JavaSymphony: Extensions for Explicit Control of Locality, Parallelism, and Load Balancing for Cluster and Grid-Computing *

AURORA TR 2002-07

Thomas Fahringer and Alexandru Jugravu
Institute for Software Science, University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
{tf,aj}@par.univie.ac.at

Abstract

There has been an increasing research interest in using Java for performance-oriented distributed applications. Many approaches tend towards automatic management of locality, parallelism and load balancing which is mostly under the exclusive control of a runtime system. However, the programmer is commonly disabled to provide crucial information about the application structure (e.g. locality relationships or affinities between data and tasks) to the compiler or runtime system which frequently results in critical performance problems. In previous work we described JavaSymphony to substantially mitigate this problem. JavaSymphony is a Java class library that allows to control parallelism, load balancing, and locality at a high level. Objects can be explicitly distributed based on virtual architectures which impose a virtual hierarchy on a distributed system of physical computing nodes. The concept of remote method invocation is used to exchange data among distributed objects and to process work by remote objects.

In this paper we describe various extensions of JavaSymphony which includes a generalization of virtual architectures. Objects can now be dynamically converted from conventional Java objects to JavaSymphony objects. The programmer can control whether a single or multiple threads execute all methods of an object at any given time. Moreover, a lock/unlock mechanism has been introduced in order to avoid inconsistent modification of data. All of these features have been implemented. Experiments conducted on a heterogeneous workstation cluster and on a SMP cluster architecture are described to demonstrate performance values for several JavaSymphony applications.

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

1. Introduction

The usage of Java for performance-oriented parallel and distributed applications [14, 12], in particular for scalable web servers, multimedia applications, and large-scale scientific applications, has gained considerable attention in recent years.

Much work has been conducted to provide flexible and high-level APIs to support programming of parallel and distributed applications based on Java. Many of these approaches, however, assume that the runtime system is able to detect parallelism, to exploit locality and to achieve efficient load balancing. Automatic load balancing and data migration can easily lead to significant performance degradation as the underlying runtime system lacks sufficient information about the distributed application. In many cases programmers are very much aware of the particular nature of their application, how to distribute data, when to migrate data, etc. On the other hand, the programmer may require information from the runtime system so as to support or to refine his strategic decisions. Therefore, a high-level API to system parameters is needed to support and enhance the interplay between programmer and runtime system, which in turn greatly mitigates the programming effort, and to improve the performance of programs targeting modern parallel and distributed computing systems.

We have introduced JavaSymphony [5] which offers a novel programming paradigm for wide classes of heterogeneous systems ranging from small-scale cluster computing [4] to large scale GRID [10] computing. JavaSymphony supports distributed object computing and is particularly well-suited for applications that require shared address space, task parallelism, and one-side message passing. JavaSymphony has been implemented by a 100% Java class library which strongly supports the programmer to specify and to control locality, parallelism, and load balancing at a high level without putting a burden on the programmer to deal with error-prone and time consuming low-level details (e.g. creating and handling of remote proxies for Java/RMI or socket communication). JavaSymphony tries to alleviate the development of parallel and distributed Java programs and to improve the associated performance. We do not claim that Java or JavaSymphony programs can achieve similar or even better performance than equivalent C or Fortran versions.

In this paper we describe various improvements of the original JavaSymphony (JS) paradigm to alleviate performance-oriented distributed and parallel programming. In the following we give a brief overview of the JS programming constructs and indicate all improvements compared to the previous JS [5] version.

- **Dynamic Virtual Distributed Architectures:** The programmer can dynamically define and modify virtual distributed architectures (VAs) that impose a virtual hierarchy on a distributed system of physical computing nodes. Previously VAs have been defined as a tree-structure with a maximum tree depth of 5 and each depth has been associated with a specific VA type (e.g. sites or clusters). We have generalized our original ideas such that every VA defines a computing infrastructure subdivided into different components each of which is associated with a unique level. Higher level VAs correspond to collections of lower level VAs that are coupled through an interconnection network.

- **Lock/Unlock VAs**: In the previous JS version, VAs could be dynamically modified, but the programmer had to control consistency with the Java-provided synchronization mechanisms. In the new JS version, a lock/unlock mechanism is provided to alleviate changing of existing VAs.

- **Access to system parameters**: JS 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.

- **Automatic and User-Controlled Mapping of Objects**: The programmer can control the creation and mapping of objects to specific components of VAs by using JS objects. Moreover, the mapping of JS objects can be done in relation to the location of other JS objects. If the programmer does not provide explicit mapping of objects, then JRS offers automatic mapping based on periodically monitored system constraints.

- **Single-threaded versus multi-threaded Objects**:

  The new JS version enables the creation of single- or multi-threaded objects. A single threaded object has one thread associated with it that executes all of its method calls. Several threads can be used to execute the methods of a multi-threaded object simultaneously. The attribute single- or multi-threaded of an object can be changed dynamically. Multi-threaded objects can be effectively used for multi-processor (for instance, symmetric multiprocessor) nodes that use a shared memory to exchange data. The threads of a multi-threaded object can be distributed on the processors of a multi-processor node thus improving the overall performance.

- **Object conversion**: In the old version, if a JS object had to be generated for a conventional Java object, then both objects had to be created at the same time with a specific JS method. In the new version, conventional Java objects can be converted dynamically to JS objects which allows to access conventional objects remotely via JS method invocations. Conventional Java objects can now exist before their corresponding JS objects are generated.

- **Lock/Unlock Objects**: A new feature has been added to control data access and method invocations of a JS object by using a lock/unlock mechanism.

- **Automatic and User-controlled Object Migration**: JS supports both automatic and user-controlled migration of objects through periodically monitoring system parameters. This feature has been part of the original JS but implemented only under the new version.

- **Asynchronous Remote and One-sided Method Invocation**: The concept of remote method invocation is used to exchange data among distributed objects and to process work (tasks) by remote objects. Whereas all RMIs under Java are performed synchronously in blocking-mode, JS 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.

In addition JS supports persistent objects that enable the programmer to explicitly store and load objects to/from external storage. Class-loading to specific architecture components is supported to reduce the overall memory requirement of an application.

Moreover, JS does not require to extend Java, modify the JVM, compiler or stub compiler. We have implemented a JS class library for cluster architectures. Currently, we are in the process to evaluate our system through the development of several JS applications.

The rest of this paper is organized as follows: The next section discusses related work. In Section 3 we describe a generalization for dynamic virtual distributed architectures. Section 4 introduces the programming model of JS. Experiments are presented and discussed in Section 5. Finally, some concluding remarks are made and future work is outlined in Section 6.

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 [17] and higher-level dedicated systems, including Globus [6] and Legion [9]. 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.

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 the Java Virtual Machine JVM (e.g. Java/RMI or object serialization) [20, 12]. JS directly benefits by these optimizations as the JRS runs on any standard compliant Java virtual machine. Other projects such as cJVM ([3] modify the JVM to adapt it to a distributed shared memory model without changing the semantics or the Java syntax.

The second category extends Java with special distribution constructs and semantics or provides class libraries to alleviate the usage of Java as a distributed/parallel programming language. JS also falls into this category. JavaParty [15] 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. A simple method is included to control object distribution and migration.

Javelin [1], Ninflet [19], SuperWeb [2], and JaWS [13], employ a three-tier architecture with the entities: brokers, clients, hosts. Clients seeking computing resources by submitting their work by 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. Bayanihan [16] follows a similar approach by supplying a framework for volunteer computing based on Java applets. These approaches are appropriate for master/slave and divide-and-conquer applications, but lack flexible communication mechanisms among hosts and also provide very little help to control locality.

Javelin [1] and JavaSpaces [7] can be considered as Linda derivatives which provide either none or only very limited support (compared to the functionality offered by JS) to control locality.

Ajents [11] has influenced JS's programming model for remote object creation, asynchronous remote method invocation and class loading. However, Ajents does not support VAs, control of object locality, one-sided remote method invocations, multi-threaded objects, locking/unlocking of objects, and access to system parameters as introduced in JS.

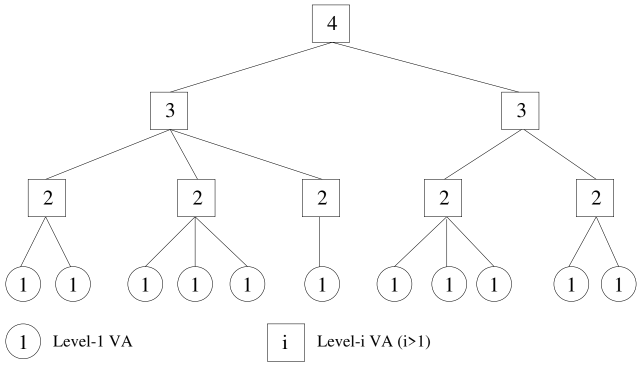

Figure 1. Example of a JavaSymphony level-4 virtual architecture which can be created with the following command: VA(4,new int[][]{{2,3,1}{3,2}})

3 Dynamic Virtual Distributed Architectures (VAs)

JS introduces the concept of *dynamic virtual distributed architectures* (called VAs in the remainder of this paper) which allows the programmer to define a structure of a heterogeneous (e.g. type, speed, or configuration) network of computing resources and to control mapping, load balancing, and migration of objects and code placement. Dynamic virtual distributed architectures (see Fig. 1) consist of a set of components each of which is associated with a level:

- level-1 VA corresponds to a single computing node such as a PC, workstation or a multiprocessor system (e.g. a symmetric multiprocessor SMP). We also refer to it as a (computing) node in the remainder of this paper.
- level-2 VA refers to a cluster of level-1 VAs (e.g. workstation or PC cluster).
- level-3 VA defines a cluster of geographically distributed level-2 VAs connected, for instance, by a wide area network.
- level-i VA with $i \geq 2$ denotes a cluster of level-(i-1) VAs which among others allows to define arbitrary complex heterogeneous GRID architecture distributed across several continents.

4 JavaSymphony Programming Model

Many programmers are well aware of how to structure a distributed application, where to place objects, which objects interact with each other, and how to exploit and to control locality and parallelism. JS on the one hand supports automatic mapping, load balancing, and migration of objects without involving the programmer. However, fully automatic systems commonly cause poor performance results due to lack of information about the application and insufficient static and dynamic analysis. JS, therefore, provides a semi-automatic mode which leaves the error-prone, tedious, and time consuming 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 at a very high level which includes:

- the setup of the virtual distributed architecture by determining which computing resources can be used and how these resources should be organized for executing a distributed/parallel program.
- 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 node of a VA if they heavily interact with each other.
- the mapping of data (objects) onto specific nodes based on system constraints (e.g. available memory larger than a specified minimal value or CPU load up to a maximum value).
- dynamic conversion of conventional Java objects to JS objects which enables access to conventional Java objects through JS method invocations. Moreover, JS objects can be dynamically modified to become single- or multi-threaded JS objects.
- placement of code (Java byte-code) on specific computing nodes which reduces the overall memory requirement of an application.

Commonly, every JS application first must register with the JS runtime system (JRS). Thereafter, VAs can be requested. In order to minimize the deleterious performance effects of Java class loading, all required classes are stored in Java archive files and loaded onto arbitrary nodes of a defined VA. Objects can be created, mapped, and migrated both on a local as well as on a remote computing node. JS supports three kinds of method invocations which includes synchronous, asynchronous, and one-sided invocations. Finally, an application should un-register from JRS.

We have implemented a full version of the JS class library which includes the JRS and the JS Shell (JS-Shell). Various system parameters (e.g. default attributes and mapping of objects, number of threads incorporated for multi-threaded objects, automatic object migration, etc.) can be changed dynamically by using the JS-Shell.

In the following sections we describe mostly those JS programming constructs which have been improved or newly added compared to the previous JS version. For details about how to register/unregister an application under JS, class loading, synchronous/one-sided method invocation, and persistent objects the reader may refer to [5].

4.1 Generate Dynamic Virtual Distributed Architectures

In order to control locality and parallelism, JS provides dynamic virtual distributed architectures which can be dynamically requested with a simple mechanism. The implementation details are handled by the JS runtime system. The programmer can define arbitrary topologies comprising of single computing nodes (e.g. PCs, workstations or multiprocessor systems), cluster architectures, cluster of clusters, and highly complex and heterogeneous grid architectures distributed over several continents. System constraints can be used to control load balancing, to honor computing site policies, etc. The basic idea is to include architecture components in a VA which obey user-defined constraints defined over static and dynamic system parameters.

Constraints

Static parameters remain unchanged during execution of an application which includes machine name, 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.

JS allows to create an object of a class *JSConstraints* which holds a set of constraints. JS constants are used to denote different system parameter types. Constraints are added to a *JSConstraints* object by invoking calls to the following methods:

setConstraints(sys_param,rel_op,[float_val|int_val|string_val]);

Each method invocation adds a constraint with the following pattern:

sys_par rel_op value

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

```
JSConstraints constr = new JSConstraints();
constr.setConstraints(JSConstraints.C_HOST_URL,"!=","milena");
constr.setConstraints(JSConstraints.C_CPU_IDLE,">=",90.0f);
constr.setConstraints(JSConstraints.C_CPU_SYS,"<=",50);
```

```
constr.setConstraints(JSConstraints.C_MEMORY_FREE_KB,
">=",10240);
```

A set of constraints is collected in object *constr* which specify that only computing nodes whose name is not "milena"can be included in a VA, the computing system should execute less than 50 % in system mode, is idle for more than 90 %, and has at least 10240 Kbytes of unused memory. The programmer can define constraints defined over approximately 40 different system parameters. In the following sections, we will show how system parameter values can be retrieved for VAs with different levels.

Creating VAs

The JS class *VA* is provided to define the topology for VAs. A set of constructors enable to create complex topologies with a single line of code. The task to allocate and administrate VAs is left to the JRS. Moreover, VAs can also be built bottom-up by starting with lower level VAs and then combine them to higher level VAs.

```
JSConstraints constr;
    // request level-1 VA
VA v1 = new VA(1);
    // request level-1 VA for which constraints hold
VA v2 = new VA(1, constr);
    // bottom-up request for level-2 VA by adding existing VAs to it
VA v3 = new VA(2);
v3.addVA(v1);
v3.addVA(v2);
    // request for level-4 VA (see Fig. 1) with 2 level-3 VA's:
    // first level-3 VA with 3 level-2 VAs with 2, 3,
    //            and 1 level-1 VAs, respectively
    // second level-3 VA with 2 level-2 VAs with 3
    //            and 2 level-1 VAs, respectively
VA v4 = new VA(4, new int[][] {{2,3,1}, {3,2}});
```

Modifying VA

The JRS returns a handle for every generated VA. These handles are first order objects which can be passed as parameters to methods. Any thread with a handle to a VA has access to and can modify or even free this VA. Concurrent changes to VAs can be prevented by using a lock/unlock mechanism provided by JS. If a thread t has a handle to a VA v and locks v, then no other thread can access v until thread t unlocks v again. A lock operation on a VA v is delayed until all unfinished methods on v have completed execution. It is recommended to use the lock/unlock mechanism in order to avoid inconsistent modification of VAs.

```
VA v1 = new VA(4, new int[][] {{1,3,2}, {2,2}});
VA v2 = new VA(3, new int[] {4,2,1});
...
v1.lock();        // lock v1 before modifying it
v1.addVA(v2);  // change v1
    // delete the first successor of the second successor of v1
v1.free(new int[]{2,1})
v1.unlock();      // unlock v1
v1.free();          // delete v1 when it is not needed anymore
```

Figure 2 illustrates the modifications applied to *v1* inside of the lock/unlock region of the previous example.

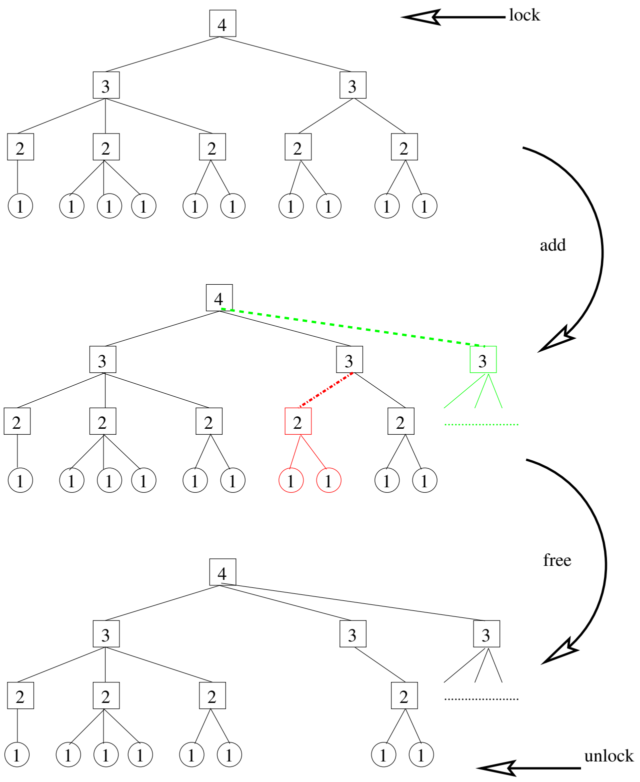

Figure 2. Modifying VAs in JavaSymphony.

Retrieving information about VA's

Specific information can be obtained for a VA, like system parameters (static or dynamic) or information about the topology (e.g., the parent or the successors in the tree structure of a VA). A static method *getLocalNode* is provided to obtain the computing node where the code is executed. The following code excerpt shows these features.

```
    // get system parameters for a VA v1 with level i
float cpuIdle = v1.getSysParamAsFloat(JSConstraints.C_CPU_IDLE);
String sURL = v1.getSysParamAsString(JSConstraints.C_HOST_URL);
int swap =
    v1.getSysParamAsInt(JSConstraints.C_SWAP_SPACE_AVAIL);
    // obtain the VA level and parent VA of v1
VA v2 = v1.getPred(v1.getLevel()+1);
    // obtain the n-th successor of v1
VA vn = v1.getVA(n);
    // obtain the local level-1 VA on which this program runs
```

```
VA v_local = VA.getLocalNode();
    // obtain the second node of the second level-2 VA
    // of the first level-3 VA of a level-4 VA
VA v4 = new VA(4, new int[][] {{1,3,2}, {2,2}});
VA v1 = v4.getVA(new int[]{1,2,2});

    // number of level-i VAs in v4
int nrLeveli = v4.nrVA(i);
    // traverse level-i VAs of v4
VAEnum e = v4.enumerateVA(1);
while (e.hasMoreVA())
{
    VA v5 = e.nextVA();
    ...
}
```

Examine System Constraints

System constraints can be used to control load balancing, to improve the program's performance, to honor computing site policies, etc. These constraints defined at the creation of the VA or created during runtime, can be examined whether they (still) hold:

```
JSConstraints constr;
    // check if the initial constraints of a VA v still hold
boolean itHolds = v.constrHold();
    // check if specific constraints hold for VA v
boolean itHolds = v.constrHold(constr);
```

4.2 Create, Map, Convert, and Free JS Objects

In order to use JS to distribute objects onto virtual architectures, we first need to encapsulate these objects into JS objects. Assuming that class files are available on every component of a VA where needed, JS objects can be created by generating instances of class *JSObject* which is part of the JS class library. A set of JSObject constructors allows us to specify the original class name for which an object is encapsulated in a JS object, the constructor arguments for this object, whether the JS object is single-threaded or multi-threaded, and the JS object location together with constraints (control where the JS object should be generated). The exact location can be indicated by providing a level-1 VA. If a higher level VA v (with level greater or equal than 2) with/without constraints (see Section 4.1) is specified, then the JRS tries to determine a level-1 VA in v which honors all constraints indicated. If only constraints but no location are provided then the JRS searches for a location that fulfills all constraints. If neither location nor constraints are provided, then the JRS will use a default location based on configuration constraints (e.g., a level-1 VA with the smallest system load and reasonable resources available) set under the JS-Shell.

```
VA v1 = new VA(1); // allocate level-1 VA
VA v2 = new VA(4,....); // allocate level-4 VA
VA vLocal = VA.getLocalNode(); // get local level-1 VA
JSConstraints constr;
    // parameters for the new object
Object[] args = new Object[] {new Integer(10)};

    // generate an object of class "ClassName" at a VA decided
    // by JRS, restricted to constraints or at the local level-1 VA
JSObject obj1 = new JSObject("ClassName"[,args][,constr][,vLocal]);

    // generate an object on a higher level VA; JRS decides
    // on which level-1 VA of v2 this object will be generated
JSObject obj1 = new JSObject("ClassName",[args,] v2);

    // generate object on a specific VA v1
JSObject obj1 = new JSObject("ClassName",[args,] v1);

    // generate obj1 on the same level-1 VA where obj2 was generated
JSObject obj1 = new JSObject("ClassName" ,obj2.getVA());
```

Every JS object can be created single- (default option) or multi-threaded which is an attribute of the object that can be changed dynamically. A single-threaded object has one thread associated with it that executes all of its methods. Whereas, a multi-threaded object can be assigned multiple threads by the JRS that execute its methods simultaneously. Even a single method of a multi-threaded object can be executed by multiple threads in parallel. The number of threads incorporated to execute multi-threaded objects can be changed dynamically through the JS-Shell. A multi-threaded object can benefit from multiprocessor (for instance, SMP) nodes where several threads – that process an multi-threaded object – share a common memory but are executed on different processors.

```
    // generate a multi-threaded object in a node of
    // a higher level VA v that honors a set of constraints
boolean multiThreaded = true;
JSObject obj1 = new JSObject(multiThreaded, "ClassName" [,args]
                        [,constr][,v]);

    // objects can be made single- or multi-threaded at runtime
obj1.singleThreaded();
obj1.multiThreaded();

    // convert a conventional Java obj to a JS object obj2
ClassName obj = new ClassName(...);
JSObject obj2 = JSObject.convertToJSObject(obj [,multiThreaded]);
```

JS objects can be generated based on existing non-JS (conventional Java) objects through the method *convertToJSObject*. The first method parameter represents the non-JS object and the second parameter indicates whether a single- or multi-threaded object should be generated.

Migration of a JS object *obj2* that has been generated through conversion based on a non-JS object *obj* is possible, however, must be carefully handled. In case of migrating *obj2* to another VA, all previous local references to *obj* should be deleted by the programmer to avoid programming errors and memory leaks.

JS objects are accessed through handles which are first order objects. They can be passed to methods and, therefore, distributed onto VAs as well. Any thread with a handle to a JS object has access to and can modify or even free this object. However, concurrent accesses to objects can be prevented by using a JS lock/unlock mechanism. If a thread t has a handle to an object and locks it, then no other thread can access this object until thread t unlocks it again. A lock operation on an object is delayed until all unfinished methods on this object have completed execution.

```
JSObject obj;
    // lock object
obj.lock();
    // invoke methods of obj
...
    // unlock object
obj.unlock();
    // free object
obj1.free();
```

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

4.3 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, JS also offers asynchronous (non-blocking) and one-sided RMI (non-blocking without results). If a method is called on an object that resides on the local node, then the JS runtime system invokes this method within the same JVM without using the RMI mechanism.

In the following we just briefly describe the asynchronous method invocation offered by JS. More details about synchronous and one-sided RMI as supported by JS can be found in [5].

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.

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

4.4 Dynamic Object Migration

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. JS offers two forms of object migration: automatic migration [5] which is controlled by JRS or explicit migration which is controlled by the programmer. The programmer can also specify the destination VA, constraints, whether the codebase(s) should be transferred. Additionally, specific codebase(s) that should be transferred, can be indicated.

Explicit migration can be encoded by the JS application programmer. For this purpose JS allows to access system parameters for VAs. System parameters for a level-i VA are averaged across its level-(i-1) VAs which is limited to system parameter values of type int and float. Methods *getSysParamAsFloat*, *getSysParamAsInt* or *getSysParamAsString* can be called for every architecture component to examine the system parameters of interest. Moreover, it can be verified through method *constrHold* whether a set of constraints (see Section 4.1) currently hold for a given architecture component. For instance, in the following code excerpt it is examined whether a level-1 VA *v1* on which a specific object resides has less than 50 % idle time or doesn't fulfill a set of constraints. 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 object. These decisions can be influenced by using the JS-Shell.

```
JSConstraints constr;
VA v1 = new VA(1); VA v2 = new VA(4,...);
JSCodebase cb; JSObject obj;
    ...
VA v3 = obj.getVA();
        // v3 on which obj resides has less than 50 % idle time
        // or constr do not hold for v3
if (v3.getSysParamAsFloat(JSConstraints.C_CPU_IDLE) < 50) ||
    !v3.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, true);
        // migrate object to a specific node
    obj.migrate(v1);
        // migrate object to a node of v2 to be destined by JRS
    obj.migrate(v2);

        // migrate object and move codebase to the destination VA
```

```
        obj.migrate(va, cb);
        obj.migrate(constr, cb);
        obj.migrate(cb);
}
```

A method *migrate* can be invoked with a node as a parameter that defines where to migrate anf object. If constraints are indicated then a node found by JRS that honors the constraints, is chosen as the target node. A Java exception will be thrown if no suitable node for migration is found. The exception must be handled by the application programmer.

5 Experiments

We have implemented a JS runtime system (JRS) that implements all the JS features described in this paper. The JRS is implemented as an agent based system that consists of a network agent system (NAS), an object agent system (OAS), and a Java JS Administration Shell (JS-Shell). The NAS is responsible to monitor the system behavior, to support the JS virtual architectures, to provide an API to system parameter and a limited fault tolerance mechanism. The OAS provides the interface to the JS application programs. It supports the administration of objects which includes creation, mapping, migration, load balancing, and deletion of objects. Furthermore, the OAS is responsible to manage RMIs, transfer method parameters, to execute these methods and the object's locations, and to return the corresponding results to the call site. The JS-Shell is used to configure physical machines, parallel architectures, clusters, grid systems, etc. under the JRS. If a VA with some constraints is requested by a JS programmer, then the NAS tries to match it with an actual physical architecture that closely resembles the requested VA and honors all constraints indicated.

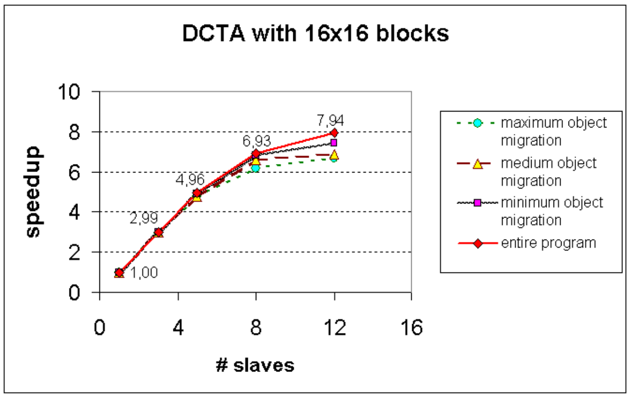

Figure 3. JavaSymphony DCTA performance for different number of slaves on a NOW.

The JS-Shell allows to configure arbitrary complex physical distributed architectures (DA) to match the structure of VAs. The mapping between VAs (requested within JS programs) and DAs is currently implemented based on the following mechanism: Every VA v with level-i must fit into a DA d with the smallest possible level-j such that all level-1 VAs of v are mapped to the physical nodes of d which assumes that the number of nodes of v is less or equal than the number of nodes of d. Several distinct VAs can be mapped to the same DA or to any component of a DA. Additionally, the mapping considers the system constraints specified when requesting VAs. If a VA is requested whose constraints cannot be fulfilled or for which no suitable DA can be found, then a Java exception is thrown which must be handled by the JS application programmer.

In this section we present various JS experiments. Firstly, we demonstrate JS for an image processing DCTA (Discrete cosine transformation algorithm) on a NOW (network of workstations) and on a SMP (symmetric multiprocessor) cluster architecture. Secondly, we have implemented a JS Jacobi Relaxation on an SMP cluster architecture.

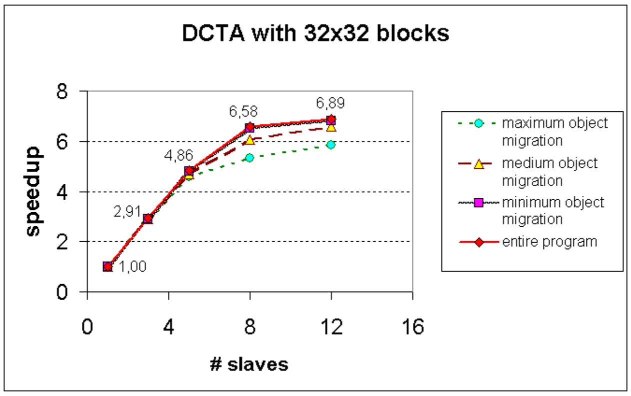

Figure 4. JavaSymphony DCTA performance for different number of slaves on a NOW.

5.1 JavaSymphony DCTA

The DCTA [18] can be used to eliminate non-essential information from images and compress digital data. This algorithm is commonly used to compress JPEG images.

We use the master/slave paradigm to encode a JS DCTA. The master sub-divides the image into square blocks of identical sizes (16x16, and 32x32). These blocks are grouped into jobs that are distributed onto a set of slaves (workstations or SMP nodes). After a slave has finished its job, it sends back the results to the master and requests a new job. Every slave is encoded by one JS object. The master will invoke job executions on these objects through JS asynchronous remote method invocations. The results of these jobs are transferred back to the master which then restores the image.

Experiment 1: Heterogeneous Network of Workstations

In this section we describe an experiment that has been conducted on a non-dedicated heterogeneous NOW with up to 13 Sun workstations comprising 6x Sun Ultras 10/440, 1x Sun Ultra 10/333, 2x Sun Ultra 10/300 and 4x Ultra 1/140-170. All Sun Ultra workstations are connected based on 100 Mbits/sec bandwidth and run Sun Solaris 8. 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.

Figures 3 and 4 visualize the speedup for the entire DCTA program measured at the master for varying number of workstations (slaves) and different problem sizes.

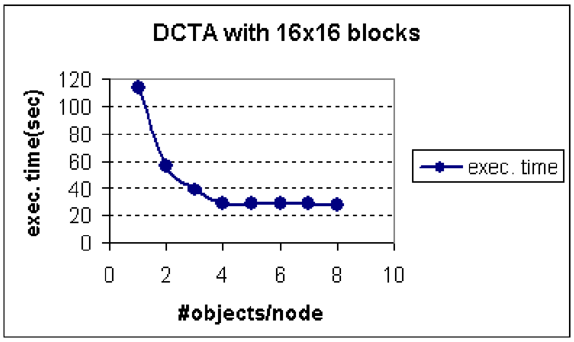

Figure 5. Performance for different number of objects per node on a single SMP node.

We observe that the DCTA performance scales for up to 5 slaves. Thereafter, we use substantially slower workstations. The fastest workstations are up to 4 times faster than the slowest workstations. At the very end of an execution, the slowest workstations are still busy with the assigned jobs whereas the faster workstations are already finished. This effect impacts the load balancing and consequently prevents any further scaling of the DCTA for more than 5 slaves on the given NOW.

Moreover, we evaluated the overhead implied by migrating objects. For this purpose we tested 3 DCTA scenarios with different degrees of artificial migration of slave objects without changing the overall load balance: minimum migration (10-20% of the objects migrate), medium migration (50% of the objects migrate) and maximum migration (all objects migrate). Note that all migration experiments did not change the load balance of objects. For every object migrated another object has been received. This policy allowed us to observe the unaltered overhead induced by object migration without side-effects caused by load imbalance.

The speedup values for these three scenarios are displayed in Figs. 3 and 4 in order to examine the extra overhead implied by migration. Due to the fact that communication costs on the given network are low compared with the computation costs, we notice that the scenarios with migration are similar to the ones without migration as long as we use faster workstations (Sun Ultra workstations). When we increase the number of workstations including also slower workstations, then the number of jobs is rather small and the average job execution time varies significantly on different workstations. This in turn aggravates the load balancing in particular for the medium and maximum migration scenario.

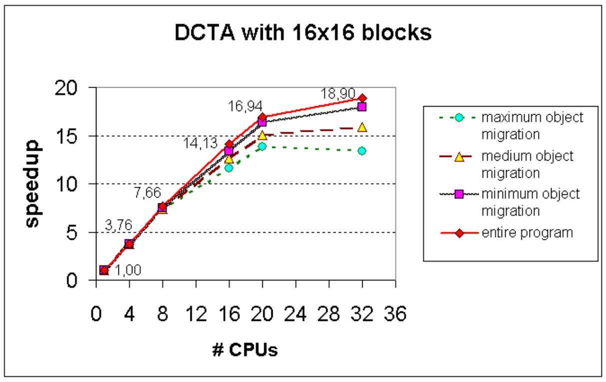

Figure 6. JavaSymphony DCTA performance for different number of SMP nodes.

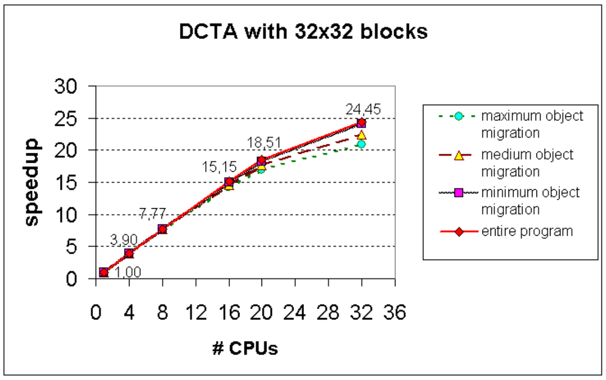

Figure 7. JavaSymphony DCTA performance for different number of SMP nodes.

Experiment 2: Cluster of SMPs

The same experiments as discussed for Experiment 1 have been conducted on a cluster of SMPs [8] with up to 8 SMP nodes (connected by FasterEthernet) each of which comprises 4 Intel Pentium III 700 MHz CPUs with 1MB L2 cache, 2Gbyte RAM, and runs under Linux 2.2.18-SMP.

The nodes on this cluster have separate IP addresses, but not their CPUs, therefore, we could not directly control the load balancing of objects through the JRS which uses the

Java RMI mechanism to distribute objects. However, Fig. 5 shows an experiment where we used a single SMP node to run the JS DCTA. By increasing the number of slave objects in a single SMP node, we can substantially improve the performance. The Linux operating system distributes the execution of JS object methods the the individual CPUs of SMP nodes. Mapping more than 4 JS objects to an SMP node with 4 CPUs does not improve the performance anymore.

Figures 6 and 7 (compare with Figs. 3 and 4) visualize again the speedup values for the entire DCTA program measured at the master for varying number of SMP nodes and different problem sizes.

Note that a single SMP node corresponds to 4 CPUs. The scaling behavior is improved compared to the experiments conducted on the workstation network, because the SMP cluster is a dedicated system that can only be used by one application at any given time. Unfortunately, the network that connects the nodes in the SMP cluster is relatively slow compared to the computing capabilities of the CPUs in this architecture. This explains why this application scales only for up to 16 CPUs.

Similar to the NOW experiments, we also evaluated the overhead implied by migrating objects. For this purpose we tested the same 3 DCTA scenarios with different degrees of artificial migration of slave objects without changing the overall load balance. According to Figs. 6 and 7, only the minimum migration scenario implies a low overhead. However, in the case of medium and maximum migration, the corresponding impact on the overall execution behavior cannot be ignored. Due to the slow network speed but fast computing capabilities of the underlying architecture, the speedup is gradually deteriorating which depends on the degree of migration.

5.2 Experiment 3: Jacobi Relaxation

In order to examine whether JS is suitable for message passing programs and to compare the performance impact of single-threaded versus multi-threaded objects, we present an experiment for a JS version of the Jacobi relaxation [21] on the same SMP cluster machine as used for Experiment 2. The Jacobi relaxation iterative method is used to approximate the solution of a partial differential equation discretized on a grid.

The algorithm consists of successive steps of computation followed by communication. We encoded a JS version that splits a square matrix into horizontal blocks which are distributed to SMP computing nodes for processing. In order to update the matrix values of a block assigned to an SMP node, we generate a JS (processing) object. Before computing new matrix values of a local block, other matrix values stored on the upper and lower neighboring SMP nodes are needed. Therefore, on every SMP node two separate JS (communication) objects are created responsible for communication and synchronization with the immediate neighboring SMP nodes. Communication objects have been generated as multi-threaded JS objects. For this experiment we compared single- versus multi-threaded processing objects on an SMP cluster configuration with a fixed number of nodes. Moreover, we employed a fixed matrix size (1000x1000) with a constant number of Jacobi iteration steps (100). In order to exemplify the effect of a multi-threaded JS object, the computational load of every processing JS object has been artificially controlled by multiplying it with a factor ranging from 1 (corresponds to original Jacobi relaxation) to 20 (original computations have been repeated 20 times). By doing so, we can examine different computation/(communication + synchronization) ratios with varying computation times but constant communication and synchronization overhead. Figure 8 visualizes the total execution time for 4 SMP nodes of the JS Jacobi relaxation based on single-threaded and multi-threaded JS processing objects, respectively. Even though the computational overhead is increased multiple times and we employ SMPs with 4 processors, the total execution time raises much slower which indicates substantial synchronization and communication costs of our Jacobi relaxation implementation. In the worst case, the performance of the Jacobi relaxation can be improved by 20 % by using multi-threaded JS objects. In the best case, the performance gain reaches 100 %. We can further conclude that by using JS multi-threaded objects for our particular JS Jacobi relaxation implementation, we can significantly improve the performance as compared with single-threaded objects.

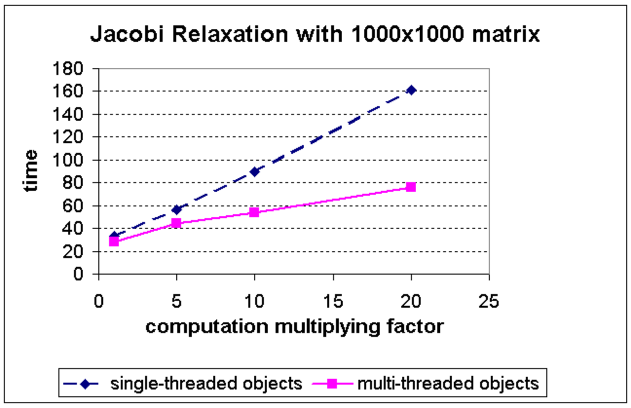

Figure 8. Comparing the effects of single- and multi-threaded JS objects for the Jacobi relaxation

6 Conclusions and Future Work

JavaSymphony is a system designed to make the development of parallel and distributed Java applications capable of seamlessly utilizing heterogeneous computing resources ranging from small-scale cluster computing to large scale GRID computing.

In contrast to most existing work, JavaSymphony allows the programmer to explicitly control locality of data, parallelism, and load balancing based on dynamic virtual distributed architectures (VAs) at a very high level. The error-prone, tedious, and time consuming low-level details (e.g. creating and handling of remote proxies for Java/RMI) are left to the underlying system. JavaSymphony virtual architectures impose a virtual hierarchy on a distributed system of physical computing nodes. JavaSymphony supports sophisticated dynamic object mapping and migrating on the basis of VAs. Conventional Java objects can be dynamically converted to JavaSymphony objects which allows to access them remotely via a variety of (synchronous/asynchronous/one-sided) remote method invocations. JavaSymphony objects can be dynamically made single- or multi-threaded depending on whether a single thread executes all methods of an object one at a time, or whether multiple threads execute the object's methods simultaneously. JavaSymphony targets distributed object computing and is particularly well-suited for applications that require shared address space, task parallelism, and one-side message passing. JavaSymphony is implemented as a collection of Java classes and runs on any standard compliant Java virtual machine. No modifications to the Java language are made and no preprocessors and no special compilers are required.

Our experimental results with an image processing application have shown that we are able to achieve reasonable performance on both a dedicated SMP cluster architecture and a non-dedicated workstation cluster. A further experiment with a Jacobi Relaxation encoded in JavaSymphony demonstrates that JavaSymphony multi-threaded objects can substantially improve the performance on SMP cluster architectures.

In the near future, we plan to evaluate JavaSymphony applications on highly dynamic and heterogeneous GRID architectures as well. Moreover, we continue to improve our techniques for automatic mapping and migration of objects and also investigate novel features to further extend the possibilities for programmers to control performance critical strategies at a high level.

References

[1] M. N. Alan. Javelin 2.0: Java-based parallel computing on the internet.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. SuperWeb: Towards a global web-based parallel computing infrastructure. In *The 11 th IEEE International Parallel Processing Symposium (IPPS)*, pages 100–106, 1997.

[3] Y. Aridor, M. Factor, and A. Teperman. cjvm: A single system image of a jvm on a cluster. In *International Conference on Parallel Processing*, pages 21–24, 1999.

[4] M. Baker and R. Buyya. Cluster computing at a glance. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 3–47. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 1.

[5] T. Fahringer. JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000), www.par.univie.ac.at/project/javasymphony*, Chemnitz, Germany, 2000. IEEE Computer Society.

[6] 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.

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

[8] Gescher homepage: http://gescher.vcpc.univie.ac.at.

[9] 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.

[10] I. Foster and C. Kesselman and S. Tuecke. The Anatomy of the Grid. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[11] 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.

[12] T. Kielmann, P. Hatcher, L. Bouge, and H. Bal. Enabling Java for High-Performance Computing. *Communications of the ACM*, 44(10):110–117, October 2001.

[13] S. Lalis and A. Karipidis. Jaws: An open market-based framework for distributed computing over the internet, 2000.

[14] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, P. Wu, and G. Almasi. The NINJA Project. *Communications of the ACM*, 44(10):102–109, October 2001.

[15] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Nov. 1997.

[16] L. F. G. Sarmenta, S. Hirano, and S. A. Ward. Towards Bayanihan: Building an extensible framework for volunteer computing using Java. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages ??–??, New York, NY, USA, 1998. ACM Press.

[17] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.

[18] G. Strang. The discrete cosine transform. *SIAM Review*, 41(1):135–147, Mar. 1999.

[19] 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.

[20] 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.

[21] W. Vetterling, S. Teukolsky, W. Press, and B. Flannery. *Numerical Recipes: Example Book (FORTRAN)*. Cambridge University Press, 1990.