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 the implementation underlying 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 describe their implementation 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*

1. Introduction

Developing distributed applications can be a daunting task, because the programmer must deal with the details of parallelism, data distribution, inter-process communication, and synchronization. Fortunately, there is no need to develop every distributed application from scratch, because many are based on the same computational paradigm, sharing a common communication and synchronization structure that is independent of the application. If an application fits a particular paradigm, the programmer only needs to supply the application-specific functions, which are inserted into the skeletal structure of the paradigm. These functions can be supplied as self-contained building blocks, which can usually be easily extracted from a sequential program based on the same paradigm, if such a program already exists. This further eases the transition form sequential to distributed programming. Another advantage of separating the paradigm-specific communication/synchronization structure from the application-specific computations is that the former can be optimized for each underlying architecture, which implicitly benefits all applications using that structure.

In [10] we introduced our environment for paradigm-based distributed computing, which allows the programmer to develop a distributed application by providing the functions needed for a chosen paradigm, and to monitor the program execution. Both the development of the application and its execution are handled through a graphics-based nonprocedural interface, which requires no knowledge of distributed programming.

The focus of the present paper is on the underlying implementation of this environment using mobile agents, and its performance evaluation. The paper is organized as follows. In section 2, the three paradigms, bag-of-tasks, branch-and-bound, and genetic programming, will be presented in detail. For each paradigm, we give a general overview, illustrate the underlying implementation using MESSENGERS, and present performance evaluations. Related work is discussed in Section 3, followed by general conclusions.

2. 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-grain 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.

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 underlying 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 [5]. 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.

The performance of distributed programs running in the PODC system is also presented. All the experiments are performed on 85 MHz SPARC station 5 workstations connected by a 100 Mbps Ethernet.

2.1. Bag-of-tasks paradigm

2.1.1. Overview

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 applications share a general structure. The first step is to initialize the problem data. Then the bag of tasks is created, where 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 actual computation is represented by a loop, 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, solves it by applying the main compute function to it, and writes the result into a file.

2.1.2. Distributed Implementation

The logical network used to implement the bag-of-tasks paradigm using MESSENGERS is a star topology as shown in Figure 1(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.

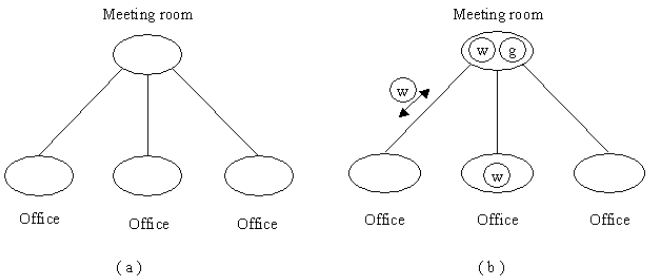

Figure 1. Logical network for a bag of tasks paradigm in the MESSENGERS System

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 1(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.

After generating a number of initial number of tasks, the task-generation messenger injects multiple worker Messengers (w), each of which is assigned an office. These workers work concurrently in the system. Each of them hops back and forth between the meeting room and its office, as shown in Figure 1(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.

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 [1] 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). Some care should be taken to make sure that the random streams are different. For example, the function should be injective.

2.1.3. Performance

The bag-of-tasks paradigm is widely used in many scientific computations. Our experiments with this paradigm were based on a Monte Carlo simulation of a model of light transport in organic tissue [15]. 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 2. The graph presents a near-linear speedup.

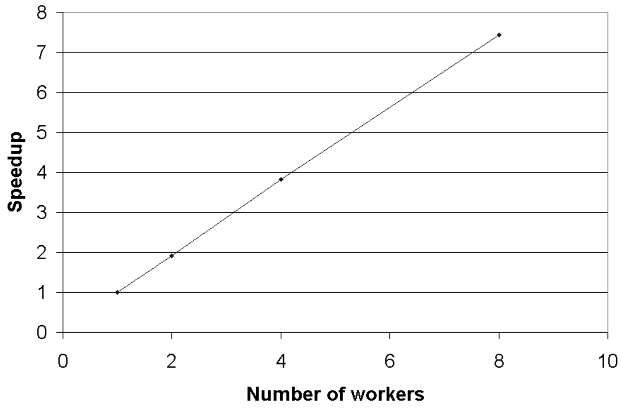

Figure 2. Speedup for the bag-of-tasks paradigm experiments

2.2. Branch-and-bound paradigm

2.2.1. Overview

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 as follows (for concreteness, assume we are solving a minimization problem). The algorithm starts by setting an initial bound B and generating the root node of the branch-and-bound tree, which is initially the sole member of the pool of problems to be solved. The following step is then repeated. A node N is selected from the pool of problems to be solved. If all subnodes of N have been explored, then it has been completely solved and so is removed from the pool. Otherwise, one of the subnodes of N is generated, and a new bound is computed for the corresponding partial solution. If the bound on the partial solution is lower than the current bound B, it is added to the pool; otherwise it is discarded (i.e., the subtree is pruned). If the generated subnode represents a complete solution and is better than the currently best solution, it becomes the current best solution and its bound replaces B. After the pool of subproblems to be solved is drained, the solution of the problem is written and the program terminates.

2.2.2. Distributed Implementation

The logical network supporting the distributed implementation of the branch-and-bound paradigm is presented in Figure 3(a). The "Meeting room" node is a central node where the initial task pool is stored, while the "Office" node is where each worker explores 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 at the start of Section 2: each task would successfully complete without this information exchange, but using an improved pruning bound discovered by another task can improve its performance significantly.

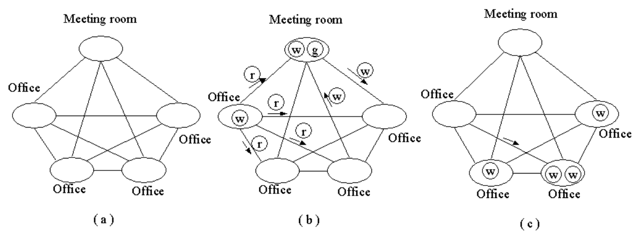

Figure 3. Logical network for branch and bound search paradigm in MESSENGERS **System**

Three types of Messengers are used to implement the branch-and-bound paradigm. Their behaviors are shown in Figure 3(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, either from the initial task pool or from the other workers, as described below. If 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 subtasks, it keeps them in a local task pool. When its local task 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 3(c). If the chosen worker has a nonempty task pool, the worker executes the "stolen" task, storing generated subtasks in its own local task pool as above, while the chosen worker keeps on working on the rest of the tasks in its local task pool. 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 until a certain number of the tasks which is linear to the number of office is generated, 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.

2.2.3. Performance

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 [8] 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 [11]. Therefore, we executed both the sequential and distributed programs ten times, each with different input data. The experiment results in Figure 4 represent the average speedup from ten runs. These show a near-linear speedup for the distributed branch-and-bound program.

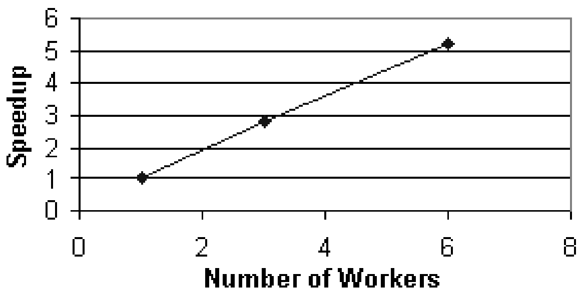

Figure 4. Speedup for the branch-and-bound paradigm experiments

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 5. 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 predictable.

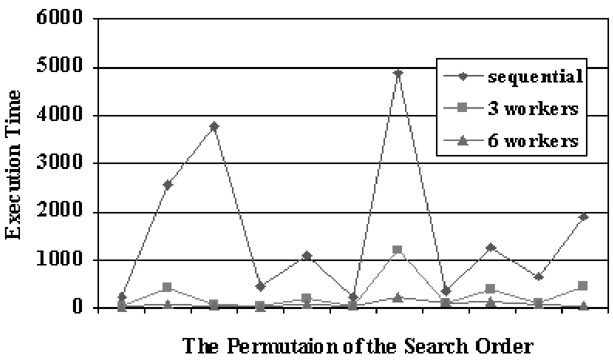

Figure 5. Execution time of branch-and-bound program with different permutation of search order

2.3. Genetic Programming Paradigm

2.3.1. Overview

The genetic programming paradigm also solves optimization problems, but using the Darwinian principles of survival and reproduction of the fittest, and genetic inheritance [6]. Unlike branch-and-bound, the genetic programming paradigm is generally applied to find a good but not necessarily optimal solution.

The basic structure of a sequential genetic programming paradigm is as follows. Initially, a random population is generated. The evolution process is then represented by a loop. In each iteration, a new generation is created by applying genetic operations such as crossover, mutation, and reproduction. This process continues until a termination condition holds; typically, the termination condition is either based on the number of iterations completed or the quality of the best solution obtained. Finally a result is designated and written to a file.

Our implementation of a distributed version of the genetic programming paradigm is based on a concurrent modification of the above paradigm. 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 coarse-grain 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).

In 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 addition to the basic genetic operations, each subpopulation selects emigrant individuals to be sent to its neighbors; emigrant selection favors individuals with better fitness. Each subpopulation also receives immigrants from its neighbors and uses them to replace selected individuals; 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 and written to the output file.

2.3.2. Distributed Implementation

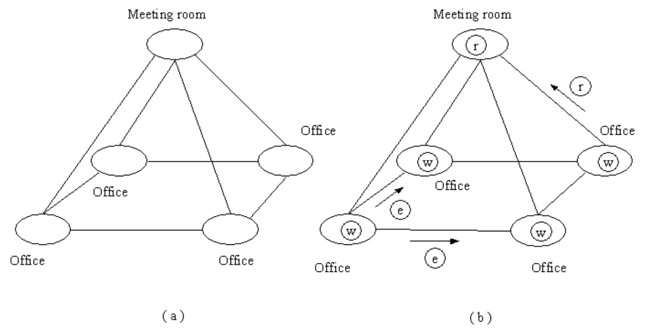

Figure 6. Logical network for the genetic programing paradigm in MESSENGERS System

Figure 6(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 6, the office nodes are connected by a ring topology.

Three types of messengers exist in the system, as illustrated in Figure 6(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. The seed is passed to each worker as an argument when the worker is injected. Some care is taken to make sure that they are different. It 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 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.

2.3.3. Performance

We tested the genetic programming paradigm, also using the travelling salesman problem. In this case we used a 30-city problem. The original sequential program was downloaded from the Web [12]. 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, which we call the *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

$$\text{fitness(optimal solution)/fitness(solution)}$$

where the fitness function evaluates a given solution. The range of the quality ratio is (0.0, 1.0]. 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 solutions would yield the same quality of solution.

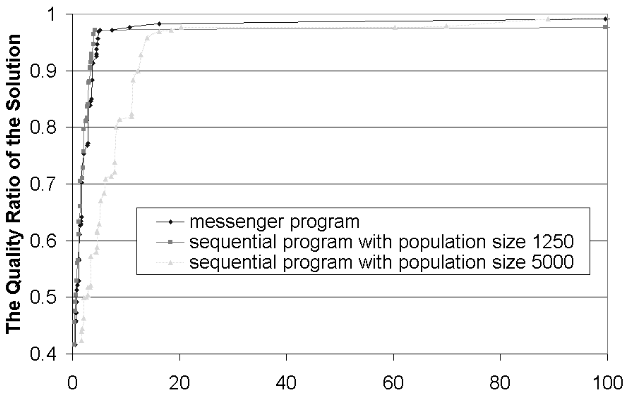

Figure 8. Representative evolution processes of the genetic programs

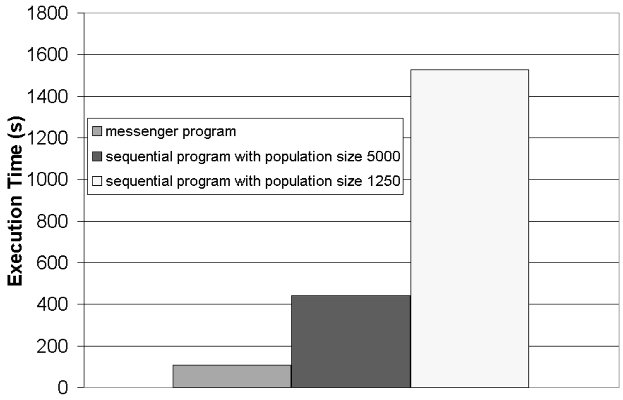

Figure 7. Speedup for the genetic programming paradigm experiements

Figure 7 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 MESSENGERS program finds the solution fastest and is four times as fast as the sequential program with population size on 5000.

We also selected a representative run of three programs and illustrate the evolution process in Figure 8. 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 MESSENGERS program with the initial population the same as the first sequential one, but which exchanges immigrants periodically during the evolution. The horizontal axis represents elasped 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 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 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.

3. 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 [3]. The algorithmic skeletons, which are conceptually the same as our paradigms, encapsulate control structures. In [3], 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 [4]. The Pisa Parallel Programming Language (P3L) [14] 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 description of some of the common parallel programming paradigms which explains the basic principles behind a paradigm-oriented programming approach [16]. Gorlatch studies extensively the parallel implementation of divide-and-conquer paradigm and its application to the FFT computation [7].

One drawback of the above approaches is that they use functional languages to formally specify the paradigms, because functional languages allow higher-order functions as parameters. However, while the functional languages elegantly abstract a paradigm, they generally produce inefficient programs [2]. The Skil language [2] 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 [17], Singh *et al.* describe a template-based programming environment; the major templates studied are pipelines and contractors (replicated processes). Siu *et al.* describe the concept of design patterns which are implemented as reusable code skeletons [18]. 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 [9] 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.

Another line of research that is related to our work is the task allocation. [19] and [13] designed a bidding strategy to allocate tasks to different machines. Our work differs from them in that in our case work exceeds capacity of the worker agents. The agents are saturated most of the time. Therefore, trying to pull tasks when they are free turns out to be a better scheme. Furthermore, our agents benefit from communicating with each other.

4. Conclusions

In this paper we have 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 implementing 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

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

[2] 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.

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

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

[5] E. Gendelman, L. F. Bic, and M. B. Dillencourt. An application-transparent, platform-independent approach to rollback-recovery for mobile agent systems. Technical Report 35, 1999.

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

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

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

[9] S. 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.

[10] H. Kuang, L. F. Bic, and M. B. Dillencourt. Podc: Paradigm-oriented distributed computing. In *7th IEEE Workshop on Future Trends of Distributed Computing Systems (FT-DCS'99)*, Dec. 1999.

[11] 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.

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

[13] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet-the popcorn project. In *Proceedings for the 18th International Conference on Distributed Computing Systems*, 1998.

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

[15] 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.

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

[17] A. Singh, J. Schaeffer, and M. 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, Jan. 1991.

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

[19] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12):1104–1113, Dec. 1980.