JavaSymphony: A Programming and Execution Environment for Parallel and Distributed Many-Core Architectures[⋆]

Muhammad Aleem, Radu Prodan, and Thomas Fahringer

Institute of Computer Science, University of Innsbruck,
Technikerstraße 21a, A-6020 Innsbruck, Austria
{aleem,radu,tf}@dps.uibk.ac.at

Abstract. Today, software developers face the challenge of re-engineering their applications to exploit the full power of the new emerging many-core processors. However, a uniform high-level programming model and interface for parallelising Java applications is still missing.

In this paper, we propose a new Java-based programming model for shared memory many-core parallel computers as an extension to the JavaSymphony distributed programming environment. The concept of dynamic virtual architecture allows modelling of hierarchical resource topologies ranging from individual cores and multi-core processors to more complex parallel computers and distributed Grid infrastructures. On top of this virtual architecture, objects can be explicitly distributed, migrated, and invoked, enabling high-level user control of parallelism, locality, and load balancing.

We evaluate the JavaSymphony programming model and the new shared memory run-time environment for six real applications and benchmarks on a modern multi-core parallel computer. We report scalability analysis results that demonstrate that JavaSymphony outperforms pure Java implementations, as well as other alternative related solutions.

1 Introduction

Over the last 35 years, increasing processor clock frequency was the main technique to enhance the overall processor power. Today, power consumption and heat dissipation are the two main factors which resulted in a design shift towards multi-core architectures. A multi-core processor consists of several homogeneous or heterogeneous processing cores packaged in a single chip [5] with possibly varying computing power, cache size, cache levels, and power consumption requirements.

This shift profoundly affects application developers who can no longer transparently rely only on Moore's law to speedup their applications. Rather, they have to re-engineer and parallelise their applications with user controlled load

⋆ This research is partially funded by the "Tiroler Zukunftsstiftung", Project name: "Parallel Computing with Java for Manycore Computers".

balancing and locality control to exploit the underlying many-core architectures and ever complex memory hierarchies. The locality of task and data has significant impact on application performance as demonstrated by [9,11].

The use of Java for scientific and high performance applications has significantly increased in recent years. The Java programming constructs related to threads, synchronisation, remote method invocations, and networking are well suited to exploit medium to coarse grained parallelism required by parallel and distributed applications. Terracotta [10], Proactive [1], DataRush [4], and JEOPARD [6] are some of the prominent efforts which have demonstrated the use of Java for the performance-oriented applications. Most of these efforts, however, do not provide user-controlled locality of task and data to exploit the complex memory hierarchies on many-core architectures.

In previous work, we developed JavaSymphony [2] (JS) as a Java-based programming paradigm for programming conventional parallel and distributed infrastructures such as heterogeneous clusters and computational Grids. In this paper, we extend JS with a new shared memory abstraction for programming many-core architectures. JS's design is based on the concept of dynamic virtual architecture which allows modelling of hierarchical resource topologies ranging from individual cores and multi-core processors to more complex parallel computers and distributed Grid infrastructures. On top of this virtual architecture, objects can be explicitly distributed, migrated, and invoked, enabling high-level user control of parallelism, locality, and load balancing. The extensions to the run-time environment were performed with minimal API changes, meaning that the old distributed JS applications can now transparently benefit from being executed on many-core parallel computers with improved locality.

The rest of paper is organised as follows. Next section discusses the related work. Section 3 presents the JS programming model for many-core processors, including a shared memory run-time environment, dynamic virtual architectures, and locality control mechanisms. In Section 4, we present experimental results on six real applications and benchmarks. Section 5 concludes the paper.

2 Related Work

Proactive [1] is an open source Java-based library and parallel programming environment for developing parallel, distributed and concurrent applications. Proactive provides high-level programming abstractions based on the concept of remote *active objects* [1], which return future objects after asynchronous invocations. Alongside programming, Proactive provides deployment-level abstractions for applications on clusters, Grids and multi-core machines. Proactive does not provide user-controlled locality of tasks and objects at processor or core level.

The JEOPARD [6] project's main goal is to provide a complete hardware and software framework for developing real-time Java applications on embedded systems and multi-core SMPs. The project is aiming to provide operating system support in the form of system-level libraries, hardware support in the form of Java processors, and tool support related to application development and performance analysis. Although focused on embedded multi-core systems, the API includes functionality for multi-core and NUMA parallel architectures.

Terracotta [10] is a Java-based open source framework for application development on multi-cores, clusters, Grids, and Clouds. Terracotta uses a JVM clustering technique, although the application developer sees a combined view of the JVMs. Terracotta targets enterprise and Web applications and does not provide abstractions to hide concurrency from the application developer who has to take care of these low-level details.

Pervasive DataRush [4] is a Java-based high-performance parallel solution for data-driven applications, such as data mining, text mining, and data services. A DataRush application consists of data flow graphs, which represent data dependencies among different components. The run-time system executes the data flow graphs and handles all underlying low-level details related to synchronisation and concurrency.

Most of these related works either prevent the application developer from controlling the locality of data and tasks, or engage the developer in time consuming and error-prone low-level parallelisation details of the Java language such as socket communication, synchronisation, remote method invocation, and thread management. High-level user-controlled locality at the application, object, and task level distinguishes JavaSymphony from other Java-based frameworks for performance-oriented applications.

3 JavaSymphony

JavaSymphony (JS) [2] is a Java-based programming paradigm, originally designed for developing applications on distributed cluster and Grid architectures. JS provides high-level programming constructs which abstract low-level programming details and simplify the tasks of controlling parallelism, locality, and load balancing. In this section, we present extensions to the JavaSymphony programming model to support shared memory architectures, ranging from distributed NUMA and SMP parallel computers to modern many-core processors. We provide a unified solution for user-controlled locality-aware mapping of applications, objects and tasks on shared and distributed memory infrastructures with a uniform interface that shields the programmer from the low-level resource access and communication mechanisms.

3.1 Dynamic Virtual Architectures

Most existing Java infrastructures that support performance-oriented distributed and parallel applications hide the underlying physical architecture or assume a single flat hierarchy of a set (array) of computational nodes or cores. This simplified view does not reflect heterogeneous architectures such as multi-core parallel computers or Grids. Often the programmer fully depends on the underlying operating system on shared memory machines or on local resource managers on clusters and Grids to properly distribute the data and computations which results in important performance losses.

To alleviate this problem, JS introduces the concept of *dynamic virtual architecture* (VA) that defines the structure of a heterogeneous architecture, which may vary from a small-scale multi-core processor or cluster to a large-scale Grid. The VAs are used to control mapping, load balancing, code placement and migration of objects in a parallel and distributed environment. A VA can be seen as a tree structure, where each node has a certain level that represents a specific resource granularity. Originally, JavaSymphony focused exclusively on distributed resources and had no capabilities of specifying hierarchical VAs at the level of shared memory resources (i.e. scheduling of threads on shared memory resources was simply delegated to the operating system).

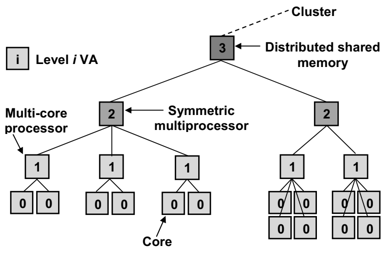


Fig. 1. Four-level locality-aware VA

In this paper, we extended the JavaSymphony VA to reflect the structure of shared memory many-core computing resources. For example, Figure 1 depicts a four-level VA representing a distributed memory machine (such as a shared memory NUMA) consisting of a set of SMPs on level 2, multi-core processors on level 1, and individual cores on the leaf nodes (level 0). Lines $6-8$ in Listing 1 illustrates the JS statements for creating this VA structure.

3.2 JavaSymphony Objects

Writing parallel JavaSymphony applications requires encapsulating Java objects into so called *JS objects*, which are distributed and mapped onto the hierarchical VA nodes (levels 0 to n). A JS object can be either a single-threaded or a multi-threaded object. The single-threaded JS object is associated with one thread which executes all invoked methods of that object. A multi-threaded JS object is associated with n parallel threads, all invoking methods of that object.

In this paper, we extend JS with a shared memory programming model based on shared JS (SJS) objects. A SJS object can be mapped to a node of level $0-3$ according to the VA depicted in Figure 1 (see Listing 1, lines 7 and 12) and cannot be distributed onto higher-level remote VA nodes. A SJS object can also be a single-threaded or a multi-threaded object. Listing 1 (line 12) shows the JS code for creating a multi-threaded SJS object. Three types of method invocations can be used with JS as well as with SJS objects: asynchronous (see Listing 1, line 14), synchronous, and one-sided method invocations.

3.3 Object Agent System

The Object Agent System, a part of JS run-time system (JSR), manages and processes shared memory jobs for SJS objects and remote memory jobs for JS objects. Figure 2 shows two Object Agents (OA). An OA is responsible for creating jobs, mapping objects to VAs, migrating, and releasing objects. The shared memory jobs are processed by a local OA, while the remote memory jobs

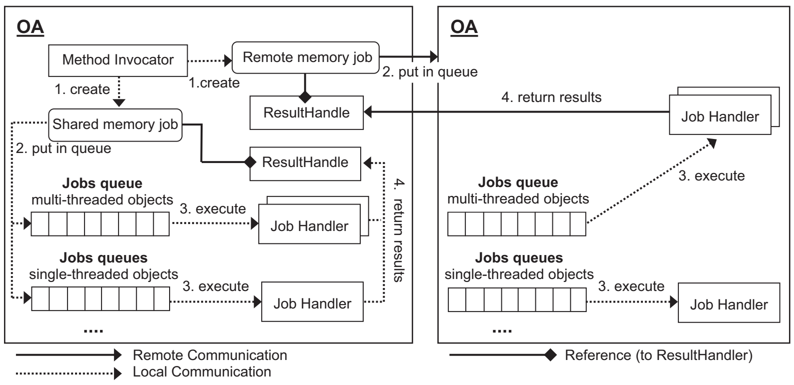


Fig. 2. The Object Agent System job processing mechanism

are distributed and processed by remote OAs. An OA has a multi-threaded job queue which contains all jobs related to the multi-threaded JS or SJS objects. A multi-threaded job queue is associated with n job processing threads called *Job Handlers*. Each single-threaded JS or SJS object has a single-threaded job queue and an associated job handler. The results returned by the shared and remote memory jobs are accessed using `ResultHandle` objects (listing 1, line 5), which can be either local object references in case of SJS objects, or remote object references in case of distributed JS objects.

3.4 Synchronisation Mechanism

JS provides synchronisation mechanism for one-sided and asynchronous object method invocations. Asynchronous invocations can be synchronised at individual or at group level. Group-level synchronisation involves n asynchronous invocations combined in one `ResultHandleSet` object (see Listing 1, lines $13-14$). The group-level synchronisation may block or examine (without blocking) whether one, a certain number, or all threads have finished processing their methods. The JS objects invoked using one-sided method invocations may use barrier synchronisation implemented using barrier objects. A barrier object has a unique identifier and an integer value specifying the number of threads that will wait on that barrier.

3.5 Locality Control

The Locality Control Module (LCM) is a part of the JSR that applies and manages locality constraints on the executing JS application by mapping JS objects and tasks onto the VA nodes. In JS, We can specify locality constraints at three levels of abstraction:

1. *Application-level* locality constraints are applied to all JS or SJS objects and all future data allocations of a JS application. The locality constraints can be

specified with the help of `setAppAffinity` static method of the `JSRegistry` class, as shown in line 9 of Listing 1.

2. *Object-level* locality constraints are applied to all method invocations and data allocations performed by a JS or SJS object (see line 12 in Listing 1). The object-level locality constraints override any previous application-level constraints for that object.
3. *Task-level* locality constraints are applied to specific task invocations and override any previous object or application-level constraints for that task.

Mapping an application, object, or task to a specific core will constrain the execution to that core. Mapping them on a higher-level VA node will delegate the scheduling on the inferior VA nodes to the JSR.

The LCM in coordination with the OAS job processing mechanism applies the locality constraints. The jobs are processed by the job handler threads within an OA. The job handlers are JVM threads that are executed by some system-level threads such as the POSIX threads on a Linux system. The locality constraints are enforced within the job handler thread by invoking appropriate POSIX system calls through the JNI mechanism. First, a unique system-wide thread identifier is obtained for the job handler thread with help of the `syscall(__NR_gettid)` system call. This unique thread identifier is then used as input to the `sched_setaffinity` function call to schedule the thread on a certain core of the level 1 VA.

3.6 Matrix Transposition Example

Listing 1 displays the kernel of a simple shared memory matrix transposition application, a very simple but much used kernel in many numerical applications, which interchanges a matrix rows and columns: $A[i, j] = A[j, i]$.

The application first initialises the matrix and some variables (lines $1 - 3$) and registers itself to the JSR (line 4). Then, it creates a group-level `ResultHandle` object `rhs` and a level 3 VA node `dsm` (lines $5 - 8$). Then the application specifies

```
 1  boolean bSingleThreaded = false; int N = 1024; int np = 4;
 2  int[] startRow = new int[np]; int[][] T = new int[N][N];
 3  int[][] A=new int[N][N]; initializeMatrix(A);
 4  JSRegistry reg = new JSRegistry("MatrixTransposeApp");
 5  ResultHandleSet rhs;
 6  VA smp1 = new VA(2, new int[]{2,2,2});
 7  VA smp2 = new VA(2, new int[]{4,4});
 8  VA dsm = new VA(3); dsm.addVA(smp1); dsm.addVA(smp2);
 9  JSRegistry.setAppAffinity(dsm); // Application-level locality
10  for(int i = 0; i < N; i = i + N / np)
11    startRow[i] = i;
12  SJSObject worker = new SJSObject(bSingleThreaded, "workspace.Worker
        ", new Object[]{A,T,N}, smp2); // Object-level locality
13  for(int i = 0; i < np; i++)
14    rhs.add(worker.ainvoke("Transpose", new Object[]{i,startRow[i]}),
          i);
15  rhs.waitAll();
16  reg.unregister();
```

Listing 1. Matrix transposition example

the application level locality (line 9) corresponding to a distributed shared memory NUMA parallel computer. The application then partitions the matrix blockwise among `np` parallel tasks (lines 10 – 11). In line 12, a multi-threaded SJS object is created and mapped onto the level 2 VA node `smp2`. The application then invokes `np` asynchronous methods, adds the returned `ResultHandle` objects to `rhs` (lines 13 – 14), and waits for all invoked methods to finish their execution (line 15). In the end, the application unregisters itself from the JSR (line 16).

4 Experiments

We have developed several applications and benchmarks using the JS shared memory programming model with locality awareness. We used for our experiments a shared memory SunFire X4600 M2 NUMA machine equipped with eight quad-core processors, where each processor has a local memory bank. Each core has a private L1 and L2 caches of size 128KB and 512KB, respectively, and a shared 2MB L3 cache. Each processor has three cache coherent hypertransport links supporting up to 8GB/sec of direct and inter-processor data transfer each.

4.1 Discrete Cosine Transformation

The Discrete Cosine Transformation (DCT) algorithm is used to remove the non-essential information from digital images and audio data. Typically, it is used to compress the JPEG images. The DCT algorithm divides the image into square blocks and then applies the transformations on each block to remove the non-essential information. After that, a reverse transformation is applied to produce a restored image, which contains only essential data.

We developed a shared memory version of JS DCT and compared it with a previous message passing-based implementation [2] in order to test the improvement of the shared memory solution against the old implementation. Figure 3 shows that the shared memory implementation requires approximately 20% to 50% less execution time as compared to the RMI-based distributed implementation. The results validate the scalability of the JS shared memory programming model and also highlight its importance on a multi-core system as compared to the message passing-based model which introduces costly communication overheads.

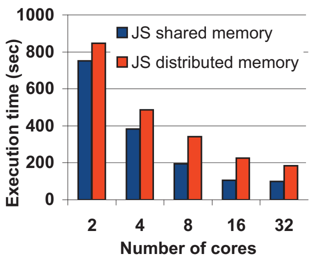


Fig. 3. DCT experimental results

4.2 NAS Parallel Benchmarks: CG Kernel

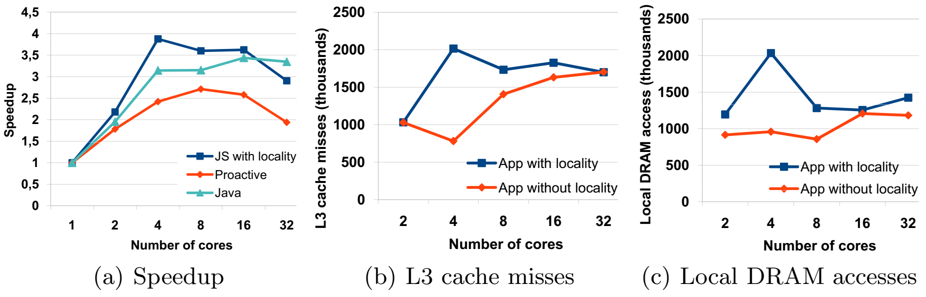
The NAS parallel benchmarks [3] consist of five kernels and three simulated applications. We used for our experiments the Conjugate Gradient (CG) kernel which involves irregular long distance communications. The CG kernel uses the power and conjugate gradient method to compute an approximation to the smallest eigenvalue of large sparse, symmetric positive definite matrices.

We implemented the kernel in JS using the Java-based implementation of CG [3]. The JS CG implementation is based on the master-worker computational model. The main JS program acts as master and creates multiple SJS worker objects that implement four different methods. The master program asynchronously invokes these methods on the SJS worker objects and waits for them to finish their execution. A single iteration involves several serial computations from the master and many steps of parallel computations from the workers. In the end, the correctness of results is checked using a validation method.

We compared our locality-aware JS version of the CG benchmark with a pure Java and a Proactive-based implementation. Figure 4(a) shows that JS exhibits better speedup compared to the Java and Proactive-based implementations for almost all machine sizes. Proactive exhibits the worse performance since the communication among threads is limited to RMI, even within shared memory computers. The speedup of the JS implementation for 32 cores is lower than the Java-based version because of the overhead induced by the locality control, which becomes significant for the relatively small problem size used. In general, specifying locality constraints for n parallel tasks on a n-core machine does not bring much benefit, still it is useful to solve the problems related to thread migration and execution of multiple JVM-level threads by a single native thread.

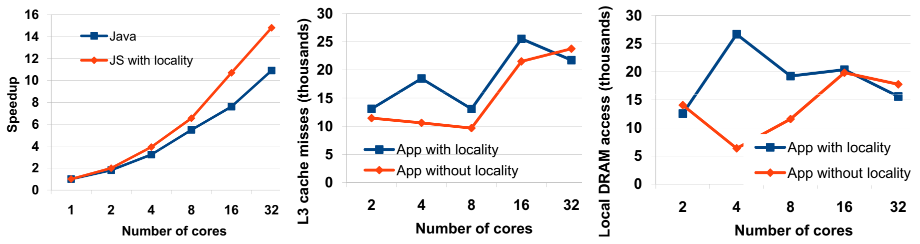
To investigate the effect of locality constraints, we measured the number of cache misses and local DRAM accesses for the locality aware and non-locality aware applications. Figure 4(b) illustrates that the number of L3 cache misses increased for the locality-aware JS implementation because of the contention on the L3 cache shared by multiple threads scheduled on the same multi-core processor. The locality constraints keep the threads close to the node where the data was allocated, which results in a high number of local memory accesses that significantly boost the overall performance (see Figure 4(c)).

Although the JS CG kernel achieves better speedup as compared to the all other versions, the overall speedup is not impressive because it involves a large number of parallel invocations (11550 – 184800 for 2 – 32 cores) to complete



(a) Speedup (b) L3 cache misses (c) Local DRAM accesses

Fig. 4. CG kernel experimental results



(a) Speedup (b) L3 cache misses (c) Local DRAM accesses

Fig. 5. Ray tracing experimental results

75 iterations. It is is a communication intensive kernel and the memory access latencies play a major role in the application performance. The non-contiguous data accesses also result in more cache misses. This kernel achieves good speedup until 4 cores (all local memory accesses), while beyond 8 cores the speedup decrease, because of increased memory access latencies on the threads scheduled on the remote cores.

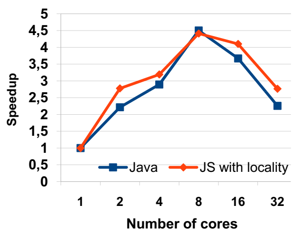
4.3 3D Ray Tracing

The ray tracing application, part of the Java Grande Forum (JGF) benchmark suite [8], renders a scene containing 64 spheres at $N \times N$ resolution. We implemented this application in JS using the multi-threaded Java version from the JGF benchmarks. The ray tracing application creates first several ray tracer objects, initialises them with scene and interval data, and then renders to the specified resolution. The interval data points to the rows of pixels a parallel thread will render. The JS implementation parallelises the code by distributing the outermost loop (over rows of pixels) to several SJS objects. The SJS objects render the corresponding rows of pixels and write back the resulting image data. We applied locality constraints by mapping objects to cores close to each other to minimise the memory access latencies.

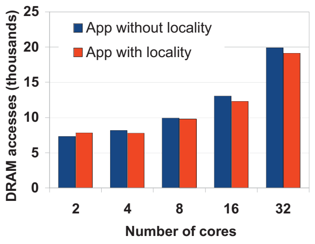
We experimentally compared our JS implementation with the multi-threaded Java ray tracer from JGF. As shown in Figure 5(a), the JS implementation achieves better speedup for all machine sizes. Figure 5(c) shows that there is a higher number of local memory accesses for the locality-aware JS implementation, which is the main reason for the performance improvement. Figure 5(b) further shows that a locality-aware version has high number of L3 cache misses because of the resource contention on this shared resource, however, the performance penalty is significantly lower compared to the locality gain.

4.4 Cholesky Factorisation

The Cholesky factorisation [7] expresses a $N \times N$ symmetric positive-definite matrix A, implying that all the diagonal elements are positive and the non-diagonal elements are not too big, as the product of a triangular matrix L and



(a) Speedup



(b) Total DRAM accesses

Fig. 6. Cholesky factorisation experimental results

its transpose L^T: $A = L \times L^T$. This numerical method is generally used to calculate the inverse and the determinant of a positive definite matrix.

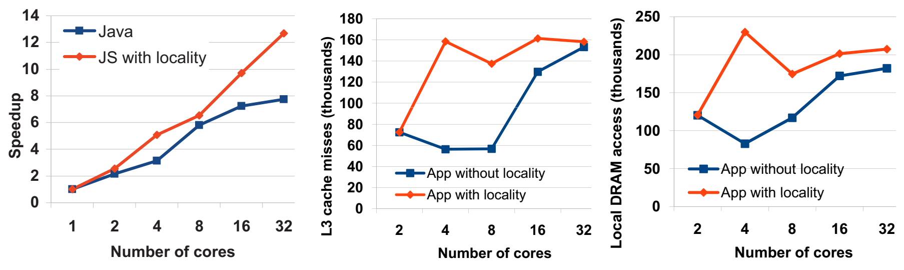
We developed a multi-threaded Java-based version and a JS-based implementation with locality constraints of the algorithm. The triangular matrix L is parallelised by distributing a single row of values among several parallel tasks; in this way, rows are computed one after another. Figure 6(a) shows that the locality-aware JS version has better speedup compared to the Java-based version, however, for the machine size 8 both versions show quite similar performance. We observed that the operating system scheduled by default the threads of the Java version one router away from the data (using both right and left neighbour nodes), which matched the locality constraints we applied to the JS version. Figure 6(b) shows that the locality-based version has a low number of memory accesses compared to the non-locality-based version due to the spatial locality.

4.5 Matrix Transposition

We developed a multi-threaded Java-based and a JS-based locality-aware version of the matrix transposition algorithm that we introduced in Section 3.6. Again, the locality-aware JS version achieved better speedup, as illustrated in Figure 7(a). The locality constraints mapped the threads to all cores of a processor before moving to other node, which resulted in L3 cache contention and, therefore, more cache misses (see Figure 7(b)). The locality also ensures that there are more local and less costly remote memory accesses which produces better application performance (see Figure 7(c)).

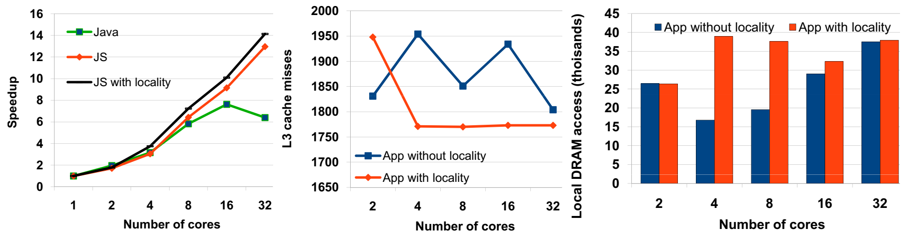
4.6 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication (SpMV) is an important kernel used in many scientific applications which computes $y = A \cdot x$, where A is a sparse matrix and x and y are dense vectors. We developed an iterative version of the SpMV kernel where matrix A was stored in a vector using the Compressed Row Storage format. We used for this experiment a matrix size of 10000×10000 and set the number of non-zero elements per row to 1000. We developed both



(a) Speedup (b) L3 cache misses (c) Local DRAM accesses

Fig. 7. Matrix transposition experimental results



(a) Speedup (b) L3 cache misses (c) Local DRAM accesses

Fig. 8. SpMV experimental results

pure Java and JS versions of this kernel by distributing the rows of the sparse matrix to several parallel threads. The resulting vector y is computed as follows: $y_i = \sum_{j=1}^{n} a_{ij} \cdot x_j$.

SpMV involves indirect and unstructured memory accesses which negatively affects the pure Java implementation with no locality constraints (see Figure 8(a)), while the locality-aware JS implementation performs significantly better. The vector x is used by all threads and, once the data values from this vector are loaded in L3 shared cache, they are reused by the other working threads on that processor. This spatial locality results in less cache misses as shown in Figure 8(b). The locality also ensures less remote and more local memory accesses as shown in Figure 8(c), which further improves the application performance.

5 Conclusions

We presented JavaSymphony, a parallel and distributed programming and execution environment for multi-cores architectures. JS's design is based on the concept of dynamic virtual architecture, which allows modelling of hierarchical resource topologies ranging from individual cores and multi-core processors to more complex symmetric multiprocessors and distributed memory parallel computers. On top of this virtual architecture, objects can be explicitly distributed, migrated, and invoked, enabling high-level user control of parallelism, locality, and load balancing. Additionally, JS provides high-level abstractions to control locality at application, object and thread level. We illustrated a number of real application and benchmark experiments showing that the locality-aware JS implementation outperforms conventional technologies such as pure Java relying entirely on operating system thread scheduling and data allocation.

Acknowledgements

The authors thank Hans Moritsch for his contribution to earlier stages of this research.

References

1. Denis Caromel, M.L.: ProActive Parallel Suite: From Active Objects-Skeletons-Components to Environment and Deployment. In: Euro-Par 2008 Workshops - Parallel Processing, pp. 423–437. Springer, Heidelberg (2008)
2. Fahringer, T., Jugravu, A.: JavaSymphony: a new programming paradigm to control and synchronize locality, parallelism and load balancing for parallel and distributed computing: Research articles. Concurr. Comput. Pract. Exper. 17(7-8), 1005–1025 (2005)
3. Frumkin, M.A., Schultz, M., Jin, H., Yan, J.: Performance and scalability of the NAS parallel benchmarks in Java. IPDPS, 139a (2003)
4. Norfolk, D.: The growth in data volumes - an opportunity for it-based analytics with Pervasive Datarush. White Paper (June 2009), `http://www.pervasivedatarush.com/Documents/WP%27s/Norfolk%20WP%20-%20DataRush%20%282%29.pdf`
5. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In: MICRO-36'-03 (2003)
6. Siebert, F.: Jeopard: Java environment for parallel real-time development. In: JTRES-06 2008, pp. 87–93. ACM, New York (2008)
7. Siegfried, B., Maria, L., Kvasnicka, D.: Experiments with cholesky factorization on clusters of smps. In: Proceedings of the Book of Abstracts and Conference CD, NMCM 2002, pp. 30–31, 1–14 (July 2002)
8. Smith, L.A., Bull, J.M.: A multithreaded Java grande benchmark suite. In: Third Workshop on Java for High Performance Computing (June 2001)
9. Song, F., Moore, S., Dongarra, J.: Feedback-directed thread scheduling with memory considerations. In: The 16th international symposium on High performance distributed computing, pp. 97–106. ACM, New York (2007)
10. Terracotta, I.: The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability. Apress, Berkely (2008)
11. Yang, R., Antony, J., Janes, P.P., Rendell, A.P.: Memory and thread placement effects as a function of cache usage: A study of the gaussian chemistry code on the sunfire x4600 m2. In: International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 31–36 (2008)