In Proceedings of 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004), in conjunction with IEEE/ACM Supercomputing Conference (SC 2004), Pittsburgh, PA, USA, November 2004, pp. 388-395.

Self-Organizing Agents for Grid Load Balancing

Junwei Cao
C&C Research Laboratories, NEC Europe Ltd., Sankt Augustin, Germany
cao@ccrl-nece.de

Abstract

A computational grid is a wide-area computing environment for cross-domain resource sharing and service integration. Resource management and load balancing are key concerns when implementing grid middleware and improving resource utilization. Grid resource management can be implemented as a multi-agent system with resource advertisement and discovery capabilities if job requests from users are associated with explicit QoS requirements. In this work agent-based self-organization is proposed to perform complementary load balancing for batch jobs with no explicit execution deadlines. In particular, an ant-like self-organizing mechanism is introduced and proved to be powerful to achieve overall grid load balancing through a collection of very simple local interactions. A modeling and simulation environment is developed to enable performance of the ant algorithm to be investigated quantitatively. Simulation results included in this work illustrate the impact of different performance optimization strategies on the overall system load balancing level, speed and efficiency.

1. Introduction

The grid is proposed to be a new computing infrastructure that provides uniform access to wide-area distributed resources [9]. Related grid technologies are layered as applications, tools, middleware and fabric [2].

Resource management and scheduling are important services of grid middleware, which must efficiently map grid-submitted jobs to available grid resources. An agent-based methodology is considered to be flexible for grid resource discovery, since agents control query processes according to their own internal logic rather than relying on a fixed function query engine [11].

In our previous work an agent-based methodology was developed for building large-scale distributed software systems with highly dynamic behaviors [4]. This has been used in the implementation of an agent-based resource management system (ARMS) for grid computing [5]. While an ARMS system can achieve grid load balancing as a result of trying to meet QoS requirements specified explicitly by users [6], self-organization is investigated in this work for agents to perform load balancing automatically for batch queuing jobs that are not explicitly associated with execution deadlines.

As summarized in [17], social insect paradigm is one of the major self-organizing mechanisms used in nowadays applications. In this work, self-organization among ARMS agents is implemented using ant-like [18] local interactions so that load balancing can be achieved as an emergent collective behavior of the system. A modeling and simulation environment is developed to enable the performance of the ant algorithm to be investigated quantitatively. Several possible performance optimization strategies are discussed. Simulation results show that the choice of different strategies can have different impact on the overall system load balancing level, speed and efficiency. These information can be provided to improve performance of an actual running grid resource management system.

There are some existing work that focus on the use of self-organization for distributed system management [12, 13, 15, 19]. Motivations of these work and corresponding approaches used are quite different from each other. The following two are most related to the work described in this paper.

- *Condor*. Condor is motivated by improving system thoughput via cycle stealing [14]. Condor-G [10] is a centralized agent for users to access multiple Condor pools via Globus [8]. A recent work described in [3] focuses on resource sharing across Condor pools using a peer-to-peer flocking technique that is self-organizing. In our work, a job scheduler, COSY [7], is utilized for local grid management. While Condor is mainly a batch computing system, COSY can provide both batch queuing and advance reservation supports. Compared with Condor-G, ARMS utilizes a decentralized agent-based approach. The grid-level self-organizing mechanism used in our work is also different from [3].

- *Messor*. Messor [16] is a grid computing system aimed at supporting the concurrent execution of large-scale parallel computations. Messor is implemented on top of the Anthill [1] framework that is based on ideas such as multi-agent systems and evolutionary programming. The ant algorithm used in this work is very similar with that in the kernel of Messor. More detailed investigation of the algorithm performance against different optimization strategies is carried out in this work.

The rest of the paper is organized as follows: Section 2 introduces the agent-based approach for grid resource management; in Section 3, self-organizing mechanisms for load balancing and related performance issues are discussed; simulation results are included in Section 4 and the paper concludes in Section 5.

2. Agent-Based Resource Management

Grid resource management is usually implemented at two layers. Traditional job schedulers for clusters and supercomputers can be used for local grid management. Management of grid resources at a higher level provides the capability of deliver grid-submitted jobs to local schedulers. This is illustrated below using an integration of ARMS [5] and COSY [7].

COSY is an NEC lightweight implementation of a job scheduler for PC clusters. The main feature of COSY is that both batch queuing jobs and advance reservations are supported. For batch jobs, COSY uses the first-come-first-served policy together with aggressive backfilling to maximize the system throughput. For jobs with explicit QoS requirements, COSY can reserve nodes to guarantee an exact start time or meet a deadline. COSY implements an additional policy that associates advance reservations with mandatory shortest notice time.

ARMS utilizes an agent-based methodology for distributed computing management. ARMS agents can be integrated with COSY schedulers to perform global grid management. This is illustrated in Figure 1.

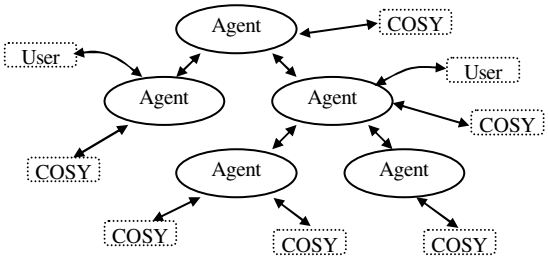


Figure 1. Grid Resource Management with ARMS and COSY

Each ARMS agent is a representative of one or multiple COSYs in a higher-level grid environment. All of agents can receive job requests from users. ARMS agents are logically organized into a hierarchy and can cooperate with each other for resource advertisement and discovery. COSY is responsible for providing local grid resource information to its corresponding agent. Agents can advertise these information along the hierarchy to neighboring agents. When a job request arrives, an agent usually looks up its own COSY information first. If there is no local resources available, an agent will make decisions to dispatch requests that can not be satisfied locally to neighboring agents. Detailed introduction to these processes can be found in [4].

Load balancing in ARMS is previously driven by QoS requirements of job requests. Users have to specify an explicit job execution deadline so that the discovery process among ARMS agents can result that a relatively free grid resource is finally allocated. In this user-centric scenario, as long as user' requirements can be met, load balancing may not be achieved if system workload is not heavy or there are batch job requests that are submitted without QoS specifications. Experimental results included in [6] show that when system workload is very heavy, load balancing can be achieved even in a user-centric scenario.

In this work, a resource-centric scenario is considered. ARMS agents are required to provide an additional mechanism for automatic load balancing of batch queuing jobs across multiple COSYs to improve overall utilization of grid resources. This can lead to an improved average response time to batch queuing jobs, though there are no explicit deadlines associated with these jobs. The required mechanism has to conform to original motivations of ARMS to provide scalability and adaptability. This is why self-organization is considered to be a good choice.

3. Self-Organizing Load Balancing

The additional self-organizing mechanism for automatic grid load balancing in ARMS does not allow a centralized control. It must introduce very limit additional workload and achieve reasonable performance. The social insect paradigm is believed to be the right choice to meet these requirements. In particular, the ant algorithm is described below.

3.1 Ant-like Self-Organization

Consider an ant colony, several species of ants are known to group objects in their environment (e.g., dead

corpses) into piles so as to clean up their nests. An artificial ant colony exhibiting this very same behavior is described in [18] using a simulation environment. The artificial ant follows three simple rules: (i) wander around randomly, until it encounters an object; (ii) if it was carrying an object, it drops the object and continues to wander randomly; and (iii) if it was not carrying an object, it picks the object up and continues to wander. Despite their simplicity, a colony of these "unintelligent" ants is able to group objects into large clusters, independent of their initial distribution. The algorithm used in ARMS (and also the one used in Messor) is inspired by the idea that converse rules may form the basis of a completely decentralized load balancing algorithm. The iterative algorithm used in ARMS can be described as:

1. An ant wanders from one agent to another randomly and tries to remember the identity of an agent that is most overloaded;
2. After a certain number of steps (*m*), the ant changes the mode to search a most underloaded agent, though still wandering randomly.
3. After the same *m* steps, the ant stops for one step to suggest the current two remembered agents (considered to be most overloaded and underloaded, respectively) to balance their workload.
4. After load balancing is performed, the ant is initialized again and starts a new loop from 1.

It is obvious that the pair of agents an ant can find in steps of one loop can only have relatively large difference in workload so that only relatively good balancing effect can be obtained. However, these local interactions indeed lead to global load balancing as an emergent collective behavior given an enough loop number. This is illustrated in Figure 2 using a simple case study.

There are 100 agents involved in this case and visualized in a 10x10 grid in Figure 2. A random selection of agents is initialized with a very high workload and the rest with 0 (see the first picture in Figure 2). Only one ant is involved in the load balancing process and configured with *m=10*. Thus each of loops includes 21 steps with 20 steps' wondering and 1 step load balancing. During wandering steps, the ant selects the next stop randomly from 8 neighboring agents of its current position (5 if currently at the edge of the agent grid and 3 if currently at a corner). Each agent is assumed to have the same amount of resources so that a load balancing process can be simplified to equalizing workload between two agents. The load balancing is processed step-by-step. The last picture in Figure 2 is the visualization of workload distribution after 1400 steps' simulation.

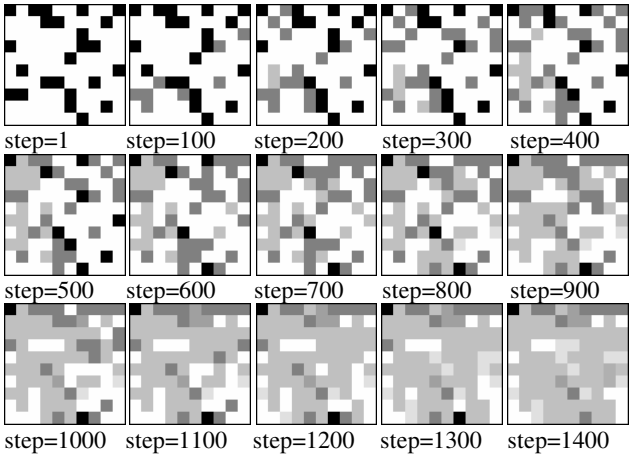


Figure 2. Self-Organizing Load Balancing: a Simple Case Study

It is obvious that after 1400 steps (~66 loops) the load balancing level of the system is significantly improved, though further balancing can still be processed given more loops. For example, the ant does not get a chance within 1400 steps to reach the agent at the upper left corner and distribute its workload to other relatively underloaded agents. There are a couple of factors that can have impact on the performance of load balancing process, e.g. the number of ants (*n*) and wandering steps (*m*). These are discussed in Section 3.4 and investigated in Section 4 using an integrated simulation environment.

3.2. System Implementation

The ant algorithm described above can be integrated into the ARMS&COSY implementation introduced in Section 2.

Each ARMS agent implemented in Java is a meta-scheduler of multiple COSYs. The COSY scheduler is implemented using C/C++ with Java APIs provided. When an agent receives a job request with no explicit QoS requirements, it will queue the job to one of its own COSYs that can finish the job execution earliest. An agent is updated with the current queue situation of all its COSYs. The workload of an agent (*w*) is represented using the average queue length of all its COSYs.

An ant exists as an XML document in the system implementation that contains all necessary data. An example can be found below:

```
<agentgrid type="ant">
  <mode>max</mode>
  <step>8</setp>
  <maxagent>192.168.7.82:7070</maxagent>
  <maxload>660</maxload>
  <minagent>195.37.154.45:7070</minagent>
  <minload>30</minload>
</agentgrid>
```

All corresponding operations required by the algorithm described in the last section are actually performed by the ARMS agent in which an ant currently residents. An ant is initialized by an agent (*mode=max* and *step=m*). Once an ant document is received by an agent, the agent will act according to the ant's current status. If the ant is searching for an overloaded agent (*mode=max* and *step>0*), the agent will check its own workload, compare it with the current value *maxload* of the ant, and update the *maxload* value accordingly. The agent then sends the updated ant document to a randomly selected neighboring agent. The ant mode is switched to *min* and the step number updated to *m* when an ant finishes its searching for overloaded agents (*mode=max* and *step=0*). Then the ant begins steps searching for underloaded agents (*mode=min* and *step>0*). When an agent finds an ant has finished all of wandering steps (*mode=min* and *step=0*), it is responsible to send a message with information on *minagent* to suggest *maxagent* to start the load balancing process. The ant document is again initialized to start a new loop. During the load balancing process, *maxagent* dispatches some of its job requests to *minagent* until they almost equalize the average queue length of their COSYs.

The ARMS agent is extended with the above self-organizing load balancing functionalities. A prototype system is implemented that integrates ARMS agents with COSY schedulers. What is most concerned is how to improve performance of the self-organizing load balancing mechanism. Several performance metrics are considered below.

3.3. Performance Metrics

There are a number of performance criteria that can be used to describe grid resource management and scheduling systems. In this work several common statistics are investigated and used to characterize the effect of grid load balancing.

Let P be the number of agents of an ARMS system and w_{pk} (p=1,2,……,P) be the workload of the agent p at the step k. The average workload of all P agents is:

$$\overline{w}_k = \frac{\sum_{p=1}^{P} w_{pk}}{P}. \quad (1)$$

The mean square deviation of w_{pk} that describes the load balancing level of the system is defined as:

$$l_k = \sqrt{\frac{\sum_{p=1}^{P} (\overline{w}_k - w_{pk})^2}{P}}. \quad (2)$$

Apart from the load balancing level, how fast workload in the system can be balanced is also concerned by system developers. For example, in the case study described in Figure 2, since only one ant is involved, load balancing is processed very slowly, though it is possible that a better balancing level can be achieved given more loops. The load balancing speed (s) is represented using an average improvement of load balancing level per step during the latest loop:

$$s_k = \begin{cases} \dfrac{l_0 - l_k}{k}, k \le 2m+1 \\ \dfrac{l_{k-2m-1} - l_k}{2m+1}, k > 2m+1 \end{cases}. \quad (3)$$

It is obvious that load balancing can not be achieved for free. The process costs additional network connections among agents. System load balancing efficiency (e) is another metrics that takes consideration of system costs for load balancing. Let c_k be the total number of agent connections that have been made to achieve a load balancing level l_k. The system efficiency e_k is represented using the average contribution of an agent connection to load balancing level improvement during the latest loop and calculated as follows:

$$e_k = \begin{cases} \dfrac{l_0 - l_k}{c_k}, k \le 2m+1 \\ \dfrac{l_{k-2m-1} - l_k}{c_k - c_{k-2m-1}}, k > 2m+1 \end{cases}. \quad (4)$$

The most effective load balancing is achieved if more improvement of l is achieved at a cost of fewer network connections c. Most of the time, these metrics described above are conflictive, that is not all metrics can be high at the same time. For example, a high load balancing level does not mean high efficiency, as sometimes good load balancing may be achieved through too many load balancing operations, leading to low system efficiency. Some common strategies are described in the section below that can be used for performance improvement.

3.4. Performance Optimization

The load balancing level and speed can be improved by increasing the number of involved ants. As indicated in the case study illustrated in Figure 2, the wandering scope of one ant is limit. If multiple ants are active simultaneously, the load balancing can be processed much faster. This of course costs more agent communications. In an actual running system, system load balancing level and efficiency can be tuned by choosing a reasonable number of ants (n).

Another ant configuration that may have impacts on system performance is the step number *m*. If a loop includes a very small number of steps, an ant will initial load balancing processes very frequently. If an ant is initialized with a larger *m*, it will wander for a longer time to find agents between which a more effective load balancing can be processed. Different system load distribution may lead to different load balancing performance. In the previous case study (see Figure 2), it may be more effective to trigger more load balancing processes at the beginning when system workload is seriously unbalanced at any area of the agent grid. But it may be more effective to have a larger *m* later, since the ant then is able to wander in a larger scope so that more overloaded agents can be reached.

There are some other ways to improve system performance for load balancing. For example, in the Messor [16] ant algorithm, instead of wandering randomly, additional load storages are used so that smarter ants can move faster towards regions that are believed to be more overloaded or underloaded. In this work, another performance optimization strategy is considered. When an ant wants to make decisions on where to move next, its residence agent will contact all its neighboring agents for latest workload information. An ant always chooses the most overloaded neighboring agent as the next stop in a *max* mode and most underloaded one in a *min* mode. This strategy is believed to be useful for ants to move fast towards interesting agents, though of course at cost of additional agent communications.

Due to an absence of a large system deployment, the above performance optimization strategies cannot be evaluated in an actual working environment. A modeling and simulation environment becomes necessary to enable these performance issues to be investigated quantitatively. Several experiments are carried out and described in the following section.

4. Performance Evaluation

In performance evaluation described in this section, agent systems can be modeled using several aspects of parameters and simulation results are displayed in different views.

4.1. System Modeling

Behaviors of an actual running ARMS&COSY system can be very complicated. However, the motivation of developing an integrated environment is not to accurately simulate system behaviors, but to enable the impacts of performance optimization strategies described in Section 3.4 to be investigated. In this work, agent systems and the ant algorithm are modeled in a simplified way using several aspects of parameters enough to outline system characteristics statistically.

- *Agents*. The number of agents is defined as a square number so that agents can be mapped and visualized in a square grid as shown in Figure 2. Each agent normally has 8 neighboring agents. All of experiments described later include 2500 agents.
- *Workload*. A workload value and corresponding distribution are used to characterize system workload. Each of system models used in Section 4.2 is configured with a workload value 25500, which is initialized to a randomly selection of 10% of agents at the beginning of simulation. This static workload model represents the situation of a short period of time in a dynamic process and is considered to be enough for performance evaluation.
- *Resources*. Resources in a system model are defined in the same way as workload. Each of system models used in Section 4.2 is also configured with a resource capability value 100, which is initialized to all (100%) of agents. The resource model is also static.
- *Ants*. A user can specify the number of ants (*n*) and wandering steps (*m*). Ants are initialized randomly in the agent grid when simulation starts. Ants can also be configured with different wandering styles: *random* or *optimal*, as introduced in Section 3.4.

Once all of above information is input in a system model, it is ready to start a simulation. There are two views in which simulation results are displayed. During each step, the workload of an agent is actually calculated by its current workload value divided by its resource capability value. Workload of all agents is mapped to a gray value between 0 and 255 so that a visualization view can be generated as shown Figure 2. The simulation environment provides another statistical view to display trends of statistics defined in Section 3.3. For example, during each step, the simulation counts the number of agent communications made for load balancing, which is increased by 1 if an ant moves from one agent to another and 2 if a load balancing is processed. Multiple models can be simulated simultaneously and statistical results can be compared in the same window.

4.2. Simulation Results

In the first experiment, the impact of the number of ants on load balancing performance is investigated. Totally 7 system models are defined with identical

information on agents, workload and resources, as described in the last section. The ant number is configured with 20, 50, 100, 200, 500, 1000, and 2000, respectively. Ants are configured to wander randomly with *m=10*. The simulation is processed for 300 steps and simulation results on all of performance metrics are illustrated in Figure 3.

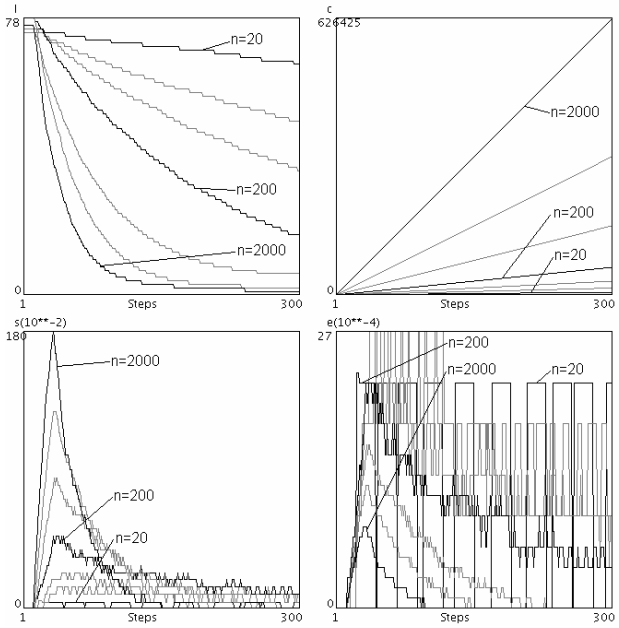


Figure 3. Performance Impact of the Number of Ants on Load Balancing: a Statistical View

In Figure 3, curves for ant numbers 20, 200 and 2000 are identified in black. Other in-between situations are gray. It is obvious that with the number of ants increased, both load balancing level and speed are improved. Especially during the early steps of the simulation, system performance is dramatically improved with more ants involved. While extremely good load balancing level and speed can be achieved with 2000 ants involved, system cost on agent communications is proved to be also extremely high, leading to very low system efficiency. It seems that with 200 ants involved, high system efficiency can be achieved as well as reasonable load balancing level and speed. The choice of 200 ants is a good tradeoff in this case study. The same result is also visualized in Figure 4.

The first picture in Figure 4 visualizes the initial workload distribution among agents during the simulation of the first system model in which *n=20*. 10% of agents are initialized as overloaded. The rest pictures show agent workload distributions of all 7 system models after the simulation of 300 steps. Initial situations of the rest 6 simulations are not included since they are quite similar to that shown in the first picture.

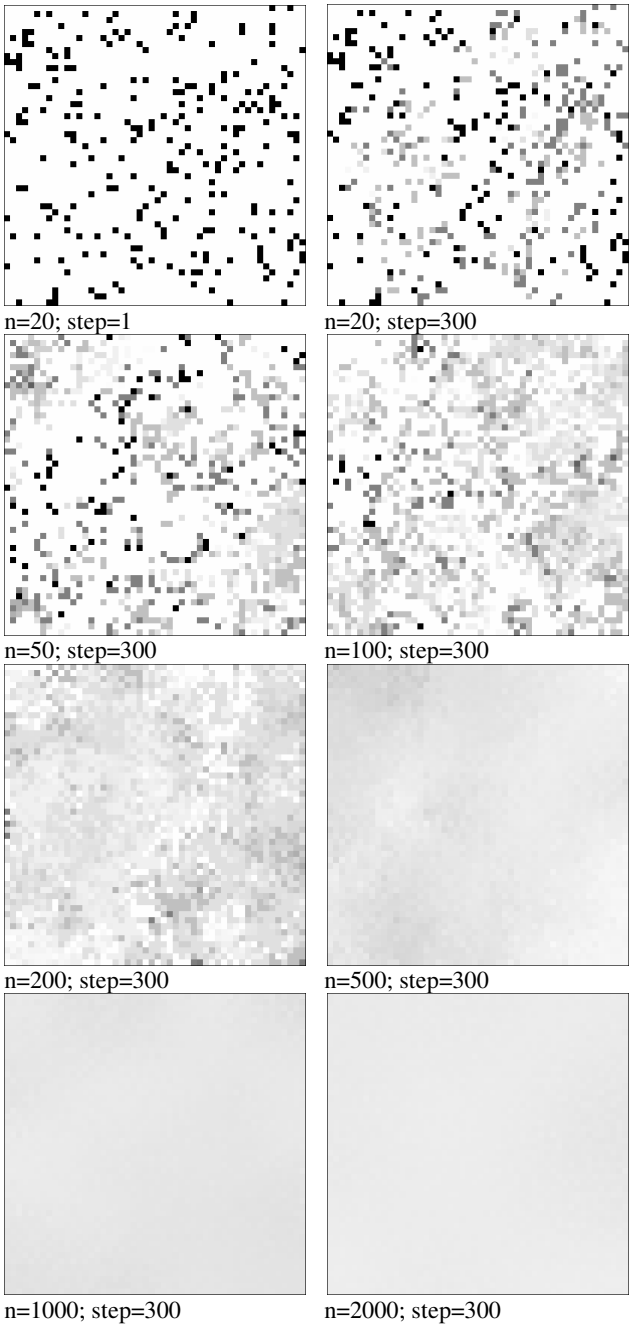


Figure 4. Performance Impact of the Number of Ants on Load Balancing: a Visualization View

Extremely good load balancing effects are achieved in the last three situations where so many ants are involved in load balancing processes. A reasonable load balancing is already achieved when only 200 ants are involved after 300 steps.

The second experiment is motivated to study impact of the number of ant wandering steps (*m*) on load balancing performance. The system model with 200 ants involved in the first experiment is chosen. 4 additional

models are desinged with various numbers of ant wandering steps: 1, 5, 20, and 50, respectively. Ants are still wandering randomly in these system models. The statistical view for the second experiment is given in Figure 5.

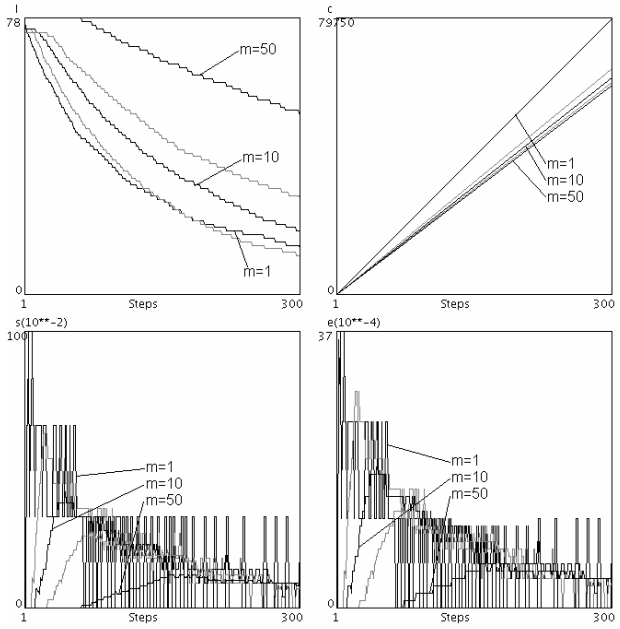


Figure 5. Performance Impact of the Number of Ant Wandering Steps on Load Balancing: a Statistical View

Only 3 situations where *m=1, 10, and 50* are identified in black in Figure 5. Curves for *m= 5 and 20* are gray. An interesting result is observed when comparing load balancing levels of *m=1 and 5*. During early steps of the simulation when workload is seriously unbalanced among agents, more load balancing processes help improve performance instead of wandering a lot. However, during later steps of the simulation when the system has achieved a reasonable load balancing level, it seems wandering more to look for more overloaded or underloaded agents in a larger scope becomes more important than processing load balancing locally and frequently. Within the simulation of 300 steps, further increases of the number of ant wandering steps to *m=10, 20, and 50* do not show any benefit. While it is possible that a better load balancing level can be achieved given further steps in these situations, the load balancing speeds are low.

Simulation results of the second experiment show that a better performance can be achieved if ants wander for 5 steps in each mode, that is, 11 steps per loop. Since each ant wanders randomly, the scope an ant can reach is actually very limit. In system models of this experiment, 2500 agents and 200 ants are involved. This means averagely an ant is responsible for load balancing among 12.5 agents. In general, a good load balancing effect can be achieved in a system model if the number of ant wandering steps in a loop equals to the average number of agents an ant is responsible for load balancing. This leads to an empirical equation for choosing *m*:

$$m \approx \frac{P}{2n}. \tag{5}$$

A third experiment is carried out to optimize ant wandering processes to improve load balancing performance. Simulation results are illustrated in Figures 6 and 7, respectively.

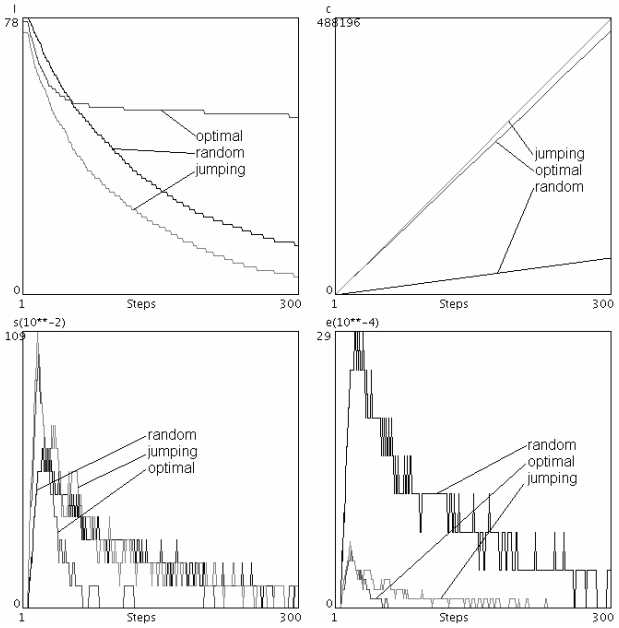
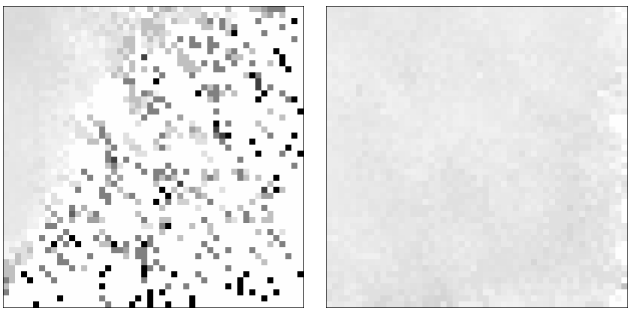


Figure 6. Performance Impact of the Ant Wandering Style on Load Balancing: a Statistical View

The system model in the second experiment in which *n=200* and *m=5* is used again and identified in Figure 6 in black. The curve associated with the *optimal* solution is dark gray. The optimization obviously does not work as well as expected. As shown in the left picture of Figure 7, the optimization guides all of the ants in one direction and a very good load balancing is only achieved in a local area of the agent grid. The similar phenomenon is also discussed in Messor.

In this experiment, an additional mechanism is introduced to enable ants "jumping" randomly once a while (20 steps in this case) before stuck in a local area. The result is identified in Figure 6 in gray and proved to work well as illustrated in the right picture of Figure 7. It is also shown in Figure 6 that the *optimal* solutions, whether with or without the *jumping* mechanism, costs

much more than the previous random wandering and leads to low system efficiency.



style=*optimal*; step=300 style=*jumping*; step=300

Figure 7. Performance Impact of the Ant Wandering Style on Load Balancing: a Visualization View

It is suggested that in a grid environment where workload is distributed randomly across all areas, ants can already result in very good load balancing by wandering randomly. Further optimal solutions can be considered to improve performance if system efficiency is not the main issue for system engineering. These experiences learned from simulation-based performance evaluation can become important references when engineering an actual working system for grid resource management and load balancing.

5. Conclusions

The main contribution of this work includes: (i) quantitative performance evaluation of an ant algorithm using a modeling and simulation approach; (ii) self-organizing load balancing of batch queuing jobs with no explicit QoS requirements across distributed grid resources; and (iii) an initial implementation of an agent-based grid management system with ARMS and COSY.

There are many important features required by a grid computing system and not discussed in this work, e.g. security and data management. Future work will focus on the refinement of the system prototype and the ant algorithm.

References

[1] Ö. Babaoglu, H. Meling, and A. Montresor, "Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems", in Proc. of 22nd IEEE Int. Conf. on Distributed Computing Systems, Vienna, Austria, pp. 15-22, 2002.

[2] M. Baker, R. Buyya, and D. Laforenza, "Grids and Grid Technologies for Wide-area Distributed Computing", Software: Practice and Experience, Vol. 32, No. 15, pp. 1437-1466, 2002.

[3] A. R. Butt, R. Zhang, and Y. C. Hu, "A Self-Organizing Flock of Condors", in Proc. of IEEE/ACM Supercomputing Conf., Phoenix, AZ, USA, 2003.

[4] J. Cao, D. J. Kerbyson, and G. R. Nudd, "High Performance Service Discovery in Large-Scale Multi-Agent and Mobile-Agent Systems", Int. J. Software Engineering and Knowledge Engineering, Special Issue on Multi-Agent Systems and Mobile Agents, Vol. 11, No. 5, pp. 621-641, 2001.

[5] J. Cao, S. A. Jarvis, S. Saini, D. J. Kerbyson, and G. R. Nudd, "ARMS: an Agent-based Resource Management System for Grid Computing", Scientific Programming, Special Issue on Grid Computing, Vol. 10, No. 2, pp. 135-148, 2002.

[6] J. Cao, D. P. Spooner, S. A. Jarvis, S. Saini, and G. R. Nudd, "Agent-Based Grid Load Balancing Using Performance-Driven Task Scheduling", in Proc. of 17th IEEE Int. Parallel and Distributed Processing Symp., Nice, France, 2003.

[7] J. Cao and F. Zimmermann, "Queue Scheduling and Advance Reservations with COSY", in Proc. of 18th IEEE Int. Parallel and Distributed Processing Symp., Santa Fe, NM, USA, 2004.

[8] I. Foster and C. Kesselman, Globus: a Metacomputing Infrastructure Toolkit, Int. J. Supercomputer Applications, Vol. 11, No. 2, pp. 115-128, 1997.

[9] I. Foster and C. Kesselman, The GRID: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, 1998.

[10] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: a Computation Management Agent for Multi-Institutional Grids", Cluster Computing, Vol. 5, No. 3, pp. 237-246, 2002.

[11] K. Krauter, R. Buyya, and M. Maheswaran, "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing", Software: Practice and Experience, Vol. 32, No. 2, pp. 135-164, 2002.

[12] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam, "Towards Self-organizing Distributed Computing Frameworks: The H2O Approach", Parallel Processing Letters, Vol. 13, No. 2, pp. 273-290, 2003.

[13] I. Liabotis, O. Prnjat, T. Olukemi, A. L. M. Ching, A. Lazarevic, L. Sacks, M. Fisher, and P. McKee, "Self-Organising Management of Grid Environments", Int. Symp. on Telecommunications, Isfahan, Iran, 2003.

[14] M. Litzkow, M. Livny, and M. Mutka, "Condor – a Hunter of Idle Workstations", in Proc. 8th IEEE Int. Conf. on Distributed Computing Systems, San Jose, CA, USA, pp. 104-111, 1988.

[15] S. Lynden and O. F. Rana, "Coordinated Learning to Support Resource Management in Computational Grids", in Proc. of 2nd IEEE Int. Conf. on Peer-to-Peer Computing, Linköping, Sweden, pp. 81-89, 2002.

[16] A. Montresor, H. Meling, and Ö. Babaoglu, "Messor: Load-Balancing through a Swarm of Autonomous Agents", in Proc. of 1st Int. Workshop on Agents and Peer-to-Peer Computing, 1st ACM Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, 2002.

[17] S. K. Mostefaoui, O. F. Rana, N. Foukia, S. Hassas, G. Di Marzo Serugendo, C. Van Aart, and A. Karageorgos, "Self-Organising Applications: A Survey", in Prof. of 1st International Workshop on Engineering Self-Organising Applications, 2nd ACM Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, Melbourne, Australia, pp. 63-69, 2003.

[18] M. Resnick, Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds, MIT Press, 1994.

[19] F. Wang, "Self-organising Communities Formed by Middle Agents", in Proc. of 1st ACM Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, pp. 1333-1339, 2002.