Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 461–470

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Management of Non-functional Attributes of Parallel Components

Yunfeng Peng, Changjun Hu, Chongchong Zhao, Shigang Li, Shucai Yao

Department of Computer Science and Technology,
University of Science and Technology Beijing
Beijing, China

Abstract

In this paper, we present an extension to the CCA component architecture. The extension defined a minimal set of non-functional attributes of parallel components. We have[a] implemented the common CCA components for the management of these attributes. Parallel components can provide some non-functional interfaces optionally. And they provide related information to the common components through these interfaces. For the optimization of parallel components implementations, the component developer should implement the attributes management parts specific to certain parallel components. The tests of the management of six of them show that our management mechanism can improves the performance of parallel components. And the test of management of the other three shows that this is an easy way of controlling parallel component execution. And it is very efficient. A real application test shows the practicability of our proposal.

Key words: parallel component, non-functional attribute, CCA, attribute management, resource management

1. Introduction

Recently, in the area of parallel computing, raising the productivity of parallel computing software is gaining more importance for two reasons. On one hand, the development of large scale parallel applications is still at stage of handwork, causing it very hard to maintain and reuse the software. On the other, new hardware architecture is available, which needs high portability and high productivity of parallel software. Based on common component technologies like COM [1], CORBA [2] and EJB [3], parallel component becomes a technology to solve these problems. Parallel components are components used to compose high performance scientific computing applications. Concerto project [4] introduced a middleware platform for parallel adaptive components. This platform provides a mechanism for resource modeling and monitoring. Furmento Nathalie advanced a layered component architecture for HPC Applications [5]. It separates component abstraction and implementation. And Lei Zhao [6] shows a method for predictive performance modeling of parallel component composition.

Researchers from American national labs and academic institutions founded Common Component Architecture (CCA) forum for parallel components [7]. In this architecture, components written in different languages can interoperate through Scientific Interface Definition Language (SIDL) [8]. CCA gives the least requirements on components [7]. This requirement is a component only has to implement the CCA's Component interface. This interface only defines one method, "setServices( )" [7]. The most important benefit of CCA is that it can incorporate the legacy codes. CCA puts a strong emphasis on performance. It is targeted for high performance scientific computing. Both parallel and distributed computing can be used in CCA [7].

These existing parallel component technologies have changed the way in which scientific computing software was designed. But some attributes of parallel components orthogonal to the scientific function need to be paid more attention. These attributes are somewhere referred as QoS (Quality of Service) [9] [10]. Sometimes, they can affect the performance of parallel components. They may also help the users to control the execution of parallel components. We defined these attributes as non-functional attributes in this paper. These non-functional attributes may appear and be discussed in other software applications [11]. And we believe that they can improve the usefulness of parallel components software.

In this paper, we present an extended component architecture to CCA. This architecture provides a unified management to the

* Yunfeng Peng..Tel.: +8613520368146
E-mail address:peng_ustb@yahoo.com.cn.

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.048

non-functional attributes of CCA functional components. We provide the management of nine attributes of functional components. They are performance, resource requirement, deployment, schedule, load balance, adaptive, execution, transaction, security and access. Our extension gives a convenient way for managing the non-functional attributes of parallel components applications.

The most important contribution is the separation of attributes management in parallel components applications. First, the management of non-functional attributes is treated as a special unit in the parallel components applications. Second, this management is divided into two parts. One part contains some non-functional components. They define the management part shared by all functional components. These components are common components to all parallel application components. The other part includes some optional non-functional interfaces. These interfaces can be provided by the functional components. The methods implemented in these interfaces realize the management parts specific to different functional components. And this management is extensible. The users can implement their own methods for their components. The tests results show that our management is effective. And this management can improve the performance of parallel components applications. Some reflective systems also gave the concept for separating interfaces for meta-level description of components [12]. They divided reflection into two parts. Structural reflection can change the functional aspects of the component [12]. Behavioral reflection is used to dynamically insert some interceptors for application monitoring [13]. Our paper focuses on the non-functional attributes of parallel components. How to control and manage these attributes in a parallel components application is our primarily concern.

The remaining sections of this paper are organized as follows: Section 2 describes the extended component architecture. Section 3 shows the management of platform resources. Section 4 introduces of non-functional attributes management mechanism. Section 5 contains some examples and tests. Section 6 concludes the paper. And In this paper, we use FC to denote functional component, and NFC stands for non-functional component.

2. Extended component architecture

CCA researches play an important role in designing scientific computing software [14]. But for a parallel component, there are many important attributes beyond a scientific function. For example, resource management and load balance can affect the performance of parallel software greatly [15] [16]. These attributes somewhere referred as QoS (Quality of Service) [9] [10], are orthogonal to the scientific functions of parallel components. We defined these attributes as non-functional attributes of parallel component. In a survey conducted by the high performance computing lab of University of Science and Technology Beijing, 96.4% of the 2000 scientific computing software users questioned said they had met the problem beyond the scientific function requirements. We defined a series of non-functional problems in a problem list. Between the users who met the non-functional problems, 93% gave at least one of nine specific problems from the problem list. These nine problems they met are named separately as adaptive, load balance, deployment, performance, schedule, execution, resource requirement, security and transaction in our list. These attributes are especially noteworthy when the execution environments are heterogeneous platforms. And the variation of implementations of parallel components makes the management of their attributes difficult. As CCA is a basic definition for parallel components. It tries to establish a well-accepted standard for scientific computing software components. It wants to incorporate as many components as possible. This made CCA not so mature for the management of non-functional attributes of parallel components.

So we defined the attributes associated with these nine problems as a minimal set of non-functional attributes of parallel components. These nine attributes are defined as follows. We defined adaptive as the capability of a parallel component. With this attribute, component can react to specific event statically or dynamically. This event can be hardware resource change, input data variation or other user defined events. The reaction can be parallel degree variation, data partition variation or other changes to the parallel component. We defined load balance as another capability of a parallel component. It stands for that a parallel component can migrate the load of it from one processor to another. We defined deployment in this way. A parallel component can have a deployment attribute. Then this component can select a suitable resource from a set of hardware resources. Deployment of this component on this resource can have better performance than others. A component with performance attribute can get its running performance data. It also can predict the performance of itself on certain resources. This parallel component can also manage its performance data. Schedule attribute give the parallel component a way to schedule tasks in the component. Base on certain policies, tasks can be scheduled in a user defined sequence. Execution attribute can suspend the execution of a parallel component. It also can continue the execution from the suspend point when the user requires. This attribute can also stop the execution of a component and exit. Resource requirement attribute gives the user a means to set and get the resource requirement of a parallel component. Security attribute controls the secure access of a parallel component. This attribute sets and performs the access authorities of this component. Only allowed users or methods can access permitted parts of the component. Transaction attribute controls the transactional method execution in parallel components. In a transactional method, some transactional operations are performed on some parallel components. If all these operations success, this method will be successful executed. If not, this method failed and will be executed from the beginning.

We divided parallel components into functional components and non-functional components. A functional component (FC) is a component performs some scientific functions the user needed in the original applications. A non-functional component (NFC) is a component we designed for the management of non-functional attributes of functional components. A NFC is a common component to all parallel applications. A CCA component can define “provides” ports and “uses” ports. Here ports are abstract interface. We take ports as interfaces for simplicity. A “provides” port contain the methods a component implements. A “uses”

port defines the methods needed from other components. Two components can be connected by matched "provides" and "uses" ports. Existing CCA parallel components only put emphasis on functional interfaces. For managing the non-functional attributes of parallel components, we defined some optional interfaces. Every interface contains methods to control a non-functional attribute. For optimization, these methods are specific to the parallel component. And they should be provided by the component developers. We call these interfaces non-functional interfaces. We also defined some non-functional components responsible for the invocation of these interfaces at suitable time. These components use these non-functional interfaces. But they have their own provides interfaces. The users use these interfaces to start non-functional components.

Figure 1 shows the extended architecture. In the top part of Figure 1, the original application consists of five functional components and a Driver component. They are connected by functional interfaces. And three functional components each provide their own non-functional interfaces. The middle part of Figure 1 is composed of our nine non-functional components. This part is connected to the top part by non-functional interfaces. The Driver component uses the start interfaces of these non-functional components. The lower part of Figure 1 is the resource pool. It is composed of resource management components on top of heterogeneous platforms. This part monitors the platform resources and provides resource information to the other parts.

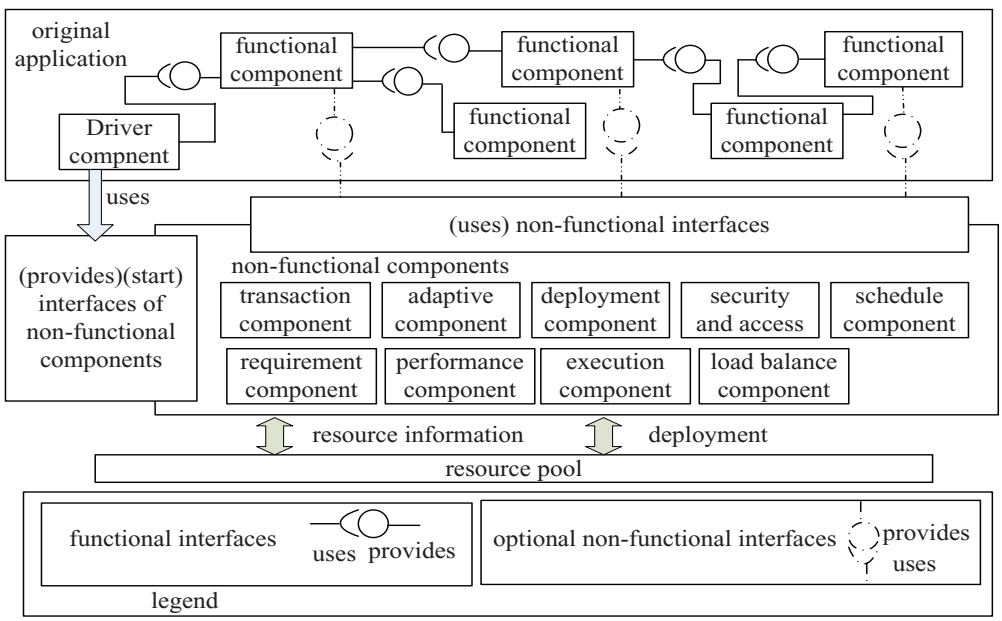

Figure 1: Extended component architecture

3. Non-functional attributes management

We defined some non-functional interfaces for parallel components. These interfaces are optional, and may have different implementations. They are provided by FCs but used by corresponding NFCs. This means the methods in these interfaces are implemented by FCs. FCs register these interfaces to NFCs. And NFCs use these non-functional interfaces to manage the attributes of FCs. We have nine NFCs. Five of them are for the performance improvement of FCs. They are adaptive, load balance, requirement, deployment, performance and schedule components. The other three, execution, transaction, security and access components are for the execution control of FCs.

The nine non-functional attributes have close relationship with the underlying execution environment. And most modern HPC applications are placed on heterogeneous platforms. So we also designed two resource management components for heterogeneous platforms. For managing the heterogeneous platforms, we modeled the resources in a platform as C++ objects [4]. We used XML (eXtensible Markup Language) [17] to describe our resources information. And we mainly concerned about the "node resource" and "network resource". We had two kinds of resource management components. They are "node management" and "network management" components. Every resource is given a unique ID. A "node management" component is in charge of registering every resource in its local node. A "network management" component is located on every node. It can give the network information from simple tests. For example, a ping-pong test can get the time of a single message passing event.

3.1. performance component

A “performance” component is connected to a FC through a “performance” interface. A “sourcep” method can give the source codes location of a component. The “performance” component performs a source code analysis. This analysis counts the numbers of different kinds of operations and communications. For loops, it counts the average number of iterations. For a switch or a branch, it will count the average number of all conditions. Comparing with benchmarks on a standard machine and networks, it establishes a rough performance model. This model contains all the components provides “sourcep” method. The “performance” component then stores it in a model database connected to it. A “dataget” method in “performance” interface can get the runtime performance data. We invoke the TAU (Tuning and Analysis Utilities) component [12] in this method. The “performance” component invokes resource management components to get the resources information. Combing the performance data from “performance” interfaces, the “performance” component can establish the accurate performance models of certain components. This model will replace the rough model in the database. The “performance” component provides a “prediction” interface. It has a “predictive” method. This method can perform a performance prediction of a combination of some FCs. This prediction is based on the prediction models in the model database. This prediction also uses the resource information from resource management components. Figure 2 shows the structure of the “performance” component.

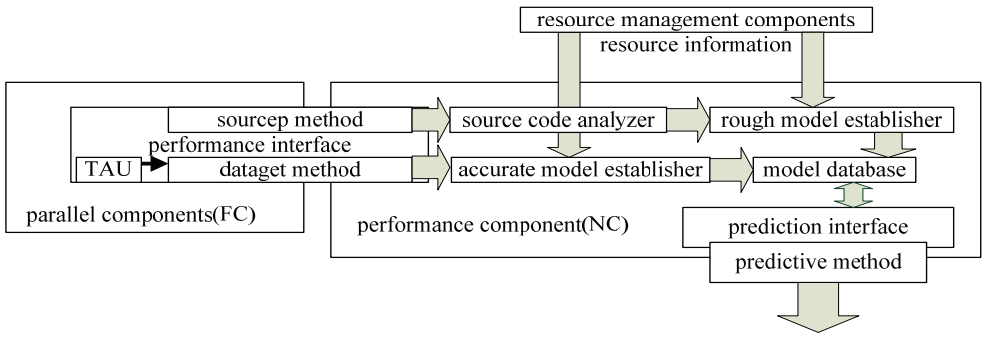

Figure 2: Performance component

3.2. requirement component and deployment component

A “requirement” component can invoke the “requirement” interface of a FC and get its basic resource requirements. This information can be sent to a “deployment” component. This component generates a deployment policy. The generation of this policy contains two procedures. First, it gets the resource information from the resource management components. And it generates every possible deployment based on the resource requirements. Second, we adopt a “heuristic” method for the selection of the best policy. For every policy, the “deployment” component invokes the “performance” component to do a prediction. This prediction is for a combination of some components. For the components that don’t provide a “performance” interface, they will not be in this combination. We supposed that they are not the prominent parts in the performance. Based on these predictions, the best policy is selected. It is sent as a parameter for the “deployment” interface of a FC. The methods in this interface deploy the component. Figure 3 shows the structure of the “requirement” component and “deployment” component.

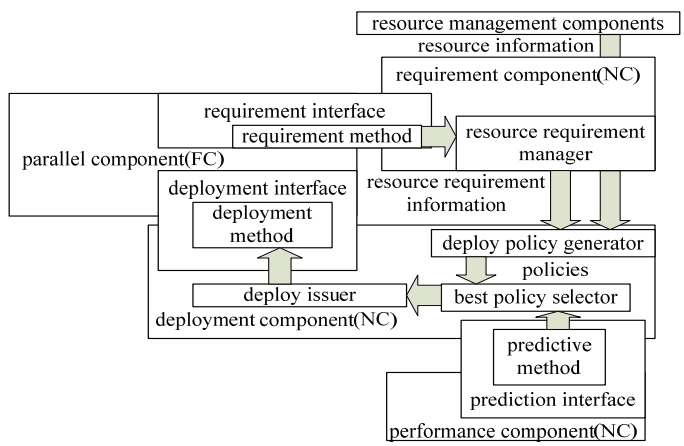

Figure 3: Requirement component and deployment component

3.3. schedule component

We designed a "schedule" component to control the tasks schedule of a FC. If a FC provides a "schedule" interface, a "criterion" method in this interface gives the schedule criterion of this FC. This criterion can be time, priority, real-time, or throughput. A "taskgq" method gives a queue of task groups in this FC. The executions of the task groups must follow the sequence in the queue. But the tasks in a group can be executed in any sequence. The "schedule" component uses the queue and the criterion to generate a schedule policy. First, it has to decide the task sequence in every group. For a time criterion, it will try to minimize the total execution time. Every task will be provided with a time weight. The task with the largest weight in a group will be scheduled. For a priority criterion, task with a high priority will be executed first. For real-time criterion, it will discard the task that doesn't respond in a set time. For a throughput criterion, it will try to minimize the execution time of a task group. The shortest task in the group will be scheduled. After that, the task group will be scheduled one by one. The "schedule" component sends this policy to the FC. And a "scheduler" method in the "schedule" interface executes this policy. Figure 4 shows the structure of the "schedule" component.

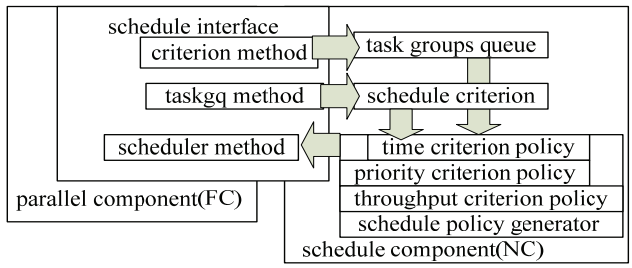

Figure 4: Schedule component

3.4. Load balance component and adaptive component

If we want to provide load balance mechanisms to a FC, we should add a "load balance" interface to it. The FC provides this interface. It can be connected to a "load balance" component uses this interface. A "load balance" component also provides a "start" interface with a "start" method. User can set the parameters for start. A "dtime" parameter set the time interval of two load detection. A "high threshold" and a "low threshold" set the high and low load threshold of a CPU. If the load of a CPU is higher than the high threshold, we call it an overload CPU. If its load is lower than a low threshold, we call it a spare CPU. A "sharen" parameter set the maximum number of CPUs to share the load. If we want to provide load balance mechanism to a FC, we first connect the "load balance" component to the FC through the "load balance" interface. The "load balance" component invokes the node management components periodically. Based on the load reports it gets, it generates a load balance policy. It will migrate the load on the over load CPUs to some spare CPUs. This policy has two constrains. First, the number of spare CPUs we can select can't exceed the value of the "sharen" parameter. Second, the selected spare CPUs must be located on nodes which meet the basic resource requirements of the FC. This can be obtained from the "requirement" component. Then the "load balance" component sends the policy to related nodes. And it will invoke the "load balance" interfaces of the FCs with the policy as a parameter. The methods of the interface then perform the actual balance operations.

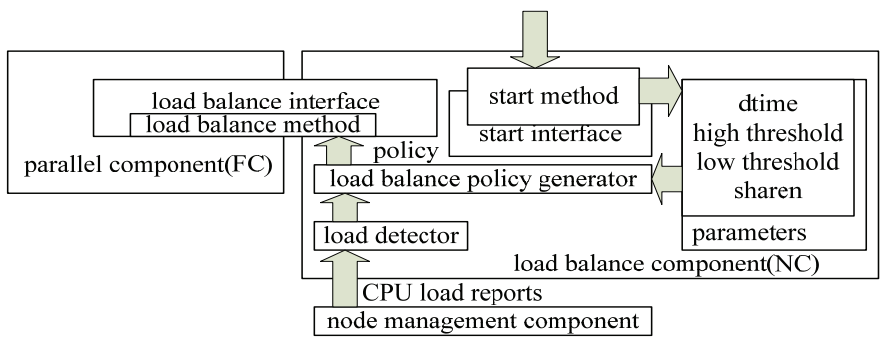

Figure 5: Load balance component

An "adaptive" component performs a similar procedure. But it monitors not only the resources changes but some other user

defined events. The “adaptive” component also has a “start” interface. And an “event” parameter sets the monitored event. When trigger event appears, the “adaptive” component generates an adaptive policy and sends it to the FC. We have four kinds of adaptive policies. They are parallel degree variation, data partition variation, component migration and implementation variation. Parallel degree variation policy represents change in the parallel degree of the FC. Data partition variation means modifying the data partition of the FC. Component migration denotes transfer of the component to another node. Implementation variation stands for use of another alternative FC implementation. The FC implementing the “adaptive” interface will get the adaptive policy. Then it executes the “self-adaptive” methods. Figure 5 shows the structure of the load balance component. And Figure 6 shows the structure of the adaptive component.

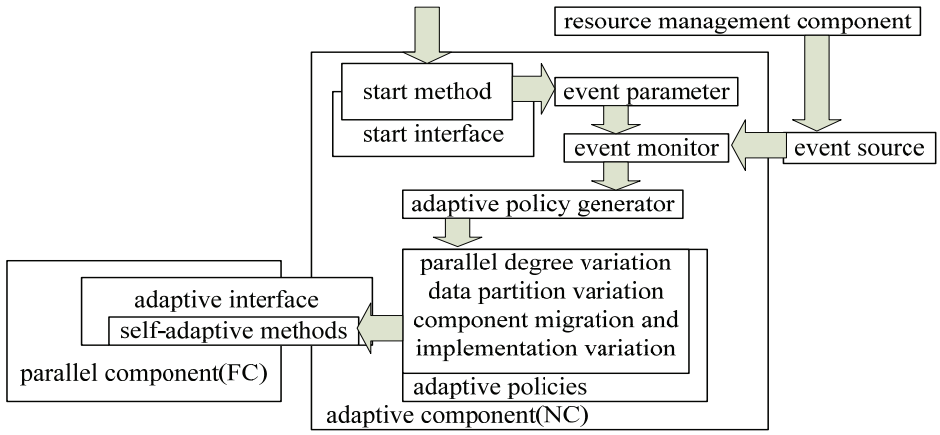

Figure 6: Adaptive component

3.5. components for the execution control

An “execution” component can invoke a “suspend” method in the “execution” interface of a FC. This method suspends the process of the FC and saves its state. It will return the result of the suspend operation and the process ID to the “execution” component. A “continue” method in this interface can use the process ID to active the process. An “exit” method can stop the execution of the FC. A “transaction” component can control the transactional operations of some FCs. Each of these FCs provides a “transaction” interface. There are two methods in this interface. The “transaction” component invokes a “state” method to get the state of the FCs. It invokes a “transactional” method on these interfaces and gets its results. This method performs some transactional operations, such as updates of a shared data. If all transactional methods success, the “transactional” component will broadcast a success message to all FCs involved. These FCs will execute their following operations. If some transactional methods fail, the broadcast message will be a fail notice, and all the FCs will roll back to the start points of these methods. Figure 7 shows the structure of the “execution” component. And Figure 8 shows the structure of the “transaction” component.

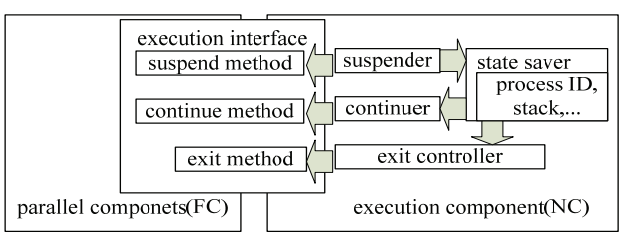

Figure 7: Execution component

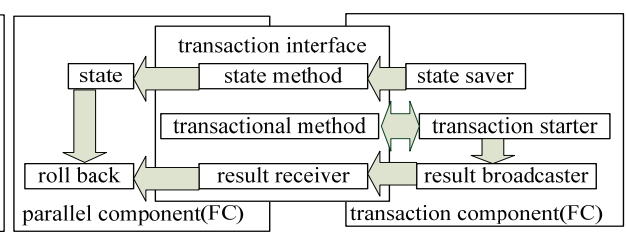

Figure 8: Transaction component

A “security and access” component controls the secure access to FCs. A FC can provide a “security and access” interface. A “set” method in this interface can be used to set the access authorities of different users. These access authorities are listed in a table and stored in a file. These authorities can be promise, deny, or security level of certain methods in the interfaces provided by a FC. The “security and access” component provides an “apply” interface with an “applyf” method. This method has a “clist” parameter. Users can set this parameter as the list of name of FCs they want to apply access control. The “security and access” component invoke an “access” method in the “security and access” interface to control the secure access to a FC. Figure 9 shows

the structure of the "security and access" component.

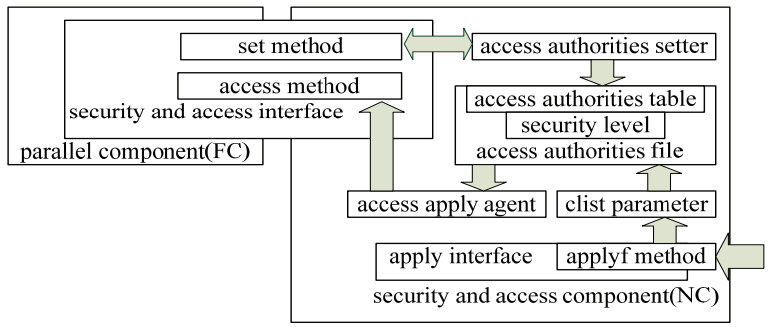

Figure 9: Security and access component

For raising the extensibility of our non-functional attributes management, we have the following settings. The methods implementations in the non-functional interfaces are specific to the FCs. But the NFCs are common and shared by all the FCs. In the above architecture, we have only realized the implementations of all NFCs. The implementations of methods in the non-functional interfaces vary dependent on the implementations of FCs.

4. Experiments and results

In the attributes we defined for FCs, there are six attributes are for the performance improvement of FCs. They are adaptive, load balance, resource requirement, deployment, performance and schedule. The other three attributes are for the runtime control of FCs. We provided the management of these nine attributes in six tests.

Our execution platform is a heterogeneous cluster. It is composed of three parts connected by Gigabit Ethernet. One part is composed by 38 SMPs. One SMP is a machine with 2 x Intel Xeon 2.8GHZ CPU/512K L2 Cache, 2G memory. The operating system is fedora Linux 9. One part is composed by some multicore machines. They are two machines with 4 x Quad-Core Intel(R) Xeon(R) Processor E7320/2.13GHZ/2X2MB L2 Cache, two machines with 4 x Quad-Core AMD Opteron(tm) Processor 8347/1.9GHZ/4096K L2 Cache and a machine with 8 x Quad-Core AMD Opteron(tm) Processor 8347/1.9GHZ/4096K L2 Cache, 8G memory. The operating system is Red Hat Enterprise Linux Server release 5.2. The other part is composed by 16 PlayStation 3(PS3) with IBM CELL BE processor, 7x256KB SRAM. The operating system is Fedora 9 for Linux ppc64.

First we tested our load balance mechanism in a small application. It had a "master" component and five "slave" components. The "master" component sent tasks to the slaves unevenly, with some slaves gotten much more numbers of tasks than other. Each of these components was deployed on a SMP. A task was some amount of simple computation. We implemented a "load balance" interface on the slaves.

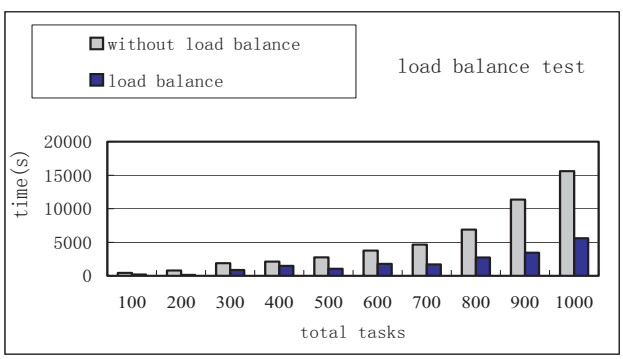

Figure 10: Load balance test

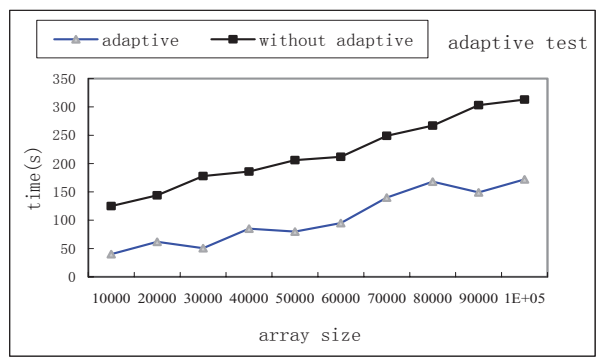

Figure 11: Adaptive test

We tested our adaptive management in anther application. In this application, a "compute" component performed certain amount of computation on a local array. The component was initialized with four processes. Each process was on a SMP. The component process first checked the local array size. If it was large enough, it would trigger an event. This event would be captured by the "adaptive" component. It generated an adaptive policy. This policy defined that the compute processes would

spawn some new processes. And an “adaptivef” method performed the spawn operation, and scattered some data to each new process. After computation, the data were collected to the original four processes.

Figure 10 gives the test results of the application. From Figure 10, we can see clearly that our load balance mechanism can improve the performance of parallel components. And Figure 11 gives the test results. The adaptive policy was increase of the parallel degree. Our adaptive mechanism achieved a much better performance because the computation didn’t involve much communication.

We tested our schedule mechanism using a time criterion. We established a composite component. There were some computing tasks groups in this component. Every task was given a time weight proportional to its execution time on a single CPU core. The “schedule” component called the “taskgq” method in the “schedule” interface on the composite component. It then got the tasks groups’ information. It generated a policy based on time criterion. The “scheduler” method in the “schedule” interface executed the policy.

We tested our “performance”, “requirement”, and “deployment” component together. We built an image processing application composed of five components. An “init” component initialized the processing environment. Then a “scatter” component read some images and sent them to some “processing” component processes. The images were median filtered by these processes. The results were collected by a “gather” component. We added “performance”, “requirement”, and “deployment” interfaces to the “scatter” and the “processing” components. For resource requirement, we defined that the “scatter” component needed memory larger than 4G. And the “processing” component needed a CPU with a frequency faster than 2.4GHZ. In our test, the machines satisfied these two conditions were used for the performance prediction of these two components. And an approximately optimized deployment was generated. The “scatter” component was deployed on a machine with 8G memory. A “processing” component process was deployed on a SMP. The other three components were randomly deployed. We considered that they didn’t affect the performance too much. Figure 12 gives the test results of our schedule mechanism. It was performed on a SMP. Figure 13 shows the test results of our deployment mechanism.

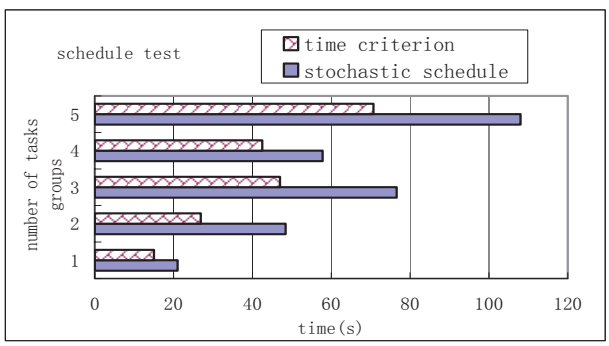

Figure 12: Schedule test

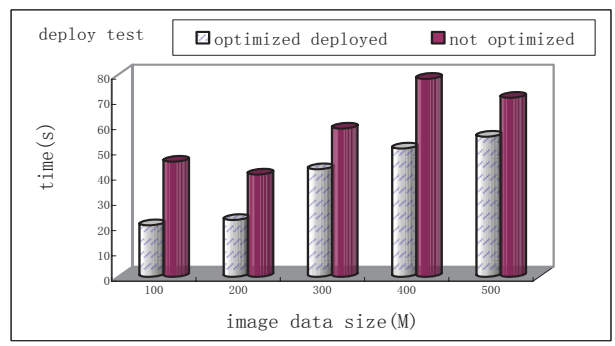

Figure 13: Deployment test

We tested our “execution”, “security”, and “transaction” component in another test. A “business” component implemented a “security and access” interface, a “transactional” interface and an “execution” interface. It was run on five processes. It first checked the authority of the current user. Each of the process then updated a local copy of some shared data. We defined these operations as a transaction.

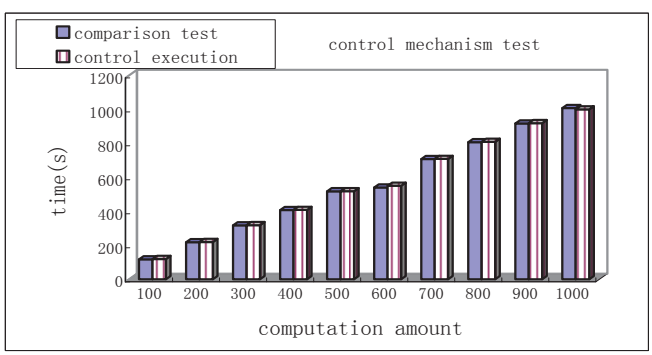

Figure 14: Control mechanism test

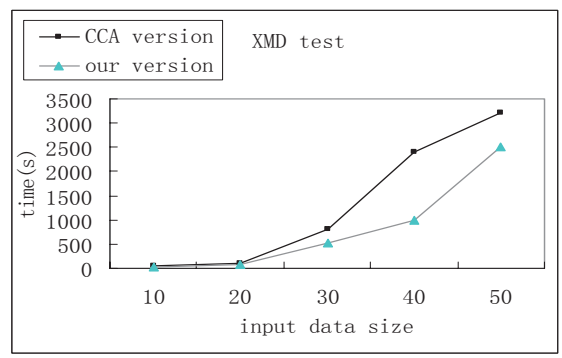

Figure 15: Real application XMD test

Then the “business” component performed some simple computation. In this procedure, the “execution” component asked the computation to pause, and waited for 20s, then continued the computation. For comparison, we tested the “business” component executed without the three optional interfaces. The comparison test slept for 20s after all computation had finished. The results are in Figure 14. In this test, all transactional operations were success.

We assumed this is the most common condition that our transactional mechanism would meet. And we can see that the additional cost brought by these interfaces was very small. If only the methods implemented by the component developer don’t bring too much cost, this management is very effective.

For real applications, we used XMD molecular dynamics for metals and ceramics program as our test example [18]. We componentized this program using the CCA tools [19]. For radiation damage simulation, we varied the particle number and iteration times as different input data sizes. We have four components in this application. A “Driver” component invoked a “Readcommand” component. This component read the input data from a file and sent these data to two other components. A “Neighborlist” component performed the “neighborlist” and force calculation. And an “Integration” component distributed the particles to different domains. The “Neighborlist” component had a “performance” interface, a “requirement” interface and a “deployment” interface. This component is computation-intensive. We defined that it needed a CPU with a frequency faster than 2.5 GHZ. The “requirement” got this resource requirement and sent it to the “deployment” component. The “deployment” component generated the best deployment policy. For comparison, we also tested this application without our non-functional attributes management mechanism in CCA version. Figure 15 shows the test results.

From these tests, we can see that our extension provided a convenient way for the management of non-functional attributes of parallel components application. This mechanism can improve the performance of parallel components applications. And it can provide useful management of the execution procedure without too much additional cost.

5. Conclusion

In this paper, we present a unified mechanism for management of multiple non-functional attributes of parallel components. We defined and implemented the common parts of this management as nine non-functional components. We also defined the management specific to the functional components. This part was defined as some optional non-functional interfaces. We described the management mechanism. We also gave some example implementations of some parallel components providing these interfaces. Our test shows that our management can improve the performance of parallel components. The tests of management of load balance, adaptive, schedule, and deployment attribute show much better performance than the same applications without them. And the test of execution procedure control shows that our control mechanism is very effective. A test of XMD [13] molecular dynamics shows the use of our proposal in real application. In the future, we will try to Triana units [20] for functional components in this paper. This may help the users of workflow applications.

Acknowledgements

This work is sponsored by the National High Technology Research and Development Program ("863"Program) of China (Contract no. 2008AA01Z109), and by Major National Science & Technology Specific Projects (Contract no. 2009ZX03004-004 and 2009ZX01045-005-002), and by Key Project of Chinese Ministry of Education under Grant No. 108008.

Reference

1. R. Abernethy, R. Morin, J. Chahin, COM/Dcom Unleashed (IN, USA: Sams Indianapolis, 1999).

2. S. Vinoski, CORBA: integrating diverse applications within distributed heterogeneous environments, Communications Magazine (IEEE), 35(2), 1997, 46-55.

3. B. Burke, R. Monson-Haefel, Enterprise JavaBeans 3.0 (CA, USA: O'Reilly Media, Inc, 2006).

4. Yves Mahéo, Frédéric Guidec, Luc Courtrai, Middleware support for the deployment of resource-aware parallel Java components on heterogeneous distributed platforms, Proceedings of the 30th EUROMICRO Conference, Washington, DC, USA: IEEE Computer Society, 2004, 144-151.

5. Nathalie Furmento, Anthony Mayer, Stephen McGough et al. A Component Framework for HPC Applications, Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, London, UK: Springer-Verlag, 2001, 540-548.

6. Lei Zhao, Stephen A Jarvis, Spooner Daniel P, et al. Predictive Performance Modelling of Parallel Component Compositions, Cluster Computing, 10(2), 2007, 155-166.

7. David E. Bernholdt, Benjamin A. Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, et al. A Component Architecture for High-Performance Scientific Computing, International Journal of High Performance Computing Applications, 20(8), 2006, 163-202.

8. Scott R Kohn, Gary Kumfert, Jeffrey F Painter, Calvin J Ribbens, Divorcing language dependencies from a scientific software library, In Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, 2001[CD], Philadelphia, PA, USA: SIAM Press, 2001, 10 pages on CD-ROM.

9. R. Marau, L. Almeida, P. Pedreiras, K. Lakshmanan, R. Rajkumar. Utilization-based schedulability analysis for switched Ethernet aiming dynamic QoS management. 2010 IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2010, 1-10.

10. Bice Cavallo, Massimiliano Di Penta, Gerardo Canfora. An empirical comparison of methods to support QoS-aware service selection. PESOS’10

Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, 2010, 64-70.

11. Francisca Losavio, Ledis Chirinos, Maria A. Perez. Quality models to design software architectures. Proceedings of International Conference on Technology of Object-Oriented Languages, 2001: 123.

12. Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, et al. Reflection, Self-Awareness and Self-Healing in OpenORB. Proceedings of the first workshop on Self-healing systems, 2002: 9-14.

13. Sun Microsystems. Java Reflection. URL: http://java.sun.com/j2se/1.3/docs/guide/reflection/index.html.

14. cca-forum. cca-tools. http://www.cca-forum.org/software/index.html.

15. Buyya R, Abramson D, Giddy J. Economy driven resource management architecture for computational power Grids. Proceedings of the 7th International Conference on Parallel and Distributed Processing Techniques and Applications, 2000. URL: http://www. buyya. Com/papers/GridEconomy.pdf.

16. M. H. Willebeek-LeMair, A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. IEEE Transations on Parallel and Distributed Systems, 1993, 4(9), 979-993.

17. Robert D. Kent, Ziad Kobti, Anne Snowdon, Akshai Aggarwal. Towards a Unified Data Management and Decision Support System for Health Care. Intelligent Interactive Multimedia Systems and Services, Smart Innovation, Systems and Technologies, 2010, (6), 205-220.

18. A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, et al. Performance Technology for Parallel and Distributed Component Software, Concurrency and Computation: Practice and Experience, 17(2-4), John Wiley & Sons, Ltd. 2005, 117-141.

19. Jon Rifkin. XMD-molecular dynamics for metals and ceramics. http://xmd.sourceforge.Net.

20. Vlado Stankovski, Martin Swain, Valentin Kravtsov, Thomas Niessen, Dennis Wegener, et al. Grid-enabling data mining applications with DataMiningGrid: An architectural perspective. Future Generation Computer Systems, 2008, 24(4): 259-279.