Autonomous Performance Control of Distributed Applications in a Heterogeneous Environment

Keping Chen
chenk@cs.man.ac.uk

Kenneth R. Mayes
ken@cs.man.ac.uk

John R. Gurd
jgurd@cs.man.ac.uk

Centre for Novel Computing
School of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom

ABSTRACT

A framework is proposed that dynamically adapts to resource changes in a distributed heterogeneous environment. In this framework, computational tasks are wrapped into autonomous entities which are able to control themselves locally. Global control is provided in a decentralised manner via control units which link with these local entities in hierarchies, monitor them and coordinate their behaviour. With these mechanisms, the framework controls performance of a distributed application in a heterogeneous environment by adjusting load balance and adapting to resource changes. Fault tolerance is provided, being viewed as a special case of performance loss. Mixed strategies are applied, including global and local control policies, and their benefits are illustrated in terms of scalability and efficiency.

Keywords

performance control, adaptivity, load balance, awareness

1. INTRODUCTION

Despite the constantly increasing performance of computing hardware, there is a continuing requirement for increasing computational power that is driven by large scale, parallel scientific problems. Integration of networking, computation and information is required in order to meet such a challenging requirement. One of the grand challenges is the dynamic and heterogeneous nature of the distributed environment; resources are non-dedicated, and may slow down for various reasons, or even appear and disappear nondeterministically. Therefore, the ability to find and utilise resources reliably and efficiently is fundamental to achieving application performance in such an environment.

Manual tuning of load balance is time consuming, or even impossible, with a large application over a large scale network. To address the complex issues in runtime performance tuning, the computing system needs the capacity of self-management. In accordance with high-level policies and objectives set by users, low-level adjustments are operated by the computer system itself at runtime *autonomously* to achieve as high performance as practicable. A single point of failure is also unacceptable, so the control mechanism must be decentralised so as to be resistant to failures.

This paper presents a framework and demonstrates its self-adaptivity to achieve load balance and fault tolerance in the presence of dynamic resource changes in a heterogeneous environment. Experiments show that this framework reacts to nondeterministic dynamic changes by runtime control, and achieves good resource utilisation by balancing the workload on each resource. The framework is efficient and scalable, and can be extended to various application structures and network topologies.

The paper is organized as follows. Section 2 introduces the autonomous performance control framework. Section 3 presents different strategies to achieve load balance and fault tolerance, using coordination between local and global policies. Section 4 presents and discusses some preliminary experimental results. Section 5 covers related work. Section 6 concludes and outlines plans for future investigation.

AUTONOMICS 2007, 28-30 October 2007, Rome, Italy

DOI 10.4108/ICST.AUTONOMICS2007.2201

2. THE FRAMEWORK

The framework is based on the performance control model of PerCo [2, 8] which is a prototype system that has the basic building blocks for controlling the performance of multi-component applications executing on the Grid [4]. In the current version of PerCo, the global aspects of performance control have a centralised implementation. The new framework presents an abstraction of the PerCo design, and offers a distributed implementation of global control while also emphasizing local control. Capability to adapt to resources joining and leaving a network is also provided. The following describes the role of each component in the framework and the coordination mechanisms used between them.

2.1 ActiveObject

An ActiveObject (AO) encapsulates process and state into a single entity. AOs are independent from each other and each has a unique name. When running, an AO is able to pause its computation and communication, save its state and migrate completely to a different resource where it can restore its state and restart from the checkpoint. The applications must provide a checkpointing mechanism. Communication between an AO and any other component is asynchronous. AOs only react in response to messages received and only affect their environment by sending mes-

sages. There are different types of messages, e.g. work stealing messages allow AOs to achieve load balance by exchanging part of their workload.

AOs monitor information about both their resource environment and the JobObjects' (see Section 2.3) execution status. AOs use this information to predict the execution time of a job. Prediction is based on the assumption that the resources on which the AOs are installed will be stable until the job finishes. Thus prediction is not always accurate, but it assists analysis of load balancing. It is assumed that the system is more balanced when the predicted times of all the AOs are similar, that is, task sizes are adjusted to the resource capabilities. An abstraction, the *satisfaction level*, is used to represent the monitored performance of an AO. A high satisfaction level indicates the AO is idle and more likely to accept tasks from other AOs, and vice versa. Any changes in the environment will affect the satisfaction level of an AO. If the satisfaction level is low, an AO can react by making decisions that are expected to increase the level (e.g. export work to other AOs).

2.2 ControlObject

A ControlObject (CO) is a management component which does not directly run any computation. COs may or may not exist on the same resources as AOs. COs are able to connect to other components when necessary, to form more sophisticated control topologies, e.g. tree or star, thus enabling scalability across different numbers of resources. COs can monitor the status of the components connected to them. This is done by periodic message exchange. In a multi-level control hierarchy, COs only monitor information from lower level COs. This design separates the responsibility for control into a control hierarchy. COs make decisions for a group of AOs. COs have more information than AOs, and so such decisions can represent a wider area of performance concern. For example, based on historical records, COs are able to identify the AOs which are more likely to have low satisfaction level, and react preemptively.

2.3 JobObject

A JobObject (JO) is an abstraction of the computational task to be executed. It encapsulates data and algorithms into a single entity. A JO can be accepted by an AO and then be executed. It is able to migrate together with the associated AO. JOs also contain application and platform independent descriptions of computational and data-related tasks. A computational task is described as three parts: the input values, the main execution body and the output values. Other parameters are also included reflecting the configuration and other needs of the computation, e.g. the requirement for a special library package. The output of a JO can be used as the input to another JO.

It is possible to adjust the work granularity inside JOs. Splitting work into finer grains allows for moving part of a task from one AO to another; merging enables more tasks to join the current computation. However, the opportunity for such activity depends heavily on the nature of application.

2.4 Resource manager

The resource manager maintains information about all components, including name, network address, etc. Any component, once created, registers itself until it is destroyed. The resource manager provides a look-up service that enables the components to find each other. After an AO migrates from one resource to another, it renews registration with the same name but a different address so that other components will be able to find it.

New resources may be added to the system to achieve higher overall performance in the case that either the application requires more resources, in order to finish the work quickly, or when resources are released by other applications, and these can be utilised to benefit performance. The resource manager discovers these potential resources, chooses the most suitable resources based on the description of each application, and allocates the resources to applications.

2.5 Overview

Figure 1 displays the relationships between the components described above, using an example with five AOs and three COs. One of the AOs is idle without a JO. The COs form two levels: the lower level includes two COs which monitor three and two AOs, respectively, and one higher level CO maintains information from these two counterparts. Communication between the components is not shown.

3. THE STRATEGIES

This section describes both local and global autonomous performance control strategies.

3.1 Awareness of an ActiveObject

AOs maintain and use information, about themselves and about the environment, which guides their behaviour. This information is called the *awareness* of an AO. Such awareness helps the decision-making of an AO. AOs take into account the topology of the network. As they exist in different domains, which may be local or remote, latency will be a large factor in the throughput of message exchange. The method to reduce communication overhead is to put local AOs close to each other, but to allow remote AOs to migrate without restrictions.

Each AO maintains a *friend list* of AOs which have exhibited low latencies in recent communications. Both local and high performance AOs have high rankings on this list. When cooperation is required, an AO first tries to contact the AOs with higher rankings on its list. When performing adaptive operations, an AO is guided by its knowledge of load status. Usually work units are shipped from busy AOs to less-busy AOs. The work to be shipped is also determined using the information in the AOs.

3.2 Awareness of a ControlObject

COs have information about the AOs to which they are connected. The awareness of a CO thus represents the requirements of a group or *region* of AOs. Based on historical data, COs are able to distinguish between busy regions and less-busy regions, and to predict busy regions in the near future. If workload is distributed in an unbalanced fashion, the CO can either force AOs to follow their mandatory decisions or make advisory decisions as guidelines for the AOs' own local decision-making.

When an AO gives no response for a period of time, the associated CO determines it to be *dead*. Having identified a dead AO, the CO notifies other components to stop them trying to contact it. The CO then starts the recovery process (see Section 3.6) for the dead AO in order to restart execution of its tasks on a different resource.

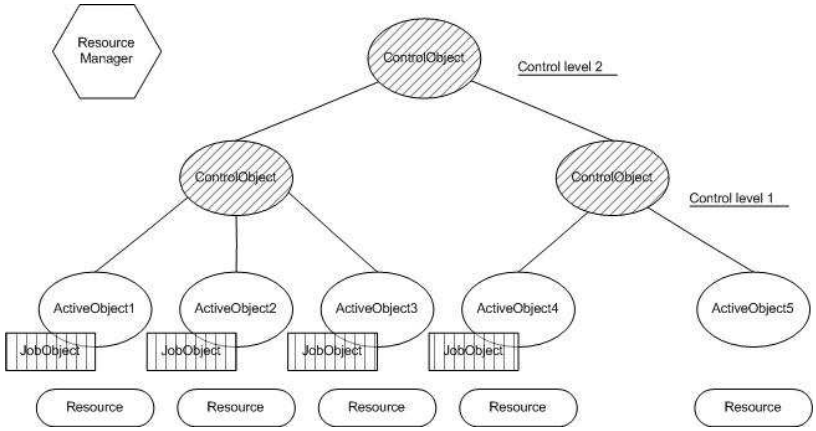

Figure 1: The architecture for autonomous performance control. Lines represent the control relationships between ControlObjects and ActiveObjects.

3.3 Local and global policies

As discussed in Section 2, AOs and COs are all able to make decisions for themselves, and thus teamwork is an essential requirement in this system. Therefore, when control is separated into a hierarchy, mechanisms are needed to reach agreement. Control operations are categorised based on the granularity of work. In fine grain cases, where work stealing is appropriate, COs are used as coordinators. COs monitor help requests from busy AOs and responses from others, and match them accordingly. This process is expected to be more efficient than random matching. In coarse grain cases, where job migration or adding new resources are necessary, COs make these decisions mandatory and all AOs must follow them.

3.4 Load balance in multi-level control

In a multi-level control hierarchy, such as a tree, load should be balanced among the branches of the trees. AOs in the same branch are local and tightly coupled while AOs in others branches may be remote and loosely coupled. The sums of the squares of predicted execution times (S) are calculated using the predicted execution time of each branch. The value S represents the load balance situation of all branches: a larger S represents larger differences in predicted execution time of different branches, thus indicating a poor load balance situation. The CO at the root makes adjustments to re-balance the branches, as long as the overhead introduced is acceptable. The approach is to swap fast and slow resources in different branches by making pairs of AOs migrate from one resource to the other and vice versa.

3.5 Scalability

COs make decisions for coarse grain cases but coordinate the adaptation process for fine grain cases, thereby reducing communication cost. When one CO becomes overloaded because it controls too many AOs, it can either authorise the control of some of its AOs by another CO, or spawn a separate CO with a subset of its AOs.

In a tree hierarchy, because all the branches are connected to the root CO, communication at the top level might become a bottleneck. The solution is to increase the number of hierarchy levels in order to separate the control into different levels. Increasing depth and reducing width of a tree enables the control to scale to a large number of computational resources. A scalability analysis is presented in Section 4.3.

3.6 Fault tolerance

A failure in a resource is treated as an extreme case of a resource slowing down; that is, of performance becoming zero. Data in AOs are replicated, the replication strategy is to let AOs save their states and disperse it to their neighbours. In the case of a loss of some AOs, COs are responsible for maintaining the application execution with an acceptable performance. Solutions are to either redistribute the failed job among existing AOs or discover an idle new resource and create a new AO to recover the execution from the replicated state. Currently the distribution target of replications is one of the neighbours on the friend list, so as to keep the replication overhead at a low level.

4. PRELIMINARY RESULTS

The evaluation experiments are based on a simulation of distributed resources using Java *threads*. Each resource is described in terms of features such as CPU speed and network address. The performance of different resources is determined by such features. Disturbance to performance of a resource is simulated by random modification of the resource features at runtime.

4.1 Application and test cases

The application is a divide-and-conquer problem, similar to the SUDA2 [7] algorithm developed in the HIPERSTAD project[1]. The process of problem solving is first to decompose the problem space into smaller subspaces, so that each subspace can be solved separately by an individual AO. The work involved in a given subspace is altered to be static and predictable in order to help monitoring the load balance situation. The test case used can be divided into a maximum of 1200 independent subtasks, and it is assumed that at the beginning the tasks are scheduled in balance, that is, all the AOs have similar predicted execution times.

In all scenarios below, each component is installed on a separate resource. In Scenarios 1, 2 and 3, the system ini-

[1]More information about HIPERSTAD can be found at www.cs.manchester.ac.uk/cnc/projects/hiperstad.php

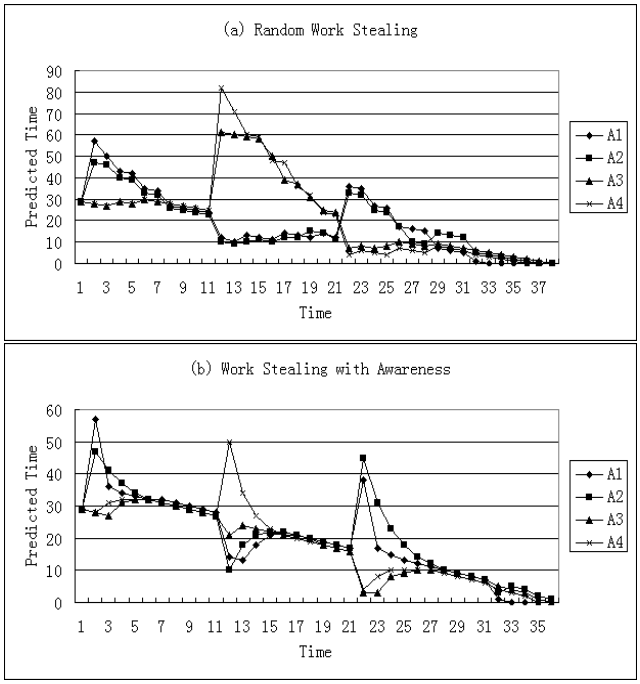

Figure 2: Work stealing in Scenario 1. Wide gaps between lines indicate load unbalances.

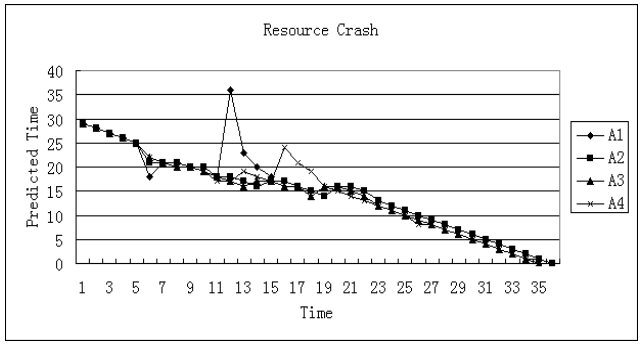

Figure 3: Recover from Failure in Scenario 2. Failure happens on A1; A4 is the neighbour of A1.

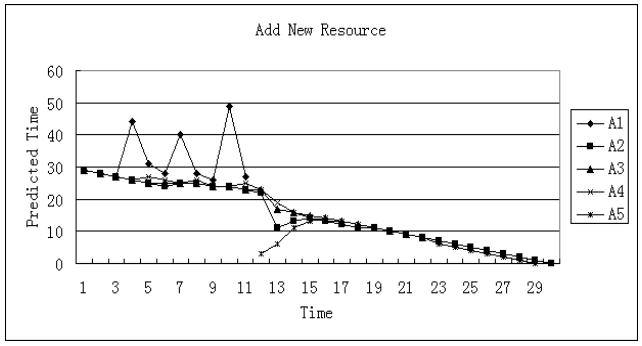

Figure 4: Add new resources in Scenario 3. A5 is the AO on the fast new resource and it replaces A1.

tially consists of four AOs and one CO. In Scenario 4, the system consists of six AOs and three COs which form a two-level hierarchical control architecture; the number of AOs in each branch is four and two, respectively.

Scenario 1 — At runtime the performance capability of all the resources vary. Three disturbances are introduced. The main operation used in this scenario is work stealing.

Scenario 2 — One resource is unstable and finally crashes during runtime. No additional resources are discovered to replace the dead resource.

Scenario 3 — One resource slows down continuously; new resources are discovered after a while and are then added into the system to replace the slowing resource.

Scenario 4 — A disturbance similar to that used in Scenario 1 is introduced to the branch with four components while the other branch maintains a constant performance capability. This tests the resource swapping feature.

4.2 Results and analysis

Figures 2, 3, 4 and 5 show the preliminary results from these experiments. In all these figures, the predicted execution time of each AO is plotted against time, and time is measured in cycles in COs. In one cycle, COs receive monitor information reported from all associated AOs.

Figure 2 displays the result for Scenario 1, which consists of two parts. In Figure 2 (a) the strategy is random work stealing. In Figure 2 (b) the work stealing is coordinated by the CO. When two AOs communicate they exchange information about their workload and previous execution history, and calculate the necessary amount of work to exchange. It can be seen from the figure that awareness helps AOs to achieve load balance more efficiently. In case (b), the job finishes earlier than in case (a).

Figure 3 displays the result for Scenario 2. One AO (A1) is installed on an unstable resource. The unstable resource crashes at about cycle 15. After detecting this, the CO finds the replicated data of the dead resource from one of its neighbours (A4). Then a neighbouring AO accepts the remaining part of the interrupted job. Finally, load balance is again reached by work stealing among the three remaining live resources.

The result for Scenario 3 is displayed in Figure 4. One AO (A1) is installed on a resource that continuously slows down. One new fast resource is discovered by the resource manager at about cycle 12, and the CO decides to add it to the system. The AO (A1) on the slow resource stops and migrates to the newly discovered resource with a new name (A5). There it reinstalls itself and recovers execution from the saved point. The new resource is fast, so the overall execution time is reduced.

Figure 5 displays the result for Scenario 4. The tree is unbalanced after the first two disturbances. In the third case (about cycle 21), the sums of the squares of predicted execution time of the two branches are so different that the top level CO (C_{top}) tries to balance its two branches. The CO at the faster branch identifies the fastest AO it monitors (A_{fast}). The CO at the slower branch discovers the slowest AO it monitors (A_{slow}). A_{slow} and A_{fast} stop execution and swap their location by migration. The system is rebalanced after this exchange, and each branch is then balanced automatically by work stealing.

4.3 Scalability analysis

The performance of the framework is measured with dif-

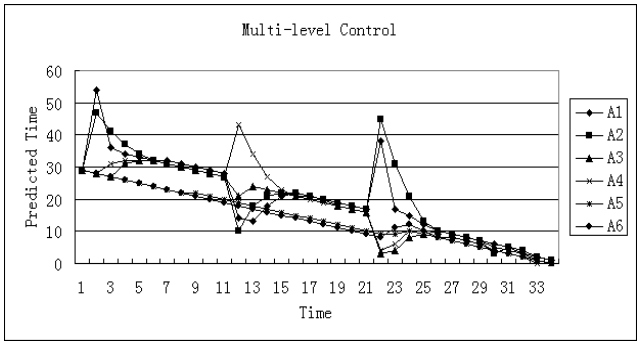

Figure 5: Multi-level Control in Scenario 4. A5 and A6 are on the same branch while the other four AOs are on the other branch. At cycle 22 A2 is swapped with A6 for load balance of the two branches.

ferent numbers of AOs using the same application. The number of AOs associated with each CO is limited to five to prevent the CO becoming a bottleneck. The communication overhead increases as the number of control levels increases. In this test the performance scales approximately linearly (less than 1% overhead) when the control level is no more than three. It is worth noting that scaling depends on the task granularity; larger problems with coarse granularity typically increase parallel efficiency [5].

5. RELATED WORK

There is a large literature on achieving autonomous self-adaptivity in distributed applications. Cactus [1] is a experimental framework which incorporates both an adaptive application structure for dealing with changing resource characteristics, and an adaptive resource selection mechanism that allows applications to change their resource allocation via migration and other approaches. The migration framework described in [9] dynamically adjusts the parallelism of applications executing on computational Grids in accordance with the changing load characteristics of the underlying resources. The framework is lightweight and involves much local autonomy. Distributed computing on P2P networks to achieve coarse-grained parallelism demonstrates the use of decentralised control in a heterogeneous environment [10]. Our framework also benefits from the control hierarchy with additional tuning mechanisms such as work stealing and component migration. The Organic Grid [3] is a self-organising distributed computing system based on autonomous scheduling of mobile agents. Its communication topology is restructured autonomously to increase utilisation of resources; fault-tolerance is also provided. Our system behaves in a similar manner with respect to local control, and also provides global coordination mechanisms.

6. CONCLUSION

A framework has been presented that controls the execution of parallel applications in a heterogeneous dynamic environment. The framework combines hierarchical global and local control policies so as to adapt to nondeterministic dynamic resource changes. Load balancing and fault tolerance are achieved to maintain acceptable performance of distributed applications. Evaluation by simulation has demonstrated the ability of the framework to adapt effectively in a dynamically changing environment. The control hierarchy enables the framework to scale to different numbers of computational resources.

To continue this work we intend to apply this approach to more complex applications and to multiple applications. The adaptive mechanisms in the framework make it possible to autonomously reschedule any incoming new jobs and reachieve load balance at runtime. We also intend to abstract performance control policies by using a distributed dynamic aspect machine [6] to separate such policies as cross-cutting concerns. The performance control concern is separated from other concerns, thus it may be able to help the decision-making process.

7. REFERENCES

[1] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The cactus worm: Experiments with dynamic resource discovery and allocation in a grid environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[2] C. W. Armstrong, R. W. Ford, J. R. Gurd, M. Lujan, K. R. Mayes, and G. D. Riley. Performance control of scientific coupled models in grid environments. *Concurrency and Computation: Practice and Experience*, 17(2-4):259–295, 2005.

[3] A. J. Chakravarti, G. Baumgartner, and M. Lauria. Self-organizing scheduling on the organic grid. *International Journal of High Performance Computing Applications*, 20(1):115–130, 2006.

[4] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

[5] J. L. Gustafon. Fixed time, tiered memory, and superlinear speedup. In *Proceedings of the 5th Distributed Memory Computing Conference(DMCC5)*, pages 1255–1260, 1990.

[6] C. Kaewkasi and J. R. Gurd. A distributed dynamic aspect machine for scientific software development. In *Proceedings of the first Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*, page 3, 2007.

[7] A. M. Manning and D. Haglin. A new algorithm for finding minimal sample uniques for use in statistical disclosure assessment. In *Proceedings of the 5th International Conference on Data Mining*, pages 290–297, 2005.

[8] K. R. Mayes, M. Lujan, G. D. Riley, J. Chin, P. V. Coveney, and J. R. Gurd. Towards performance control on the grid. *Philosophical Transactions of the Royal Society of London: Series A*, 363(1833):1793–1805, 2005.

[9] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[10] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *Proceedings of the third International Workshop on Grid Computing(Grid'02)*, pages 1–12, 2002.