Node Sharing for Increased Throughput and Shorter Runtimes – an Industrial Co-Scheduling Case Study

Andreas de Blanche
andreas.deblanche@tetrapak.com[1]
andreas.de-blanche@hv.se[2]

[1]Engineering Excellence Department
AB Tetra Pak, Sweden

Thomas Lundqvist
thomas.lundqvist@hv.se[2]

[2]Engineering Science Department
University West, Sweden

ABSTRACT

The allocation of jobs to nodes and cores in industrial clusters is often based on queue-system standard settings, guesses or perceived fairness between different users and projects. Unfortunately, hard empirical data is often lacking and jobs are scheduled and co-scheduled for no apparent reason. In this case-study, we evaluate the performance impact of co-scheduling jobs using three types of applications and an existing 450+ node cluster at a company doing large-scale parallel industrial simulations. We measure the speedup when co-scheduling two applications together, sharing two nodes, compared to running the applications on separate nodes. Our results and analyses show that by enabling co-scheduling we improve performance in the order of 20% both in throughput and in execution times, and improve the execution times even more if the cluster is running with low utilization. We also find that a simple reconfiguration of the number of threads used in one of the applications can lead to a performance increase of 35-48% showing that there is a potentially large performance increase to gain by changing current practice in industry.

1 INTRODUCTION

This case-study evaluates the performance impact of co-scheduling industrial engineering simulations at a large manufacturing company in Sweden, AB Tetra Pak. We evaluate the difference between scheduling different types of parallel jobs on nodes dedicated to each job or co-scheduling several jobs to the same nodes. This is illustrated in Figure 1: In Figure 1a two different jobs are scheduled on one node each and in Figure 1b the jobs are co-scheduled on both nodes, thus sharing resources. Breslow et al. [1] refer to this as job-striping.

To evaluate the impact of co-scheduling with job-striping on the runtime and throughput of engineering simulations, we study two proprietary (3rd party) and one open-source software used in production at AB Tetra Pak: Abaqus, LS-Dyna, and Lammps. The cases, in terms of configurations and input model data, executed by the programs are real production cases that represent a large body of simulations they run.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice.
COSH2018, January 2018, Manchester, United Kingdom
© 2018 Copyright held by the authors. Published in the TUM library.
https://doi.org/10.14459/2018md1428537

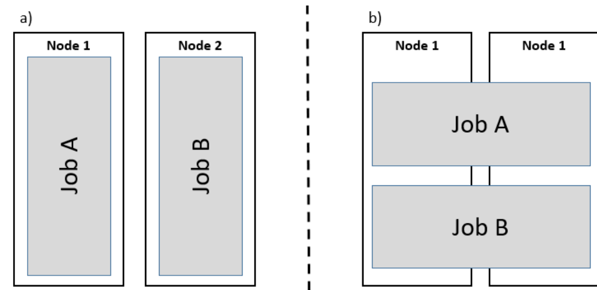

Figure 1: Example of different scheduling strategies on 24-core nodes. In a) job A is executing on 24 cores in node 1 and B is executing on 24 cores in node 2. In b) the two jobs are co-scheduled over two nodes, each job is using 12 cores in each machine.

We first measure the execution-time impact of co-scheduling, then analyze the performance impact of different scheduling strategies, and finally, we look at thread versus MPI-processes configuration trade offs for the Abaqus software. Based on the results, we draw the following conclusions:

- Co-scheduling two jobs, with job-striping, results in around 20% higher throughput as well as 20% shorter execution times compared to executing the same two jobs as in Figure 1a.
- When a cluster is not fully utilized the execution times can be decreased further by executing the jobs over two nodes instead of on one node. Hence, it is beneficial to fill up all nodes without co-scheduling jobs before starting to co-schedule.
- With in-depth knowledge about the programs' co-scheduling slowdowns, the job scheduler can increase the performance by another 5%.
- By reconfiguring the ratio of threads versus MPI processes for one of the programs, Abaqus, the performance is increased by between 35% to 48% compared to the default setting.

2 CLUSTERS FOR INDUSTRIAL ENGINEERING AND CO-SCHEDULING

There is a class of relatively small high-performance computing clusters that are used by industrial companies to perform engineering simulations. These systems range in scale from around 500 up to 100,000 cores. They use, as an example [2], Finite Element (FEM) simulations to simulate folding of e.g. paper, heat treatment, welding, or car crashes, and computational dynamics (CFD) algorithms

to simulate, e.g., liquids flowing through a system of pipes, or air through a turbine or around a car.

A common denominator for industrial simulation systems is that they often run proprietary software and that the cost of software licenses is an order of magnitude higher than the cost of hardware procurement, energy use and maintenance combined [3]. To complicate matters the hardware and licenses are often shared between many engineers, thus, fairness and prioritization between users and projects are important. Our case-study company has a cluster of 450+ multi-core compute nodes where they run approximately ten major programs.

In the context of fairness and basic prioritization the engineers at Tetra Pak are allowed, at any given time, to run two jobs using a maximum of 24 cores each in the main queue. They chose 24 cores since that is the number of cores in a single node and since the scaling of the most expensive software declines after that point (when executing implicit FEM models). The maximum number of cores per job might very well change in the future. However, during off-hours (6pm-8am and weekends), another lower-priority queue is opened where no restrictions apply.

Many previous studies [4] [5] [6] [7] [8] have shown promising results in the area of co-scheduling with throughput increases in the range of 5-40% for the specific workloads. In [1], Breslow et al. made the case for co-location of jobs by job-striping where they by co-scheduling pairs of jobs on a set of nodes improved the average throughput by 12%.

The goal of co-scheduling is to decrease the slowdown caused by resource contention when executing programs that use different resources at the same time. Co-scheduling is as important on the operating system level as on the compute node or cluster level. At the operating system, or intra-node level; Süß, et al. [9] looked at different scheduling strategies to improve the operating system co-scheduling efficiency and Eyeman and Eeckhout [10] showed that their probabilistic symbiosis approach achieved a 19% reduction in job turnaround time for a four-threaded SMT enabled processor.

When possible, the intra node co-scheduling problem can be made easier by co-scheduling processes that do not compete for resources on the same node. On a large cluster many (co-) scheduling decisions, i.e. which program to run on which nodes together with which other program(s), are taken every minute. Several techniques [3] [11] [12] [13] [14] for resource usage and co-scheduling slowdown characterization have been suggested. Breitbart et al. [4] suggest that processes should be migrated to other nodes if resource contention causing large slowdowns are detected, which is a technique commonly used for virtual machines in data centers and clouds.

The main question we ask in this case-study is, would it be better to run the simulations according to Figure 1a or 1b. Would the jobs benefit or be hurt by co-scheduling? However, one should keep in mind that in case 1a there are P processes or threads from the same program competing for resources and in 1b there are $P/2$ processes from program A and $P/2$ processes from program B competing for resources.

3 HARDWARE AND SOFTWARE

The co-scheduling slowdown measurements were carried out on compute nodes equipped with two Intel Xeon E5-2650 v4 processors. Each E5-2650 has 12 cores and one node has 128 GB of RAM. The cluster has a 56-Mbps InfiniBand network for communication between nodes.

The programs studied were two proprietary Finite Element solvers, Abaqus (ABQ) and LS-Dyna (LSD), and one open-source molecular-dynamics simulator, Lammps (LAM). Abaqus is developed by Simulia, 3DS and the standard implicit solver is used to perform a simulation on a paperboard. The model used as input takes approximately 4 hours to execute in parallel on a single node using 24 cores. The Abaqus standard solver can be executed on a single node or over several nodes. It uses MPI to spawn one process per node and then each process creates several threads to carry out the simulation in parallel. Abaqus is licensed on a per thread basis, the more threads you use, the more licenses are required. The license usage scales with $roundDown(T^{0,422})$, where T is the number of threads.

Lammps is a molecular dynamics code, and an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator [15]. It is distributed as an open-source code under the GPL license. Lammps use MPI to distribute jobs over the assigned cores. The model used is a real production model and it executes in 1 hours and 49 minutes using one node and 24 cores.

The third software is Ls-Dyna from Livermore Software Technology Corporation (LSTC). LS-Dyna is mostly known for running large explicit FEM models over hundreds or thousands of cores. However, in this case-study we use the implicit solver to run a production case that has been shortened in order to not execute for so long. The shortened model finishes in 28 minutes when executed on 24 cores in a single node. The LS-Dyna solver use MPI to distribute processes over cores and it allocates one license per core.

All measurements were performed on the production system and many different, but identical, nodes were used. The programs were always spawning 24 worker threads or processes over one or two nodes. When calculating the co-scheduling execution times of jobs with different length the software with the shorter execution time was padded with extra executions so the pressure on the longer job was sustained during the entire execution. To get reliable data each measurement was repeated at least three times, and the average execution time was used. The repeat measurements were performed on different nodes and on different days.

4 CO-SCHEDULING EVALUATION

From this point on we will refer to the Abaqus software and the FEM model used as ABQ, the Lammps software and that model as LAM, and the LS-Dyna software and model as LSD. The first column in Table 1 shows the execution time, in minutes, when running the jobs on all cores in a single node. The second column shows the execution time when running the same job over two nodes, only utilizing half of the cores in each node and letting the remaining cores be idle. As can be seen all jobs experience a speedup when they are split over several nodes. Thus, indicating that the single node performance is limited by resources other than the cores. The speedup when allocating two (half-)nodes is between 12% to 29%. The measurements also show that the negative impact of the single

Table 1: Comparison between executing each type of job (software and model) using 24 cores on one node or splitting the job over two nodes, utilizing 12 cores on each node. Execution times are given in minutes.

	Execution on single node	Execution on two half-full nodes	Speedup on two nodes
ABQ	246 min	196 min	20%
LAM	109 min	95 min	12%
LSD	28 min	20 min	29%

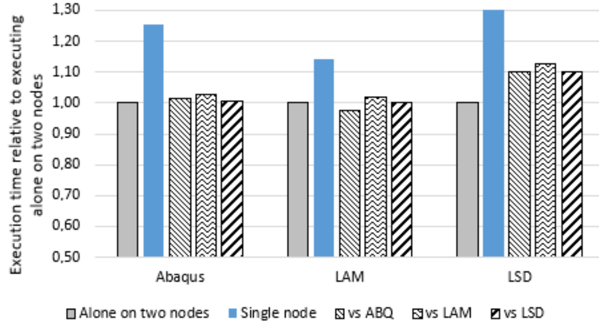

Figure 2: The impact of co-scheduling. All jobs but Single node are executed over two nodes. All numbers are relative to the execution time of a job executing alone over two nodes.

node resource sharing is far greater than the added communication cost that arise when a job is distributed over two nodes.

The speedup gained comes with the cost of doubling the hardware allocation for a job. A bit non-intuitive, this can still be cheaper when accounting for software license costs. As earlier stated it is common in the industrial engineering field that the license costs represent $9/10^{th}$ of the total cost and the hardware only $1/10^{th}$. Hence, doubling the hardware increases the job cost by 10% but decreases the execution time by 12-29%.

Figure 2 shows the relative execution time of different execution possibilities for our jobs. The first two bars show a single job execution either over two nodes or on a single node. Executing the job over two nodes is significantly faster. The three last bars show how the execution time is affected when the job is executed over two nodes and co-scheduled with another job (job-striping). As can be seen, the execution time increases slightly for ABQ and LAM, up to 3% while LSD sees execution time increases of between 10% and 13%. LSD is most sensitive to resource competition, which was anticipated given that it saw the largest benefit of being split over two nodes. Also, LAM has a large negative impact on co-scheduled processes, also when co-scheduled with itself. Hence, not combining anything with LAM would be a good idea. We currently have no explanation about why LAM is executing faster when co-scheduled with Abaqus.

Co-scheduling two jobs on two nodes increase the per-job execution time compared with executing one job over two nodes. However, executing one job per node is always slower by 11% to 21%, on average 17%. The real benefit of co-scheduling lies in the throughput. In Figure 2 all bars, except *Alone*, represent the completion of two jobs. The execution slowdown when co-scheduling two LSD jobs, which has the largest slowdown, over two nodes is 10% but two LSD jobs are able to finish simultaneously. Table 2 summarizes the throughput increase compared to executing one job over two nodes, which is at least a factor of 1.82, as well as compared to executing two jobs on separate nodes, which is at least a factor of 1.12.

Table 2: Throughput increase when co-scheduling two jobs over two nodes compared to one job over two nodes and two jobs executing on one node each.

Compared to	ABQ ABQ	ABQ LAM	ABQ LSD	LAM LAM	LAM LSD	LSD LSD
Single job over two nodes	1.97x	2.00x	1.90x	1.96x	1.87x	1.82x
Two jobs on one node each	1.24x	1.20x	1.26x	1.12x	1.18x	1.27x

Co-scheduling two jobs over two nodes (job-striping) results in a 0-13% slowdown in execution times, compared to executing one job over two nodes. But, the throughput is increased by 82-100%. If we compare with executing two jobs on two separate nodes, the co-scheduling execution times are 12-27% shorter and the throughput is 12-27% higher. These measurements show that by co-scheduling ABQ, LAM, and LSD, we get both shorter execution times as well as higher throughput, compared to when two jobs are executed on one node each.

5 COMPARING CO-SCHEDULING STRATEGIES

To illustrate the impact of co-scheduling, we now investigate different (co-)scheduling strategies based on our three programs. What would be the best way to schedule a set of random jobs given equal distribution between ABQ, LAM, and LSD? We choose to evaluate the difference between eight different scheduling strategies:

Default: Use the default algorithm of running one job on one node. Keeping all processes of the same job on the same node. This strategy is often used to minimize communication latencies.

50% usage: Each job is executed over two nodes, effectively leaving half of the cores idle.

Same type: All jobs are co-scheduled so that two jobs of the same software using the same model share two nodes, i.e., two ABQ jobs share two nodes, etc.

Terrible Twins: Two jobs are co-scheduled over two nodes but they are never co-scheduled with another instance of the same program. In earlier studies [16] [17], we showed that, on average, it would be slightly beneficial not co-schedule several instances of the same program on the same node, but instead co-schedule different programs.

Keep ABQ: ABQ jobs are co-scheduled with another instance of themselves over two nodes. LAM and LSD are co-scheduled over two nodes with each other.

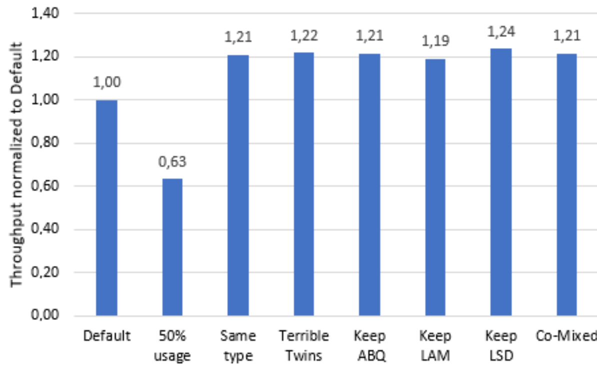

Figure 3: Throughput comparison between the eight different co-scheduling strategies on a 100% utilized cluster

Keep LAM: LAM jobs are co-scheduled with another instance of themselves over two nodes. ABQ and LSD are co-scheduled over two nodes with each other.

Keep LSD: LSD jobs are co-scheduled with another instance of themselves over two nodes. ABQ and LAM are co-scheduled over two nodes with each other.

Co-mixed: Jobs are co-scheduled over two nodes together with another job. All combinations are allowed, and no restrictions apply.

Figure 3 shows the throughput given an even amount of work of each job type and a 100% cluster utilization. It is interesting to note that the *50% usage* approach performs at 63% of the Default approach, even though it only utilizes half of the cores in the cluster. This is thanks to the jobs' shorter execution times when running alone.

The six co-scheduling approaches can be divided into three generic and three specific alternatives. The generic ones, that do not require any knowledge about the specific programs, are *Same type*, *Terrible twins* and *Co-Mixed*. They all perform approximately the same, around 21% better then *Default*. *Co-mixed* performs 0.4 percentage points better than *Same type* and 0.4 percentage points worse than *Terrible twins*. This goes well with earlier studies on *Terrible twins* [15,16] which have shown the same pattern. However, among the specific alternatives (Keep ABQ, LAM, or LSD) the spread is 4.5 percentage points. This shows that with in-depth knowledge of the jobs it is possible to outperform the generic methods by, in this case, 1.8 percentage points, and the Default by 23.4%. To conclude, in a fully utilized (100% loaded) cluster there is a significant benefit of co-scheduling two programs on two nodes. Turning to Table 3 it becomes obvious that the benefit is even higher when the cluster is not fully utilized. This is due to the faster execution times when a job runs over two nodes instead of a single node. At 75% load $2/3^{rd}$ of the jobs are co-scheduled and $1/3^{rd}$ is executing over two nodes, alone.

When the cluster has a utilization of 50% or lower, all co-scheduling approaches are simply placing jobs on two nodes while not performing any actual co-scheduling. Co-scheduling starts when the utilization rises above 50% and we can see that all six co-scheduling approaches have quite similar performance. The *Co-Mixed* approach is the easiest to implement since no consideration is taken to which

Table 3: Throughput comparison between the scheduling approaches. All values are in comparison with the Default approach. The rows are for a 100%, 75%, 50%, and 25% loaded cluster.

utilization	Default	50% usage	Same type	Terrible Twins	Keep ABQ	Keep LAM	Keep LSD	Co-Mixed
100%	1.00	0.63	1.21	1.22	1.21	1.19	1.24	1.21
75%	1.00	0.95	1.24	1.24	1.21	1.22	1.27	1.24
50%	1.00	1.27						
25%	1.00	1.27						

jobs should be co-scheduled. *Terrible Twins* deliver slightly better performance, and the heuristic is quite simple; do not put two instances of the same program on the same nodes. However, implementing *Keep LSD* renders the best overall performance. The drawback of the Keep approaches is that we must first measure the performance of all combinations before we can select which ones to co-schedule and which ones to not.

Not only are the throughputs higher for the six co-scheduling approaches in Table 3 compared to *Default*, the job execution times are also between 11% and 29% shorter.

6 ABAQUS AND MP_HOST_SPLIT

The Abaqus software has a setting that makes it possible to specify the number of MPI processes to use per node, mp_host_split. The default value is one process per node, and that setting was used in all previous measurements. To evaluate the impact of this setting, we decided to set mp_host_split to 4 and measure the impact it has on the execution time, throughput and co-scheduling slowdowns.

When mp_host_split is set to 4 and Abaqus is executed on a single 24-core node, it will start four MPI processes and each process will spawn 6 threads. This is the same division that would happen if the job was spread out over four nodes. In some sense, mp_host_split is splitting up the workload in four smaller pieces and co-scheduling them on the same node. From now on we will refer to this mixed parallelization approach with 4 processes per node as ABQ^{split}. Figure 4 illustrates the number of processes and threads for the different settings. As exemplified in Figure 4d, ABQ^{split}, when executed over two nodes, will create four MPI processes on each node which spawns three threads each.

Table 4 compares the execution times of ABQ^{split} with ABQ. On a single node, there is a 48% speedup. When executing over two nodes ABQ^{split} is 35% faster. However, ABQ^{split} do not gain much by executing over two nodes, the difference between executing on a single node compared to two nodes is only two percent.

Splitting the ABQ job into several MPI tasks with fewer threads is recommended. The reasons behind this is probably twofold. First, performance almost never scales linearly with the number of threads, the law of diminishing returns state that the benefit of each new thread is less and less. So, solving four problems sequentially using 24 threads at a time might not be as fast as solving the four

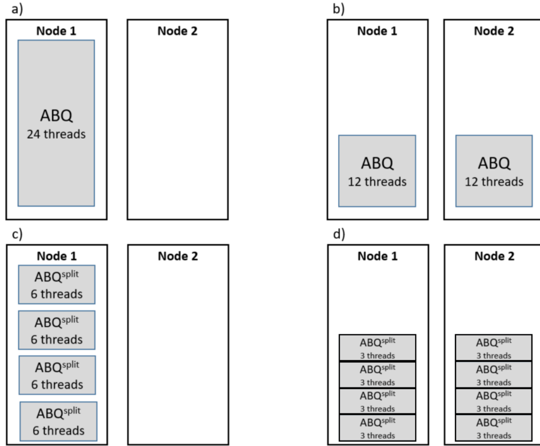

Figure 4: The four images a-d illustrates how a 24-process job of ABQ and ABQ distributes its MPI processes and threads when executing on a single node or over two nodes.

Table 4: Comparison between standard ABQ (1 processes per node) and ABQ^{split} (4 processes per node). The 24 threads are evenly divided between the processes.

	Execution single node	Execution two half-full nodes	Speedup on two nodes
ABQ	246 min	196 min	20%
ABQ^{split}	129 min	127 min	2%
Speedup ABQ^{split}	48%	35%	

problems in parallel using six threads on each problem. Secondly, by using four MPI processes chances are that not all four processes will use the same off-core resources at the same time. Hence, if the processes resource requests are interleaved the resource congestion will decrease. Thus, the resource demands of ABQ^{split} will, be more even, over time compared to ABQ.

When looking at co-scheduling slowdowns, ABQ^{split} is more susceptible to slowdowns than ABQ, as can be seen in Figure 5. When ABQ^{split} is co-scheduled, over two nodes, with another instance of ABQ^{split} the execution time is increased by 17%, compared to 2% for ABQ. When co-scheduled with LAM, the slowdown is 21%, compared to 3% for ABQ. But LAM executes one percentage point faster when combined with ABQ^{split} then with ABQ.

In conclusion, ABQ^{split} makes better use of the resources, and executes between 23% and 35% faster than standard ABQ in all co-scheduling combinations. The drawback is that it becomes more susceptible to co-scheduling slowdowns, as illustrated in Figure 5.

ABQ^{split}'s increased susceptibility to co-scheduling has a direct impact on the (co-)scheduling approach where the scheduler first starts to execute all programs over two nodes and starting to co-schedule them when the load goes above 50%. The approach still increases the overall throughput for the general co-scheduling mixes (*Same type*, *Terrible Twins*, and *Co-Mixed*) with between 9%

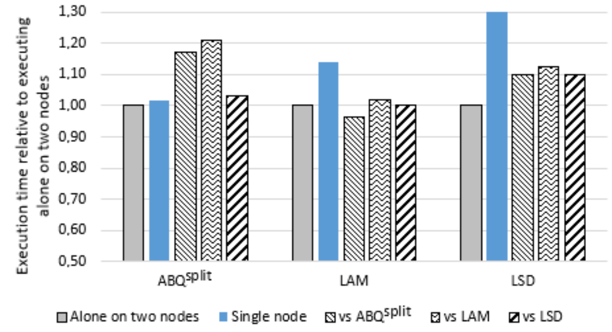

Figure 5: The impact of co-scheduling. All jobs but *Single node* are executed over two nodes. All numbers are relative to the execution time of a job executing alone over two nodes.

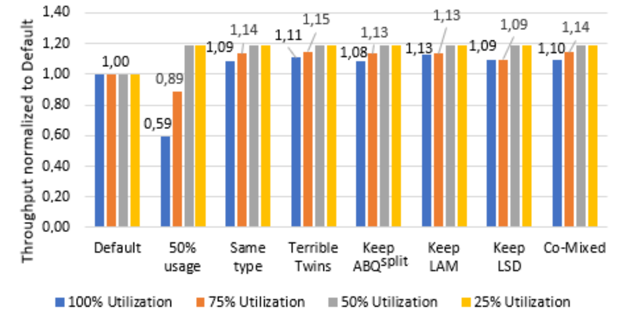

Figure 6: Throughput comparison between the eight different co-scheduling strategies. At four different cluster utilization levels.

and 11% on a fully loaded cluster. When the load is 50% or less the throughput is 19% better then Default.

The *Terrible Twins* approach has a throughput that is two percentage points higher than Same type and one point higher than *Co-Mixed*. Turning to the approaches that require program-specific knowledge we can see that Keep LSD has the best throughput increase with 14%. One has to keep in mind, though, that with loads above 50% some ABQ^{split} jobs will see an execution time increase compared to *Default*. Still, the execution times are always significantly better than for ABQ.

7 CONCLUSION

This case-study examined the performance impact of co-scheduling jobs over two nodes, i.e. job-striping, compared to running each job on a separate node. The overall conclusion, based on experimental results of three different engineering programs, executing production models, on a cluster with 24-core nodes, is that co-scheduling always results in higher throughput. In this case the throughput increase was 12-27% which is higher then the average job-striping throughput increase of 12% shown in [1].

Execution times are also improved: spreading a job over two nodes, utilizing 24 of 48 cores, improves the execution time by 12-29% compared to executing the exact same job on a single node. However, adding a second, co-scheduled, job to the same two nodes results in a slowdown, but only by 0-13%. Hence, node sharing (co-scheduling) both increases the throughput and shortens the run times of co-scheduled jobs.

This is even more so when the cluster is running with less than 100% utilization. Then, some, or all, nodes will run half full, and these nodes will not experience any co-scheduling slowdowns, but will still gain from the execution time speedups. Hence, we propose a scheduling strategy that first allocates jobs to empty nodes and only when there are no empty nodes start to co-schedule.

Based on an analysis of eight scheduling strategies, we can also conclude that different rules about which applications to co-schedule have a minor effect on the performance. The largest gain, an average speedup of 21%, has already been obtained by spreading a job over two nodes. The impact of different scheduling rules results in speedups between 19% up to 24%. One of the better strategies is also the simplest one to implement: first place jobs on empty nodes, when there are no empty nodes just randomly place jobs on nodes without relying on any prior knowledge on slowdowns.

One final conclusion that can be made is that program settings can have a large influence on the results. By changing a setting in the Abaqus software that increases the number of processes and reduces the number of threads per process, the execution time for this application is reduced by 48% when running on a single-node and 35% when spread out over two nodes. However, the program now also becomes more sensitive to co-scheduling, resulting in larger co-scheduling slowdowns. Nevertheless, the overall performance is still much better than when using the original program settings.

Being only a case study, there are many options still to explore. For example, only three programs have been studied. Other options would be to execute programs on greater number of nodes or co-scheduling three or more programs on the same node. This case study clearly shows that there is much to gain by distributing programs and enabling co-scheduling.

REFERENCES

[1] A.D. Breslow, L. Porter, A. Tiwari, M. Laurenzano, L. Carrington, D.M. Tullsen, and A.E. Snavely. The Case For Colocation of HPC Workloads. In *Concurrency Computat.: Pract. Exper.*. Volume 28, Issue 2 February 2016.

[2] P. Lindstrom and A. de Blanche. Integration and Optimization of a 64-core HPC For FEM- and/or CFD Welding Simulations. In *Proceedings of the NAFEMS NORDIC seminar on Improving Simulation Prediction by Using Advanced Material Models*, 5âĂŞ6 November 2013, Lund, Sweden, 2013.

[3] A. de Blanche and S. Mankefors-Christiernin. Minimizing Total Cost and Maximizing Throughput - A Metric for Node versus Core Usage in Multi-Core Clusters. In *Proceedings of the International conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 2010.

[4] J. Breitbart, S. Pickartz, J. Weidendorfer, S. Lankes, and A. Monti. Dynamic Co-Scheduling Driven by Main Memory Bandwidth Utilization. In *IEEE International Conference on Cluster Computing (CLUSTER)*, September, Honolulu, Hi, USA, 2017.

[5] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04*, 2004.

[6] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA '11: Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 283–294, New York, NY, USA, 2011. ACM.

[7] J. Breitbart, J. Weidendorfer, and C.R. Trinitis. Automatic Co-scheduling based on Main Memory Bandwidth Usage. In *Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, US, 2016.

[8] A. de Blanche and T. Lundqvist. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing*, 71(4):1451–1483, 2015.

[9] T. Süß, N. DÃűring, R Gad, L. Nagel,A. Brinkman, D. Feld, E. Schricker, and T. Sodderman. Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling. In *Co-Scheduling of HPC Applications, book chapter, ed. Trinitis, C. and Weidendorfer, J.* , 2017.

[10] S. Eyerman and L. Eeckhout. Probabilistic Modeling for Job Symbiosis Scheduling on SMT Processors. In *ACM Transactions on Architecture and Code Optimizations (TACO)*, Vol 9, No 2, June 2012.

[11] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS on Architectural support for programming languages and operating systems*, pages 129–142, New York, NY, USA, 2010. ACM.

[12] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[13] J. Mars, L. Tang, R. Hunt, K. Skadron, and M.L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of IEEE/ACM international symposium on microarchitecture*, New York, USA, 2015.

[14] J. Weidendorfer and J. Breitbart. Detailed Characterization of HPC Applications for Co-Scheduling. *1st COSH Workshop on Co-Scheduling of HPC Applications, HiPEAC*, Prague, Jan, 2016.

[15] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. In *Journal Computational Physics*, 117, 1-19, 1995.

[16] A. de Blanche and T. Lundqvist. Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedules. *1st COSH Workshop on Co-Scheduling of HPC Applications, HiPEAC*, Prague, January, 2016.

[17] A. de Blanche and T. Lundqvist. Disallowing Same-program Co-schedules to Improve Efficiency in Quad-core Servers. *2st COSH Workshop on Co-Scheduling of HPC Applications, HiPEAC*, Stockholm, January, 2017.