JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-Computing*

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

ABSTRACT

There has been an increasing research interest in extending the use of Java towards performance-oriented programming for distributed and concurrent applications. Numerous research projects have introduced class libraries or language extensions for Java in order to support automatic management of locality, parallelism and load balancing which is almost entirely under the control of a runtime system and frequently results in critical performance problems. In previous work we described JavaSymphony to substantially alleviate this problem. JavaSymphony is a Java class library that allows the programmer to control parallelism, load balancing, and locality at a high level. Objects can be explicitly distributed and migrated based on a high-level API to static/dynamic system parameters and dynamic virtual distributed architectures which impose a virtual hierarchy on a distributed system of physical computing nodes.

In this paper we describe various important extensions to the original JavaSymphony API which includes a generalization of virtual architectures that can be used to specify and to request arbitrary heterogeneous distributed and concurrent architectures. The number of threads that execute an object's methods can be controlled dynamically through single- and multi-threaded objects. Conventional Java objects can be dynamically converted to JavaSymphony objects. A (un)lock mechanism has been introduced in order to avoid inconsistent modification of objects or virtual architectures. A sophisticated event mechanism for asynchronous communication, coordination, and interaction is provided. Several synchronization constructs including barrier synchronization and synchronization for asynchronous method invocations have been included.

Experiments are presented to demonstrate the effectiveness and efficiency of JavaSymphony.

1. INTRODUCTION

The usage of Java for performance-oriented or even high-performance parallel and distributed applications has been and still is rather a controversial issue. On the one hand, Java [2] offers code mobility, object-orientation, portability, interoperability, multi-threading, synchronization, and communication APIs, all of which is highly useful to implement parallel and distributed applications. On the other hand, Java programs are commonly interpreted which slows down the resulting performance significantly. Several efforts to improve the Java execution behavior have been made in the last couple of years, such as just-in-time compilation and many optimizations in the Java Virtual Machine (JVM) implementation. Recent results [9] showed that optimized Java code can perform comparably to Fortran or C for specific classes of applications. Moreover, 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 lead to performance degradation as the underlying runtime system lacks sufficient information about the distributed Java application. In many cases programmers are very much aware of the particular nature of their application, how to distribute data, which data should be mapped together with other data, when to migrate data, etc. Programming paradigms that disable the programmer to provide the runtime system with this information may neglect a substantial potential for performance improvement.

We have introduced JavaSymphony [6] which is a programming paradigm for wide classes of heterogeneous systems ranging from small-scale cluster architectures [4] to large scale GRID infrastructures [7]. JavaSymphony is a 100% Java 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 low-level details (e.g. generating and handling of remote proxies for Java/RMI, thread programming, or socket communication).

JavaSymphony supports so-called dynamic virtual distributed architectures (VAs) which impose a virtual hierarchy on a distributed system of physical computing nodes. A high-level API to a large variety of static and dynamic system parameters including peak computing and storage capability, idle times, available memory size, network latency, etc. is provided which can be used to constrain and specify arbitrary complex architectures. JavaSymphony supports dynamic object mapping and migration on the basis of VAs. Additionally, JavaSymphony provides persistent objects that enable the programmer to explicitly store and load objects to/from external storage. Class-loading to specific architecture components may reduce the overall memory requirement of an application. JavaSymphony targets distributed object computing and is particularly well-suited for applications that require shared address space, task parallelism, one-sided message passing, and asynchronous interaction and coordination. 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/special compilers are required.

In this paper we describe various important extensions to our original JavaSymphony API. This includes a generalization of VAs such that every VA defines a computing infrastructure subdivided into different components each of which is associated with a unique level. 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. Conventional Java objects can be dynamically converted to JavaSymphony objects in order to access them remotely via a variety of (synchronous/asynchronous/one-sided) remote method invocations. VAs and objects can be globally locked and unlocked for exclusive access and modification operations. A sophisticated event mechanism for asynchronous communication, coordination and interaction is provided. Moreover, we included several synchronization constructs not available in standard Java including barrier synchronization and synchronization for asynchronous method invocations. Note that due to space limitations, this paper focuses mostly on describing the new JavaSymphony language constructs. For implementation details, the reader may refer to [12].

The next section discusses related work. In Section 3 we describe novel JavaSymphony concepts including JavaSymphony dynamic virtual architectures, objects, events, and synchronization mechanisms. Section 4 presents experimental results. Finally, some concluding remarks are made and future work is outlined in Section 5.

2. RELATED WORK

There is a large amount of related work that tries to overcome system complexity for the development of performance-oriented distributed and parallel Java programs. These efforts can be broadly classified into two categories. The first category (e.g. JavaParty [10] and Charlotte [5]) extends the Java language itself which requires to employ a pre-compiler or changes to the Java compiler and/or JVM. The second category (e.g. JavaSymphony) provides a class library that runs on any standard compliant Java virtual machine.

Charlotte [5] supports a distributed shared memory on top of the JVM by changing the semantics of Java. It does not enable the programmer to control locality of data. Instead, programs supported by Charlotte alternate sequential and parallel phases and define routines for parallel execution.

Javelin [1], Ninflet [14], and JawS [8], employ a three-tier architecture with the entities: brokers, clients, hosts. Clients seeking computing resources by submitting their work in form of applets or Java objects, register with a broker and submit their work in the form of an applet/Java object. Hosts are donating resources, contact the broker and run 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. Migration mechanism is provided by Ninflet and Jaws.

JavaParty [10] extends Java with a class modifier *remote*. Objects generated for remote classes are distributed among several computing resources. JavaParty simplifies RMI programming at the cost of increased complexity of the actual Java code produced. ProActive (Java//), provides extensive functionality (polymorphism, future objects, sophisticated synchronization libraries, etc.) The RMI details are not sufficiently transparent and the programming effort can be considerable:

3. JAVASYMPHONY

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. JavaSymphony 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. 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, socket communication, thread programming) to the underlying system whereas the programmer controls the most important strategic decisions.

Commonly, every JavaSymphony application first must register with the JavaSymphony runtime system (JRS). Thereafter, virtual architectures can be requested. 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 VA. 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 before it terminates.

In the remainder of this section we describe the extensions of JavaSymphony which includes novel concepts about JavaSymphony virtual architectures, objects, synchronization, and events.

3.1 Dynamic Virtual Distributed Architectures

JavaSymphony 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. VAs (see Fig. 1) cover wide classes of heterogeneous systems ranging from small-scale cluster architectures to large scale Grid infrastructures. VAs consist of a set of components each of which is associated with a level:

A 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 a level-1 VA as a (computing) node in the remainder of this paper. A level-2 VA refers to a cluster of level-1 VAs (e.g. workstation or PC cluster). A level-3 VA defines a cluster of geographically distributed level-2 VAs connected, for instance, by a wide area network. A level-i VA with $i \geq 2$ denotes a cluster of level-(i-1) VAs which includes arbitrary complex heterogeneous GRID architectures distributed across several continents.

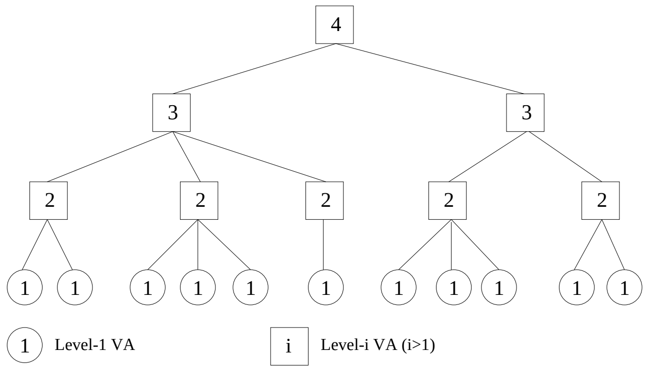

Figure 1: Example of a JavaSymphony level-4 virtual architecture which can be created with the following command: VA(4,new int[][]{{2,3,1}{3,2}})

JavaSymphony allows dynamic and flexible creation and modifications of VAs. VAs which correspond to complex topologies can be created top-down with a single line of code (see Fig. 1) or bottom-up, by combining lower level VAs into higher level VAs which is illustrated by the following code excerpt:

```
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 level-1 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}});
```

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.

A key advantage of JavaSymphony over other systems is the provision of a high-level API to static and dynamic parameters. The user can specify system constraints over these parameters 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. On the other hand, constraints can be use to examine the properties of physical resources, which comprise static parameters like machine name, operating system, peek performance parameters, etc., or dynamic parameters such as system load, idle times, available memory etc. Constraints are added to a *JSConstraints* object by invoking calls to the following method:

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

Each method invocation adds a constraint with the following pattern: *sys_param 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,"!=","r2d2");
constr.setConstraints(JSConstraints.C_CPU_IDLE,">=",90.0f);
constr.setConstraints(JSConstraints.C_MEMORY_FREE_KB,
">=",10240);
```

A set of constraints is collected in object *constr* which specifies that only computing nodes whose name is not "r2d2" can be included in a VA, the computing system should be 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. More details about the API to system parameters and system constraints can be found in [12, 6].

3.2 JavaSymphony Objects

In order to use JavaSymphony 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 a class *JSObject* which is part of the JavaSymphony class library. A set of JSObject constructors allow the specification of 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 (see subsection 3.3), and the JS object location together with constraints which control where the JS object should be created. 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 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[] {...};

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

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

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

    // create obj1 on the same level-1 VA
    // where obj2 has been created
JSObject obj1 = new JSObject("ClassName",obj2.getVA());
```

In what follows, we briefly mention some important features of JS objects. For more details the reader may refer to [6, 12].

- (Un)lock mechanism for JSObjects: 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 invoke methods of this object. In order to provide consistent modification and exclusive access to JS objects, JS provides an (un)lock mechanism. If a thread t has a handle to an object and locks it, then no other thread can access this object (its methods) until thread t unlocks it again. A lock operation on an object is delayed until all unfinished methods on this object have completed execution.
- 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 (*sinvoke*), JavaSymphony also offers non-blocking asynchronous (*ainvoke*) and one-sided RMI (*oinvoke* - non-blocking without results). All three method invocation types have similar signatures: a method name followed by a list of parameters and optionally a list of parameter types are specified, but the returned results are different: for *sinvoke* an object which represents the actual result of the method is returned. For *ainvoke* we obtain a *ResultHandle* which allows to retrieve the result at a later time. No result is returned by *oinvoke*.
- 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. JavaSymphony offers two forms of object migration: automatic migration which is controlled by JRS or explicit migration which is controlled by the programmer. The programmer can also specify the destination VA, constraints, and whether or not the codebase(s) should be transferred to the destination VA.

3.3 Single- and Multi-threaded Objects

Every JS object can be processed in single- or multi-threaded mode 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. This feature allows to impose a sequential order on all remote method invocations of an object. 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.

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

JS objects can be created 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 created. In this way, conventional non-JS objects can be accessed remotely via JS method invocations. Conventional objects may exist before their corresponding JS objects are created. Migration of a JS object *obj2* that has been created through conversion based on a non-JS object *obj* is possible.

3.4 Synchronization of Asynchronous Method Invocations

In concurrent and distributed systems programmers commonly synchronize a set of threads or processes. In the presence of asynchronous method invocations, we found numerous cases, where a set of threads that execute methods simultaneously possibly on different computing nodes, should be synchronized in a join operation.

For this purpose, JavaSymphony enables a programmer to group a set of result handles – each one associated with a unique asynchronous remote method invocation – by using a class *ResultHandleGroup*. This class provides several methods to block or examine (without blocking) whether one, a certain number, or all threads finished processing their methods. Frequently methods of different objects – that reside on different computing nodes – are executed in parallel in order to implement load balancing. By using a synchronization mechanism we can easily determine which object is idle because the execution of its method has finished. In the following code excerpt, we demonstrate how to synchronize asynchronous method invocations.

```
JSObject obj[n];
ResultHandleSet rhs;
ResultHandle rh;
Object[] params;
...
for(i=0; i < n; i++) {
      // add a ResultHandle and index i (optional) to
      // the ResultHandleSet rhs
```

```
    rhs.add(obj[i].ainvoke("run", params), i);
}
...
    //non-blocking test if at least 5 methods are finished
if( rhs.isReady(5) ) {...}
    //non-blocking test if all methods are finished
if( rhs.isAllReady() ) {...}

    //block until at least 5 methods are finished
if( rhs.waitReady(5) ) {...}
    // block until all methods are finished
if( rhs.waitAll () ) {...}
    // get results one by one without specific order;
    // block until the first method has returned results
rh = rhs.getFirstReady();
while(rh ! = null){
        // get the results
    ResultClass result = (ResultClass)rh.getResult();
    ... // process results
        // get index of idle object
    index = rhs.getIndex(rh);
        // invoke next method on idle object for load balancing
        // and add ResultHandle in ResultHandleSet again
    rhs.add( obj[index].ainvoke("run", params), index);
        // get ResultHandle of a method that finishes next;
        // block until results returned
    rh = rhs.getNextReady(); }
```

3.5 Barrier-like Synchronization

JavaSymphony provides a barrier for methods called in JS objects. The flow of control for a set of threads can be suspended until all of these threads reach a certain barrier point. Once all threads reached the barrier point, all of them can resume execution simultaneously.

The mechanism is straightforward, but improper usage can lead to performance degradation or even deadlocks. A number of barriers can be defined for each JS application by using a static method *newBarrier* – part of JSRegistry class – with an identifier that uniquely identifies the barrier, and the number of the threads n which have to wait at this barrier. The barrier is visible by all objects of the application. The execution of the threads reaching a barrier point is suspended until n threads reach this point. Thereafter, all threads can resume execution beyond the barrier. The following code excerpt demonstrates the usage of the JS barrier operation.

```
    // a barrierId defines a unique synchronization point
int barrierId = 17;
    // define a barrier for two (remote) threads.
JSRegistry.newBarrier(2, barrierId);
obj1.oinvoke("runThread1", params);
obj2.oinvoke("runThread2", params);
...
    // inside runThread1
int barrierId = 17;
    // suspend execution until runThread2
    // reaches the synchronization point
JSRegistry.barrier(barrierId);
...

    // inside runThread2
int barrierId = 17;
    // suspend execution until runThread1
    // reaches the synchronization point
JSRegistry.barrier(barrierId);
...
```

3.6 JavaSymphony Events

Programs that incorporate objects reacting to a change of state somewhere outside the objects possible on a different computing site, are common in both single address space and distributed systems. Commonly user or system actions are modeled as events to which other objects in the program react. Events also represent a mechanism for asynchronous communication. Java has a number of event models, differing in various subtle ways. All of these involve an object generating an event in response to some change of state either in the object itself or in the external environment. At some earlier stage, an event consumer will have registered interest in this event and will have suitable methods called on them.

JS follows a general event model where objects can subscribe as consumers for various types of events. Events with a specific type can be produced and the registered consumers will be notified. An event consumer handles the event by providing an appropriate method. A specific advantage of the event mechanism is that JS does not restrict the types of objects which receive or produce events. Any Java object distributed by using JavaSymphony can consume or produce events without implementing dedicated interfaces or extending dedicated JS classes. JavaSymphony supports three types of events:

- **User Defined Events** are generated explicitly by the user. They are used to support asynchronous communication and interaction among arbitrary Java objects (not restricted to JS objects). The programmer must provide the code for the producer, which generates an event and a method which is invoked by the consumer when the notification for the event arrives.

- **Middleware Events** are produced and controlled by the JRS in the event of, for instance, VA unavailable, (un)registration of a new application, object (un)lock or VA (un)lock, etc. The user must provide only the method that is invoked when notification for the event arrives, whereas the JRS produces these events.

- **System Events** are invoked due to changes of dynamic system parameters such as idle time, available memory, swap space allocated, etc. All of these parameters can be accessed by the programmer through the JS API for static/dynamic system parameters (see Section 3.1). For the consumer object a set of constraints, a constant that controls the generation of an event, and a method which is invoked if the event occurs, are specified as constructor parameters. The constant determines the event generation if the set of constraints holds, does not hold, or changes. The JRS monitors the resources and examines whether the event must be generated.

An object that wants to consume a user-defined or JS-middleware event creates a *JSEventConsumer* which describes the type (middleware or user-defined) and properties of the event in which it is interested. A derived class *JSSystemEventConsumer* is used for system events. When building an instance of *JSEventConsumer*, the programmer provides the following information: a reference to the object that consumes the event, a unique event type identifier, and the consumer method which will be invoked when the event occurs; Events can be filtered by specific parameters passed to the constructor. For instance, events can be limited to a list of producers (JSObject) or to a producer that resides at a specific location (VA). A set of constants for event types is defined as part of the *JSConstants* class. For example, C_USER_TYPE implies the definition of user-defined events; C_APP_REGISTERED denotes an event generated when a new application registers itself with the JRS; C_SYSTEM_EVENT corresponds to system events. Similarly, to filter the events, C_ANY_LOCATION indicates that the consumer accepts events from any producer; C_LIST_VA_EVENT restricts the acceptance of events produced at a specific list of VAs; C_LIST_JSOBJECT_EVENT restricts acceptance of events produced by objects in a specific list of JSObjects.

For *JSSystemEventConsumer* a *JSConstraints* object encapsulate the constraints which will be checked in order to produce a system event. An additional parameter controls the generation of the event, if the constraints become valid (JS_CONSTRAINTS_HOLD), invalid (JS_CONSTRAINTS_NOT_HOLD), or their status changes (JS_CONSTRAINTS_CHANGE). For every different set of constraints, a distinct system event will be generated by the JRS which causes all consumers to be notified accordingly.

The consumer subscribes an event by using the *subscribe* method of the *JSConsumerEvent* class. If specific events should no longer be received, then the consumer will use the *unsubscribe* method.

Only user-defined events can be explicitly produced by the programmer through the *JSEventProducer* object. The first constructor parameter indicates the object that generates an event. The second parameter refers to the unique type of the generated event which must match with the second parameter of *JSEventConsumer*. Moreover, the same list of constants defined in *JSConstants* is used to restrict the list of consumers. The user-defined event must be explicitly produced by invoking the *produceEvent* method of the *JSEventProducer*. The parameters passed to this method are forwarded to the consumer methods by the JRS. The following code excerpt demonstrate the usage of the JS event mechanism.

```
// ****** Code for Event Consumer ********
...
  // define types for user defined and middleware events
int userEvType = JSConstants.C_USER_TYPE + 1;
int middleEvType = JSConstants.C_APP_REGISTERED;
JSObject listObj[]=.....; // list of remotes objects
VA listVAs[]=.....; // list of VAs
JSConstraints constr1;

  // subscribe for a user defined event; no restriction on
  // event producers; handleMethod will handle events.
JSEventConsumer cEv1 = new JSEventConsumer(this,
      userEvType, JSConstants.C_ANY_LOCATION,
      "handleMethod");
  // events can be produced only on VAs in listVAs
JSEventConsumer cEv2 = new JSEventConsumer(this,
      userEvType, JSConstants.C_LIST_VA_EVENT,
      listVAs, "handleMethod");
  // event can be produced only by JSObjects in listObj
JSEventConsumer cEv3 = new JSEventConsumer(this,
      userEvType, JSConstants.C_LIST_JSOBJECT_EVENT,
      listObj, "handleMethod");
  // subscribe for a middleware event which can be produced
  // anywhere;the event is generated when
  // a new application registers with JS
JSEventConsumer cEv4 = new JSEventConsumer(this,
      middleEvType,JSConstants.C_ANY_LOCATION,
      "handleMethod");

  // subscribe for a system event which can be produced
  // only by the VA va when the validity for a set of
  // constraints constr changes
JSSystemEventConsumer cEvSystem =
  new JSSystemEventConsumer(this, JSConstants.C_VA_EVENT,
      va, "handleMethod",
      constr, JSConstants.JS_CONSTRAINTS_CHANGE);
...
  // subscribe for event cEv1
cEv1.subscribe()
...
  // unsubscribe for event cEv1
cEv1.unsubscribe()
...

// ****** Code for Producer of User-Defined Events ********
...
int userEvType = JSConstants.C_USER_TYPE + 1;
Object listObj[]=.....; // list of remotes objects
Object listVAs[]=.....; // list of VAS

  // produces a user-defined event of type userEvType
  // no restriction on event consumers
JSEventProducer pEv1 = new JSEventProducer (this, userEvType,
  JSConstants.C_ANY_LOCATION);
  // notify only those consumers registered on VAs in listVAs
JSEventProducer pEv2 = new JSEventProducer (this, userEvType,
  JSConstants.C_LIST_VA_EVENT, listVAs);
  // notify only those consumers in listObj
JSEventProducer pEv3 = new JSEventProducer (this, userEvType,
  JSConstants.C_LIST_JSOBJECT_EVENT, listObj);
...
  // produce a user-defined event; parameters will be
  // transmitted to the handleMethod of matching consumer
Object params[]=.....;
pEv1.produceEvent(params)
```

4. EXPERIMENTS

In the following we present several experiments for three different JavaSymphony applications to examine whether JavaSymphony can be used to implement various algorithms and programming models for concurrent and distributed computing, in particular for cluster architectures.

Our experiments have been conducted on two connected beowulf cluster architectures. The first one (slow cluster) consists of 4 SMP nodes (connected by FastEthernet) with 2 CPUs (Pentium II, 400MHz, 512 MB ECC Ram) each. The second (fast) cluster consists of 16 4-way SMP nodes (connected by Myrinet) with Pentium III Xeon (700MHz) CPUs and 2GB ECC RAM main memory per SMP. Both clusters run under Linux 2.4.17-PMC-SMP and use Sun Java 2 SDK, version 1.3.1 with a JIT compiler and native threads.

4.1 DES encryption/decryption

In the following we present an experiment with the DES encryption/decryption algorithm [16] to compare the performance of three Java-based programming paradigms for concurrent and distributed applications including JavaSymphony, JavaParty, and ProActive. JavaParty is centered around semi-automatic load balancing and locality control where most strategic decisions are taken by the underlying runtime system. In contrast JavaSymphony and ProActive provide explicit user control of load balancing and locality. ProActive represents a lower-level programming paradigm that does not provide the user with an API to system information. For our experiments we used the publicly available JavaParty version 1.05b [11] and ProActive, version 0.9.1 [13].

The DES encryption/decryption algorithm uses a key of 56 bits which is extended with another 8 parity bits. DES tries to detect the key which has been used to encrypt a message using DES, based on a "brute-force" approach (every possible key is tested). The assumption is that we know a string that must appear in the encrypted message.

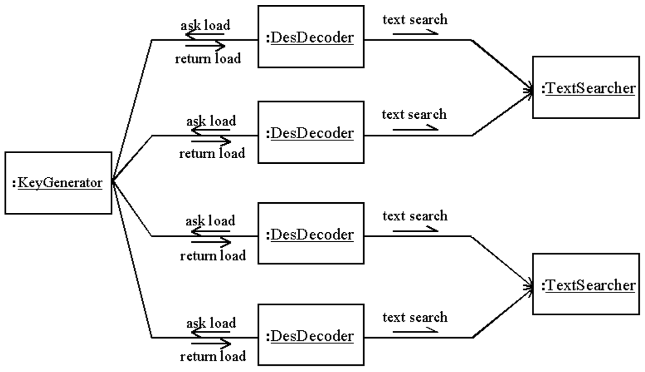

Figure 2: JavaSymphony DES decoding algorithm design

Figure 2 shows our design of the DES decoding algorithm. The *DesDecoder objects* process a space of possible keys which are provided by one or several *KeyGenerator objects*. A *DesDecoders* acquires the keys from the *KeyGenerator* through a synchronous method invocation. The *KeyGenerator* keeps track of the keys that have already been generated. After the *DesDecoders* have decoded a message by using their assigned keys, a one-sided message is invoked to transfer the decoded messages to a *TextSearcher* object that searches the known string in the messages. The 3 different DES versions have been encoded as follows:

- **JavaSymphony version:** Two different clusters are used, one for each functional component. The computational intensive component (DesDecoder) has been placed onto the fast cluster, whereas the less computational component (TextSearcher) has been mapped onto the slower cluster. The KeyGenerator component (with very small computational overhead) is placed together with the DesDecoder in order to reduce communication. JavaSymphony requires the user to create a virtual architecture and to use synchronous and one-sided method invocation.

- **JavaParty version:** The second version is based on JavaParty which introduces a new class modifier called *remote*. Only objects that are generated based on remote classes can be distributed. In order to parallelize a program, JavaParty requires the programmer to generate specific threads which execute methods of remote objects on different nodes. A JavaParty specific pre-compiler is incorporated to compile JavaParty programs to Java RMI programs.

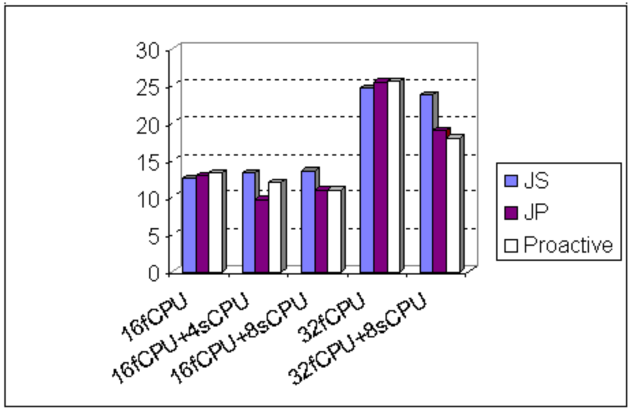

Figure 3: Comparative performance analysis for a JavaSymphony, JavaParty, and Proactive DES decryption version on a heterogeneous cluster of SMP clusters

- **ProActive version:** The third version is based on ProActive which actually provides a class library that offers extensive functionality but at a rather low level. Future objects and one-sided method invocations were used to parallelize DES. In contrast to JavaParty no explicit threads had to be generated for this purpose. The RMI details are not sufficiently transparent, and mapping objects required to start RMI servers and connect to their addresses. Objects can be explicitly mapped onto a one-dimensional architecture structure (e.g. a set of connected nodes). Java objects are automatically substituted by stubs and can be locally or remotely accessed fully transparently.

We could further improve the performance for these DES versions as follows:

- The load balancing for the *DesDecoders* can be dynamically controlled. When a DesDecoders finishes a job, it request a new set of keys from the *KeyGenerator.*

- Since text processing takes much less time compared to key processing, we made a *TextSearcher* to serve more than 1 *DesDecoder.*

- As *TextProcessors* are less computational intensive, they have been explicitly mapped onto the slower cluster, whereas the *DesDecoders* are processed by the fast cluster. JavaSymphony can determine whether a cluster is faster or slower through its high level interface to system parameters. JavaParty and ProActive do no allow to control the mapping of objects based on performance information of the underlying computation resources. For the JavaParty and ProActive version, we used a round-robin mapping strategy.

Figure 3 compares the speedup values for the three code versions for different cluster sizes where *fCPU* and *sCPU*, respectively, mean that CPUs of the faster (with Pentium

III CPUs) and slower (with Pentium II CPUs) cluster have been used. The speedup is computed relative to a sequential version run on a single CPU of the faster cluster.

JavaParty and ProActive perform only slightly better than JavaSymphony on a homogeneous cluster. This is caused by a more complex communication protocol used by JavaSymphony to hide the RMI details from the programmer. JavaParty uses a separate JavaParty pre-compiler to compile remote JavaParty objects into RMI remote objects. There is only a reduced runtime overhead involved due to direct compilation.

However, if a heterogeneous cluster architecture is available, then JavaSymphony provides the capability to map parts of an application to specific clusters. If we employ both fast and slow cluster in a single experiment then under JavaSymphony we can place the TextSearcher on the slower cluster and the other components on the faster cluster. For the heterogeneous cluster architecture, JavaSymphony clearly outperforms the JavaParty and ProActive versions. The performance for 32 fast CPUs and 8 slow CPUs is deteriorating compared to using only 32 fast CPUs as the additional communication between the two different clusters cannot be compensated with additional computing resources. But even for this cluster configuration, JavaSymphony achieves better performance than JavaParty and ProActive.

Both JavaParty and ProActive versions achieve similar performance. From this result one may conclude that the performance yielded by JavaParty, which uses a special pre-compiler, can also be obtained with a class library that uses the standard Java compiler but at the cost of a more complex programming method of ProActive that only partially shields the programmer from low level details.

Overall, we believe that for heterogeneous architectures a system such as JavaSymphony is likely to achieve superior performance compared to semi-automatic systems, as JavaSymphony allows to control parallelism, locality and dynamic load balancing. Moreover, JavaSymphony shields the programmer from low-level details by supporting a high-level programming paradigm.

4.2 Jacobi Relaxation: single- vs. multi-threaded objects

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 [15] on the fast SMP cluster machine. 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 (1-dim. row-wise distribution) 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 which are responsible for communication and synchronization with the immediate neighboring SMP nodes. Communication objects have been generated as multi-threaded JS objects. The code excerpt for this implementation is presented in Fig. 4. Synchronization is done by employing a variable which indicates what iteration is currently being processed. The computation phase for the new iteration will be suspended until the corresponding lines from the neighbors (with the same iteration number) will arrive from the lower and upper neighbors. At the end of the computation phase the updated matrix border lines will be asynchronously sent to the same neighbors.

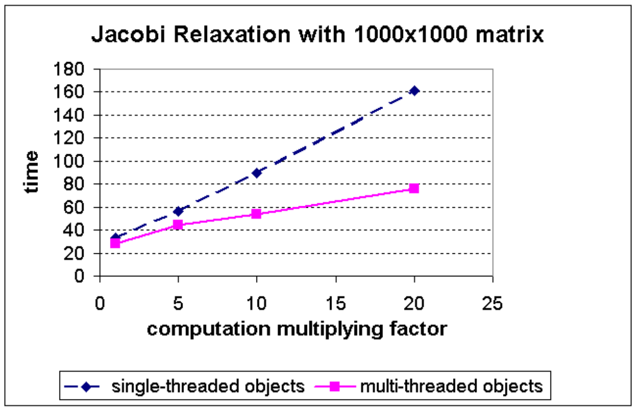

Figure 5: Comparing the effects of single- and multi-threaded JS objects for the Jacobi relaxation

For this experiment we compared two versions: one that uses single-threaded and another which employs multi-threaded processing objects on a SMP cluster configuration with a fixed number of nodes. Overall on every SMP node there will be three (two multi-threaded communication and one processing) JS objects. A single-threaded processing object can only use one CPU of an SMP node, whereas a multi-threaded object can use several CPUs and, therefore, exploit intra-node parallelism. For our experiments we used SMP nodes with 4 CPUs. The experiments are based on 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 5 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 the 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 the particular JS Jacobi re-

```
// INITIALIZATION PART - construct a communication object for upper and lower neighbor;
// communication objects encode synchronization
   downLocalObj = new JSNeighbour ();
   downJSObj = JSObject.convertToJSObject(downLocalObj);
   upLocalObj = new JSNeighbour ();
   upJSObj = JSObject.convertToJSObject(upLocalObj);
...
// ALGORITHM PART - for generic iteration
processNewIteration(){ ...
      // block until data is received from neighbors for the current iteration;
      // variable iter is used to  synchronize objects
   lowerData = (double[])downJSObj.sinvoke("getData", new Object[] { new Integer(iter) } );
   upperData = (double[])upJSObj.sinvoke("getData", new Object[] { new Integer(iter) } );
   doComputation();
   ...
      // COMMUNICATION PART - asynchronously send data to neighbors
   downJSObj.oinvoke("send", new Object[] { new Integer(iter),  lastLine } );
   upJSObj.oinvoke("send", new Object[] { new Integer(iter),  firstLine } );
   iter++; processNewIteration();
... }
```

Figure 4: JS Jacobi Relaxation (without JS events or JS barrier-synchronization)

```
// INITIALIZATION PART - encode producer and consumer to communicate with neighbors
   // producer for event to be sent to the upper neighbor; event type matches with the neighbor's consumer type
 prodUp = new JSEventProducer(thisBlock, JSConstants.C_USER_TYPE + 2* index, JSConstants.C_ANY_LOCATION, null);
  // consumer for the event produced by the upper neighbor; event type matches with neighbor's producer type
 consUp = new JSEventConsumer(thisBlock, JSConstants.C_USER_TYPE + 2* index -1,
   JSConstants.C_ANY_LOCATION, "processEvent" );
 consUp.register();
... // similar for lower object
...
// ALGORITHM PART - for generic iteration
void processNewIteration(){ ....
   doComputation();
     // matrix lines (lastLine, firstLine) are attached to the events as parameters
   prodUp.produceEvent( new Object[] { firstLine, new Integer(index), new Integer(iter)} );
   prodDown.produceEvent( new Object[] { lastLine, new Integer(index), new Integer(iter) } );
}
...
// PROCESSING EVENT
     // parameter types match with the parameters of the method that produces events
public void processEvent(double[] line, Integer source, Integer iteration) {
     // block until current iteration iter matches with the iteration sent by the neighbor
  wait_until_iteration(iteration, iter);
  update_matrix_bound(source, line);
     // test whether both data from both neighbors have been updated for the current iteration;
  if( all_data_received(iter) )
  {  // start new iteration
     iter++; processNewIteration();   }
}
```

Figure 6: JS Jacobi Relaxation with events

laxation implementation, we can significantly improve the performance as compared with single-threaded objects.

Note that the programmer could mimic a multi-threaded object on a SMP node by creating several single-threaded objects. However, by doing so, programming gets more complex as methods of different objects must be invoked to achieve the same effect with a single JS multi-threaded object. The main purpose of multi-threaded objects is to exploit parallelism within a single object on shared memory multi-processors without the need to create multiple objects and to call methods of different objects.

4.3 Jacobi Relaxation: events and barrier - synchronization

The JS Jacobi Relaxation version of Section 4.2 required explicit programming for synchronization. In this section we will exemplify two JS extensions namely events and barrier synchronization that simplify the programming effort substantially.

Figure 6 shows a Jacobi Relaxation code excerpt based on JS user-events for synchronization and communication. This version includes processing objects but no communication objects. JS-provided classes for event consumers/producers can be used to encode synchronization and communication. The variable *iter* refers to the current iteration number and synchronizes all objects before proceeding with the next iteration. Variable *iter* is transmitted via events among processing objects. Matrix border rows are transmitted as event parameters between neighboring objects. The *index* member of the processing object class uniquely identifies each processing object and is also incorporated to compute the unique event-types that refer to neighboring objects.

Figure 7 shows a code excerpt of a Jacobi Relaxation variation based on JS barrier synchronization which is the most simple version to implement. The barriers are declared in the initialization phase. A barrier is placed at the end of every iteration which blocks every thread until all threads reach this point. Thereafter, all threads continue processing at the same time by accessing the data from their neighbor-

```
// INITIALIZATION PART - for each iteration a JS barrier object is build
   // noObjects represents the number of threads which will be suspended
for(id =0; id <maxIteration; id++)
   JSRegistry.newBarrier(noObjects, id);
...
// ALGORITHM PART - iteration encoding
void processNewIteration(){ ....
  doComputation();
     // barrier-synchronization blocks until all threads reach this point
  JSRegistry.barrier(iter);
     // communication is direct and synchronous; neighbors provide the corresponding lines
  objUp.sinvoke("getLastLine", new Object[] {});
  objDown.sinvoke("getFirstLine", new Object[] {});
  iter++; processNewIteration();
... }
```

Figure 7: JS Jacobi Relaxation with JS barrier-synchronization

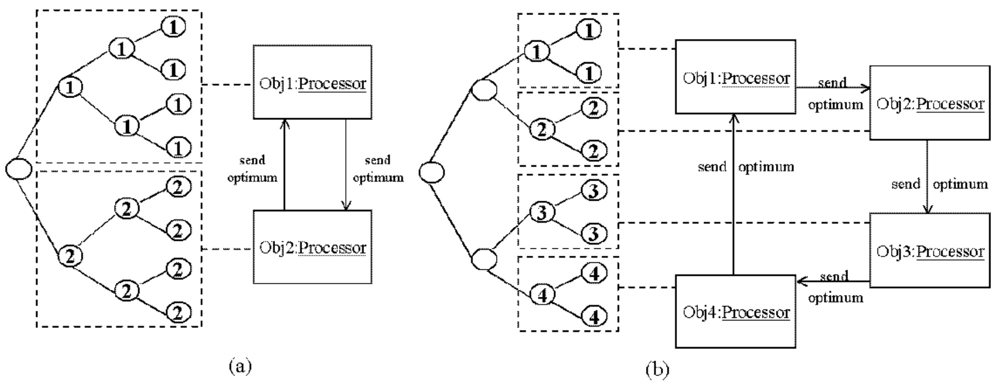

Figure 8: B&B space solution divided between 2 computing objects (a) or 4 computing objects(b)

ing objects via JS synchronous remote method invocation.

4.4 Branch&Bound application.

Branch&Bound is a technique for solving problems according to a *divide & conquer* strategy. We used this technique for solving a *discrete optimization problem* [3], which consists in searching the optimal value of a function $f : x \in Z^n \rightarrow R$, and the solution $x = \{x_1, ...x_n\} \in Z^n$ in which the function's value is optimal. $f(x)$ is called *cost function*, and its domain is generally defined by means of a set of m constraints on the points of the definition space. Constraints are generally expressed by a set of inequalities:

$\sum_{i=1}^{n} a_{i,j} x_i \leq b_j \qquad \forall\, j \in \{1, ..., m\}$

and they define the *solutions space* of the problem.

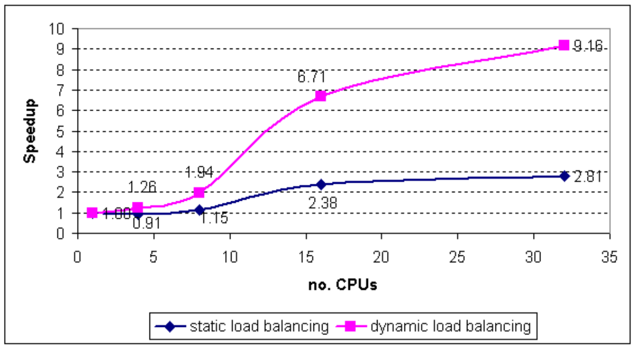

Figure 9: Exp.1. JavaSymphony Branch&Bound algorithm performance for different number of CPUs

For the sequential algorithm we used the Branch&Bound strategy described above. For the parallel implementation, the tree-searching is split between JavaSymphony objects, within the nodes of a JavaSymphony level-2 VA. We shall denote these objects as computing objects, which work in parallel to find the solution with the optimal value. Subtrees of the search tree which represent the solution space will be distributed to each computing object (see Fig. 8). Based on a level-2 VA, we can encode a ring communication pattern. When a better value for the optimum is found by one of the computing objects, it will be transmitted to its (e.g. left) neighbor in the ring. The neighbor will update its own local best value if necessary, and send this value further to its left neighbor. All the computing objects will update their local best value until one with a better or equal value stops this transmission.

Two different parallel approaches are studied. In the first approach equal parts of the general searching tree are distributed at the beginning to the computing objects, with no later redistribution, while the latest best value of the cost is transmitted to the others. In the second approach we distribute the sub-trees between the computing objects at the beginning and, in addition, work is redistributed whenever one of the computing object becomes idle.

A characteristic of B&B parallel algorithms is that the gain in performance is very much dependent on the input data. The execution time for different workloads cannot be predicted. Depending on the input data, the performance may change. We performed experiments for two different, randomly generated problems with $m = 40$ equations and $n = 10$ variables.

No. CPUs	Total time(ms)	Avg. Time(ms)	Nodes visited	Overhead 1	Overhead 2	Efficiency
Sequential						
1 CPU	186258.00		36665880			
Static load balancing						
4 CPUs	204817.00	152526.50	110959591	653.25	0	0.2273
8 CPUs	162590.00	105733.50	153053162	656.75	0	0.1432
16 CPUs	78108.00	34456.00	108259200	612.00	0	0.1490
32 CPUs	66272.00	23239.13	136903433	1006.16	0	0.0878
Dynamic load balancing						
4 CPUs	147980.00	147960.50	96469135	652.00	394.00	0.3147
8 CPUs	95847.00	95785.50	119274168	572.00	1334.00	0.2429
16 CPUs	27760.00	27225.69	72932293	463.69	2000.06	0.4193
32 CPUs	20340.00	19274.44	80687271	490.00	4457.44	0.2862

Table 1: Performance data for B&B problem 1. Overhead 1 for transferring the optimum; Overhead 2 for work redistribution

No. CPUs	Total time(ms)	Avg. Time(ms)	Nodes visited	Overhead 1	Overhead 2	Efficiency
Sequential						
1 CPU	352285.00		35378744			
Static load balancing						
4 CPUs	162750.00	157204.50	62515907	200.50	0	0.5411
8 CPUs	106455.00	101661.75	73955665	263.00	0	0.4137
16 CPUs	47838.00	37430.25	54204058	158.81	0	0.4603
32 CPUs	46667.00	33081.47	93405849	91.50	0	0.2359
Dynamic load balancing						
4 CPUs	163934.00	163906.75	62412733	176.50	262.00	0.5372
8 CPUs	105029.00	104863.63	73822916	190.25	1168.00	0.4193
16 CPUs	38989.00	38794.31	50727885	196.50	2488.50	0.5647
32 CPUs	34765.00	33795.06	80594720	120.75	4746.47	0.3167

Table 2: Performance data for B&B problem 2. Overhead 1 for transferring the optimum; Overhead 2 for work redistribution

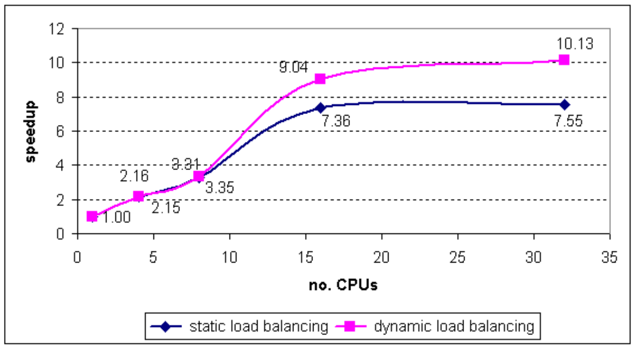

Figure 10: Exp.2. JavaSymphony Branch&Bound algorithm performance for different number of CPUs.

The speedup results are shown in Figs. 9 and 10. Notice that the performance varies quite dramatically although the problem size (m and n values) is the same for all experiments. Detailed performance measurements for the two experiments can be found in Tables 1 and 2, respectively. The number of nodes analyzed (visited) indicates the variation in computing requirements. Two overheads due to communication are presented: the overhead caused by transmitting the local optimum to the rest of the computing object and the overhead due implied by balancing the workload (only for the dynamic load balancing version). The difference between the total computation time and the average computation time reflects the overhead due to unequal workloads.

In the first experiment (Fig. 9), for up to 8 CPUs the speedup is raising only by a factor of 2 (in the dynamic load-balancing version). By using 16 CPUs, however, the performance improves by a factor of 3 compare to the 8 CPU version. Table 1 shows that the number of visited nodes in the search tree is increasing with the number of CPUs used. If we use 16 CPUs for the dynamic load balancing version, the number of nodes suddenly drops. As we have an increased number of CPUs available, it is more likely to find the optimum faster which stops the entire search. Parts of the search tree are faster eliminated and work can be redistributed. The number of the nodes searched is reduced also for the version based on static load balancing. However, the severe difference between the average and total execution time implies a poor load balance which deteriorates the overall performance.

The second experiment (Fig. 10) shows a constantly improving performance behavior for up to 16 CPUs which even yields super-linear speedup relative to an 8 CPU version (see Table 2). A version that employs 32 CPUs increases the overhead for balancing the workload in the dynamic load balancing version which prevents further gain in performance.

5. CONCLUSIONS AND FUTURE WORK

JavaSymphony is a system designed to make the development of concurrent and distributed Java applications capable of seamlessly utilizing heterogeneous computing resources ranging from small-scale cluster computing to large scale GRID computing. JavaSymphony provides a rich set of 100 % pure Java classes and methods to program important parallel and distributed programming concepts at a high-level which allows the programmer to explicitly control locality, parallelism, and load balancing.

In this paper we described several important extensions to the JavaSymphony programming API. This includes a generalization of virtual architectures that can be used to specify and to request arbitrary heterogeneous distributed architectures. The number of threads that execute an object's methods can be controlled dynamically through single- and multi-threaded objects. Conventional Java objects can be dynamically converted to JavaSymphony objects. A (un)lock mechanism has been introduced in order to avoid inconsistent modification of objects or virtual architectures. A sophisticated event mechanism for asynchronous communication and interaction is provided. Moreover, we included several synchronization constructs including barrier synchronization and synchronization for asynchronous method invocations.

We encoded three Java-based versions of a encryption/decryption algorithm which included a system that focuses on semi-automatic parallelization, a system that enables the programmer to explicitly control locality and parallelism at a rather low-level, and JavaSymphony. For this experiment, we demonstrated that JavaSymphony can substantially outperform the other two systems on a heterogeneous cluster architecture. JavaSymphony provides more explicit control to exploit the capabilities of heterogeneous architectures and loses only little performance on a homogeneous target machine. Semi-automatic systems can prevent the programmer to exploit important machine and system characteristics to further tune the performance. Whereas low-level systems that support explicit control of parallelism and locality can be tedious and complex to use. We believe that JavaSymphony represents a good compromise between semi-automatic and low-level programming that can make a substantial difference for heterogeneous concurrent and distributed architectures and for systems whose computational load changes dynamically.

We are currently evaluating JavaSymphony applications on highly dynamic and heterogeneous GRID architectures. Moreover, we investigate to upgrade the JavaSymphony programming paradigm to support mobile computing.

6. REFERENCES

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

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

[3] Aversa, R., Martino, B. D., Mazzocca, N., and Venticinque, S. Mobile agents for distribute and dynamically balanced optimization applications. in High-Performance Computing and Networking(Lecture Notes in Computer Science vol.2119), ed. By B. Hertzberger et al. (Springer, Berlin, 2001) pp.161-170, 2001.

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

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

[6] Fahringer, T. 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)* (Chemnitz, Germany, Nov. 2000), IEEE Computer Society.

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

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

[9] Moreira, J. E., Midkiff, S. P., and Gupta, M. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems 22*, 2 (Mar. 2000), 265–295.

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

[11] Javaparty homepage: http://www.ipd.uka.de/javaparty/.

[12] Javasymphony homepage: http://www.par.univie.ac.at/project/javasymphony.

[13] Proactive homepage: http://www-sop.inria.fr/sloop/javall/.

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

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

[16] Wiener, M. J. Efficient DES key search, technical report TR-244, Carleton University. In *William Stallings, Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.