Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

FLORIAN WENDE, THOMAS STEINKE, ALEXANDER REINEFELD

The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing

ZIB-Report 15-05 (January 2015)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing

Florian Wende, Thomas Steinke, Alexander Reinefeld

Abstract: With the growing number of hardware components and the increasing software complexity in the upcoming exascale computers, system failures will become the norm rather than an exception for long-running applications. Fault-tolerance can be achieved by the creation of checkpoints during the execution of a parallel program. Checkpoint/Restart (C/R) mechanisms allow for both task migration (even if there were no hardware faults) and restarting of tasks after the occurrence of hardware faults. Affected tasks are then migrated to other nodes which may result in unfortunate process placement and/or oversubscription of compute resources.

In this paper we analyze the impact of unfortunate process placement and oversubscription of compute resources on the performance and scalability of two typical HPC application workloads, CP2K and MOM5. Results are given for a Cray XC30/40 with Aries dragonfly topology. Our results indicate that unfortunate process placement has only little negative impact while oversubscription substantially degrades the performance. The latter might be only (partially) beneficial when placing multiple applications with different computational characteristics on the same node.

1 Introduction

With the transition from petascale to exascale computers, the large number of functional components (computing cores, memory chips, network interfaces, power supplies) will greatly increase the risk of hardware- and software failures. Even current supercomputers, like the IBM Blue Gene/Q *Sequoia* system at Lawrence Livermore National Laboratory were reported to have a mean time to failure (MTTF) of 3.5 to 7 days [5], which could drop to just 30 minutes for an exascale system. With only little improvement of the MTTF of the hardware components—and related, the resilience of software components like parallel filesystems—checkpoint/restart (C/R) will become indispensable. With C/R, applications must be able to handle dynamic reconfigurations during runtime and system software is needed to provide fault tolerance at a system level.

As part of our research on the development of a fast and fault-tolerant, microkernel-based system infrastructure[1] we developed a fast in-memory C/R mechanism that writes erasure-coded checkpoints to the main memory or neighboring nodes. The checkpoints are application-triggered and hence only few state information needs to be written which allows very frequent checkpoints. In case of component failures, the state information of the crashed processes is

[1]DFG Priority Program SPP 1648 "Software for Exascale Computing" project FFMK "A fast and fault tolerant microkernel-based system for exascale computing."

re-assembled from the saved erasure-encoded blocks and the processes are restarted on other (non-faulty) resources. This inevitably results in a non-optimal process mapping after restarting a crashed application.

With a particular node allocation at the time before the failure, restarting a large job within a narrow time frame, e.g. to remain on schedule, most probably results in unfavorable process placement in terms of network distance and bandwidth. Moreover, if the job was so large that it occupied exclusively the complete supercomputer, it can only be resumed after the failure by oversubscribing (multi-allocating) some of the resources. Performance breakdowns thus are expected, possibly causing the entire computation to slow down.

In view of these scenarios, we investigated the impact of unfavorable process placement and oversubscription of compute nodes on the performance of two typical application workloads: CP2K and MOM5 namely. We ran our experiments on a Cray XC30/40 supercomputer with proprietary Aries network routers and dragonfly topology.

For the placement experiments, we first determine network latencies and bandwidths across the entire machine. Unfavorable process placements are given by node allocations with some nodes 'far away' from the others, e.g. in terms of network latency. Application benchmark setups will incorporate the different placements as well as oversubscription of compute resources.

Our results indicate that the process placement has only minor impact on the program performance. On the Cray XC30/40 the network provides almost homogeneous communication latencies and bandwidths across the entire installation, even though different layers (blade, chassis, intra- and inter-cabinet) and technologies (copper, fiber optics) are involved.

Oversubscription, in contrast, causes performance degradation larger than a factor 1.5 per application. We restrict our investigations to cases where a single application oversubscribes the available compute resources. For different workloads we study the meaningfulness of using Simultaneous Multi-Threading (SMT). The Hyper-Threading (HT) technology on Intel processors, for instance, makes a physical CPU core appear as two logical CPU cores that can feed shared execution resources concurrently to increase the instruction throughput. In particular, with HT the task scheduling is offloaded from OS kernel to the hardware.

2 Cray XC30/40

We first introduce the Cray XC30/40 installation at Zuse Institute Berlin which was used for our experiments. The system is operated as part of the compute and storage facilities of the North-German Supercomputing Alliance HLRN. The installation (for details see [3]) comprises a total of ten Cray racks with two processor types :

- XC30: 744 compute nodes, each with two Intel Ivy Bridge 12-core CPUs (nominal clock rate is 2.4 GHz) and 64 GB main memory, and
- XC40: 1128 compute nodes, each with two Intel Haswell 12-core CPUs (nominal clock rate is 2.5 GHz) and 64 GB main memory.

In summary, a total of 44,928 CPU cores are available to the HPC users. The compute nodes are connected via the Cray Aries interconnect with dragonfly topology [1, 2]. The compute

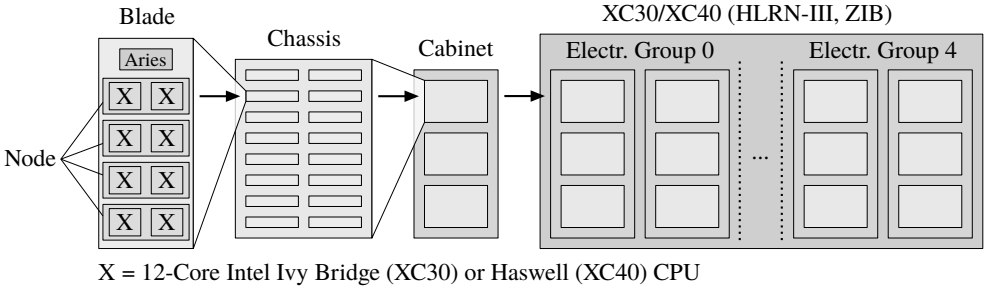

Figure 1: Cray XC30/40 component hierarchy: node → blade → chassis → cabinet → electrical group.

resources are hierarchically organized as follows: (i) a *blade* contains four compute nodes (eight CPU sockets) and one Aries router, (ii) 16 blades form one *chassis*, (iii) three chassis make up a *cabinet*, and (iv) two cabinets form an *electrical group* (Fig. 1). Our XC30/40 installation at ZIB comprises five electrical groups.

Within the electrical groups communication is performed over a two-dimensional all-to-all copper-based network: communication over the chassis' backplane in one dimension, and in the other dimension all-to-all communication within the subgroups given by corresponding nodes within the six chassis in the same electrical group. Electrical groups are connected by a fiber-channel-based all-to-all network (Fig. 2).

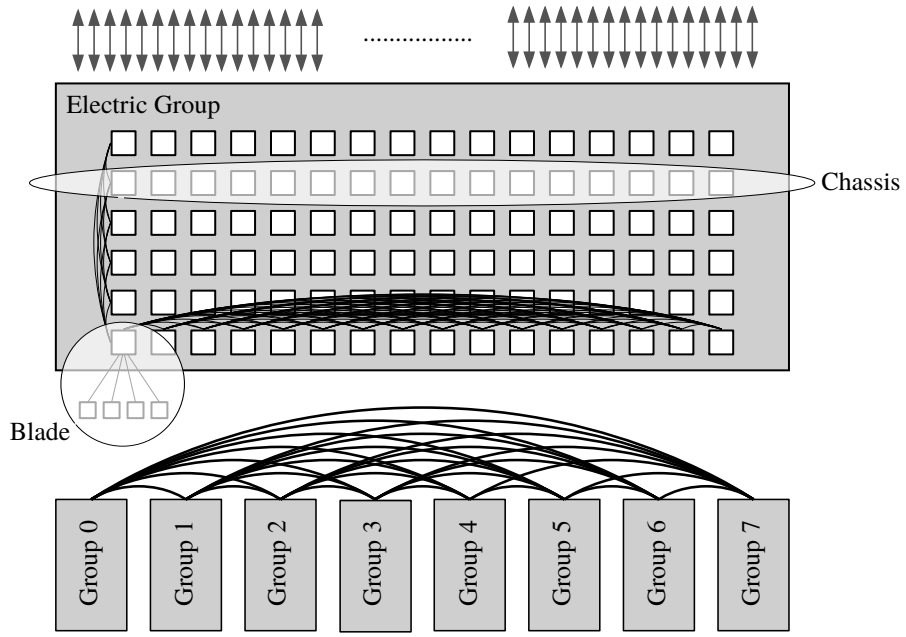

Figure 2: Structure of the Cray XC30/40 network: two-dimensional all-to-all communication within electrical groups (copper-bases), and all-to-all communication across electrical groups (fiber-channel-based).

2.1 Latencies and Bandwidths of the XC30/40 Network

The XC30/40 has a flat network structure which, even when scaling up the number of electrical groups, assures low-latency communication at high bandwidths across the entire system. Typical workloads on the XC30/40 are either pure MPI or hybrid MPI/multithreaded codes (e.g. MPI + OpenMP). Our statistics on the job submissions show that pure MPI applications usually use 16 to 24 MPI processes per node. Applications that are memory bound may see better performance with less than 24 MPI processes (or threads) per node symmetrically distributed across the two CPUs.

For MPI applications, we therefore asses the node-to-node communication bandwidths and latencies for cases where (i) $n = 1$, (ii) $n = 12$ and (iii) $n = 24$ MPI process(es) reside(s) on each of two compute nodes. The two nodes thereby are placed along the different layers of the node hierarchy of the XC30/40. Beside the different n values, case (ii) and (iii) additionally differ in all n processes reside on CPU socket 0 for (ii), and on both CPU sockets for (iii). As the network interface is attached to the PCIe root complex of CPU socket 0, the QPI (QuickPath Interconnect) latency adds to the network latency twice for (iii). For (ii) QPI does not affect the network latency. We therefore consider the cases (ii) and (iii) separately.

To request specific compute nodes of the XC30/40 we need to use appropriate MOAB features. To start 48 MPI processes that are symmetrically distributed across two different electrical groups and are numbered alternately with respect to the electrical groups, the batch script may look as follows (we use cabinet c8-0 and c9-0):

```
#PBS -l nodes=1:ppn=24:c8-0+1:ppn=24:c9-0
..
export MPICH_RANK_REORDER_METHOD=0 # round-robin process placement
aprun -n 48 -N 24 ..
```

Latencies and per-link bandwidths shown in Table 1 have been measured with the pingpong test. Our implementation determines minimum and averaged transfer times as well as transfer rates (bandwidths) accumulated across all participating MPI processes. For the latter, we determine a time frame (concurrency window) in which all processes have started and none of them

Table 1: Latencies ℓ (in µs) and per-link bandwidths b (in GB/s) of the XC30/40 network for n pairs of communicating MPI processes placed along the different layers of the XC30/40's node hierarchy.
$\ell_{\min}$: Minimum transfer time over {0,1,2,4}-byte packages in µs.
ℓ_{avg} : Averaged transfer time over {0,1,2,4}-byte packages in µs.
b: Averaged bandwidth over {1,2,4}-MByte packages in GB/s.

Node-to-Node: n processes per node	$n = 1$		$n = 12$			$n = 24$		
Communicating processes in . . .	$\ell_{\min}$	b	$\ell_{\min}$	ℓ_{avg}	b	$\ell_{\min}$	ℓ_{avg}	b
same blade, different node	1.64	9.45(1)	1.69	1.78(1)	12.6(1)	1.75	2.08(1)	11.8(1)
same chassis, different blade	1.78	9.38(1)	1.85	1.92(1)	12.9(1)	1.92	2.29(1)	12.0(1)
same cabinet, different chassis	1.76	9.40(1)	1.81	1.93(1)	12.9(1)	1.86	2.23(1)	12.0(1)
same electrical group, different cabinet	1.78	9.40(1)	1.84	1.95(1)	12.9(1)	1.90	2.26(1)	12.0(1)
different electrical group	2.31	6.77(7)	2.45	2.85(2)	12.8(1)	2.50	2.82(2)	11.8(1)

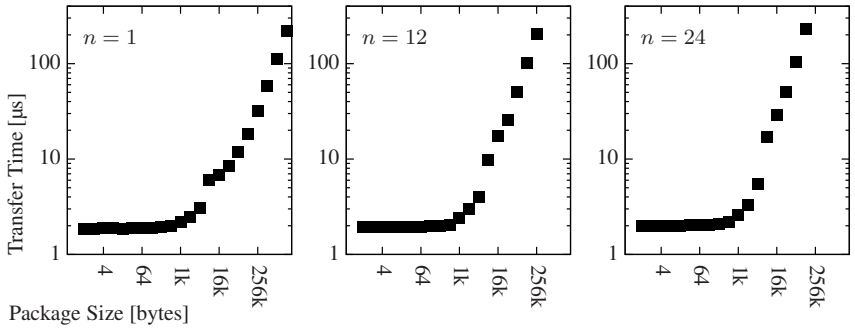

Figure 3: Transfer times (in µs) for packages of different size and for (i) $n = 1$, (ii) $n = 12$ and (iii) $n = 24$ processes per node.

has finished communication. That is, for all messages transferred within the concurrency window all processes share the per-link bandwidth of the network. Our method underestimates, but never overestimates the bandwidth.

Latencies are determined as both minimum and average transfer times for packages of size $\{0, 1, 2, 4\}$ bytes, and bandwidths are deduced from transfer times for packages of size $\{1, 2, 4\}$ MBytes. We consider different pairs of nodes and multiple successive iterations of the pingpong test. Furthermore, we set the environment variable `MPICH_GNI_MDD_SHARING=disabled` to make MPI use dedicated memory domain descriptors for better use of system resources.

According to Table 1 the Aries interconnect provides almost homogeneous performance in terms of latency and bandwidth across the entire installation. Latencies reach down to 1.64 µs for two communicating processes placed in the same blade. With $n = 12$ processes per node, average latencies increase by about 9% (1.78 µs latency in the same blade). Increasing the number of processes per node to $n = 24$ adds twice the QPI latency and gives a further increase of the average latencies of about 16% (2.08 µs latency in the same blade). Communication over the optical links between electrical groups results in 2.31 µs latency for $n = 1$, and 2.82 µs average latency for $n = 24$ processes per node. However, for communication within the same electrical group minimum latencies remain below 2 µs independently of the process placement. Fig. 3 illustrates the transfer times (including the latency plateau at small package sizes) for the cases (i), (ii) and (iii). Messages up to about 1 kByte can be transferred within almost the same amount of time.

Per-link bandwidths are determined as the accumulated bandwidth of all n processes per node. The per-link bandwidths saturate at about 13 GB/s when using multiple MPI processes. Highest bandwidths are obtained with $n = 12$ processes per node, placed on CPU socket 0. With $n = 24$ processes per node the bandwidth decreases by about 1 GB/s.

All values given in Table 1 have been determined with almost no other jobs running on the machine. Therefore, impact of adaptive routing, as provided by the Cray Aries interconnect [2], therefore should be insignificant. However, for subsequent investigations we use only parts of the XC30/40 installation. Network resources therefore are not exclusively available to us. We expect to observe performance degradation of applications due to adaptive routing and unfortunate process placements. Cases where processes reside in different electrical groups are of particular interest.

3 The Workloads

We investigate the impact of unfortunate process placements and resource oversubscription on the following workloads: CP2K and MOM5. Subsequently, we describe our compilation and simulation setups for both CP2K and MOM5, and investigate their scaling and communication behavior for selected inputs.

3.1 CP2K

CP2K is an MPI/OpenMP parallel program to perform atomistic and molecular simulations of solid state, liquid, molecular, and biological systems. It implements density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW) and classical pair and many-body potentials.[2]

Compile: We use the CP2K branch 2.4. On both the Cray XC30 and XC40 CP2K has been compiled with the Intel Fortran compiler 13.1.3 using the optional compile flag `-xcore-avx-i` on XC30 and `-xcore-avx2` on XC40, and without OpenMP. The libraries `libint` (version 1.1.5), `libxc` (version 2.2.1) and `libsmm` have been optimized for the XC30 respectively XC40 compute nodes.

Execution Setup: We use the test inputs for molecular dynamics simulations of water ensembles that are part of the CP2K distribution. We configured the setups to perform five molecular dynamics update steps (MD-section in the `H2O-xyz.inp` file). For all benchmarks we use 16 MPI processes per compute node (symmetrically distributed among the two CPU sockets).

Scaling Behavior

For the three input setups `H2O-256.inp`, `H2O-512.inp`, `H2O-1024.inp` we consider the strong scaling behavior of CP2K using 64, 128, 256, 512, and 1024 MPI processes. Fig. 4 displays the speedups over the execution using 64 MPI ranks.

For the larger inputs with 512 respectively 1024 water molecules, the strong scaling behavior is acceptable for up to 512 MPI processes on both XC30 and XC40. Scaling to significantly more than 512 processes would require increase the working size.

The right-hand side plots also incorporate the performance gain that is obtained by running the recompiled application on the XC40 instead of the XC30. About a factor 1.3 - 1.6 gain can be noted for CP2K. The differences mainly result from a slightly larger core clock on the XC40, and from fused-multiply-add support on the Haswell CPUs.

Application Profile

To determine the application profile, the CP2K executable was instrumented for sampling and tracing experiments using Cray PAT (Performance Analysis Tool). As tracing experiments produce huge amounts of data (and further introduce unavoidable overheads), we first run sampling experiments to lay the focus on the relevant routines—the instrumentation of the executable can

[2]Information are taken from the CP2K web page: `http://www.cp2k.org`

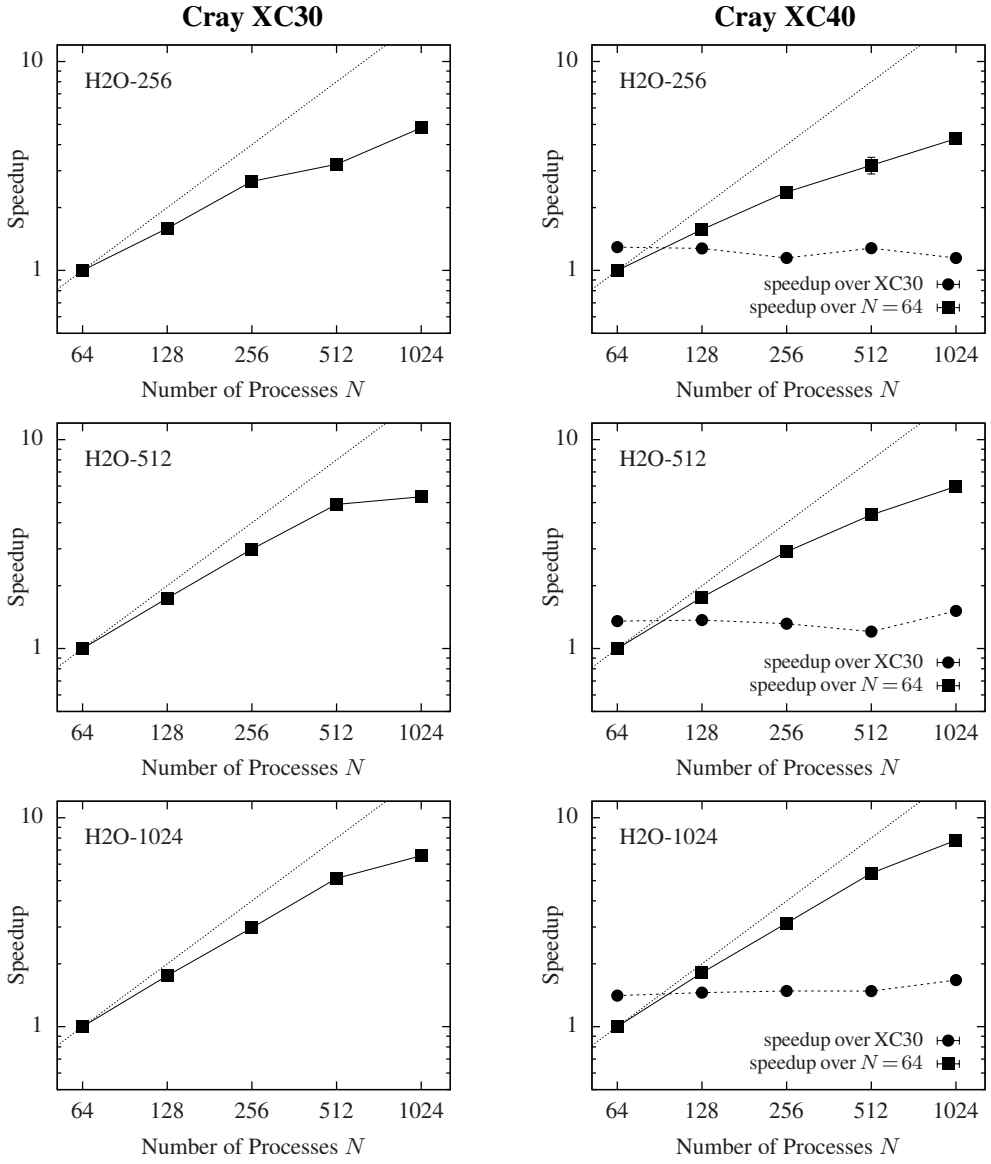

Figure 4: Strong-scaling of CP2K for three different inputs (`H2O-[256,512,1024].inp`) on a Cray XC30 respectively XC40. Given are speedups over the execution using 64 MPI processes. For the XC40, we additionally give speedups of the XC30 execution.

then draw on the information gathered throughout the sampling. Furthermore, we restrict our investigations to the (smaller) `H2O-512.inp` setup with just one molecular dynamics update step. The setup was processed by 64, 128, 256 and 512 MPI processes. Table 2 lists some data generated by Cray PAT which splits the results of sampling/tracing experiments into three sections: User, MPI and ETC. The profile remains almost the same with increasing number of MPI processes used for the computation. The majority of the execution time is spend in library calls (more than 50% MKL and `libsmm`) associated with the ETC section.

Table 2: Application profile and MPI message information for CP2K. We use 64, 128, 256 and 512 MPI processes for the `H2O-512.inp` input, and 512 MPI processes for the `H2O-1024.inp` input (marked with "†"), respectively. The Cray PAT tool splits the results of sampling/tracing experiments into three sections: User, MPI and ETC. More than 50% of the time associated with ETC is spent in MKL and `libsmm` library calls.

				MPI messages (per process, averaged)			
	Fraction of exec. time				Distribution (count)		
Processes	User	MPI	ETC	Avg. Size	$< 256\,\mathrm{B}$	$[256\,\mathrm{B}\,..\,1\,\mathrm{MB})$	$\geq 1\,\mathrm{MB}$
64	19.0%	17.2%	63.3%	190.0 kB	680 k (53%)	562 k (44%)	29 k (2%)
128	16.8%	20.3%	62.2%	140.0 kB	1.11 M (67%)	490 k (29%)	69 k (4%)
256	16.6%	19.9%	63.1%	60.4 kB	2.06 M (79%)	473 k (18%)	80 k (3%)
512	17.3%	21.3%	61.2%	33.0 kB	3.91 M (86%)	649 k (14%)	9 k (0%)
512†	6.9%	22.0%	70.8%	90.9 kB	3.11 M (82%)	537 k (14%)	142 k (4%)

Communication Behavior

The communication behavior of CP2K for the `H2O-512.inp` input has been determined with Cray PAT for runs with 64, 128, 256 and 512 MPI processes. Table 2 contains average values of the message sizes and message counts as well as information about the distribution of different sized messages—we use the categories "small" ($< 256\,\mathrm{B}$), "medium" ($256\,\mathrm{B}\,..\,1\,\mathrm{MB}$) and "large" ($> 1\,\mathrm{MB}$) for the message sizes. Small messages are those for which the data transfer is latency bound (see Fig. 3). Because of the increase of the number of small messages, the placement of the processes (e.g. in different electrical groups) might have an impact on the program performance. Almost all messages within the "medium" category can be assigned to FFT routines. Messages that fall into the "large" category are rare and seem to have their origin in the user routines.

Fig. 5 displays the communication matrix for runs with 64 respectively 512 MPI processes for the `H2O-256.inp` input. In both cases a (next-to-) nearest neighbor communication scheme can be extracted (see the minor diagonal pixels; the origin is in the top-left corner).

Furthermore, MPI processes seem to be organized in smaller groups, corresponding to the light-gray blocks along the diagonal. While the block structure is well-marked in case of using 64 MPI processes, with 512 MPI processes it is not directly visible. However, the blocks correspond to more or less inexpensive communication, so that it might be possible that Cray PAT excluded data that is below the threshold for being factored into the plot. In both images a few very expensive and seemingly regularly distributed communication pairs can be noted (the dark pixels; in the right-hand side image we introduced arrows to point them out).

From the left-hand side image (for 64 MPI processes) it can be deduced that processes $0\,..\,7$ are somehow distinguished as they receive data from all other processes (gray bar on the left). The right-hand side image seems to display a similar pattern: groups of 32 processes g_i each send data to process i (gather) and vice versa (scatter). The scatter pattern, however, cannot be found in the left-hand side image.

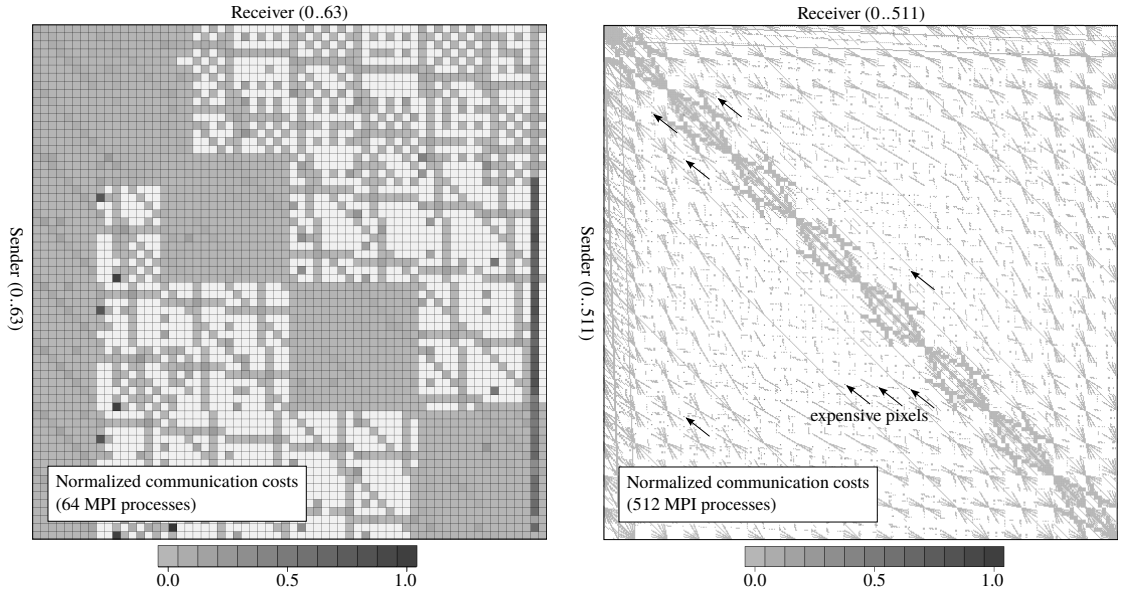

Figure 5: Communication matrix of CP2K for the `H2O-256.inp` input. Given are the normalized communication costs for (left) 64 MPI processes and (right) 512 MPI processes. In the right-hand side graphics, some expensive communication paths (corresponding to single pixels) are pointed to by arrows, for illustration purposes.

3.2 MOM5

The Modular Ocean Model (MOM) is an MPI parallel program to perform numerical ocean simulation. It has been utilized for research and operations from the coasts to the globe [4].[3]

Compile: We use MOM version 5.0.2. On both the Cray XC30 and XC40 MOM5 has been compiled with the Intel Fortran compiler 15.0.1 using the optional compile flag `-xcore-avx-i` on XC30 and `-xcore-avx2` on XC40. We use the `netcdf` library provided by Cray.

Execution Setup: We use a simulation setup modeling the Baltic Sea with nine nautical miles resolution. The input is configured to perform a 10 day simulation of the Baltic Sea in August 1961. For all benchmarks we use 16 MPI processes per compute node (symmetrically distributed among the two CPU sockets). Furthermore, simulation data is read and written to a local per-node RAM disc, and moved from/to disc storage before and after the simulation.

Scaling Behavior

We consider the strong scaling behavior of MOM5 using 64, 128, 256, 512 and 1024 MPI processes. Fig. 6 displays the speedups over the execution using 64 MPI ranks.

The strong scaling behavior for the chosen input is acceptable for up to 512 MPI processes. Similar to CP2K, scaling to larger numbers of MPI processes would require changes in the simulation setup.

[3] Information are taken from the MOM web page: `http://mom-ocean.org/web`

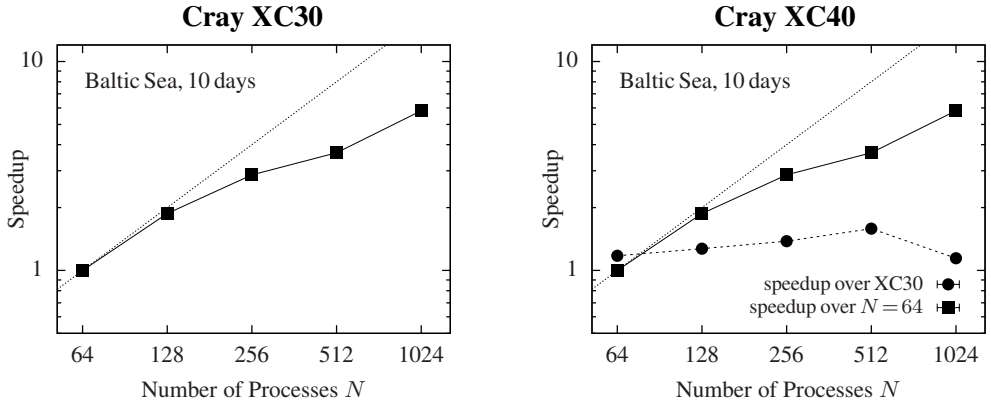

Figure 6: Strong-scaling of MOM5 for the Baltic Sea input on a Cray XC30 respectively XC40. Given are speedups over the execution using 64 MPI processes.

Executing MOM5 on the XC40 (after recompilation) gives a performance gain over execution on the XC30 of about a factor 1.2 - 1.5.

Application Profile

The application profile of MOM5 is determined with Cray PAT. Again we perform a combination of sampling and tracing experiments. We use the 10 day Baltic Sea simulation setup, and vary the number of executing MPI processes over $\{64, 128, 256\}$. Runs with more than 256 processes failed due to unknown reasons.

Table 3 lists the portions of the sections User, MPI and ETC on the program execution time. The majority of the time is spent in the user code section. The impact of I/O (via NetCDF) becomes negligible with increasing number of MPI processes, whereas the MPI portion gains weight proportionally.

Table 3: Application profile and MPI message information for MOM5. We use an input setup for a 10 day of simulation of the Baltic Sea. Cray PAT splits the results of sampling/tracing experiments into three sections: User, MPI and ETC.

				MPI messages (per process, averaged)			
	Fraction of exec. time				Distribution (count)		
Processes	User	MPI	ETC	Avg. Size	$< 256\,\mathrm{B}$	$[256\,\mathrm{B}\,..\,64\,\mathrm{kB})$	$[64\,\mathrm{kB}\,..\,1\,\mathrm{MB})$
64	79.1%	12.6%	7.5%	15.5 kB	304 k (31%)	643 k (67%)	18 k (2%)
128	77.1%	13.7%	8.2%	10.5 kB	387 k (38%)	622 k (60%)	19 k (2%)
256	74.1%	20.6%	4.4%	7.3 kB	529 k (48%)	551 k (50%)	20 k (2%)

Communication Behavior

The communication behavior of MOM5 for the Baltic Sea input has been determined with Cray PAT for runs with 64, 128, and 256 MPI processes. Table 3 contains average values of the message sizes and message counts as well as information about the distribution of different sized messages—we use the categories "small" ($< 256\,\mathrm{B}$), "medium" ($256\,\mathrm{B}\,..\,64\,\mathrm{kB}$) and "large" ($64\,\mathrm{kB}\,..\,1\,\mathrm{MB}$) for the message sizes. Small messages are latency bound (see Fig. 3). As the amount of small messages increases with the number of MPI processes, it might be possible that some impact on the program performance can be seen when processes are placed in different electrical groups. The majority of the messages have their origin in the user routines.

Fig. 7 displays the communication matrix for runs with 120 MPI processes for the Baltic Sea setup with 6 hours of simulated time. Similar to CP2K a (next-to-) nearest neighbor communication pattern can be extracted (see the minor diagonal pixels; the origin is in the top-left corner). Some neighbor communication paths are not present, which might be because of the input itself (here the geometry of the Baltic Sea area), or because of Cray PAT excluded data that is below the threshold for being factored into the plot.

The vertical and horizontal lines at the left and top of the matrix indicate that process 0 is distinguished. In fact, process 0 gathers simulation data from all other processes throughout the execution (vertical line). The horizontal line corresponds to a scatter operation with process 0 being the root.

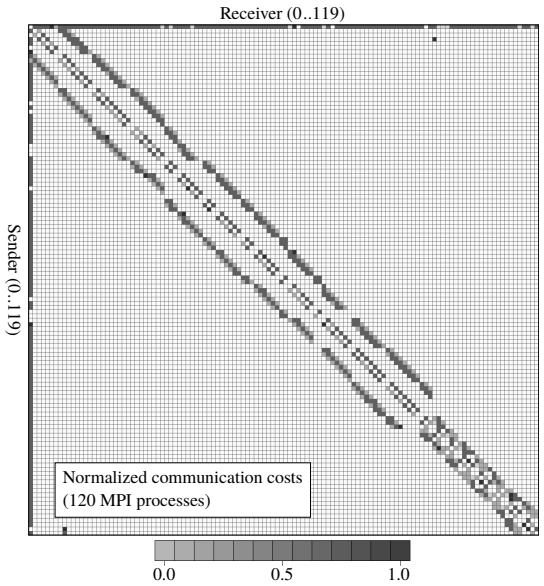

Figure 7: Communication matrix of MOM5 for the Baltic Sea input and 6 hours of simulated time. Given are the normalized communication costs for 120 MPI processes.

4 Task Placement Experiments

Computing at the speed of exascale (and beyond) will only be possible on systems with millions of processing units. Unfortunately, the large number of functional components (computing cores, memory chips, network interfaces) will greatly increase the probability of failures, and it can thus not be expected that an exascale application will complete its execution on exactly the same resources it was started on.

We therefore investigate the performance impact of placing a subset of the MPI processes far away from all other processes. In the absence of dynamics routing, "far away" on the XC30/40 means to place processes in different electrical groups—with dynamic routing, however, for communication within the same electrical group it is not guaranteed that packages are routed only within that electrical group.

We consider process placements across four cabinets in two electrical groups: (c6-0, c7-0) and (c8-0, c9-0). The distribution of the processes is encoded into the string "$n_6 - n_7 - n_8 - n_9$" where n_i refers to the number of processes placed in cabinet i. We use the `H20-1024.inp` input for CP2K and the Baltic Sea setup with 10 days of simulated time for MOM5. In both cases we use 512 MPI processes for the execution.

As for CP2K the first 16 processes seem to be distinguished (see Fig. 5), placing them far away from the remaining 496 processes should result in reduced performance. Furthermore, more than 80% of the data transfers for the chosen setup are bound by the network latency. The latter increases for inter-electrical-group communication by about a factor 1.2 (see Fig. 3).

For the MOM5 application process 0 is distinguished according to Fig. 7. Placing this process far away from the other processes should result in reduced performance. As for half (or more; see Tab. 3) of the MPI messages the communication performance is bound by the network latency, the same considerations as for CP2K should apply.

We consider the following process placements:

(a) 0-0-0-512 : all 512 processes in c9-0.

(b) 0-n_7-0-n_9 : processes $0 .. n_7 - 1$ (with $n_7 \in \{1, 2, 4, 8, 16\}$) in c7-0, and processes $n_7 .. 511$ in c9-0.

(c) 0-128-0-384 : processes $0 .. 127$ in c7-0, and processes $128 .. 511$ in c9-0.

(d) 0-256-0-256 : processes $0 .. 255$ in c7-0, and processes $256 .. 511$ in c9-0.

(e) 0-128-128-256 : processes $0 .. 127$ in c7-0, and processes $128 .. 255$ in c8-0, and processes $256 .. 511$ in c9-0.

(f) 128-128-128-128 : 128 processes in each cabinet.

The timings for the different placements are illustrated in Fig. 8. The execution (a) with all MPI processes in the same cabinet is the fastest for both CP2K and MOM5. For CP2K the performance decrease by up to 8.4% for the setups (b) can be explained by the first 16 processes being the roots of gather/scatter operations—that is, all processes send data to these processes and receive data from them. Placing the first 16 processes in a different electrical

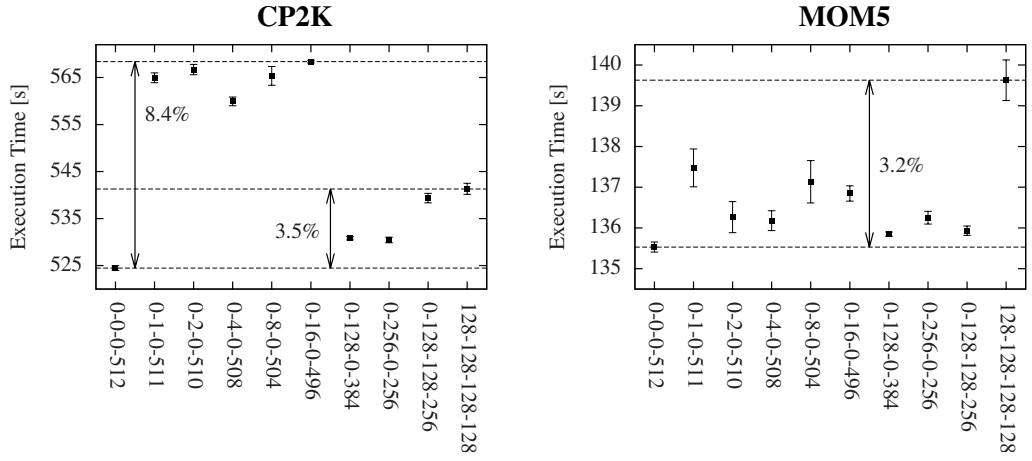

Figure 8: Process placement of 512 MPI processes executing CP2K (`H2O-1024.inp` input) and MOM5 (Baltic Sea, 10 days simulated time). The placement is encoded into the string "$n_6 - n_7 - n_8 - n_9$" (see the text).

group other than the one the remaining processes are placed in introduces larger latencies and lower bandwidths for the communication. The other setups (c) – (f) with an increasing number of processes close to the first 16 processes show performance degradations by up to 3.5% compared to the fastest execution.

In case of hardware failure(s) (in the context of C/R), restarting CP2K on a different (and maybe unfortunate) node allocation should happen such that processes 0 .. 15 (for the `H2O-1024.inp` input) are placed close to the majority of MPI processes.

The MOM5 application seems to be less affected by unfortunate process placements. Although MPI process 0 is central to collective communication, the impact of moving it to a different electrical group results in just 1.4% performance decrease. Other than for CP2K the number of messages that are bound by network latency is about 50% for MOM5 (for CP2K it is about 80%). The impact on the MPI fraction of the execution therefore is much lower. Restarting MOM5, e.g. after hardware failure(s), on unfortunate node allocations results in a minor increase of the program execution time.

5 Oversubscription Experiments

Restarting an application might not just result in executing it on an unfortunate node allocation, but even in executing it on a reduced set of hardware resources. Subsequently, we consider only cases where compute nodes are not shared across different applications. That is, the application itself oversubscribes its allocation, meaning n processes are placed on $n' < n$ physical execution units. With our experiments, we want to answer the following questions: What is the impact of executing two processes on one physical CPU core? Can SMT (e.g. Hyper-Threading on Intel CPUs) help reduce performance degradation when oversubscribing the compute resources? Can we measure performance degradation caused by reduced per-process (lower-level) L1/L2 cache capacity?

Fig. 9 illustrates the per-node process-core assignments that are used to give answers to the above questions (we use 16 processes per node; each XC30/40 node provides 24 physical CPU cores 0 .. 23 and 24 logical CPU cores 24 .. 43, named Hyper-Threads 0 and 1 hereafter):

No-oversubscription: 16 processes reside on physical CPU cores 0 .. 7 and 12 .. 19.

Pattern 1: 16 processes reside on physical CPU cores 0 .. 3 and 12 .. 15,
that is, the respective CPU cores are two-fold oversubscribed:
(i) per-process fraction on L3 cache remains the same.
(ii) per-process fraction on L1/L2 cache is half as large.
(iii) context switch between processes via Linux scheduler.

Pattern 2: 16 processes reside on physical CPU cores 0 .. 3 and 12 .. 15
and logical CPU cores 24 .. 27 and 36 .. 39:
(i) per-process fraction on L3 cache remains the same.
(ii) per-process fraction on L1/L2 cache is half as large.
(iii) process interleaving via Hyper-Threading.

Pattern 3: 32 processes reside on physical CPU cores 0 .. 7 and 12 .. 19,
and logical CPU cores 24 .. 31 and 36 .. 43:
(i) per-process fraction on L3 cache is half as large.
(ii) per-process fraction on L1/L2 cache is half as large.
(iii) process interleaving via Hyper-Threading.

Pattern 1 is used to investigate the effect (e.g. mutual cache pollution) of reducing the per-core L1/L2 caches by executing two processes on the same physical core. Furthermore, the impact of context switches between the two processes might be significant. For both CP2K and MOM5, we expect to see at least a factor 2 lower performance for the same node allocation, but with half the number of CPU cores used.

Pattern 2 serves as a testing candidate for improving the CPU core utilization(s) by using Hyper-Threading—idle CPU cycles maybe the result of e.g. pending memory accesses, I/O,

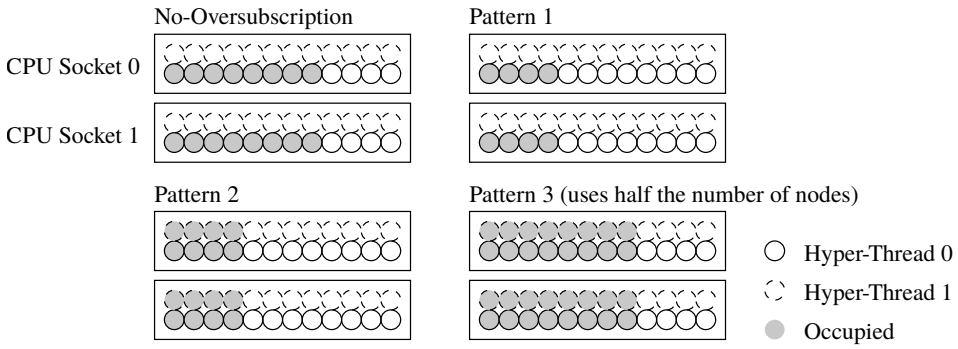

Figure 9: Per-node process placements in the context of oversubscription. We assume an Intel CPU with Hyper-Threading enabled (as is the case on our XC30/40). Cores where processes are placed on are gray-colored.

or network communication. As the per-core process interleaving is done in hardware via the Hyper-Threading mechanism, the overhead of expensive context switches is removed and idle CPU cycles can be effectively used. The effective per-process L1/L2 cache sizes are reduced. However, for both CP2K and MOM5, we expect to see an improved CPU core utilization for the same node allocation, but with half the number of CPU cores used. The performance decrease therefore should be below a factor 2.

Pattern 3 is used to investigate both the effect of reduced L1/L2 and L3 cache sizes and improved CPU core utilization via Hyper-Threading.

For the assessment of oversubscription results the following statement applies: *The cross-over point for performance decreases is 2.0, i.e. a performance decrease (per workload) smaller than 2.0 results in a higher overall computational throughput on the system.*

CP2K

The results of the oversubscription experiments for CP2K are illustrated in Fig. 10. The left-hand side sub-plots are program execution times showing the weak- and strong scaling of CP2K in the presence of oversubscription. The right-hand side sub-plots show the performance decreases over the no-oversubscription case. The fastest execution (with respect to time-to-solution) is the version not using oversubscription.

For pattern 1 our expectations prove right.The performance decreases by more than a factor 2 for setups `H2O-[256,512].inp` with increasing number of MPI processes. With the problem size fixed, using more and more MPI processes results in a reduction of the per-process problem size. The latter might compensate for the reduced per-process L1/L2 cache size (because of oversubscription). On the other hand, an increased number of processes may result in more communication. For the `H2O-1024.inp` setup the performance reduction is only slightly worse than a factor 2. The impact of reduced per-process cache sizes and context switching seems to be not significant in this case.

For patter 2 and 3 we observe an improved utilization of the compute resources. That is, the execution of CP2K exhibits a non-negligible amount of idle CPU cycles, which can be used by means of process interleaving (here via Hyper-Threading).

While pattern 1 and 2 assume the same amount of nodes allocated for program execution, pattern 3 maps to the case where the execution happens on a reduced set of compute resources. By doubling the number of processes per node, the same number of processes can run on half the number of nodes. As the same amount of physical CPU cores is used for the execution of CP2K, the observed performance decrease again means that the core utilization could be improved.

MOM5

Fig. 11 shows the results for the Baltic Sea setup. Pattern 1 slows down the execution by more than a factor 2. Using Hyper-Threading results in an improved CPU core utilization. The slowdown in that case increases with the number of MPI processes, whereas for CP2K it decreases. This indicates that for CP2K (and for the chosen setups) the number of CPU idle cycles grows with the number of processes. This might be caused e.g. by an increased number of pending

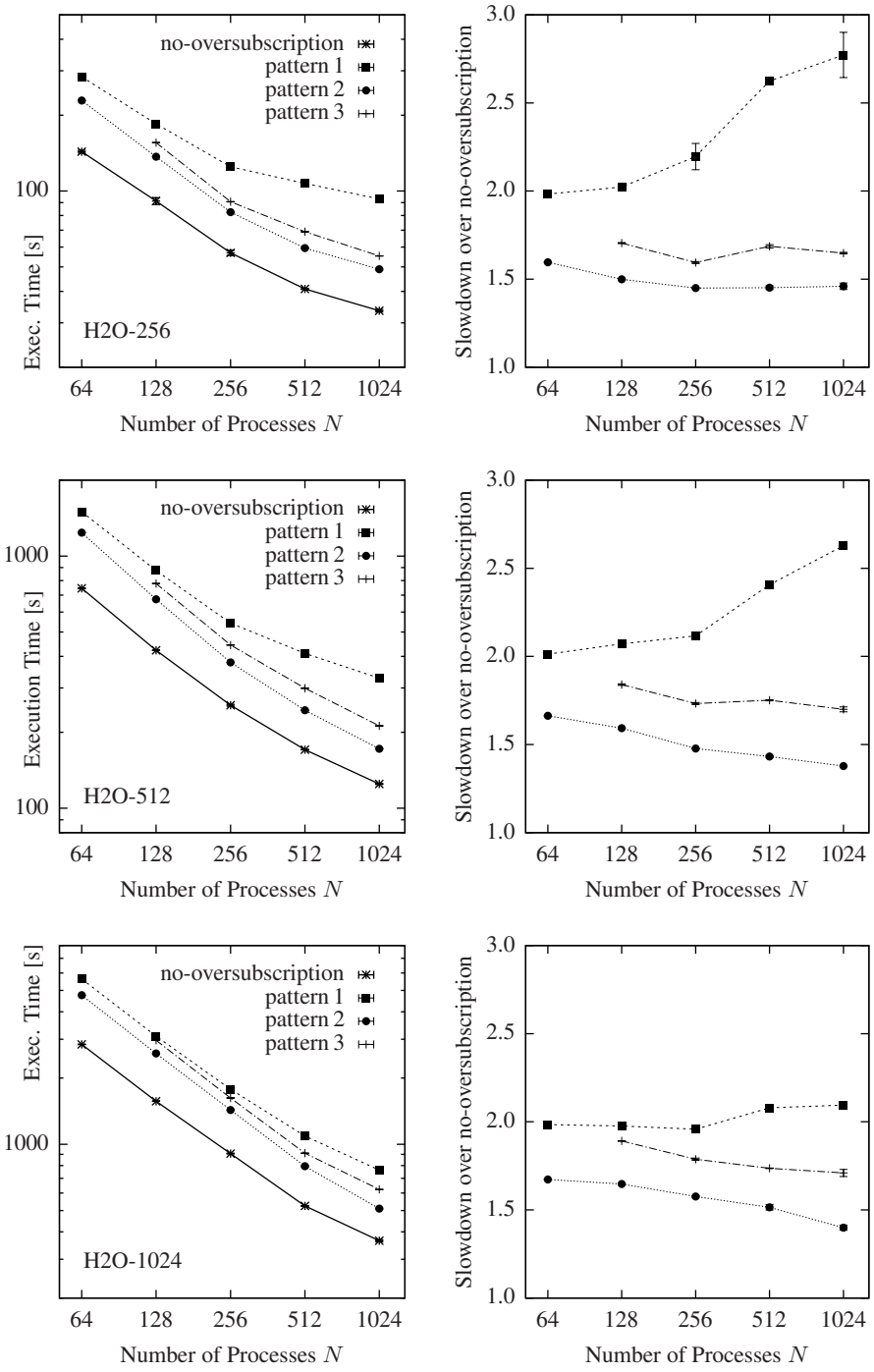

Figure 10: Oversubscription experiments for CP2K. We use the `H2O-[256,512,1024].inp` inputs and vary the number of MPI processes over {64, 128, 256, 512, 1024}. The meaning of pattern [1,2,3] is described in the text.

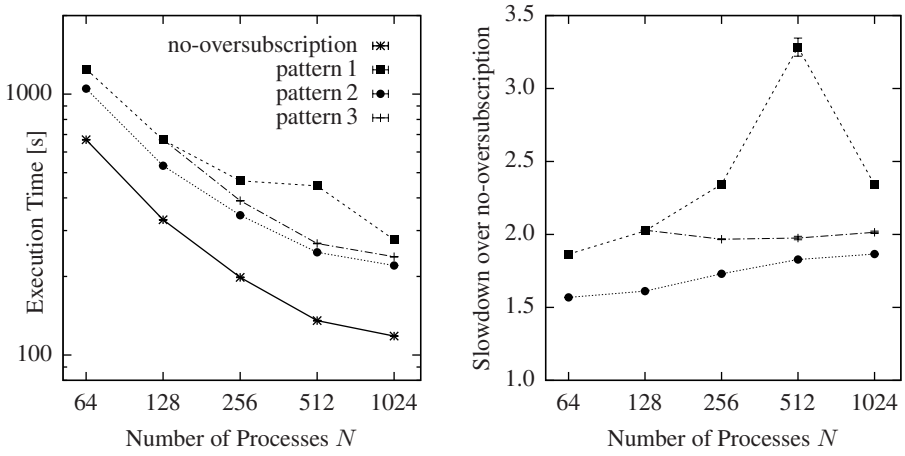

Figure 11: Oversubscription experiments for MOM5. We use the Baltic Sea setup with 10 days of simulated time, and vary the number of MPI processes over {64, 128, 256, 512, 1024}. The meaning of pattern [1,2,3] is described in the text.

memory and/or IO requests and/or network accesses. For MOM5 it seems to be the other way around.

For pattern 3, we observe almost exactly a factor 2 slowdown over the execution not using oversubscription. The compute resources seem to be well utilized in that case.

5.1 Summary

For exascale computing it is essential to achieve tolerance against hardware faults of any kind. This can be achieved by using checkpointing methods for task migration and restarting (C/R). We investigated the impact of the process placement and resource oversubscription in the context of C/R, e.g. caused by the failure of any kind of computer hardware. For different unfortunate process placements, we found the impact of resource oversubscription on the program performance be the most relevant.

Our experiments for both CP2K and MOM5 show that oversubscribing compute resources in the context of C/R is a viable option to temporarily operate on a reduced set of hardware resources. Particularly the results for pattern 3 indicate that oversubscription can help improve the CPU core utilization and hence the efficiency of the computation as a whole.

As many (supercomputer) installations provide the SMT feature (e.g. Hyper-Threading on Intel platforms) it will be necessary to further investigate the effective use of it in the context of exascale computing. Beside the patterns described in that section, it would be of interest to study the concurrent execution of at least two applications on a set of shared compute resources. Investigations on that point have been carried out by Iancu et al. [6].

Acknowledgments

This work was supported by the DFG Priority Program "Software for Exascale Computing" (SPPEXA, SPP 1648), project FFMK, "A fast and fault tolerant microkernel-based system for exascale computing." We also thank the HLRN Supercomputing Alliance for their support with computer time.

References

[1] D. Abts and J. Kim, *High Performance Datacenter Networks*, Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2011.

[2] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, *Cray XC Series Network*, Cray Inc., 2012.

[3] W. Baumann, G. Laubender, M. Läuter, A. Reinefeld, Ch. Schimmel, Th. Steinke, Ch. Tuma, and S. Wollny, *HLRN-III at Zuse Institute Berlin*, chapter 4 in J.S. Vetter (ed.): *Contemporary High Performance Computing: From Petascale toward Exascale, Volume Two*, CRC Press, ISBN 9781498700627, 2015.

[4] S. M. Griffies, *Elements of the Modular Ocean Model (MOM)*, NOAA Geophysical Fluid Dynamics Laboratory, Princeton, USA, 2012.

[5] J. Hsu, *When will we have an exascale supercomputer?*, IEEE Spectrum, pp. 13–14, January 2015.

[6] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, *Oversubscription on Multicore Processors*, IPDPS 2010, pp. 1–11.