ICS

TECHNICAL REPORT

Paradigm-Oriented Distributed Computing Using Mobile Agents

Hairong Kuang
Lubomir F. Bic
Michael B. Dillencourt

UCI-ICS Technical Report No. 99-38
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

September 20, 1999

Information and Computer Science
University of California, Irvine

Paradigm-Oriented Distributed 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 distributed computing that uses the concept of well-known paradigms. 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 main features of the proposed approach, called PODC, 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 a Web-based interactive graphics interface through which programs are constructed, invoked, and monitored.

The three paradigms presently supported in PODC are the bag-of-tasks, the branch-and-bound, and genetic programming. We demonstrate their use, implemetation, and performance within the mobile agent-based PODC environment.

Keywords: *Programming Skeletons, Paradigm-Oriented Computing, Distributed Computing, Mobile Agents, a Bag of Task, Branch and Bound, Genetic Programming*

Technical areas: *Distributed Programming Environment, Agent-based Computing*

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 application is significantly more difficult than developing sequential applications. Besides solving the computational problems, the user must also deal with the partitioning and distribution of data, structuring the program as a collection of concurrent activities, and coordinating their execution. In addition, issues of fault-tolerance, resource management, and load balancing must be considered. As a result, only expert programmers are able to take advantage of the widely available networks and clusters of computers.

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 program. One of the pioneering ideas was to provide programming skeletons [Col89], 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.

Paradigm-oriented distributed computing has a number of advantages over unrestricted distributed programming. First, the domain knowledge is separated from issues of distribution and parallelization. The user, who is the domain specialists, only needs to supply application-specific sequential functions, which are used as building blocks for the distributed application. The parallelism is implied by the structure of the paradigm and is exploited automatically. This greatly simplifies the programming task, reduces program development cost, and eliminates the sources of potential programming errors. Second, since all instantiations of a paradigm share the same structure, it is not necessary to fine-tune each application separately. Instead, investing the time and effort into optimizing the underlying common structures for best performance automatically benefits all applications that are based on the same paradigm.

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 would make 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 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 [FBD99, FBDM98], 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 three common computing paradigms: (1) bag-of-tasks, (2) branch-and-bound search, and (3) genetic programming. After presenting the overall system architecture in Section 2, the three paradigms will be presented in detail in Section 3. Section 4 illustrates the user's point of view when employing the paradigms, while Section 5 presents the underlying implementation using MESSENGERS. Performance evaluations are discussed in Section 6, followed by general conclusions.

2 System Architecture

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, 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.

PODC has a 3-tier client/server architecture. The top level is the client, which interacts with the users through a 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, which runs the application on a given computer network. The 3-tier architecture makes it possible to provide the user a Web-based interactive graphic interface. It also allows the user to be off-line while the application is running and to retrieve the running status and the results later.

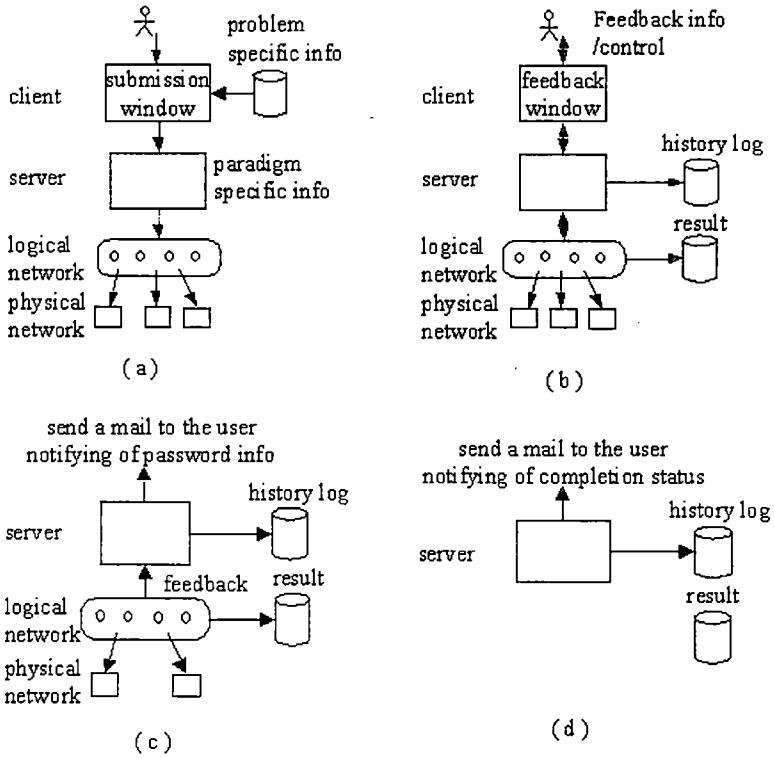


Figure 1: System Architecture

Figure 1 illustrates the different stages of developing and running an application in PODC. Assuming the user already has the necessary sequential functions for the application and has stored them as files, Figure 1(a) shows the submission of an application through a special submission window in the client. First, the user chooses the desired paradigm from a list. By selecting a paradigm, the user implicitly specifies the parallel algorithm for the generated distributed program. The user's email address is also required, to identify and notify the user. Then the user provides problem specific information by specifying the relevant file names and other information relevant to the selected paradigm. This information works as a road map for identifying the actual parameters and 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 structured as a collection of multiple tasks, each of which is implemented as a Messenger, which hops to its own workstation where it works on the task. The underlying MESSENGERS system performs the necessary distribution, load balancing, and fault-recovery automatically and transparently. 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.

Figure 1(b) shows the system while the application is running. During this time, the user is able to monitor the progress using a graphics display, which continuously provides specific information about the amount of work performed thus far, any errors encountered, partial solutions, and other useful statistics. The feedback information is generated by the distributed application and sent to the server, which is then passed to the client and 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. This is particularly useful with long running applications. If the user leaves the system, a password is issued, implemented as a unique application identifier. Figure 1(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 who chooses to reconnect to the server and check the status of the running application later can do so by presenting the password together with the user's identifier (the user's email address). This recreates the on-line monitoring situation shown in Figure 1(b).

When the application terminates, the MESSENGERS system is shut down and the user is sent a notification via email, as illustrated in Figure 1(d). At that time, the user is able to retrieve the results through a Web browser or ftp tool.

3 Overview of the Paradigms

Currently PODC supports three paradigms: bag-of-tasks, branch-and-bound search, and genetic programming. There are three main reasons for choosing these paradigms. First, many applications that fit those paradigms are highly computationally intensive and thus can benefit from multiple computers to improve their performance through parallelism. Second, an application following these paradigms can easily be divided into large numbers of coarse grained tasks. Third, each of those tasks is highly asynchronous and self-contained, and there is limited communication among the tasks. Each task receives information from other tasks at the time of its creation in the form of parameters and it passes its results to other tasks at the time of its termination. Any other information exchanges among running tasks are *non-essential* in the sense that they do not affect the correctness, although they may have significant impact on the performance. 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 independent coarse-grain tasks.

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 application-specific components must be expressed in the form of sequential functions. However, certain conventions concerning input and output must be obeyed. In particular, the main compute functions are not allowed to use any read/write commands. The reason is that these functions will be performed on different nodes or may even be moved at runtime. Thus any open files or other I/O connections would become meaningless. To solve the problem, each paradigm requires the specification of special functions for input and for output; these functions can use unrestricted read/write commands supported by C/Unix. The communication between these input/output functions and the compute functions is restricted to passing a single data structure using a reference. That is, the input function may read any data; it packages it into a data structure defined by the user; and it returns a pointer to the data structure when it terminates. This data structure then may be passed to a compute function by reference. When the compute function terminates, it returns its results in an output data structure, also defined by the user. This is then passed to an output function by reference, which writes it into a file.

3.1 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, since they do not need to be solved in any particular order. At each iteration, a worker grabs one task from the bag and computes the result.

The bag-of-tasks paradigm has three characteristics. First, the inputs and outputs of the program are usually represented by large data structures. Since there are potentially many input/output values, they may not fit into main memory but must be read from and written to files during runtime. Second, all tasks are completely independent, except that the individual outputs may have to be combined at the end into a single file. Third, the execution order of tasks does not effect the correctness of the final output. Thus an arbitrary order may be applied to the task pool.

```
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.    WriteResult( outFile, R );
11.  }
12. }
```

Figure 2: The bag-of-tasks paradigm specification

Figure 2 shows the structure of the bag-of-tasks paradigm. The first step is to initialize the problem data (line 2). Then the bag of tasks (BT) is created, which is accomplished by the loop in lines 3-5. 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 on lines 7-10 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 compute function f to it (line 9), and writes the result into a file (line 10). Specifying a bag-of-task application requires specifying four functions:

1. **InitData**: this is the initialization function that generate the problem data.
2. **Termination_condition**: this specifies the condition for determining when the bag of tasks is empty, either explicitly as a fixed integer or implicitly by reaching the end of file.
3. **GenNextTask**: this is the input function that produces a new task whenever it is called. The state of task generation might also be recorded in the input problem data.
4. **Compute**: this is the main compute function that solves a task and generates a result.
5. **WriteResult**: this is the output function that saves the result R in a file.

In addition, 2 data structure definitions must be provided:

1. The data structure describing a task (T).
2. The data structure describing the result (R).

3.2 Branch-and-bound paradigm

Branch-and-bound search is applicable to various combinatorial optimization problems, and is generally applied when the goal is to find the exact optimum. A branch-and-bound search is described as a search through a tree, in which the root node corresponds to the original problem to be solved, and each descendant node corresponds to a subproblem of the original problem. Each leaf corresponds to a feasible solution. The search tree is constructed dynamically during the search and consists initially of only the root node. To speed up the search, a subtree is pruned if it can be determined that it will not yield a solution that is better than the best currently-known solution.

The basic structure of the branch-and-bound paradigm is presented in Figure 3. Without loss of generality, we assume the goal of the program 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. The algorithm maintains a pool L of tree nodes that have yet to be explored. S is the current best solution, and B is the bounding value of S.

The algorithm starts by setting the bound B to infinity (line 2), initializing the problem data (line 3), and generating the root node of the branch-and-bound tree, R (line 4). Lines 5–7 are optional: an improved starting bound B may be generated 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). The following while loop (lines 10–23) then represents the branch-and-bound search. 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, a new bound is computed for the subproblem. Lines 16–20 then accomplish the pruning: If SN is a partial solution whose bound is lower than the current bound B, it is added to the pool (line 17); otherwise it is discarded, i.e., the subtree is pruned. If SN is a complete solution and is better than the currently best solution S, it replaces S and its bound replaces B (line 19-20). After the pool of subproblems to be solved is drained, the solution to the problem is written (line 24) and the program terminates. Specifying a branch-and-bound application requires specifying seven functions:

1. **GenProblemData**: this is the function that initializes the static problem data.
2. **GenRootNode**: this is the function that generates the root node of the search tree.
3. **GenInitSol**: this is the function that generates an initial feasible solution, which is used to provide an initial pruning bound.

```
1. BranchAndBound() {
2.   B = +∞;
3.   D = GenProblemData();
4.   R = GenRootNode( D );
5.   if ( an improved initial 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.    }
23.  }
24.  WriteSol( S );
25. }
```

Figure 3: The branch-and-bound paradigm specification

4. **GenBound**: this is the function that computes a value for a particular node in the subtree. For a leaf node, which corresponds to a feasible solution, the value returned is the value of that feasible solution. For a non-leaf node N in a minimization problem, the value returned is a lower bound on the value of any feasible solution corresponding to a leaf node that that is a descendant of N.

5. **NextBranch**: this is the branching function that defines how to divide a problem into subproblems. Repeated calls to this function return the subproblems and then the value Nil, with each subproblem being returned exactly once.

6. **IsSol**: this is the function that check whether a tree node is a solution or not (i.e., whether it is a leaf node in the branch-and-bound tree).

7. **WriteSol**: This is a function that writes the result to an output file.

In addition, the definition of the data structure describing a tree node must be provided.

3.3 Genetic programming paradigm

The genetic programming paradigm also solves optimization problems, but using the Darwinian principles of survival and reproduction of the fittest, and genetic inheritance [Gol89]. Unlike branch-and-bound, the genetic programming paradigm is generally applied to find a good but not necessarily optimal solution.

Figure 4 shows the basic structure of a sequential genetic programming paradigm. The first step generates the static problem data (line 2) and then randomly creates an initial population P of size S (line 3). The while loop on lines 4-6 represents the evolution process. In each iteration, a new generation is created by applying genetic

```
1. Genetic() {
2.   D = GenProblemData();
3.   P = GenInitPop( S, D );
4.   until (Termination_condition) {
5.     P = CreateNextGen( P, D );
6.   }
7.   I = BestIndividual( P );
8.   WriteSol( I );
9. }
```

Figure 4: The Genetic programming paradigm specification

operations such as crossover, mutation, and reproduction (line 5). This process continues until a termination condition holds; typically, the termination condition is either based on the number of iterations completedd or the quality of the best solution obtained. Finally a result is designated (line 7) and written to a file (line 8).

Our implementation of a distributed version of the genetic programming paradigm is based on the concurrent modification of the paradigm shown in Figure 5. The population is divided into multiple subpopulations, which evolve independently but occasionally exchange individuals. This concurrent scheme allows the execution of genetic algorithms to achieve higher performance through course-grained parallelism, while also providing a good quality solution by mixing individuals from different subpopulations (so that the effect is that of having one large distributed population rather than a number of small, unrelated populations).

Figure 5 shows the structure of the concurrent genetic programming paradigm. The population pools form the nodes of a connected graph, with edges connecting certain pairs of nodes, so that each population pool has a well-defined set of neighbors. (In our implementation, we support a complete graph, a ring, or a toroidal grid, but any graph is in principle possible.) The processing for each population pool is shown in the loop on lines 3–19. In addition to the basic genetic operations, each subpopulation selects emigrant individuals to be sent to its neighbors as shown in lines 9-14; emigrant selection favors individuals with better fitness. Each subpopulation also receives immigrants from its neighbors and uses them to replace selected individuals as shown in lines 15-17; the selection of individuals to be replaced is biased towards selecting individuals with bad fitness. Once the basic loop is complete, each subpopulation determines its best individual; the best of all of these is then computed (lines 21–28) and written to the output file (line 29). Note that the pseudocode assumes that a minimization problem is being solved, so a "good" fitness value is actually a low value.

Specifying a concurrent genetic programming application requires specifying nine functions:

1. **GenproblemData**: This is a function that generates the initial problem data.
2. **GenInitPop**: This is a function that generates the an initial population pool of a specified size.
3. **Termination_condition**: This is a predicate that specifies the termination criterion.
4. **CreateNextGen**: This is a function that takes a population pool as a parameter and applies genetic operations to create a next generation of the population pool.
5. **SelectGoodIndividual**: This is a function that selects an emigrant from a population, typically by random selection biased towards individuals with good fitness values.
6. **Replace**: This is a function that, given an immigrant, either selects an individual to be replace by the immigrant or discards the immigrant. The selection of the individual to be replaced is typically done by random selection biased towards individuals with bad fitness values.
7. **BestIndividual**: This is a function that designates the best individual of a population as the result.
8. **Fitness**: This is a function that measures the fitness of an individual.

```
1. ConGenetic() {
2.   D = GenProblemData();
3.   for( each subpopulation pool ) {
4.     P = GenInitPop( PopulationSize, D );
5.     until (Termination_condition) {
6.       for( EmigrationInterval iterations ) {
7.         P = CreateNextGen( P, D );
8.       }
9.       for( each neighbor N of this polulation pool) {
10.        for( EmigrationRate iterations ) {
11.          I = SelectGoodIndividual( P );
12.          send I to N;
13.          }
14.      }
15.      for( all the immigrants M ) {
16.        P = Replace( P, M );
17.      }
18.    }
19.  }
20.  BF = +∞;
21.  for( each subpopulation pool P ) {
22.    I = BestIndividual( P );
23.    F = Fitness( I, D );
24.    if( F < BF ) {
25.      BF = F;
26.      BR = I;
27.    }
28.  }
29.  WriteSol( BR );
30. }
```

Figure 5: The concurrent genetic programming paradigm specification

9. **WriteSol**: This is a function that writes the result to an output file.

The definition of the data structure describing an individual (I) in the population pool must be provided. Also, several controlling parameters must be specified: the size of a subpopulation (*PopulationSize*), the number of new generations to be created between sending out a wave of emigrants (*EmigrationInterval*), and the number of emigrants to be sent out in a wave (*EmigrationRate*).

In the concurrent genetic programming paradigm, the functions **SelectGoodIndividual** and **Replace** are optional. If the user does not provide these two components, each subpopulation will evolve independently and there will not be any exchanging of individuals among subpopulations.

4 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 user specifies and starts the application through a *submission window*. In this window, the user specifies application-specific program components. This allows the user-specified program to focus on the key application-specific computational functions; the coordination and distribution of the computation is left to the system. 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 programming environment is paradigm oriented, so 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.

4.1 Component Specification: The Submission Window

Figure 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 sequential source programs, header file(s), and program file(s). Below that, the user specifies the goal of optimization (whether to find the maximum solution or minimum solution), the branching rule, the bounding rule, the procedure to determine whether a tree node represents a complete solution, the data structure describing a tree node, and initialization functions. 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 of $-\infty$ for a maximization problem. An output procedure for a task needs to be specified so that we can write the solution to a file. Note that the above components correspond exactly to those indentified for this paradigm in Section 3.2. Finally, the user can specify a recommended number of machines to use in the computation.

4.2 Interaction with the Running Application: The Feedback Window

While the application is running, the user and the running application are able to interact. Figure 7 shows the feedback window of a branch-and-bound application, which communicates the running status of the application to the user.

At the very top of the window, a moving icon indicates that the application is still running. Below that is a graph showing 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. This graph allows the user to assess the progress that the application is making. 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), text describing the currently optimal solution, and any error/success messages.

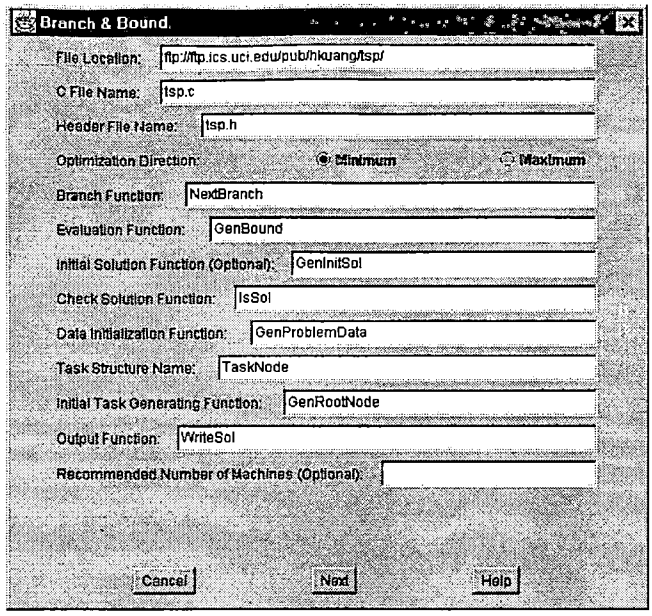


Figure 6: Interface for submitting task

The user can also interact with the running application. For example, if the most recently found solution is satisfactory, the user can stop the execution rather than wait for it to find an exact optimal solution. The user can also modify certain parameters through the feedback window. In the current implementation, there are no run-time modifiable parameters in the branch-and-bound paradigm. However, the genetic programming paradigm does have certain parameters that can be modified at run-time, such as the frequency at which emigrants are generated.

5 Distributed Implementation of Paradigms

In this section, we discuss the distributed implementation of the paradigms in the agent-based MESSENGERS system. One important feature of the MESSENGERS system is that it allows a Messenger to navigate through a logical network, carrying its own state and behavior. The mapping of a logical network to the underling hardware network can be done explicitly by the messenger script or implicitly by the system. As a result, the Messengers can adapt to the changing network. Fault tolerance and load balancing are provided at the system level, transparently to the user as described in [GBD99]. For the purpose of describing the distributed implementation of each of the paradigms, we first describe the logical network for each paradigm, and then discuss the behavior of each messenger.

5.1 Bag-of-Tasks Paradigm

The logical network used to implement the bag-of-tasks paradigm using MESSENGERS is a star topology as shown in Figure 8(a). The "Meeting Room" node is a central node where the bag of tasks is stored, while the "Office" node is where a worker solves a task. Since each task can be executed independently, no information needs to be exchanged between workers, and therefore no link exists between office nodes.

Multiple worker Messengers (w) work concurrently in the system. Each of them hops back and forth between

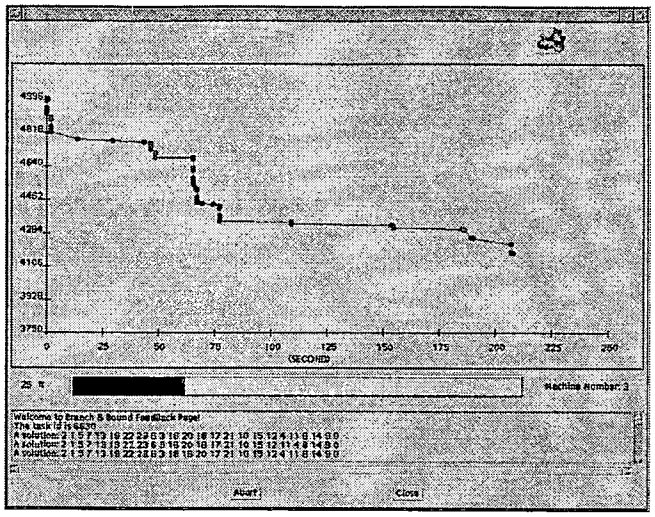


Figure 7: Feedback window for branch-and-bound paradigm

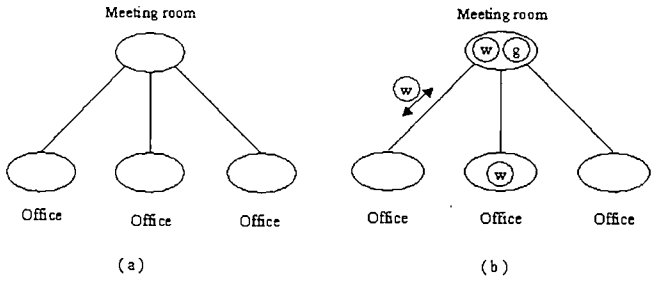


Figure 8: Logical network for a bag of tasks paradigm in MESSENGERS System

the meeting room and its office, as shown in Figure 8(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. Because no new tasks are added dynamically to the bag of tasks, a worker terminates when the bag of tasks becomes empty.

Because the number of tasks in the bag-of-tasks paradigm 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 Figure 8(b) exists for this purpose. The task-generation messenger stays at the central node. When the number of tasks in the task pool falls below a certain threshold, the task-generation messenger generates additional new tasks. When the number of results rise above a threshold, the task-generation messenger writes the available results to the output file. In this way, I/O operations are performed concurrently with the execution of tasks. The task-generation messenger terminates when all the results have been written to the output file, thus effecting global termination.

The number of "office" nodes (i.e., the number of workers) is not fixed, but can be dynamically changed to adapt to changes in the availability or workload of machines in the network. It is easy to implement this feature in the MESSENGERS system. Adding a worker is done by injecting a worker messenger into the system: the worker adds an office node and a link to the meeting room to the logical network, and then starts to work. If a machine is to be removed from the system, a control Messenger is injected, informing the system of the change. The office node residing on the machine to be removed is deleted, and the worker Messenger will die as it has no office to work at. The non-preemptive scheduling policy of MESSENGERS [BFD96] ensures that if the worker is in the middle of a task, it will complete the task before the control Messenger removes the node.

One implementation detail worth noting is the approach to random number generation. In computations using random numbers, it is useful for the computation performed by each task to be repeatable. This requires that sequence of random numbers used by any given task is the same over multiple runs. The usual approach to generating random numbers in distributed computation is using a different random number seed on each machine. Since the assignment of tasks to machines may vary over multiple runs, this will not result in a repeatable computation.

In the PODC implementation of the bag-of-tasks paradigm, each task is associated with a different seed. When a worker takes a task to its office, it takes the seed along with it and sets the seed for the local random number generator before executing the task. The computation will be repeatable provided the seed associated with each task is the same over multiple runs. One simple way to ensure this is to make the seed for each task be a function of the task number (i.e., of its position in the list of generated tasks).

5.2 Branch-and-Bound Paradigm

The logical network supporting the distributed implementation of the branch-and-bound paradigm is presented in Figure 9(a). The "Meeting room" node is a central node where the initial task pool is stored, while the "Office" node is where each worker is exploring a portion of the search space. When a worker finds a solution that is better than the best previously-known solution, the new pruning bound is communicated to other workers. The process of notifying the other workers of the new pruning bound is facilitated by fully connecting the office nodes in the logical network. Note that the communication of the new pruning bound to other workers is an example of non-essential information exchange as defined in Section 3: each task would successfully complete without this information exchange, but using an improved pruning bound discovered by another task can improve its performance significantly.

Three types of Messengers are used to implement the branch-and-bound paradigm. Their behaviors are shown in Figure 9(b). An initialization Messenger (g), which stays in the meeting room, generates the static problem data, the initial pruning bound, and the initial task pool. Multiple worker messengers (w) exist in the system, one per office node. Each worker repeatedly attempts to find a task to perform, as described below. If

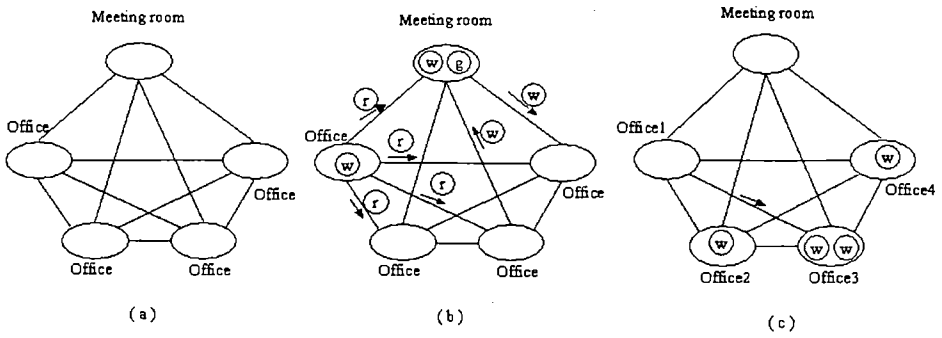


Figure 9: Logical network for branch and bound search paradigm in MESSENGERS System

a worker successfully finds a task, it hops to its office to work on the task. When a worker is unable to find a task, it sets a flag in the meeting room and then terminates. Global termination is detected when the last worker becomes idle. When a worker messenger finds a better solution, runner messengers (r) are created. These runner messagers hop to the other office nodes to update the pruning bound.

Balancing the workload among the workers requires some care. The effect of pruning makes it difficult to predict the load of a task in advance. We use a two-part strategy to try to keep all active workers busy. First, tasks are obtained on demand from the meeting room. When a worker is looking for a new task, it first hops to the meeting room and attempts to obtain a task from the initial task pool. If this pool is not empty, it takes the task to its office and explores the corresponding portion of the problem space locally. As the worker generates new subproblems, it keeps them in a local subproblem pool. When its subproblem pool becomes empty, it once again looks for a new task.

The second part of the load-balancing strategy comes into play when a worker goes to the meeting room and finds that the initial task pool is empty. The worker chooses another worker at random and attempts to "steal" a task from the chosen worker's local task pool as shown in Figure 9(c). If the chosen worker has a nonempty task pool, the worker executes the "stolen" task, storing generated subproblems in its own local task pool as above. Otherwise, the worker randomly chooses another worker to steal a task from. After a certain number of unsuccessful attempts (currently set at half the number of active workers), the worker terminates.

One issue related to the load balance and pruning bound updating is the stalling of worker messengers. Because the MESSENGERS system uses a nonpreemptive scheduling mechanism, a worker messenger has to occasionally voluntarily relinquish the processor ("stall") to allow a runner messenger or a worker messenger trying to steal a task to run on its node. The frequency of stalling is an important factor affecting system performance. If a worker stalls too frequently, it will add system overhead because of the cost of context switching. On the other hand, if a worker stalls too infrequently, it may perform work that would have been unnecessary if it had allowed a runner to update its pruning bound or another worker to steal one of its tasks. We experimented with a very simple stalling strategy: the worker stalls after processing a fixed number of subproblems. Our experiments suggest that an appropriate stalling frequency is once every 100-150 subproblems.

Another design consideration is management of the subproblem pool, which in turn is closely related to the selection rule. Task pool organizations as a last-in-first-out stack, a priority queue, or a first-in-first-out queue correspond, respectively, to depth-first, best-first, and breadth-first selection rules. In our present implementation, selection from the initial task pool is done using breadth first, and selection from the local task pool is done using a depth-first strategy. Using breadth-first selection on the initial task equalizes the granularity of the initial tasks, while using depth-first on the local task pools keeps local search fast and minimizes memory use by the workers.

5.3 The Genetic Programming Paradigm

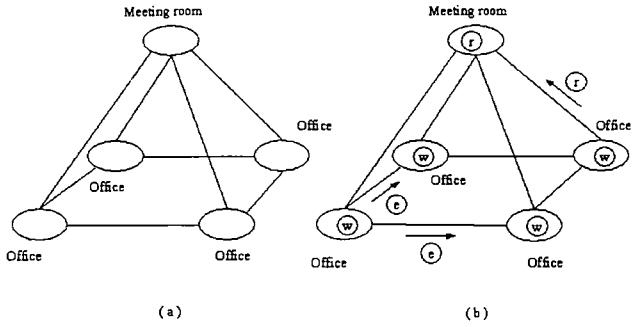


Figure 10: Logical network for the genetic programing paradigm in MESSENGERS System

Figure 10(a) illustrates one possible 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. The office nodes can be connected using various network topologies: we presently support fully connected, toroidal mesh, and ring topologies. The topology affects the speed at which a good solution arising in one population pool will be disseminated to other population pools. It also plays an important role in determining the communication cost: a more highly connected topology will mix individuals better, but at the time it will result in higher communication cost. At the other extreme, it is possible to specify that no pairs of office nodes are connected. In this case, no individuals will be exchanged among office nodes. In the example of Figure 10, the office nodes are connected by a ring topology.

Three types of messengers exist in the system, as illustrated in Figure 10(b). The worker messengers (w) execute the basic genetic algorithm, one per office. Because the genetic algorithm is probabilistic, each worker messenger starts with setting a different seed at the node and randomly generates an initial population pool. It then repeatedly applies genetic operations to create subsequent generations until the termination condition is satisfied. Whenever an individual is generated that is better than the previous best in the entire system, a runner (r) messenger will send it to the meeting room node and report 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.

After a fixed interval of generations, where the frequency is specified by the user, a worker exchanges individuals with neighboring workers. A certain number of emigrants are selected and distributed. Exporter messengers (e) are generated, carrying emigrants to the neighbors and using them to replace individuals of lower fitness in the neighbors' population pools. Two parameters (in addition to the network topology) control the migration of individuals. These parameters are set by the user and may be modified at runtime through the feedback window. One is the *emigration interval* (the number of generations between emigrations), and the other is the *emigration rate* (the number of emigrating individuals). Increasing the emigration rate or decreasing the emigration interval causes more population mixing but increases the communication cost.

6 Performance

This section describes the performance of distributed programs running in the PODC system. All the experiments are performed on 85 MHz SPARCstation 5 workstations connected by a 10 Mbit Ethernet.

The bag-of-tasks paradigm is widely used in many scientific computation. Our experiments with this

paradigm were based on a Monte Carlo simulation of a model of light transport in organic tissue [PKJW89]. The simulation runs as follows. Once launched, a photon is moved a distance where it may be scattered, absorbed, propagated undisturbed, internally reflected, or transmitted out of the tissue. The photon is repeatedly moved until it either escapes from or it is absorbed by the tissue. This process is repeated until the desired number of photons has been propagated. 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. To offset the cost of communication, each task simulates the movements of 1000 photons. The number of photons simulated is 1,000,000. The experiment results are shown in Figure 11. The graph presents a near-linear speedup.

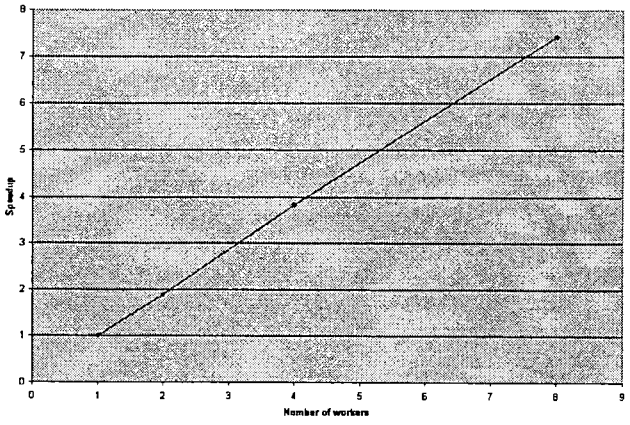


Figure 11: Speedup for the bag-of-tasks paradigm experiments

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 [HK70] of the unvisited cities in a partial tour. We used data for 24 cities.

During our experiments, we observed nondeterministic performance behavior of the distributed branch-and-bound program [LS84]. Therefore, we executed both the sequential and distributed programs ten times, each with different input data. The experiment results in Figure 12 represent the average speedup from ten runs. These show a near-linear speedup for the distributed branch-and-bound program.

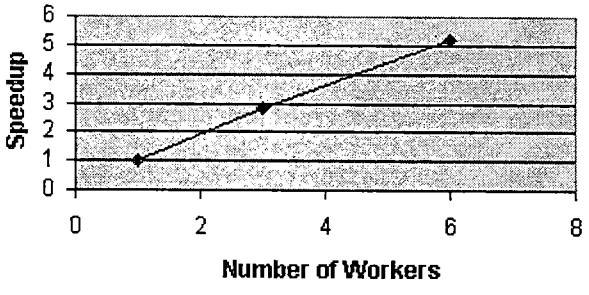


Figure 12: Speedup for the branch-and-bound paradigm experiments

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 as a result of exploring the problem space concurrently. To explore how the search order effects the performance, we conducted another type of experiment. We ran both sequential and distributed programs ten times, each time is with the same input data, but with a different permutation of the initial task pool. The experiment results are shown in Figure 13. The horizontal axis represents permutations of the initial task pool, while the vertical axis represents the execution times of the programs in seconds. These experiments show that the program execution time is greatly influenced by the search order. For example, for the sequential program, the minimum execution time is 241.2s, while the maximum execution time is 4877.6s depending on the search order. The graph also shows that introducing multiple workers working simultaneously has a significant smoothing effect on the execution time, making the execution time much more predicable.

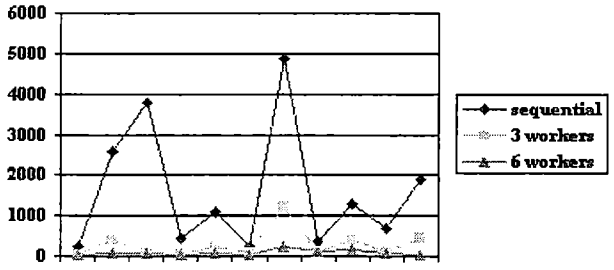


Figure 13: Execution time of branch-and-bound program with different permutation of search order

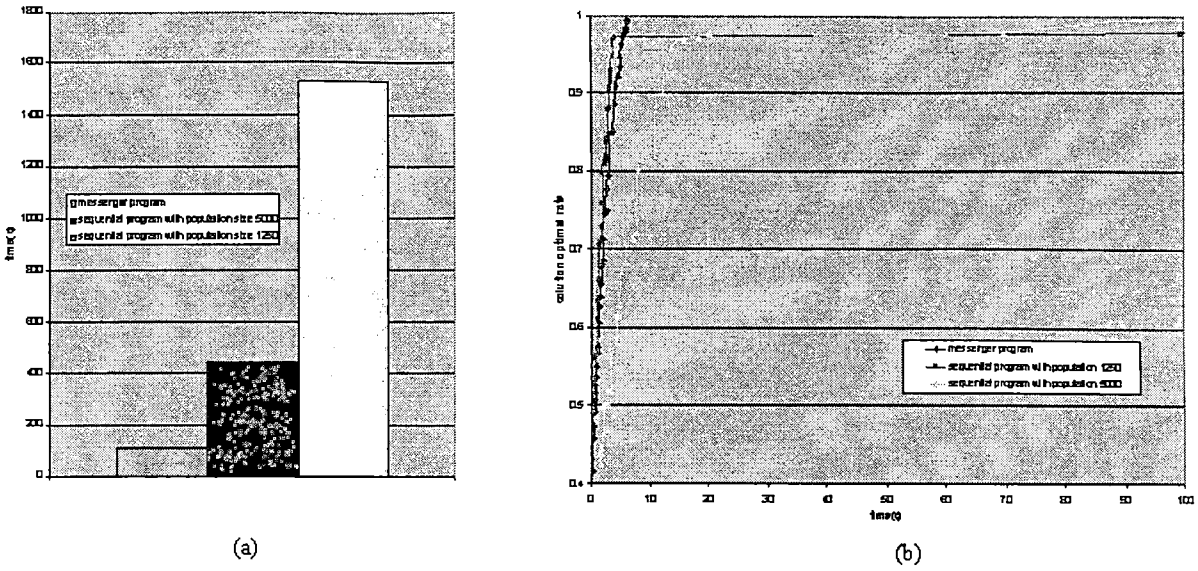
We tested the the genetic programming paradigm using also the travelling salesman problem. The size of the TSP was 30 cities. The original sequential program was downloaded from the Web [LaL96]. We restructured the source code to fit into our paradigm and also added a mutation operator. In this program, the tours are encoded as a 2-dimensional array (a NxN matrix) of bits that store city adjacencies in both directions. For each iteration, a tournament selection method is used to choose a list of tours, the best two of which are combined to produce two offspring using crossover, and one of which is mutated. The crossover operation places the edges that are shared by the parent in both children. The mutation operation takes a reciprocal exchange strategy that swaps two cities. The distributed programs were run on a network of four workstations, each of which contains an office node. All the office nodes are connected as a ring. Because genetic programs are probabilistic, we ran each program 10 times.

To describe the performance of genetic programs, we define a term, called a quality ratio, which shows how close a solution is to the best solution. In case of minimization , the quality ratio of a given solution is defined as

fitness(optimal solution)/fitness(solution)

where the fitness function evaluates a given solution. The range of quality ratio is (0.0, 1]. The bigger the quality ratio, the closer is the solution to the optimal solution. The sequential program attained a quality ratio of 99.4% after running for an indefinite period of time. We then used this ratio as termination condition for the distributed programs, so that all solution would yield the same quality of solution.

Figure 14(a) compares the average performance of three programs. The population size of a worker messenger in the first program is 1250. The other two are sequential programs with population size 1250 and 5000 respectively. From the figure we can see that the messenger program finds the solution fastest, which is four times as fast as the sequential program with population size on 5000.



(a) (b)

Figure 14: Speedup for the genetic programming paradigm experiements

We also selected a representative run of three programs and illustrate the evolution process in Figure 14(b). The grey line is the sequential program with population size 1250. The white line is the sequential program with population size 5000 whose population includes that of the first sequential program. The black line is the worker messenger with the initial population the same as the first sequential one, but which exchanges immigrants periodically during the evolution. The horizontal axis represents time in seconds, while the vertical axis shows the quality ratio.

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 time to improve this solution. The sequential program with the larger population performs better after this point. The worker messenger 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 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 shortest time.

7 Related Work

The paradigm-oriented approach has been employed to build parallel computation models in which programmers do not have to be aware of parallelism at all. One example of this approach was developed by Murray Cole, who presents an algorithmic skeleton approach to controlling parallelism [Col89]. The algorithmic skeletons, which are conceptually the same as our paradigms, encapsulate control structures. In [Col89], four skeletons (divide and conquer, interactive combination, cluster, and task queue) are specified, and their possible implementation in grid based parallel architecture are discussed. Similar work using paradigms for reduction and mapping over pairs, pipelines, and farms, has been done by Darlington's group at Imperial College [DFH93]. The Pisa Parallel Programming Language (P3L) [Pel93] uses a set of parallel paradigms such as pipeline, worker farms, and reductions as basic constructs to implicitly express parallelism. They discuss a parallel implementation on a massively parallel architecture. Other related work includes Rabhi's that describes some of the common parallel programming paradigms which explains the basic principles behind a paradigm-oriented programming

approach [Rab95]. Gorlatch studies extensively the parallel implementation of divide-and-conquer paradigm and its application to the FFT computation [Gor98].

The major drawback of the above approaches is that they use functional languages to formally specify the paradigms, because the functional languages allow higher order functions as parameters. However, while the functional languages elegantly abstract a paradigm, they generally produce inefficient programs [BK96]. The Skil language [BK96] developed at Aachen University of Technology represents a step away from purely functional solutions, by integrating functional features with an imperative (C-based) language, but the paradigms are still expressed by functions. Our approach allows the user to program application-specific components in C language, and it allows the user to specify those components via a graphical interface. This Web-based interface eliminates the difficulty in specifying an application using a not commonly used language. The supporting C language provides the desired flexibility and it also ensures the efficiency of the running code.

Many programming tools and environments have been developed to support paradigm oriented programming. In [SSG91], Singh et al. describe a template-based programming environment; the major templates studied are pipeline and contractor (replicated process). Siu *et al.* describe the concept of design patterns which are implemented as reusable code skeletons [SSGS96]. These two systems differ from our paradigm approach in that the templates or patterns are common structures in parallel programs. Therefore, the user needs to be aware of parallelism as part of the programming task, and takes some responsibility for dividing the problem. ParAgent [KM97] is a tool to parallelize legacy code. The approach uses high-level knowledge of parallelization, with the user providing a roadmap for the parallelization. The kinds of problems supported are programs applying mathematical techniques, such as finite difference, boundary element, and finite element methods. The parallel code is executed on multiprocessor machines. In contrast, our system supports distributed computing over a network of workstations. Using a mobile agent infrastructure, we make the paradigm-oriented computing feasible in a dynamic heterogeneous computing environment.

8 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 a Web-based interactive graphics interface through which programs are submitted, invoked, and monitored.

By implementating three widely used paradigms—bag-of-tasks, branch-and-bound, and genetic programming—we have demonstrated the viability of this approach for use in heterogeneous and dynamically changing cluster of connected 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.

References

[BFD96] L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8), Aug. 1996.

[BK96] G. H. Botorog and 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)*, pages 243–252. IEEE Computer Society Press, 1996.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[DFH93] J. Darlington, A.J. Field, and P.G. Harrison. Parallel programming using skeleton functions. PALE'93, Parallel Architectures and Languages Europe, June 1993.

[FBD99] M. Fukuda, L. F. Bic, and M. B. Dillencourt. Messages versus messengers in distributed programming. *Journal of Parallel and Distributed Computing*, 57:188–211, 1999.

[FBDM98] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Distributed coordination with messengers. *Science of Computer Programming*, 31(2), 1998.

[GBD99] E. Gendelman, L. F. Bic, and M. B. Dillencourt. Technical Report 35, 1999.

[Gol89] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, MA, 1989.

[Gor98] Sergei Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *The Journal of Supercomputing*, 12(1/2):85–97, January 1998.

[HK70] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[KM97] Suraj C. Kothari and S. Mitra. Parallelization agent: A new approach to parallelization of legacy codes. Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[LaL96] Michael LaLena. Travelling salesman problem using genetic algorithms, 1996. http://www.lalena.com/ai/tsp/.

[LS84] T. H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communuications of the Association of Computing Machinery*, 27(9):594–602, June 1984.

[Pel93] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, Dipartimento di Informatica, Universita' di Pisa, March 1993.

[PKJW89] S. A. Prahl, M. Keijzer, S. L. Jacques, and A. J. Welch. A Monte Carlo model of light propagation in tissue. In *Dosimetry of Laser Radiation in Medicine and Biology*, SPIE Institute Series Vol. IS 5, pages 102–111. 1989.

[Rab95] Fethi A. Rabhi. A parallel programming methodology based on paradigms. In *Transputer and Occam Developments*, pages 239–252. IOS Press, 1995.

[SSG91] Ajit Singh, Jonathan Schaeffer, and Mark Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transations on Parallel and Distributed Systems*, 2(1):52–66, January 1991.

[SSGS96] S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. PDPTA'96, August 1996.