Proceedings of the 2007 IEEE Swarm Intelligence Symposium (SIS 2007)

Co-ordination in Intelligent Ant-Based Application Service Mapping in Grid Environments

Sharath Babu Musunoori
SIMULA Research Laboratory
P.O.Box 134, 1325 Lysaker, Norway
sharath@simula.no

Geir Horn
SINTEF ICT
P.O.Box 124, 0314 Oslo, Norway
Geir.Horn@sintef.no

Abstract— **A key problem of component-based grid application configuration is to map services onto the execution nodes of the grid environment such that all services of the application satisfy some minimum quality requirements. This problem is known to be NP-hard. This paper presents two extensions to our previous ant-based application service mapping heuristic, in order to establish some coordination among the working agents in a decentralized environment and improve its convergence rate. The first extension proposes to use gatekeepers that learn to guide the movement of intelligent foraging ants. The second extension is a coordination mechanism to allow an ant to draw on the learned knowledge of its co-workers. The simulation evaluation of the proposed extensions shows that inclusion of scaled sleep time further improves the convergence rates while the gatekeepers seem to have a negative effect.**

Keywords: Service Configuration, Mapping, Partitioning, Learning Automata.

I. Introduction

Computational grids are widely accepted as future platforms for parallel and distributed systems, since grid middleware aims to solve problems of resource sharing and management across organizational domains. However, achieving acceptable performance in a computational grid environment remains a difficult engineering challenge. On the other hand, a range of distributed applications can be understood as a composition of services [2]. The useful application experienced by the user is consequently a service graph with corresponding communication among the services. Besides an overall functional behavior of an application, its performance is estimated as a measure of the non-functional (quality) behavior of services within in the application. For this, services in this distributed application need to acquire a set of resources (e.g. memory, CPU, disk, input/output) such that they can perform their functional task and contribute to the overall application performance. This is particularly relevant to a broad class of applications that require fulfilment of some quality requirements in order to satisfy users' needs. Such concerns occur not only in real-time and multimedia applications, but also in any application where a user is waiting for output. A grid node can thus be seen as a *capsule* holding and executing a subset of an application's services. In the physical world there will be connections among the capsules with certain network properties like bandwidth, transmission delays and packet jitter.

Consequently, the fundamental problem addressed here is the configuration of an application for a dynamic grid environment. This is the problem of partitioning in a stochastic environment: *How to split the application's service set* $\mathbb{S}$ *onto a set of heterogeneous capsules* $\mathbb{C}$*, each offering a set of resources to the running services, such that all services get sufficient resources to allow them to achieve at least the minimum quality to be useful for the application and no capsule is oversubscribed with respect to its resources?*

The optimal service configuration is the one that simultaneously tries to maximize the quality levels for all services from the given set of available resources of the grid environment. However, examining the whole solution space is impossible since an exhaustive search through all possible configurations in the solution space takes too long. Furthermore, computational and communication resources in the grid environment may join and leave very often. Thus, the objective is to seek a *feasible* solution instead of the optimal solution. A feasible service configuration is the one that tries to achieve at least minimum required quality levels for all services from the given set of resources.

Despite the success of various heuristics in solving the combinatorial optimization problems like the one at hand, they do not scale and fail to deliver for large systems since they often need full control over the system in order to make global decisions. Additionally, these methods may not live up to their expectations if the solution space has stochastic behavior. One natural extension to this is to model a system where several individual entities make appropriate decisions that will eventually lead to a global convergence. Such an extension may create a decentralized environment for decision making where all agents coordinate with each other to generate a global behavior for the system.

The ant system (AS) was inspired by the natural behavior of social elements (a colony of ants or wasps) of an ecosystem. With an objective of completing a given task, each ant walks within the environment (usually a graph) and updates its objective function while cooperating with other ants working for the same purpose [1]. Such a foraging behavior of multiple ants create a collective intelligence, which is capable in nature to generate useful solutions. The present authors have previously shown how the metaphor can be applied to the problem at hand [4]. In order to improve convergence on a solution the ants were made intelligent by the use of *learning automata* in [5]. The use of learning automata for this problem

is motivated by the fact that the fastest known equipartitioning algorithm is based on a fixed-structure stochastic learning automata (FSSA) [8]. Recently learning automata have also been successfully used to map the processes (services) of an application onto processing nodes [3] [6], and that problem resembles the one at hand.

In order to improve the convergence beyond what was achievable with learning ants in [5], this paper proposes two conceptual enhancements to the previous works:

- *Gatekeepers*: *Learning Automata* [7] whose job is to advice the ants issued by a service what will be the best next capsule for their movements. This is a *static* intelligence in the capsules to complement the *dynamic* or moving intelligence of the ants. The gatekeepers are described in Section III.
- *Ant-to-ant co-ordination*: A drawback of the previously cited approaches to solve this problem with an ant system is that the ants were acting in complete isolation. This contrasts the natural approach where real ants do exchange information about what is the better location of food. Thus, a co-ordination mechanism is proposed that obeys the requirement of no global intelligence and individually acting ants, still exploring the knowledge collectively *learned* by the ants. The mechanism is described in Section IV.

The algorithms have been tested and the results are analyzed through simulations on randomly generated sets of application services and grid environments, see Section VI. The analysis in Section VII of the results obtained from simulations shows that the proposed methods provide viable solutions to the complex problem of application service partitioning in grid-like environments.

II. Services and Their Ants

A. The Ants

Inspired by the basic principles of ant colony [1] and learning automata [7] [8], an ant-based service partitioning logic has been designed [5]: each service employs several intelligent *ants*. Each ant is an agent that acts as a proxy for the service it has been created for. The ant actually move in a random walk among the capsules and check the quality levels of its service where it to be deployed in the capsule currently hosting the ant. Let $\mathbb{A}_{S_i} = \{A_{S_i,1}, A_{S_i,2}, \ldots, A_{S_i,|\mathbb{A}_{S_i}|}\}$ be a set of ants created for a service S_i. Similarly let $\mathbb{A} = \{\mathbb{A}_{S_1}, \mathbb{A}_{S_2}, \ldots, \mathbb{A}_{S_{|\mathbb{S}|}}\}$ be the set of all ants created for all services in $\mathbb{S}$ respectively. When released all ants take a step before they update their service quality on the destination capsule. Afterwards, the ants will continue to move until all services in $\mathbb{S}$ find a suitable place.

To terminate an unsuccessful search, the ants are assumed to die after some time according to the biological system inspiring this work. Thus, a death rate $d \in \langle 0,1 \rangle$ is associated with every move an ant does. In this way the number of surviving ants will be binomially distributed with survival probability $1-d$, and the expected number of ants from service S_i alive after k moves is

$$|\mathbb{A}_{S_i}(k)| \quad = \quad |\mathbb{A}_{S_i}(0)|\,(1-d)^k \tag{1}$$

Given that only a fraction ξ of the ants should survive $2|\mathbb{C}|$ iterations, one has

$$d \quad = \quad 1 - \exp\left(\frac{\log(\xi)}{2|\mathbb{C}|}\right) \tag{2}$$

When an ant arrives in a capsule, the perception or happiness of the ant is measured by its service quality, given the current location of all the other application services. If the ant is happy with its current placement it sleeps in the current capsule for a certain time period, $t_{S_i}(k)$ before it continues to move further. The happiness for an ant is defined as the average distance between actual and minimum qualities, i.e., $Q_{S_i,j} - Q^{min}_{S_i,j}$ for all quality dimensions $j = 1, \ldots, |\mathbb{Q}_i|$. These distances are scaled to their $[Q^{min}_{S_i,j}, Q^{max}_{S_i,j}]$ interval. The sleeping time after k moves, $t_{S_i}(k)$, for an ant of service S_i is defined as:

$$t_{S_i}(k) = \begin{cases} -1 & : \quad Q_{S_i,j} < Q^{min}_{S_i,j} \\ \frac{|\mathbb{S}|*|\mathbb{A}_{S_i}(k)|}{|\mathbb{Q}_i|} \sum_{j=1}^{|\mathbb{Q}_i|} \frac{Q_{S_i,j} - Q^{min}_{S_i,j}}{Q^{max}_{S_i,j} - Q^{min}_{S_i,j}} & : \quad Otherwise \end{cases} \tag{3}$$

Any positive value of $t_{S_i}(k)$ means the capsule has enough resources to host its service.

The ants can make an intelligent decision whether to stay with its present capsule or move to another [5]. The ant re-evaluates the quality when it wakes up, and if the capsule still has the capacity to host its service, there is no reason for it to move around. In this case the ant has become more *confident* that the current capsule is the best one for its service. In other words, each ant has a confidence level counter counting the number of times it goes to sleep in the same capsule. In the same way, when an ant wakes up and finds that the capsule no longer is able to satisfy its service, its confidence in the capsule is reduced. However, this does not imply that the ant should move because the unfavorable interim conditions could be caused by the current time variate service configuration. The ant should move only if its confidence is reduced to zero, otherwise it should hold on to the capsule a little longer and move to the end of the ant execution list without sleeping. In this way the other ants get the opportunity to move and change the service configuration before the ant re-evaluates the situation.

B. The Services

The sleeping times of individual ants force them to flock onto one or more capsules capable of hosting their service. This allows a service to detect the "best" capsule based on location and confidence of its ants. The services themselves move based on their ants' confidence. In other words, a service is said to be present in the capsule or the computing node with the maximum total confidence of its ants.

There is a question whether to count the confidence left behind by the dead ants or not. In [5], the confidence gained by the dead ants was also considered while calculating the total

confidence of local ants for a service placement. Positively one can see this as the dying ants leave their confidence behind for their co-workers. In other words, already learned knowledge of a dead ant can still be useful for its service to get a consistent impression of the environment.

However, this is not necessarily true because if one capsule is optimal for some time for a service before another capsule becomes optimal, it is likely that the ants owned by the service flock on to the first capsule and consequently the confidence of some of these ants will continue to exist in the first capsule in the form of dead ants. In such case, the service will continue to select the first capsule until the total confidence of its ants on the second capsule is increased sufficiently for the second capsule to be chosen. In this respect confidence gained by the dead ants may not help the services in finding better places, instead this may force the services to be biased towards one or more capsules resulting in low convergence rate. To avoid this situation, the dead ants' confidence should be omitted or should be reduced over the time. One innovation of this paper is thus not to include the dead ants when computing the confidence of the capsules.

III. Gatekeepers: Guiding the agents

The job of the gatekeeper is to advice the ants of a service what will be the best next capsule. For this it keeps a probability vector the possible capsules and in the worst case of a complete capsule graph this vector has length $|\mathbb{C}|$. Thus, a gatekeeper is a variable structure stochastic automaton [7] defined by a probability vector $\mathbf{p}(t) = \big(p_1(t), \ldots, p_{|\mathbb{C}|}(t)\big)$ with $p_i(t)$ being the probability that capsule $i \in \{1, \ldots, |\mathbb{C}|\}$ will be recommended as the next capsule for an ant to make a move at time t.

Initially each service creates $|\mathbb{C}|$ gatekeepers and locates them on each of the $|\mathbb{C}|$ capsules respectively. When an ant issued by a particular service is to move from a capsule, it interacts with gatekeeper of its service on that capsule. The gatekeeper then proposes a destination capsule based on the probabilities in its action probability vector.

The basic functionality of *Learning Automata* (LA) can be described in terms of a sequence of repetitive feedback cycles in which the automaton interacts with the environment. During a cycle, the automaton chooses an action, which triggers a response from the environment, a response that can be either a reward or a penalty. The automaton uses this response and the knowledge acquired from previous actions to determine its next action. By learning to choose the optimal action, the automaton adapts itself to the environment.

Fundamentally, in the terminology of the learning automata theory, the gatekeeper gets a *feedback* from the ant when it has computed the quality of its service at the new capsule. If the ant finds "food", the gatekeeper is *rewarded* for its recommendation. On the other hand, if the capsule is not good for the service owning the ant, the gatekeeper is *penalized*.

In this work the gatekeepers are modelled as *Linear Reward-Penalty automata* (L_{R-P}). Given that the environment is highly non-stationary because the services move with the ants, the action probability vector should be able to change over time. Using this scheme, the possible capsules to be recommended are called "actions" whose probability vector $\mathbf{p}(t)$ is changed on both a positive feedback and a negative feedback. On a favorable response for a chosen action i, a small portion is reduced from each action probability value $p_j(t)$ for all $j \neq i$ and the removed probability mass is added to the current action probability value $p_i(t)$:

$$\begin{aligned} p_j(t+1) &= \lambda p_j(t) \qquad \forall \quad j \neq i \\ p_i(t+1) &= 1 - \sum_{j \neq i}^{|\mathbb{C}|} p_j(t+1) \\ &= 1 - \lambda \sum_{j \neq i}^{|\mathbb{C}|} p_j(t) \\ &= 1 - \lambda(1 - p_i(t)) \\ &= \lambda p_i(t) + (1 - \lambda) \end{aligned}$$

Note that the first update is done for all the elements of the probability vector corresponding to the capsules not recommended. Since the vector $\mathbf{p}(t)$ already was a probability vector, $\sum_{j \neq i}^{|\mathbb{C}|} p_j(t) = 1 - p_i(t)$, enabling the update for probability of the recommended capsule to be simplified.

The value $0 < \lambda < 1$ is a learning parameter, for which a small value leads to fast convergence and high variance, while a larger value slows the convergence and lowers the variance.

When there is a negative feedback the gatekeeper only knows which recommendation was *not* a good one, but not which of the other recommendations that would have been better and should have increased probability. The updating scheme will in this case reduce the probability of the action that was penalized, and distribute the removed probability mass onto the other actions. One only needs to ensure that the probability vector remains a probability vector, hence

$$\begin{aligned} 1 &= \sum_{m=1}^{|\mathbb{C}|} p_m(k+1) = \lambda p_j(k) + \sum_{m \neq j}^{|\mathbb{C}|} (\lambda p_m(k) + c) \\ 1 &= \left(\lambda \sum_{m=1}^{|\mathbb{C}|} p_m(k)\right) + \left(\sum_{m \neq j}^{|\mathbb{C}|} c\right) \\ 1 &= \lambda + (|\mathbb{C}| - 1)c \end{aligned}$$

and solving for the constant c gives

$$c = \frac{1 - \lambda}{|\mathbb{C}| - 1}$$

Thus the L_{R-P} automaton responds to the unfavorable responses from the environment for recommending capsule i by updating its action probability values as follows:

$$\begin{aligned} p_j(t+1) &= p_j(t) + \frac{1 - \lambda}{|\mathbb{C}| - 1} \forall j \neq i \\ p_i(t+1) &= \lambda p_i(t) \end{aligned} \tag{4}$$

Given an abstract situation where the reward and penalties are generated by an unknown stationary probability distribution, the action probabilities estimated by the L_{R-P} automaton will not be identical to the true underlying probability

distribution. This is strengthened by the fact that the L_{R-P} automaton is an ergodic scheme that will never converge to any fixed set of probabilities, but have an ever changing action probability vector. However, the interest here is not in the rigorous estimates as such, but in the capability of the gatekeeper to guide the ants. For this the action probabilities of the L_{R-P} automaton may be sufficient.

IV. Co-ordination of the Agents

One desirable property of the proposed system is that the ants (agents) work without any central intelligence. It is however likely that an ant may be more successful in its search if it is able to exploit information also from other ants, like real ants do in nature when meeting fellow workers on the path. This section proposes a simple co-ordination mechanism by enhancing the already introduced concepts. The fundamental idea is to allow the ants of less confident services to move more around than the ants representing more confident services. Here the confidence of a service is understood as the total confidence of its ants.

Referring to the Section II-A, each arriving ant evaluates the quality achievable by its service on the capsule and if this value exceeds the minimum quality required by the service the ant goes to sleep in that capsule. Recall that the ant sleeps for a time $t_{S_i}(k)$ proportional to the quality obtained. Except creating randomness in the execution order of intelligent ants, this does not help to establish any communication among the agents that are expected to coordinate with each other.

Therefore it is proposed to use a scaling factor which allows an ant to sleep as a function of the confidence of its co-workers. Along with its current sleep time measured by (3) the final scaled sleep time for an ant of a service S_i after k moves:

$$T_{S_i}(k) \quad = \quad (f_{S_i}(k)+1)\,t_{S_i}(k) \tag{5}$$

Here $f_{S_i}(k)$ is a scaling factor which represents the communication among the moving ants of a service. The scaling factor could be any monotonically increasing function of the confidence level attained by all the ants of a service. In other words, each ant estimates the scaling factor as a function of the confidence levels of its co-workers so that the ant can place itself in an appropriate place within the execution queue. For a higher value of the scaling factor, an ant is expected to sleep for a longer time. In principle this should allow the ants with zero confidence, and also the ants with less group confidence, to move to the front of the ant execution queue and make more moves than their more confident colleagues.

A simple linear scaling factor for an ant $A_{S_i,m}$ of a service S_i can be defined as

$$f_{S_i,m}(k) \quad = \quad \frac{\sum_{j\neq m}^{|\mathbb{A}_{S_i}(k)|} c_j}{(|\mathbb{A}_{s_i}(k)|-1)\,c_{max}} \tag{6}$$

Thus, the scaling factor calculated for an ant is a ratio between the total confidence of all remaining ants of a service and the maximum total confidence achievable by all remaining ants of the service. This ratio is a value between zero and one. But if there is a large number of ants working for their services, the above defined linear function may not be efficient enough to discriminate the execution order for the different ants.

Therefore it is suggested to use a non-linear function to derive an appropriate scaling factor. For example, one can use the following simple non-linear function assuming that this will influence the overall convergence performance of the ant-based algorithm.

$$f_{S_i,m}(k) \quad = \quad log\left(\sum_{j\neq m}^{|\mathbb{A}_{S_i}(k)|} c_j\right) \tag{7}$$

V. The algorithm

The final algorithm consists of two parts derived from the above discussion:

- A subroutine for the functional behavior of an intelligent ant shown in Figure 1. Observe that step 8 of this subroutine has two versions. One that will be used when there are no gatekeepers, i.e. the next capsule to move to has to be selected completely at random. The second variant is for the case where there is a gatekeeper advising the next capsule. In this case the ant will move to the capsule, check if there is sufficient quality for its service on that capsule and provide the necessary feedback to the gatekeeper.
- The overall algorithm of Figure 2. The main loop of the algorithm takes the first ant out of the time sorted execution queue and gives this ant the possibility to execute according to the above sub-routine for the ants. After the ant has moved, its service will evaluate the situation and move to a new capsule if there is a change in the capsule with the maximum confidence level. This loop continues until either all ants have died, a non-convergence, or a feasible solution to the partitioning problem has been found, i.e. the system has converged.

VI. Simulation

Before we continue to test the above described ant-based algorithm for a set of services, $\mathbb{S}$ in a grid setting with $|\mathbb{C}|$ computational nodes, it is important to make sure that there is at least one feasible configuration such that all services are satisfied with their minimum quality requirements. For this we first generate the grid computational environment with random amount of resources. Then the minimum quality requirements of all services are set such that there is a feasible configuration $\mathbb{\Theta}$. The feasible service configuration thus generated here, will fully exploit the available communication and computational resources of the grid environment. More details on how this was done can be found in [5].

The generation of an ordered feasible solution $\mathbb{\Theta}$ is necessary because the proposed algorithm will never converge if the service configuration problem has no solution. In order to test the applicability of the proposed algorithm for the service configuration problem, it should be invoked from a random starting configuration. This is constructed by first redistributing

Parameters
A service, S_i; Death rate, d.
Initialization
Nest of an ant is the host capsule of its service: Nest($A_{S_i,j}$)=HostCapsule(S_i) $\forall\ j \in [1, |\mathbb{A}_{S_i}|]$.
1: **begin** INTELLIGENT ANT
2: Calculate quality ($Q_{S_i,j}\ \forall\ j$) for its service S_i.
3: **if** ($Q_{S_i,j} \geq Q^{min}_{S_i,j}\ \forall\ j$)
4: Increase Confidence.
5: **else**
6: Decrease Confidence.
7: **if** (Confidence level is zero)
8: Move to a capsule C_n selected randomly, $C_n \in \mathbb{C}$.
OR
8A. Move to a capsule C_n selected by gatekeeper on the current capsule C_m.
8B. Calculate service quality ($Q_{S_i,j}\ \forall\ j$).
8C. Return the feedback received from C_n to the gatekeeper on C_m.
9: Decay with the rate, d. See section II-A.
10: Calculate sleep time, $t_{S_i}(k)$. See (3).
11: Calculate final scaled sleep time, $T_{S_i}(k)$. See (5).
12: **if** (Feasible Configuration Check)
13: Force S_i to move to the capsule C_m.
14: go to step 16.
15: Sleep for time, $T_{S_i}(k)$.
16: **end**

Fig. 1. Subroutine for the functional behavior of an intelligent ant

Input
A set of capsules, $\mathbb{C}$; A set of services, $\mathbb{S}$.
Communication pattern, $comm(S_i, S_j) \forall\ i, j$.
The quality requirements, $Q^{min}_{S_i,j}$, $Q^{max}_{S_i,j}\ \forall\ i, j$.
Assumptions
There is at least one feasible solution.
Initialization
Create a random partition, $\mathbb{P}(0)$.
Output
A feasible partition of the service set $\mathbb{S} = \cup_{i=1}^{|\mathbb{C}|} \mathbb{S}_i$ such that at least minimum quality requirements of each service S_i are satisfied.
1: **begin**
2: Create $x * |\mathbb{C}|$ ants for all S_i in $\mathbb{S}$.
3: Release all ants in $\mathbb{A}$.
4: **do**
5: Allow an ant to perform its task.
5A. Choose the ant at the head of the execution queue.
5B. Execute the ant subroutine: INTELLIGENT ANT. See Figure 1.
6: The service owning the executed ant is placed based on its ants' confidence.
7: **if** (AllAntsDead)
8: Report "No Convergence".
9: Go to step 13.
10: **until**
11: All services in $\mathbb{S}$ achieve at least minimum specified quality levels, i.e. $Q_{S_i,j} \geqslant Q^{min}_{S_i,j}\ \forall\ i, j$.
12: Report the service partition (configuration).
13: **end**

Fig. 2. Pseudo code for the basic algorithm to partition the services

the $|\mathbb{C}|$ first services one by one onto the capsules ensuring that each capsule host at least one service. This is deterministic rehashing since these services were allocated to only the first capsules in the above construction of feasible configuration. Then the remaining services are assigned randomly to host capsule $U[1, |\mathbb{C}|]$. Note that this may indeed create a very skewed distribution of the services per capsule. In order to have control on the goodness of the random numbers the *ran2* generator proposed by [9, pp. 282–286] was used.

The algorithm was implemented in C++ based on the framework of [3] and the test software ran on the *Condor*[1] LAN grid system at the University of Oslo. The usefulness of the algorithm is tested for a range of grid platform sizes measured in terms of number of computational nodes (capsules), and application sizes in terms of the number of services. It is also important that the average number of services per capsule differs. By keeping the expected number of services per capsule, $E\{|\mathbb{S}_m|\} = \{4, 6, 8, 10\}$, the tests are conducted for all combinations of, $|\mathbb{C}| = \{4, 8, 16\}$ and $|\mathbb{S}| = E\{|\mathbb{S}_m|\} \times |\mathbb{C}|$.

[1] http://www.cs.wisc.edu/condor/

VII. Discussion

The proposed solution methods are evaluated based on the convergence of the algorithm on feasible solutions. The five variants of our ant based algorithm would naturally terminate when all the ants have died. An evaluation of convergence only makes sense if there is at least one feasible solution. This is precisely the reason for Θ as initialized in section VI. With Θ, there is at least this feasible solution guaranteeing that a non-convergence is caused by a failing search algorithm. Initially each service dispatched $5|\mathbb{C}|$ ants and the death rates are computed by (2) with $\xi = 0.75$.

The fraction of 300 simulations capable of finding a feasible solution is given in the Table I for the capsule set sizes 4, 8, and 16. When the confidence of dead ants is being used for placing a service, the reported convergence rates [5] were very low in particular for 8 and 16 capsule combinations. In contrast, convergence rates of intelligent ants presented in this paper show significant improvement and this shows the necessity to eliminate the dead ants' confidence.

The first enhancement of this paper was to use external

TABLE I

THE CONVERGENCE (CON) RESULTS FOR 4, 8 AND 16 CAPSULES.

$\|\mathbb{C}\|$	$\|\mathbb{S}\|$	Action Type	Convergence(%)
4	16	Intelligent Ants	96.0
		L_{R-P} Gatekeeper	92.66
		ScaledSleeping - Linear	95.66
		ScaledSleeping - Nonlinear	97.33
		L_{R-P} - ScaledSleeping	93.66
	24	Intelligent Ants	97.33
		L_{R-P} Gatekeeper	97.66
		ScaledSleeping - Linear	97.66
		ScaledSleeping - Nonlinear	98.66
		L_{R-P} - ScaledSleeping	100.0
	32	Intelligent Ants	94.66
		L_{R-P} Gatekeeper	94.33
		ScaledSleeping - Linear	96.0
		ScaledSleeping - Nonlinear	97.0
		L_{R-P} - ScaledSleeping	97.66
	40	Intelligent Ants	89.66
		L_{R-P} Gatekeeper	92.33
		ScaledSleeping - Linear	89.66
		ScaledSleeping - Nonlinear	89.66
		L_{R-P} - ScaledSleeping	95.33
8	32	Intelligent Ants	99.0
		L_{R-P} Gatekeeper	99.33
		ScaledSleeping - Linear	98.33
		ScaledSleeping - Nonlinear	100.0
		L_{R-P} - ScaledSleeping	98.66
	48	Intelligent Ants	98.66
		L_{R-P} Gatekeeper	99.33
		ScaledSleeping - Linear	98.66
		ScaledSleeping - Nonlinear	97.66
		L_{R-P} - ScaledSleeping	99.0
	64	Intelligent Ants	90.66
		L_{R-P} Gatekeeper	86.0
		ScaledSleeping - Linear	87.33
		ScaledSleeping - Nonlinear	93.66
		L_{R-P} - ScaledSleeping	91.0
	80	Intelligent Ants	70.0
		L_{R-P} Gatekeeper	68.66
		ScaledSleeping - Linear	73.33
		ScaledSleeping - Nonlinear	75.33
		L_{R-P} - ScaledSleeping	72.0
16	64	Intelligent Ants	100.0
		L_{R-P} Gatekeeper	100.0
		ScaledSleeping - Linear	100.0
		ScaledSleeping - Nonlinear	100.0
		L_{R-P} - ScaledSleeping	100.0
	96	Intelligent Ants	99.66
		L_{R-P} Gatekeeper	96.33
		ScaledSleeping - Linear	99.0
		ScaledSleeping - Nonlinear	99.0
		L_{R-P} - ScaledSleeping	97.66
	128	Intelligent Ants	83.0
		L_{R-P} Gatekeeper	64.33
		ScaledSleeping - Linear	82.0
		ScaledSleeping - Nonlinear	91.0
		L_{R-P} - ScaledSleeping	74.66
	160	Intelligent Ants	47.0
		L_{R-P} Gatekeeper	24.0
		ScaledSleeping - Linear	49.83
		ScaledSleeping - Nonlinear	49.33
		L_{R-P} - ScaledSleeping	39.33

learning agents that act as gatekeepers for the moving ants and provide some feedback when the ants require to know where to move. The use of such gatekeepers together with intelligent ants is labelled as "L_{R-P} Gatekeeper". In this setting both moving ants and gatekeepers are considered to be intelligent and thus it is expected that the ants will quickly converge to a feasible solution with higher convergence rates. However, gatekeepers seem to have a negative affect on intelligent ants, since the convergence results are inferior to the ones achieved with just intelligent ants.

It is indeed a paradox that applying learning twice gives the worst results. However, from an information theoretical point of view, it is the same quality evaluation that is used twice: first to provide a feedback to the gatekeeper of the capsule just left behind by an ant, and second to set the initial ant confidence to unity provided that the ant went to sleep in the new host capsule. Using the same information two times does not bring any additional information, and hence the performance of intelligent ants with gatekeepers should be identical to the the performance without the gatekeepers since it can hardly be any better.

The reduction in performance when using gatekeepers comes from the outdated advice they bring. Consider one gatekeeper and one ant. The ant moves according to the advice of the gatekeeper, and is satisfied in its new capsule. After some sleep the ant wakes up and re-evaluates the quality achievable for its service. Unfortunately, it is no longer sufficient and the ant moves on. However, the gatekeeper is ignorant of this new information, and is likely advice the same capsule also to the next ant passing. In other words, in a stationary environment the gatekeepers would have performed better, but with the services moving around based on the location of their ants, the environment becomes non-stationary and the advises given by the gatekeepers will necessarily be outdated. For the ant it is better to act as if nothing is known of the environment than listening to outdated information.

There are two changes that may make the gatekeepers perform better with intelligent ants:

- *Increase the number of ants dispatched by a service*. This will have a cost in terms of extra memory consumption by the algorithm as well as in execution time. However, it will allow more ants to pass a gatekeeper more frequently such that its probability vector will be more frequently updated enabling the gatekeeper to adapt to the changed environment.
- *Unlearn as ants move further*. The idea is that the ant gives feedback to the gatekeeper two times: A positive feedback when it arrives to a capsule and finds the quality level acceptable. Then a negative feedback if it has to leave the capsule at some later time because the environment has changed to the degree that the capsule is no longer capable of sustaining the ant's service. This requires the ant to remember the last gatekeeper it interacted with. Note that this approach is philosophically different from the one originally proposed in this paper since adding memory is adding intelligence. This because

intelligence can be understood as memory and an algorithm to use the remembered information, and learning as the way to update the remembered information based on new information.

If an ant has non-zero confidence (see step 7 in Figure 1), it makes no move but continue to sleep in its current location. In other words, the corresponding gatekeeper in that capsule is asked only if the ant has zero confidence and moves to a remote capsule. However, the cases where the ants increase or even decrease their confidence levels, are not informed to their gatekeepers on that capsule. In particular, if there are not enough resources available to satisfy the ant's service quality (see step 2 in Figure 1), the corresponding gatekeeper has to wait until the the ant's confidence is reduced to zero before the gatekeeper is allowed to make changes to its action probability vector. This situation leaves two intelligent groups of agents (intelligent ants and gatekeepers) partly isolated. This would also indicate that the further investigations are required to get more insight and improve the convergence.

The second enhancement was to use a scaling factor to represent the communication among the ants as a function of confidence of co-workers of an ant. The motivation for introducing a scaling factor was to force unsatisfied ants and their services to move more often than the happy ants that still can sleep in a safe capsule. This naturally allows less confident ants to make more moves giving them more opportunities to gain enough confidence on behalf of their services. When the concept of scaling factor (see equations (5) (6) (7)) is applied together with intelligent ants, the convergence results are slightly better than the ones achieved with intelligent ants. Further the use of a non-linear scaling factor have the advantage over linear scaling factor, at least for 8 and 16 capsule combinations. Although it is tempting to assert that higher scaling factors would yield better results, it is not clear how the long sleep times of happy ants would compensate the overall execution time of the algorithm.

When both gatekeepers and (non-linear) scaling factor were used together with intelligent ants, the resultant convergence rates are better than the ones of L_{R-P} Gatekeepers (see Table I). However it is important to note that this small improvement was due to scaling factor and underperformance was due to gatekeepers.

VIII. Conclusion

This paper builds on previous attempts to solve the problem of service allocation in a dynamic grid environment by the use of intelligent foraging ants searching the capsule graph for a good placement for the service they are representing. Two extensions proposed in this work are: (1) learning gatekeepers to guide the movement of the ants among the capsules to enhance the efficiency of the capsule graph exploration (2) a scaled sleep time where each ant's sleep time is a function of the confidence of its co-workers. The results obtained from extensive simulations on randomly generated application services and grid environments, show that the use of scaled sleep time improves the convergence rates because the variations in sleep times of ants allow the less confident ants to move more often. However it was surprising to learn that the guiding elements, gatekeepers, have no positive effect when combined with intelligent ants. Instead the convergence results achieved with gatekeepers are inferior to the ones achieved with intelligent ants presented in Section II-B. In the future, we will focus on identifying more efficient coordination techniques that will further improve the convergence rates.

References

[1] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[2] D. Gannon et al. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Journal of Cluster Computing, Special Issue on Grid Computing*, July 2002.

[3] Geir Horn and B. John Oommen. A fixed-structure learning automaton solution to the stochastic static mapping problem. In H. J. Siegel, David A. Bader, and Jean-Luc Gaudiot, editors, *Proceedings 19th IEEE International Parallel and Distributed Processing Symposium*, pages 297b – 297b, Denver, Colorado, April 2005. IEEE.

[4] S. B. Musunoori and G. Horn. Ant-based approach to the quality aware application service partitioning in a grid environment. In *IEEE Congress on Evolutionary Computation (CEC 2006)*. IEEE, 2006.

[5] S. B. Musunoori and G. Horn. Intelligent ant-based solution to the application service partitioning problem in a grid environment, 2006. Accepted to the Sixth International Conference on Intelligent System Design and Applications (ISDA 2006).

[6] S.B. Musunoori and G. Horn. A Fixed Structure Learning Automaton Solution to the Quality Aware Application Service Configuration in a Grid Environment. In *Proceedings of 17th International Conference on Parallel and Distributed Computing and Systems (PDCS 2005), Phoenix, AZ, USA*, November 2005.

[7] K. S. Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction*. Prentice-Hall, Englwood Cliffs, NJ, 1989.

[8] B. John Oommen and Daniel C. Y. Ma. Deterministic learning automata solutions to the equipartitioning problem. *IEEE Transactions on Computers*, 37(1):2–13, January 1988.

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 2002. ISBN 0-521-75033-4.