On the Evaluation of JavaSymphony for Heterogeneous Multi-core Clusters*

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. Programming hybrid heterogeneous multi-core cluster architectures is today an important topic in scientific and mainstream communities. To address this challenge, we developed JavaSymphony providing high-level programming abstraction and a middle-ware that facilitates the development and high-performance execution of Java applications on modern shared and distributed memory architectures. In this paper we present results of programming and executing a three-dimensional ray tracing application on a heterogeneous many-core cluster architecture.

1 Introduction

Multi-core processors have emerged today as a viable source of processing power. The emergence of multi-core trend was the result of heat dissipation and power consumption problems related to high clocked single-core processors. A multi-core processor consists of several homogeneous or heterogeneous cores packaged in a single chip. Already, there are many-core processors with hundreds of cores and the majority of top 500 supercomputers is being based on multi-core cluster architectures.

To exploit the underlying many-cores, applications need to be re-engineered and parallelised with user controlled load balancing and locality, heterogeneity of machines, and complex memory hierarchies. The Java programming constructs related to threads, synchronisation, remote method invocations, and networking are well-suited to exploit medium to coarse grained parallelism. Today, there are many research efforts [2,5,6,9,10] which focus on parallel Java applications for multi-core shared memory systems and clusters. Most of these efforts, however, do not provide user-controlled locality of task and data to exploit the complex memory hierarchies on many-core clusters. The locality of task and data have significant impact on an application's performance as demonstrated in [3,8].

In previous work [1,4], we developed JavaSymphony (JS) as a Java-based programming paradigm for programming conventional parallel and distributed infrastructures such as heterogeneous clusters, computational Grids, and shared memory multi-cores. JS provides a unified API to program both shared, as well as

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

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 23–30, 2011.
© Springer-Verlag Berlin Heidelberg 2011

distributed memory applications. JS's design is based on the concept of dynamic virtual architecture, which allows programmer to define a hierarchical structure of heterogeneous computing resources (e.g. cores, processors, machines, clusters) and to control load balancing, locality, and code placements. On top of the virtual architecture, objects can be explicitly distributed, migrated, and invoked, enabling high-level user control of parallelism, locality, and load balancing. Previously [1], we described JS run-time system and locality control mechanism for shared and distributed memory applications. In this paper, we present new experiments based on a 3D ray tracing application using a heterogeneous multi-core cluster architecture.

The paper is organised as follows. Next section discusses the related work. Section 3 presents the JS overview, including JS run-time system, dynamic virtual architectures, and locality control mechanisms. Section 4 presents experimental results and section 5 concludes the paper.

2 Related Work

Proactive [2] is a Java-based library and parallel programming environment for parallel and distributed applications. Proactive provides high-level programming abstractions based on the concept of remote *active objects* [2]. In contrast to Proactive's single-threaded *active objects*, JS provides multi-threaded remote objects. Alongside programming, Proactive also provides deployment-level abstractions. Proactive has no functionality to map an *active object* and thread to specific processors or cores of a multi-core cluster.

Jcluster [9] is a Java-based message passing library for programing parallel applications. Jcluster provides a dynamic load balancing scheduler that is based on the Transitive Random Stealing algorithm. The dynamic scheduler enables any node in a cluster to get a task from other used nodes to balance the load. Jcluster scheduler has no support for multi-core processors and no functionality to map a task or object to specific processor and core in a multi-core cluster environment.

Parallel Java [5] is a Java-based API for shared and distributed memory parallel applications. It provides programming constructs similar to MPI and OpenMP. A hybrid programming model can also be used to combine both shared and distributed memory features in a parallel program. Although Parallel Java provides a multi-threaded approach for shared memory multi-cores, it has no capability to map threads to specific processors and cores in a cluster.

VCluster [10] implements a new programming model which allows migration of virtual threads (instead of complete processes) to other JVMs on the same or on different multi-core nodes in a cluster. Thread migration can be used for dynamic load balancing in a parallel multi-threaded application. VCluster does not provide any functionality to map a thread to a specific processor or core of a multi-core cluster.

MPJ Express [6] is a Java-based message passing framework that provides a portable and efficient communication layer. Although MPJ Express uses shared

memory communication inside a multi-core computing node, it has no capability to control the locality of threads at processor or core level.

Most of the related work either prevents the application developer from controlling the locality of data and tasks, or engage the developer in time consuming and error-prone low-level parallelization details of the Java language. High-level user-controlled locality of the application, object, and task distinguishes JavaSymphony from other Java-based frameworks for multi-core cluster programming.

3 JavaSymphony

JavaSymphony (JS) is a Java-based programming paradigm for developing parallel and distributed applications. JS provides high-level programming constructs which abstract low-level infrastructure details and simplify the tasks of controlling parallelism, locality, and load balancing. Furthermore, it offers a unified solution for user-controlled locality-aware mapping of applications, objects and tasks on shared and distributed memory infrastructures. In this section, we provide an overview of some of the JS features, while complete description and implementation details can be found in [1,4].

3.1 Dynamic Virtual Architectures

The Dynamic Virtual Architecture (VA) [4] concept introduced by JS allows the programmer to define structure of heterogeneous computing resources and to control mapping, load balancing, migration of objects, and code placements. Most existing work assumes flat hierarchy of computing resources. In contrast to that, JS allows programmer to fully specify the multi-core architectures [1].

VA has a tree like structure, where each VA element has a certain level representing a specific resource granularity. Figure 1 depicts a four-level VA representing a heterogeneous cluster architecture consisting of a set of shared memory (NUMA or UMA) nodes on level 2, multi-core processors on level 1, and individual cores on the leaf nodes (level 0).

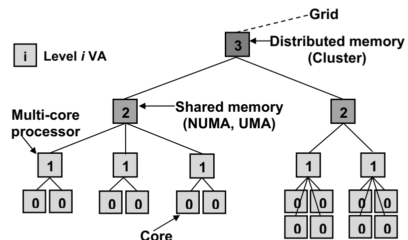


Fig. 1. Four-level locality-aware VA

3.2 JavaSymphony Objects

Writing a parallel JavaSymphony application requires encapsulating Java objects into so called *JS objects*, which are then distributed and mapped onto the hierarchical VA nodes (levels 0 to n). A JS object can be either a single or a multi-threaded object supporting three types of method invocations: asynchronous, synchronous, and one-sided.

3.3 Object Agent System

The Object Agent (OA) System [1], a part of JS run-time (JSR), processes remote as well as shared memory jobs. An OA is responsible for creating jobs, mapping objects to VAs, migrating, and releasing objects. An OA has a multi-threaded job queue which is associated with n job processing threads called *Job Handlers*. The results returned by the jobs are accessed using `ResultHandle` objects.

3.4 Locality Control

The Locality Control Module [1] applies and manages locality on the executing JS application by mapping the JS objects and tasks onto the VA nodes. In JS, we can specify locality constraints at three levels of abstraction: application, object, and task-level. 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 (e.g. multi-core processor, SMP, NUMA, cluster) will constrain the execution on the corresponding resource and delegate the scheduling to the inferior VA nodes.

4 Experiments

We developed a JS-based version of a multi-threaded 3D ray tracing application (JSRT) that is part of the Java Grande Forum (JGF) benchmark suite [7]. JSRT is a large-scale application that creates several ray tracer objects, initialises them with scene (64 spheres) and interval data, and renders at $N \times N$ resolution. The JSRT application is parallelised by distributing the outermost loop (over rows of pixels) to n JS objects which are mapped to the cores of the parallel machine. We experimented on a heterogeneous cluster (HC), which consists of two types of nodes outlined in Table 1:

Listing 1 shows the core of the JSRT application. First, it creates and initialises the required data structures (lines $1-3$) and then registers itself to the JS run-time system (line 4). Then, it creates a level-3 (cluster) and several level-2 (NUMA, UMA) VA nodes (lines $5-6$). The level-2 VA nodes are then initialised and added to the level-3 VA node (lines $7-9$). Then, several ray tracer objects are created (line 11), initialised, and mapped to the VA nodes (lines $12-13$). Afterwards, the rendering method (`render`) is asynchronously invoked on each `rayTracer` object and the handle objects (`ResultHandle`) returned are saved (lines $14-16$). After collecting the results (checksum values)

Table 1. The heterogeneous cluster architecture

Node architecture	*No. of nodes*	*Processor*	*Processors per node*	*Network*	*Shared caches*
NUMA	2	Quad-core Opteron 8356	8	Gigabit ethernet	L3/Processor
UMA	13	Dual-core Opteron 885	4	Gigabit ethernet	Nil

from all invoked JS objects, they are validated (line 18) to check the correctness of the algorithm. Then, the rendered images are collected (lines $19-21$) from all `rayTracers` and merged into one file. Finally, the JSRT application un-registers from the JS run-time system (line 23).

```
 1 boolean bSingleThreaded = false; int npPerMachine = 8; int np = 64;
 2 long checksum=0; int nMachines = np/npPerMachine; int k=0; int i, j;
 3 ResultHandle[] rhSet = new ResultHandle[np];
 4 JSRegistry reg = new JSRegistry("JSRTApp"); //register JSRTApp to JSR
 5 VA cluster = new VA(3,nMachines); //level-3 VA node
 6 VA[] computeNodes = new VA[nMachines];
 7 for(i=0; i<nMachines; i++) {
 8  computeNodes[i] = new VA(2); //level-2 VA nodes
 9  cluster.addVA(computeNodes[i]); } //add level-2 VA nodes to level-3
10 ... //Initialization of data structures
11 JSObject[] rayTracers = new JSObject[nMachines]; //distributed
      objects
12 for(i=0; i<nMachines; i++) //create raytracers at level-2 VA nodes
13  rayTracers[i] = new JSObject(bSingleThreaded,"jsRayTracer.Worker",
      new Object[]{width,height,np},computeNodes[i]);
14 for(i=0; i<nMachines; i++)
15  for(j=0; j<npPerMachine; j++, k++) //invoke render tasks
16   rhSet[k] = rayTracers[i].ainvoke("render",new Object[]{k});
17 ... //get and sum the checksum values
18 jsRayTracerValidate(checksum); //check for correctness
19 for(i=0; i<nMachines; i++) { //get and save rendered images
20  rhSet[i] = rayTracers[i].sinvoke("getImage",new Object[]{});
21  renderdImage[i] = (int[]) rhSet[i].getResult(); }
22 ... //merge and save Image data to file
23 reg.unregister(); //un-register from JSR
```

Listing 1. The core JS code of the JSRT application

4.1 Heterogeneous Cluster

On the heterogeneous cluster, we experimented using up to 128 cores and five different versions of the JSRT application. The default version labelled `JSRT` is based on a *machine-fill* scheduling strategy, in which we first entirely filled the NUMA-based nodes by invoking up to 64 parallel tasks before moving to the UMA-based nodes (8 tasks per node). We select NUMA nodes first, since they have four times as many cores as the UMA nodes. This scheme requires less number of nodes, thus results in low VA and communication overheads.

Figure 2(a) shows that, although the default version (`JSRT`) achieved decent speedup, the machine-fill scheduling negatively affected the application performance on the NUMA nodes. In particular, we observed that 22% of the threads were slower, as illustrated by the load imbalance metric in Figure 2(b), calculated as follows:

$$LI = \frac{T_{\max} - T_{avg}}{T_{\max}} \cdot 100,$$

where $T_{\max}$ and T_{avg} represent the maximum and the average times of the parallel threads. To eliminate the load imbalance, we applied first two optimisations labelled `JSRT+OPT1` and `JSRT+OPT2` that shifts 10%, respectively 20% of the threads from the NUMA nodes to other free nodes of the cluster. These versions achieved better speedup results (see Figure 2(a)) and reduced the load imbalance to about $9\% - 11\%$, respectively $2.81\% - 4.83\%$, as displayed in Figure 2(b).

In the next step, we applied locality constraints on the optimised versions (labelled `JSRT+OPT1+LOC` and `JSRT+OPT2+LOC`) and achieved up to 50.14% more speedup over the default version (see Figure 2(a)).

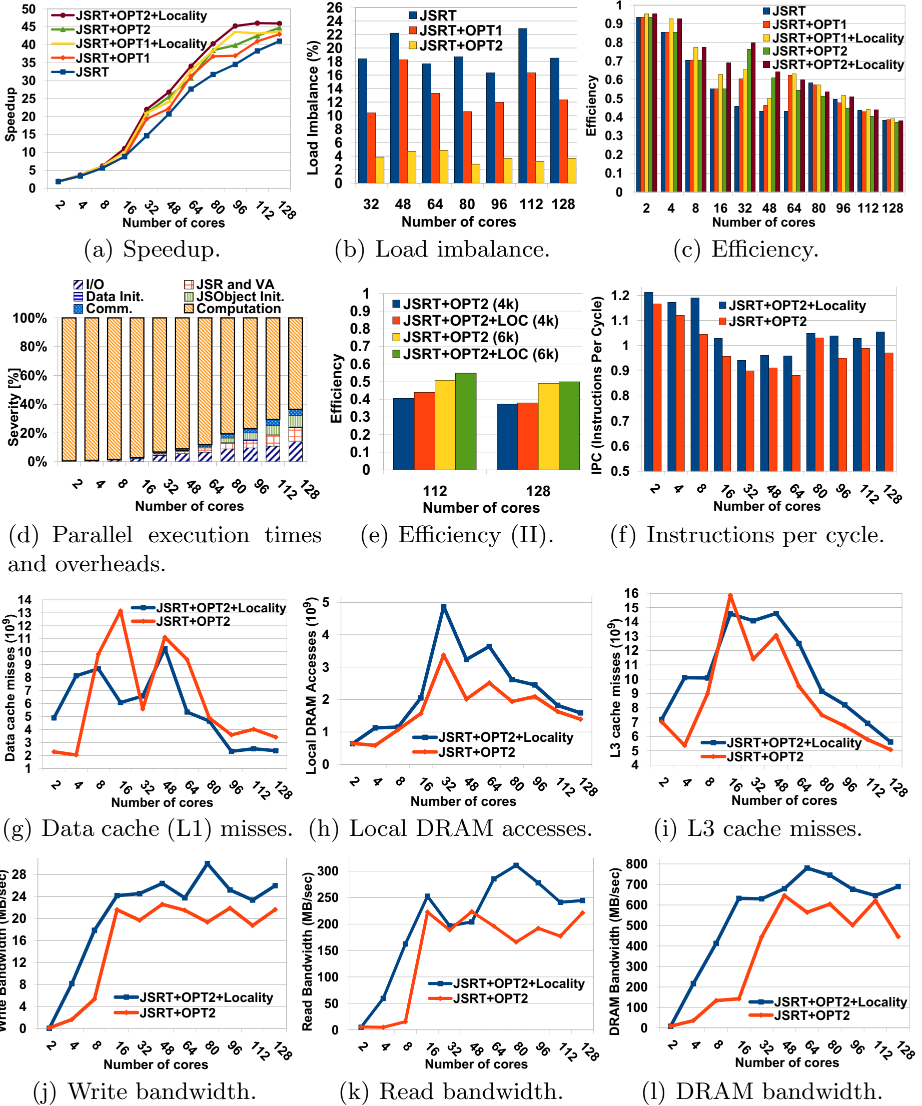
Figure 2(c) shows the efficiency of these experiments calculated as the ratio between the speedup S and the weighted processor count due to the slight difference in processor speed of the two clusters:

$$E = \frac{S}{\sum_{\forall C \in HC} \frac{T_{\min}}{T_C}},$$

where T_C is the sequential execution time of the JSRT application on core C and $T_{\min}$ is the sequential execution time on the fastest core: $T_{\min} = \min_{\forall C \in HC} \{T_C\}$. The efficiency achieved by the different `JSRT` versions is quite good ($95\% - 52\%$), although it dropped down to 45% and 39% for the 112 and 128 machine sizes. To understand this reduced efficiency, we measured the overheads T_O encountered in each execution and calculated their *severity* as the ratio to the total parallel execution time T: $S = \frac{T_O}{T}$ (see Figure 2(d)). For the large machines size ($112 - 128$ cores), we observed increasing overhead severities related to the JSR and VA creation ($7.53\% - 9.53\%$), instantiation of JS objects on remote nodes ($6.67\% - 7.82\%$), communication ($4.10\% - 4.55\%$), and I/O ($10.86\% - 14.26\%$) limited the application performance and caused the efficiency to decrease below 50%.

The efficiency can be improved by choosing larger problem sizes. For example, Figure 2(e) illustrates that the larger 6000×6000 problem size labelled `JSRT+OPT2 (6k)` achieves up to 38.82% increase in efficiency compared to the 4000×4000 problem size labelled `JSRT+OPT2 (4k)`.

To investigate the effects of the locality constraints, we measured the number of instructions per cycle that are up to 13.46% higher for the locality-aware implementation compared to the non-locality aware version (see Figure 2(f)). The locality constraints keep the threads close to the node and processor where the data has been allocated, which results in a high number of local DRAM memory accesses that significantly improve overall performance (see Figure 2(h)). We also observed less number of data cache (L1) misses for the locality-aware `JSRT` version compared to the non-locality-aware version (see Figure 2(g)). Figure 2(i) illustrates that the number of L3 cache misses has increased for the locality-aware version because of contention on the L3 cache shared by multiple threads on the NUMA nodes.



(a) Speedup. (b) Load imbalance. (c) Efficiency.

(d) Parallel execution times and overheads. (e) Efficiency (II). (f) Instructions per cycle.

(g) Data cache (L1) misses. (h) Local DRAM accesses. (i) L3 cache misses.

(j) Write bandwidth. (k) Read bandwidth. (l) DRAM bandwidth.

Fig. 2. Heterogeneous cluster experimental results

We further investigated the performance results by measuring system read, write, and DRAM bandwidth utilisation in the locality-aware and non-locality-aware versions. Figure 2(j) shows that the locality-aware version has a higher write bandwidth of up to 10.51% to 55.05% for large machine sizes (32 – 128 cores). The locality-aware version also shows 4.27% to 87.47% higher system read bandwidth for large machine sizes between 32 – 128 cores (see Figure 2(k)) and the DRAM bandwidth utilisation shown in Figure 2(l) is similarly between 3.93 – 54.78% higher.

5 Conclusions

In this paper, we presented JavaSymphony, a parallel and distributed programming and execution environment for multi-core cluster 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 processors to more complex symmetric multiprocessors and distributed memory parallel computers. JS allows user controlled locality control and load balance of applications, objects, and tasks.

We presented the JS implementation of a 3D ray tracing application followed by experimental results on a heterogeneous cluster architecture. Our improved locality-aware and optimised implementation improved the speedup of the application up to 50.14% on the heterogeneous cluster. We also conducted and presented a low-level analysis, which highlighted the reasons of better speedup achieved by the locality-aware JS implementation.

References

1. Aleem, M., Prodan, R., Fahringer, T.: JavaSymphony: A programming and execution environment for parallel and distributed many-core architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 139–150. Springer, Heidelberg (2010)
2. Caromel, D., Leyton, M.: Proactive parallel suite: From active objects-skeletons-components to environment and deployment. In: César, E., et al. (eds.) Euro-Par Workshops. LNCS, vol. 5415, pp. 423–437. Springer, Heidelberg (2008)
3. Chai, L., Gao, Q., Panda, D.K.: Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In: IEEE International Symposium on Cluster Computing and the Grid, vol. 0, pp. 471–478 (2007)
4. 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)
5. Kaminsky, A.: Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In: 21st IEEE International Parallel and Distributed Processing Symposium, pp. 1–8. IEEE Computer Society, Los Alamitos (2007)
6. Shafi, A., Manzoor, J.: Towards efficient shared memory communications in MPJ express. In: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–7. IEEE Computer Society, Los Alamitos (2009)
7. Smith, L.A., Bull, J.M.: A multithreaded Java grande benchmark suite. In: Third Workshop on Java for High Performance Computing. pp. 97–105 (2001)
8. Yang, R., Antony, J., Rendell, A.P.: A simple performance model for multithreaded applications executing on non-uniform memory access computers. In: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications, pp. 79–86. IEEE Computer Society, Los Alamitos (2009)
9. Zhang, B.Y., Yang, G.W., Zheng, W.M.: Jcluster: an efficient Java parallel environment on a large-scale heterogeneous cluster: Research articles. Concurr. Comput.: Pract. Exper. 18(12), 1541–1557 (2006)
10. Zhang, H., Lee, J., Guha, R.K.: VCluster: a thread-based Java middleware for smp and heterogeneous clusters with thread migration support. Softw., Pract. Exper. 38(10), 1049–1071 (2008)