Scalable Computing: Practice and Experience
Volume 10, Number 4, pp. 419–428. http://www.scpe.org
ISSN 1895-1767

LOAD BALANCING METRICS FOR THE SOAJA FRAMEWORK

RICHARD OLEJNIK*, IYAD ALSHABANI*, BERNARD TOURSEL*, ERYK LASKOWSKI†, AND MAREK TUDRUJ†‡

Abstract. The paper describes system and program metrics used for load balancing algorithms for Java program execution in the SOAJA (Service Oriented Adaptative Java Applications) executive environment. This environment aims in maintaining design and execution of large scale computing tasks in complex networked Grid environments. SOAJA services provide means for static and dynamic load balancing with the use of special metrics obtained by Java object observation. SOAJA comprises mechanisms and algorithms for automatic placement and adaptation of application objects, in response to evolution of resource availability. Under control of SOAJA, parallel Java objects can be optimally allocated to Grid nodes before execution and next migrated at runtime to less loaded nodes to maintain the balance of loads of constituent JVMs. SOAJA mechanisms employ computation power metrics based on measurements of the idle time of processor nodes and communication bandwidth metrics for network resources based on statistical assessment of the existing traffic. Due to these mechanisms the granularity of computing and distribution of the application elements on the Grid platform can be optimally controlled.

Key words: service-oriented architecture, adaptive software, load balancing, grid computing, distributed computing

1. Introduction. The execution of an irregular distributed application usually generates a load imbalance, which brings down the overall effectiveness of the execution platform. Additionally, in a multi-user Grid context, the availability of computing resources can notably vary over time. Thus, an optimization subsystem, embedded in the execution environment (or in the distributed application) is essential.

Load balancing is one of important procedures applied to heuristically optimize execution time of parallel programs. A general classification and an overview of load balancing methods are presented in [4, 3]. The paper deals with an asynchronous approach to load balancing, in which the load balancing activities are performed in parallel with computations. Load balancing can be further divided into static load balancing where an computational load is partitioned among executive units by an algorithm executed before program execution and dynamic load balancing, where the load decomposition is adaptively changed during computations, following the system resources availability.

This paper is concerned equally with static and dynamic load balancing. In Java-based computing on Grid static load balancing has received relatively small attention. In this paper we present a two phase approach to load balancing of Java programs execution in Grid. The first phase consists of a static load balancing, which determines an initial deployment of application Java objects over the network of Java Virtual Machines. This static load balancing algorithm scenario includes execution of the application for a representative set of data to be able to detect some static properties concerning computational and communication aspects and to be able to use this properties for an initial deployment of program elements before execution. This phase of load balancing is based on tracing of the load of virtual machines and method invocations.

The second phase of the load balancing process is dynamically organized during program execution. It is based on three basic operations: JVM load observation, detection of the load imbalance and load migration if the imbalance exists. A dynamic agent approach is used to implement these operations. An observation system of the execution environment and the distributed application is an important part of dynamic load balancing. The system observes two areas:

a) the load of different nodes within the platform and the network,
b) the evolution of the applications.

Using the information provided by these observations, several metrics to detect and compute the load imbalance of processors have been proposed in the paper. They differ from standard measures known in the literature [2].

This paper describes SOAJA overall architecture and then deals with its internal concepts. The rest of the paper is composed of 5 parts. In the first part the general assumptions for the SOAJA framework are presented. Part 2 explains the use of web services in SOAJA. Part 3 discusses the relations between web services and functions of DG-ADAJ. Part 4 describes the load imbalance detection mechanisms in SOAJA. Part 5 describes the load imbalance correction mechanisms.

*Computer Science Laboratory of Lille (UMR CNRS 8022), University of Sciences and Technologies of Lille, France ({Richard.Olejnik, Iyad.Alshabani, Bernard.Toursel}@lifl.fr)
†Institute of Computer Science Polish Academy of Sciences, Warsaw, Poland ({laskowsk, tudruj}@ipipan.waw.pl)
‡Polish-Japanese Institute of Information Technology, Warsaw, Poland

2. SOAJA and ADAJ Frameworks. The SOAJA (Service Oriented Adaptative Java Applications) is a Java framework, which provides the infrastructure, components and services enabling a platform-independent execution of complex distributed data mining workflows [7]. It manages data access and resources sharing, including databases, information systems and hardware resources access. It supports resource discovery and provides context-aware recommendations for the dynamic composition of data mining operations and workflows. The underlying agent-based layer of the SOAJA infrastructure provides means to orchestrate very large, heterogeneous and dynamic workflows hardware with optimized load balancing across multiple nodes, constituting the Grid. The main services of the framework are observation, measuring of the JVM load, measuring of the physical processor load, load balancing service, and parallel data services.

The SOAJA environment is deployed in a Grid infrastructure, provided that an JVM is installed on each processor node. The SOAJA is the extension of the ADAJ environment to make it scalable on the Grid and to support service oriented architecture [5].

2.1. General Architecture of ADAJ. The design of distributed programs and optimization of their performance is not a simple task. Indeed, a programmer must consider construction of its application in the most effective way, while taking account of heterogeneity of the executive environment. To overcome these encumbrances, the ADAJ framework has been proposed [1].

ADAJ (Adaptative Distributed Application in Java) is a programming and execution environment for parallel and distributed Java applications. Design objectives of ADAJ are as follows: simplifying the programmer's work by hiding problems related to the management of parallelism, facilitating the development of applications and allowing their automatic or quasi-automatic deployment in heterogeneous environments, ensuring effective implementation of parallelism, by mechanisms of inter- and intra-applications load balancing.

The ADAJ provides mechanisms at middleware level that enable dynamic and automatic adaptation of computations to the structure of the executive platform and changes in resource availability. ADAJ includes four major features: a library containing the necessary tools to facilitate parallel programming, an observation system which scans the environment during its execution and retrieves the information necessary to optimize the program, a system that calculates the load of JVM and physical processors using the information gathered by the observation system, a system that allows correcting load imbalance by objects migration based on information from the observation and the calculation of loads.

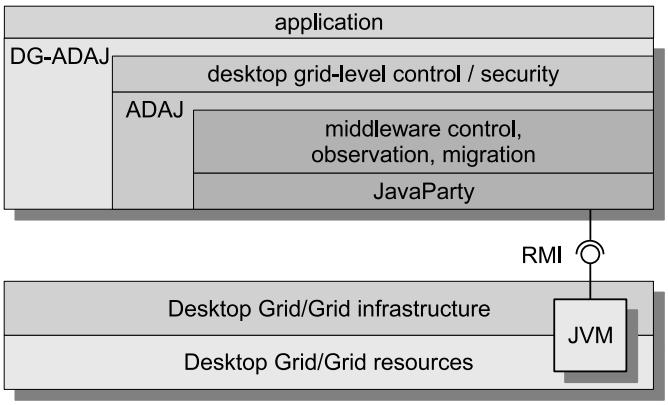

Fig. 2.1. *ADAJ layered architecture*

The ADAJ architecture is based on the multi-layered design pattern. The internal structure of the ADAJ consists of the following layers (Fig 2.1):

1. distributed programming layer (DPL) — JavaParty
   (a) provides a notion of global object (remotely accessible),
   (b) provides migration facility (used to move global objects between JVMs),
   (c) deploys an application on computing nodes.
2. load balancing layer — ADAJ core
   (a) constructs and maintains dynamic relationship graph, based on global objects' placement and observation data,
   (b) detects load imbalance and instructs local agents to equilibrate the load,

(c) depends on global objects, migration and deployment infrastructure provided by JavaParty.

3. application layer — Distributed Collections (Fragments)
   (a) provides an application construction framework, which supports programmers and eases distributed application development.

The current implementation of the ADAJ is based on JavaParty [14], which facilitates the usage of RMI protocol by application programmers. The JavaParty allows execution of distributed Java applications on workstations connected via a network. JavaParty has introduced the concept of remote objects that can be distributed in a transparent manner. It compensates for the drawbacks of the RMI protocol because it conceals the addressing and communication mechanisms. Using JavaParty, it is sufficient to annotate Java classes with the word `remote` that will give access to remote objects from any part of an application without publishing them explicitly in service names space as RMI.

DG-ADAJ is a version of the ADAJ environment implemented for Desktop Grids. Initially, DG-ADAJ was conceived as an extension of the ADAJ system (see [10]), built for cluster computing. It has been re-engineered to extend its for larger scale distributed computing and to introduce some special security mechanisms, which provide reliable application execution in Desktop Grids [12].

2.2. Web Services in SOAJA. SOAJA uses WSRF [9] standard and allows instantiation of statefull services for applications. The statefullness of services is exploited by the application clients that allows asynchronous communication among software components instantiated for the client on different platforms.

The Fig. 2.2 illustrates the main web services that constitutes the SOAJA environment.

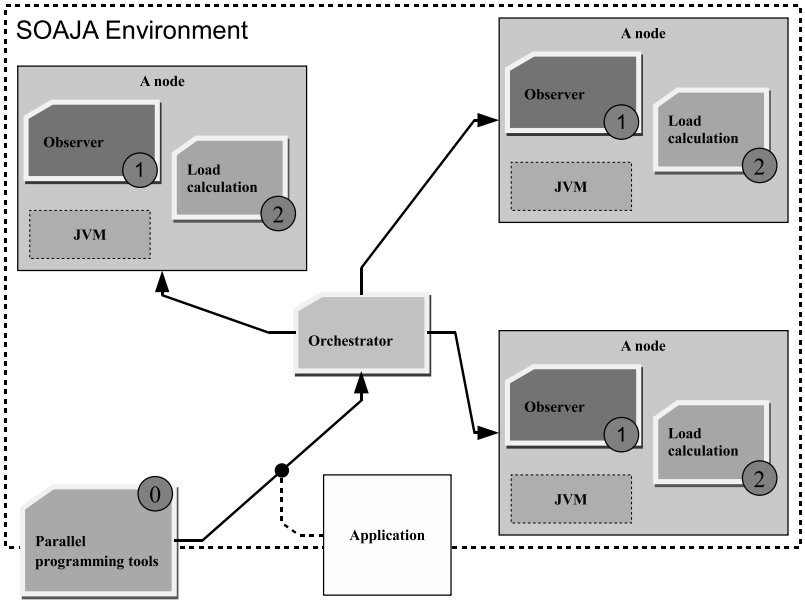

Fig. 2.2. *SOAJA environment*

Various architectural styles and middleware components can be used for the implementation of a SOA-based environment. The Enterprise Service Bus (ESB) is a middleware technology which provides the necessary features with which it is possible to implement SOA. In a typical ESB usage scenario, the ESB layer itself is deployed over the existing infrastructure. Based on this infrastructure, the ESB layer offers the necessary support for transport interconnections while it exposes the existing subsystems through a specific set of adapters. With the help of the ESB as a SOAJA implementation technology, services are exposed in a uniform manner, using open standards, so that any client, who is able to consume web services over a generic or specific transport, is able to access them. For example, the SOAJA has been used to implement data mining application in the WODKA project [11].

SOAJA is a platform developed on top of DG-ADAJ, by adding the web services layer and gaining SOA-specific properties, Fig 2.3. With computing grid as a main target, the SOAJA platform provides a uniform and transparent interface to the infrastructure of the DG-ADAJ platform. The SOAJA platform, through its

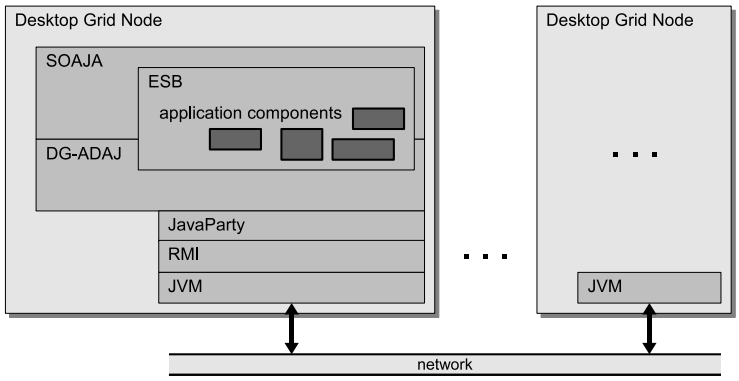

FIG. 2.3. *SOAJA layered architecture*

orchestration layer, provides also a possibility to execute single-cluster applications in the DG-ADAJ environment [1, 13]. With DG-ADAJ executions controlled via an ESB, the orchestration layer is able to offer both the support for execution of complex compositions, as well as elementary execution of applications in the underlying DG-ADAJ environment. This way, the SOAJA environment hides implementation details of different Grid environments behind the set of various web services, offering the necessary support for integration, interoperability and reliable messaging. As a side effect, by employing the orchestration layer, we enable a programming-in-the-large paradigm necessary to assure the development and support of long living, asynchronous processes.

The SOAJA platform is going to help efficient execution of heterogeneous applications enabled by DG-ADAJ by offering basic support for workflow deployment and enactment. The DG-ADAJ environment could thus be freed from some of the placement, distribution and execution tasks, by moving significant parts of these to the higher layer of the ESB and the enactment environment. With SOAJA, the ability to design component-based and service oriented applications, developed over the DG-ADAJ platform is extended with the help of web services open standards, by offering the possibility to access both local and remote components, eventually deployed on different DG-ADAJ environments.

3. Program execution optimization in SOAJA. Load balancing is the fundamental procedure applied in order to optimize execution time of parallel programs. In the case of the architecture of the SAOJA environment, the load balancing is achieved by such a distribution of the application components (objects) among active Grid nodes that guarantee a possibly high efficiency of the overall application execution. To make the load balancing effective during the whole run-time of an application, the SOAJA employs equally both static and dynamic load balancing. This is accomplished by the optimization of the following two aspects of application execution: initial objects deployment and dynamic load balancing.

3.1. Initial object deployment optimization. An optimization feature of SOAJA is an initial application objects deployment on JVMs, which results in a shorter execution time. The initial placement of application objects to JVMs is the service of the orchestrator shown in Fig. 2.2. The initial object deployment optimization algorithm follows the pattern of static parallelization method in multithreaded Java program. It defines decomposition of Java code into parallel threads distributed on a set of JVMs, so as to reduce program execution time [13]. The control decisions are taken to determine which of the classes should be distributed and what the mapping of objects and data components (fragments) should be to JVM nodes, to reduce direct inter-object communication and to balance loads of the JVMs. When applied to an application run under DG-ADAJ control, it will determine an initial distribution of its objects among Java Virtual Machines (JVMs) assigned to Grid active nodes, thus leading to a reduction of the total execution time.

The proposed optimization algorithm employs an image of application program, based on an analysis of the byte code generated by Java compiler. This analysis identifies control dependencies between byte code instructions. They are represented in adequate MDG (Method Dependence Graph) and MCG (Method Call Graph) graphs of the program [6, 13]. The number of mutual method calls and the number of thread spawns during program execution for representative input data are measured using observation mechanisms described later in the paper. Program behavior, including all created objects in each class, all called methods, as well as all spawned threads, are registered in trace files.

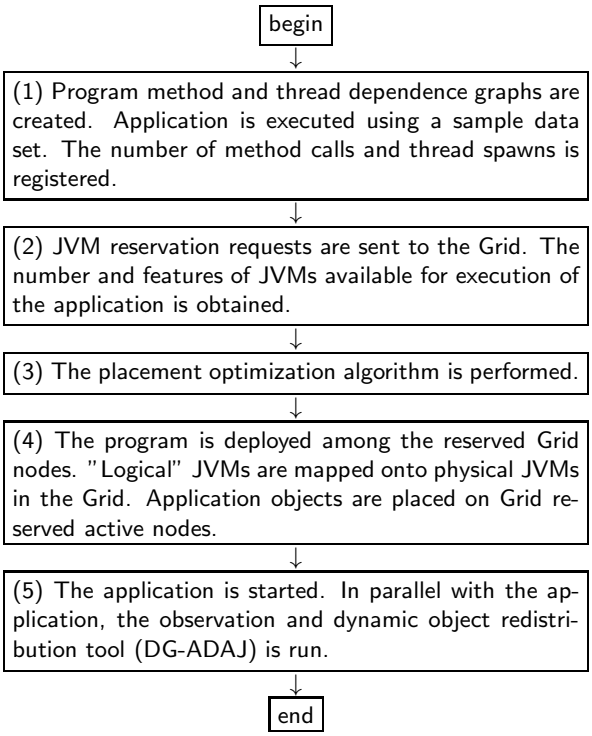

FIG. 3.1. *The control flow of an application execution.*

The flow of actions during application execution is shown in the diagram in Fig. 3.1. The first three blocks in the diagram determine an initial optimized placement of application objects and perform the respective objects distribution over Grid on JVMs nodes. This part of the algorithm starts with execution of the application using some representative sample data. For that, the number of available JVMs nodes on the Grid must be known. The number of method calls and spawned threads that have appeared during execution is recorded. Next, the program method and thread dependence graphs are annotated with the recorded data.

At the beginning of the object deployment optimization algorithm we treat all objects as remote objects (respectively all classes are distributed classes). Based on the recorded control data, the algorithm decides which classes should remain distributed and how the involved objects should be placed on a set of JVMs assigned to active nodes on the Grid. We use the heuristics based on the following principles: the strong locality of method calls has to be preserved inside each parallel thread and the number of inter-thread calls, which cross the boundaries of JVMs has to be reduced. To fulfill such object requirements we designate all calls inside a single thread to the same JVM and optimize distribution of threads across available sets of JVMs by applying load balancing methods.

The algorithm consists of two phases (see [6] for details). In the first phase the MCG graph is traversed in the DFS (Deep-First-Search) manner to agglomerate method calls executed in single thread. In this step, the algorithm finds the MCG subgraphs, which are constructed of vertices connected by edges connecting calls inside threads. We assume that each subgraph is executed on a dedicated JVM. Subgraphs are built at the level of single objects. In case when an object belongs to different subgraphs, the new subgraph is constructed and a unique JVM number is assigned to it. We expect that, in most cases, at the end of this phase, the number of found subgraphs is far bigger than the number of available JVMs in the system.

In the second phase of the algorithm, we clusterize the subgraphs obtained in the previous phase until the number of clusters is equal to the number of JVMs. The general outline of this phase is similar to Sarkar's [15] edge-zeroing clustering heuristics. At each clustering step, the algorithm finds the subgraphs, which are connected by edges with the biggest weight value and which connect nodes placed on different JVMs. All nodes of those subgraphs are assigned to the same JVM while the total number of remote calls decreases. The algorithm stops when the number of clusters is equal to the assumed number of JVMs.

The first phase of the algorithm makes that method calls inside programmer-declared threads are local to the JVM. This allows exploiting thread level parallelism without introducing large inter-JVM communication overheads. The gain of the second step of the heuristics comes from reduction of RMI calls to remote objects.

3.2. Load balancing strategy. The workstations used in the network are heterogeneous, but they have different and variable computing capabilities over time. The load imbalance occurs when the differences in workload between the workstations become too big. An application execution may not be optimal because some workstations have too much work and the others have not enough. Let's consider that the network works correctly and that the DG-ADAJ environment is running. We distinguish two main steps in load balancing: detection of imbalance and its correction, if necessary. The first step uses measurement tools to know the functional state of workstations. The second consists in migrating of the load from overloaded workstations to underloaded workstations in order to balance the workload.

The observation mechanism of applications in the DG-ADAJ environment aims at providing knowledge of the applications behavior during their execution. This knowledge is gathered by observing activity of constituent objects.

There are two types of objects in DG-ADAJ (Fig. 3.2):

global objects These are global objects that can be created remotely in any JVM. They are remote accessible. There is only one copy of a global object in all environment. The global objects are also migratable, i. e. they can be moved from one JVM to another.

local objects These objects are traditional Java objects. They can be used in only one JVM at the place where they reside. If another JVM needs such object, it will create of a new copy of the object concerned. Obviously, local objects cannot be migrated.

We observe only global objects as the observation and migration of all objects would generate a considerable work overhead compared to the profits brought by load balancing.

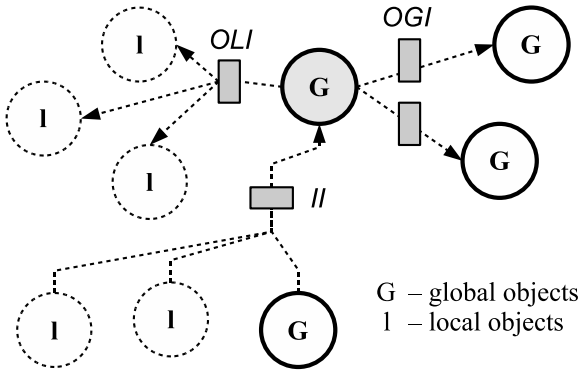

FIG. 3.2. *Relations between objects*

4. Detection of load imbalance. In this first phase, the purpose is to obtain knowledge about of the functional state of workstations composing the cluster. As the environment is heterogeneous, it is necessary to know not only the load of workstations but also their working capabilities. Such capability is directly related to the workstation computing power, so we have to estimate it. This measurement needs to be made only once when the workstation joins the system. We call it *the calibration* in this paper.

We then need the workstations workload measurement at a given time. For this purpose, we observe this part of the CPU time that is used by DG-ADAJ applications. However, such measurement is not normalized and cannot serve for comparisons between different workstations. It is thus necessary to balance it against different computing power of the workstations. After a series of measures, we compare the values found on different machines and we determine if there is a load imbalance.

The heterogeneity disallows us to compare measurements taken on workstations whose computing powers are different. Before we are able to compare the workstations load, we have to normalize the measurements. The normalization of the workloads is made by means of the power indications. After experiments to determine the workstation power, we found that the formula, which allows us to compare the workstations load is the

product of the power index and the CPU time use rate by a thread, which gives the availability index of a CPU:

$$Ind_{availability} = Ind_{power} * \%Time_{CPU}$$

At this point, we have availability indices of all workstations. By comparing these indices, we will be able to detect the load imbalance. An imbalance is characterized by too big dispersion of the availability indices of workstations composing the network. It is difficult to fix a threshold not to be overpassed to characterize a load imbalance. But we can define an interval in which the dispersion of the availability indices remains acceptable at a given moment. We are going to be interested in the gap between the minimal and maximal availability indices found during a series of measures. If the distance between these values is too big, we conclude that there is a load imbalance, which can to be considered using the following condition:

$$Stability = (\max(Ind_{availability}) \leq \alpha * \min(Ind_{availability}))$$

If this inequality is verified, the availability indices are close enough not to present a too big imbalance. Otherwise, the range of the values is too big and an imbalance of workload between workstations is detected. The central point of this inequality is the value taken by the coefficient α. We tried to clarify it first by using statistical tools then by using an experimental way. We cannot finally give a unique value for this coefficient. Nevertheless, we can restrict its value to the interval $[1.5 \ldots 2.5]$. These experimental values give good results because they are neither too restrictive nor too tolerant for the load imbalance.

The tolerance for the load imbalance among all workstations depends directly on the value of the coefficient α. The smaller this coefficient is, the more we are demanding as to the load balance between workstations. Indeed, a low coefficient allows only a small difference between loads of workstations. This can be seen as a guarantee of the quality of the balancing results. However, a low tolerance of the imbalance leads to much more frequent detection of the imbalance. The consequence is to activate the mechanism of load balancing more frequently and therefore make it more expensive in time. Not only the migrated objects can not do their computing, but they can also block the execution of others objects, which are awaiting results. In addition, the coefficient α is to be chosen according to the number of machines in the network. When this number is big, the coefficient must be increased and vise versa.

5. Correction of load imbalance. In this phase, we are in the state in which a load imbalance has been detected. To correct this imbalance, we have to classify the workstations into three categories according to their availability index:

1. **Overloaded** workstations: availability indices are low,
2. **Normally loaded** workstations: availability indices are medium,
3. **Underloaded** workstations: availability indices are high.

The purpose of the load balancing is to transform the workstations categories: overloaded and underloaded into the category normally loaded. To do it, we need to migrate the workstations load from the first category to the third one.

5.1. Classification of workstations. We use the K-Means algorithm [8] to build the categories of workstations based on the computed availability indices. The K-Means algorithm allows to classify a distribution of n values into k categories by choosing k centers for categories. We want to classify workstations into the categories: overloaded, normally loaded and underloaded. For this, we use this algorithm by taking the computed availability indices and $k = 3$, to obtain 3 categories finally.

The three centers that we choose are the minimum, average and maximum availability indices. The average index is simply the average of indices measured during the last series of measures over the whole network. By comparing the distances of workstations availability indices from the three centers, the three categories of workstations will be identified. The center represented by the minimum index builds the category of overloaded workstations, the center using the average one builds the category of normally loaded workstations and finally the center based on the maximum index is used to build the category of underloaded workstations. The important thing is therefore to have the overloaded and underloaded categories in order to be able to move the load from the overloaded workstations to the underloaded workstations.

5.2. Choice of candidates for migration. To correct load imbalance, we have to migrate the load from overloaded workstations to underloaded workstations. Firstly, we must identify the load that we want to migrate. The loads are represented by the activities of the objects which are running on the JVMs. We need

to select an object on each JVM in the overloaded nodes. Let's see how to choose such an object, so that its migration changes the load balance in the network.

The migrated entity is necessarily a global object because it is not intrinsically linked to the JVM on which it currently runs. Among the global objects in a JVM, some of them have more suitable characteristics to be migrated. These characteristics are related with the other computer objects and the load quantity carried out. Two relations are involved:

a) the attraction of a global object to the JVM,
b) the weight of the global object.

The attraction of a global object to a JVM is expressed in terms of communication links, which it shares with other global objects inside the same JVM on the computing node. A strong attraction involves frequent communication, which will be realized as remote communication after object migration. This communication will then have a higher cost than the current cost. A small attraction will permit to leave a global object the current computing node and to run it on another one, without introducing significant amount of additional remote communication. So, the less the object is attracted by the current JVM, the more interesting it is to be selected as a migration candidate.

The computational weight of migrated global object gives the quantity of load to be removed from the current machine. An object, whose quantity of work is big i. e. shows a continuous activity, should not be migrated. In addition, by migrating a too big quantity of work (load), we could reverse the role of the involved machines (the source will become target and vice-versa). In contrast, the migration of an object with small quantity of a work does not bring significant load variation improvements. Furthermore, the migration cost will not be compensated with the new generated load distribution. In conclusion, the decision should be to move an object whose quantity of work is neither too big, nor too small. Thus, the smaller the distance is to the average object loads, the more the object is interesting for migration.

The observation of the objects activity is done by counting the activation methods. These activations can be done by global or local objects. The observation of a global object (only these objects are observable), includes (see Fig. 3.2):

1. observation of object invocations to each global object, including him: **OGI** (OutputGlobalInvocation),
2. the observation of object invocations to all local objects: **OLI** (OutputLocalInvocation),
3. the observation of others objects invocations to the considered object: **II** (InputInvocation).

The attraction of the global object *obj* to the actual JVM:

$$attr(obj) = \sum_{o \in JVM} (OGI(obj, o) + OGI(o, obj))$$

Distance compared to the average quantity of work of the *obj*:

$$dist_{m_{WP}}(obj) = |WP_{obj} - m_{WP}|$$

Where $m_{WP} = \frac{\sum_{o \in JVM} WP_{obj}}{n}$ (n is the number of global objects on the JVM) and

$$WP_{obj} = OGI(obj, obj) + II(obj) + OLI(obj)$$

These formulas allow to compute the attraction of an object to the local JVM in order to compare it with the attractions of other objects of this JVM. The comparison formulas are:

1. Global attraction measure:

$$\%attr(obj) = \frac{attr(obj)}{\sum_{o \in JVM} attr(o)}$$

2. Migration distance, compared to the average quantity of work of the object:

$$\%dist_{m_{WP}}(obj) = \frac{dist_{m_{WP}}(obj)}{\sum_{o \in JVM} dist_{m_{WP}}(o)}$$

The most interesting object to migrate is selected based on the weighted sum of these relations:

$$Classification(obj) =$$
$$\alpha_{attr} * \%attr(obj) + (1 - \alpha_{attr}) * \%dist_{m_{WP}}(obj)$$

α_{attr} is a real between 0 and 1. Its choice remains experimental. Let us notice however that the bigger α_{attr} is, the bigger is the weight of the object attraction.

5.3. Selection of the target for migration. The migration of global objects reduces the load of a JVM running on an overloaded workstation and therefore the global load cost of this workstation. The question now is: where migrate these objects? Naturally, the potential destinations are one or more underloaded workstations. However, the choice of one of these computing nodes can be more or less convenient from the point of view of work and communication quantity of the object to migrate. For an object selected for migration, we must find the best target according to these criteria.

The first criterion to quality as a target is the attraction of the selected object to this workstation. We say that the attraction of an object—candidate for migration, is big when it communicates a lot with the global objects in the target JVM. A relationship of attraction of the global object *obj* to JVM_i is defined as follows:

$$attrext_i = \sum_{objext \in JVM_i} (OGI(objext, obj) + OGI(obj, objext))$$

The more the object is externally attracted by an underloaded machine, the more it is interesting that this machine is chosen as a migration destination for it. This criterion should not be the only one during selection of target for migration. In fact, it is possible that two underloaded workstations have the same amount of communication with the candidate for migration. In this case, the second criterion will be the workstations' availability indices. We naturally prefer the one whose availability index is the highest, because it is actually the least loaded.

To complete discussion of the criteria for choosing a target for migration, we should take into account the number of *(waiting)* threads in the JVM of the potential targets. We consider them, however, as potential load, which must be taken under consideration with the related load currently done on the machine.

These three points are obvious constraints to be met by the target machine:

1. the external attraction of the object is maximal to the target machine,
2. the quantity of work on the target machine is minimal,
3. the number of waiting Java threads is minimal.

These three conditions are gathered in a formula in order to designate the underloaded workstation, which is the most favorable for the selected object migration. Firstly, we have to normalize all the values related in the interval $[0 \dots 1]$. We then obtain:

$$\%attrext_i = \frac{attrext_i}{\sum_j attrext_j}$$

$$\%Ind^*_{availability_i} = \frac{Ind^*_{availability_i}}{\sum_j Ind^*_{availability_j}}$$

where

$$Ind^*_{availability} = Ind_{availability} - Ind_{availability} * \frac{NbThread_{wait}}{NbThread_{total}}.$$

The availability index was corrected by the potential work of the workstation, represented by the waiting threads. We have not considered this index during the classification of workstations into three categories, because we want to classify workstations according to their measured quantity of work and not a potential one. Indeed, threads waiting must be seen as supplementary work about which we know nothing. They may start executing in one second quite well as in one hour. Their consideration is thus justified only when we want to examine our measurements in perspective as it is the case here. We thus chose to decrease the raw indication of availability by means of the relationship between the number of waiting threads and the total number of threads staying in the machine.

The aggregation function should account for the two described components by giving them different weights. The balanced sum of them is:

$$Quality_i = \alpha_q * \%attrext_i + \beta_q * \%Ind^*_{availability_i}$$

with α_q and $\beta_q \in [0 \dots 1]$.

For an object which is a candidate for migration, this formula is applied to all JVM potential node targets. The workstation which maximizes this sum will be chosen as new location for the object. The choice of coefficients α_q and β_q is experimental but the sum of the two must be equal to 1 (it was therefore $\alpha_q = 1 - \beta_q$). The weight of one or the other value can be increased by changing these coefficients. The migration of the object allows to eliminate communication on the network and to reduce considerably the waiting time for replies. For that purpose, the coefficient α_q must be the most important to promote the machines for which the attraction of the object is maximal. For example, we can use the coefficients $\alpha_q = 0.6$ and $\beta_q = 0.4$.

6. Conclusions. The paper has described a run-time environment SOAJA for Java program execution optimization to provide load balancing of JVMs implemented on the Grid platform. The SOAJA mechanisms first adjust the initial parallelization granularity and placement of the application program elements on JVMs. Then, the objects distribution is adjusted to the evolution of the availability of system resources at run time by object migration. To optimize the initial object deployment and JVMs load balancing the SOAJA infrastructure applies system and program metrics obtained by its special observation mechanisms. They are used by components and services to enable static and dynamic load balancing of the nodes of a Grid. The system is currently under implementation inside the GRID 5000 project.

REFERENCES

[1] I. Alshabani, R. Olejnik and B. Toursel. *Parallel Tools for a Distributed Component Framework*. 1st International Conference on Information & Communication Technologies: from Theory to Applications (ICTTA04). Damascus, Syria, April 2004.
[2] J. Cao et al., *Grid load balancing using intelligent agents*, Future Generation Computer Systems, 21 (2005), pp. 135—149.
[3] K. Devine et al., *New challenges in dynamic load balancing*, Applied Numerical Mathematics, 52 [2005], pp. 133—152, Elsevier.
[4] R. Diekmann, B. Monien, R. Preis, *Load Balancing Strategies for Distributed Memory Machines*, Karsch/Monien/Satz (ed.): "Multi-Scale Phenomena and their Simulation" World Scientific, pp. 255-266, 1997.
[5] T. Erl, *Service-oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River: Prentice Hall PTR, 2005. ISBN 0-13-185858-0.
[6] V. Felea, E. Laskowski, B. Toursel, M. Tudruj, *Optimizing Object Oriented Programs Based on the Byte Code-Defined Data Dependence Graphs*, Procs. of Concurrent Information Processing and Computing (CIPC NATO ARW), Sinaia, Romania, pp. 34–46, 2003.
[7] V. Fiolet, G. Lefait, R. Olejnik, B. Toursel, *Optimal Grid Exploitation Algorithms for Data Mining*, In Proc. of ISPDC 2006, IEEE Computer Society, July 2006, pp. 246-252.
[8] J. A. Hartigan et M. A. Wong, *A K-Means clustering algorithm*, Applied statistics, Vol. 28, pp. 100-108, 1979.
[9] OASIS Web Services Resource Framework (WSRF) `http://www.oasis-open.org/committees/wsrf/`
[10] R. Olejnik, A. Bouchi, B. Toursel. *An Object Observation for a Java Adaptative Distributed Application Platform.* Intl. Conference on Parallel Computing in Electrical Engineering PARELEC 2002, pp. 171-176, Warsaw, Poland, September 2002.
[11] R. Olejnik, F. Fortis, B. Toursel, *Webservices Oriented Datamining in Knowledge Architecture*, Accepted to publication in Future Generation Computer System (FGCS)—The International Journal of Grid Computing: Theory, Methods and Applications
[12] R. Olejnik, B. Toursel, M. Ganzha, M. Paprzycki, *Combining Software Agents and Grid Middleware*, Advanced in Grid and Pervasive Computing, C. Cerin and K.-C Li Editors, LNCS 4459, pp. 678-685, Springer Verlag, Berlin, Heidelberg, 2007.
[13] Olejnik R., Toursel B., Tudruj M., Laskowski E., *Byte-code scheduling of Java programs with branches for desktop grid*, Future Generation Computer Systems, Vol. 23, Issue 8, November 2007, pp. 977-982, ©Elsevier Science.
[14] M. Philippsen, M. Zenger. *JavaParty—Transparent Remote Objects in Java.* Concurrency; Practice & Experience, Vol. 9. No. 11. pp. 1225-1242. November 1997.
[15] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, 1989.

Edited by: Dana Petcu
Received: September 30th, 2009
Accepted: November 3rd, 2009