Proceedings of the

3rd Workshop on Co-Scheduling of HPC Applications (COSH 2018)

Manchester, United Kingdom
January 23, 2018
Co-located with HiPEAC 2018

Workshop Co-Chairs

Carsten Trinitis and Josef Weidendorfer

DOI: 10.14459/2018md1428535

Published in the TUM library (mediaTUM), January 2018

(This page intentionally left blank)

Contents

(This page intentionally left blank)

Foreword to the Proceedings of the 3rd Workshop on

Co-Scheduling of HPC Applications (COSH 2018)

Co-located with HiPEAC 2018
in Manchester, United Kingdom, January 23, 2018

These proceedings comprise the papers from the 3rd Workshop on Co-Scheduling of HPC Applications (COSH 2018), co-located with the HiPEAC 2018 conference in the city of Manchester, United Kingdom, European Union. Three papers plus a keynote talk by Pedro Moura Trancoso were presented at the workshop.

Once again, the workshop attracted submissions from a range of countries, highlighting the international character of the workshop. Each submission has been reviewed by at least three program committee members and, at the end, we selected three high quality papers for presentation and publication.

The invited keynote was given by Pedro Moura Trancoso of the University of Cyprus (Cyprus) and Chalmers University of Technology (Sweden) about recent research on scheduling for the many-core era.

We would like to say a big thank you to the members of our program committee who did a great job in reviewing all submissions thoroughly and conscientiously. Furthermore, we also want to take the opportunity and thank the entire HiPEAC Conference Committee and especially the Workshops & Tutorials Chairs as well as the on-site organization team for making this event possible and successful.

Manchester was a great location for the conference and its workshops. The event led to fruitful discussions around co-scheduling which resulted in new insights for all attendees.

January 2018

Carsten Trinitis, Josef Weidendorfer
(COSH Workshop Co-Chairs)

COSH: Workshop Background and Topics

The task of a high performance computing system is to carry out its calculations (mainly scientific applications) with maximum performance and energy efficiency. Up until now, this goal could only be achieved by exclusively assigning an appropriate number of cores/nodes to parallel applications. As a consequence, applications had to be highly optimised in order to achieve even only a fraction of a supercomputer's peak performance which required huge efforts on the programmer side.
This problem is expected to become more serious on future exascale systems with millions of compute cores. Many of today's highly scalable applications will not be able to utilise an exascale system's extreme parallelism due to node specific limitations like e.g. I/O bandwidth. Therefore, to be able to efficiently use future supercomputers, it will be necessary to simultaneously run more than one application on a node. To be able to efficiently perform co-scheduling, applications must not slow down each other, i.e. candidates for co-scheduling could e.g. be a memory-bound and a compute bound application.
Within this context, it might also be necessary to dynamically migrate applications between nodes if e.g. a new application is scheduled to the system. In order to be able to monitor performance and energy efficiency during operation, additional sensors are required. These need to be correlated to running applications to deliver values for key performance indicators.

COSH Workshop Topics:

- co-scheduling concepts and challenges,
- modeling of performance and energy efficiency,
- application classification regarding resource demands,
- task migration for workload balancing,
- case studies in co-scheduling.

Technical Program Committee

Georgios Goumas, National Technical University of Athens, GR
Stefan Lankes, RWTH Aachen University, DE
Thomas Lundqvist, University West, SE
Konstantinos Nikas, National Technical University of Athens, GR
Carsten Trinitis, Technische Universität München, DE
Josef Weidendorfer, Technische Universität München, DE
Andreas de Blanche, University West, SE

(This page intentionally left blank)

Co-Scheduling in a Task-Based Programming Model

Thomas Becker
Chair for Computer Architecture and Parallel Processing
Karlsruhe Institute of Technology, Germany
thomas.becker@kit.edu

Dai Yang, Tilman Küstner, Martin Schulz
Chair for Computer Architecture and Parallel Systems
Technical University of Munich, Germany
d.yang@tum.de, {kuestner, schulzm}@in.tum.de

ABSTRACT

The rising on-node concurrency, combined with limited resources, makes it increasingly hard for a single application to exploit the computational power of a node. Co-Scheduling, i.e., the concurrent use of a single node by two or more complementary applications, can help mitigate this situation and achieve higher efficiency.

In this paper, we propose an extension to HALadapt, a library for task-based programming, which leverages its dynamic profiling approach to provide concurrency throttling and combines it with its ability to coordinate execution across multiple runtime instances. Using a real-world example application, MLEM, co-scheduled with a compute-intensive synthetic workload *stressgen*, we show that the runtime system of HALadapt can efficiently coordinate multiple independent instances on a single node, leading to a performance improvement of up to 43% in the best case.

1 MOTIVATION

Efficient allocation of computational resources is a major challenge for High Performance Computing (HPC) systems. While today's systems are mostly scheduled on a node-basis, i.e., entire nodes are allocated to applications, this strategy may not fit with future systems as the degree of on-node parallelism continues to increase. Many existing applications are not ready for such an increased degree of intra-node parallelism, which limits their scaling. To combat this challenge, prior work proposed a new approach, called *Co-Scheduling*, which aims at scheduling multiple, different applications on the same node simultaneously. This approach works best, with minimal interference between the applications, if these applications have complimentary resource requirements, e.g., one of the applications is memory-bound and another one is compute-bound. In order to reach this desirable outcome, we need 1) a way to characterize these applications accordingly, and 2) a mechanism to coordinate their scheduling at runtime.

In this work, we tackle both challenges by extending the task-based runtime system HALadapt [15]. Using its profile-guided scheduling approach, we first implement a form of concurrency throttling, which enables the runtime system to pick the right level of concurrency. We then rely on HALadapt's queuing mechanism that enables the coordination across independent runtime instances to ensure the efficient execution of multiple co-scheduled applications. In particular, we make the following contribution:

- We extend the task-based runtime HALadapt to support profile-guided dynamic concurrency throttling.
- We combine concurrency throttling with HALadapt's inter-application scheduling support.
- We show that HALadapt's runtime system can efficiently coordinate multiple independent instances in a single node.

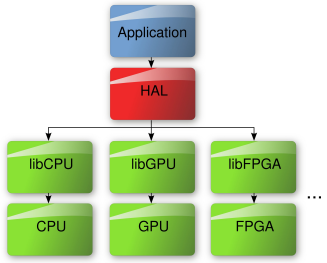

Figure 1: Library-based approach for implementations in HALadapt additionally working as intermediate layer between hardware and application

Using the MLEM application , a real-world workload from the area of medical imaging, as an example to evaluate our approach, we co-scheduled it with a compute-intensive load to evaluate our contributions. We show a) that we can efficiently execute both workloads together as a single task graph with a 71% improvement over the naive baseline, and b) up to 43% improvement when scheduled as two separate HALadapt instances.

The remainder of this paper is structured as follows. We describe the task-based runtime system HALadapt in Section 2 and our extensions made to support efficient co-scheduling in Section 3. We describe our experimental setup and the target applications in Section 4 and present our results in Section 5. We then describe related work in Section 6 and wrap up with conclusions and future work in Section 7.

2 THE HALADAPT RUNTIME SYSTEM

HALadapt [15] is a task-based runtime system implemented as a library. As illustrated in Figure 1, it is designed to abstract the underlying hardware and to separate application development from the actual implementation of individual kernel variants. The user simply defines his or her kernels representing a specific functionality, e.g., a task like a matrix multiplication, and provides either one or multiple implementation variants targeting different processing units and programming models. HALadapt currently supports OpenMP, OpenCL and CUDA kernels in addition to sequential implementations. To support system portability, variants are packaged into dynamic libraries, which are only loaded if all required dependencies are fulfilled. HALadapt does not impose a particular limit in terms of granularity for these kernels, leaving it to user preference. Tasks can be scheduled directly or collected in task graphs.

COSH2018, January 2018, Manchester, United Kingdom

https://doi.org/10.14459/2018md1428536

As selecting a suitable variant of a kernel in a given situation usually requires a complex decision process and is hard to determine for the user, HALadapt offers adaptive scheduling mechanisms [14]. These mechanisms use a history-based profiling approach to learn the characteristics of given kernel variants, like execution and data transfer times. A history database using problem size as the key stores these characteristics and can then be used to make decisions on which kernel to use by predicting runtimes for given problem sizes and target architectures.

As HALadapt is located in user space, it is by default only able to schedule kernels belonging to the same process. In order to benefit from the knowledge stored in the history database across applications, concurrent HALadapt instances can communicate via shared queues stored in a shared address space. These queues represent all available processing units on the node, allowing multiple HALadapt instances to notice if a processing unit is used by another process. This mechanism also enables efficient coordination between multiple applications in a co-scheduling environment, i.e., where they share resources instead of just queueing for them.

3 CONCURRENCY THROTTLING

In order to make use of this coordination mechanism and to allow multiple applications to share parallel resources (in our case cores in a multi-core system), we first need to extend the HALadapt scheduler to dynamically adapt the degree of parallelism and with that the number of used cores. In particular, we create a mechanism which can detect the scaling capability of OpenMP kernels and then create a fitting number of OpenMP threads and bind them to selected processing cores allowing kernel variants from concurrent applications to efficiently use the available system resources.

To enable this functionality, we extend the history-based profiling mechanism to also include the number of cores as part of the database key. This enables HALadapt to associate stored characteristics with the number of cores used and hence predict execution times for varying core numbers. These predictions can then be used as part of the scheduling mechanism to not only find the best variant of a kernel, but also the most suitable level of concurrency.

For further optimization and stability, our mechanism uses a minimal scaling factor: if the execution time does not improve at least by the minimal scaling factor when adding an additional processing core, no additional core is granted to the kernel. Additionally, we always select the variant, which minimizes the completion time of a kernel, similar to the HEFT scheduling algorithm [24].

For a submitted task graph potentially consisting of several dependent and independent kernels, our mechanism works as follows:

(1) The kernels included in the task graph are first sorted into a linked list respecting the given kernel dependencies and the user-defined call order.
(2) The kernels are then given to the scheduling mechanism in order of the sorted list.
(3) For every kernel the mechanism selects the variant configuration with lowest predicted completion time. If an implementation candidate adds an additional core, the predicted execution time has to improve by at least the scaling factor else the candidate is discarded.

When the mechanism has selected an implementation and an appropriate number of cores, HALadapt sets the number of OpenMP threads to be created accordingly and uses *hwloc* [6] to bind the threads to the selected processing cores.

4 TARGET WORKLOAD

We test our approach using a real-world workload, a 3D image reconstruction algorithm for positron emission tomography (PET), and co-schedule it with a synthetic compute-intensive load. All measurements are conducted on *sk1*, a dual-socket system with two Intel Xeon Scalable Silver 4116 (Skylake-SP) processors with 12 cores each and 32 GB of RAM. To simplify our initial scheduling mechanism, we ignore effects from the NUMA setup such as differences in memory latency and bandwidth. While this can lead to degraded performance, it is not critical to our contribution, which aims at demonstrating the possibility of reusing unused computational resources.

4.1 PET Image Reconstruction with MLEM

Our application, which we refer to as MLEM in the remainder of the paper, uses the Maximum Likelihood Expectation Maximization (MLEM) algorithm [20] to reconstruct 3D images from data obtained from Positron Emission Tomography (PET) scanners. PET is a functional imaging technique in nuclear medicine, in which radionuclide is injected into the subject's body, where it undergoes beta decay. After travelling a very short distance (typically less than 1 mm), the positron interacts with an electron. The result of this annihilation event is a pair of high-energy photons traveling in opposite directions which can be detected by a ring of scintillator crystals positioned around the subject.

Reconstructing a 3D image from the scanner readout (a list of detected events) is an inverse problem. While multiple (accelerated) algorithms exist, MLEM is commonly seen as the one providing the highest quality. The algorithm requires two inputs: the scanner readout and the system matrix. The system matrix describes all scanner properties in a concise way, like the spatial arrangement of the detectors and the physical effects during image acquisition. The matrix is sparse, as each pair of detectors can only detect events coming from a tube-shaped section of the full field of view of the scanner. The system matrix can be obtained in different ways: direct measurement using a probe, analytical models like the detector response function (DRF) model [21], or Monte Carlo simulation of the imaging process. For our application example, we use a system matrix describing the small animal PET scanner Madpet-II [19], generated by the DRF model. The matrix has around 1.6×10^9 non-zero elements and consumes around 12.8 GB of memory stored in compressed sparse row (CSR) format.

The MLEM algorithm starts with an estimate of the image which it improves in each iteration. Per iteration, the algorithm performs the following steps:

(1) Calculate the estimated scanner readout by using the current image approximation. This forward projection corresponds to an SpMV of the system matrix A and the image vector.
(2) Calculate a correction vector by correlating the estimated readout with the actual readout. This is an element-wise vector operation.

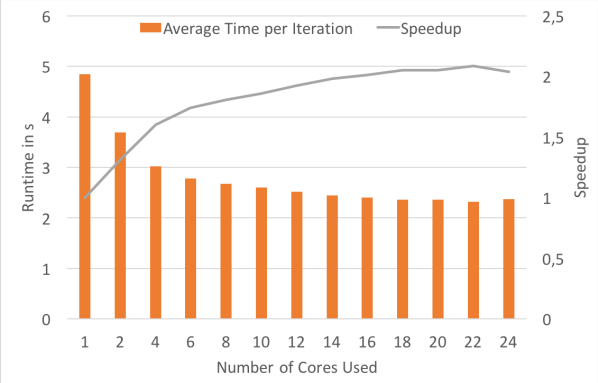

Figure 2: Native OpenMP MLEM scaling behaviour on *sk1* without HALadapt

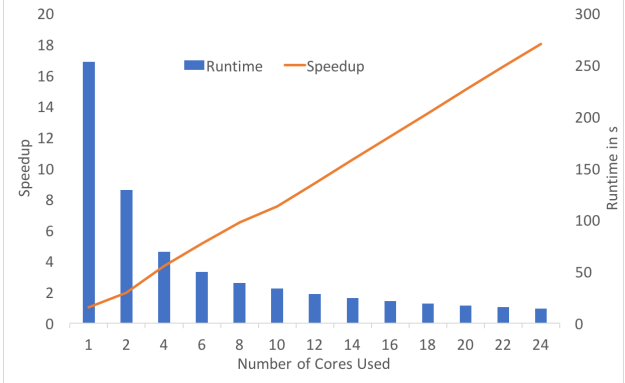

Figure 3: Native Performance of our stressgen code on *sk1* without HALadapt

(3) Transfer the correction factor into the image domain. This is called back-projection and corresponds to an SpMV of the transposed system matrix A^T and the correction vector.
(4) Apply the correction factor and a normalization to the image in order to obtain a new image estimate. This is again a vector operation.

Although the transposed system matrix is required in the back-projection step, we do not actually have to store the transposed matrix, as $y = A^T x \Leftrightarrow y^T = x^T A$. All calculations are performed in single precision. The two SpMV operations dominate the vector operations, so that the time per iteration is given by the speed of the SpMV operations. We derived our implementation from an MPI-parallelized version of MLEM developed earlier [16]. The SpMV operations are parallelized by dividing the sparse matrix into groups of rows containing approximately the same number of non-zero elements. Each group is then assigned to a HALadapt thread. Due to this approach, there is only one communication step (a reduction) at the end of each iteration.

In prior work we have shown that the original MPI-based MLEM code is memory-bound [27] and this behavior also applies to the HALadapt/OpenMP version of MLEM used here. Figure 2 shows the runtimes of MLEM for different fixed numbers of OpenMP threads, set using `OMP_NUM_THREADS`. The results show that the speedup does not increase significantly when run on more than six cores. The overall performance of the native OpenMP version of MLEM is similar to the MPI version [27].

4.2 Compute Intensive Application stressgen

To test our co-scheduling approach, we use a simple, compute intensive kernel (stressgen) to be scheduled concurrently with MLEM. The code for this kernel is shown in Figure 4, which takes it base idea from the standard Linux tool *stress*[1]. In order to reduce the memory footprint and to fit into the L1 cache of sk1 to a minimum, we chose a vector of 3500 double values. Figure 3 shows the scalability of stressgen alone.

[1]https://people.seas.harvard.edu/~apw/stress/

```
void cla(double* p, int length){
#pragma omp parallel for schedule (dynamic, 100)
        for(int i=0; i<length; i++){
        for(int j=0; j<5000000; j++)
                p[i] = sqrt(p[i]);
                p[i] = pow(p[i], 2);
    }
}
```

Figure 4: Compute Kernel for the stressgen

4.3 Profile Training

In order to test our modified HALadapt version, we first need to complete the profiling step while varying the number of cores. In our current setup, we have a static profiling process, which is implemented by calling the execution kernel 24 times, with one additional core for each kernel each time. We consider all four compute kernels for the MLEM code and the one for stressgen, but we then discard the kernels `calcCorrel_CPU` and `calcUpdate_CPU` due to their short runtime. While, this profiling step can lead to a high overhead, especially for long running compute kernels, it is typically compensated by the fact that the same application is executed many times in a similar setup and environment, as it is common for HPC codes. Improvement to this profiling process, such as using hints on resource requirements for applications provided by the programmer or using interpolation and performance prediction, are currently under development.

5 RESULTS

In the following we show experimental results of running MLEM in combination with the stressgen kernel. For this we port MLEM to HALadapt, which mainly entails switching from object data structures to regular arrays. These changes have some small impact on runtimes when comparing OpenMP and HALadapt executions, but these differences do not affect the results.

As MLEM has a lower execution time than stressgen, we run 10 MLEM iterations, which matches the real-world set up for MLEM. As the baseline, we use a combination of the fastest MLEM and stressgen executions which means both codes use all 24 processing

Table 1: Total execution time of the MLEM and stressgen co-scheduled in one instance

	Baseline	One HALadapt instance
Total execution time	40.44 s	23.61 s
Speedup	1x	1.71x

Table 2: Total execution time of the MLEM and stressgen co-scheduled in two instances

	Baseline	Two inst. sc.1	Two inst. sc.2
Total execution time	40.44 s	28.28 s	54.54 s
Speedup	1x	1.43x	0.74x

cores and are executed successively. This results in a total execution time of 40.44 s as the sum of 10×2.36 s for ten MLEM iterations and 16.84 s for the stressgen with 24 threads, as measured on the sk1 system described in Section 4, which we use for all experiments.

5.1 Effectiveness of Co-Scheduling with HALadapt in a Single Task Graph

The first set of experiments combines the MLEM application and the stressgen kernel in a single task graph and then uses a single HALadapt instance to execute it, thereby allowing a cooperative scheduling of the two kernels. To reduce the impact of variation, we execute the graph ten times and report the average execution time for the whole graph.

Using this setup, the integrated scheduling mechanism created the following schedule (as shown in Figure 5 (left)):

- MLEM is separated into four kernels with only the forward and backward projection having an OpenMP implementation. The other kernels are executed sequentially as described in Section 4.1.
- HALadapt mapped the forward projection to the first four processing cores and the backward projection to the first six.
- The stressgen kernel was mapped to the last 17 processing cores.
- One core is left idle.

The resulting schedule for MLEM matches the results from the native execution in Figure 2, which also shows a scaling limitation when using more than six threads. The results of the combined schedule in Table 1 show that the co-scheduling mechanism reduces the execution time (which includes the overhead of the scheduling mechanism) significantly and is able to reach a speedup of 1.71. The core left idle is mainly due to the minimal scaling factor of 10%, which we chose in this paper. This setting means that if the application cannot scale more than 10% with an additional core, it will not get more resources. This limit is reached after six cores for MLEM and 17 cores for stressgen.

5.2 Effectiveness of Co-Scheduling with HALadapt in Disjoint Processes

In the second experiment, we run the two codes, MLEM and stressgen, in two different HALadapt instances. As mentioned before, HALadapt is able to share the waiting queues of the abstracted processing units with other HALadapt instances, thereby allowing cooperative scheduling. The different HALadapt processes get started with a slight delay to ensure an ordered scheduling.

The problem in this scenario is that we are not able to measure the kernel time separately. Instead, we have to measure the execution time of the complete processes including all data initialization and setup for the HALadapt runs, which reduces the effectiveness. For the second scenario, the mechanism created the following schedule:

- HALadapt maps the forward projection to the first four cores;
- the backward projection to the first five processing cores;
- the stressgen kernel to the last 17 processing cores again;
- leaving effectively two cores idle.

Overall, this execution ends up using one core less than in the single instance scenario. This is because in this experiment the data initialization and setup had to be included in the measurements of the HALadapt runs limiting the scaling behaviour of stressgen. The measured execution times (average of 10 runs) are listed in Table 2. Due to the static scheduling at the beginning of the task graph launch, the order in which the two task graphs are launched matters. In the scenario in which the stressgen kernel was scheduled first (scenario 2 in Table 2), it reserves 21 of 24 cores, leaving MLEM with only using the sequential kernels, increasing execution time by around 25%. If MLEM is started first, the schedule is as discussed above and shown in Figure 5 (right). In this scenario (cf. scenario 1 in Table 2), HALadapt is able to achieve a speedup of 1.43 compared to the baseline, which is less than in the single instance case. The scheduling order is crucial in this case because we have only implemented a simple heuristic, which tries to optimize the number of processing units being used for a single application without considering following applications. This means that if an application with a good scaling behaviour is scheduled first almost all cores will be reserved for this application leaving no resources for following ones. In addition to the differences in the schedule, a slight decrease in speedup can also be contributed to the fact that HALadapt instances now have to share the waiting queues of the processing units and accesses to wait queues have to be enforced to be mutually exclusive via locks thus increasing the overhead for creating a schedule. Since the time measurements we obtained always include all initialization and setup time from HALadapt, the speedup is reduced when compared to the fist experiment.

6 RELATED WORK

The concept of co-scheduling has gained traction in recent years. For example, Haritatos et al. [12] and Weidendorfer et al. [4] discuss classifications of application based on their resource usage and Bhadauria et al. [3] and Haritatos et al. [11] show first achievements in improving performance and energy efficiency using co-scheduling. Breitbart et al. [5] provide a detailed case study of co-scheduling scenarios in an HPC environment. Süß et al. [22] describe extensions for resource-aware workload scheduling and discuss the impact of co-scheduling on applications and De Blanche et al. [8, 9] present ways to ensure co-scheduling effectiveness. Further work focused on techniques for active resource partitioning to

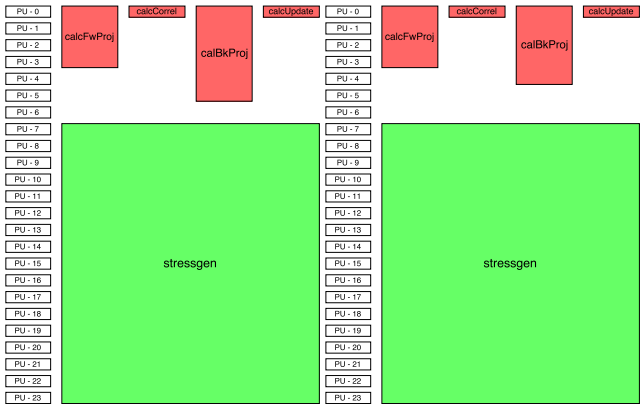

Figure 5: Left: Mapping result of the first experiment; Right: Mapping result of the second experiment

minimize application impact, such as cache partitioning discussed by Papadakis et al. [18] and Weidendorfer et al. [26].

Runtime systems for task-parallel systems for heterogeneous systems is also a growing field in research. One of the prominent representatives is StarPU [1]. As HALadapt, StarPU abstracts the underlying hardware and separates the application from its implementation. For a given task it selects the best implementation according to a desired scheduling algorithm. Although StarPU supports OpenMP, it does not analyze the scaling behavior of tasks to optimize the number of processing cores being used. The same is true for Nanos++ [23], a runtime system which is used to implement OmpSs [10], a programming model close to OpenMP with support for accelerators, and the Open Community Runtime (OCR) [17], which is specifically designed to support exascale systems. Legion [2] is a runtime system for parallel architectures, that detects tasks by analyzing the data organization of an application through so-called logical regions. In addition, Legion offers a mapping interface that enables users to implement their own mapping scheme along with a default mapper that selects the fastest implementation and allows task stealing, but no automatic concurrency variation.

The concept of concurrency throttling, i.e., varying the number of threads or adapting the number of processing cores being used, has been adopted also in other runtimes. Wang et al. [25] evaluate the influence of varying the number of threads on an IBM's Power8 system using loop granularity. In contrast to our work, Wang et al. do this by hand and do not use a learning based mechanism. Curtis-Murray presents the runtime system ACTOR [7], which is able to optimize the number of threads being used by predicting the resulting performance and choosing the most efficient thread count. In contrast to our method, ACTOR is limited to a single application whereas we can either include several applications as tasks in a single task graph or can cooperate between several HALadapt instances.

Another approach to optimizing the number of threads, and consequently the number of processing cores being used, is autotuning. Karcher et al. [13] propose an online autotuner called Perpetum, which is able to tune parameters like the number of threads being used of competing multicore applications at runtime. The drawback of this method is that the user has to know and mark the computational hotspots of an application him- or herself. The user also has to tell the autotuner which parameters to tune and state a parameter range. Both requirements add an additional burden on the user and lead to inefficiency due to missed tuning opportunities.

7 CONCLUSION AND FUTURE WORK

In this work, we show that task-based programming models can benefit from co-scheduling approaches. We build on the existing HALadapt runtime system, which already includes mechanisms to share individual node resources between different tasks across multiple processes and we extend it using a scheduling mechanism that optimizes the number of cores reserved by a kernel by considering its resource requirements and scaling properties. These two features combined enable a novel mechanism for efficient sharing of all cores on a node across multiple independent applications.

To achieve this, we first extend HALadapt's profiling mechanism, used in the base runtime to enable history driven predictions of task implementations, to be core-aware. This allows HALadapt to monitor the runtime for a varying number of cores and to use the number of cores as a key in its performance database. We then added a new scheduling heuristic that only adds a new core to an OpenMP kernel if the runtime improved by a user configurable, yet static scaling factor. Overall, this allows 1) HALadapt to characterize the scaling properties of individual kernel implementations and with that to adjust the level of concurrency as needed and 2) HALadapt instances to coordinate to ensure minimal interference during co-scheduling. Our experiments show that, when combining a memory and a compute bound application, our system can provide up to 71% improvement in a single HALadapt instance and up to 43% in two separate instances compared to a non co-scheduled baseline.

Our experiments, however, also show cases in which the co-scheduling heuristics fail by scheduling applications in the wrong order, leading to an overall slowdown. This can be mitigated using a more complex scheduling mechanism: instead of making local decisions for kernels in order of a sorted list, a random search-based algorithm could be used to overcome local minima and optimize the task graph on a global level.

A second area of improvement is a reduction of the necessary profiling overhead. The current implementation status requires profiling runs for all possible number of cores, therefore increasing the number of necessary runs linearly with the number of available cores. We can decrease this overhead by only conducting profiling runs for a specific set of numbers of cores and then using interpolation or curve fitting to predict intermediate data points, although this can potentially decrease prediction accuracy. Other options are incremental refinements of predictions at runtime or a more aggressive pruning of the search tree in low-performing regions of the search space.

Finally, we plan on using HALadapt's capabilities to abstract the hardware in heterogenous systems and extend also these features to improve co-scheduling. In particular, applications written using HALadapt can provide multiple variants for different hardware types (e.g., Multicores, GPUs or FPGAs) for each computational kernel. We can use this to expand the possible scheduling options and with that further improving throughput of HALadapt applications.

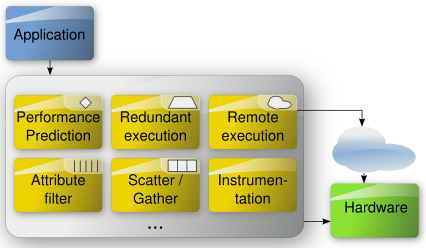

Figure 6: All services in HALadapt are separated from each other and can be used and modified independently

As HALadapt is implemented in a modular fashion and all these modifications can be added without a need to change the underlying architecture. Services like scheduling and profiling are organized as so-called call stack entities (Figure 6) and can be selected by the user independently of other call stack entities. Consequently, a modification of one call stack entity does not require changes in HALadapt's base architecture or other call stack entities, providing us with an extensible framework to implement and evaluate co-scheduling in HPC systems.

ACKNOWLEDGMENT

This work was funded by German Federal Ministry of Education and Research (BMBF) under grant title 01IH16010D (Project ENVELOPE).

REFERENCES

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. https://hal.inria.fr/inria-00550877

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 66.

[3] Major Bhadauria and Sally A McKee. 2010. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 189–199.

[4] Jens Breitbart and Josef Weidendorfer. 2017. Detailed Application Characterization and Its Use for Effective Co-Scheduling.. In *Co-Scheduling of HPC Applications*. IOS Press, 69–94.

[5] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. 2015. Case study on co-scheduling for HPC applications. In *Parallel Processing Workshops (ICPPW), 2015 44th International Conference on*. IEEE, 277–285.

[6] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. Hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP '10)*. IEEE Computer Society, Washington, DC, USA, 180–186. https://doi.org/10.1109/PDP.2010.67

[7] Matthew Curtis-Maury. 2008. *Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems*. Ph.D. Dissertation.

[8] Andreas De Blanche and Thomas Lundqvist. 2016. Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedule. In *COSH Workshop on Co-Scheduling of HPC Applications HIPEAC 2016*, Vol. 1. 1–6.

[9] Andreas de Blanche and Thomas Lundqvist. 2017. Disallowing Same-program Co-schedules to Improve Efficiency in Quad-core Servers.. In *COSH/VisorHPC@ HiPEAC*. 13–20.

[10] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 02 (2011), 173–193.

[11] Alexandros-Herodotos Haritatos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios Koziris. 2014. LCA: A memory link and cache-aware co-scheduling approach for CMPs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 469–470.

[12] Alexandros-Herodotos Haritatos, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2016. A resource-centric application classification approach. In *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*. 7.

[13] Karcher, Thomas and Pankratius, Victor. 2011. *Run-Time Automatic Performance Tuning for Multicore Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–14. https://doi.org/10.1007/978-3-642-23400-2_2

[14] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. 2011. Cost-aware Function Migration in Heterogeneous Systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11)*. ACM, New York, NY, USA, 137–145. https://doi.org/10.1145/1944862.1944883

[15] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. 2012. Seamlessly Portable Applications: Managing the Diversity of Modern Heterogeneous Systems. *ACM Trans. Archit. Code Optim.* 8, 4, Article 42 (jan 2012), 20 pages. https://doi.org/10.1145/2086696.2086721

[16] Tilman Küstner, Josef Weidendorfer, Jasmine Schirmer, Tobias Klug, Carsten Trinitis, and Sybille Ziegler. 2009. Parallel MLEM on Multicore Architectures. In *ICCS 2009: 9th Int. Conf. on Computational Science*, G. Allen et al. (Ed.). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-01970-8_48

[17] T. G. Mattson, R. Cledat, V. CavÃĺ, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2016.7761580

[18] Ioannis Papadakis, Konstantinos Nikas, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2017. Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning. In *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*.

[19] Magdalena Rafecas, Brygida Mosler, Melanie Dietz, Markus Pögl, Alexandros Stamatakis, David P. McElroy, and Sibylle I. Ziegler. 2004. Use of a Monte Carlo-Based Probability Matrix for 3-D Iterative Reconstruction of MADPET-II Data. *IEEE Trans. on Nuclear Science* 51, 5 (2004). https://doi.org/10.1109/TNS.2004.834827

[20] L. A. Shepp and Y. Vardi. 1982. Maximum Likelihood Reconstruction for Emission Tomography. *IEEE Transactions on Medical Imaging* 1, 2 (1982), 113–122. https://doi.org/10.1109/TMI.1982.4307558

[21] D. Strul, R. B. Slates, M. Dahlbom, S. R. Cherry, and P. K. Marsden. 2003. An improved analytical detector response function model for multilayer small-diameter PET scanners. *Physics in Medicine and Biology* 48 (2003), 979–994. http://www.ncbi.nlm.nih.gov/pubmed/12741496

[22] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Eric Schricker, and Thomas Soddemann. 2016. Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling.. In *COSH@ HiPEAC (extended versions)*. 142–162.

[23] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. 2007. Support for OpenMP tasks in Nanos v4. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. IBM Corp., 256–259.

[24] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (Mar 2002), 260–274. https://doi.org/10.1109/71.993206

[25] Wei Wang and Edgar A León. 2015. Evaluating DVFS and Concurrency Throttling on IBM's Power8 Architecture. In *SC15: The ACM/IEEE International Conference for High Performance Computing Networking, Storage, and Analysis*. IEEE.

[26] Josef Weidendorfer, Carsten Trinitis, Sebastian Rückerl, and Michael Klemm. 2017. Cache-Partitionierung im Kontext von Co-Scheduling.. In *Konferenzband des PARS Workshops*. Gesellschaft für Informatik.

[27] Dai Yang, Josef Weidendorfer, Tilman Küstner, Carsten Trinitis, and Sibylle Ziegler. 2017. Enabling Application-Integrated Fault Tolerance. *Advances in Parallel Computing, Conference Proceeding of PARCO* (2017). Accepted for Publication.

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 (3^{rd} 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.

COSH2018, January 2018, Manchester, United Kingdom

https://doi.org/10.14459/2018md1428537

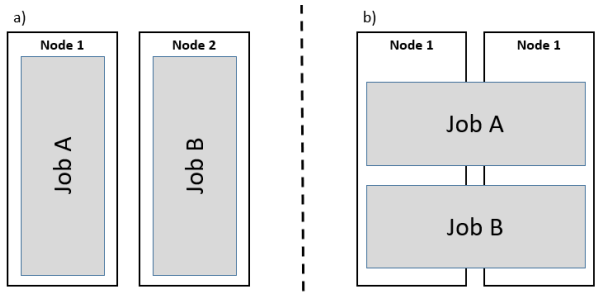

Figure 1: Example of different scheduling strategies on 24-core nodes. In a) job A is executing on 24 cores in node 1 and B is executing on 24 cores in node 2. In b) the two jobs are co-scheduled over two nodes, each job is using 12 cores in each machine.

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%

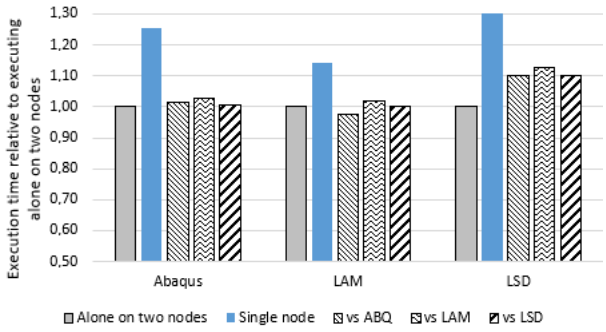

Figure 2: The impact of co-scheduling. All jobs but Single node are executed over two nodes. All numbers are relative to the execution time of a job executing alone over two nodes.

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.

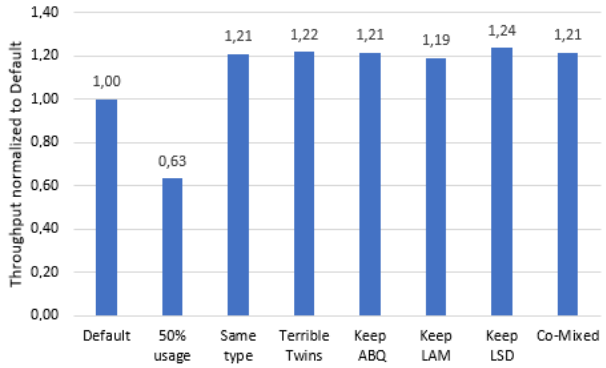

Figure 3: Throughput comparison between the eight different co-scheduling strategies on a 100% utilized cluster

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

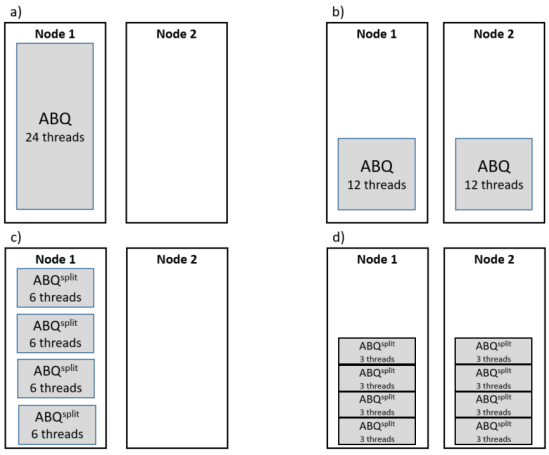

Figure 4: The four images a-d illustrates how a 24-process job of ABQ and ABQ distributes its MPI processes and threads when executing on a single node or over two nodes.

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%

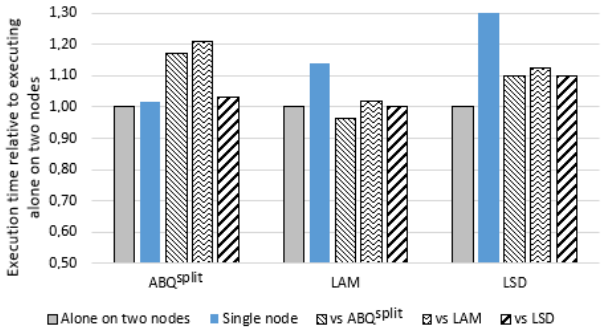

Figure 5: The impact of co-scheduling. All jobs but *Single node* are executed over two nodes. All numbers are relative to the execution time of a job executing alone over two nodes.

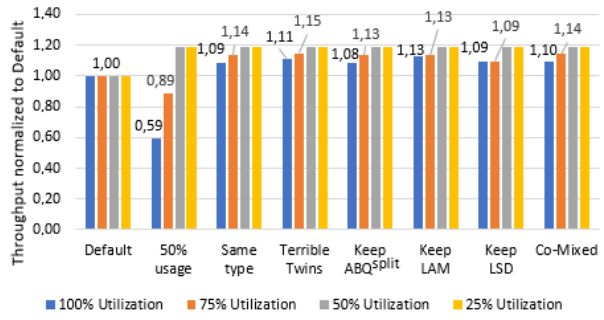

Figure 6: Throughput comparison between the eight different co-scheduling strategies. At four different cluster utilization levels.

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.

A Case Study for a New Invasive Extension of Intel's Threading Building Blocks *

Martin Schreiber
Department of Computer Science / Mathematics,
University of Exeter
EX4 4QF Exeter, Great Britain
M.Schreiber@exeter.ac.uk

Tobias Weinzierl†
Department of Computer Science,
Durham University
DH1 3LE Durham, Great Britain
tobias.weinzierl@durham.ac.uk

ABSTRACT

We study codes deploying multiple MPI ranks to one node where each rank is parallelised with TBB. A static assignment of cores to ranks here is disadvantageous if the load is not perfectly balanced, the runtime is subject to fluctuations or one MPI rank runs through phases with low concurrency. We propose an extension to TBB where developers manually annotate which code parts could exploit further cores. The cores are then dynamically associated with ranks. Our approach is decentralised, lightweight and minimally invasive w.r.t. code modifications. Some brief performance studies suggest that a flexible, permanently changing assignment of cores to compute ranks can outperform a static distribution, while greedly haggling over cores throughout a simulation might perform even better.

KEYWORDS

MPI+X load balancing, Invasive computing, Compute balancing, Dynamic resource scheduling

1 INTRODUCTION

With clock rates in supercomputers plateauing, future generations of high-end computers will obtain their unprecedented capabilities from an increase of the number of cores that are integrated into every single node. Their nodes are small systems on chips. Though some roadmaps and funding calls demand for radical re-writes of our simulation software such that the codes exhibit omnipresent concurrency, we do believe that many supercomputer users will "simply" decompose machines with many cores per node logically into machines with many nodes and fewer cores. They will deploy multiple MPI ranks per node and assign each rank a subset of the available cores per node, as they want to continue to use their multi-decade legacy codes which cannot be rewritten from scratch. At the same time, supercomputing codes will retain code fragments with nested fork-join (NFJ) patterns as well as MPI synchronisation points. They will continue to run through phases with limited concurrency. Without novel node usage paradigms, many scientists will thus struggle to exploit the full potential of new supercomputers with their existing codes. A novel paradigm however has to come along with marginal code modifications. Otherwise, it will struggle to become accepted. We propose such a paradigm for Intel's Threading Building Blocks (TBB).

Classic load balancing uses cost models and runtime measurements to determine homogeneous workload distributions. It minimises runtime differences between ranks. This mitigates but does not eliminate inefficient node usage patterns which arise inevitably from thread-based NFJ and MPI synchronisation, i.e. once we assign each rank a fixed (and usually the same) number of cores to be exploited via multithreading. If sequential or low-concurrency phases per rank remain, even load-balanced codes continue to run into phases where few cores of a node are utilised. Temporal ill-balancing is amplified by algorithmic fragments with unpredictable workload (localised Newton-solves, e.g.), IO scattered among the nodes, or performance fluctuations caused by vector instructions, e.g. We focus on the classic time stepping from the ExaHyPE code [3] in this manuscript. ExaHyPE uses adaptive mesh refinement (AMR) with multipe MPI ranks, each parallelised with TBB, per compute node. Here, cores inevitably run into waits for other ranks per time step, while some internal program parts fork threads and join them again: Large fragments of each time step exhibit an enormous concurrency while the remainder scales only up to few cores. No matter how sophisticated the load balancing, a fixed association of cores to ranks is inappropriate at certain program phases.

The present paper follows a paradigm shift. Instead of assigning work to fixed rank-core assemblies, we make cores dynamically join the ranks that most need additional compute power. The core-rank association follows the workload, i.e. data distribution, and the core per rank ratio changes over time. From a programming model point of view, such a paradigm makes ranks invade cores when compute resources are needed and they retreat from these cores afterwards to free resources for other ranks. It falls into the class of invasive programming [9]. While we focus solely on one MPI code, all techniques also work if different applications run concurrently.

Our approach is minimalist: We disable features such as masking and make each rank running on one node occupy all hardware threads. If hyperthreading is disabled, physical threads correspond to cores. We thus use the terms as synonyms. For R ranks per node with C cores, each core is overbooked as we launch in total $R \cdot C$ (logical) threads. All the ranks on one node agree through a shared memory region, a scratchpad, which cores may be used by which

*This work received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). It made use of the facilities of the Hamilton HPC Service of Durham University.
†Corresponding author

COSH2018, January 2018, Manchester, United Kingdom

https://doi.org/10.14459/2018md1428538

rank for their work. This establishes a lightweight communication channel between ranks. Once a rank has obtained c cores for work, it spawns $C - c$ lock tasks. Lock tasks make their designated threads sleep. This effectively releases the core for other ranks to perform their work. Invasion of cores in this model corresponds to a termination of the corresponding lock tasks.

Invasive programming and invasive code ingredients have been proposed before (see e.g. [2, 6, 8]). The present integration of a straightforward, lightweight, scratchpad-based, decentralised, and easy to use invasive extension of TBB into existing C++ codes is, to the best of our knowledge, however new. We believe that the proposed approach can help scientists to exploit manycore architectures more efficiently. For the very first time, standard load balancing is given a powerful and lightweight assistant: Our programming concept allows for an easy-to-use and fine granular adaption of computing resources throughout all phases. Communication overheads of a centralised resource manager are eliminated by the utilisation of a inter-program shared scratchpad.

The remainder is organised as follows: We start from a description of our minimalist application programming interface (Sec. 2) before we highlight the technical realisation. Both ingredients allow us to compare the proposed solution to existing approaches (Sec. 4). We quickly sketch the integration of our new techniques into an existing solver (Sec. 5) hereafter before some preliminary performance studies uncover the techniques' potential. A brief wrap up and outlook in Sec. 7 close the discussion.

2 APPLICATION PROGRAMMING INTERFACE

Our application programming interface (API) picks up idioms of TBB. The proposed API itself is implemented on top of TBB as an additional layer. An existing TBB parallelisation acts as starting point. We require the user to instantiate a class `SHMInvade` whenever additional cores would be of use, i.e. a user has to insert further code statements.

The construction of `SHMInvade` checks the scratchpad content for additional cores to "help out" on the computation. Here, the number of additionally requested cores is used as an upper bound for the new cores to be invaded. These additional cores are released automatically by the destructor of the `SHMInvade` object. Alternatively, users may manually free the cores before. The concept mirrors the realisation of TBB's `scoped_lock`.

```
foo(); // ran on default number of cores
/// try to get up to three additional cores
SHMInvade invade(3);
tbb::parallel_for ...{
  if (i==0) {
    // try to book up to two more cores
    SHMInvade invade(2);
    bar();
  } // release two more cores
  ...
}
```

Our approach

(1) neither requires the user to alter the original TBB code,
(2) nor does it require the user to keep track of how many cores actually have been grabbed (invaded)—this information is stored internally within the `SHMInvade` object,
(3) nor does it put any constraints on the invasion cardinality.

For expensive interior operations (`bar()` in the example above), it might be advantageous to book cores on short note and then to release them immediately, as other ranks running in SPMD mode might benefit from released compute resources. Overall, it however also is possible to work with very few `SHMInvade` objects held on an outer loop level: our demonstrator code for example can invade cores prior to the time stepping loop, and it releases those cores once this loop has terminated and the rank enters an MPI communication phase. The invasive infrastructure supports both granularity paradigms and any hybrid combination of them.

Our invasive realisation offers a robust, reliable resource management: Resources are physically allocated and freed by individual applications through instances of `SHMInvade`. As the ownership of compute data is encapsulated within the object, we can ensure that no core-to-rank assignment is corrupted. We also foster that programmers release resources on time and no instance behaves greedily. While our invasive realisation is minimally invasive w.r.t. programming—code is solely augmented—our work realises two design criteria which require developers to revise and rethink their solutions:

- Algorithm-aware programming: The application has to be aware where its algorithms run into arithmetically challenging phases. Such an awareness drives the sophisticated insertion of `SHMInvade` instances as it is not for free to instantiate this object.
- Resource-aware programming: Each application is responsible to take care of the resource demands of the other ranks running on the same node. A sophisticated application optimises the overall resource utilisation locally per rank under consideration of the performance and resource requirements of other ranks.

3 REALISATION

Once a rank starts up on a node, it issues TBB threads on all of the node's cores. In a second step, it establishes access to a scratchpad, a common memory region used by all threads. From hereon, the `SHMInvade` objects take over control as they argue through the scratchpad whether additional cores can be invaded, or they notify the other ranks through the scratchpad if cores become available.

3.1 Lock tasks and work stealing

We invert the actual invasion procedure: Ranks do not actively invade cores, but they actively retreat from cores. We assume that TBB's workstealing keeps all C threads busy by default. If a rank has to retreat from a core—it retreats from all cores besides its "master" one at startup—it spawns a lock task and marks, protected by a system mutex, the thread to be freed. Lock tasks are scheduled through TBB's `enqueue` command and thus are starvation resistant. If TBB's work stealing issues a lock task on a misfitting hardware thread, this lock tasks reschedules itself and yields immediately. If a lock task is ran on its corresponding thread and the thread is marked to be freed, the tasks sends the executing thread to sleep.

If a lock task is ran on its corresponding thread and the thread is to be used, the lock task terminates. Waking up threads is thus accomplished by unsetting the marker for a particular thread. This eventually terminates a lock task. Oversubscription of cores on purpose is not supported yet by our approach.

3.2 Scratchpad

We use a POSIX shared memory object as scratchpad for the ranks. Each process allocates a shared memory mapped region using `shm_open(...)`, `ftruncate(...)` and `mmap(...)`. Such shared memory objects behave as in-memory shared files between the MPI ranks. Using `ftruncate` to resize the memory object assures that the object is initialised with 0-values in case that it wasn't initialised before. An additional field `is_initialized` indicates that the data in the shared memory region was not initialised yet and allows the first rank creating the object to set up all data appropriately.

Besides a `spin_lock` used to lock the content of the shared memory block, the scratchpad holds some global properties (such as the number of known cores administered through it) plus a table of all cores on this machine and a table of all ranks (see Table 1).

At startup, each rank acquires a unique index within the shared memory region. The indices range from 0 to $R-1$.

```
(1) num_active_threads = 0
(2) lock()
(3)  reg_process_pids[counter_reg_processes] = PID
(4)  reg_process_cores[counter_reg_processes] = 0
(5)  user_data_per_process[counter_reg_processes] = {0,
     ..., 0}
(6)  counter_reg_processes++
(7) unlock()
```

Each rank with process ID `pid` corresponds to one entry within `reg_process_cores`. To invade a core, we lock the shared memory region, and we determine how many cores are globally available (`num_free_cores`). Once determining the number of cores to invade, the fields `num_free_cores` and `reg_process_cores` are updated. This is accomplished within the locking phase to avoid race conditions. After this, the access to the shared memory region is unlocked. All other TBB-related actions (see previous section) are executed after unlock the shared memory region. Retreating from a core works in a similar way.

Overall, the scratchpad avoids a centralised resource manager [2, 6, 8]. No master process exists and the applications have to bargain amongst themselves for the best resource allocation. This removes the overheads of communicating the master process as well as starting and scheduling the runtime of the master process.

We close this section with a brief discussion on the scalability limits of locking the shared-memory region. The scalability limitations for this depends on the overheads of the mutual exclusive access, the time to read/update the shared memory region and the number of processes which perform this in parallel. We used spinlocks for their low overheads compared to system mutices. This keeps the serial overheads low. We assure that the invasive operations are only used to invade/retreat cores and that user-space operations are only allowed to block-wise load and write data during a spinlock phase. Finally, we focus on commodity multicore chips, only. Additional hardware-related scalability limitations such as false-sharing and NUMA effect might arise but were not further investigated in the present work but might lead to unexpected behaviour.

3.3 User data exchange

Our approach technically relies on a greedy invasion of compute cores. We outsource a responsible and sustainable usage of cores to the users. As such, it is convenient to offer an additional table in the shared memory regions, which developers can use to exchange properties between the different ranks on one compute node. The unique index per node from $\{0,\ldots,r-1\}$ identifies which rank is allowed to write into which row of such a table, while all ranks may read all entries.

4 RELATED WORK AND CONTEXT

Tailoring and optimising the assignment of computing resources to processes is not new, dynamic, i.e. online approaches are rare though. We notably refer to work published under the umbrella of [9] for related work. Reacting to changing resource requirements, i.e. to invade resources and to retreat from resources, can for example be realised through a centralised [2, 6] or distributed [7] resource manager. Our approach abandons the idea of any resource manager and instead proposed a marketplace where ranks can grab cores in a first-come first-served manner. It allows us to require users to augment their codes but not changing them while it puts the responsibility for sustainable resource usage on them. In practice, we consider it to be a convenient strategy in HPC, as we (i) assume that multiple ranks running on the same node are not perfectly balanced anyway, (ii) have to synchronise in regular time intervals, (iii) have NFJ source code fragments and (iv) run into close-to-serial communication phases regularly. If one rank thus invades, at one point, an inappropriate number of cores, it finishes earlier and frees these resources earlier. The total balancing smooths out over given time intervals such as time steps.

The two omnipresent HPC standards MPI and OpenMP lack support for such lightweight dynamic resource assignment. For recent MPI, adding new computing resources is part of the standard. Despite recent investigation of fully malleable MPI [4], it remains unclear how "cheap" such MPI splits are, what implications for the data transfer (message passing) arise, and whether current MPI implementations and supercomputer job schedulers support an aggressive growth of MPI processes. In OpenMP, a dynamic resource management was investigated in [2, 6, 8], e.g. All approaches however only support resource reallocation in rather outer loops. It is not possible to invade additional cores while other logical threads of the application run concurrently as the scheduler has to be restarted. Finally, we note that projects alike [3] investigate into work stealing in-between MPI ranks. If such a technique is applied in-between ranks deployed to the same node, it smoothes out load imbalances per node, too. Yet, all paradigms start from the motivation to balance the work given a certain hardware configuration. Our approach tailors the hardware configuration towards the actual runtime behaviour.

Type	Identifier	Description
`mutex`	`spin_mutex`	Spin mutex to avoid race conditions
`bool`	`is_initialized`	True if this data structure is already initialized
`int`	`max_available_cores`	Maximum number of possible threads ($\leq C$ (max cores)
`int`	`num_free_cores`	Current number of free cores ($\leq C$ (max cores)
`int` (atomic)	`counter_reg_processes`	Number of registered processes using SHMInvade
`pid_t`	`reg_process_pids[]`	PIDs of registered programs
`int`	`reg_process_cores[]`	Number of used cores for each program
`void`	`user_data_per_process[][]`	User-specific data for each process

Table 1: Global data structures with data to be filled in. All variables are declared as `volatile`, hence are not cached in registers but always read from memory.

Figure 1: Two screenshots of the two-dimensional Euler equations applied to an initial energy distribution derived from the ExaHyPE logo.

5 INTEGRATION

For our proof-of-concept, we investigate two different approaches. In the first approach, we make each rank start per time step from only one core and rely on an on-the-fly invasion to balance out an appropriate distribution of cores to ranks. In the second approach, we make all ranks running on one node to minimise their total runtime in a joint afford. The ExaHyPE engine [3] acts as testbed. We simulate the two-dimensional Euler equations with the ADER-DG explicit time stepping scheme [5] on dynamically adaptive grids (Figure 1). The software base is interesting for multiple reasons:

- The code runs on a dynamically adaptive grid where the grid changes in each and every time step.
- The ADER-DG scheme is a predictor-corrector scheme where a computationally expensive predictor phase is followed by a reasonably cheap non-linear Riemann solve for the discontinuities plus a correction step.
- The predictor solves the underlying equations with Picard iterations per cell: the cost per cell thus depends heavily on the iteration steps for the nonlinear problem. It varies in each and every time step.

ExaHyPE is built upon the Peano sources which is an open-source spacetree code. We add all `SHMInvade` invocations there, i.e. our invasion approach is agnostic of the actual application domain.

5.1 Invade throughout computation

This approach does not hold any `SHMInvade` permanent over extensive time. Instead, we focus on the code's underlying nested parallel fors and tasks. Prior to its task spawning and the loops, we insert `SHMInvade` commands.

We note that the code uses the same computational infrastructure for all different algorithmic phases, i.e. if we augment one loop by invasion commands, this augmentation applies to both predictor, Riemann solve and correction. The invasion does not take the computational intensity into account. Furthermore, no rank does directly interact with any other rank to decide whether it should try to invade cores: We may assume that only the coarser levels of the nested parallelisation succeed in invading cores. Yet, once one rank frees cores on a rather coarse level, other, very fine-granular, invasion attempts might pass through. No fairness policy is in place.

5.2 High-level integration

For our fair alternative approach, we start from the assumption that each rank faces its own strong scaling challenge w.r.t. the cores available to it. A modified Amdahl's law [1]

$$t_r(c_r) = f_r \cdot t_r(1) + (1 - f_r)\frac{t_r(1)}{c_r} + s_r \cdot c_r \tag{1}$$

yields a reasonable description of the scaling behaviour (Fig. 2) global behaviour of one rank. Yet, the values f_r (serial code fraction), $t_r(1)$ (serial runtime) and s_r (startup cost of the cores/threads) differ per rank $r \in \{0, \ldots, R-1\}$. Here, $1 \leq c_r \leq C$ is the number of cores/threads available to the rank. Each rank runs the same code (SPMD) but we run dynamic AMR and thus obtain different performance characteristics per node. Our invasive strategy consists of two steps per iteration of the underlying solver:

(1) Runtime analysis: Each rank tracks its own compute time relative to the cores that are available to it. This allows the rank to calibrate its $(f_r, t_r(1), s_r)$ quantities to its own behaviour. It runs its own online machine learning algorithm.

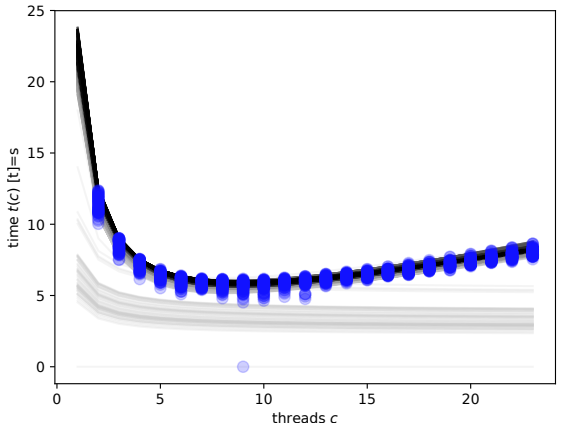

Figure 2: Scaling studies with varying core counts. The blue dots are the measurements, the black lines result from our non-linear weighted regression. The darker the stroke, the more data has been available to the regression.

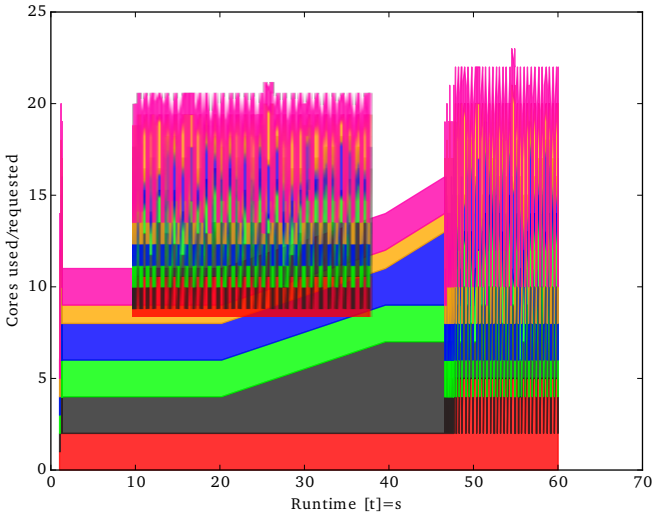

Figure 3: Example core per rank distribution incl. zoom-in for our global approach where the core-to-rank distribution is subject to a global optimisation, i.e. it is changed per simulation time step according to (2).

(2) Each rank notifies all other ranks about its $(f_r, t_r(1), s_r)$. A global optimisation to reduce the overall compute time then instructs the rank whether it should release some cores or try to grab more cores.
(3) Each rank tries to invade its optimal number of ranks prior to the time step kick off and retreats from the cores as soon as the traversal terminates and the rank continues with tidying up all MPI messages and waiting for the next time step instruction.

Runtime analysis. With N measurements $\left((c_r^{(1)}, t_r^{(1)}), \ldots, (c_r^{(N)}, t_r^{(N)})\right)$ of runtimes of a rank r for various (invaded) core numbers, we obtain an overdetermined non-linear data fitting problem to determine f_k, $t_r(1)$ and s_r. Let the first entry in this series, i.e. $(c_r^{(1)}, t_r^{(1)})$ is the most recent measurement. We formalise the calibration per node as

$$\forall r : \min_{f_r, t_r(1), s_r} \frac{1}{2} \sum_{n=1}^{N} q^n \| f_r \cdot t_r(1) + (1 - f_r) \frac{t_r(1)}{c_r^{(n)}} + s_r \cdot c_r^{(n)} - t_r^{(n)} \|^2$$

with a temporal weighting $q \in (0, 1)$ which makes more recent measurements more significant. $q = 0.9$ is used in the experiments.

The problem is a constrained, non-linear, weighted regression problem from machine learning. We solve it iteratively via Gauss-Newton shifts that we manually constrain after each Newton iteration such that $0 < f_r < 1$ and $t_r, s_r > 0$. Furthermore, updates to the three quantities are made sliding averages. Simpler schemes such as Picard experimentally did fail in our setups, so we assume that there is a lack of contraction properties. Given the temporal weighting, we may assume that the triple f_r, t_r, s_r yields a reasonable accurate description of a ranks runtime behaviour after few grid sweeps.

Global optimisation of core distribution. The global workload distribution now reads

$$\min_{c_r \geq 1} \max_r \; t(c_r) \quad \text{with} \quad \sum_r c_r \leq C,$$

which we smoothly approximate by the penalised

$$\min_{c_r} \frac{1}{2m} \sum_k t(c_r)^{2m} + \frac{\alpha}{2} \left(C - \sum_r c_r \right)^2 \tag{2}$$

with $\alpha > 0, m \in \mathbb{N}^+$. Setting the derivative zero yields the optimality condition.

Our code uses the scratchpad to exchange the (f_r, t_r, s_r) values determined from the measurements. As part of the machine learning process, each rank dumps its regression results and as each rank can read all entries from the scratchpad, each rank can determine the solution to the overall core optimisation problem locally. As (f_r, t_r, s_r) for most of our setups change only smoothly, these solves are not synchronised at all, i.e. some ranks might reuse calibration data from previous time steps, while other ranks already rely on updated data.

6 RESULTS

We ran experiments on 24 core Intel E5-2650V4 (Broadwell) nodes. The Broadwells run at 2.4 GHz and are connected by Omnipath. Intel's 2017.2 C++ compiler is used with the accompanying Intel MPI library. We deploy six MPI ranks on each single node, and pick out particularly interesting or characteristic results. All timings are real-time measurements sampled every time a time step terminates.

The global optimisation (Figure 3) starts with one core per ranks and remains quasi-stationary for the program's start-up phase where many system calls (memory allocations) are required. Not much concurrency is observed here. Once the actual computation

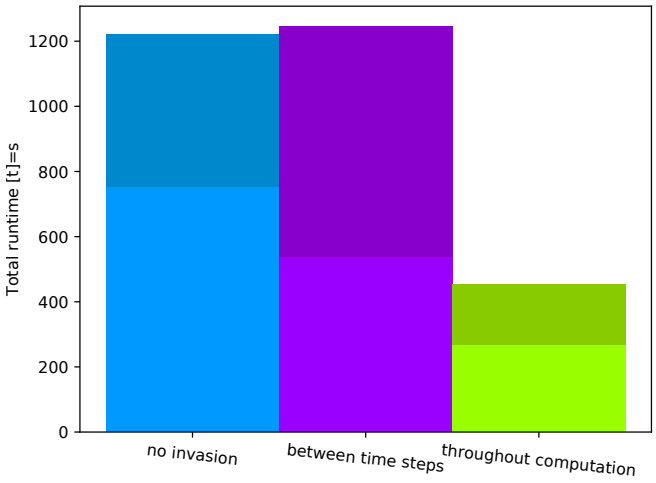

Figure 4: Global runtimes for a static core-to-rank association (TBB only) and the two different invasion approaches. The darker bars in the background track the total time-to-solution. The lighter bars in the foreground track only arithmetically intense solver steps.

starts, it yields a seismogram-like pattern. Each rank releases its cores after the traversal immediately but other ranks struggle to benefit from it as they are, for reasonably balanced MPI applications, already close to the time step completion, too. An amortised slack of 3–4 cores is observable which remain unused. A more aggressive invasion throughout the solve which does not try to fix the core count once per time step promises to fill in these slacks.

While our measurements (Fig. 2) suggest that the regression requires a few hundred steps to converge, the invasion seems not to suffer from this fact—if all rank estimates are off by the same ratio, the haggling for cores still seems to come up with reasonable core distributions. An exact study of this behaviour however is subject of future research.

We next compare invasion throughout the actual computation with the previous invasion in-between time steps and a non-invasive baseline with a homogeneous core distribution. Hereby, we track either the total runtime or the computationally intense code parts only. We see the global optimisation paying off if we focus only on the arithmetically intense code parts. If the computational intensity however is small, one invasion per time step yields worse performance even than a non-invasive approach. Our results suggest (not shown) that our simple Amdahl model in (1) does not hold anymore. The totally dynamic approach outperforms a per time step invasion robustly. If the computational intensity is high, the overhead induced by frequent invasion tries is negligible. If the computational intensity is low, any approach without fine-grain invasion is doomed to fail right from the start.

7 SUMMARY AND OUTLOOK

We propose an invasive extension of TBB which require users to insert only very few lines of code into their applications. It works without any centralised decision making algorithm (resource manager). The integration into a sophisticated simulation code suggests that a rank-global optimisation of resource usage is, counter-intuitively, not superior to an anarchic, greedy grabbing of resources.

Future work comprises, on the one hand, detailed studies on the invasion behaviour. Notably, we have to evaluate whether hybrids of global optimisation and greedy resource invasion can outperform all presented runs. On the other hand, locality- and affinity-aware assignment of invaded cores is not yet integrated into the invasion code base. Also, temporarily core overbooking might improve the performance for scenarios where cache thrashing plays a non-significant role, overheads induced by invasion are outperformed through non-exclusive core assignment or the program idles/stalls and the flow control is handed over to the operating system. Such features might provide a further invasion boost.

ACKNOWLEDGEMENTS

The authors appreciate support received from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). Thanks are due to all members of the ExaHyPE consortium who made this research possible, notably Dominic E. Charrier and Benjamin Hazelwood. This work made use of the facilities of the Hamilton HPC Service of Durham University. Both authors appreciate former funding through the Transregional Collaborative Research Centre 89—Invasive Computing (DFG funded).

REFERENCES

[1] G. M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67 (Spring))*. ACM, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[2] Michael Bader, Hans-Joachim Bungartz, and Martin Schreiber. 2013. Invasive computing on high performance shared memory systems. In *Facing the Multicore-Challenge III*. Springer, 1–12.

[3] M. Bader, M. Dumbser, A.A. Gabriel, H. Igel, L. Rezzolla, and T. Weinzierl. 2017. ExaHyPE—An Exascale Hyperbolic PDE Engine. (2017). http://www.exahype.org

[4] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. 2016. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 82–97.

[5] M. Dumbser, O. Zanotti, R. LoubÃĺre, and S. Diot. 2014. A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws. *J. Comput. Phys.* 278 (2014), 47–75.

[6] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. 2012. Invasive computing with iOMP. In *2012 Forum on Specification and Design Languages (FDL)*. IEEE, 225–231.

[7] S. Kobbe. 2015. *Scalable and Distributed Resource Management for Many-Core Systems*. Ph.D. Dissertation. Chair for Embedded Systems (CES), Department of Computer Science, Karlsruhe Institute of Technology (KIT).

[8] M. Schreiber, C. Riesinger, T. Neckel, H.-J. Bungartz, and A. Breuer. 2015. Invasive Compute Balancing for Applications with Shared and Hybrid Parallelization. *International Journal of Parallel Programming* 43, 6 (2015), 1004–1027.

[9] J. Teich et al. 2017. Transregional Collaborative Research Centre 89. (2017).

Author Index