

Automatic Co-scheduling Based on Main Memory Bandwidth Usage

Jens Breitbart^(✉), Josef Weidendorfer, and Carsten Trinitis

Department of Informatics, Chair for Computer Architecture,
Technical University Munich, Munich, Germany
{j.breitbart,josef.weidendorfer,carsten.trinitis}@tum.de

Abstract. Most applications running on supercomputers achieve only a fraction of a system's peak performance. It has been demonstrated that co-scheduling applications can improve overall system utilization. In this case, however, applications being co-scheduled need to fulfill certain criteria such that mutual slowdown is kept at a minimum. In this paper we present a set of libraries and a first HPC scheduler prototype that automatically detects an application's main memory bandwidth utilization and prevents the co-scheduling of multiple main memory bandwidth limited applications. We demonstrate that our prototype achieves almost the same performance as we achieved with manually tuned co-schedules in previous work.

1 Introduction

Most applications running on supercomputers achieve only a fraction of a system's peak performance, even though carefully optimized applications are able to get close to this limit. It seems unlikely that code written by non-experts will provide higher system utilization in the foreseeable future, especially with computer architecture permanently evolving, making it a moving target for optimizations. Furthermore, expected trends such as increased core counts, specialization and heterogeneity will make it even more difficult to exploit available resources.

A possible way to increase overall system utilization without optimizing the code itself is co-scheduling, i. e., running multiple applications with different resource demands on the same node¹. Such an approach may reduce single application performance. However, it increases overall application throughput of the whole system and thereby produces more results with a given time frame or energy budget. A major challenge for efficient co-scheduling is the detection of an application's resource requirements and predicting the applications performance when co-scheduled with another application.

It is obviously not feasible for HPC compute centers to run every possible application combination to decide on optimal co-schedules. As a possible solu-

¹ A node is one endpoint in the network topology of an HPC system. It consists of general purpose processors with access to shared memory. Optionally, a node may be equipped with accelerators such as GPUs.

tion, we present a mechanism to detect application memory bandwidth requirements at runtime and use Linux control groups (cgroups²) to suspend applications if multiple applications require a high amount of main memory bandwidth. These mechanisms are implemented in a prototype application scheduler. We present a set of schedules with various applications and benchmarks and demonstrate that for these applications our scheduler works as expected and co-scheduling can increase performance and save energy. For energy measurements we present measurements of a whole node using a node-external power distribution unit (PDU). The PDU, a MEGWARE³ Clustsafe unit, takes the complete system power consumption including power supply into account. The results are almost identical to manually tuned co-scheduling results we presented previously [1].

The paper is organized as follows: First, Sect. 2 gives a detailed overview of the hardware used for our measurements, followed by an introduction to our test applications in Sect. 3. Section 4 analyzes the used applications and shows that depending on the application characteristics using all cores does not necessarily guarantee an optimal result. The following section (Sect. 5) discusses shared hardware resources in an HPC node. Sections 6 and 7 introduce our new library and scheduler. The next section discusses the results achieved with our scheduler. The paper finishes with an overview on related work and conclusions, in Sects. 9 and 10, respectively.

2 Hardware Overview

In this section we will give a brief overview of the hardware used in this paper and how energy consumption measurements were carried out.

All benchmarks were run on a 2 socket NUMA system. The system is equipped with two Intel Xeon E5-2670⁴ CPUs, which are based on Intel's Sandy Bridge architecture. Each CPU has 8 cores, resulting in a total of 16 CPU cores in the entire system. One CPU core has support for two hardware thread contexts (HTC, often called Hyperthreading) resulting in a total of 32 HTCs for the whole system. The L3 cache is shared among all CPU cores. The base frequency of the CPU is 2.6 GHz, however, the CPU typically changes the frequency of its cores based on the load of the system. Therefore, clock frequencies can vary between cores at the same time. When a core is idle, the operating system (OS) puts it into sleep state, which significantly reduces power consumption. In case only a fraction of its cores are used, the CPU can increase core clock frequencies (Intel Turbo Boost) up to 3.3 GHz. This is typically done to increase the performance of applications not being able to utilize all available CPU cores, as the CPU is less power efficient at higher frequencies. The so-called thermal design

² <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>

³ <http://www.megware.com/>

⁴ <http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2-60-GHz-8-00-GTs-Intel-QPI>

power (TDP) of each CPU in our system is 115 W, i.e. the CPU consumes about 115 W on average when all 8 cores are active.

Each CPU has its own set of memory controller with its own dedicated DRAM memory, yet there is only a single memory address space. Each core can access every memory location. Accesses to memory of a remote CPU, however, have a higher latency and can lead to contention. Memory is distributed among the CPUs by the OS using a first touch policy, which is the default on Linux (i.e. a memory page is allocated as close as possible to the core first writing to it). The location of the memory page is not changed unless explicitly requested by the OS or the user application. Our system is equipped with a total of 128 GB of RAM (64 GB per CPU). Furthermore there are both a QDR Infiniband network card and an Ethernet network card in the system, however these were idle during our measurements. All data required for the benchmark were stored on a local SSD.

Our energy measurements were carried out using a MEGWARE Clustsafe, which measures the energy consumed by the entire system. Clustsafe is a PDU developed by the MEGWARE company and typically used in their HPC system installations to monitor and control the power consumed by the system. Further, accumulated energy consumption can be provided to developers and system administrators by one counter per PDU outlet which can be queried across the network. According to MEGWARE, Clustsafe measures energy consumption with an accuracy of $\pm 2\%$. We use Clustsafe to measure the energy consumption on the primary side comprising all components of the system including cooling, network devices and storage.

3 Test Applications

We used two example applications and two benchmarks in this paper:

- a slightly modified version of MPIBlast 1.6.0⁵,
- an example application using the CG solver algorithm provided by the LAMA [2] library,
- the PRACE⁶ application proxy benchmark Hydro, and
- the heat benchmark developed at Technische Universität München.

3.1 MPIBlast

MPIBlast is an application from computational biology. Using MPI-only, it is a parallel version of the original BLAST (Basic Local Alignment Search Tool) algorithm for heuristically comparing local similarity between genome or protein sequences from different organisms. To this end, the program compares input sequences to sequence databases and calculates the statistical significance

⁵ <http://mpiblast.org/>

⁶ <http://www.prace-ri.eu/>

of matches. BLAST is used to infer functional and evolutionary relationships between sequences as well as help identify members of gene families.

Due to its embarrassingly parallel nature using a nested master-slave structure, MPIBlast allows for perfect scaling across tens of thousands of compute cores [3]. The MPI master processes hand out new chunks of workload to their slave processes whenever previous work gets finished. This way, automatic load balancing is applied. MPIBlast uses a two-level master-slave approach with one so-called super-master responsible for the whole application and possibly multiple masters distributing work packages to slaves. As a result, MPIBlast must always be run with at least 3 processes of which one is the super-master, one is the master, and one being a slave. The data structures used in the different steps of the BLAST search typically fit into L1 cache, resulting in a low number of cache misses. The search mostly consists of a series of indirections resolved from L1 cache hits. MPIBlast was pinned using the compact strategy, i. e., the threads are pinned closely together filling up CPU after CPU.

Our modified version of MPIBlast is available on GitHub⁷. In contrast to the original MPIBlast 1.6.0 we removed all `sleep()` functions calls that were supposed to prevent busy waiting. On our test-system, this resulted in underutilization of the CPU. Removing sleeps increased performance by about a factor of 2. Furthermore, our release of MPIBlast updated the Makefiles for the Intel Compiler to utilize inter-procedural optimization (IPO) which also resulted in a notable increase in performance.

In our benchmarks we used MPIBlast to search through the DNA of a fruit-fly (*Drosophila melanogaster*)⁸. The DNA was queried with 4056 snippets created from itself.

3.2 LAMA

LAMA is an open-source C++ library for numerical linear algebra, emphasizing on efficiency, extensibility and flexibility for sparse and dense linear algebra operations. It supports a wide range of target architectures including accelerators such as GPUs and Intel MIC by integrating algorithm versions using OpenMP, CUDA and OpenCL at a node level, and MPI to handle distributed memory systems. We used the latest development version of LAMA committed to its development branch on Sourceforge (commit 43a7ed⁹).

Our test application concentrates on LAMA's standard implementation of a conjugate gradient (CG) solver for x86 multi-core architectures. This purely exploits multi-threading (no MPI), taking advantage of Intel's MKL library for basic BLAS operations within the step of the CG solver. Each solver iteration involves various global reduction operations, resulting in frequent synchronization of the threads. However, static workload partitioning is sufficient for load balancing among threads. Due to the nature of a CG solver, there is no way

⁷ <https://github.com/jbreitbart/mpifast>

⁸ <ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/drosoph.nt.gz>

⁹ <http://sourceforge.net/p/libama/git/ci/43a7ed>

to exploit caches by tiling or blocking. As involved data structures (vectors and sparse matrices) do not fit into processor caches for reasonable use cases (which is also the case in our setting), performance is fundamentally limited by main memory bandwidth and inter-core/node bandwidth for reduction operations. Often, off-chip bandwidth capacity of multi-core CPUs can already be fully exploited by 2 or 3 cores. Thus, for a CG solver implementation for such a multi-core CPU, we expect to obtain the best performance with only a few cores, as using more, only would result in higher congestion regarding memory accesses. We use scattered pinning for the CG solver, i. e., threads were distributed equally among the CPUs. This allows the CG solver to use the memory bandwidth of both CPUs with less threads.

The CG solver of LAMA was applied on a matrix generated with LAMA's matrix generator. The sparse matrix has a size of $2000 * 2000$ elements and is filled with a 2-dimensional 5-point stencil.

3.3 HYDRO

HYDRO is not a low-level benchmark, but an application proxy benchmark that is being used to benchmark European Tier-0 HPC systems. HYDRO serves as a proxy for RAMSES¹⁰ [4], which is a Computational Fluid Dynamics application developed by the astrophysics division in CEA Saclay. HYDRO contains all performance relevant algorithms and communication patterns of the original application, but it is simplified and trimmed down to only about 1500 lines of code (compared to about 150,000 lines of code of the original RAMSES). Subsequently, HYDRO was ported to various programming languages and parallel programming models including Fortran, C/C++, OpenMP, MPI, hybrid MPI/OpenMP, CUDA, OpenCL and OpenACC [5]. Our experiments are based on the hybrid MPI/OpenMP C99 implementation. HYDRO's performance, similarly that of LAMA's CG solver is limited by main memory bandwidth, as its data typically does not fit into L3 cache. For our tests we use two processes, i. e., one per CPU package, and increase the number of threads for each process as this results in optimal performance for the benchmark.

3.4 Heat

Heat is a benchmark providing various implementations of an iterative Jacobi method for solving the heat dissipation problem on a regular 2-D square domain. The basic parallel implementation (called algorithm 2) uses OpenMP and two simple loops to iterate across the matrix. As a result, it is inherently main memory bandwidth limited. In contrast, algorithm 9, a more sophisticated version of this benchmark, uses cache-oblivious diamond tiling [6] and as a result is not limited by main memory bandwidth, but compute bound.

¹⁰ <http://www.itp.uzh.ch/~teyssier/ramses/RAMSES.html>

4 Application Analysis

Figure 1 shows the scalability of all applications/benchmarks on our test-system. The figure shows that the CG solver provides the best performance with 11 threads (42.7 s), however there is hardly any benefit compared to running with 8 threads (44.0 s). Overall, the CG solver only scales linearly up to 2 threads. Hydro and heat – algorithm 2 behave almost identical with a minimum runtime at 12 cores (Hydro) and 10 cores (heat – algorithm 2), but both hardly increase performance with more then 8 cores (Hydro) and 6 cores(heat – algorithm 2). MPIBlast scales almost linearly up to 16 CPU cores and heat – algorithm 9 scales almost linear up to 11 cores, but than hardly increases performance any

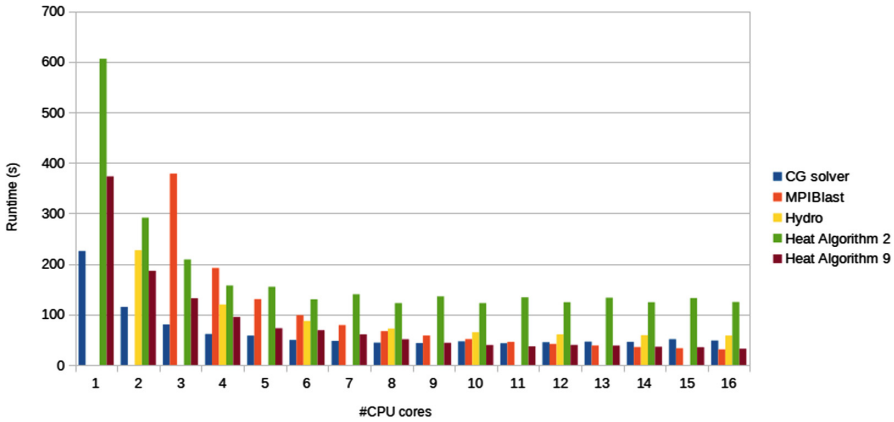


Fig. 1. The scalability of our test applications. We only use one HTC per core.

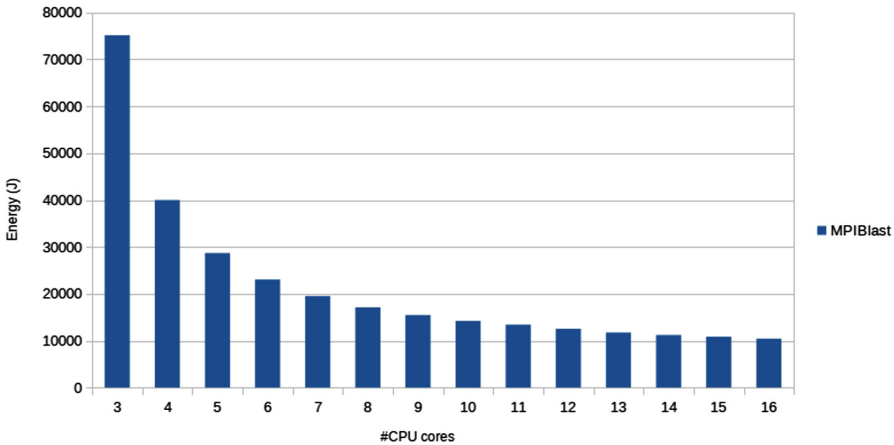


Fig. 2. Power required while running MPIBlast (Watts) and the energy required for one run (Joule).

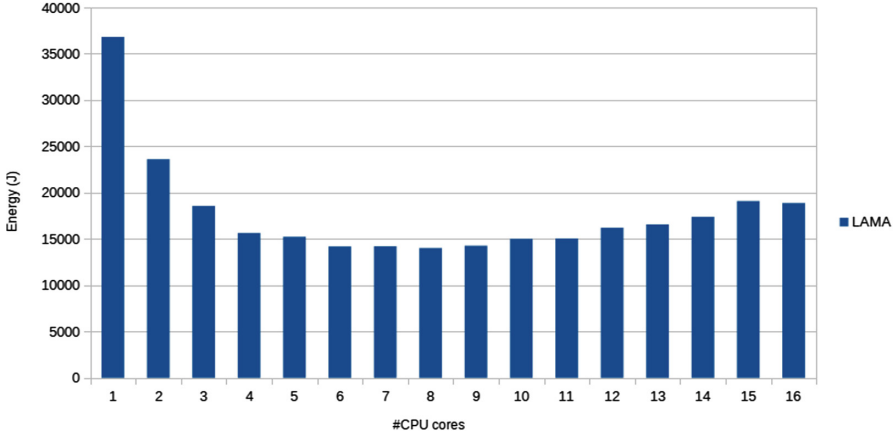


Fig. 3. Energy required for one LAMA CG solver run (Joule). We only use one HTC per core.

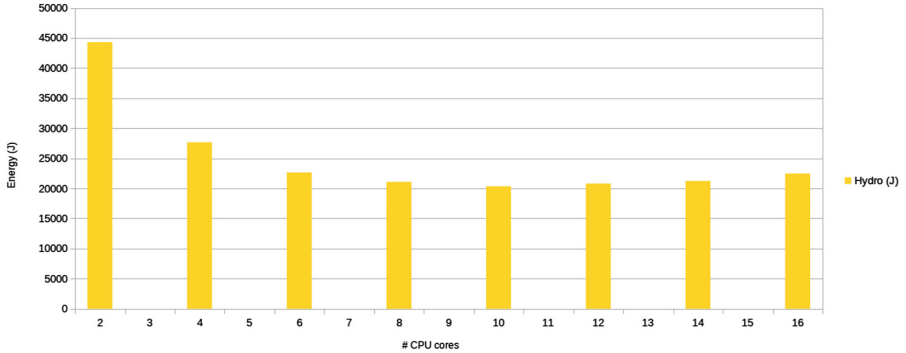


Fig. 4. Energy required for one Hydro run (Joule). We only use one HTC per core.

further. We only show even number of CPU cores for Hydro, as we use two processes with equal number of threads.

Figure 3 shows both the average power used during the scalability runs in Watt and the energy required to complete a single run of the CG solver in Joule. The Watts measured by the different sensors are indicated by lines, and the total energy integrated over the time required to complete a single run of the CG solver (often called *energy-to-solution*) is indicated by bars. It should be noted that the minimum energy-to-solution is not obtained when the CG solver provides the best performance, but with 8 cores, instead. Again Hydro (see Fig. 4) behaves almost identically, as well as heat – algorithm 2 (see Fig. 5).

Figure 2 shows the same information for MPIblast. MPIblast scales well, and the minimal energy-to-solution is obtained when using 16 CPU cores. Heat – algorithm 9 again has an optimal energy-to-solution at the point where it performs best. Figure 6 shows energy-to-solution for heat – algorithm 9.

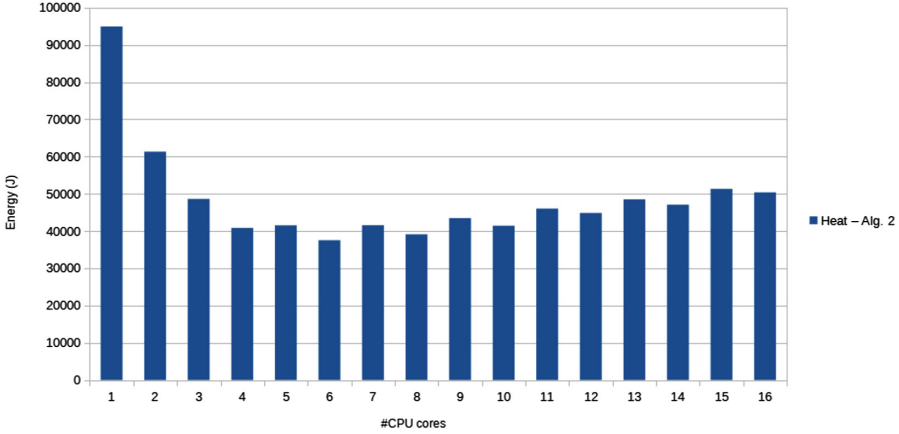


Fig. 5. The energy required for one heat – algorithm 2 run (Joule). We only use one HTC per core.

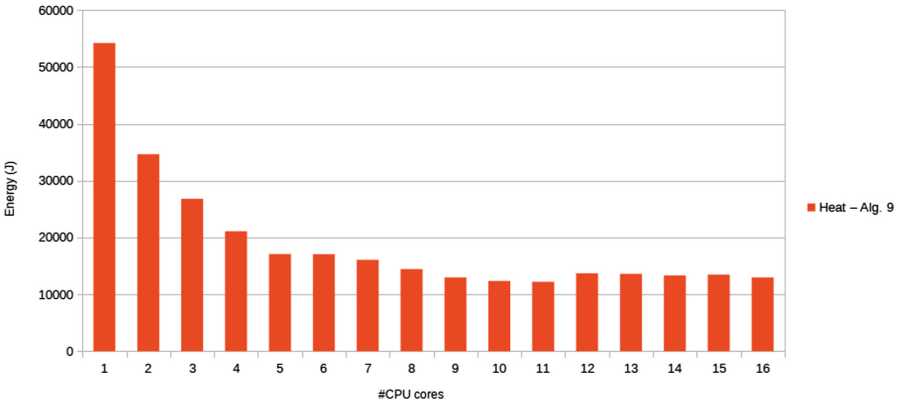


Fig. 6. The energy required for one heat – algorithm 9 run (Joule). We only use one HTC per core.

5 Shared Hardware Resources Within an HPC Node

In this section we discuss the various shared hardware resources that can limit co-scheduling performance.

At **core level**, each HTC has its own set of registers, but shares the instruction pipeline and both L1 and L2 caches with the second HTC of the same core. The instruction pipeline has dedicated hardware for floating point, integer and SIMD instructions, which can be co-issued with various constrains. As a result, co-scheduling an integer and floating point heavy application can potentially increase the utilization of the CPU core, as we have demonstrated before [1].

All cores on the same **package** share the L3 cache, the interconnect between CPU packages and main memory bandwidth. Co-scheduling multiple applications with a large L3 cache working set results in a high number of L3 cache misses and drastically reduces performance [7]. The same holds true for main memory bandwidth. Co-scheduling multiple applications with high main memory bandwidth requirements drastically reduces the performance of both applications. Based on our experience, the inter-package interconnect is typically not a limiting factor for co-scheduling.

Overall, based on our experience both main memory bandwidth and L3 cache usage conflicts can degrade co-scheduling performance up to a point at which overall system throughput is worse than dedicated scheduling. Co-scheduling different applications on a single CPU core can increase performance further, but is not essential. As a result, for all shown measurements in this paper we only use one HTC per CPU core. We leave out L3 working set detection for future work, as Intel has just recently introduced its Cache Allocation Technology (CAT)¹¹ that can be used to detect L3 cache working sets via hardware support, but is not supported at our test system. Main memory bandwidth usage is the main topic of this paper moving forward.

6 Main Memory Bandwidth Utilization (libDistGen)

Unfortunately current x86 CPUs do not provide any direct way to measure main memory bandwidth utilization, i. e., there is no performance counter that provides this information. As a result, we must deduce this information from other measurements. We leverage the fact that with co-scheduling an application never uses all CPU cores and we can use the other cores to run small benchmarks.

In previous work [7], we showed that effective co-scheduling can be predicted based on stack reuse histograms¹². Stack reuse histograms can be used to (estimate) the cache working set as well as if an application is main memory bandwidth limited. However, computing such a histogram typically results in multiple orders of application slowdown, as the we must simulate a whole application and analyze every memory access. As a result, we introduced a micro-benchmark called DistGen that can be used to get similar results. DistGen can be configured to produce memory accesses with certain stack reuse patterns. When co-scheduled with an application, we can detect peaks in the stack reuse histogram of the application based on the slowdown of DistGen. A detailed analysis can be found in [7], however all previous work was designed for off-line analysis.

Based on the original DistGen, we now introduce libDistGen, a library designed to be incorporated into schedulers or agents that collect on-line information to be used by the scheduler. libDistGen's interface is simple and consists of just three functions:

¹¹ <https://www-ssl.intel.com/content/www/us/en/communications/cache-monitoring-cache-allocation-technologies.html>

¹² The *Stack Reuse Distance*, introduced in [8], is the distance to the previous access to the same memory cell, measured in the number of distinct memory cells accessed in between. For the first access to an address, the distance is infinity.

distgen_init() is called to initialize the library. The system must be idle when this function is called, as we run various benchmarks to assess the maximum performance of the system. Depending on the numbers of cores in the system and the available memory bandwidth, this function call can take up to a few minutes to complete.

distgen_membw_available() estimates the percentage of the currently available main memory bandwidth for a given set of CPU cores compared to maximum available memory bandwidth of these CPU cores. The runtime of this function call is less than a second.

distgen_membw_max() is mainly available for debugging purposes. It returns the maximum available memory bandwidth for a given set of CPU cores of the system in GB/s.

distgen_membw_available() is implemented by processing an array with the CPU cores for which the available main memory bandwidth should be estimated. The array is larger than the L3 cache of the CPUs, so all accesses go to main memory. We measure the runtime of the accesses to the array and compare these to measurements made during *distgen_init()*. It is important to note, that these memory accesses will eventually complete, even if all other cores are running memory bandwidth limited code. As a result, we will never directly measure an available memory bandwidth of 0%, but memory bandwidth is typically equally distributed among the cores at hardware level if all cores execute memory bandwidth limited code. *distgen_membw_available()* is designed to consume as much main memory bandwidth as possible by doing hardly any computation and only accessing one byte per cache-line. These characteristics have to be considered when interpreting the return value of *distgen_membw_available()* in a scheduler.

libDistGen is available as open source on GitHub¹³.

7 Poor Mans Co-Scheduler (poncos)

The Poor Mans Co-Scheduler (poncos) is our scheduler prototype built on top of libDistGen and libponci¹⁴, which is a small wrapper for Linux control groups (cgroups). Cgroups can be used to limit which CPU cores a set of applications are allowed to use as well as transparently freeze and restart these applications. Cgroups provide plenteous of other options and are typically used to implement containers (like e. g., Docker¹⁵), but we only use the functionality named before.

For now, poncos reads a job queue from a file using a straightforward co-scheduling algorithm to run the applications listed in this file. Our algorithm briefly follows a scheme of:

1. start the first application on a subset of the available CPU cores
2. wait until the initialization phase of that application has completed (see description below)

¹³ <https://github.com/lrr-tum/libdistgen>

¹⁴ <https://github.com/lrr-tum/ponci>

¹⁵ <https://www.docker.com/>

3. use *distgen_membw_available()* on the remaining free CPU cores to detect the available memory bandwidth for the free cores
4. start the next application in the queue
5. wait until the initialization phase of the new application has ended
6. pause the old application (using cgroups)
7. use *distgen_membw_available()* on the CPU cores of the paused application to detect of available memory bandwidth
8. restart the old application
9. decide if both applications can be co-scheduled based on the available memory bandwidth
 - (a) yes: wait until one application has completed
 - (b) no: pause the new application and resume it after the old one has been completed
10. continue with 4. until the queue is empty

The current form of the algorithm expects a uniform behavior of the application during runtime. This is not true for all HPC applications, but seemingly for a large fraction of them, as other tools like for example [9] rely on the same behavior and work fairly well. In general, phase detection in applications should not be done via libDistGen as this requires the application to be paused, but phase detection should be done using hardware performance counters as demonstrated by Chetsa et al. [10]. However, libDistGen can also be used to provide information per application phase (if the phase is long enough) and this information can be used to decide if co-scheduling should be applied. For example, one could decide to only co-schedule applications if at maximum one of them has a memory bandwidth limited application phase.

We currently do not detect the end of the initialization phase, but rely on a timer that fits well with our test applications. However, in general this can be done via the mechanisms described by Chetsa et al. [10] as well.

As mentioned before, *distgen_membw_available()* will never return 0% memory bandwidth available and one has to be careful when interpreting the return value. When calling *distgen_membw_available()* to estimate the available memory bandwidth on half of the system's CPU cores, 50¹⁶% means that there is memory bandwidth limited code running on the other half of the available CPU cores and one should not co-schedule another memory bandwidth limited application. Our scheduler currently prevents co-scheduling if the sum of all applications' memory bandwidth estimations is above 90%. We use 90% instead of 100%, as we already noticed a decrease in performance once congestion on main memory gets closer to the maximum. However, this is expected behavior as the current hardware does not guarantee fair resource distribution and slowing down a particular core can decrease overall application performance due to synchronization.

Poncos is available as open source on GitHub¹⁷.

¹⁶ The theoretical minimum of distgen is at about 33%, as distgen only reads from main memory and the other half can issue both reads and writes.

¹⁷ <https://github.com/lrr-tum/poncos/tree/one-node-only>

8 Evaluation

For our evaluation we split our test system in two scheduling domains each consisting of 4 CPU cores per socket, i. e., a total of 8 cores. We choose this setup, as memory bandwidth limited applications can typically not efficiently use more than half of the cores of a socket. More cores only adds to the congestion on the memory controller and decreases performances, as discussed in Sect. 4.

In general, libDistGen works as expected with each possible pair of the applications and benchmarks listed in Sect. 3. Table 1 lists the estimated available main memory bandwidth required for the application, and based on the algorithm described in the previous section, we can deduce that poncos will prevent the co-scheduling of

- Hydro
- Lama
- Heat with algorithm 9

with each other, whereas all other combinations are fine. The resulting schedules based on our setup is rather straight forward and we only show the results of 2 input queues.

The first queue only consists of the two heat variants:

- heat – algorithm 2 (`heat -r 9000 -i 5000 -a 2 -t 8`)
- heat – algorithm 9 (`heat -r 9000 -i 5000 -a 9 -t 8`)
- heat – algorithm 9 (`heat -r 9000 -i 5000 -a 9 -t 8`)
- heat – algorithm 2 (`heat -r 9000 -i 5000 -a 2 -t 8`)
- heat – algorithm 9 (`heat -r 9000 -i 5000 -a 9 -t 8`)
- heat – algorithm 9 (`heat -r 9000 -i 5000 -a 9 -t 8`)

Figure 7 shows the runtime of queue one. In co-scheduling we only show the critical path of the scheduling. The whole schedule was completes after both runs of heat – algorithm 2 have ended, as all runs with heat – algorithm 9 could be co-scheduled with an run of heat – algorithm 2. As we can see, co-scheduling in this case increases overall application throughput, even though heat – algorithm 2 itself runs slower. The total energy consumption (see Fig. 8) of co-scheduling is

Table 1. The main memory bandwidth available for half of the cores according to libDistGen, while the other half is running the listed application. Estimated usage for the application is compute via $1 - (distgen_membw_available() - 0,33)/(1 - 0,33)$.

Application	<i>distgen_membw_available()</i>	Estimated usage for the application
Hydro	52.7	70.5
Lama	46.6	79,7
MPIBlast	92.5	11.1
Heat – Algorithm 2	41.0	88.1
Heat – Algorithm 9	76.5	35.1

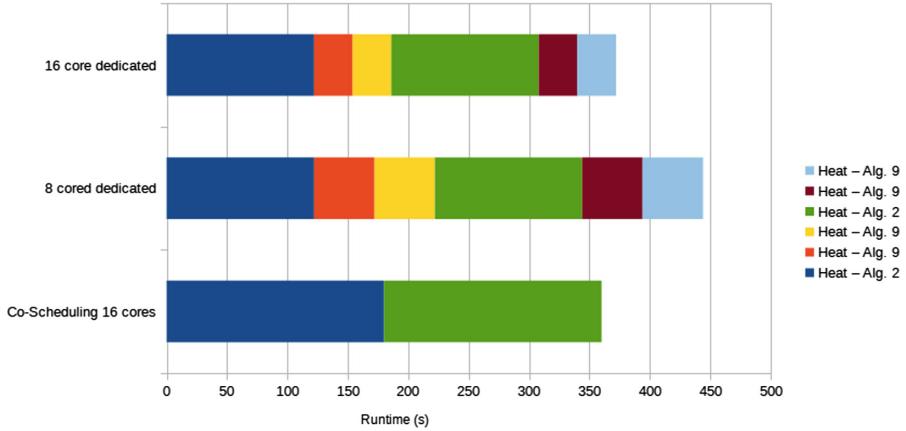


Fig. 7. The runtime of queue 1 with both dedicated scheduling and co-scheduling.

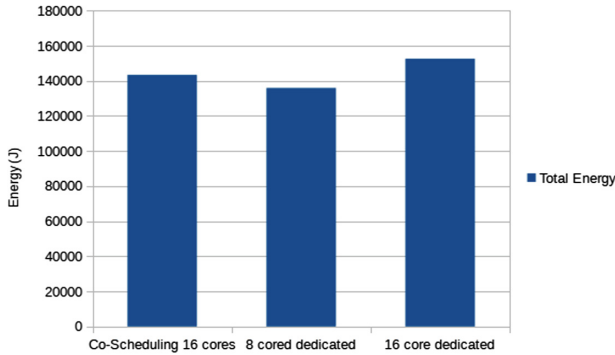


Fig. 8. The energy consumption of scheduling queue 1 with both dedicated scheduling and co-scheduling.

also better than dedicating all 16 cores to the individual applications, but just dedicating 8 cores provides a better energy-to-solution.

Our second example queue consists of:

- LAMA CG solver
- MPIBlast
- LAMA CG solver

The Figs.9 and 10 show the total runtime and energy-to-solution of the schedules of queue 2. In Fig.9 we again only show the runtime of the critical path, i.e., at the beginning LAMA is running by itself while we wait for the initialization phase to be completed and then run our measurements. After that MPIBlast is started and runs until the completion of the queue. Both LAMA runs are finished before the MPIBlast run is complete. We see a notable decrease in both runtime and energy consumption when co-scheduling MPIBlast and

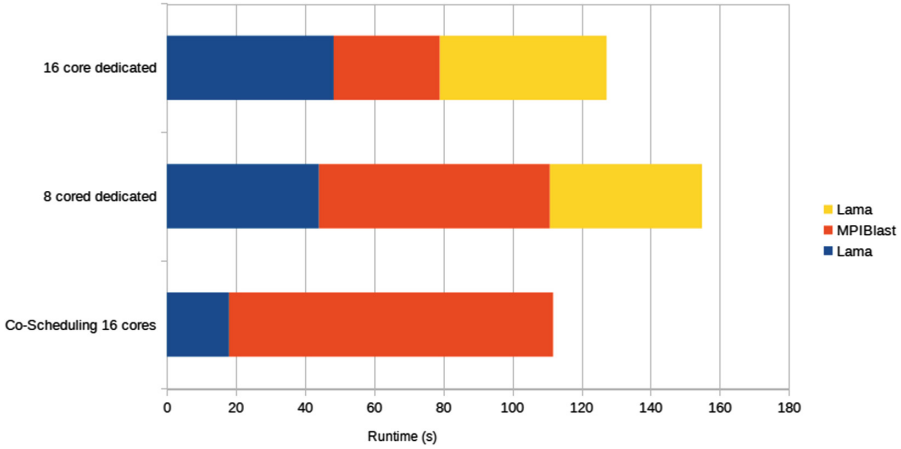


Fig. 9. The runtime of queue 2 with both dedicated scheduling and co-scheduling.

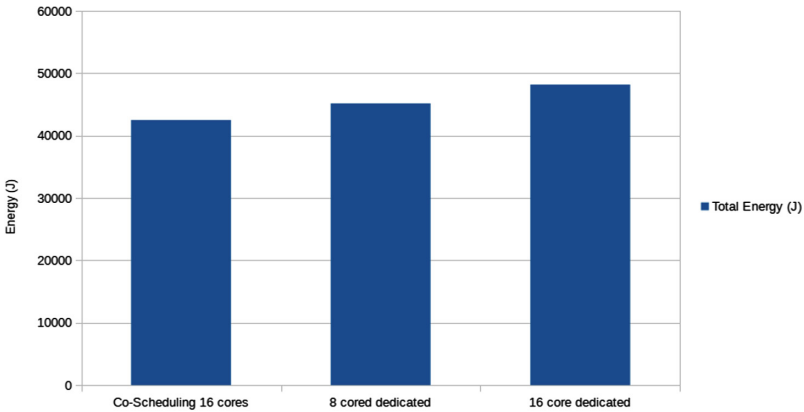


Fig. 10. The energy consumption of scheduling queue 2 with both dedicated scheduling and co-scheduling.

LAMA. These results match well with our previous manual fine tuning of the MPIBlast/LAMA co-schedule previously published in [1].

Both queues have been selected so that co-scheduling is possible. In case the queue does not allow for co-scheduling, we expect to see a small decrease in performance and a small increase in energy consumption due to the additional measurements. However, these effects seem to be within the order of measurements noise, as we could not directly measure any clear overhead.

9 Related Work

On server and desktop systems with multiple cores or hardware thread contexts simultaneous scheduling of different applications is the norm. However, in

HPC systems, most larger compute centers hardly apply any co-scheduling. Co-scheduling is typically used only for purely sequential jobs which cannot utilize all cores in a single node.

A different approach with the same goal as co-scheduling is to use power capping and dynamic voltage frequency scaling (DVFS) to reduce the power consumption of existing systems. Such an approach can obviously not increase the overall throughput of an HPC system, but increase its energy efficiency. For example Wang et al. [11] discuss a scheduling heuristic that uses DVFS to reduce overall system power consumption. The Adagio [12] tool uses DVFS to reduce the idle time of the system by analyzing the time spent in blocking MPI function calls and decreases the performance of CPU cores accordingly.

The Invasive Computing research project [13] works on an approach to have applications dynamically react to changes of their resource requirements and potential request additional resources or return resources that are no longer used. Schreiber et al. [14] for example present applications that automatically balance their work load.

Another approach to increase system efficiency is to work on the infrastructure used in the HPC centers. Auweter et al. [15] give an overview of this area and describe how a holistic approach including monitoring the various jobs can help to improve efficiency without modifying the applications itself.

Characterizing co-schedule behavior of applications by measuring their slow-down against micro-benchmarks is proposed by different works. MemGen [16] is focussing on memory bandwidth usage, similar to Bandwidth Bandit [17] which is making sure not to additionally consume L3 space. Bubble-Up [18] is similar tool accessing memory blocks of increasing size. All these tools are not designed for optimizing the schedule at runtime.

10 Conclusions and Future Work

In this paper we presented a library for on-line application analysis to guide co-scheduling and present a basic prototype scheduler implementation, which shows that this information can actually be used to implement co-scheduling. Our approach works well with all tested applications and overall system throughput and energy consumption with co-scheduling varies based on the input.

In this paper, we only concentrated on main memory bandwidth, but other resources like L3 cache usage are also important to identify if co-scheduling should be applied. In future work, we will concentrate on L3 cache usage. Furthermore, this work only explores co-scheduling on a single node. We plan to extend our experiments to a multi-node setup.

As part of the FAST project¹⁸ we plan to integrated our approach with an improved Slurm¹⁹ scheduler that uses predetermined application statistics and runtime measurements to co-schedule applications.

¹⁸ <http://www.fast-project.de/>

¹⁹ <http://slurm.schedmd.com/>

Acknowledgments. We want to thank MEGWARE, who provided us with a Clust-safe to measure energy consumption. The work presented in this paper was funded by the German Ministry of Education and Science as part of the FAST project (funding code 01IH11007A).

References

1. Breitbart, J., Weidendorfer, J., Trinitis, C.: Case study on co-scheduling for HPC applications. In: 44th International Conference on Parallel Processing Workshops (ICPPW), pp. 277–285 (2015)
2. Kraus, J., Förster, M., Brandes, T., Soddemann, T.: Using lama for efficient amg on hybrid clusters. *Comput. Sci. Res. Dev.* **28**(2–3), 211–220 (2013)
3. Lin, H., Balaji, P., Poole, R., Sosa, C., Ma, X., Feng, W.-C.: Massively parallel genomic sequence search on the blue gene/p architecture. In: International Conference for High Performance Computing, Networking, Storage and Analysis. SC 2008, pp. 1–11. IEEE (2008)
4. Teyssier, R.: Cosmological hydrodynamics with adaptive mesh refinement—a new high resolution code called ramses. *Astron. Astrophys.* **385**(1), 337–364 (2002)
5. Lavallée, P.-F., de Verdière, G.C., Wautelet, P., Lecas, D., Dupays, J.-M.: Porting and optimizing HYDRO to new platforms and programming paradigms lessons learnt (2012). http://www.prace-project.eu/IMG/pdf/porting_and_optimizing_hydro_to_new_platforms.pdf
6. Bertolacci, I.J., Olschanowsky, C., Harshbarger, B., Chamberlain, B.L., Wonnacott, D.G., Strout, M.M.: Parameterized diamond tiling for stencil computations with chapel parallel iterators. In: Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 197–206. ACM (2015)
7. Weidendorfer, J., Breitbart, J.: Detailed characterization of HPC applications for co-scheduling. In: Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications, p. 19, January 2016
8. Bennett, B.T., Kruskal, V.J.: LRU stack processing. *IBM J. Res. Dev.* **19**, 353–357 (1975)
9. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: Automated optimization of thread-to-core pinning on multicore systems. In: Stenström, P. (ed.) Transactions on High-Performance Embedded Architectures and Compilers III. LNCS, vol. 6590, pp. 219–235. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19448-1_12
10. Tsafack Chetsa, G.L., Lefèvre, L., Pierson, J.-M., Stolf, P., Da Costa, G.: Exploiting performance counters to predict and improve energy performance of HPC systems. *Future Gener. Comput. Syst.* **36**, 287–298 (2014). <https://hal.archives-ouvertes.fr/hal-01123831>
11. Wang, L., Von Laszewski, G., Dayal, J., Wang, F.: Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 368–377. IEEE (2010)
12. Rountree, B., Lownenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.: Adagio: making DVS practical for complex HPC applications. In: Proceedings of the 23rd International Conference on Supercomputing, ser. ICS 2009, pp. 460–469. ACM, New York (2009). <http://doi.acm.org/10.1145/1542275.1542340>

13. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive computing: an overview. In: Hübner, M., Becker, J. (eds.) *Multiprocessor System-on-Chip*, pp. 241–268. Springer, New York (2011)
14. Schreiber, M., Riesinger, C., Neckel, T., Bungartz, H.-J., Breuer, A.: Invasive compute balancing for applications with shared and hybrid parallelization. *Int. J. Parallel Program.* 1–24 (2014)
15. Auweter, A., Bode, A., Brehm, M., Huber, H., Kranzlmüller, D.: Principles of energy efficiency in high performance computing. In: Kranzlmüller, D., Toja, A.M. (eds.) *ICT-GLOW 2011*. LNCS, vol. 6868, pp. 18–25. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-23447-7_3](https://doi.org/10.1007/978-3-642-23447-7_3)
16. de Blanche, A., Lundqvist, T.: EnglishAddressing characterization methods for memory contention aware co-scheduling. *Engl. J. Supercomput.* **71**(4), 1451–1483 (2015)
17. Eklov, D., Nikoleris, N., Black-Schaffer, D., Hagersten, E.: Bandwidth bandit: Quantitative characterization of memory contention. In: *2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–10 (2013)
18. Mars, J., Vachharajani, N., Hundt, R., Soffa, M.L.: Contention aware execution: online contention detection and response. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. *CGO 2010*, pp. 257–265. ACM, New York (2010)