# 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.

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.

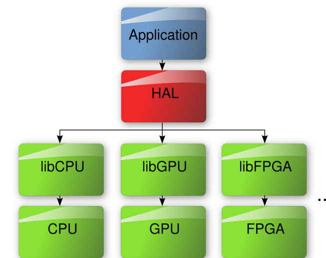


**Figure 1: Library-based approach for implementations in HALadapt additionally working as intermediate layer between hardware and application**

## 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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice.
*COSH2018, January 2018, Manchester, United Kingdom*
© 2018 Copyright held by the authors. Published in the TUM library.
https://doi.org/10.14459/2018md1428536

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 $1.6 \times 10^9$ 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.

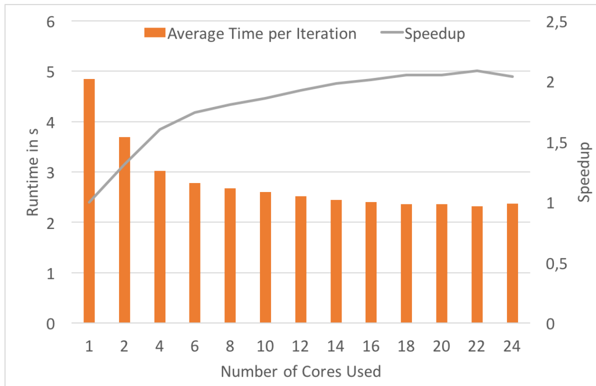


**Figure 2: Native OpenMP MLEM scaling behaviour on *sk1* without HALadapt**

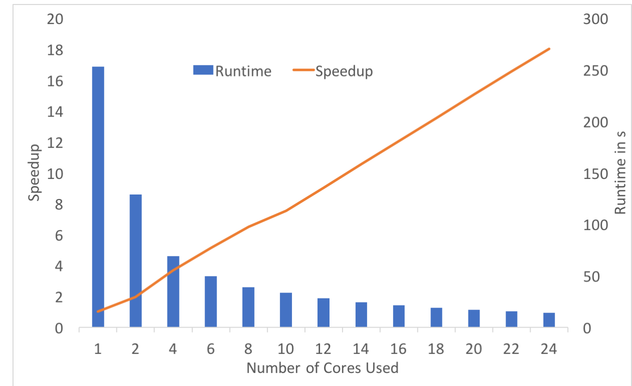


**Figure 3: Native Performance of our stressgen code on *sk1* without HALadapt**

(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 |

cores and are executed successively. This results in a total execution time of 40.44 s as the sum of $10 \times 2.36$ 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

**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 |

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

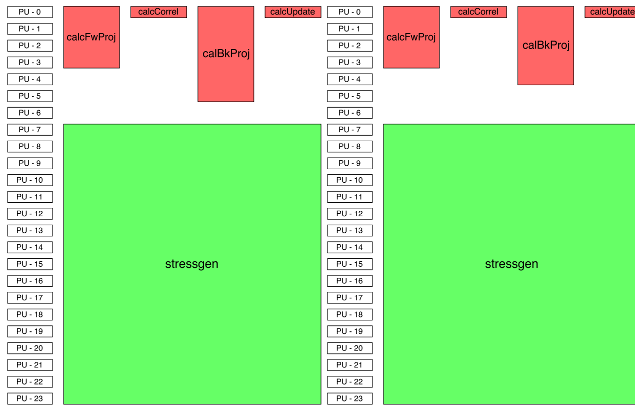


**Figure 5: Left: Mapping result of the first experiment; Right: Mapping result of the second experiment**

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.

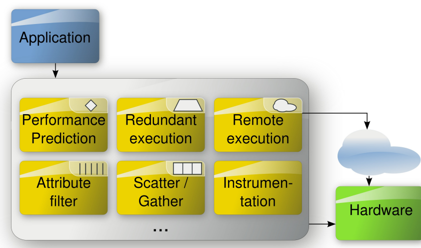


**Figure 6: All services in HALadapt are separated from each other and can be used and modified independently**

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.