On the Evaluation of JavaSymphony for Cluster Applications *

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

Beniamino Di Martino, Salvatore Venticinque
Dipartimento di Ingegneria dell' Informazione - Second University of Naples
Real Casa dell'Annunziata - via Roma, 29, 81031 Aversa (CE) - Italy
{beniamino.dimartino, salvatore.venticinque}@unina.it

Hans Moritsch
Department of Statistics and Decision Support System, University of Vienna
Universitätsstrasse 5/3 - A-1010 Vienna - Austria
hans.moritsch@univie.ac.at

Abstract

In the past few years, increasing interest has been shown in using Java as a language for performance-oriented distributed and parallel computing. Most Java-based systems that support portable parallel and distributed computing either require the programmer to deal with intricate low-level details of Java which can be a tedious, time-consuming and error-prone task, or prevent the programmer from controlling locality of data. In contrast to most existing systems, JavaSymphony – a class library written entirely in Java – allows to control parallelism, load balancing, and locality at a high level. Objects can be explicitly distributed and migrated based on virtual architectures which impose a virtual hierarchy on a distributed/parallel system of physical computing nodes. The concept of blocking/non-blocking remote method invocation is used to exchange data among distributed objects and to process work by remote objects. In this paper we evaluate the JavaSymphony programming API for a variety of distributed/parallel algorithms which comprises backtracking, N-body, encryption/decryption algorithms and asynchronous nested optimization algorithms. Performance results are presented for both homogeneous and heterogeneous cluster architectures. Moreover, we compare JavaSymphony with an alternative well-known semi-automatic system.

1 Introduction

The usage of Java for performance-oriented parallel and distributed applications [11, 9], in particular for scalable web servers, multimedia applications, and large-scale scientific applications, has gained considerable attention in recent years.

Although there is no agreement whether Java can ever satisfy the needs of high-performance driven applications, it is commonly used to exploit medium grain shared memory parallelism based on threads as well as medium to coarse grain data and task parallelism based on remote method invocation (RMI) and socket communication.

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

In previous work, we have introduced JavaSymphony [5, 6], a programming paradigm that provides a distributed and parallel computing infrastructure for wide classes of het-

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

erogeneous systems ranging from small-scale cluster computing to large scale wide area meta 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.

In this paper we examine the performance of several JavaSymphony applications on both homogeneous and heterogeneous cluster architectures. We have extensively tested the suitability of the JavaSymphony programming API for a variety of algorithms including backtracking, N-body, encryption/decryption algorithms and asynchronous nested optimization algorithms. Moreover, for one application we developed both a JavaSymphony and a JavaParty [12, 13] implementation in order to compare the effectiveness of an explicit mechanism to control parallelism (as offered by JavaSymphony) with a semi-automatic system (JavaParty).

The rest of the paper is organized as follows: Section 2 discusses related work. In the following section we give an overview of the JavaSymphony API. A detailed discussion of performance results for various JavaSymphony applications on both a homogeneous cluster and a heterogeneous cluster of SMP clusters is given in Section 4. Finally, in Section 5 some concluding remarks are made and future work is outlined.

2 Related Work

In order to improve the performance of distributed and parallel Java programs, several research groups introduced Java-based computing systems that benefit by Java's platform independence.

JavaParty [12, 13] 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.

Javelin [3], Ninflet [15], SuperWeb [1] employ a three-tier architecture with the entities: brokers, clients, hosts. Clients seeking computing resources register with a broker and submit their work in the form of an applet. Hosts are donating resources, contact the broker and run applets. Javelin makes it relatively easy for a user to act as a computing server by leveraging the existing WEB technology.

Javelin [3], Jada [4] and JavaSpaces [7] can be considered as Linda derivatives which provide either none or only very limited means (compared to the functionality offered by JavaSymphony) to control locality.

3 Overview of the JavaSymphony Programming API

JavaSymphony is a 100% Java library which supports the programmer to specify and to control locality, parallelism, and load balancing at a high level without dealing with error-prone and low-level details (e.g. creating and handling of remote proxies for Java/RMI or socket communication).

The JavaSymphony programming API is centered around two key components: dynamic virtual distributed architectures and JavaSymphony objects which are explained in the following:

3.1 Dynamic Virtual Distributed Architectures (VAs)

Most existing Java infrastructures that support performance-oriented distributed and parallel computing either hide the underlying physical architecture or assume a single flat hierarchy of a set (array) of computing nodes. This simplified architecture view does not reflect heterogeneous architectures such as clusters of clusters or Grid systems. Commonly the programmer fully depends on the underlying system to properly distribute data and computations on such architectures.

In order to alleviate this problem, 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. VAs (see Fig. 1) 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 SMP node. We also refer to it as a (computing) node in the remainder of this paper. A level-i VA with $i \geq 2$ denotes a cluster of level-(i-1) VAs which among others allows to define arbitrary complex clusters, clusters of clusters, and even heterogeneous GRID architectures distributed across several continents. Every level-(i-1) VA can only be part of one unique level-i VA. In Fig. 1 it is shown how to create a level-4 VA with 2 level-3 VA's under the JavaSymphony programming API. The first level-3 VA has 3 level-2 VAs with 2, 3, and 1 level-1 VAs, respectively. The second level-3 VA has 2 level-2 VAs with 3 and 2 level-1 VAs, respectively. A single line of JavaSymphony code is sufficient to request this rather complex virtual architecture.

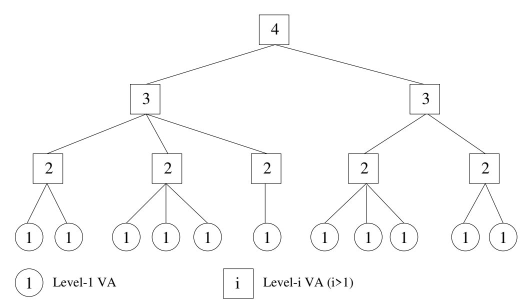

Figure 1. Example of a JavaSymphony level-4 virtual architecture which can be created with the following command: VA(4,new int[][][]{{2,3,1}{3,2}})

Key features of the VAs are the following:

- **Constraints:** Static (e.g., machine name, operating system, etc.) and dynamic parameters (e.g., system load, idle times, etc.) can be indicated when requesting a VA. Only architecture components are included into a VA that honor all of these constraints. Constraints defined over parameters can be checked any time during execution of a JS application, which enables a programmer to control, for instance, load balancing.

- **Flexible creation of VAs:** VAs can be created top-down or bottom-up. For top-down creation of VAs, indicated constraints must be fulfilled by every single node in this VA. Whereas if a VA is created in a bottom-up fashion, for every single VA component individual constraints can be specified.
- **Dynamic modification of VAs:** Computing resources can be dynamically added or removed to/from a VA.
- **Retrieving information about VAs:** A large variety of system parameters can be requested for individual VA components or as statistical values across all nodes of a VA.

3.2 JavaSymphony Objects (JS objects)

In order 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.

JS objects can be explicitly mapped to specific level-1 VAs, which may be part of a higher level VA. Constraints can be provided to control the mapping of JS objects to VAs. JS objects are accessed through handles which are first order objects. Handles 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. The key features of JS objects are given as follows:

- **Multi-threaded versus single-threaded objects:** A JS object can be created single- or multi-threaded (default option), which is an attribute of an 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.
- **Mapping and migrating JS objects:** JS objects can be dynamically migrated to a specific level-1 VA or to a higher level VA or by indicating a set of constraints (defined over system parameters) that must be fulfilled by the VA onto which an object is migrated.
- **Converting Java objects:** Conventional Java objects can be dynamically converted to JS objects which allows to access them remotely via JavaSymphony method invocations.
- **Lock/Unlock JS objects:** In order to prevent concurrent access on the same JS object possibly from different threads or VAs, JavaSymphony supports a lock/unlock mechanism.
- **Remote method invocation:** Remote method invocation is the main mechanism to exchange data among distributed objects and to process work by remote objects. Besides synchronous and one-sided method invocations, JavaSymphony also offers asynchronous method invocations, which enable the user to continue the execution and retrieve the results at a later time. The invocation does not block, but immediately returns a handle. Execution continues at the calling site. By using the handle, queries can be invoked to determine whether the result is available.

JavaSymphony provides a variety of additional functionality which is further described in [6].

4 Experiments

We have implemented the full JavaSymphony programming API as described in this paper. In the following we present several experiments whose objective was to examine whether JavaSymphony can be used to effectively and efficiently implement various distributed algorithms and programming models for parallel and distributed infrastructures 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. If not explicitly mentioned then all experiments are conducted with the second cluster. 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 Backtracking

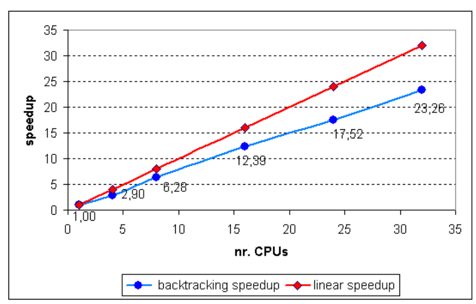

Figure 2. Speedup measurements for the distribute JS Backtracking algorithm

Backtracking is a scheme for solving a series of sub-problems each of which may have multiple solutions where the solution chosen for one sub-problem may affect the possible solutions of later sub-problems.

For the following experiment, we have chosen a particular problem that can be solved with backtracking. However, our implementation can be generalized for a large class of computing intensive backtracking problems. The selected problem has been inspired by the problem of computing the number of distinct strings when applying a *flattening operator* on a planar word [14]. For this purpose we must find

No. CPUs	exec. time	avg. exec. time	avg. ideal exec. time	avg. comp. time	avg. comm. time
Sequential					
1 CPU	823785				
Parallel					
4	283777	214585	205946	214316	219
8	131249	110709	102973	110022	603
16	66474	56395	51487	55209	1032
24	47022	39599	34324	37373	1900
32	35412	31105	25743	28089	2277

Table 1. Timings (in ms) for backtracking algorithm

all different possibilities to move a number of n identical objects along m consecutive positions in a m-tuple (1, 2, 3 ... m). One object is moved at a time from a position i to the next position $i + 1$ until all objects reside on position m such that $1 \leq i \leq m - 1$. Such a move is represented by $[i, i + 1]$. Initially all objects start at position 1 in the m-tuple. For example, if we have to move 2 objects along 3 positions in a tuple $(1, 2, 3)$, we can find two solutions, represented as ordered sets:

$\{$[1,2], [1,2], [2,3], [2,3] $\}$ and $\{$[1,2], [2,3], [1,2], [2,3]$\}$

The solution for this problem by using backtracking is obvious: Every object is moved from one position to the next until all the objects are placed on the last position. At each step, several different moves are possible. Consecutive possible moves yield sub-problems that can be solved in parallel with the other sub-problems. As simple as it looks, the requested numbers of moves raises very fast: for 5 objects and 6 positions we obtain 701149020 different possibilities for moving the objects from position 1 to 6. The sequential algorithm has exponential complexity, which requires extensive computation time.

We encoded a JavaSymphony program for this algorithm, where the sub-problems obtained at various steps are distributed among independent computing objects (JS-objects), and the search for all possible move options is done in parallel. Each computing object maintains a queue for a set of jobs (i.e. sub-problems as defined above) that can be passed to idle computing objects. This algorithm requires a high computational effort. Communication is low even in the case of dynamic load balancing when idle computing objects receive additional jobs from busy objects. Since the sub-problems have different sizes, additional overheads occur in the final phase of the program, when every computing object solves a sub-problem of the backtracking algorithm and all the queues are empty. This behavior can cause load imbalance that may deteriorate the overall performance.

Figure 2 shows the overall speedup values compared with the ideal speedup for varying number of CPUs on the fast cluster (with 4way SMPs). Even though we do not achieve perfect performance, the speedup values consistently raise with increasing number of CPUs. For this reason, we consider the described JavaSymphony implementation to be very suitable for compute intensive backtracking algorithms.

In order to explain the performance losses, Table 1 displays experimental values for the total execution time, average execution time (across all computing objects), ideal execution time (theoretically computed assuming perfect speedup), computation time (spent by every object without any communication or parallelization overhead), and communication overhead (time needed for transmitting jobs to idle computing objects). All tabulated values except total execution times are average values across all computing objects.

Clearly, load imbalance – for reasons mentioned above - can raise noticeable for increasing number of CPUs which is shown by the difference between total execution time and average execution time. The overhead for communication is reasonably small. The small difference between average computation and execution time specifies the overhead implied by the JavaSymphony middleware (see [6]).

4.2 DES encryption/decryption

A large research community is working towards highly advanced encryption algorithms with the objective to provide secure data transfers in particular within distributed systems such as the Internet. Every encryption algorithm undergoes extensive testing phases to examine the level of security it provides. In this experiment we have developed a JavaSymphony testing algorithm that tries to decrypt a message by using the well-known DES (Data Encryption Standard – [16]) method.

DES uses a key of 56 bits which is extended with another 8 parity bits. Our decryption algorithm tries to detect the key which has been used to encrypt a message by 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. For the sake of demonstration, we examine only a sub-set of all possible keys from the total number of 2^{56} possible keys. The JavaSymphony program exploits both data parallelism by dividing the set of keys to be tested among different computing objects, and functional parallelism by subdividing the application into two functional components. The first component decodes the encoded message, and the second one post-processes the decoded message in order to determine whether the correct encryption key has been found by searching the known string in the decoded message. Depending on the length of the encoded message, these two components may imply very different computational complexity.

One interesting aspect of this experiment is that the underlying functional parallelism is appropriate to examine whether two different clusters – one for each different functional component – achieve better performance compared with a single homogeneous cluster.

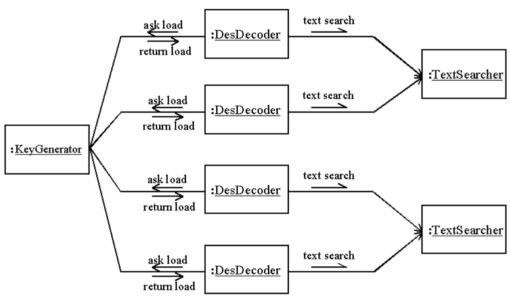

Figure 3. JavaSymphony DES decoding algorithm design

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

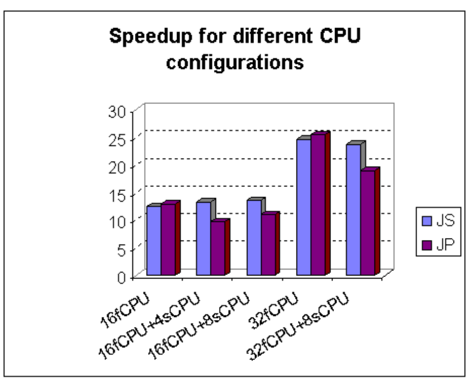

Figure 4. Performance for DES decryption algorithm: JavaSymphony versus JavaParty

We encoded two different versions in order to compare the effectiveness of JavaSymphony which explicitly allows to control parallelism and load balancing with a semi-automatic systems: The first version is based on JavaSymphony which uses two different clusters, 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 second version is based on JavaParty [12, 13] that limits user control over mapping of objects, load balancing, and locality. JavaParty extended Java by introducing a new class modifier called *remote*. Only objects that are generated based on remote classes can be distributed. Moreover, JavaParty requires to build specific threads to be run on different nodes. A JavaParty specific JVM must be used and a separate compiler is incorporated to compile JavaParty programs to Java RMI programs. JavaSymphony requires the user to create a virtual architecture and to employ asynchronous and one-sided method invocations. For our experiments we used the publicly available JavaParty version 1.05b [13]. In order to improve the performance of the algorithm we proceeded 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 enabled the *TextSearcher* to serve more than 1 *DesDecoder*.
- As *TextSearchers* are less computational intensive, they have been explicitly mapped onto the slower cluster, whereas the *DesDecoders* are processed by the fast cluster.

  JavaParty does not allow the user to control the mapping of objects but instead uses either random or round-robin mapping. For our JavaParty experiments we selected round-robin mapping which achieved better performance.

Note that the first two steps have been done for both JavaParty and JavaSymphony versions.

We tested both JavaSymphony and JavaParty versions on a homogeneous and a heterogeneous cluster of SMP clusters as outline at the beginning of this section.

Figure 4 compares the speedup values for the two 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 performs only slightly better than JavaSymphony for experiments that use a homogeneous cluster. This is caused by a more complex communication protocol used by JavaSymphony to hide the RMI details from the programmer. On the other hand JavaParty employs a separate JavaParty compiler to compile remote JavaParty objects. There is no extra runtime overhead involved due to direct compilation.

However, if a heterogeneous cluster architecture is available, then JavaSymphony can exploit its ability to map parts of an application to specific clusters. In the case where we use both fast and slow clusters that are connected via FastEthernet as described at the beginning of this section, then under JavaSymphony we can place the TextSearcher on the slower cluster and the other components on the faster cluster. For this heterogeneous cluster configuration, JavaSymphony clearly outperforms the JavaParty version. The performance for 32 fast CPUs and 8 slow CPUs is deteriorating compared to using only 32 fast CPUs because the cost of additional communication between the two different clusters cannot be compensated by additional computing resources. But again, for this configuration JavaSymphony achieves better performance than JavaParty.

Therefore, we believe that for non-dedicated (system load may dynamically change due to several unrelated applications simultaneously using the same architecture) or heterogeneous architectures, a system such as JavaSymphony which allows to to control parallelism, locality and dynamic load balancing at a high level, is likely to achieve superior performance compared to semi-automatic systems.

4.3 N-body application

This application solves the well-known N-body problem which determines the motion of a group of interacting particles starting with some initial configuration of positions and velocities in a specified volume of space. Our implementation is based on a sequential code developed in previous work [2] which incorporates a simple approximation: The force on each particle is computed by agglomerating distant particles into groups and using their total mass and center of mass as a single particle. The algorithm consists of three phases which are repeated in an outermost loop:

1. The volume in which the particles move is subdivided and represented by an octree. Every node in this tree uniquely represents a portion of the volume in which a set of particles may reside.
2. Forces aging on each particle are computed through traversing the octree.
3. The velocities and the positions of the N particles are updated by using the new values for the forces.

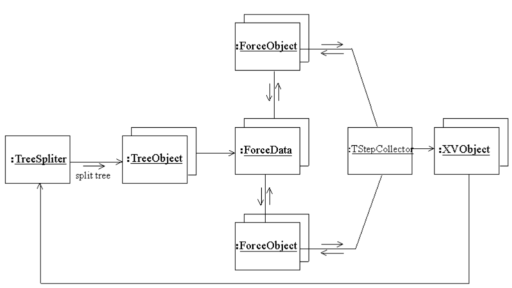

Figure 5. N-body algorithm design

We subdivided the N-body algorithm into different functional components that are encoded as separate JavaSymphony objects. Figure 5 shows these components and their interaction which are described in the following:

- One *TreeSplitter* and several *TreeObjects* process the first phase of the algorithm. The *TreeSplitter* starts building the first levels of the octree. The sub-trees are distributed to the *TreeObjects* which construct the full octree in parallel.
- Several *ForceData* objects, each being associated with a few *ForceObjects*, are responsible for the second (most computational intensive) phase of the algorithm.

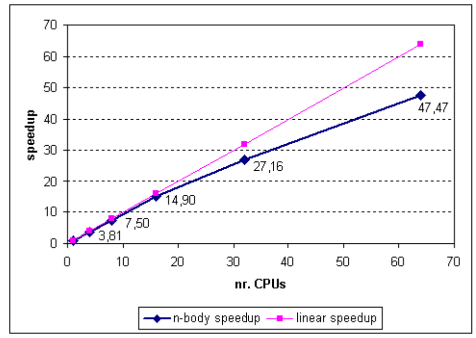

Figure 6. N-body performance behavior

The octrees created by the *TreeObjects* are collected by the *ForceData* objects and subsequently processed in parallel by the *ForceObjects*. The *ForceObjects* compute the forces for a range of particles that may depend on other particles which must be found in the octree. The *ForceObjects* have direct (local) access to the octree in the *ForceData*.

In the JavaSymphony algorithm we mapped at least one *ForceData* object and a number of *ForceObjects* (equal to the number of CPUs available on a SMP node) to the same SMP node. The *TStepCollector* controls the dynamic load balancing algorithm. Every time a *ForceObject* becomes idle, the *TStepCollector* will assign a new set of particles to be processed.

- In the final phase all particles are collected by *TStepCollector* and forwarded to the *XVObjects* which compute the new positions and velocities of all particles. Thereafter, the particles are forwarded to the *TreeSplitter* in order to restart the next iteration. This phase has not been parallelized as it represents a small portion of the overall computation. In order to eliminate extra communication the *TStepCollector*, *XVObjects*, and *TreeSplitter* have all been placed on the same SMP node.

Figure 6 visualizes a very reasonable speedup behavior for the JavaSymphony N-body application. The three algorithm phases are executed in a sequential order. Broadcast operations between phases 1 and 2 and barrier synchronization between phases 2 and 3 affect the general performance of the parallel algorithm. Phase 2 is dominating the overall computational effort. This phase can be parallelized without communication (except at the beginning and at the end) by exploiting data parallelism which explains the good scaling behavior for small and medium number of processors. For increasing number of processors, phases 1 and 3 become more significant due to the impact of barrier synchronization and broadcast operations.

Table 2 shows various timings for this experiment which includes the overall execution time and timings for all three algorithm phases. Moreover, we were able to isolate the overhead due to JS middleware in phase 2 (see the row named *phase 2 (JS overhead)*). The timings of *phase 2 (all)*

No. CPUs	Total time	phase 1	phase 2 (all)	phase 2 (JS overhead)	phase 2 (communication)	phase 3
4 CPUs	314406	1096	309797.50	12130.00	4606.25	2545
8 CPUs	163361	1174	161496.50	4002.75	1814.13	1261
16 CPUs	81132	1680	77555.94	1875.94	784.75	2611
32 CPUs	44168	2465	38849.75	876.62	528.63	3071
64 CPUs	22143	4026	12971.94	1067.56	506.75	3822

Table 2. N-Body timings (in ms)

corresponds to the execution time of the entire phase 2. We can clearly see, that phase 2 dominates the overall performance. The execution times for phases 1 and 3 increase for larger processor numbers due to the barrier and synchronization overhead, which prevents linear speedup. The JavaSymphony middleware overhead is very small, compared with the total time for phase 2. The communication overhead as part of phase 2 decreases for larger cluster sizes because the communicated data remains constant but is distributed onto an increasing set of ForceObjects.

4.4 Asynchronous Nested Benders Decomposition

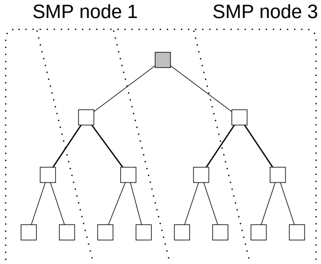

Figure 7. Mapping of 15 tree nodes onto 4 SMP nodes

The Aurora Financial Management System under development at the University of Vienna is a decision support system for asset liability management (ALM). The classical problem of dynamic asset-liability management seeks an investment strategy in which an investor chooses a portfolio of various assets, in such a way that some risk-adjusted objective is maximized (such as the return of investment), subject to the uncertainty of future markets' development, future obligations, and other constraints.

The system contains an optimization module, which solves a stochastic, multistage optimization problem in order to find an optimal investment plan. Realistic models for the future development of asset prices taking into account many risk factors result in very large scenario trees.

Among other methods, the optimization module makes use of decomposition techniques, which allow for solving large scale tree structured optimization problems very efficiently. In addition, due to their inherent parallelism, they are well suited for parallel implementation.

The entire optimization problem is decomposed into a set of local problems, corresponding to the nodes of the

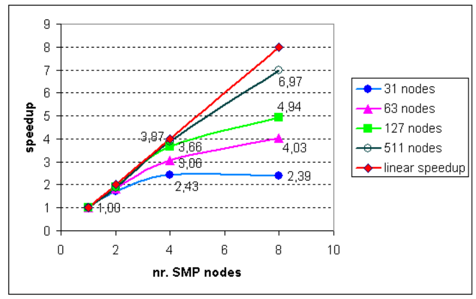

Figure 8. Speedup results for different tree sizes

scenario tree. Every tree node represents an active object (executed by a single thread). In an iterative manner, each node contributes to the solution of the entire problem by solving its local problem, updating it by using information from neighbour nodes, solving the updated problem etc.

An asynchronous parallel version of the nested Benders decomposition method has been implemented on top of different parallel programming platforms [10, 8]. The JavaSymphony version has been built starting from existing Java code, which contains an algorithm layer calling asynchronous communication operations implemented in an underlying coordination layer. The coordination layer and and the initialization part has been adapted to the JavaSymphony API. The threads associated with the tree nodes are encapsulated within JS objects; the algorithm code remained unchanged.

We used an optimal load balancing strategy for mapping the tree nodes onto SMP computing nodes (see Fig. 7). Communication among tree node objects residing on the same compute node is performed through direct access, and through JavaSymphony remote method invocation between objects residing on different computing nodes, respectively.

We tested the application on binary scenario trees of different sizes. The node objects are working on test problems reflecting realistic optimization problems. The speedup results for a number of SMP compute nodes (each comprising 4 CPUs) are presented in Fig. 8.

Some of our measurements not presented in this paper due to space constraints show that the effective time for computation (spent on the CPU) is small as compared to the actual execution time. The latter is strongly influenced by synchronization between nodes and idle times due to waiting criteria to be met, and by the large number of tree node objects, each having its own thread of execution, on a compute node. For the tree with 31 nodes, the SMP nodes are used inefficiently because of the small number of tree nodes on each CPU and the small computation time for each tree node. The speedup improves with the number of tree nodes, because the CPUs are used more efficiently. The best performance is achieved for 511 tree nodes. We believe that our JavaSymphony-based implementation of the nested Benders decomposition is suitable for realistic optimization problems (in terms of tree and local problem sizes).

5 Conclusions and Future Work

JavaSymphony provides a programming paradigm and software infrastructure to alleviate distributed and parallel programming that effectively exploits heterogeneous resources ranging from small-scale cluster computing to large-scale wide-area meta-computing. In contrast to most existing work, JavaSymphony allows the programmer to explicitly control locality of data and load balancing at a high level. We have implemented a runtime system that supports the full JavaSymphony programming API as outlined in this paper.

We have extensively tested the JavaSymphony programming API based on the current runtime system. The objective was to examine the suitability of JavaSymphony for a variety of distributed algorithms and applications on both homogeneous and heterogeneous cluster architectures and to evaluate the resulting performance. Each of the programs could be implemented with a reasonable effort. We compared JavaSymphony with a semi-automatic parallelization system. Based on the experiments described, we observed that for distributed algorithms that expose a simple parallelization structure, semi-automatic systems can achieve slightly better performance than JavaSymphony. However, if heteregenous architectures are provided for parallelization and more complex load balancing and control of data locality is needed, then JavaSymphony clearly outperforms the semi-automatic system. The programming effort – even if lines of codes are compared – for both JavaSymphony and the semi-automatic system was very similar for simple parallelization cases. However, in the case of more complex problems, JavaSymphony also requires more work, whereas the semi-automatic system fully relies on the capability of the runtime system. The performance achieved by JavaSymphony was notably good for most test cases examined. Overall we believe that JavaSymphony is a promising new programming paradigm for distributed and parallel Java applications that achieves good performance with a reasonable effort without dealing with low level details.

We are currently in the process to examine JavaSymphony also for applications that run on Grid infrastructures.

References

[1] 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.

[2] R. Aversa, B. D. Martino, N. Mazzocca, M. Rak, and S. Venticinque. Integration of mobile agents and openmp for programming clusters of shared memory processors: a case study. accepted for publication in proc. of PaCT 2001 Conference, 8-12 Sept. 2001, Barcelona, Spain, 2001.

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

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

[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] T. Fahringer and A. Jugravu. JavaSymphony: New Directives to Control Locality, Parallelism, and Load Balancing for Cluster and Grid-Computing. Technical Report TR2002-11, Institute for Software Science, University of Vienna, February 2002.

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

[8] H.Moritsch and G.Ch.Pflug. Java Implementation of Asynchronous Parallel Nested Optimization Algorithms. In *Proceedings of the 3rd Workshop on Java for High Performance Computing, Sorrento, Italy*, June 2001.

[9] 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.

[10] E. Laure and H. Moritsch. Portable Parallel Portfolio Optimization in the Aurora Financial Management System. In *Proceedings of SPIE ITCom 2001 Conference: Commercial Applications for High-Performance Computing*, Denver, Colorado, August 2001.

[11] 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.

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

[13] Javaparty homepage: http://wwwipd.ira.uka.de/javaparty/.

[14] G. Stefanescu. "Interactive systems": - from folklore to mathematics. In *Proc. 6th International Workshop on Relational Methods in Computer Science*, pages 208–221, Oisterwijk (near Tilburg),The Netherlands, 2001. Also Springer LNCS 2002, to appear.

[15] 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.

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