Distributing evolutionary computation in a networked multi-agent environment

Wei-Po Lee *, Hong-Yi Ke
Department of Information Management, National Sun Yat-sen University, Kaohsiung, Taiwan

A R T I C L E   I N F O

Article history:
Received 8 December 2009
Received in revised form 3 September 2010
Accepted 29 November 2010

Keywords:
Mobile agent
Evolutionary algorithm
Distributed computing
Autonomic computing
Gene regulatory network
Evolutionary robotics

A B S T R A C T

It has become increasingly popular to employ evolutionary algorithms to solve problems in different domains, and parallel models have been widely used for performance enhancement. Instead of using parallel computing facilities or public computing systems to speed up the computation, we propose to implement parallel evolutionary computation models on networked personal computers (PCs) that are locally available and manageable. To realize the parallelism, a multi-agent system is presented in which mobile agents play the major roles to carry the code and move from machine to machine to complete the computation dynamically. To evaluate the proposed approach, we use our multi-agent system to solve two types of time-consuming applications. Different kinds of experiments were conducted to assess the developed system, and the preliminary results show its promise and efficiency.

© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

Evolutionary algorithms (EAs) have been developed to construct more dynamic models of computational intelligence. They simulate the process of natural evolution to search for the fittest through selection and re-creation over generations. These algorithms are population-based optimization processes—based on the collective learning process within a population of individuals in which each individual represents a search point in the space of potential solutions for a given problem. Generally speaking, the dimension of the solution space is determined by the length of the chromosome. For instance, in the binary encoding scheme, a chromosome with length \( n \) indicates that the evolution techniques are expected to find the appropriate solution from a space with \( 2^n \) candidates. It can be observed that, when the length of the chromosome is reasonably increased to match the increase in task complexity, the solution space will grow exponentially. Consequently, the search will become increasingly difficult.

In order to use EAs to solve more difficult problems, two inherent features of EAs must be taken into consideration seriously. The first is premature convergence. As is well known, one of the attractive features of evolutionary algorithms is that they can quickly concentrate on searching promising areas of the solution space. But this feature also has a negative effect of losing population diversity before the goal is reached. In other words, the EA converges to local optima. To overcome this problem, parallel model EAs have been proposed to divide the original population into several subpopulations to maintain population diversity, and their efficiency has been confirmed [1–3]. The other problem that needs to be encountered is the computation time. As mentioned above, the EA is a population-based approach, and it has to evaluate all population members before selecting the fittest to survive. When using an EA to solve a problem, the population size must be reasonably large in order to allow the EA to search the space globally, and it typically increases with respect to the increasing problem difficulty. As a result, an inordinate amount of time is required to perform all the evaluations for a hard problem. To enhance the search
performance and speed up EAs at the same time, it is therefore critical to develop methods of running parallel model EAs on high-performance computing environments.

Conceptually, the central idea of the parallel model EA mentioned above is to divide a big population in a sequential EA into multiple smaller subpopulations and to distribute them on separate processors. One way to support this parallelism is to implement the parallel models on parallel computers. Depending on the number of subpopulations involved, different types of parallel computers have been used. For a small number of subpopulations, distributed memory MIMD (Multiple Instruction stream, Multiple Data stream) computers are often used. On the other hand, massively parallel SIMD (Single Instruction stream, Multiple Data stream) computers are used to implement parallel models with a large number of subpopulations. Though parallel computers can speed up EAs dramatically, they are not generally available. Also, it is expensive to upgrade their processing power and memory. The other way is to take public computing systems to speed up the computation [4,5]. The system of public computing is designed as a software platform utilizing heterogeneous computing resources from volunteer computers. It is an approach of distributing computation through the Internet, and has emerged as a useful computing architecture for tackling large-scale computation problems. However, computation with public computer architecture has some requirements that are not always available, such as little task interdependency, light network communication, and high network reliability. In particular, because the volunteer computational nodes are outside the system’s control, extra work (e.g., redundant computing that conducts the same work on different machines [6]) is needed to ensure the correctness of the results.

To make the parallel model EAs more realistic, in this paper we propose a flexible and efficient alternative, agent-based methodology to work on networked computers that are locally available and manageable to support parallelism. As the autonomous components in the autonomic computing paradigm [7,8], agents can sense the environment situations and then respond to the environment by acting reactively and proactively. Moreover, different agents can manage their behaviors and cooperate with each other to achieve their goal. With these unique characteristics, our agent-based system can operate in an adaptive way, in which mobile agents play the major roles that dynamically allocate available machines for the evolutionary computation (EC) code. To verify the proposed approach, we developed a prototype application system on a middleware platform JADE (Java Agent Development Framework) in which the agents control their own thread of execution and can easily be programmed to initiate the execution of actions autonomously. We also applied our agent-based system to two types of time-consuming EA applications. The preliminary results show the promise and efficiency of our approach.

2. Background

It has become increasingly popular to employ EAs to solve problems in different domains, and parallel models have been used for performance enhancement. As mentioned above, to parallelize the above evolutionary procedure is to divide a big population in a sequential EA into multiple smaller subpopulations so that they can be evaluated on separate processors simultaneously. According to the subpopulation size, parallel EAs can be categorized into two types: coarse-grain and fine grain. A coarse-grain (or island model) EA divides the whole population into a small number of subpopulations in which each subpopulation is evaluated by an independent EA in a separate processor. In this model, each subpopulation is manipulated by a sequential EA, and the selection and genetic operations are limited to happen only in the local subpopulation. A communication (or migration) phase is introduced in the island model EA to periodically select some promising individuals from each subpopulation and send them to other subpopulations. A common strategy is to use the selected individuals to substitute the worst ones in the neighboring subpopulations. In this way, an EA has higher possibility to maintain population diversity and to protect good solutions found locally. The coarse-grain EA is usually implemented on distributed memory MIMD architecture that is composed of a collection of processors and their associated memories, with the processors interconnected to provide a means for communicating. In this architecture, each processor has the capability to execute an independent instruction stream.

On the other hand, the fine-grain model EA divides the original population into a large number of subpopulations in which each of them includes only a small number of individuals. This model is designed to take advantage of machines with a large number of processors (1000 or more), and, in the ideal case, each individual is evaluated on a different processor. In this model, the selection and mating are restricted to occur only between an individual and its localized neighborhoods. The network topology in a fine-grain-type EA determines how the local optimals spread to the entire population, and this affects the performance of EA profoundly. How to constrain the interactions between subpopulations is yet to be investigated. The fine-grain model is usually implemented on massively parallel SIMD computers that contain a control processor to drive several task processors: the control processor issues a single instruction stream to all task processors that execute the instructions simultaneously for their own individual data streams.

Implementing parallel model EAs involves expensive parallel computers that normally cannot be expected in a campus-based computing environment. A possible alternative is to use a public computing system, such as the BOINC (Berkeley Open Infrastructure for Network Computing [4]) instead. But it should be noted that public computing is different from grid computing. Grid computing shares secured, trusted and centrally managed resources within or between organizations, while in public computing the resources are contributed mostly from network volunteers, and the reliability of these machines is thus not guaranteed. Therefore, a promising method without expensive hardware facilities and with the control of the computing resources is to map parallel model EAs onto a set of locally networked computers. There are some possible
ways to manage the operation of such a computational framework, for example a client–server technique or an agent-based approach [9–11]. Client–server is the most common paradigm of distributed computing at present. However, in this paradigm, all components are stationary in respect to execution. It is thus not suitable for the goal here—no specific computational resources are preserved and dedicated to EAs, so the EC code needs to be moved occasionally to the available machines. Under such circumstances, a mobile agent-based design paradigm provides the better choice [12,13]. Mobile agents are software agents that are capable of transmitting themselves (their program and their state) across a computer network and recommencing execution at a remote site. Several multi-agent platforms are publicly available, including the most popular toolkits AgentTCL (which was later renamed D’Agent [14]), Tracy [13], Aglets [15], and JADE [16]. Application developers can use them as platforms and focus on the software development issues at the application level. In this work, we chose to use the JADE toolkit to develop an adaptive agent-based computational system to parallelize EAs. The relevant details and experiments are described in the following sections.

3. Adaptive multi-agent approach to distribute evolutionary computing

3.1. Networked computing environment

As indicated previously, our goal here was to distribute (map) EC code on available personal computers (PCs) without using a powerful connection machine. In fact, a network of PCs can be considered a MIMD-DM (Distributed Memory) computer, and the current networked computing environment provides an ideal platform to realize distributed computation. Although an extensive number of external computers can also be included to participate in the computation, considering the convenience of collecting experimental data for further analysis and the stability of networking, we chose to configure a set of PCs in our laboratory as the interconnected computing environment to speed up EC.

After preparing the computational infrastructure, we then designed an EC model that can easily operate on the framework in a distributive manner. Though there have been many models proposed and some researchers have tried to compare the performance of using coarse-grain and fine-grain models for parallelizing EC, the results came out to be inconclusive. Some prefer fine-grain models, but others favor coarse-grain ones. Therefore, an appropriate way is to perform parallelism based on the machine availability. In this work, we adopt a coarse-grain model and distribute the computation on our networked computing environment described above to achieve parallelism.

Our coarse-grain EC performs island model parallelism, in which each subpopulation is designed to allocate in one of the networked computers. The distributed EC continues for a certain number of generations before migration happens. During the migration interval (i.e., the period of consecutive generations), computation for each subpopulation is independent from that in others, so evolution for different subpopulations can proceed simultaneously on different machines. Due to the variation of machine specifications (e.g., CPU and memory utilities) and the discrepancy of evaluation time in individuals, computers running different subpopulations may take different amounts of time to complete the corresponding evolution. To perform individual migration as in the original island model, in our work a synchronized strategy is used to organize the distributed computation. That is, the communication phase can start only when all subpopulations have evolved for a predefined number of generations. After exchanging individuals in the communication phase, all subpopulations evolve independently again. The above procedure operates iteratively until the termination criterion is met.

To implement the above model, a binary n-cube topology is used in which each cube node corresponds to one of the networked computers. In this model, migration happens only between immediate neighbors along different dimensions of the hypercube, and the communication phase is to send a certain number of the best individuals of each subpopulation to substitute the same number of worst individuals of its immediate neighbors at a regular interval. Fig. 1 illustrates the concept of the model and the operations with an example (n = 2).

3.2. Multi-agent framework for distributed EC

The above network-based architecture allows us to run several subpopulations at the same time, each in a corresponding processor, to speed up the process of evolution. Also, through the simple network protocol, communications between different subpopulations can be easily achieved. Though distributing the EC in this way is much cheaper than using a parallel computer, as can be observed, several machines have to be preserved to contribute their computation power for running different subpopulations. This is in fact not practical for a computing laboratory environment in which the computational resources are shared by many end-users.

To exploit the networked computing power and to prevent our EC applications from occupying the public computing resources for a long period of time, we have developed an adaptive agent-based framework to manage the execution and communication for different subpopulations dynamically. From the point of view of system operation, the proposed framework has two major features: distributed system topology with peer-to-peer networking, and software component architecture with agent paradigm. In this way, end-users can access computing resources without extra restrictions. Meanwhile, software agents only take over idle machines to perform EC. In addition, with the agent-based design, some special system characteristics, such as autonomy, adaptivity, and sociality, can be observed from the experiments.
Our agent-based system consists of two parts: the first part mainly includes agents responsible for the computation at the higher (application) level, and the second part includes agents taking care of the communication details at the lower (system) level. For the first part, three types of agent are developed: the mobile agent to execute and carry EC code, the status agent to report machine status, and the synchronization agent to record the evolving progress of subpopulations. Different agents communicate with each other in a prespecified language through a common channel that is compliant with established FIPA standards to ensure the interoperability between agents. In the proposed computation framework, mobile agents play the major roles because they take the responsibility to execute the EC code for different subpopulations, and they have the ability to migrate from machine to machine to achieve the goals. When traveling, a mobile agent first packs its code, data and the running state, and then moves to a destination machine. After arriving at the target machine, the mobile agent continues to execute the code it brought over. Though "code mobility" has been investigated in the research of distributed computing and software engineering, the agent characteristics (especially the autonomy) make our mobile agent-based methodology different from those of cluster-wide process migrations. One main difference between the mobile agents and the most popular Java applets is that mobile agents initiate the migration process, whereas the migration of Java applets is initiated from other software components (e.g., the Web browser). Another difference is that mobile agents usually migrate more than once from host to host, whereas Java applets migrate only from a server to a client, and do not allow the client to migrate to another client or back to the server. Hence, mobile agents can control their own execution, but the lifetime of an applet is bound to that of the application software initiating this applet.

With these unique characteristics, mobile agent-based systems provide a flexible framework to build distributed applications more efficiently, and we adopt such a framework to distribute EC on the networked computers. As mentioned above, in the proposed approach, a mobile agent can only execute its code on an idle machine. To provide machine information, status agents are thus developed to reside in available machines to record the utility information of computing resources, and the mobile agents can then find suitable machines for running the code accordingly. In addition, as indicated above, to perform individual migration within the island model EC, a synchronized method is used to control the evolutionary stages. That is, all subpopulations have to evolve for a predefined number of generations before the communication phase is allowed to happen. A synchronization agent is thus created in the main host to record the evolutionary progress of different subpopulations, and all subpopulations have to wait for each other according to this information. The details of how to dynamically balance the computational load between the machines are described in the following section.

To support the adaptive computing architecture, in the second part we chose JADE as a platform and built the agents described above on it. JADE is compliant with FIPA standard specifications, so the agents developed on it can thus interoperate with other agents built with the same standard. In our integrated computing environment, one of the networked computers plays the role of the "main host" where it is running the JADE main container. Once the platform is activated, the JADE default agents in the main container, including the AMS (Agent Management System), DF (Directory Facilitator), and RMA (Remote Monitoring Agent) are instantiated. The AMS agent exerts supervisory control over access to and use of the agent platform; the DF agent provides the default yellow page service (e.g., to give the IP address of the networked machines) in the platform; and the RMA allows controlling the life cycle of the agent platform and of all the registered agents. From the functional point of view, this platform provides the basic services for distributed peer-to-peer applications in a networked environment. With these default agents, we can ignore the details of the middleware issues on a distributed architecture, and only concentrate on building agents to constitute our distributed evolutionary computing framework to solve application tasks. Fig. 2 illustrates the different levels of the developed computing environment and the overall agent allocation.

3.3. Adaptive load-balance strategy

To enhance the performance of adaptive and distributed computing, we developed a load-balance strategy to work with the above multi-agent approach and to map the multi-agent framework to a networked computing environment. Initially, a status agent is created for each machine involved in the computation, and the status agent (named SA) in the main host is responsible for maintaining the information of individual machine status (e.g., the CPU and memory utility) reported by
other status agents. In our design, each mobile agent takes the responsibility for running a subpopulation. Therefore, after the mobile agent (named MA) starts the computation in the main host, it clones itself with the EC code for each node of the prespecified n-cube model. Then each mobile agent needs to find a machine in reality to perform the relevant computation, according to the information provided by the SA. If there are several machines available for a specific mobile agent, it chooses the one with higher value of resource measurement, which is defined as the product of normalized CPU-idle time (reported by the SA) and CPU speed (machine specification) in this work.

As is indicated, a mobile agent in the proposed framework can only execute its EC code in an idle machine. Therefore, during the period of execution, if a status agent in any machine (except the main host) detects the existence of a new end-user, it informs the mobile agent in the same machine to suspend the execution of its corresponding EC code and to release the computing resources to the end-user. The mobile agent then asks the SA about the machine availability in order to find a free machine to resume the execution. If the mobile agent can find a machine with CPU utility lower than a certain threshold, it asks the SA to reserve this machine, and then carries its code and related information to that machine to continue the computation from the point it is interrupted; otherwise, it keeps staying in the same machine and waits for any available one. Because subpopulations have to wait for each other and will release their computation resources while waiting, eventually each mobile agent can find a free machine for the subpopulation not yet finished. In the worst case when all machines are taken by end-users during the execution, the mobile agents distributed in them can move back to the main host to continue their computation. Fig. 3 illustrates the operations among different agents. In this way, the load balance can be achieved in a dynamic and adaptive way.

In the above load-balance strategy, an ontology is defined in the JADE environment for agent communication. By taking the ontology as reference, agents can create and send messages with consistency in semantics. Fig. 4 shows the ontology, in which AID represents an agent with two attributes, agent name and computational node (i.e., “pc” in the figure). The computational node is an object that has six attributes to provide relevant node information on machine performance (i.e., “performance”), processor speed (i.e., “cpuSpeed”), processor idle rate (i.e., “cpuIdleRate”), remained memory (i.e., “ram”), the number of mobile agent in the node (i.e., “taskAgentNum”), and a JADE container (i.e., “location”). The JADE container (i.e., “ContainerID”) is an object associated with two other attributes, “name” and “address”, and it is responsible for communicating with the default agents to perform system functions at a lower (detailed) level.

3.4. Application tasks

To verify the proposed agent-based methodology, we use it to solve two types of time-consuming applications. The first type is a “high-dimensional optimization” problem, in which the number of parameters to be determined will increases along with the problem scale. A typical example is the modeling of a gene regulatory network (GRN), which is one of the most important issues in systems biology research. The problem is to infer GRNs (i.e., to determine the network parameters) from the expression data collected by a microarray. The details of the modeling procedure can be found in [18, 19].

In Section 4.1, we will use our agent-based approach to solve the problem of GRN modeling as the first example. Our work adopts the widely used S-system model to represent GRNs. In this model, the systematic structure is described as

$$\frac{dX_i}{dt} = \alpha_i \prod_{j=1}^{N} X_i^{g_{ij}} - \beta_i \prod_{j=1}^{N} X_i^{h_{ij}}.$$  

Here, $X_i$ is the expression level of gene $i$ and $N$ is the number of genes in a genetic network. The non-negative parameters $\alpha_i$ and $\beta_i$ are rate constants that indicate the direction of mass flow. The real number exponents $g_{ij}$ and $h_{ij}$ are kinetic orders that reflect the intensity of interaction from gene $j$ to $i$. The above set of parameters defines an S-system model.

To infer an S-system model is to estimate all of the above $2N(N + 1)$ parameters simultaneously. Genetic algorithms have been proposed for parameter estimation, and some promising results have been obtained. However, as can be observed, it becomes difficult to determine the large number of parameters for a GRN when the complexity of regulation increases along
with the number of genes involved. To deal with the scalability problem, a large population size and advanced techniques are needed to work with the traditional GA to improve the search performance, and they inevitably introduce even more computational cost. Therefore, it is important to develop methods to reduce the time for conducting a single experimental run. We will present how this can be achieved by the proposed agent-based approach in Section 4.1.

The second type of application is the one spending large amounts of time in evaluating individuals. A typical example is the work of evolving robot controllers [20]. The process of evolving controllers is similar to that of traditional evolution-based work. It first specifies a fitness function which quantitatively describes the desired robot behavior, and then begins the evolution process: generating an initial population consisting of different control systems; evaluating the control system on a robot to determine the corresponding performance; and applying genetic operators on the current robot population to create a new population according.

Though using evolutionary approaches to develop robot controllers is promising, it is in fact very time consuming. The computational cost of evaluating an individual depends mainly on the following factors: the number of environmental trials in which the robot is evaluated, the number of time steps to evaluate a controller in a single trial, and the time the robot spends to perform a single action. Though using a simulator to evolve controllers can shorten the time of moving a real robot, it also introduces additional computation time in solving the differential equations used to describe the dynamics of robot visual and motion systems. This shows that the time taken to evolve robot controllers can easily increase to a few days when the target tasks become more complicated, and it is thus far beyond what we can afford. The above analysis indicates the importance of distributing EC on different machines to speed up the experiments.
4. Implementation and experimental results

Following the proposed methodology of mapping mobile agent-based framework onto a networked computing environment to distribute EC, we used it to conduct two series of experiments for each of the above applications to investigate the corresponding effects. The first series is to examine the performance of distributed computing, in which a static strategy is used. In contrast, the second series of experiments is to evaluate the performance of using mobile agents with an adaptive load-balancing strategy. The main differences of the two strategies lie in their task context: in the static strategy, the machines for the experimental runs are reserved and all agents (including mobile agents) are defaulted to have no mobility, while, in the dynamic strategy, users are allowed to access the machines arbitrarily and mobile agents can move freely according to the machine availability. The details are described below.

4.1. The first application

4.1.1. Genetic representation, fitness function and operators

The first application is to infer the network parameters for a chosen model so that the model can produce the desired network behavior (i.e., fitting the given time-series data). In the experiments, we used a GA to solve this optimization problem. Because the main purpose of this work was to investigate how to use networked computers to speed up EC (rather than to compare the performance of different GA methods), we thus only implemented the traditional GA to solve this problem and then focused on the analysis of our agent-based distributed computation.

As in other evolution-based methods, the first step in evolving a network model is to define an appropriate representation to characterize the solution. Here we take a direct encoding scheme to represent solutions, in which the network parameters of the S-system model described in Section 3.4 (i.e., \( \alpha_i, \beta_i, g_{ij}, \) and \( h_{ij} \) in the model equation) are arranged as a linear string chromosome of floating numbers.

The next step is to define a fitness function to measure the performance of individuals. Similar to the curve-fitting problem, the goal here is to minimize the accumulated discrepancy between the gene expression data recorded in the dataset (i.e., the desired values) and the values produced by the inferred model (i.e., the actual values). Therefore, the fitness function is defined directly as the mean squared error over the time course as

\[
    f = \frac{1}{N} \sum_{i=1}^{N} \left( \sum_{t=1}^{T} \left( \frac{X^d_i(t) - X^d_i(t)}{X^d_i(t)} \right)^2 \right),
\]

in which \( X^d_i(t) \) is a desired expression level of gene \( i \) at time \( t \), \( X^d_i(t) \) is an actual value obtained from the model, \( N \) is the number of genes in the network, and \( T \) is the number of the data points measured for a gene. A small penalty term measuring the connection between genes can be added to the fitness function to discourage the connections, but it is not used in this work because it is not the main focus here.

With the above fitness function, the individuals can be evaluated and their performance can be determined. Then a certain selection scheme is used to choose parent individuals and some genetic operators are applied on them to create new (children) individuals. In our implementation, the tournament selection scheme is employed to choose parents with better performance for creating new individuals.

Three genetic operators, reproduction, crossover, and mutation, are applied to create a new population for the next generation. The reproduction is simply to duplicate some individuals and to add them to the new population, and the mutation is to randomly generate new values within the specified range for genes. Also, the arithmetical crossover is developed to recombine numerical values of the parent chromosomes. In the arithmetical crossover, a strategy of linear combination is used to create new individuals. That is, if \( s^v_i \) and \( s^w_i \) are the chosen parents to be crossed after the \( k \)-th position, and \( v_i^k \) and \( w_i^k \) are the \( i \)-th genes (after \( k \)) of \( s^v_i \) and \( s^w_i \), respectively, at generation \( t \), then the \( i \)-th genes for the new individuals \( s^v_i^{t+1} \) and \( s^w_i^{t+1} \) are

\[
    v_i^{t+1} = a \cdot v_i^k + (1 - a) \cdot w_i^k, \\
    w_i^{t+1} = a \cdot v_i^k + (1 - a) \cdot w_i^k.
\]

To avoid the problem of producing offspring outside the convex solution space, we use the property of the convex spaces to determine \( a \) as \( f(s^v_i) / (f(s^v_i) + f(s^w_i)) \) and \( a \in [0, 1] \).

4.1.2. Results of different agent-based distributed strategies

The above GA method has been implemented with the agent-based framework to determine the network parameters for two sets of expression data. The first is a popular dataset reported in [21]. The original network for this dataset consists of five nodes, and their relationship can be described as the following:

\[
    \dot{X}_1 = 15.0X_3 X_5^{0.1} - 10.0X_1^{2.0} \\
    \dot{X}_2 = 10.0X_4^{2.0} - 10.0X_2^{2.0}
\]
\[
\dot{X}_1 = 10.0X_2^{-0.1} - 10.0X_2^{-0.1}X_3^{2.0}
\]
\[
\dot{X}_4 = 8.0X_1^{2.0}X_5^{-0.1} - 10.0X_4^{2.0}
\]
\[
\dot{X}_5 = 10.0X_4^{2.0} - 10.0X_5^{2.0}.
\]

With the time-series data generated from the original network model (including 30 time points), the GA was then used to construct the original network model (i.e., to determine the parameters) reversely from the data. For this task of five gene nodes, there were 60 network parameters in total to be determined. In this experiment, a population of 800 individuals was used, and the evolution was continued for 1000 generations. Fig. 5 presents the network behavior in a successful run (i.e., with an error less than 0.1), in which the x-axis represents the time point, and the y-axis is the concentration of the five gene nodes. This shows that the developed GA can be used to evolve the GRN successfully.

To evaluate the performance of our agent-based approach, we conducted two series of experiments, for static and dynamic strategies, respectively. In the first series, different numbers of subpopulations were used with a static strategy, including one population of 800 individuals, two populations of 400 individuals, four populations of 200 individuals, and eight populations of 100 individuals. To support distributed computation, the coarse-grain island model was used, and the model was configured as a binary \( n \)-cube. Each cube node represented a subpopulation, and a mobile agent was created to perform the relevant computation. In the experiments, the communication phase happened every ten generations, and the number of migrants was 4% of a subpopulation. These values were chosen because they were found to give the best performance in a small pilot study.

As described in Section 3, to realize distributed EC on networked computers, our agent-based system has been developed using the basic features supplied by the JADE platform. In the experiments, we arranged eight networked computers as a distributed computing environment to execute the JADE platform. Each computer has a Pentium 4 specification, including a 2.8 GHz CPU and 2 GB RAM, and it ran Windows NT operating system. Different computers were connected through a 100 Mbps local area network. Among the eight computers, one played the role of the “main host” where it was running the JADE main container. Once the platform was activated, the JADE default agents AMS, DF, and RMA were instantiated, in which the DF agent provided other agents with the information (e.g., the IP address) about the hosts connected to the JADE platform. Each machine in this framework had a status agent to report its corresponding status, and a mobile agent to take care of the computation for each subpopulation. Also, a synchronization agent was created in the main container to activate the communication phase for exchanging individuals between subpopulations. Fig. 6 illustrates the arrangement.

Because this series of experiments was mainly to measure how the developed agent-based framework can be used to exploit the networked computing power to speed up the EC, the computing environment had to remain static. Hence, the end-users were not allowed to access the machines used for the experiments. The agents also remained static; they were not allowed to migrate between different machines. That is, the mobile agents here were created only to communicate with agents on other machines—the major task was to send and to receive individuals during the communication phase. As mentioned above, criteria of different numbers of subpopulations have been used, and the results were compared. To be more objective, ten independent runs were conducted for each criterion. Fig. 7 (the upper part) shows the average computational time (over the ten runs) spent for each criterion. In this figure, the first data point (i.e., 374.98 s) of the “ideal” set indicates the time required for running the single population GA (without using any agent) to obtain a solution, and the other three data points of this set represent the ideal situations (theoretically) for speeding up the experiments (the time needed is 1/2, 1/4, and 1/8 of the original run) with more machines (\( m \) is the number of machines used). The lower part of Fig. 7 indicates how the experiments are speeded up. As can be observed in Fig. 7, in the real situation, some extra time is inevitably needed for initiating and running the multi-agent framework. Therefore, the results for the static agent-based strategy are thus not as perfect as in the ideal situation. Nevertheless, the time for running a single experiment can be largely reduced, depending on the number of computers used in the experiments. This shows the efficiency of our approach.
In contrast to the above static strategy, the second series of experiments was to show how our mobile agent approach can support distributed computation in an adaptive manner. In this series of experiments, different numbers of machines were connected to the JADE platform in which only one host was preserved for the experiments. The preserved host was initiated as the main container to enable the default agents AMS, DF, and RMA, and to run the experiment. Here, the end-users were allowed to use other computers as they usually do. As in the first series of experiments, each host was a container that included a mobile agent and a status agent. Also, a synchronization agent was allocated in the main container. Initially, the GA code was executed on the main container, and then the mobile agent on this host checked with the status agent to find other available hosts. For any free host, a duplicated mobile agent packed the GA code and relevant information, moved to the target host, and started the execution for a new subpopulation. After that, the mobile agents moved among different hosts according to the load-balancing strategy described in Section 3, to execute the corresponding GA code for the application task.
Fig. 8. A part of one experimental run that shows that the agents moved autonomously among different computational nodes (left figure) according to the machine availability. The right figure indicates the resource utility of node 3.

To compare the performance of dynamic and static strategies, experiments were conducted on 2, 4, and 8 machines as in the first series of experiments. In the preliminary test, we noticed that the time required for running a single experiment in this application was in fact not long (i.e., less than 10 min), and consequently it became difficult to collect experimental results with users sharing the computational resources arbitrarily. Therefore, in the experiments we randomly blocked machines for several short periods of time (10, 20, or 30 s) during each single run to simulate the situation of resource sharing. The results for this dynamic agent-based strategy are also shown in Fig. 7, in which each data point is obtained from ten runs. As can be seen, all runs have been speeded up by the proposed approach. This figure indicates that the cost of the adaptive agent-based distributed computation is higher than that of the static strategy presented above. This is due to the fact that the performance of the dynamic strategy depends on the real situation of resource sharing in which the end-users are allowed to access the machines at any time. Although the dynamic strategy cannot speed up the experiments in a linear manner, it is more practical to realize distributed EC in an ordinary networked computing environment.

As can be observed in the experiments, the proposed agent-based system has presented some special system characteristics mentioned in Section 3.2, in which the most notable is the agent autonomy. Autonomy is generally considered as the degree of freedom from external control. Here, the mobile agents can sense the machine condition (availability) reported by status agents and decide how to move from machine to machine to achieve their goal of performing computation. The agents have shown a certain level of autonomy because they made decisions without the supervision of human operators in a dynamic environment (the users were allowed to access the computers in the second series of experiments). Fig. 8 shows a part of one experimental run to illustrate the agent autonomy. In this example, three mobile agents resided in the same machine (node 1), and two agents moved to other machines (nodes 2 and 3) when they were available. After the users started to use nodes 2 and 3 at time units 4 and 5 (indicated as “end user”) respectively, the agents moved back to node 1 and continued their computation.

In addition to the above dataset, we also prepared a new dataset to evaluate the performance of the proposed methodology. The second dataset is an artificial dataset collected from the popular GRN simulation software Genexp [22]. An 8-node gene network was defined (that is, 144 network parameters to be determined) and the simulation was run for 30 time steps for data collection. Different numbers of subpopulations mentioned above were used to infer the original network model reversely from the data. For each criterion, ten independent runs were conducted in which the experimental settings were the same as in the runs for the first dataset. Fig. 9 shows the network behavior obtained from a successful run, and
Fig. 9. The behavior of the evolved gene network for the second dataset.

Fig. 10. Computational cost and the speeding-up situations for different experimental settings (m is the number of computers).

Fig. 10 lists the computational time and the speeding-up situations for different experimental settings. As can be seen, similar to the results for the first dataset, both static and dynamic strategies can efficiently reduce the experimental time. This is very important for EC as it is a population-based approach and involves evaluation of a large number of solutions.

4.2. The second application

4.2.1. Genetic representation and operations

The second application is a time-consuming task of evolving controllers, in which evaluations for individuals take a large amount of time. For this application, we employed a genetic programming (GP; see [23]) system to evolve the controllers. GP is an extension of traditional genetic algorithms with the basic distinction that, in GP, the individuals are dynamic tree structures rather than fixed-length vectors. GP aims to evolve dynamic and executable structures often interpreted as computer programs to solve problems without explicit programming. As in computer programming, a tree structure in GP is constituted by a set of functions (sometimes called non-terminals) as the internal nodes of the trees, and by a set of terminals as the external nodes (leaves) of the trees. The construction of a tree is based on the syntactical rules that extend a root node to appropriate symbols (i.e., functions/terminals), and each function is extended again by suitable rules accordingly, until all the branches in a tree end up with terminals. Hence, the first step in employing GP to solve a problem
is to define appropriate functions, terminals, and the syntactical rules associated for the program development. The search space in GP is the space of all possible tree structures composed of functions and terminals.

Because the main goal of this work was to show how our approach can improve the performance of EC, we used a simulated robot in the experiments. In our simulation, a robot is assumed to have a round physical with eight infrared sensors (i.e., IRs) positioned around the body pointing radially outwards, and they can function within a certain distance and a certain bearing. Fig. 11(a) illustrates the sensor arrangement. The motion system of the robot is regarded as a process with natural dynamics, and is modeled by a set of first-order differential equations.

To evolve the controllers, we defined the terminal set as \{IR1, IR2, ..., IR8, R\}, and the function set as \{PROG, AND, OR, NOT, >\}. In the terminal set, "IR1", "IR2", ..., "IR8" were defined to represent the available sensors on the robot, and they gave the normalized sensor responses. The symbol "R" was defined to represent the set of possible numerical constants between 0 and 1 inclusively. Whenever a terminal symbol R appears in the tree creation procedure, a random number is generated to associate with R accordingly. For the function set, the symbol "PROG" was defined as a dummy node, only for convenient manipulation by a GP system. The logic components "AND", "OR", and "NOT" were defined to constitute the main frame of the controller to combine sensor information to form a controller. Each logic component evaluates its argument(s), performs its function, and returns a Boolean value to the function symbol (i.e., the parent node) on the top. Finally, the comparator ">" was defined to construct perception information (obtained from the sensors defined above) into sensor conditionals. According to our design, a sensor conditional has a constrained syntactic structure; it existed in the form of X>Y, where X and Y must be any normalized sensor response or numerical threshold determined genetically. For example, a sensor conditional IR3>IR5 will return the true/false result after comparing the normalized sensor readings from sensor IR3 and sensor IR5. Fig. 11(b) illustrates an example of a controller. In such a structure, the evaluation results of the subtrees immediately below the root node were interpreted as motor commands to drive actuators: the first and second subtrees are for the left motor, and the third and fourth subtrees are for the right motor. The motor commands are then converted into actual motions.

In our GP, three genetic operators (reproduction, crossover, and mutation) are used to create new tree individuals. Reproduction simply copies the original parent tree to the next generation; crossover randomly swaps subtrees for two parents to generate two new trees; and mutation randomly regenerates a subtree for the original parent to create a new individual. As the tree structure has been specially designed, when the crossover is performed, all syntactic constraints must be satisfied to guarantee the correctness of new trees.

The next step is to evaluate tree individuals to determine their fitness for the creation of a new population. This is normally done by predefining a fitness function that quantitatively describes the requirements of a target task first, and then by executing the corresponding codes for tree individuals in the environment of the particular problem. In the experiments, we used our GP system to evolve robot controllers to achieve an obstacle avoidance task as an example. The obstacle avoidance task requires the robot to move forward as straight as possible but not bump into any obstacle. Therefore, we defined the fitness function as a penalty function, which is the linear combination of the of the sensor and motor activities:

$$
\sum_{j \in \text{Cases}} \sum_{t=1}^{T} (\alpha \cdot \text{response}_{\text{max}}(t) + \beta \cdot (1 - v(t)) + \gamma \cdot w(t)).
$$

In the above equation, Cases is the set of test trials the robot is situated in; T is the number of time steps of a single trial; \(\text{response}_{\text{max}}(t)\) is the largest IR response at current time step \(t\); \(v(t)\) is the normalized forward speed; \(w(t)\) is the absolute value of the normalized rotation speed; and \(\alpha, \beta, \gamma\) are the weights expressing the relative importance of the above three factors.

### 4.2.2. Results of different agent-based distributed strategies

As in the first application, we conducted two series of experiments with static and dynamic strategies respectively to access the performance of our agent-based methodology. In the first series, we used multiple subpopulations with a static strategy to evolve controllers, including one population of 400 individuals, two populations of 200 individuals, four populations of 100 individuals, and eight populations of 50 individuals. The binary \(n\)-cube island model was used, and
Fig. 12. The evolved robot behavior.

each cube node was responsible for a subpopulation. In the experiments, the communication phase happened every ten generations, and the number of migrants to be exchanged was 4% of a subpopulation, as before. The GP system described above was developed to evolve controllers, and the typical behavior of a successful controller is shown in Fig. 12. It shows that our GP method is able to evolve controllers successfully.

Again, for this application, two series of experiments were conducted. The first series was to investigate whether the proposed agent-based framework can effectively speed up the EC on networked computers. Hence, the agents remained static, and the mobile agents were created only to send and to receive individuals during the communication phase. To compare the effect of using different numbers of computers, the above configurations of multiple subpopulations were used in the experiments. Each subpopulation was executed on one computer, and the end-users were not allowed to access the machines used for the experiments. As an individual in GP is a different size tree, the times needed for evaluating different individuals were not the same. Therefore, for each criterion, five independent runs were conducted. The computational time (averaged from the five runs) spent for each criterion is listed in Fig. 13 (the upper part). The speeding-up situations are also listed in the lower part of this figure for comparison. As can be observed, compared to the first application, the time for running a single experiment here is much closer to the ideal situation—it is reduced in an almost linear manner. This is because in this application the extra computational effort needed for agent communication becomes relatively small compared to the time for running a time-consuming application of evolving controllers.

After evaluating the performance of the static strategy, the second series of experiments was to examine how the proposed mobile agent approach can achieve adaptive distributed computation in a dynamic networking environment. Here, we did not repeat the experiments with different numbers of machines as before, but instead connected eight computers to the JADE platform for performance evaluation. In the experiments, only one host was preserved for running GP experiments. The preserved host was initiated as the main container, and end-users were allowed to use the other seven computers. Each host was a container that included a mobile agent and a status agent, and a synchronization agent was allocated in the main container. As in the first application, for any free host, a duplicated mobile agent packed the GP code and relevant information, moved to the target host, and started the execution for a new subpopulation. Then the mobile agents followed the presented load-balancing strategy to move among different computers to execute its GP code.

Because running a single experiment in this application takes much longer time than in the first application, we are now able to collect results for the real situations of resource sharing, in which the end-users were allowed to freely use the seven machines connected to the mobile agent framework. Due to different user-accessing situations, the experiments of adaptive distributed computation were conducted six times on different days. Fig. 13 summarizes the results. In this figure, \( m \) is the number of computers used in the experiments, and \( d_1 \)–\( d_6 \) are the indices for the days using eight computers with a dynamic strategy. It shows that all runs have been speeded up, and verifies the efficiency of our mobile agent-based approach. As in the first application, though the computational cost of a dynamic strategy is higher than that of static strategy presented above and the experiments with a dynamic strategy cannot be speeded up in a linear manner, the dynamic strategy is nevertheless a more practical way to achieve distributed computation in reality.

5. Conclusions and future work

In this paper, we have indicated the importance of using parallel models to speed up EC. Instead of using expensive parallel computing facilities or unreliable public computing architecture, we also proposed mapping the parallel models onto easily available networked PCs to realize the parallelism. We then presented a multi-agent framework to dynamically distribute EC on different machines. With many unique agent characteristics, mobile agents play the kernel roles in our framework to manage the execution and communication for the EC code. They control their own actions to carry the code to be executed toward available computing resources. To evaluate the proposed framework, different sets of experiments were conducted to solve two types of time-consuming applications. For each type of application, we first employed a
static strategy with our multi-agent framework to exploit the computational power of networked computers to support distributed EC. Then we used an adaptive mobile agent strategy to dynamically execute EC code on different machines, but not to compete resources with end-users. The results show that both static and adaptive methods can efficiently speed up the computation, while the dynamic strategy is a more practical choice in reality.

Regarding our implementations and experiments of agent-based distributed computation, many agent characteristics can be observed, though the goal-oriented strategy has been predefined mathematically and the dynamic motions of the agents are simple. First of all, the mobile agents have a certain level of autonomy. They are able to achieve their own goals without any user intervention. They also have mobility to move among different machines to execute the EC code. As the agents can sense the machine conditions and use the conditions to adjust their actions accordingly, they have some adaptivity and reactivity. In addition, the agents communicate with each other to send and receive information by the specially designed method and cooperate to achieve the goal. They are thus considered to have some kind of interactivity and sociality. With the above characteristics as other intelligent agents in real-world applications, the proposed mobile agent-based approach has shown its potential toward heavy computation in a large networking environment.

Based on the work presented, we are currently addressing the issues of system reliability and fault tolerance. We are designing a mechanism to ensure that our framework can recover from unexpected faults caused by the hardware defects or inappropriate user operations. From the perspective of agent-based computing, we plan to develop a learning method for our agents so that they can learn to recover from different system faults according to their experiences. Also, we will investigate whether the proposed approach can be used to include more machines to solve more difficult problems. In particular, with the mechanism to deal with uncertainty and unreliability problems, we intend to include external machines through the Internet to conduct large-scale computation.

Acknowledgement

This work has been supported by the National Science Council of Taiwan, under contract NSC-97-2221-E-110-063-MY2.

References


