Scheduling JavaSymphony Applications on Many-Core Parallel Computers*

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. JavaSymphony is a Java-based programming and execution environment for programming and scheduling the performance oriented applications on multi-core parallel computers. In this paper, we present a multi-core aware scheduling extension to JavaSymphony capable of mapping parallel applications on large multi-core machines and heterogeneous clusters. JavaSymphony scheduler considers several multi-core specific performance parameters and application types, and uses these parameters to optimise the mapping of JavaSymphony objects and tasks. We evaluate the performance of JavaSymphony scheduler using several real scientific applications and benchmarks on a multi-core shared memory machine and a heterogeneous cluster.

1 Introduction

Multi-core processors [2] add an additional level of parallelism to the existing parallel computers and scheduling parallel applications becomes even more challenging. To speedup applications, a scheduler is required to consider the application properties (e.g., communication and computation needs) and architectural characteristics (e.g., network and memory latencies, heterogeneity of machines, memory hierarchies, machine load). Today, there are many research efforts [3,4,6,8,9], which target application scheduling on multi-core parallel computers. Some of them [3,6,8], however, either consider at most one architectural characteristic or they [9] are limited to a specific parallel computing architecture (e.g., shared or distributed memory computers). We extend our Java-based parallel programming paradigm with a scheduler capable of scheduling jobs on multi-core parallel computers. To the best of our knowledge, we are the one who provides a scheduler for Java applications which considers the low level multi-core specific characteristics (e.g., network and memory latencies, bandwidth, processor speed, shared cache, machine load).

In previous work [1], we developed JavaSymphony (JS) as a Java-based programming paradigm for parallel and distributed infrastructures such as shared

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

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 167–179, 2011.
© Springer-Verlag Berlin Heidelberg 2011

memory multi-cores and heterogeneous clusters. JS's design is based on the concept of dynamic virtual architecture, which allows the programmer to fully define a hierarchical structure of the underlying computing resources (e.g., cores, processors, machines, and clusters) and to control load balancing and locality.

A main drawback of the JS's design is the fact that the mapping of objects and tasks to computing cores has to be performed manually by the programmer.

To fill this gap, we extend JS with a scheduler based on a non-preemptive static scheduling algorithm capable of mapping the JS parallel applications (e.g., shared, distributed, and hybrid memory applications with high degree of regularity). Many architecture specific factors (e.g., processor speed, memory and network latencies, resource sharing, and machine load) influence the performance of a parallel application. Considering alone the target architecture is not sufficient to determine the sensitivity of a performance factor. The application class (e.g., communication and computation needs) and the architectural features collectively determine the performance sensitivity of a factor. Therefore, we propose a method based on training experiments that determines the sensitivities of performance factors with respect to the application classes (e.g., communication and computation-intensive) and multi-core architectures (e.g., shared memory machines, heterogeneous clusters). The training data consists of sorted lists of Performance Factor (PF), which is used by the JS scheduler as guidelines.

The paper is organised as follows. Next section discusses the related work. Section 3 presents the JS overview. Section 4 presents the JS scheduler, including its architecture, methodlogy, and algorithm. Section 5 presents experimental results and section 6 concludes the paper.

2 Related Work

Jcluster [8] is a Java-based message passing parallel environment. It provides a load balancing task scheduler based on transitive random stealing algorithm. The scheduler allows the idle nodes to steal tasks from the busy nodes. In contrast to the JS, they consider only the load balancing factor on clusters.

Proactive [3] is a Java-based parallel environment providing high-level programming abstractions based on the concept of active objects. Alongside programming, Proactive provides deployment-level abstractions for applications on multi-core machines, clusters, and Grids. In contrast to the JS, Proactive does not provide functionality to map an active object to a specific core or processor.

Parallel Java [6] is a Java-based programming environment. It provides programming constructs similar to the OpenMP and MPI. Parallel Java's scheduler keeps track of the busy and idle nodes in a cluster, and schedules the jobs by selecting one of the idle nodes. In contrast to the JS, they only consider the availability of nodes (free machines) as the main scheduling criteria.

In [9] the authors studied the impact of the shared resource contention on the application performance. To avoid the shared resource contention, they proposed a scheduling algorithm which allocates jobs in order to balance the cores' cache miss rates. In contrast to our approach, their scheduler only considers shared memory multi-cores and does not schedule applications on clusters.

In [4], the authors presented an energy-aware scheduling algorithm for heterogeneous multi-core machines. They use profiling to collect the different characteristics of a parallel program and then fuzzy logic is applied to estimate the suitability among program characteristics and cores. In contrast to our approach, they do not consider several important performance-sensitive factors such as processor computing power, co-scheduling of threads, and latencies.

Most of the related work, either considers few multi-core characteristics or they are limited to the specific target architectures. To the best of our knowledge, no Java-based scheduler considers the low-level multi-core and application related characteristics.

3 JavaSymphony

JavaSymphony is a Java-based programming paradigm for developing parallel and distributed applications. JS's high-level constructs abstract low-level infrastructure details and simplify the tasks of controlling parallelism and locality. It offers a unified solution for user-controlled locality-aware mapping of objects and tasks on shared and distributed memory architectures. Here, we provide an overview of some of the JS features, while complete details can be found in [1].

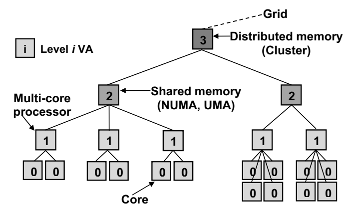


Fig. 1. Four-level VA

JS's design is based on the concept of the dynamic Virtual Architecture (VA) [1]. A VA allows the programmer to define the structure of heterogeneous computing resources and to control mapping, load balancing and migration of objects. Most existing work [3,6] assumes a flat hierarchy of computing resources. In contrast, JS allows the programmer to fully specify the multi-core architectures [1] by defining as a tree 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.

Writing a parallel JS application requires encapsulating Java objects into so called *JS objects*, which are distributed and mapped onto the hierarchical VA nodes (levels 0 to n). The object agent system [1], a part of JS runtime environment, processes remote as well as local shared memory jobs. It is responsible for creating jobs, mapping objects to VAs, migrating, and releasing objects.

Previously, the JS programmer was responsible to create the required VAs and to manually map the objects and tasks onto the VA nodes which we plan to automatise by developing a scheduler that automatically creates the required VAs and manages the mappings of the JS objects and tasks.

4 JavaSymphony Scheduler

JavaSymphony scheduler is a multi-core aware scheduler that operates in a two stages. In stage-1, training experiments are conducted offline to study the performance impacts of the different factors (e.g., processor speed, memory/network

latencies, bandwidth, co-scheduling) with respect to the two architectures and application classes (for simplicity). In stage-2, the JS scheduler uses the collected training data as guidelines to optimise the selection of the target computing resources (e.g., machines, processors, and cores). For the training and the validation experiments, we use several applications which we classify in two classes (e.g., compute-intensive and the communication-intensive). The classification of the applications is performed by measuring the computational needs of the applications, Section 5.1 describes in detail the classification experiment and the related results.

4.1 System Architecture

Figure 2 shows the JS system architecture. The JS runtime [1] is an agent-based system and has two main components: object and network agent system. The object agent system has two components: the Public Object Agent (PubOA, one for each machine), and the Application Object Agent (AppOA, one for each JS application). The network agent system monitors and interacts with the corresponding multi-core machine.

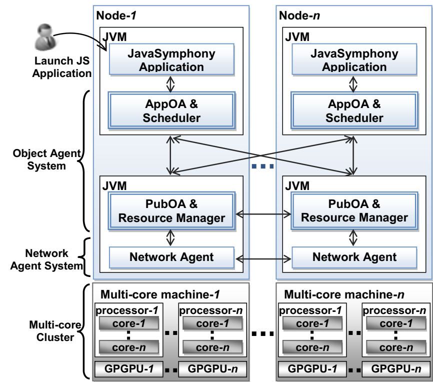


Fig. 2. The JS System Architecture

The JS scheduler has two modules: resource manager and scheduler. The resource manager is part of the PubOA and interacts with the multi-core machine with the help of the network agent. The resource manager acquires and keeps track of the physical computing resources (e.g., cores, processors, and multi-core machine) and collects the machine related information: network and memory latencies, memory hierarchies and bandwidth and processor details (e.g., topology, speed). The scheduler is part of the AppOA and runs along with the executing JS application. The scheduler uses the information provided by the resource manager and the programmer (in form of PF lists) to sub-optimally schedule the JS objects and tasks on the multi-core resources.

4.2 Scheduling Methodology

The JS scheduler considers following performance factors together with two application classes (e.g., compute and communication intensive) to determine the performance sensitivities for the factors.

1. `Network latency`: The amount of time required by a message to travel from one machine to another in a cluster.
2. `Memory latency`: The time delay which occurs for a message to travel from a main memory module to a processor (e.g., NUMA latencies).

3. `Bandwidth`: The amount of data transferred (in a second), from one machine to another in a cluster (network bandwidth) or from a memory module to a processor (memory bandwidth, such as in NUMA-based machines).
4. `Co-scheduling`: Co-scheduling of the parallel threads is achieved by mapping n threads on a multi-core processor (having n cores and shared last level cache). The not-co-scheduled execution is obtained by mapping threads on different processors (avoiding the sharing of the last level cache). The co-scheduling ratio (cos) shows the performance improvements or degradations and is calculated as follows: $cos = \frac{T(p)_{not-co-scheduled}}{T(p)_{co-scheduled}}$, where $T(p)$ is the parallel execution time of the application. The cos value greater than 1 shows improved, less than 1 shows degraded, and the cos value 1 shows no change in the performance of the application.
5. `Machine load`: The number of parallel tasks mapped on a multi-core machine. In this work, we only consider the load during the computational stage (excluding the external load). It plays significant role in the performance of the hybrid memory applications. If a machine's load is not balanced, then the over-loaded machines will face more contention on the shared resources.
6. `Processor speed`: The processor speed represents the computing power (clock frequency) of a processor.

In stage-1, the training experiments are conducted to determine the significance of the above mentioned factors for the available multi-core architectures and application classes (e.g., communication and computation intensive). We manually search a sub-optimal VA node (e.g., core, processor, or machine) by considering a performance factor and map the JS tasks onto the selected VA node. For example, to search a sub-optimal VA node with respect to the *processor speed* factor, we select the fastest available resources (e.g., cores, processors, and machines) and manually map the JS tasks onto the selected VA nodes. Using this methodology, we collect the performance impact data for all the factors with respect to the two parallel architectures and the application classes.

In stage-2, the JS scheduler utilises the collected training data (in the form of PF lists) and makes the scheduling decisions using that. To schedule a JS application on a parallel architecture, the scheduler requires the training data for the similar application class on the target parallel architecture. The JS scheduler uses a repetitive optimisation method (to find a sub-optimal VA node) by considering all the performance factors in the PF list. After optimising with respect to all the factors, the scheduler selects one of the VA node (e.g., core, processor, or machine) and maps the task onto the selected resource.

4.3 Algorithm

A JS application consists of two schedule entities: the coarse-grained JS objects and the fine-grained JS tasks. Therefore, we designed the scheduling algorithm to operate in two phases: in phase-1 the JS objects are scheduled and in phase-2 the JS tasks.

Algorithm 1. JavaSymphony Scheduler (main part)

```
   Input: AppPrgMdl,AppClass,Arch,JSObjectQ,JSTaskQ,Rlist
   Output: JS scheduled application
1  begin
2    while true do
3      phase ⟵ 1;                                      /* phase-1 (object) scheduling */
4      while JSObjectQ ≠ ∅ do
5        Object obj ⟵ pop(JSObjectQ);                             /* get next JS object */
6        VA v ⟵ GetOptNode(phase, AppClass, Arch, AppPrgMdl);
7        Schedule(obj, v);                             /* map JS object obj to VA node v */
8        UpdateResources(Rlist, obj, v);
9      end
10     phase ⟵ 2;                                        /* phase-2 (task) scheduling */
11     while JSTaskQ ≠ ∅ do
12       Task tsk ⟵ pop(JSTaskQ);                                    /* get next JS task */
13       VA v ⟵ GetOptNode(phase, AppClass, Arch, AppPrgMdl);
14       Schedule(tsk, v);                               /* map JS task tsk to VA node v */
15       UpdateResources(Rlist, tsk, v);
16     end
17   end
18 end
```

Algorithm 1 shows the main scheduling algorithm. First, the input data items are declared: *AppPrgMdl* (JS application's programming model e.g., shared, distributed, or hybrid), *AppClass* (compute-/communication-intensive), *Arch* (target architecture e.g., shared or distributed memory computer), *JSObjectQ* (JS object queue), *JSTaskQ* (JS task queue), and *Rlist* (resource status). In line 2, main scheduling loop starts. First, the scheduling phase is updated (line 3) and phase-1 scheduling starts (line 4). In line 5, the object *obj* is extracted from the object queue. Then, *GetOptNode* method (Algorithm 2) is invoked (line 6) which returns a sub-optimal VA node v, by considering the scheduling phase (*phase*), the application class (*AppClass*), the architecture (*Arch*), and the programming model (*AppPrgMdl*). Then, the *obj* object is mapped to the VA node v (line 7). Afterwards, the mapping details are passed to the resource manager (line 8). In the phase-2, the JS tasks are scheduled (lines $10 - 16$). First, the scheduling phase is updated (line 10) and the phase-2 scheduling starts at line 11. A task *tsk* is extracted from the task queue (line 12). Then, the *GetOptNode* method is invoked (line 13) that returns a sub-optimal VA node v, by considering the scheduling phase, application class, architecture, and programming model. Then, the task *tsk* is mapped to the VA node v (line 14) and the mapping details are passed to the resource manager (line 15).

Algorithm 2 shows the *GetOptNode* method. First, a VA node v is created (line 2). The existing VA nodes (e.g., cores, processors, machines, and cluster) are acquired in *vaNodes* (line 3). The performance factors list (*PFlist*) is read (line 4). For the object scheduling, the *getAppVaNode* method is invoked with parameters: the programming model, the application class, the architecture, all VA nodes, and the PFlist (lines $5 - 6$). The *getAppVaNode* method returns a sub-optimal VA node by considering the application programming model (e.g., shared, distributed, or hybrid memory). For example, if a hybrid memory JS application is scheduled, then it returns a VA node representing a multi-core

Algorithm 2. JavaSymphony Scheduler - *GetOptNode* method

```
    Input: AppPrgMdl,AppClass,Arch,phase
    Output: VA v
 1  begin
 2      VA v ⟵ ∅;
 3      VA[ ] vaNodes ⟵ ReadSystemVaNodes();                /* read all VA nodes */
 4      Vector PFlist ⟵ ReadSystemPFList(Arch, AppClass);
 5      if phase=1 then                            /* phase-1, object scheduling */
 6          v ⟵ getAppVaNode(AppPrgMdl, AppClass, Arch, vaNodes, PFlist)
 7      else if phase=2 then                         /* phase-2, task scheduling */
 8          VA[ ] optNodes ⟵ vaNodes;
 9          while PFlist.hasNext() do                /* find a sub-optimal VA node */
10              Object pfct ⟵ PFlist.getNext();
11              optNodes ⟵ getBestFitNodes(pfct, optNodes);
12          end
13          v ⟵ optNodes[0];
14      end
15      return v;
16  end
```

machine in a cluster (optimising only the network performance factors). In phase-2 (line 8), first all VA nodes are assigned to *optNodes*. After the start of the loop in line 9, a performance factor *pfct* is obtained (line 10). In line 11, *getBestFitNodes* method is invoked, this method returns a subset of VA nodes by considering optimisation with respect to a factor (*pfct*). For example, when the method is called using the performance factor *processor speed*, then it returns a subset of the sub-optimal VA nodes which represents the fastest available machines, processors, and cores. After optimising all the factors in *PFlist*, a sub-optimal VA node is assigned to v (line 13). In line 15, the VA node v is returned.

5 Experiments

We developed several JS-based real applications and benchmarks and experimented using two types multi-core parallel computers: shared memory machines (`m01` $-$ 02) and a heterogeneous cluster (`HC`, an aggregation of `m01` $-$ 02 and `k01` $-$ 03 machines). Table 1 outlines the details of the experimental setup.

5.1 Experimental Methodology

We perform two types of experiments for training and validation of the JS scheduler. Before the experiments, we classify all the applications in two classes: the *communication-intensive* and the *computation-intensive*.

Table 1. The Experimental Setup

Nodes	*Node architecture*	*Shared caches*	*Processor*	*Processor cores per node*	*Network*
m01 − 02	NUMA	L3	Quad-core Opteron 8356	32 (8 × 4)	Gigabit Ethernet
k01 − 03	UMA	-	Dual-core Opteron 885	8 (4 × 2)	Gigabit Ethernet

To classify an application, we measured the application execution time performing pure computational tasks (e.g., add, multiply, divide). For that, we measure the performance counters RETIRED_X87_FLOATING_POINT_OPERATIONS and CPU_CLOCK_UNHALTED and calculate the time (in seconds) consumed by each of the compute operations using the formula: $Time(op) = \frac{Count(op) \times CyclesPer(op)}{CpuFrequency}$, where op denotes addition, multiplication or division. Then, we sum the time consumed by the compute-operations and calculate the percentage of computation time from the overall execution. The results (shown in Figure 3) are then used to classify all the applications in compute-intensive (more than 50% time in computations) and the communication-intensive (less than 50% time in computations) classes. Within each class, we use some of the applications for the training phase and the rest for the validation experiments.

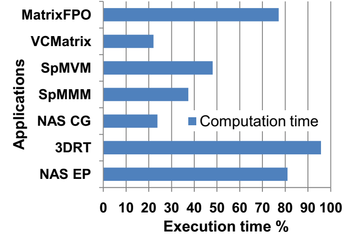


Fig. 3. Application classification

In the training experiments, we manually map JS tasks to VA nodes by searching for a sub-optimal node (considering a performance factor e.g., processor speed, latency, co-scheduling). For example, to search a sub-optimal node with respect to the *latency* factor results in the subset of the VA nodes which minimize the latency (both network and memory) of the mapped JS tasks. Similarly, we collect the performance gains achieved by each factor (whichever applicable, e.g., bandwidth on m01-02 machines is modeled by the latency, therefore it is not listed) with respect to the two parallel architectures and the application classes. Table 2 shows the results (along the performance gains achieved as compared to the default executions) as a sorted list of the performance factors (PF). The PF lists highlight the significance of the different performance factors with respect to the target architectures and the application classes.

In the validation experiments, the JS applications are scheduled by the scheduler based on the application programming model (shared, distributed, or hybrid), architecture type (shared or distributed memory computer), and application class (compute or communication intensive). The PF lists are also supplied to the scheduler (Algorithm 1), which uses them as guidelines for optimising the search and the selection of the target VA nodes. The JS scheduler creates the required VAs and sub-optimally maps the JS entities (objects and tasks) onto the VA nodes. The experiments conducted on the m01/02 machines contain up to three executions: an optimised (using the PF lists) execution by the scheduler, a

Table 2. Performance factors lists - average speedup gains

	Compute-intensive application class	*Communication-intensive application class*
$m01-02$	latency (13.28%), co-scheduling (5.19%)	latency (16.69%), co-scheduling (9.05%)
HC cluster	processor speed (30.15%) machine load (24.7%) co-scheduling (15.16%) latency (10.99%)	processor speed (25.5%) machine load (24.35%) latency (6.35%) co-scheduling (2.68%)

default Linux-scheduled execution (denoted as LSO), and a scheduler-based execution with optimisations applied using the shuffled factors (the *co-scheduling* factor is used before the *latency*) of the PF lists (PRM). The experiments conducted on the HC cluster contain up to four executions: a JS scheduler-based optimised execution (using PF lists), a Linux scheduled un-optimised execution (LSO) with the machine access order: m01-02 and k01-k03, the LSOR (Linux scheduled) with reverse machine access order: k01 − 03 and m01 − 02, and the JS scheduled execution with shuffled factors (for compute-intensive applications the *latency* factor is swapped with the *co-scheduling* and for communication-intensive applications the *machine load* factor is swapped with the *processor speed*) in the PF list (PRM).

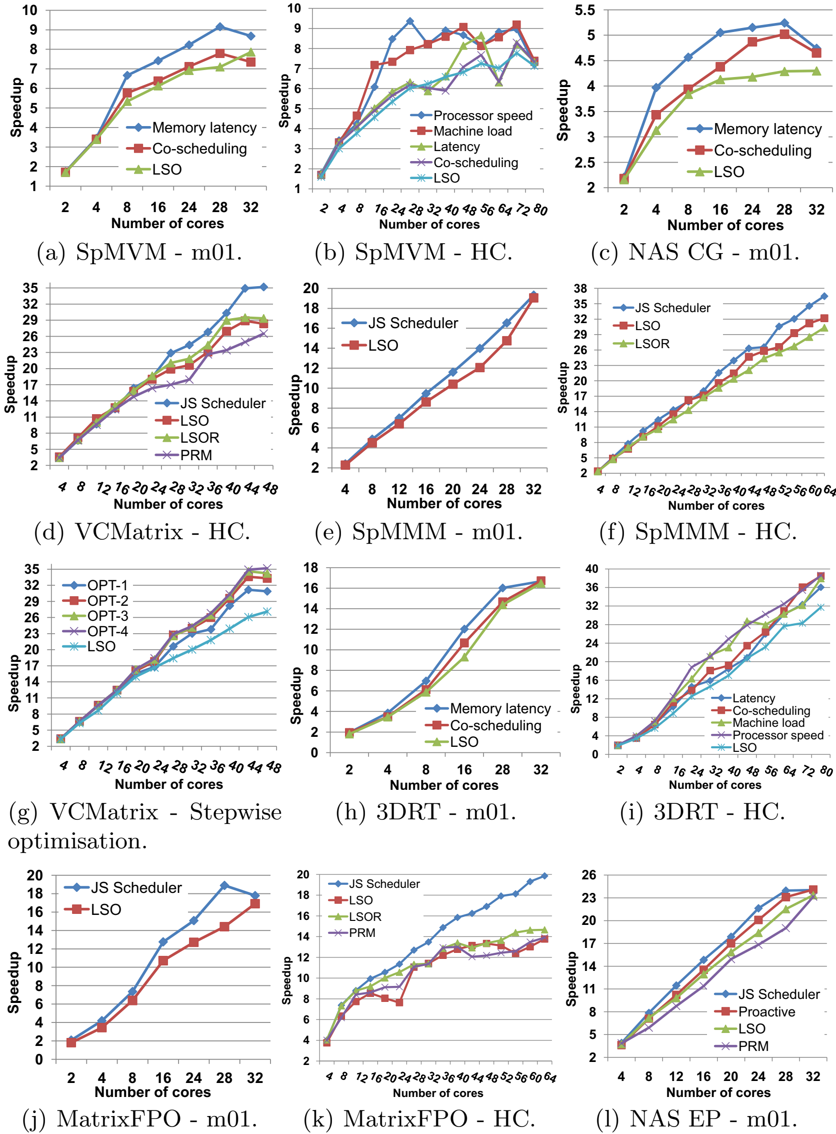
5.2 Communication-Intensive Applications

We used four communication-intensive applications for the training and the validation experiments: the Sparse Matrix-Vector Multiplication (SpMVM), the NAS parallel benchmarks CG kernel, the Variance Co-Variance Matrix computation (VCMatrix), and the Sparse Matrix-Matrix Multiplication (SpMMM).

Training Experiments. The SpMVM kernel computes $y = A \cdot x$ where A is a sparse matrix, and x, y are the dense vectors. The y is computed as follows: $y_i = \sum_{j=1}^{n} a_{ij} \cdot x_j$. We use 15000×15000 matrix, with 4000 non-zeros/row. Figure 4(a) shows the experiment results on the m01. The results show that, the *memory latency* based mappings of the JS tasks result in 15.41% improved speedup (on average) as compared to the LSO. The mappings of the parallel tasks based on *co-scheduling* factor achieves 2.69% better speedups as compared to the LSO. Figure 4(b) shows the SpMVM experimental results on the HC cluster. The results show that, the mapping of the tasks based on the *processor speed* factor achieves 25.5% speedup (on average) as compared to the LSO (using the HC machines in a random order). The other mappings (considering other factors) achieve: *machine load* (24.35%), *latency* (memory and network, 6.35%), and *co-scheduling* (2.68%), better speedups as compared to the LSO.

The CG kernel uses the power and conjugate gradient method to compute an approximation to the smallest eigenvalues. We used the Java-based implemention [5] of the kernel to develop the JS CG version. Figure 4(c) shows the experiment (size: C) results of the JS CG on m01. The *memory latency* factor based mapping of the application achieves on average 17.97% and the *co-scheduling* based mappings achieves 8.72% better speedups as compared to the LSO.

Validation Experiments. The VCMatrix computes the co-variances and variances of a matrix. The diagonal values of the resultant matrix represent variances and the off-diagonal represent co-variances. Figure 4(d) shows the experiment (matrix size: 2200×2200) results on the HC cluster. The JS scheduled execution outperformed the other three executions (PRM, LSO, and LSOR) for most of the machine sizes and achieves better speedup up-to 24.18% as compared to the LSO and LSOR, and up-to 39.96% as compared to the PRM. The JS scheduler optimises the performance of the application on the HC cluster by utilising



(a) SpMVM - m01. (b) SpMVM - HC. (c) NAS CG - m01.

(d) VCMatrix - HC. (e) SpMMM - m01. (f) SpMMM - HC.

(g) VCMatrix - Stepwise optimisation. (h) 3DRT - m01. (i) 3DRT - HC.

(j) MatrixFPO - m01. (k) MatrixFPO - HC. (l) NAS EP - m01.

Fig. 4. JS scheduler experimental results - training and validation

the fastest available resources (e.g., cores, processors, and machines), reducing the data contention on m01 – 02 (by balancing the machine load), and reducing memory latencies (in the m01 – 02 NUMA machines).

The SpMMM multiplies two sparse matrices to computes the resultant sparse matrix. Figure 4(e) shows the experiment (matrix size: 10000×10000, 1000 non-zeros/row) results on the m01 machine. The JS scheduled execution achieves better performances, and gains up-to 15.92% more speedups as compared to

the LSO. Figure 4(f) shows the experiment results on the HC cluster. The JS scheduler-based execution achieves better results and achieves up-to 15.13% (as compared to LSO) and 21.17% (as compared to LSOR) more speedups.

To investigate the optimisation effects for the different factors, we experimented with VCMatrix application on the HC cluster and step-wise optimised the LSO execution. First, we use the PF list with only one factor (most significant) and scheduled the application. We then add other factors stepwise and schedule the application. The results (shown in Figure 4(g)) show that, the OPT-1 (using the top most factor) achieves up-to 19.47% speedup as compared to the LSO. The OPT-2 (using the top two factors) achieves up-to 10.18% more speedups as compared to the OPT-1. The OPT-3 and the OPT-4 (using the top three and four factors) achieves further speedups up-to 3.04% and 2.75%. The results show that, the most of the performance (here, up-to 83.66%) can be achieved by optimising the two most significant factors.

5.3 Computation-Intensive Applications

We use three compute-intensive applications for the training and validation experiments. These are: the 3D Ray Tracing (3DRT), the NAS benchmarks EP kernel, and the Matrix Transposition with Floating Point Operations (MatrixFPO).

Training Experiments. We developed the JS-based versions of the 3DRT application using the Java-based version from Java Grande benchmarks [7]. It is a large-scale application that creates several ray tracers, initialises them with scene data (64 spheres), and renders at N×N resolution. Figure 4(h) shows the experiment (image size: 4000×4000) results on the m01. The results show that, the *memory latency* factor based mapping of the 3DRT achieves on average 13.28% improved speedup as compared to the LSO. The mappings of the application based on the *co-scheduling* factor achieves on average 5.19% better speedups as compared to the LSO. Figure 4(i) shows the experiment results on the HC cluster. The *processor speed* based mapped application achieves best speedup (on average 30.15% more) as compared to the LSO. The speedups achieved by mappings (considering the other factors) are: *machine load* (24.7%), *co-scheduling* (15.16%), and *latency* (memory and network, 10.99%), better speedups as compared to the LSO.

Validation Experiments. The MatrixFPO transposes a matrix, and performs several floating point operations (e.g., addition, multiplication, and division) at each transpose step. Figure 4(j) shows the experiment (matrix size: 10000×10000) results on the m01. The JS scheduled execution of the application achieves up-to 31.09% better speedups as compared to the LSO. Figure 4(k) shows the experiment results on the HC cluster. The JS scheduler based execution achieves better speedups for most of the machine sizes and achieves up-to 48.75% (as compared to the LSO), 35.47% (as compared to the LSOR), and 44.05% more speedups (as compared to the PRM execution). The JS scheduled execution

achieves better performance results as compared to other executions because of the optimisations (using the PF list for the target application class and the architecture) applied by the JS scheduler.

The NAS EP kernel is used to measure the computational performance of parallel computers. Figure 4(l) shows the experiment (data size: 16777216×100) results on the `m01`. We experimented and compared the JS scheduled, LSO, PRM, and the Proactive [3] based executions. The results show that, the JS scheduled execution achieves better speedups as compared to all the other executions and achieves up-to 16.82% (as compared to the LSO), up-to 12.13% (as compared to the Proactive), and up-to 30.32% (as compared to the PRM) more speedups. Proactive exhibits the low performance since it has no capability to map the *active objects* on specific cores. Although the EP kernel has small memory footprints still the remotely scheduled Proactive objects cause some performance degradations as compared to the JS scheduled execution.

6 Conclusions

In this paper, we presented a multi-core aware scheduling extension to JavaSymphony. JS scheduler provides the locality controlled mapping of JS tasks and objects. JS scheduler makes scheduling decisions by considering both the application class and the multi-core specific performace factors. We presented JS scheduler's architecture, methodlogy, and algorithm. We developed several real applications and benchmarks, and experimented using two real multi-core parallel computers. The experimental results show that, the JS scheduled parallel applications outperform other scheduling heuristics and technologies such as Proactive and operating system relying application scheduling.

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. Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese, B.: Piranha: a scalable architecture based on single-chip multiprocessing. In: In the 27th ISCA 2000, p. 282. ACM, New York (2000)
3. Caromel, D., Leyton, M.: Proactive parallel suite: From active objects-skeletons-components to environment and deployment. In: Euro-Par Workshops, pp. 423–437. Springer, Heidelberg (2008)
4. Chen, J., John, L.K.: Energy aware program scheduling in a heterogeneous multi-core system. In: Proceedings of the IEEE International Symposium on Workload Characterization, 2008, pp. 1–9. IEEE Computer Society, Los Alamitos (2008)
5. Frumkin, M.A., Schultz, M., Jin, H., Yan, J.: Performance and scalability of the NAS parallel benchmarks in Java. In: IPDPS, p. 139a. IEEE Computer Society, Los Alamitos (2003)

6. Kaminsky, A.: Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In: 21st IPDPS. IEEE Computer Society, Los Alamitos (2007)
7. Smith, L.A., Bull, J.M.: A multithreaded Java grande benchmark suite. In: Third Workshop on Java for High Performance Computing (2001)
8. 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)
9. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. In: ASPLOS 2010, pp. 129–142. ACM, New York (2010)