3	1
2	0	2	0	2	0	2	0	2	0	2	0
3	1	3	1	3	1	3	1	3	1	3	1
2	0	2	0	2	0	2	0	2	0	2	0
3	1	3	1	3	1	3	1	3	1	3	1
2	0	2	0	2	0	2	0	2	0	2	0
3	1	3	1	3	1	3	1	3	1	3	1

Fig. 13. Distributed odd–even individual-based simulation paradigm.

responding 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 [15] 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, illustrated in Fig. 13, ensures that entities are processed in the same order, irrespective of the partitioning of the user grid. We label each user grid cell with its index (x, y) , and then label the cell with a number in the range 0–3, computed as $(x \bmod 2) + (y \bmod 2)$. At each time step, we update the states of all entities located at user grid cells with a particular label value (starting with 0) before proceeding to the next label value.

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 Fig. 13. 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. As this example illustrates, our odd–even scheme requires expanding the size of the exchanged boundary by a factor of 4.

The odd–even labeling scheme represents a graph-theoretical coloring of the user grid: two cells whose boundaries share either an edge or at a corner are assigned different labels. If two entities are located in two different grid cells that are assigned the same label, the distance between

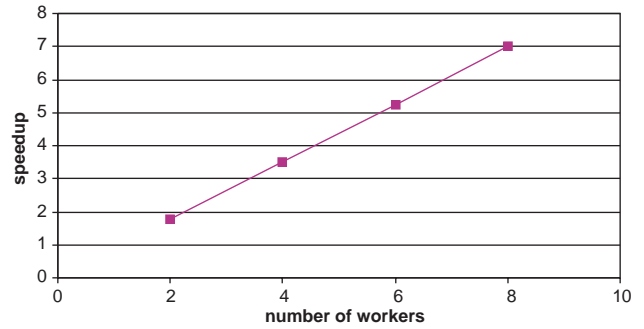


Fig. 14. Speedup for the bag-of-tasks paradigm experiments.

the entities is at least as large as the length of the shortest side of a grid cell. A typical individual-based system model includes 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 . 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.

5. Performance evaluation

5.1. Bag-of-tasks paradigm

Our experiments with the bag-of-tasks paradigm were based on a Monte Carlo simulation of a model of light transport in organic tissue [19]. The sequential program was provided by the Beckman Laser Institute and Medical Clinic at UC Irvine. Because the model assumes that the movement of each photon in the tissue is independent of all other photons, this simulation fits well in the bag of tasks paradigm. The number of photons simulated is 1,000,000, and each task simulates the movements of 1000 photons. The experimental results are shown in Fig. 14. The graph presents a near-linear speedup.

5.2. Branch-and-bound paradigm

We tested the branch and bound paradigm on a well-known combinatorial problem, the Traveling Salesman Problem (TSP). In the sequential TSP program, the bounding rule is based on a minimum spanning tree [11] of the unvisited cities in a partial tour. We used data for 24 cities.



Fig. 15. Speedup for the branch-and-bound paradigm experiments.

Table 1
Summary statistics for Branch-and-bound execution times with different search order permutations

	Mean	Max	Min	StdDev
Sequential	1590.1	4877.6	241.1	1555.4
MESSENGERS—3 workers	293.7	1217.4	59.1	341.8
MESSENGERS—6 workers	93.0	236.6	34.6	59.6

During our experiments, we observed nondeterministic performance behavior of the distributed branch-and-bound program [16]. Therefore, we executed both the sequential and distributed programs ten times, each with different input data. The experiment results in Fig. 15 represent the average speedup from 10 runs. These show a near-linear speedup for the distributed branch-and-bound program.

The nondeterministic behavior of the distributed branch-and-bound search is partly due to the fact that the distributed branch-and-bound program implicitly changes the search order by exploring the problem space concurrently. To investigate how the search order effects the performance, we conducted another type of experiments. We ran both sequential and distributed programs ten times, each time with the same input data, but with a different permutation of the initial task pool. The results of this experiment, which are summarized in Table 1, show that the program execution time is greatly influenced by the search order. For example, the maximum and minimum execution times for the sequential program are, respectively, 4877.6 and 241.1 s, a ratio of more than 20. The table also shows that introducing multiple workers working simultaneously has a significant smoothing effect on the execution time, making the execution time more predictable.

5.3. Genetic programming paradigm

We tested the genetic programming paradigm also using the traveling salesman problem. In this case we used a 30-city problem. The original sequential program is due to Lalena [17]. We restructured the source code to fit into our paradigm and added a mutation operator. The distributed programs were run on a network of four workstations. Be-

cause genetic programs are probabilistic, we ran each program 10 times.

To ensure a fair performance comparison, we defined the *quality ratio* of a particular computed solution to a minimization problem as $\frac{\text{fitness}(\text{optimal solution})}{\text{fitness}(\text{computed solution})}$, where the fitness function evaluates a given solution. Our sequential program attained a quality ratio of 99.4% after running for an indefinite period of time. We then used this ratio as the termination condition for the distributed programs, so that all solutions would yield the same quality of solution.

We ran the genetic algorithm for a sequential program with a population size of 1250, for a sequential program with a population size of 5000, and for a distributed (MESSENGERS) program with a population size of 1250 per worker (i.e., a total population of 5000). The respective average running times were 442.3, 1526.5, and 106.6 s. Thus the MESSENGERS program found the solution fastest and was four times as fast as the sequential program with a population size of 1250.

We also selected a representative run of three programs and illustrate the evolution process in Fig. 16. The square line is the sequential program with population size 1250. The triangle line is the sequential program with population size 5000 whose population includes that of the first sequential program. The diamond line is the MESSENGERS program with the same initial population as the first sequential one, but which exchanges immigrants periodically during the evolution.

The graph shows that initially the genetic program with the smaller population size can find good solutions the fastest. However, after it finds a solution with a quality ratio 97.1%, it takes much longer to improve this solution. The sequential program with the larger population performs better after this point. The MESSENGERS program combines the benefits of the two sequential programs. Initially it behaves as the sequential program with the smaller population. Later it converges slightly slower because of the greater diversity of individuals resulting from the exchange of immigrants. In the final stage, it beats both sequential programs and finds the near optimal solution with the quality ratio of 99.4% in the shortest time.

5.4. Finite difference paradigm

We tested the finite difference paradigm using Metropolis Monte Carlo algorithm, which solves the Ising model [1,13]. In our experiments, the simulated space is a 2D toroidal grid. We varied the grid size, and also tried two different partition strategies: strip partition (i.e., the user grid is partitioned into strips) and rectangular partition (i.e., the user grid is partitioned into rectangular blocks).

Fig. 17 shows the speedup of the distributed finite difference programs using 9 machines. The speedup of the distributed programs increases with the computation-to-communication ratio. Also, programs with a rectangular

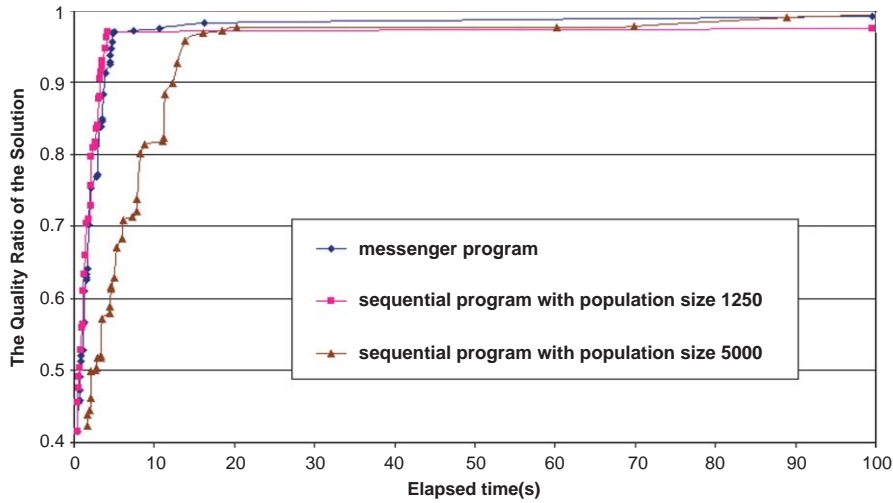


Fig. 16. Representative evolution processes of the genetic programs.

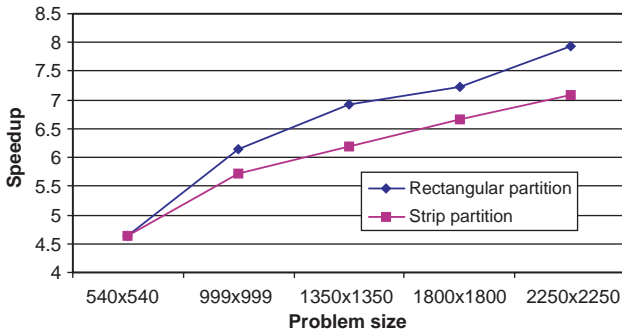


Fig. 17. Speedup for the finite difference paradigm experiments.

partition have consistently better speedup than those with a strip partition. This is because programs with a rectangular partition have a smaller boundary and thus smaller amounts of communication data than programs with a strip partition.

5.5. Individual-based simulation paradigm

We tested the individual-based simulation paradigm using a modification of the fish schooling model described in [12]. This model assumes a 2D 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. The simulation space is a 2D 300×300 toroid in which fish move as a single school of fish for 500 simulation steps. We ran each program three times. The execution times presented are the averages of the three runs. Fig. 18 shows the speedup for increasing the numbers of simulated fish. The figure shows that the speedup increases with the problem size. This is due to the increasing computation-to-communication ratio. We can also see that the delayed state update program has a better

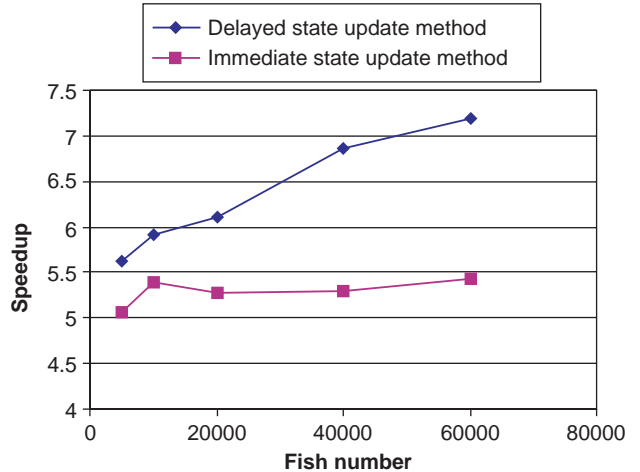


Fig. 18. Speedup for the individual-based simulation paradigm experiments.

speedup than the immediate state update program. This is because each time the boundary information is exchanged, the immediate state update program sends and receives messages four times the size of the delayed state update program. The immediate state update program also needs to duplicate redundant computations.

6. Conclusions

In this paper we presented an approach to distributed computing that uses the concept of well-known paradigms. Its main features, which differentiate it from other approaches, are the following: (1) It is intended for loosely-coupled network environments, not specialized multiprocessors; (2) it is based on an infrastructure of mobile agents; (3) it supports programming in C, rather than a functional or special-

purpose language, and (4) it provides an interactive graphics interface through which programs are submitted, invoked, and monitored.

By implementing five widely used paradigms—bag-of-tasks, branch-and-bound, genetic programming, finite difference, and individual-based simulation—we have demonstrated the viability of this approach for use in heterogeneous and dynamically changing clusters of commodity workstations or PCs. One of the main reasons for the flexibility and portability of the PODC environment is the use of mobile agents, which provide a virtual environment within which the given paradigms can be implemented independently of any specific networking or architectural constraints. The performance tests indicate that, for the chosen paradigms, the resulting overhead is minimal, allowing the system to deliver nearly linear speedup for many types of applications.

Although our system requires the users to partially restructure their sequential programs to adapt them to the PODC environment, our experience shows that the effort required is very small. The benefit is a significant improvement in performance due to the achieved parallelism.

References

- [1] K. Binder, D.W. Heermann, Monte Carlo Simulation in Statistical Physics, Springer, Berlin, 1988.
- [2] G.H. Botorog, H. Kuchen, Skil: an imperative language with algorithmic skeletons for efficient distributed programming, in: Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5), IEEE Computer Society Press, Silver Spring, MD, 1996, pp. 243–252.
- [3] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
- [4] J. Darlington, A.J. Field, P.G. Harrison, Parallel programming using skeleton functions, in: PARLE'93, Parallel Architectures and Languages Europe, June 1993.
- [5] M. Fukuda, L.F. Bic, M.B. Dillencourt, Messages versus messengers in distributed programming, J. Parallel Distributed Comput. 57 (1999) 188–211.
- [6] M. Fukuda, L.F. Bic, M.B. Dillencourt, F. Merchant, Distributed coordination with messengers, Sci. Comput. Programming 31 (2) (1998).
- [7] E. Gendelman, L.F. Bic, M.B. Dillencourt, An application-transparent, platform-independent approach to rollback-recovery for mobile agent systems, Technical Report 35, University of California, Irvine, 1999.
- [8] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.
- [9] L. Greengard, V. Rokhlin, A fast algorithm for particle simulation, J. Comput. Phys. 73 (1987) 325–348.
- [10] G. Hartvigsen, S.A. Levin, Evolution and spatial structure interact to influence plant-herbivore population and community dynamics, Proc. Roy. Soc. London Ser. B 264 (1997) 1677–1685.
- [11] M. Held, R.M. Karp, The traveling-salesman problem and minimum spanning trees, Oper. Res. 18 (1970) 1138–1162.
- [12] A. Huth, C. Wissel, The simulation of the movement of fish schools, J. Theoret. Biol. 156 (1992) 365–385.
- [13] M.H. Kalos, P.A. Whitlock, Monte Carlo Methods, vol. I, Basics, Wiley, New York, 1986.
- [14] H. Kuang, L.F. Bic, M.B. Dillencourt, Paradigm-oriented distributed computing using mobile agents, Technical Report 38, University of California at Irvine, 1999.
- [15] H. Kuang, L.F. Bic, M.B. Dillencourt, Repeatability, programmability, and performance of iterative grid-based computing, Technical Report, Information and Computer Science, University of California, Irvine, 2001.
- [16] T.H. Lai, S. Sahni, Anomalies in parallel branch-and-bound algorithms, Comm. Assoc. Comput. Mach. 27 (9) (June 1984) 594–602.
- [17] M. LaLena, Travelling salesman problem using genetic algorithms, 1996, <http://www.lalena.com/ai/tsp/>.
- [18] S. Pelagatti, A Methodology for the Development and the Support of Massively Parallel Programs, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, March 1993.
- [19] S.A. Prahl, M. Keijzer, S.L. Jacques, A.J. Welch, A Monte Carlo model of light propagation in tissue, in: Dosimetry of Laser Radiation in Medicine and Biology, SPIE Institutes for Advanced Optical Technologies Series, vol. IS 5, SPIE Optical Engineering Press, 1989, pp. 102–111.
- [20] F.A. Rabhi, A parallel programming methodology based on paradigms, in: Transputer and Occam Developments, IOS Press, 1995, pp. 239–252.
- [21] C.W. Reynolds, Flocks, herds, and schools: a distributed behavioral model, Comput. Graphics 21 (4) (July 1987) 25–34.
- [22] S. Siu, M. De Simone, D. Goswami, A. Singh, Design patterns for parallel programming, in: PDPTA'96, August 1996.
- [23] E.F. Van de Velde, Concurrent Scientific Computing, Springer, Berlin, 1994.

Hairong Kuang received her M.S. degree in Information and Computer Science from the University of California, Irvine, in 1999 and her Ph.D. in Information and Computer Science from the same university in 2002. She currently works as an Assistant Professor of Computer Science at the California State Polytechnic University, Pomona. Her research interests are parallel and distributed computing.

Lubomir F. Bic received his M.S. degree in Computer Science from the Technical University Darmstadt, Germany, in 1976 and his Ph.D. in Information and Computer Science from the University of California, Irvine, in 1979. He is currently Professor and Co-Chair of the Computer Science Department at the University of California, Irvine. His primary research interests lie in the areas of parallel and distributed computing. Currently he is co-directing the Messengers Project, which explores the use of self-migrating threads to simplify the programming of computationally intensive applications and to improve their performance in distributed computer environments.

Michael B. Dillencourt is an Associate Professor of Computer Science at the University of California, Irvine. He holds an MA degree in Mathematics (University of Wisconsin, 1975), an M.S. degree in Computer Science (University of Wisconsin, 1976), and a Ph.D. in Computer Science (University of Maryland, 1988). His primary research interests are distributed computing and algorithm design and analysis.