JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications *

Thomas Fahringer
Institute for Software Science, University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
tf@par.univie.ac.at

Appeared as distinguished paper in IEEE Intl. Conf. on Cluster Computing, CLUSTER 2000, Chemnitz, Germany, Dec. 2000

Abstract

Most Java-based systems that support portable parallel and distributed computing either require the programmer to deal with intricate low-level details of Java which can be a tedious, time-consuming and error-prone task, or prevent the programmer from controlling locality of data. In this paper we describe JavaSymphony, a programming paradigm for distributed and parallel computing that provides a software infrastructure for wide classes of heterogeneous systems ranging from small-scale cluster computing to large scale wide-area meta-computing. The software infrastructure is written entirely in Java and runs on any standard compliant Java virtual machine.

In contrast to most existing systems, JavaSymphony provides the programmer with the flexibility to control data locality and load balancing by explicit mapping of objects to computing nodes. Virtual architectures are specified to impose a virtual hierarchy on a distributed system of physical computing nodes. Objects can be mapped and dynamically migrated to arbitrary components of virtual architectures. A high-level API to hardware/software system parameters is provided to control mapping, migration, and load balancing of objects. Objects can interact through synchronous, asynchronous and one-sided method invocation. Selective remote classloading may reduce the overall memory requirement of an application. Moreover, objects can be made persistent by explicitly storing and loading objects to/from external storage.

A prototype of the JavaSymphony software infrastructure has been implemented. Preliminary experiments on a heterogeneous cluster of workstations are described that demonstrate reasonable performance values for a small test program.

1. Introduction

Distributed and parallel computing have been investigated for many years but recently research on this topic has gained new impetus due to the explosive growth of the Internet on the one hand, and the availability of the portable programming language Java [1] on the other hand. Java is very popular due to its ability to support code mobility, object-orientation, portability, multi-threading, synchronization mechanisms, and communication APIs. Although there is no agreement whether Java can ever satisfy the needs of high-performance driven applications, it is commonly used to exploit medium grain shared memory parallelism based on threads as well as medium to coarse grain data and task parallelism based on remote method invocation (RMI) and socket communication.

Much work has been conducted in the area of improving Java Virtual Machine (JVM) implementations (for instance, Java RMI or object serialization), providing a high-level veneer that removes some of Java's RMI and/or socket communication complexity, and introducing language extensions or class libraries to support transparent distributed objects. Most research projects that offer a software infrastructure for high-level distributed and parallel programming, however, lack any programmer control over locality of data. Automatic distribution and dynamic migration of objects can easily lead to significant performance degradation as the underlying runtime system has little information about the distributed Java application. Frequently only system load and application monitoring is considered for distribution of objects. Monitoring based systems may detect performance inefficient mappings of objects and high interaction among computing nodes in a distributed system. However, at the time where such an effect is detected, performance has already been lost and probably expensive migration must be invoked to reduce performance degradation. Most of the time, programmers are very much aware of the particular nature of their application, how to distribute objects, which objects should be mapped together with other objects, when to migrate objects, etc. Programming paradigms that do not allow to specify this information lose a strong potential for increased performance.

In this paper we introduce JavaSymphony, a programming paradigm for distributed and parallel computing that provides a distributed and parallel computing infrastructure for wide classes of heterogeneous systems ranging from small-scale cluster computing to large scale wide area meta computing. JavaSymphony provides a class library which is entirely written in Java and runs on any standard compliant JVM. The key features of JavaSymphony – currently not included in Java – which substantially alleviate performance-oriented distributed and parallel programming:

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

- **Dynamic Virtual Distributed Architectures:** The programmer can dynamically define and modify virtual distributed architectures that impose a virtual hierarchy on a distributed system of physical computing nodes. Virtual architectures consist of a set of components: computing nodes, clusters (collection of nodes), sites (collection of clusters), and domain (collection of sites). Virtual architectures can be restricted by a system of constraints in order to include only those computing resources that satisfy the needs of an application (hardware/software requirements) and honors the policy of a computing site (e.g. only use idle workstations). Multiple virtual architectures can be defined that possibly share common architecture components.
- **Access to system parameters**: JavaSymphony provides a high-level API to a large variety of system parameters, including CPU load, idle times, available memory size, number of processes and threads, network latency, network bandwidth, etc. These system parameters can be requested from the JavaSymphony runtime system (JRS) and are commonly used to define constraints for requesting virtual architectures and/or controlling mapping, migrating, and load balancing of objects.
- **Automatic and User-Controlled Mapping of Objects**: The programmer can control the creation and mapping of objects to specific components of virtual architectures. Mapping of objects can be done in relation to the location of other objects. E.g a set of objects may be placed physically close to each other or even on the same processing node if they heavily interact with each other. If the programmer does not provide explicit mapping of objects, then JRS offers automatic mapping based on periodically monitored system constraints.
- **Automatic and User-controlled Object Migration**: JavaSymphony supports both automatic and user-controlled migration of objects through periodically monitoring system parameters.
- **Asynchronous Remote and One-sided Method Invocation**: Whereas all RMIs under Java are performed synchronously in blocking-mode, JavaSymphony in addition supports also asynchronous remote method invocation. A handle is returned that can be used in the future to determine the availability and access of the method's result. Moreover, one-sided method invocation is provided which eliminates the need to return any result or wait for the method to be completed.
- **Selective Remote Classloading**: Instead of replicating all Java classes to all nodes executing an application, classes may be considered to be loaded only to the nodes that actually need them. JavaSymphony supports classloading to specific architecture components. This feature can reduce the overall memory requirement of an application.

In addition JavaSymphony supports persistent objects that enable the programmer to explicitly store and load objects to/from external storage. Moreover, JavaSymphony does not require to extend Java, modify the JVM, compiler or stub compiler. A prototype software infrastructure for JavaSymphony has been implemented as an agent based system. We are currently in the process to evaluate our system through the development of several JavaSymphony applications.

The rest of this paper is organized as follows: The next section discusses related work. In Section 3 we describe and discuss dynamic virtual distributed architectures. Section 4 presents the programming model of JavaSymphony which includes register/unregister applications under JRS, generation of virtual architectures, class loading, creation, mapping and freeing of objects, method invocations, object migration, and persistent objects. Section 5 describes the implementation of JRS. Experiments are presented and discussed in Section 6. Finally, some concluding remarks are made and future work is outlined in Section 7.

2 Related Work

There is a large amount of related work which has made collaborative use of computational resources over a global network, including low-level communication systems such as MPI [8] and PVM [24] and higher-level dedicated systems, including Globus [9], Legion [12], and NetSolve [4]. Although these systems offer heterogeneous collaboration of multiple systems in parallel – some of them in wide-area setting – they involve rather complex maintenance of different binary code, multiple execution environments, etc. CORBA [19] defines a middleware that bridges distributed objects across heterogeneous environments. It allows client objects to invoke server objects across the network. All objects, as long as they expose a well-defined interface in the Interface Definition Language (IDL) that describes the services they provide to other objects, can be invoked anywhere in the network. CORBA as well as Globus and Legion can be used to build the JavaSymphony runtime system. However, we decided to use Java/RMI instead assuming that it entails less complexity and overhead.

Jini [2] provides a sophisticated technology to interconnect generic devices that provide services to other devices or users. Devices and their services register under a lookup service. Services are located by using the lookup service. Once connections are made to devices, the lookup service is no longer involved in resulting interactions between clients and servers of services. Jini could be used to build part of the runtime system of JavaSymphony (see Section 5). However, whereas JRS is currently built on a thin protocol layer to provide JavaSymphony functionality (such as providing virtual architectures), we believe that performance problems may arise by using Jini due to larger protocol overheads.

In order to overcome system complexity, several research groups introduced Java-based global computing systems that benefit by Java's platform independence. These efforts can be broadly classified into two categories. The first category concentrates on improving the implementation of JVM (e.g. Java/RMI or object serialization) [27, 26, 21, 20]. The second category extends Java with special distribution primitives and semantics or provides class libraries to alleviate the usage of Java as a distributed programming language. JavaParty [22] extends Java with a class modifier remote. Objects generated for remote classes can be distributed. JavaParty greatly simplifies RMI programming at the cost of increased complexity of the actual Java code produced. JavaParty offers transparent object migration. It is claimed that user-controlled dynamic changing of object distribution strategies is enabled without providing further details of how this is done. JavaParty also handles static methods and variables. Both JavaParty as well as other systems such as Charlotte [3] support a distributed shared memory on top of the JVM that inherently does not enable the programmer to control locality of data.

Javelin [5] and Ninflet [25] employ a three-tier architecture where the Javelin's broker, client, and hosts correspond to the Ninflet system's dispatcher, Ninflet, and the server. Clients seeking computing resources by submitting their work in form of applets, register with a broker and submit their work in the form of an applet. Hosts are donating resources, contact the broker and run applets. Javelin makes it relatively easy for a user to act as a computing server by leveraging the existing WEB technology.

Javelin [5], Jada [6] and JavaSpaces [10] can be considered as Linda derivatives which provide either none or only very limited means (compared to the functionality offered by JavaSymphony) to control locality.

ObjectSpace Voyager [11] and Aglets [15] are mobile agent systems that do not target efficient, global computing which requires extensive communication among objects. Typically these systems do not provide references to remote objects which limits interaction among objects such as remote method invocation.

Ajents [14] has influenced JavaSymphony's programming model for remote object creation, asynchronous remote method invocation and class loading. However, Ajents just as most other systems does not allow the programmer to explicitly control object locality. Ajents also does not support virtual architectures, one-sided remote method invocations, selected classloading to specific computing nodes, and access to hardware/software system parameters as introduced in JavaSymphony. Ajents, however, offers sophisticated checkpointing mechanism and allows to migrate objects while their methods are executing.

Another source of influence for JavaSymphony's programming model is the OpusJava [17, 16] system. OpusJava is a Java based framework for distributed high performance computing that provides a high level component infrastructure and facilitates a seamless integration of HPF [13] modules into distributed environments via its interface to the HPF based coordination language Opus [18]. Although the main focus of OpusJava is the interoperability of high level parallel languages, such as HPF and Java, it may be used as a pure Java framework that provides similar means for remote object creation, synchronous and asynchronous method invocation, and object migration as JavaSymphony. However, OpusJava only provides basic support for a user driven mapping of objects and does not have an elaborated concept of virtual architectures. We are currently investigating a possible combination of JavaSymphony and OpusJava, in particular we intend to employ the elaborated architectural features of JavaSymphony, such as virtual architectures and on-line status information, within the OpusJava framework.

3 Dynamic Virtual Distributed Architectures

Most programmers are well aware of how a distributed application should be structured, where to place objects, which objects interact with each other, and how to exploit locality. JavaSymphony supports automatic mapping, load balancing, and migration of objects without involving the programmer (see Section 5.1). However, fully automatic systems commonly cause poor performance results due to lack of information about the application and insufficient static and dynamic analysis. JavaSymphony, therefore, provides a semi-automatic mode which leaves the error-prone and tedious low-level details (e.g. creating and handling of remote proxies for Java/RMI) to the underlying system whereas the programmer controls the most important strategic decisions which includes:

- the setup of the virtual distributed architecture by determining which processing nodes, clusters, collection of clusters, wide-area computing infrastructure, etc. should be used for executing a distributed/parallel program. System constraints can be specified in order to include only those computing resources that satisfy the needs of an application (hardware/software requirements) and honors the policy of a computing site (e.g. only use idle workstations).
- the mapping of data in relation to other data. E.g a set of objects may be placed physically close to each other or even on the same processing node if they heavily interact with each other,
- the mapping of data (objects) onto specific processing nodes based on system constraints (e.g. nodes with a minimum amount of memory available or a maximum of CPU load)
- placement of code (Java byte-code) on specific computing nodes which reduces the overall memory requirement of an application, and

JavaSymphony introduces the concept of *dynamic virtual distributed architectures* (called virtual architectures in the remainder of this paper) which enables the programmer to define a structure of a heterogeneous (in terms of type, speed, or configuration) network of computing resources and to support mapping, load balancing, and migration of objects and code placement. Every virtual architecture (see Figure 1) defines a domain which is subdivided into nodes, clusters, and sites. At the lowest level computing *nodes* can be selected which commonly corresponds to arbitrary PCs or workstations. Several nodes can be combined to form a *cluster* which usually correspond to a local PC/workstation cluster. At the next higher level a *site* can be defined which connects a set of geographically distributed clusters for instance via WANs (wide area networks). At the highest level several sites can be combined to form a *domain* which may define a large computational grid that can be distributed across several continents. Note that every node belongs to a unique (cluster,site,domain) triple. Similarly, every cluster belongs to a unique pair (site,domain) and every site to a specific domain. Virtual architectures can be dynamically created and modified which will be described in the next section. Every component (node, cluster, site, and domain) of a virtual architecture is controlled by a manager which is not seen by the application programmer but used to implement JRS. More details about the implementation of virtual architectures are given in Section 5.

4 JavaSymphony Programming Model

In this section we describe the JavaSymphony programming model. Commonly, every JavaSymphony application first must

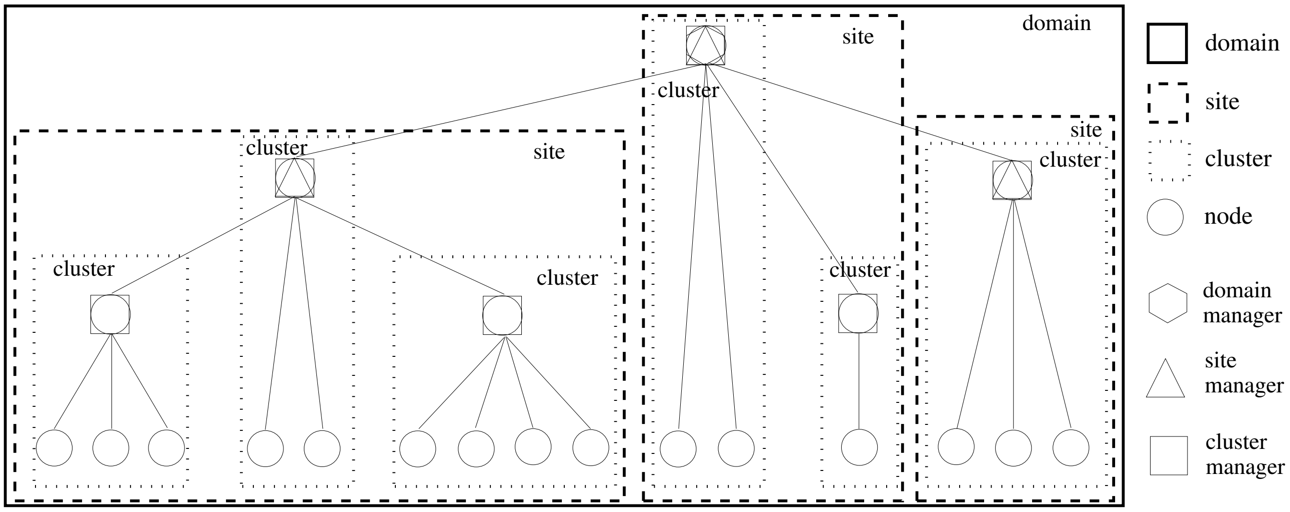

Figure 1. Example of a JavaSymphony Virtual Architecture Domain

register with the JavaSymphony runtime system (JRS). Thereafter, virtual architectures can be defined. In order to reduce the impact of Java class loading, all required classes are stored in Java archive files and loaded onto arbitrary nodes of a defined virtual architecture. Objects can be created, mapped, and migrated both on a local as well as on a remote computing node. JavaSymphony supports three kinds of method invocations which includes synchronous, asynchronous, and one-sided invocations. Finally, an application should un-register from JRS.

4.1 Register/Un-register Application

Every JavaSymphony application first needs to register with the underlying JRS which is then aware that this application is accessing its services.

```
...
// register application with JRS
JSRegistration reg = new JSRegistration();
...
// un-register application
reg.unregister();
...
```

An application should un-register from JRS as soon as none of the objects generated under JRS are still needed. Un-registration enables JRS to reduce the underlying book-keeping overhead and allows the garbage collector to deallocate memory.

4.2 Generate Dynamic Virtual Distributed Architectures

In order to specify locality, JavaSymphony provides dynamic virtual architectures. The programmer can define arbitrary topologies comprising nodes, clusters, sites and a domain based on the concept introduced in Section 3. Architecture constraints are introduced in order to specify system constraints, to control load balancing, to honor computing site policies, etc. The basic idea is to include only nodes in a virtual architecture which obey user-defined constraints defined over static and dynamic system parameters. Static parameters are not changed during execution of an application program which includes name of a machine, operating system, cpu type, peak performance parameters, etc. Dynamic parameters can change while the application program is executing which comprises system load, idle times, available memory, number of context switches or system calls, etc.

JavaSymphony allows to create an object of a class *JSConstraints* which holds a set of constraints. Constraints are added to this object by invoking calls to method *setConstraints(system_parameter,relational_operator,number_string)*. Each method invocation adds a constraint with the following pattern:

system_parameter relational_operator number_string

where relational_operator corresponds to arbitrary relational operators and number_string refers to floating point/integer numbers or strings. For instance, consider the following JavaSymphony code excerpt:

```
JSConstraints constr = new JSConstraints();
constr.setConstraints(JSConstants.NODE_NAME,"!=","milena");
constr.setConstraints(JSConstants.CPU_SYS_LOAD,"<=",10);
constr.setConstraints(JSConstants.IDLE,">=",50);
constr.setConstraints(JSConstants.AVAIL_MEM,">=",50);
constr.setConstraints(JSConstants.SWAP_SPACE_RATIO,">=",0.3);
```

A set of constraints is collected in object *constr*. The constraints specify that a computing node with the name “milena” cannot be included in a virtual architecture (yet to be requested from JRS). The system executes less than 10 % in system mode, is idle for more than 50 %, has at least 50 MBytes of unused memory, and the ratio of used to available swap space is less than 0.3. Overall the programmer can define constraints defined over approximately 40 different system parameters.

The programmer can request a virtual architecture from JRS by generating nodes, clusters, and sites that form a domain.

Nodes

Nodes can be created and released by the programmer as follows:

```
    // request arbitrary node
Node n1 = new Node();

    // request node with name "rachel"
Node n2 = new Node("rachel");

    // request node for which constraints hold
Node n3 = new Node(constr);

    // determine the associated cluster, site, and domain of n1
Cluster c1 = n1.getCluster();
Site s1 = n1.getSite();
Domain d1 = n1.getDomain();

n1.freeNode(); // release node n1 from application
```

Node *n1* is requested without specifying any constraints. In this case JRS will allocate a node with low system load and reasonable resources (e.g. memory) available. The programmer wants *n2* to be the node with name "rachel". For node *n3* all constraints collected in object *constr* - as defined previously - must hold. According to Section 3 every node is associated with a unique cluster, site, and domain which can be determined by invoking methods *getCluster*, *getSite*, and *getDomain*, respectively. A node can also be released from a given application by using method *freeNode*.

Clusters

A cluster can be requested by indicating the number of nodes to be included in the cluster. Optionally, a set of constraints can be specified which must be satisfied by every node in the cluster. In the following example a cluster *c1* with 5 nodes is requested. Note that "[...]" expresses optionality in all code skeletons of this paper. A cluster can also be defined by adding individual nodes each of which may honor a specific set of constraints. For instance, cluster *c2* is defined by adding nodes *n1*, *n2*, and *n3* – which are instances of class *Node* – to it.

```
Node n1, n2, n3;
    // allocate cluster with 5 nodes
Cluster c1 = new Cluster(5[,constr]);

    // define individual cluster which contains nodes n1, n2, and n3
Cluster c2 = new Cluster();
c2.addNode(n1); c2.addNode(n2); c2.addNode(n3);

    // determine current number of nodes in cluster
c1.nrNodes();
    // access node-3 in cluster
Node n3 = c1.getNode(3);
    // determine site of cluster
Site s1 = c1.getSite();
    // determine domain of cluster
Domain d1 = c1.getDomain();
    // release node n2 from cluster c2
c2.freeNode(n2);
    // release node-2 from cluster c2
c2.freeNode(2);
    // release cluster c2
c2.freeCluster();
```

Method *nrNodes* can be invoked for every cluster object in order to determine the current number of nodes included in the cluster. The nodes of a cluster *c* are numbered from 0 to *c.nrNodes() - 1*. Methods *getNodes*, *getSite*, and *getDomain* are used to access individual nodes, the site, and the domain of a cluster. Nodes of a cluster can also be released by invoking method *freeNode*. The entire cluster can be released through method *freeCluster*.

Sites

Sites can be generated and modified similar as done for clusters. In the following code excerpt a site s1 with 3 clusters is generated. If constraints are used then they must hold for all nodes in the site. A site can also be defined based on already existing clusters by generating an instance of class *Site* and calling methods *addCluster*.

```
int[] SiteNodes = {2,4,5};
    // request for site with 3 clusters with 2, 4
    // and 5 nodes, respectively
Site s1 = new Site(SiteNodes [,constr]);

    // define individual site which contains cluster c1 and c2
Site s2 = new Site();
s2.addCluster(c1); s2.addCluster(c2);

    // determine current number of clusters and nodes in the site
s1.nrClusters();
s1.nrNodes();
    // access cluster-1 in site
Cluster c1 = s1.getCluster(1);
    // access node-1 in cluster-2 of site s1: alternative-1
Node n1 = s1.getCluster(2).getNode(1);
    // access node-1 in cluster-2 of site s1: alternative-2
Node n1 = s1.getNode(2,1);
    // determine domain of site
Domain d1 = s1.getDomain();
    // release node-1 from cluster-2 of site s1: alternative-1
s1.freeNode(2,1);
    // release node-1 from cluster-2 of site s1: alternative-2
s1.getCluster(2).freeNode(1);
    // release cluster 1 of site s1
s1.freeCluster(1);
    // release cluster c2 of site s1
s1.freeCluster(c2);
    // release site s1
s1.freeSite();
```

The number of clusters and nodes in a site can be obtained by using methods *nrClusters* and *nrNodes*. Clusters can be accessed by invoking method *getCluster(int ClusterID)*. There are two alternatives to reference the nodes of a site. Firstly, method *getNode(int*

ClusterID, int NodeID) which accesses node with NodeID in cluster with ClusterID. NodeID (ClusterID) must be in the range between 0 and *nrNodes-1* (*nrClusters*-1). Secondly, method *getCluster* can be used to access a specific cluster of a site which can be further referenced by using the previously defined cluster method *getNode*. The domain of a site can be accessed by using method *getDomain*. Partially or fully releasing a site is supported similarly as done for clusters.

Domains

Domains are build similar to clusters and sites by incorporating multidimensional arrays. In the following code excerpt a domain with 2 sites is allocated.

```
int[][] DomainNodes = {{1,3,5},{6,4}};
      // request for domain with 2 sites
      // site-1 with 3 clusters with 1, 3, and 5 nodes, respectively
      // site-2 with 2 clusters with 6 and 4 nodes, respectively
Domain d1 = new Domain(DomainNodes [,constr]);

      // define individual domain which contains site s1 and s2
Domain d2 = new Domain();
d2.addSite(s1); d2.addSite(s2);

      // determine current number of sites, clusters and nodes
      // in the domain
d1.nrSites();
d1.nrClusters();
d1.nrNodes();
      // access site-2 in domain
Site s2 = d1.getSite(2);
      // access node-3 in cluster-2 of site-1 in domain : alternative-1
Node n1 = d1.getSite(1).getCluster(2).getNode(3);
      // access node-3 in cluster-2 of site-1 in domain : alternative-2
Node n1 = s1.getNode(1,2,3);

      // release node-3 from cluster-2 of site-1: alternative-1
d1.freeNode(1,2,3);
      // release node-3 from cluster-2 of site s1: alternative-2
d1.getSite(1).getCluster(2).freeNode(3);
      // release cluster-2 of site-1: alternative-1
d1.freeCluster(1,2);
      // release cluster-2 of site-1: alternative-2
d1.getSite(1).freeCluster(2);
      // release site-1 of domain d1
d1.freeSite(1);
      // release site s1 of domain d1
d1.freeSite(s1);
      // release domain d1
d1.freeDomain();
```

The first site has 3 clusters with 1, 3, and 5 nodes, respectively. The second site has 2 clusters with 6 and 4 nodes, respectively. Constraints can be optionally specified which must hold for all nodes in the domain. Individual domains based on already existing sites can be allocated by using method *addSite*. Domains can also change dynamically by using methods *freeNode*, *freeCluster*, *freeSite*, and *freeDomain*. The current number of nodes, clusters, and sites in the domain can be determined by invoking methods *nrNodes*, *nrClusters*, and *nrSites*, respectively.

4.3 Class Loading

JavaSymphony enables the programmer to generate objects both locally and remotely. As JavaSymphony is built on top of the Java RMI mechanism, we require all objects that can be created remotely to be serializable. Before an object can be generated, the class file of this object commonly must be located either locally in the CLASSPATH or at an arbitrary URL. JavaSymphony assumes that all Java class files are available at the nodes of a given virtual architecture before objects are generated. This reduces the amount of data transferred when objects are created. For this purpose, JavaSymphony enables to build a codebase which is then transferred (by using method *codebase.load*) as Java archive file to arbitrary components of a virtual architecture. JavaSymphony, therefore, not only supports the programmer to control data (objects) locality but also program locality. Only those components of a virtual architecture may store a class file that need it.

```
Node node; Cluster cluster; Site site; Domain domain;

   // initialize a codebase
JSCodebase codebase = new JSCodebase();

   // a Java archive or class file is added to the codebase
codebase.add("../classes.jar");
codebase.add("../testclasses.class");

   // Java archive or class file is fetched from URL
   // and added to the codebase
URL classURL =
new URL("http;//www.par.univie.ac.at/JS/test/file.class");
codebase.add(classURL);

   // load codebase to a node of a virtual architecture
codebase.load(node);
   // load codebase to all nodes of a cluster, site, or domain.
codebase.load(cluster);
codebase.load(site);
codebase.load(domain);
   // free codebase
codebase.free();
```

A method *free()* can be invoked on a codebase object which frees the codebase and associated memory.

4.4 Create, Map, and Free Objects

Assuming that class files are available on every component of a virtual architecture where needed, objects can now be created by generating instances of class JSOBj which is part of the JavaSymphony class library. The first parameter of the *new* command for object includes the class name for which an object has to be generated. Optionally, a second parameter indicates where to place the object which can be local on the node where the program is being executed, on a specific node of a domain, or on a node of a specific cluster, site, or domain. In the latter case JRS chooses a node with the smallest system load and reasonable resources available. The same accounts if no mapping parameter is indicated. A predefined class *JS* as part of the JavaSymphony class library offers various static methods which includes among others a method *getLocalNode* to determine the local node. A set of constraints (see Section

4.2) can be provided as a third parameter when generating an object to restrict the virtual architecture components on which the object can be generated and to improve load balancing of objects.

```
    // get node on which this application is being executed
Node local = JS.getLocalNode(); JSConstraints constr;
Node node; Cluster cluster; Site site; Domain domain;

    // generate an object of class "class_name" at
    // a node decided by JRS or restricted to constraints
JSObj obj1 = new JSObj("class_name" [, constr]);
    // generate object on the local node
JSObj obj1 = new JSOBj("class_name",local);

    // generate object on a specific node
JSObj obj1 = new JSOBj("class_name",node);

    // generate object on an arbitrary node of a cluster, site,
    // or domain decided by JRS or restricted to contraints
JSObj obj1 = new JSOBj("class_name" [,cluster|site|domain ,constr]);

    // generate obj1 on the same node
    // where obj2 has been generated
JSObj obj1 = new JSOBj("class_name" ,obj2.getNode());
    // generate obj1 on the same cluster, site,
    // or domain where obj2 has been generated
JSObj obj1 = new JSOBj("class_name",obj2.getCluster() [,constr]);
JSObj obj1 = new JSOBj("class_name",obj2.getSite() [,constr]);
JSObj obj1 = new JSOBj("class_name",obj2.getDomain() [,constr]);
    // free object
obj1.free();
```

Moreover, the programmer has a choice to map an object on the same node, cluster, site or domain where some other node already resides. If obj1 should be generated on a cluster, site, or domain on which obj2 resides, then JRS or a user-provided set of constraints decides on which node within this cluster, site, or domain, obj1 actually will be generated.

Finally, an object if no longer needed should be released by the programmer through invoking method *free* which reduces the overall book-keeping effort and enables the garbage collector to deallocate the memory for this object.

4.5 Method Invocation

Java/RMI imposes blocking remote method invocation which prohibits overlapping of waiting time – for results of remote method invocations to arrive – with some useful local computations. In addition to synchronous (blocking) RMI, JavaSymphony also offers asynchronous (non-blocking) and one-sided RMI (non-blocking without results).

Synchronous Method Invocation

Synchronous method invocation (by using predefined method *sinvoke* of an object) blocks the calling site for as long as the result arrives. Parameters are passed as an array of objects. JavaSymphony always returns a method invocation result of class Object which must be explicitly casted to the actual class of the result. In the following code excerpt a method with name "method_name" with parameters *Param1()* and *Param2()* is invoked based on object *obj*.

```
JSObj obj = new JSOBj("class_name");
...
Object[] params = {new Param1(), new Param2()};
ResultClass result = (ResultClass)obj.sinvoke("method_name",params);
```

Asynchronous Method Invocation

Asynchronous method invocations (by using predefined method *ainvoke* of an object) are commonly employed to parallelize computations. Again an array of objects is used to hold the method parameters. The method call, however, does not block but immediately returns a handle. Execution continues at the calling site. If a pre-defined method *handle.isReady* returns TRUE then the result is available, FALSE otherwise. If the calling site wants to block until the result has arrived – for instance, because no other useful computations can be done – then method *handle.getResult* can be called. Note that this method returns the result object of type Object. It must be explicitly casted to the actual class of the result.

```
    // invoke remote method with parameters; a handle is returned
    // to refer to the method's result in the future
Object[] params = {new Param1(), new Param2()};
ResultHandle handle = obj.ainvoke("method_name",params);
...
    // verify whether result is available
if (handle.isReady()) {
    // wait for result to arrive in blocking mode
    ResultClass result = (ResultClass)handle.getResult();
}
...
    // wait for result to arrive in blocking mode
    // without checking for available result
ResultClass result = (ResultClass)handle.getResult();
```

One-sided Method Invocation

A one-sided method invocation (by using predefined method *oinvoke* of an object) is used in case that it is not necessary to wait for the completion of a remote method invocation and no result is returned. This method invocation can improve the performance of the application because there is no need to transfer back a result from a node that hosts the remote object. Moreover, one-sided method invocation reduces some book-keeping overhead of JRS.

```
Object[] params = {new Param1(), new Param2()};
obj.oinvoke("method",params);
```

4.6 Dynamically Migrate Objects

Objects can be migrated during execution of an application. JRS, however, verifies before object migration, whether any of its methods are currently being executed. If so, then migration is delayed until all unfinished method invocations have completed execution, otherwise the object can be immediately migrated. JavaSymphony offers two forms of object migration: automatic migration which is controlled by JRS or explicit migration which is controlled by the programmer.

Automatic object migration can be enabled through the JS-Shell (see Section 5). In the automatic mode JRS periodically examines whether the creation constraints of a virtual architecture provided by the programmer still hold. If no programmer constraints have been provided, then a set of system constraints defined by the JS-Shell must hold. In both cases if the constraints of certain virtual architecture components do not hold then the objects on these components will be migrated to another component for which the constraints hold.

Explicit migration can be encoded by the JavaSymphony application programmer. For this purpose JavaSymphony allows to access system parameters for virtual architectures. System parameters for clusters, sites, and domains are averaged across the contained nodes. Method *getSysParam* can be called for every architecture component to examine the system parameter of interest. Moreover, through method *constrHold* it can be verified whether a set of constraints currently hold for a given architecture component. These are the same parameters that are also used to restrict requests for virtual architectures according to Section 4.2. For instance, in the following code excerpt it is examined whether the node *n1* on which an specific object resides has less than 50 % idle time. If so, then this object can be migrated by using method *migrate*. If *migrate* is called without any parameters then JRS decides where to migrate the node.

```
JSConstraints constr;
Node node; Cluster cluster; Site site; Domain domain;
JSObj obj;

Node n1 = object.getNode();
        // node on which object resides has less than 50 % idle time
        // set of constraints constr hold for node n1
if (n1.getSysParam(JSConstants.IDLE) < 50) ||
      n1.constrHold(constr)) {
        // migrate object to a node destined by JRS
      obj.migrate();
        // migrate object to a node according to a set of constraints
      obj.migrate(constr);
        // migrate object to a specific node
      obj.migrate(node);
        // migrate object to a node of a cluster, site, or domain
        // to be destined by JRS or optionally based on constraints
      obj.migrate(cluster|site|domain [,constr]);
}
```

Method *migrate* can be invoked with a node as parameter that defines where to migrate the object. If a cluster, site or domain is indicated without constraints then it is up to JRS to which node within the given domain component the object should be migrated. If constraints are indicated then a node found by JRS that honors the constraints is chosen as the target node for object migration.

4.7 Persistent Objects

JavaSymphony provides facilities to make objects persistent by saving and loading them to/from external storage. An object can only be stored/loaded when none of its methods are currently executing which is verified by JRS. A unique string will be returned by JRS for the object just stored.

```
JSObj obj; String str;

      // save object on external storage
str = obj.store();
...
      // load object from external storage
JSObj obj = new JSObj(str);
```

Objects are loaded by invoking the constructor of *JSObj* with the string *str* that uniquely identifies a previously stored object under JRS. JavaSymphony persistent objects can be easily implemented through JDK's object serialization mechanism.

5 JavaSymphony Runtime System (JRS)

The JavaSymphony Runtime System (JRS) is implemented as an agent based system (see Figure 2) that consists of a network agent system (NAS), an object agent system (OAS), and the JavaSymphony Administration Shell (JS-Shell). The nodes on which JRS is installed are configured by using the JS-Shell. The set of nodes can be changed by adding or removing nodes dynamically during execution of JavaSymphony applications (JSAs) by using JS-Shell. On every node a single network agent (NA) is placed which monitors the system behavior. The JS-Shell controls the network agents by using the Java/RMI mechanism.

Every NA is associated with an object agent (OA) that provides an interface to JSAs. The object agent is subdivided into two parts. Firstly, a public object agent (PubOA) which is integrated with the NA in a single JVM, and secondly, an application object agent (AppOA) which is created for every JSA. AppOAs and JSAs on the one hand, and PubOAs and NAs on the other hand interact by local (direct) method invocation. Whereas between AppOA and PubOA on the same node and on different nodes, the Java RMI mechanism is used.

5.1 The Network Agent System

The network agent system [7] is responsible to monitor the system behavior. It provides a limited fault tolerance mechanism in case that some node does not respond anymore, and determines system performance parameters that can be accessed by the JS-Shell, the OAS, and the JSA programmer (see Section 4.6).

Every component (node, cluster, site, and domain) of a virtual architecture is controlled by a manager which is transparent to the application programmer. Both NAS and JS-Shell can determine whether a node becomes a cluster, site, or domain manager (see Figure 1). Note that virtual architectures are stored within the JVM that holds both PubOA and NA. Therefore, both NA and PubOA have access to information about virtual architectures. The nodes in a cluster are controlled by a cluster manager which by itself is a node of the cluster. Similar accounts for sites. Only a cluster manager can be a site manager and only a site manager can be a domain manager. This also means that a cluster, site or domain manager are at the same time part of a cluster. In addition to management tasks every manager node can also be used as a computing node for JSAs. Moreover, managers determine system performance parameters of their architecture component and forward them to other managers or nodes in the system. The system parameters provided by JRS are determined by invoking the *exec*

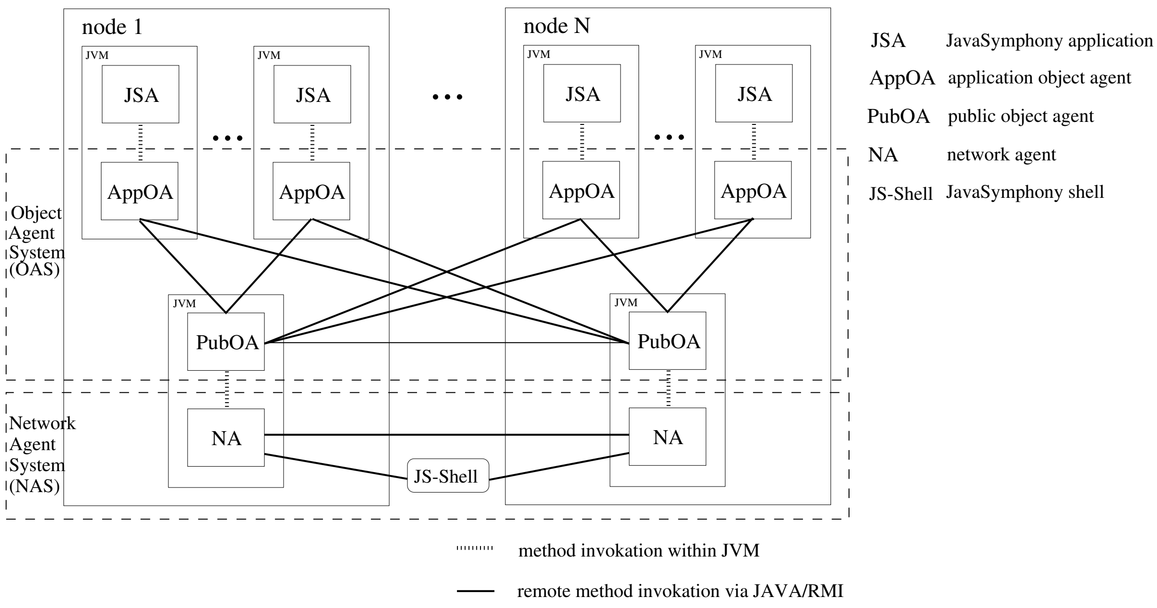

Figure 2. JavaSymphony Runtime System Architecture (JRS) based on object agent and network agent system.

method of the *java.lang.Runtime* class [23] with various system performance commands.

The resources are periodically monitored at every node to collect up-to-date values of processor and network parameters which includes static and dynamic system parameters. Static parameters remain unchanged during execution of an application program which comprises node name, IP address, architecture type, total memory size, operating system, peak performance parameters, etc. Dynamic parameters may change while a program is executing which includes CPU load, idle times, available memory size, number of processes and threads, number of context switches or system calls, network latency, network bandwidth, etc. Overall JRS supports close to 40 different system parameters. Moreover, monitoring is employed to examine the resources for system failures.

The nodes forward the observed system parameters to their associated cluster manager which averages these values across all cluster nodes and stores them locally. The cluster manager forwards these data to the site manager which collects all data from its clusters and finally sends averages values to the domain manager. Every manager locally stores system data for all its architecture components (e.g. a site manager is at the same time a cluster manager) and forwards them to the next higher level in the virtual architecture hierarchy. The performance measurement and collection periods can be controlled under the JS-Shell. Storage size for these data is kept reasonably small as only the least recently measured data are kept. Currently we do not maintain a history of measurements, although, it would be easy to support it.

Managers periodically examine their virtual architecture components as well as the manager of the next lower and higher hierarchy (for instance, a site manager examines its associated cluster managers and domain manager) for system failures. If a certain node does not respond within a predefined time period (changeable under JS-Shell) then this node is said to have caused a failure and will be released by JRS according to the following simplified fault tolerance mechanism:

- If a non-manager or backup-manager (see below) node of a cluster failed, then the manager of this cluster simply releases this node.
- If a manager node failed then a backup manager within the same hierarchy releases the manager and takes over as the new manager of this virtual architecture component. The backup manager then informs the JS-Shell, all of its associated lower-level and higher-level managers, and nodes in its component about the failure. For instance, a backup site manager would inform all of its cluster managers, the nodes of its cluster, and the domain manager about the failure. Finally, as the backup manager becomes the new manager, a second backup manager (pre-defined by the JS-Shell) is activated.

Upon failure of a node, the JS-shell and the associated PubOA of this node are informed either by the manager (if a non-manager node fails) or backup-manager (if a manager node fails) about this failure through the Java/RMI mechanism. Note that currently the object agent system does not exploit information about system failures provided by the NAS. Future work will address the issue of allowing the object agent system to at least partially recover from certain system failures.

Network agents are implemented as a set of threads within a single JVM. For more details the reader may refer to [7].

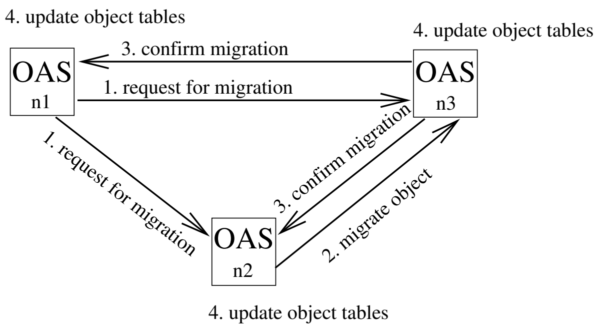

Figure 3. Object migration under the JavaSymphony Object Agent System.

5.2 The Object Agent System

The object agent system (OAS) directly interacts with JavaSymphony applications. OAS supports the administration of objects which includes creation, mapping, migration, load balancing, and deletion of objects. Furthermore, OAS is responsible to manage RMIs, transfer method parameters, to execute these methods at the object's location, and to return the corresponding result to the call site.

Every JSA has a specific AppOA associated that stores the following information for every generated object of this JSA in the *local-objects-table*:

- unique object handle
- location of PubOA where object instance is generated and stored
- result objects for invoked methods of this object
- flag that specifies whether any method of the object is currently being executed

For every generated object, AppOA returns a handle to JSA. The actual object instance is generated by the local AppOA or a remote PubOA which can be user-defined (see Section 4.4) through mapping objects onto specific components of a virtual architecture. If no mapping is specified then the AppOA invokes the PubOA to determine a node that must honor a set of constraints (e.g. node with smallest system load) defined under JS-Shell. Note if an object instance of a JSA is generated locally, then it is always stored in the *local-objects-table* of the associated AppOA. An object which is generated on a remote node n is stored – with similar information as mentioned above – in the *remote-objects-table* of the PubOA on n. Every PubOA that possesses a handle to an object has full access to the object's data and methods. Object handles are associated with information about the location of the object and the AppOA from which the object originates. Object handles (first-order objects) can be passed to methods of other objects that may reside on arbitrary nodes. Methods are always executed at the local AppOA or at remote PubOAs where the object has been generated and the corresponding results are sent back to the local AppOA. An AppOA that invokes a method on a remote object always directly interacts with the AppOA or PubOA that holds the object.

Object migration commonly involves an AppOA *ao* associated with a JSA on a node $n1$, a PubOA *pa1* that holds the object instance on node $n2$, and a PubOA *pa2* on node $n3$ to which the object should be migrated. The following protocol (see Figure 3) is executed for an object migration from *pa1* to *pa2*:

1. *ao* requests *pa1* and *pa2* to migrate object from *pa1* to *pa2*.
2. *pa1* transfers object to *pa2*.
3. *pa2* confirms migration to *ao* and *pa1*.
4. *ao*, *pa1*, and *pa2* update their object tables.

The object migration protocol is very similar if the object is stored or migrated to the local AppOA. This protocol ensures that the AppOA from which the object originates (site of associated JSA) is always aware about the location of the object. If a remote method invocation fails due to object migration, the new object location is accessed from the AppOA from which the object originates. This is done automatically by the OAS (see Figure 4).

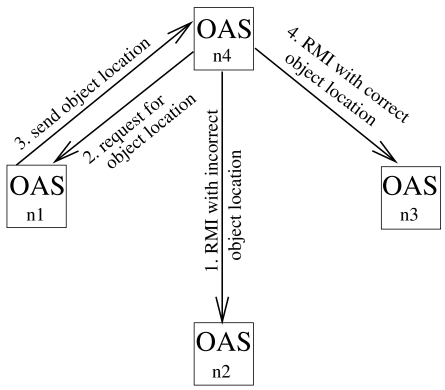

Figure 4. RMI on migrated object under the JavaSymphony Object Agent System.

An AppOA requests a virtual architecture from the local NA via the PubOA of the same node. The PubOA stores all virtual architectures generated by any JSA on the local node which includes the following information: virtual architecture identification and specification, identification of associated JSA and AppOA, constraints that must hold by this architecture as indicated during creation, etc. The PubOA periodically examines whether the constraints of the stored virtual architectures are still fulfilled through accessing system parameters via its local NA. A list of all architecture components that no longer fulfill these constraints is sent to the corresponding AppOA. The AppOA is then trying to migrate (according to the object migration protocol mentioned above) all objects originating from its JSA that are on this list to other architecture components which fulfill the original constraints. To maintain all mapping constraints (for instance, if several objects have been mapped on the same node for locality reasons) it is tried to migrate all objects of an overloaded node to some other node for which a given set of architecture constraints hold. To maintain locality JRS tries to migrate objects of one node to another node within the same cluster of the original node. If the constraints do not hold for any node in the cluster of the original node then another cluster within the same site of the original node is tried next, and so forth. Automatic migration can cause performance degradation if a large number of objects is transferred. Therefore, it is possible to enable/disable automatic migration under the JS-Shell.

Both AppOA and PubOA are implemented as a collection of threads under JDK 1.2.1. AppOA consists of several threads which includes: one thread for interaction with the associated JSA, one thread for every asynchronous method invocation in order to overcome blocking Java/RMI, and one thread for interaction with the local and all remote PubOAs. On the PubOA there is one thread running for every local AppOA, one thread for all remote AppOAs, one thread for all remote PubOAs, and one thread for the local NA.

6 Experiments

A preliminary version of JavaSymphony has been implemented which includes all functionality described in this paper except for object migration and persistent objects. In this section we describe an experiment that has been conducted on a non-dedicated heterogeneous cluster of 13 Sun workstations comprising Sparcstations 4/110, Sparcstations 10/40, Sparcstation 5/70, Sun Ultras 1/170, Sun Ultras 10/300, and Sun Ultras 10/440. All Sun Ultra workstations are connected based on 100 Mbits/sec bandwidth, whereas communication among all other workstations rely on 10 Mbits/sec bandwidth. All workstations run Sun Solaris 7. These workstations are used by individual people for their regular work. We used Sun's JDK 1.2.1 with a JIT compiler and native threads as the platform JVM.

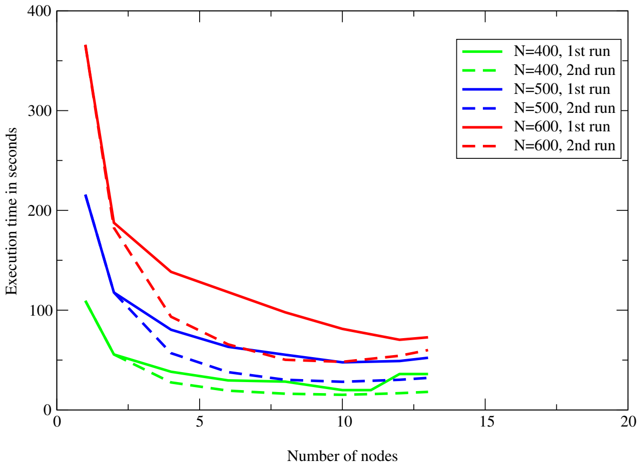

Figure 5. JavaSymphony matrix multiplication performance for different problem sizes and system loads.

For our experiment we implemented a 2-dimensional matrix multiplication ($A * B = C$) using a master-slave parallelization strategy as shown in Figure 6. For the sake of demonstration Figure 6 shows only the most important excerpt of our application. We omitted in particular exception handling and any kind of error checking. Class *Aux* is an auxiliary class with several methods for initializing arrays and setting up slave tasks. Class *Matrix* provides methods for initializing arrays A, B, and C, and merging results in matrix C. At the beginning, the application registers with the JRS. A cluster with *nr_nodes* nodes is requested from JRS. A codebase with all important byte-code is transferred to every node of the cluster. Matrix B is replicated on the entire cluster by using a one-sided invocation of method *init*. Sets of rows (defined by variable *rows_per_task*) of matrix A define a task to be executed by an individual node. The number of rows does not change during execution of the application. In each iteration of the WHILE-loop every node of the cluster is examined whether it is waiting for a task or whether it is still executing a task. If a node is waiting for a task then a new task is assigned to it via asynchronous invocation of method *multiply*. If a node is still executing a task, then it is verified whether a result is already available. If so, then the result is merged with matrix C and the node is marked as being ready for a new task. The WHILE-loop is exited if all tasks have been processed and the results are returned. After the parallel matrix multiplication finished, the application un-registers from JRS.

Figure 5 shows the time required to complete a matrix multiplication of two N*N matrices on the indicated heterogeneous workstation cluster for varying workstation (node) numbers. Note that the times plotted for the one-node-experiments are based on a sequential matrix multiplication that does not use JavaSymphony at all. We ran each experiment (defined by a specific N and a set of nodes) twice for two different system loads based on identical set of nodes. The first set of experiments (see solid line execution time functions in Figure 5) was conducted during the day when the workstations have been used by individual people for their everyday work (e.g. program development, e-mailing, etc.). The second set of experiments (see dashed line execution time functions in Figure 5) has been done at night with very little system load implied by individual users. If we compare both experiments, then it can be clearly seen, that the overall performance exploited from the workstation cluster is considerably better at night where almost linear speed-up is achieved for up to 6 nodes. Beyond 6 nodes the scaling behavior deteriorates. The execution times for the experiments conducted during the day scales up to 2 nodes. Adding up to 10 nodes reduces the execution time. For all experiments, using more than 10 nodes increases the execution time of the matrix multiplication which is mostly due to a larger number of RMIs.

Note for the reviewer: We plan to add more experiments if the paper is accepted.

7 Conclusions and Future Work

JavaSymphony provides a programming paradigm and software infrastructure to alleviate distributed and parallel programming that effectively exploits heterogeneous resources ranging from small-scale cluster computing to large-scale wide-area meta-computing. In contrast to most existing work, JavaSymphony allows the programmer to explicitly control locality of data and load

```
public class MatrixMultiply {
    public static void main (String args[]) {
    ...
        // register JavaSymphony application
    JSRegistration reg = new JSRegistration();

        // allocate cluster
    Cluster c1 = new Cluster(nr_nodes);

        // define codebase and load on cluster c1
    JSCodebase cb = new JSCodebase();
    cb.add("../matrix-test/classes.jar");
    codebase.load(c1);

        // allocate and initialize matrices A, B, and C for matrix multiplication: A*B = C
    float [] A = new float[dimA1*dimA2];
    float [] B = new float[dimA2*dimB2];
    float [] C = new float[dimA1*dimB2];
    Matrix.initMatrix(A,B,C);

        // copy matrix B to all cluster nodes
    Object[] paramB = {dimA2,dimB2,B};
    for (i=0;i < c1.nrNodes(); i++) {
        JSObj DistrMpy[i] = new JSObj("Matrix",c1.getNode(i));
        DistrMpy[i].oinvoke("init",paramB);
    }

        // determine nr of tasks to be processed by cluster nodes
    nr_tasks = dimA1/rows_per_task;
    if (dimA1 % rows_per_task != 0) nr_tasks++;
    next_task = 0;
    Aux.initArray(nodeBusy,-1); // init nodeBusy with "-1" for every cluster node

        // distribute tasks (set of rows of matrix A) to nodes of cluster
    while (next_task < nr_tasks) {
        for (i=0;i < c1.nrNodes(); i++) {
            if (nodeBusy[i] >= 0) { // node is executing task
                if (hdl[i].isReady()) { // result is available
                    Matrix.mergeResult((ResultData)hdl[i].getResult()); // merge result in matrix C
                    nodeBusy[i] = -1; // set node to be free again
                }
            }

                if (nodeBusy[i] < 0) { // node is free to work on next task
                Object[] paramA = {Aux.setupTask(next_task,rows_per_task,dimA1,matrixA)};
                ResultHandle hdl[i] = DistrMpy[i].ainvoke("multiply",paramA);
                nodeBusy[i] = next_task; // set node to be busy again
                next_task++;
            }
        }
    }
        ... // do something with the result

    reg.unregister(); // unregister JavaSymphony application
    ...
    }
}
```

Figure 6. Code skeleton of master/slave JavaSymphony matrix multiplication (A*B=C)

balancing. Moreover, JavaSymphony supports sophisticated remote method invocation mechanisms, persistent objects, a high-level API to access hardware/software system parameters, and selective remote classloading. Preliminary experiments have shown that our software infrastructure achieves reasonable performance on a heterogeneous cluster of workstations.

We are currently in the process to evaluate JavaSymphony with larger applications. Moreover, we are extending JavaSymphony to handle static methods and variables, and to enable recovery of the OAS from certain system failures. Finally, we continue to improve our techniques for automatic mapping and migration of objects.

ACKNOWLEDGMENTS

The author is very grateful to Erwin Laure from the Institute for Software Science, University of Vienna, for many fruitful discussions on the design and implementation of JavaSymphony and for proof-reading this paper. Furthermore, the author thanks L. Hofer and A. Kulin for implementing the network agent system and G. Aschenbrenner and T. Pfeiffer for implementing the object agent system.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[2] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.

[3] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: metacomputing on the Web. In K. Yetongnon and S. Hariri, editors, *Proceedings of the ISCA International Conference. Parallel and Distributed Computing Systems, Dijon, France, 25–27 September, 1996*, volume 1, pages 2–??, Raleigh, NC, USA, 1996. International Society of Computers and Their Applications (ISCA).

[4] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[5] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, Nov. 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[6] P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag: Heidelberg, Germany, Apr. 1997.

[7] T. Fahringer, L. Hofer, and A. Kulin. The JavaSymphony Network Agent System (in german). Technical Report TR2000-09, Institute for Software Science, University of Vienna, May 2000.

[8] M. P. I. Forum. *Document for a Standard Message Passing Interface*, draft edition, Nov. 1993.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[10] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

[11] G. Glass. ObjectSpace voyager — the agent ORB for Java. *Lecture Notes in Computer Science*, 1368:38–??, 1998.

[12] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 08 1994. Mon, 28 Aug 1995 21:06:39 GMT.

[13] High Performance Fortran Language Specification Version 2.0, January 1997.

[14] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an environment for parallel, distributed and mobile java applications. In *Proceedings of ACM 1999 Java Grande Conferernce*, pages 15–25, San Francisco, CA, June 1999.

[15] D. B. Lange and M. Oshima. *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley, Reading, MA, USA, Sept. 1998.

[16] E. Laure. Distributed High Performance Computing with OpusJava. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo'99, 17-20 August 1999, Delft, The Netherlands*, pages 590–597. Imperial College Press, Apr. 2000.

[17] E. Laure. OpusJava: A Java Framework for Distributed High Performance Computing. *Future Generation Computer Systems*, in print 2001.

[18] E. Laure, P. Mehrotra, and H. Zima. Opus: Heterogeneous Computing With Data Parallel Tasks. *Parallel Processing Letters*, 9(2):275–289, June 1999.

[19] D. S. Linthicum. CORBA 2.0? *Open Computing*, 12(2):68–??, Feb. 1995.

[20] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI. In *Proceedings of the ACM Java Grande Conference, San Francisco, CA.*, pages 152–159, New York, NY, June 1999. ACM Press.

[21] M. Philippsen and B. Haumacher. More Efficient Object Serialization. *Lecture Notes in Computer Science*, 1586:718–??, 1999.

[22] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Nov. 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[23] Sun Microsystems. Java class packages.

[24] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339 (or 315–340??), Dec. 1990.

[25] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: a migratable parallel objects framework using Java. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages ??–??, New York, NY 10036, USA, 1998. ACM Press.

[26] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in Java. In *Proceedings of the ACM Java Grande Conference*, New York, NY, June 1999. ACM Press.

[27] A. Wollrath, J. Waldo, and R. Riggs. Java-centric distributed computing: Providing a homogeneous view of a heterogeneous group of machines. *IEEE Micro*, 17(3):44–??, May/June 1997.