

The Role of Idle Waves, Desynchronization, and Bottleneck Evasion in the Performance of Parallel Programs

Ayesha Afzal, Georg Hager, and Gerhard Wellein

Abstract—The performance of highly parallel applications on distributed-memory systems is influenced by many factors. Analytic performance modeling techniques aim to provide insight into performance limitations and are often the starting point of optimization efforts. However, coupling analytic models across the system hierarchy (socket, node, network) fails to encompass the intricate interplay between the program code and the hardware, especially when execution and communication bottlenecks are involved. In this paper we investigate the effect of *bottleneck evasion* and how it can lead to automatic overlap of communication overhead with computation. Bottleneck evasion leads to a gradual loss of the initial bulk-synchronous behavior of a parallel code so that its processes become desynchronized. This occurs most prominently in memory-bound programs, which is why we choose memory-bound benchmark and application codes, specifically an MPI-augmented STREAM Triad, sparse matrix-vector multiplication, and a collective-avoiding Chebyshev filter diagonalization code to demonstrate the consequences of desynchronization on two different supercomputing platforms. We investigate the role of idle waves as possible triggers for desynchronization and show the impact of automatic asynchronous communication for a spectrum of code properties and parameters, such as saturation point, matrix structures, domain decomposition, and communication concurrency. Our findings reveal how eliminating synchronization points (such as collective communication or barriers) precipitates performance improvements that go beyond what can be expected by simply subtracting the overhead of the collective from the overall runtime.

Index Terms—Parallel distributed computing, scalability, bottleneck, synchronization, desynchronization, performance modeling, performance optimization



1 INTRODUCTION

WHITE-BOX (i.e., first-principles) performance modeling of distributed-memory applications on multicore clusters is notoriously imprecise due to a wide spectrum of random disturbances whose performance impact is multifaceted. Possible sources are system noise, variations in network performance, application load imbalance, and contention on shared resources. Among the latter, the memory interface of a processor on a ccNUMA domain and the network interface of a compute node are only the most prominent ones. The typical “lock-step” pattern of many parallel programs in computational science, where computation phases alternate with communication in a regular way, can be destroyed by these effects. We call this process *desynchronization*. It arises even if the application is completely balanced and all computation and communication phases take the same amount of time on all processors, breaking the inherent translational symmetry of the underlying software and hardware. As a consequence, simply adding modeled computation and communication times does not yield reliable runtime predictions. There have been numerous efforts to assess, categorize, and reduce disturbances. In contrast, there has been relatively little work on studying *disturbance*

propagation across the processes of MPI-parallel programs, which goes under the name of *idle waves*. The propagation speed of idle waves and the dynamics of their interaction with the system (e.g., with natural system noise or with each other) can be modeled accurately in some cases. In the presence of bottlenecks, desynchronization can be initiated by the presence of idle waves and lead to a stable, persistent desynchronized state, which we call *computational wavefront*. In stark contrast to the general wisdom that perfect synchronization is always desirable, the computational wavefront state can lead to a better utilization of the bottlenecked resource by automatic overlap of communication overhead with useful work. This mechanism depends on the presence of a bottleneck among processes and is influenced by the “strength” of the bottleneck, i.e., how many processes are needed to saturate it. Canonical examples are the memory bandwidth on a ccNUMA domain and the network connection of a compute node. Still, a full quantitative understanding of the performance consequences of *desynchronization*, *idle wave propagation*, and *computational wavefronts* is still lacking, as is their impact on real-world applications.

In this paper we address the latter by investigating distributed-memory bulk-synchronous parallel benchmarks that perform computation and communication in a “lock-step” pattern. They interact with either contended or scalable resources available across the allocated set of compute nodes. The definition of contention is as follows: We assume that the N MPI processes access multiple shared resources, such as memory bandwidth, shared cache bandwidth, node

- A. Afzal, G. Hager and G. Wellein are with the Erlangen National High Performance Computing Center (NHR@FAU) Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany. E-mail: {ayesha.afzal, georg.hager, gerhard.wellein}@fau.de
- G. Wellein is with Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany.

TABLE 1: Programs and the investigated parameter space.

| Parallel codes | Parameter space for analysis |
|----------------|---|
| STREAM Triad | provoked and spontaneous disturbances matrix topology, communication concurrency communication scheme, domain decomposition |
| SpMVM | |
| ChebFD | |

network injection bandwidth, or even the full-system bisection bandwidth. Each resource i applies to a group of N_i MPI processes, which leads to N/N_i contention groups. For example, on a 20-core CPU with a single ccNUMA domain per package we have $N_{\text{memBW}} = 20$ if one MPI process is running per core. A program is *resource scalable* if there are no contention groups. This can happen because there are no contended resources or because the execution mode of the program does not expose them. In the example above, if one MPI process with 20 OpenMP threads is running per ccNUMA domain, there is no memory bandwidth contention group among processes.¹

Table 1 shows an overview of the microbenchmarks and the two mini-apps under investigation and which part of the parameter space we investigate for each. The selection aims to provide a spectrum of program properties and how idle waves and desynchronization impact their performance. The strongly memory-bound STREAM Triad code was augmented from its original [1] by adding next-neighbor communication in order to construct the cleanest possible setup that can show the desired effects. The SpMVM benchmark issues back-to-back sparse matrix-vector multiplications and thus represents many sparse iterative algorithms. It is also memory bound on the node level but adds additional complexity via problem-dependent communication patterns. Finally, in case of Chebyshev Filter Diagonalization (ChebFD) [2], all collectives can be eliminated from the algorithm without limiting its functionality, which makes it a representative of the growing field of communication-avoiding algorithms [3], [4]. The choice of a blocking factor also allows to fine-tune its computational intensity. In Sect. 3, each program will be described in more detail. We restrict ourselves to pure MPI applications; the basic phenomenology of hybrid MPI+OpenMP codes in terms of desynchronization and have been addressed in [5].

Relevance and generalization of desynchronization

Desynchronization is a very common phenomenon in parallel programs. As shown in previous work [5], [6], the presence of a resource bottleneck is decisive for the initial lock-step mode to become unstable even under natural system noise without any explicit disturbance from the outside. We call this instability *bottleneck evasion*. In absence of globally synchronizing operations, an initially small deviation from lock-step is allowed to evolve into an eventually stable pattern. This is especially easy to observe in implementations of algorithms such as synchronization-free polynomial filters [7], [8] or communication-avoiding Krylov subspace methods [3], [4]. However, even in the presence of collectives it is possible for idle waves and the desynchronized pattern to survive, depending on the particular implementation

of the collective [9]. In the desynchronized state, different loop kernels with different behavior towards chip-level bottlenecks can execute concurrently on different cores, or execution of code can overlap with communication-induced waiting time. The performance impact of desynchronization for the whole program mainly depends on the saturation characteristics of the relevant hardware bottlenecks and where the system eventually settles in terms of how many cores execute which code at any point in time. If back-to-back program phases with different behavior towards a common bottleneck overlap in time, their characteristics also govern the further evolution of the desynchronized state [10]. This phenomenon is not restricted to standard multicore architectures; it is also common in task-parallel programs and in GPUs where threads execute different kernels in parallel [11].

Idle waves and computational wavefronts

In prior work we also presented a validated analytic model for the idle wave propagation velocity [6], [9], which shows the influence of execution and communication properties of the application, with a special emphasis on sparse communication patterns. The propagation speed is the number of processes the delay travels per time unit, and it is measured in ranks per second. We pointed out how idle waves decay under system noise and due to topological differences in communication characteristics among parts of the system.

In the presence of a memory bandwidth bottleneck, the propagation of idle waves is superimposed by the bandwidth limitations, which cause a decay of the wave over time [5], [12]. In this scenario, a deliberate injection of an idle period can initiate an idle wave which, after its eventual disappearance, leaves an “echo” in the form of a desynchronized computational wavefront. Hence, adding some delay on one of the processes can accelerate the transition to the desynchronized state, possibly leading to better overall time to solution. This constitutes the important connection between idle waves and desynchronization.

1.1 Related Work

1.1.1 Noise

Extensive research [13], [14], [15], [16], [17], [18], [19], [20], [21] has been conducted for almost two decades to characterize noise, identify sources of noise outside of the control of the application, and pinpoint its influence on collective operations. Explicit techniques for asynchronous communication, mitigation of noise, MPI process placement, dynamic load balancing, synchronization of OS influence, lightweight OS kernels, etc., have been explored. In contrast, the present paper investigates the favorable consequences of noise as an enabling factor for desynchronization and – in case of parallel programs with bottleneck(s) among processes – automatic partial or full overlap of communication and computation. Hoefler et al. [22] used a simulator based on the LogGOPS communication model to investigate both point-to-point (P2P) and collective operations to study the influence of system noise on large-scale applications. They found that application scalability is mostly determined by the noise pattern and not the noise intensity. However, their data was not taken on a real cluster and it is neither aware of

1. This does not mean that there is no memory bandwidth contention; it just plays no role for desynchronization phenomena.

node-level bottlenecks nor does it take the system topology and different kinds of delay propagation into account. In the specific context of idle wave decay, Afzal et al. [9] found that the noise intensity is the main influence factor rather than its detailed statistics.

1.1.2 Parallel computing dynamics

There is very little research on idle wave propagation and spontaneous pattern formation in parallel code, especially in the context of memory-bound programs. Hence, none of the existing prior work addressed spontaneous pattern formation and desynchronization. Markidis et al. [23] used a LogGOPS simulator [22] for a phenomenological study of idle wave propagation. They concluded that isolated idle periods propagate among MPI processes as nondispersive, damped linear waves. However, their simulator is neither aware of communication topology, concurrency, and mode (rendezvous vs. eager) nor does it consider the socket-level character of the code and the quantitative investigation of the connection between damping and noise. Their speculation that idle waves are described by a linear cannot be upheld [6]. Gamell et al. [24] observed the emergence of idle periods in the context of failure recovery and failure masking of stencil codes. Boheme et al. [25] presented a tool-based approach to attribute propagating wait states in MPI programs to their original sources, helping to identify and correct the root issues. Kolakowska et al. [26] carried out a study of the virtual time horizon in conservative parallel discrete-event simulations (PDES). However, in all studies, the global properties of such waves, like damping and velocity, and the interaction with memory-bound characteristics of the application were ignored.

Afzal et al. [5], [6], [9], [9], [12], [27] were the first to investigate the dynamics of idle waves for a variety of communication patterns, (de)synchronization processes, and computational wavefront formation in parallel programs with core-bound and memory-bound code, showing that nonlinear processes dominate there. It turned out that on real cluster systems with their complex node-level topology, MPI-parallel memory-bound programs exhibit local, non-static idles waves (variable frequency and speed) which ripple through the MPI processes. These disturbances ultimately settle down into a global steady state, the computational wavefront. This transition time is dependent on numerous factors, such as communication volume, system size, seeds for naturally occurring one-off disturbances (“kicks”), etc. Our work takes up from this point and extends it towards investigating the influence of such dynamics on the performance of real-world distributed-memory parallel codes.

1.1.3 Performance modeling and optimization

Performance modeling is a powerful tool to investigate the properties of code in order to get insight into its bottlenecks and, consequently, identify optimization opportunities. In contrast to white-box (i.e., first-principles) performance models, accurate black-box approaches for describing the performance and scalability of highly parallel programs have been available for some time. These typically employ curve fitting, machine learning, and general AI methods [28], [29]. A lot of related research has also been

TABLE 2: Key hardware and software traits of systems.

| Systems | Meggie (M) | SuperMUCNG (S) |
|-----------------------------|------------------------------------|-------------------------------|
| Processor | Intel Xeon Broadwell EP | Intel Xeon Skylake SP |
| Processor Model | E5-2630 v4 | Platinum 8174 |
| Base clock speed | 2.2 GHz | 3.10 GHz (2.3 GHz used*) |
| Physical cores per node | 20 | 48 |
| Numa domains per node | 2 | 2 |
| Last-level cache (LLC) size | 25 MB (L3) | 33 MB (L3) + 24 MB (L2) |
| Memory per node (type) | 64 GB (DDR4) | 96 GB (DDR4) |
| Theor. memory bandwidth | 68.3 GB/s | 128 GB/s |
| Node interconnect | Omni-Path | Omni-Path |
| Interconnect topology | Fat-tree | Fat-tree |
| Raw bandwidth p. lnk n. dir | 100 Gbit s ⁻¹ | 100 Gbit s ⁻¹ |
| Compiler | Intel C++ v2019.5.281 [†] | Intel C++ v2019.4.243 |
| Optimization flags | -O3 -xHost | -O3 -qopt-zmm-usage=high |
| SIMD | -xAVX | -xCORE-AVX512 |
| Message passing library | Intel MPI v2019u5 | Intel MPI v2019u4 |
| Operating system | CentOS Linux v7.7.1908 | SUSE Linux ENT. Server 12 SP3 |
| ITAC | v2019u5 | v2019 |
| LIKWID | 5.0.1 | 5.0.1 |

* A power cap is applied on SuperMUC-NG, i.e., the CPUs run by default on a lower than maximum clock speed (2.3 GHz instead of 3.10 GHz).

[†] oneapi/2021.1.1 is used, whenever specified specifically.

done on code optimization, focusing on new data structures, efficient algorithms, and parallelization techniques, all of which require explicit programming [30]. In the present paper we investigate a specific mechanism – automatic communication overlap – which does not need any code changes. Note that this does not mean that “traditional” means of ensuring communication overlap such as implicit or explicit asynchronous progress threads, DMA transfers, etc., become non-essential. Depending on the details, automatic overlap may not be able to hide all the communication cost or – in general – settle in the most favorable state in terms of time to solution. What we want to highlight here is the impact of automatic desynchronization on performance in terms of easily obtainable metrics and a well-defined experimental procedure, and to identify which properties of the code-machine interaction influence this effect.

1.2 Contribution

This paper tries to fathom the role of desynchronization in parallel application programs by analyzing microbenchmarks and popular proxy applications. It makes the following relevant contributions:

- The slope of a computational wave, i.e., the degree of desynchronization among processes, is directly correlated with idle wave speed on the same system. This serves as a guiding principle to assess desynchronization phenomena in applications.
- An established computational wavefront can be shifted through the system (along the process direction) by deliberately injecting a delay which, however, has to be strong enough to provoke the change.
- Slow idle wave propagation speed caused by a small matrix bandwidth facilitates automatic overlap of communication and computation in SpMVM.
- In ChebFD, the particular domain decomposition and its impact on communication distances and idle wave speed is decisive for automatic communication overlap.

This paper is organized as follows: We first provide details about our experimental environment and methodology in Sect. 2. In Sect. 3 we discuss the performance phenomenology and implications of desynchronization on a stream-like

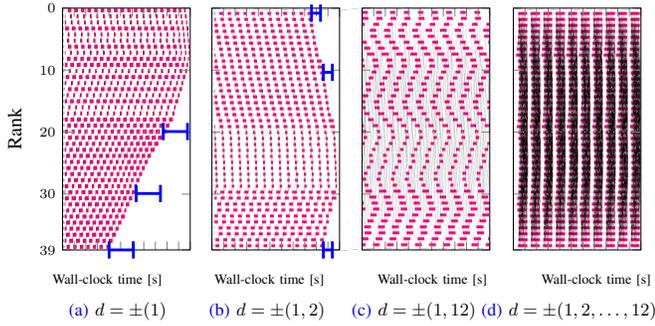


Fig. 1: Snippets of MPI trace timelines of a fully developed computational wavefront state in the MPI-augmented STREAM Triad with different communication topologies ((a)–(d)), open boundary conditions, and 5×10^4 iterations on 40 cores (four ccNUMA domains) of the Meggie cluster. Only the waiting time spent in the MPI library (red) and the message transfers (black lines) are shown.

microbenchmark, sparse matrix-vector multiplication (SpMVM), and a Chebyshev filter diagonalization application (ChebFD). In Sect. 4 we summarize the paper and give an outlook to future work.

2 TEST BED AND EXPERIMENTAL SETUP

2.1 HPC platforms and architectures

To ensure the broad applicability of our results, we conduct most experiments on two different clusters:

- 1) SuperMUC-NG², an Omni-Path cluster with two Intel Xeon “Skylake SP” CPUs per node and 24 cores per CPU (hyper-threading enabled)
- 2) Meggie³, an Omni-Path cluster with two Intel Xeon “Broadwell” CPUs per node and 10 cores per CPU (hyper-threading disabled)

Both systems have significant differences in the numbers of cores per ccNUMA domain and their memory bandwidth. Although hyper-threading is active on SuperMUC-NG, we ignore it in this work. Further details of the hardware and software environments can be found in Table 2.

2.2 Parameters, notations, and methodology

Runtime traces were visualized using the Intel Trace Analyzer and Collector (ITAC) tool. Process-core affinity was enforced using the `I_MPI_PIN_PROCESSOR_LIST` environment variable. The clock frequency was always fixed (2.2 GHz base clock speed on Meggie and 2.3 GHz in case of SuperMUC-NG because of the power capping mechanism). Unless otherwise noted, to enable overlap via hiding communication in parallel with computation supported by the MPI implementation, we set the `I_MPI_ASYNC_PROGRESS` environment variable to one and use the Intel MPI multithreaded optimized `release_mt`⁴ library, which supports

2. <https://doku.lrz.de/display/PUBLIC/SuperMUC-NG>

3. <https://anleitungen.rze.fau.de/hpc/meggie-cluster>

4. <https://software.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/environment-variables-for-asynchronous-progress-control.html>

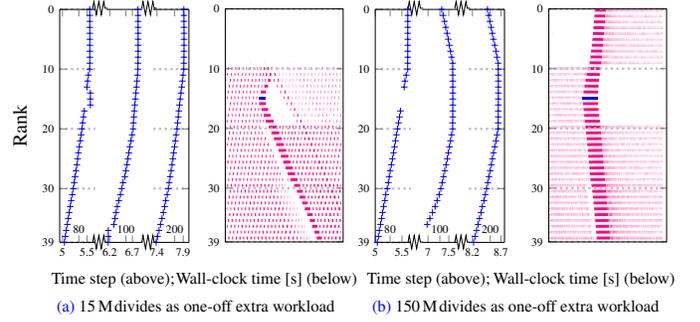


Fig. 2: Stability of computational wavefronts against (a) small and (b) large disturbances. The slope of the wavefront is the same (60 ranks/s) before the extra workload is injected (79th iteration) and after the idle wave dies out (200th iteration). (a) The “lagging” ccNUMA domain (first socket) remains the same for a small disturbance. (b) The “lagging” ccNUMA domain shifts to the second socket, which is where the large disturbance was injected.

asynchronous progress threads. Working sets for memory-bound cases were chosen large enough to not fit into the available cache, i.e., at least $10\times$ the size of all last-level cache (LLC), which is the L3 caches (non-inclusive victim L3 caches) in the Broadwell (Skylake) processors of Meggie (SuperMUC-NG). All floating-point computations are done in double precision. Individual kernel executions were repeated at least 15 times to even out variations in runtime. We report statistical fluctuations if they were significant.

2.3 Desynchronization speedup metric

We use performance measurements to quantify the speedup caused by desynchronization via bottleneck evasion. Experimentally, we determine the quantity

$$P_D = \frac{P_{\text{barrier_free}} - P_{\text{barrier}}}{|P_{\text{barrier}}|}, \quad (1)$$

which measures the speedup of the program when eliminating an artificial barrier synchronization. To make this comparison useful, the actual barrier overhead (as measured by a microbenchmark) is subtracted from the overall runtime before calculating P_{barrier} . A higher P_D factor indicates a more effective communication overlap and better scalability.

3 EVALUATION AND IMPLICATIONS

This section describes analysis results for the selected microbenchmarks and proxy applications with a focus on idle waves and the automatic overlap communication and computation via computational wavefronts. The goal is not to provide a comprehensive analysis of each application but an assessment of desynchronization impact. Great care has been taken to separate the speedup observed by desynchronization from other desirable effects, such as the removal of synchronizing collectives or the reduction of communication volume.

3.1 MPI-augmented STREAM Triad

We start with the simple case of the pure-MPI version of the McCalpin STREAM Triad [1] loop (`A(:)=B(:)+s*C(:)`). It

TABLE 3: Structure of the MPI-parallel SpMVM implementation. Split-wait and non-split are implementation alternatives.

| Listing 1: CRS based SPMVM kernel | Listing 2: SPLIT-WAIT mode | Listing 3: NON-SPLIT mode |
|--|--|--|
| <pre> 1: double :: valA[nnz], b[nr], x[nr] ; 2: int :: colIdxA[nnz], rowPtrA[nr + 1], tmp ; 3: for row = 0 : nr - 1 do 4: tmp = 0.0 ; 5: for idx = rowPtrA[row] : rowPtrA[row + 1] - 1 do 6: tmp += valA[idx] * x[colIdxA [idx]] ; 7: end for 8: b[row] += tmp ; 9: end for </pre> | <pre> 1: while iter ≤ nlters do 2: MPI_Irecv ; 3: MPI_Isend ; 4: local_spMVM (A, x, b) ; 5: MPI_Wait ; * 6: remote_spMVM (A, x, b) ; 7: MPI_Barrier ; 8: swap (b, x) ; 9: end while </pre> | <pre> 1: while iter ≤ nlters do 2: MPI_Irecv ; 3: MPI_Isend ; 4: MPI_Wait ; * 5: local_spMVM (A, x, b) ; 6: remote_spMVM (A, x, b) ; 7: MPI_Barrier ; 8: swap (b, x) ; 9: end while </pre> |

* Two MPI_Wait routines wait for both MPI receive and send requests to complete.

allows a straightforward application of the Roofline model to predict the memory-bound parallel performance limit on a ccNUMA domain as $P = b_S/B_C$, where b_S is the domain memory bandwidth and $B_C = 12$ byte/flop is the code balance (assuming that streaming stores are used, i.e., write-allocate transfers do not apply). A constant overall working set of 2.4 GB (10^8 elements) is distributed evenly among the MPI processes. Communication is added after each sweep of the loop to mimic a real MPI-parallel program. The communication topology can be varied, but non-blocking point-to-point calls (MPI_Isend/MPI_Irecv) together with a final MPI_Waitall are used in all cases. The implementation uses open boundary conditions (i.e., process 0 ($n - 1$) only communicates with processes 1 and above ($n - 2$ and below) and bidirectional direct-neighbor communication (i.e., each MPI process i sends and receives 16 384 B to and from $i \pm 1$ after each STREAM phase).

3.1.1 Shape of computational wavefronts

Figure 1 shows MPI traces of the MPI-augmented STREAM Triad on 40 processes (two nodes of Meggie) with different communication topologies. In these figures, code execution is white and MPI waiting time is red. With pure next-neighbor communication as in (a), the number of concurrently active (i.e., code-executing) processes per ccNUMA domain settles in the vicinity of the performance saturation point⁵, as was already pointed out in [5]. In (b) one can observe the difference to next-pair communication ($d = \pm(1, 2)$): The developed desynchronized state (the computational wavefront) is about $3\times$ steeper, i.e., it has a smaller amplitude. This correlates with the higher idle wave velocity for longer-distance communication scenarios [9]. This restricts the communication-computation overlap, and the number of concurrently active processes per ccNUMA domain is higher than what is needed for saturation. With a mixed short-/long-range communication topology as in (c), computational wavefront structures on different scales overlap, i.e., two periodicities can be observed which emerge from the long- and short-distance communication, respectively. Compact long-distance communication as in (d) with $d = \pm(1, \dots, 12)$ causes high-velocity idle waves [9], which leads to steep computational wavefronts. Starting at a communication distance of at least $d = \pm(1, \dots, 8)$, the processes keep in lockstep for a system size of 40 ranks as in the example. This is due to the comparatively small system

size; idle waves are so fast that they leave the system in a single compute-communicate cycle.

3.1.2 Wavefront stability

The question arises how stable a developed computational wavefront is against disturbances like system noise or single one-off delay injections. After all, the translational symmetry of the system should not favor a particular position of the “lagger,” i.e., the slowest process. In all our measurements, natural system noise was never able to alter the shape or position of computational wavefronts. However, long one-off delay injections can. In Fig. 2 we show the results of an experiment on the Meggie system, where a fully developed computational wavefront state is disturbed by an injection on a ccNUMA domain different from the one where the lagger initially resides. To spark an idle wave, a series of floating-point divide operations are performed by rank 15 at time step 80 as illustrated by blue bars in the second and forth graph of Fig. 2. For one-off disturbances, we use a core-bound workload which does not impose an additional strain on the memory interface. The overall impact is independent of the nature of the disturbance though. As a consequence, after the ensuing idle wave has run out, the lagger shifts to the domain where the injection took place if the injection is strong enough. There is currently no first-principles understanding about what “strong enough” means; experimentally, we observe that the idle wave must at least be able to travel (despite the inevitable damping) far enough as to intrude the slowest socket.

3.2 Sparse Matrix-Vector Multiplication (SpMVM)

The multiplication of a sparse matrix with a dense vector ($\vec{y} = A\vec{x}$) is a central component in numerous numerical algorithms such as linear solvers and eigenvalue solvers. For large matrices, the performance of sparse matrix-vector multiplication (SpMVM) is memory bound on the node level due to its low computational intensity. Distributed-memory parallelization requires the matrix A and the vectors x and y to be distributed across MPI processes. This can cause significant communication overhead if the pattern of nonzeros in the matrix is very scattered.

An SpMVM kernel is usually the dominant part of a larger algorithm (such as Conjugate-Gradient); sometimes, several SpMVM kernels are executed in a back-to-back manner. Together with the properties described above, SpMVM constitutes an interesting test bed for desynchronization phenomena. In this section we investigate such a sequence of MPI-parallel SpMVMs, with left-hand side (LHS) and

5. A saturation point is the minimum number of processes required to achieve the maximum memory bandwidth on the ccNUMA domain.

TABLE 4: Measured walltime minimum, maximum, and median for execution (rows 1–3) and communication (rows 4–6) of one MPI-only SpMVM with the HHQ-large matrix on SuperMUC-NG, using strong scaling from 96 processes (two nodes) up to 1296 processes (27 nodes) using barriers between successive SpMVMs. Row 7 shows the mean per-process message sizes (transmitted via rendezvous protocol at small processes count till eager limit), and the last row denotes the median of the communication-to-execution time ratio (CER). Color coding is used as a guide to the eye (white to pink scale).

| Phase vs. Rank-order | 96-pe | 144-pe | 240-pe | 480-pe | 720-pe | 960-pe | 1296-pe | 96-ep | 144-ep | 240-ep | 480-ep | 720-ep | 960-ep | 1296-ep |
|------------------------|-------|--------|--------|--------|--------|--------|---------|-------|--------|--------|--------|--------|--------|---------|
| Exec min [ms] | 31.58 | 18.04 | 9.67 | 4.28 | 3.13 | 2.44 | 1.82 | 26.09 | 19.92 | 9.72 | 3.92 | 2.72 | 1.99 | 1.35 |
| Exec max [ms] | 64.92 | 45.73 | 27.55 | 13.49 | 10.53 | 8.25 | 6.24 | 53.36 | 36.22 | 22.13 | 11.49 | 7.98 | 6.59 | 4.76 |
| Exec median [ms] | 53.5 | 35.74 | 18.51 | 9.05 | 6.53 | 5.06 | 3.85 | 48.47 | 30.87 | 17.57 | 8.23 | 5.7 | 4.54 | 3.03 |
| Comm min [ms] | 20.7 | 15.38 | 6.72 | 5.62 | 3.57 | 4.03 | 2.51 | 9.28 | 8.5 | 3.73 | 2.48 | 2.04 | 1.18 | 2.45 |
| Comm max [ms] | 38.73 | 29.49 | 21 | 18.46 | 14.83 | 12.97 | 11.32 | 19.36 | 18.78 | 17.71 | 15.36 | 11.59 | 8.53 | 10.16 |
| Comm median [ms] | 29.48 | 24.56 | 16.17 | 14.75 | 11.28 | 9.78 | 7.99 | 16.17 | 15.01 | 14.41 | 12.17 | 8.96 | 6.62 | 6.13 |
| Mean P2P msg size [kB] | 2390 | 1460 | 957 | 480 | 302 | 213 | 153 | 1310 | 848 | 505 | 260 | 178 | 137 | 105 |
| CER median | 0.55 | 0.69 | 0.87 | 1.63 | 1.73 | 1.93 | 2.08 | 0.33 | 0.49 | 0.82 | 1.48 | 1.57 | 1.46 | 2.02 |

right-hand side (RHS) vectors swapped after every step. There is no explicit or implicit synchronization among MPI processes.

3.2.1 Implementation

A compressed storage format must be chosen for the sparse matrix so that the SpMVM can be carried out efficiently. On multicore CPUs, the standard Compressed Row Storage (CRS) format is typically a good choice. It allows for a compact implementation of the kernel that enables to exploit the relevant bottleneck (memory bandwidth) in many cases (see Listing 1 of Table 3). CRS requires one-dimensional arrays for matrix entries (`valA[]`), column indices (`colIdxA[]`), and row pointers (`rowPtrA[]`). If the matrix entries are in double precision and the indices are 32-bit integers, the minimum code balance for CRS-SpMVM is 6 byte/flop⁶ [31], [32].

In the MPI-parallel SpMVM implementation, contiguous blocks of matrix rows (and corresponding LHS and RHS vectors) are assigned to the processes so that the number of matrix nonzeros per process is as balanced as possible. Each process can compute the part of the SpMVM for which it already holds the LHS and RHS entries right away. Matrix entries outside of this column range require communication of the corresponding RHS values. Splitting the operation into “local” and “remote” kernels causes an additional memory traffic of $16/n_{nzr}$ byte per multiply-add because the local result vector must be updated twice in memory [31].

Two different implementations were tested:

- 1) **SPLIT-WAIT** mode: Communication is initiated with non-blocking MPI calls before the local SpMVM and finalized after it. Only after the call to `MPI_Wait` can the remote SpMVM kernel be executed. This allows for overlapping communication with the local SpMVM if the MPI implementation supports it; see Listing 2 of Table 3.
- 2) **NON-SPLIT** mode: The full non-blocking remote communication is initiated and finalized before the local and remote SpMVM kernels are called. This rules out any communication overlap by MPI; see

6. Per iteration, the kernel carries out 2 flops and causes a minimum data traffic of 8 byte for the matrix entry and 4 byte for the column index.

TABLE 5: Key specifications of symmetric sparse matrices.

| Matrix-order | Bandwidth | $n_{\text{electrons}} - n_{\text{sites}} - n_{\text{phonons}}$ [§] | $n_r = n_c^*$ | n_{nz}^* | n_{nzr}^\ddagger | Size [GB] [¶] |
|--------------|-----------|---|---------------|------------|--------------------|------------------------|
| HHQ-large-pe | high | 3 – 8 – 10 | 60988928 | 889816368 | 13 | 10.9 |
| HHQ-large-ep | low | 3 – 8 – 10 | 60988928 | 889816368 | 13 | 10.9 |
| HHQ-small-pe | high | 6 – 6 – 15 | 6201600 | 92527872 | 15 | 1.14 |
| HHQ-small-ep | low | 6 – 6 – 15 | 6201600 | 92527872 | 15 | 1.14 |

[§] The described quantum system comprises $n_{\text{electrons}}$ electrons on n_{sites} lattice sites coupled to n_{phonons} phonons.

^{*} n_r , n_c and n_{nz} are the total number of rows, columns and non-zero entries of sparse square matrix respectively.

[‡] The inner loop length of the CRS SpMVM kernel $n_{nzr} (\approx \frac{n_{nz}}{n_r})$ is the average number of non-zero entries in each row of the sparse matrix.

[¶] Data set size is estimated by $12n_{nz} + 4n_r$ (eight byte per matrix entry and four byte for column indices).

Listing 3 of Table 3. In this case, the two kernel calls could be fused for improved computational intensity, but we want to keep the properties of the underlying kernels unchanged for the experiments shown here.

3.2.2 Test matrices

For benchmarking we use real, symmetric matrices that describe a strongly correlated one-dimensional electron-phonon system in solid state physics (Holstein-Hubbard Hamiltonian) [33]. The key specifications of the matrices are shown in Table 5. Due to the moderate number of nonzeros per row (13 and 15, respectively), the minimum code balance is about 6.9 byte/flop and 7.1 byte/flop, respectively (assuming optimal reuse of the right-hand side vector; see also [32]). Overall we use four variants that emerge from two different problem sizes (numbers of electrons, phonons, and lattice sites) and two different orderings of the degrees of freedom (phonons first vs. electrons first). The “phonons first” numbering (labeled “pe”) produces a more scattered matrix, whereas with “electrons first” (labeled “ep”) the nonzeros are closer to the diagonal (see (a) and (b) of Figs. 3 and 4). The motivation behind the different problem sizes (10.9 GB and 1.135 GB for the matrix, respectively) is that the smaller problem can fit into the aggregate last-level cache of the CPUs in the chosen clusters at a moderate node count, removing the memory bandwidth bottleneck at the socket level. The matrices were generated using the scalable matrix collection (`ScMaC`) library.⁷

7. The `ScMaC` library allows for scalable generation of large matrices related to quantum physics applications. The open source implementation is available for download at <https://bitbucket.org/essex/matrixcollection/> and documentation of matrices can be found at https://alvbit.bitbucket.io/scamac_docs/_matrices_page.html.

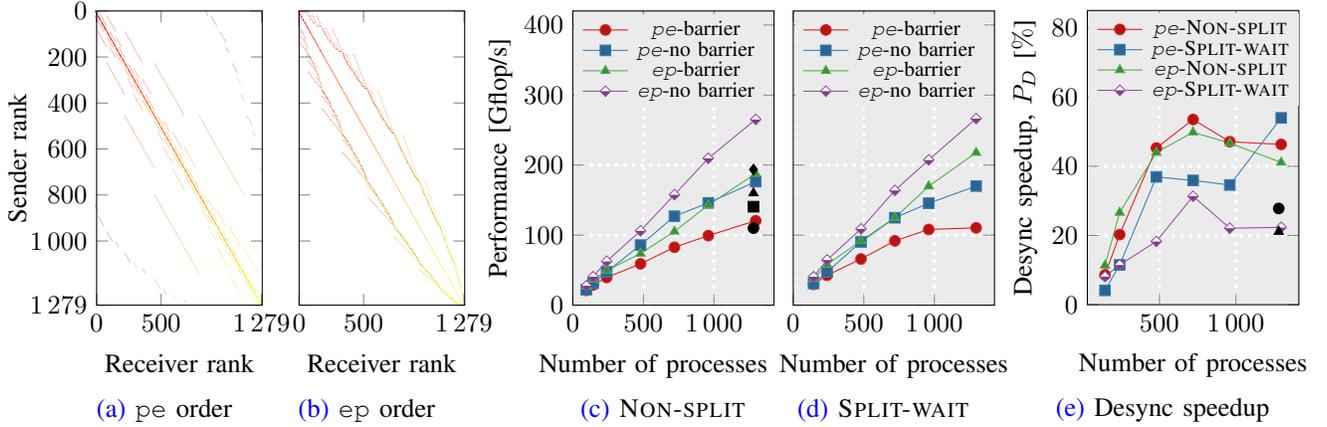


Fig. 3: (a-b) The communication topology of HHQ-large matrices using periodic boundary conditions in the wider pe order (left/right bandwidth of 41385344) and slimer ep order (left/right bandwidth of 12907776), respectively. (c-d) Strong scaling performance for 10^6 iterations on SuperMUC-NG using NON-SPLIT and SPLIT-WAIT algorithms, respectively. (e) Performance boost as a result of improved overlap in the absence of barriers in the NON-SPLIT and SPLIT-WAIT implementation on the SuperMUC-NG system. Markers in black at 1280 processes mark performance and the P_D factor for the NON-SPLIT algorithm on the Meggie system.

3.2.3 Matrix topology and communication schemes

The communication characteristics of distributed-memory SpMVM depend strongly on the structure of the sparse matrix. Thus we expect the pe versions of the Hamiltonians to have larger communication overhead. The sparsity pattern impacts the node-level performance and bandwidth saturation as well, however, due to the indirect access to the RHS vector. Table 4 shows execution and communication properties of one SpMVM execution with the large pe and ep matrices, respectively, for different numbers of MPI processes on the SuperMUC-NG system. To keep the MPI processes in lockstep, an MPI barrier was called before the SpMVM (the barrier time is not part of the reported communication time). The data shows that the more scattered pe matrix clearly causes much higher communication overhead, especially at lower process counts where pe incurs more communication partners per rank than ep . It can be seen that the communication overhead in SpMVM is significant but not dominant at 96 processes. The last row of the table shows the median of the communication-to-execution time ratio (CER), which can serve as a rough indicator of communication boundedness. The minimum, maximum, and median numbers for execution and communication times indicate that even in a single SpMVM without desynchronization there is considerable variation in both metrics across processes.

In order to fathom the consequences of desynchronization, we compare the barrier version of the benchmark (i.e., a barrier after each SpMVM) with the barrier-free version. Performance for the barrier version was calculated by subtracting the actual barrier time (as determined by a separate benchmark) from the measured walltime. Any observed speedup of the barrier-free version must thus be caused by automatic overlap of communication via desynchronization of processes. Figures 3(c) and (d) show strong scaling performance for the HHQ-large matrices on SuperMUC-NG. The behavior of the split (c) and non-split (d) variants is similar. Note that the best version (ep without barrier) is

strongly communication bound at 1296 processes: Assuming a socket memory bandwidth of 100 GB/s, the Roofline limit is 760 Gflop/s, while the observed performance is only about 270 Gflop/s. The speedup P_D (defined in Section 2.3) caused by bottleneck evasion via desynchronization is shown in Fig. 3(e). Depending on the matrix structure and the communication scheme, performance gains between 20% and 55% (out of a theoretical maximum of 100%) can be observed. This goes with a significant improvement in scalability.

Although the details of matrix partitioning and communication topology add a considerable amount of variation, the speedup P_D shows the expected behavior along the scaling curve: It starts out small because the communication overhead is small (albeit significant), providing only minor opportunity for overlapping. As the number of processes grows, this benefit becomes larger until at some point communication and computation take roughly the same amount of time. This is when no further speedup can be expected. Scaling up further, the benefit drops because communication is dominant. One can also see that the non-split communication scheme (circles and triangles) generally shows higher speedup than the split-wait scheme (squares and diamonds). This is expected because no-split has no potential for asynchronous MPI communication in the lock-step case; this leaves more opportunity for overlap in the desynchronized case.

Note that the “slimmer” ep matrix with its smaller communication radius supports stronger desynchronization due to a lower idle wave velocity [5], [9]. This effect is counteracted, however, by the smaller absolute communication overhead of ep , which is why no clear advantage of ep in terms of overlap can be observed in Fig. 3(e). Note also that particular process counts can interact with the inherent structure of the matrix, which leads to more or less favorable communication topologies and adds extra variation to the scaling behavior. The general trend is similar but less pronounced on the Meggie system, as shown by the

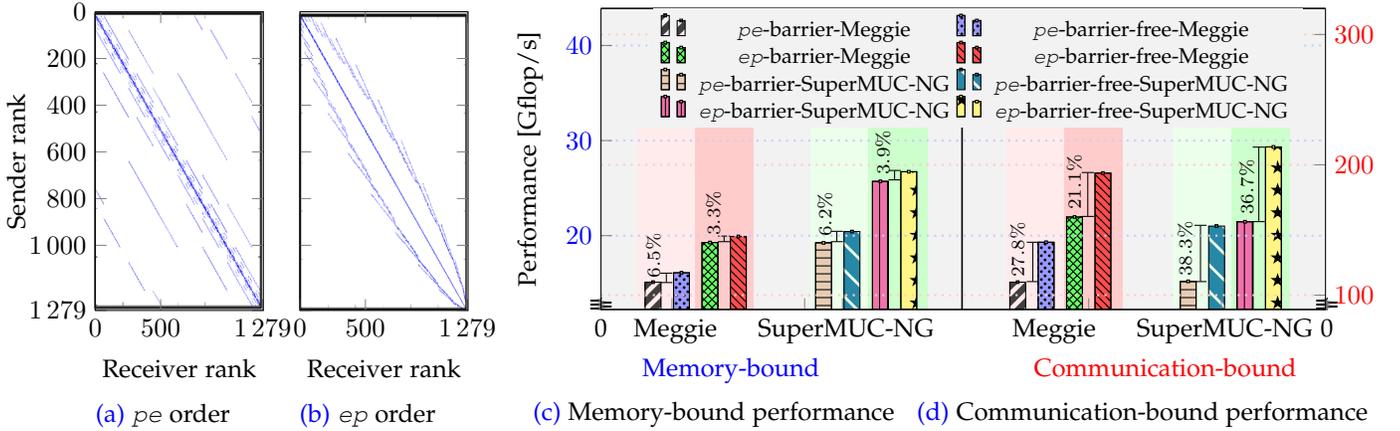


Fig. 4: (a-b) Communication topology of HHQ -small-order pe and HHQ -small-order ep Hamiltonian matrices. (c) Performance (blue y -axis) and speedup for barrier-free (desynchronized) execution in the memory-bound case (60 processes on Meggie and 96 processes on SuperMUC-NG) and in non-split mode. The percentage increments denote the speedup of the no-barrier versions. (d) Same data but for the communication-bound case with performance on the red y -axis (1280 processes on Meggie and 1296 processes on SuperMUC-NG).

black markers in Fig. 3. This can be attributed to the larger fraction of cores needed per ccNUMA domain to achieve memory bandwidth saturation compared to SuperMUC-NG.

In this experiment, the matrices were large enough to keep the execution memory bound even at large scale, which made memory bandwidth the relevant bottleneck for desynchronization. The HHQ -small matrices in Table 5 fit into the aggregate last-level cache (LLC) on 23 nodes and beyond (for Meggie) and 18 nodes (for SuperMUC-NG), respectively. The LLC shows much better bandwidth scalability than the memory interface, so the bottleneck shifts to the network communication for larger node counts. In Fig. 4 we show performance and speedup results for the small matrices on small (memory bound, (c)) and large (network bound (d)) numbers of nodes on the two test systems. As before, the actual barrier duration has been subtracted from the execution time of the version with barriers so that the speedup observed when removing the barrier can be attributed to desynchronization.

We concentrate on the non-split variant here. For small numbers of processes (Fig. 4(c)), the speedups are small, which is expected because the memory bandwidth is the bottleneck and the communication overhead is small so that there is little opportunity for overlapping. For the in-memory case, the small ep matrix shows the same desynchronization speedup of 3.9% as its large counterpart. The more communication-intensive pe matrix, on the other hand, shows an extra speedup of 1.45% (this data is not contained in the figure). The behavior persists in similar ways on both systems. In the network-bound case (Fig. 4(d)), however, the now-dominant communication can overlap with code execution, which yields speedups of 38% and 37% in the pe and ep cases, respectively. We attribute the minor difference in behavior between the two matrices to the small message sizes, which make most of the point-to-point communication latency bound. Compared to the large-matrix cases, the in-cache execution leads to lower speedup by desynchronization (4.4% for ep and 8% for pe).

On the Meggie system, the lower CER causes an additional boost by 15.6% (ep) and 10.5% (pe), respectively.

In summary, our SpMVM experiments have shown that significant performance speedups can be obtained via desynchronization when the execution is limited by memory bandwidth or communication and synchronizing collectives (i.e., barriers or collectives with synchronizing implementations) between back-to-back SpMVMs are removed. Note that we relied on the natural irregularities of the sparse matrices to destabilize the lock-step pattern; no explicit noise injection was required.

Key takeaway: In MPI-parallel SpMVM, speedup by automatic overlap of communication and computation is facilitated by (i) a larger communication overhead, which is connected with a more spread nonzero distribution in the matrix, (ii) a slow idle wave propagation speed, which is caused by a low matrix bandwidth, and (iii) a slow synchronized baseline that uses the simple non-split communication scheme.

3.3 Chebyshev Filter Diagonalization (ChebFD)

The Chebyshev filter diagonalization (ChebFD) is a polynomial filtering algorithm that is popular in quantum physics and chemistry. It allows to compute parts of eigenvalue spectra of large sparse matrices and is amenable to multiple node-level and communication optimizations [2], [8]. For example, a blocking parameter (number of block vectors n_b) enables flexible tuning of the code balance of the main iteration loop; larger block size causes lower code balance.

3.3.1 Implementation

Our open-source⁸ implementation of ChebFD is built with the GHOST [34] building block library using tailored kernels and 64 bits global and 32 bits local index size, respectively. It employs the standard CRS sparse matrix data format and

⁸. Available as part of the GHOST package at <https://bitbucket.org/essex/ghost>

Algorithm: Structure of the MPI-parallel CHEBFD($H, \vec{U}, \vec{W}, \vec{X}$) implementation. NON-SPLIT, SPLIT-WAIT and PIPELINE modes are implementation alternatives.

| | NON-SPLIT mode | SPLIT-WAIT mode | PIPELINE mode |
|---|---|--|---|
| define vector blocks $\vec{U}, \vec{W}, \vec{X}$ | for $k = 0 : \frac{n_s}{n_b} - 1$ do | for $k = 0 : \frac{n_s}{n_b} - 1$ do | for $p = 0 : n_p - 1$ do |
| twice SPMMV() $\vec{U} \leftarrow (\alpha \vec{H} + \beta) \vec{X},$ $\vec{W} \leftarrow 2(\alpha \vec{H} + \beta) \vec{U} - \vec{X}$ | for $p = 0 : n_p - 1$ do swap(\vec{W}, \vec{U}); comm_init_finalize(\vec{U}); $\vec{W} \leftarrow 2(\alpha \vec{H} + \beta) \vec{U} - \vec{W}$ | for $p = 0 : n_p - 1$ do swap(\vec{W}, \vec{U}); comm_int(\vec{U}); local_kernel; | swap(\vec{W}, \vec{U}); comm_init(\vec{U}_0); MPI_wait(); for $k = 0 : \frac{n_s}{n_b} - 2$ do |
| BAXPY() and BSCAL() $\vec{X} \leftarrow g_0 c_0 \vec{X} +$ $g_1 c_1 \vec{U} + g_2 c_2 \vec{W}$ | $\vec{\eta}_p \leftarrow \langle \vec{W}, \vec{U} \rangle;$ $\vec{\mu}_p \leftarrow \langle \vec{U}, \vec{U} \rangle;$ $\vec{X} \leftarrow \vec{X} + g_p c_p \vec{W};$ | MPI_wait(); remote_kernel; | comm_init(\vec{U}_1); kernel $_k$ (); MPI_wait(); |
| Main loop: NON-SPLIT/SPLIT/PIPELINE mode | end for end for | end for end for | end for kernel $_{\frac{n_s}{n_b}-1}$ (); end for |

TABLE 6: Key specifications of Hamiltonian matrices for the ChebFD application.

| Matrix | Traits [§] | Data-type | $n_r = n_c$ | n_{nz} | n_{nzpr} | Size [GB] [‡] |
|----------|--------------------------|----------------|-------------|------------|------------|------------------------|
| TOPI-EHN | $m_x-m_y-m_z-128-128-64$ | complex double | 268435456 | 3487563776 | 13 | 70.8 [‡] |
| SPIN26* | 26-13-1 | double | 10400600 | 145608400 | 14 | 1.8 |
| SPIN28* | 28-14-1 | double | 40116600 | 601749000 | 15 | 7.4 |
| SPIN30* | 30-15-1 | double | 155117520 | 2481880320 | 16 | 30.4 |

[§] Traits for the TOPI-EHN matrix represent the $m_x-m_y-m_z-n_x-n_y-n_z$, while for the SPIN matrices mark n_{up} -disorder-seed.

[‡] Eight (sixteen) byte for double (complex double) precision numbers of matrix entries, and four-byte indexing for 32-bit integers are considered.

* SPIN matrices comprising periodic boundary conditions with *upper* count are equal to the half of the *lower*.

uses row-major ordering within a block of n_b vectors to facilitate SIMD vectorization. The algorithm contains a sparse matrix-multiple-vector multiplication (SpMMV) and a series of BLAS-1 vector operations. Non-blocking communication is performed via MPI_Isend/MPI_Irecv/MPI_Waitall sequences. Asynchronous progress was disabled in the Intel MPI library as well as in GHOST. The code supports hybrid MPI+OpenMP parallelization; unless otherwise noted, we use the pure MPI version here.

We compare three communication schemes:

- 1) **NON-SPLIT mode:** blocking MPI communication, followed by computation. It performs computation of ChebFD polynomials to a block \vec{U} of n_b vectors at a time.
- 2) **SPLIT-WAIT mode:** naive implicit overlapping of non-blocking MPI communication of the non-local vector elements with local computations. Only after completing the outstanding receives via MPI_Wait can the remote part of the kernel be done. It increases the main memory data traffic since the local result vector must be updated twice.
- 3) **PIPELINE mode:** pipelined asynchronous non-blocking MPI communication with the subspace blocking scheme, which does not require any extra memory traffic. If n_s is the subspace size, for sufficiently large $\frac{n_b}{n_s}$ this scheme enables explicit effective overlap of computation on the current subblock (local_kernel) with the communication needed for the next sub-block. Details can be found in [8].

The polynomial filter degree (we use $n_p = 500$ here, which is a relevant value for practical applications) applies

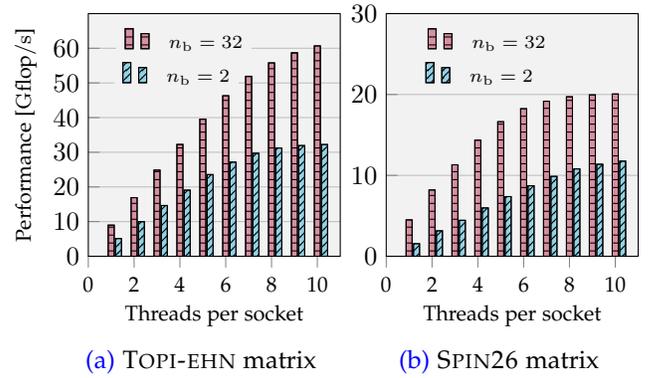


Fig. 5: Single-socket performance scaling of ChebFD with OpenMP on a contention domain of the Meggie system for the block vector sizes of 2 and 32 and the (a) topological insulator matrix (1.76 speed up at single thread) and (b) SPIN26 matrix (2.89 speed up at single thread owing to efficient data accesses), respectively.

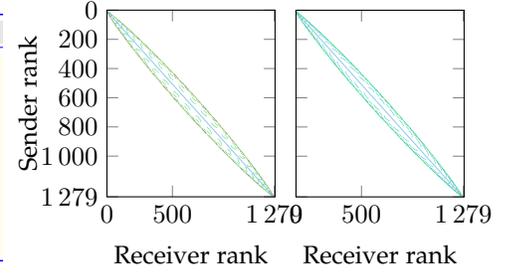
independently to all search vectors n_s . In the algorithm, n_{iter} is the number of iterations; the number of sought inner eigenvalues of a topological insulator ($n_s = 128$) is taken to be a multiple of block vector size ($n_b = 2$ or $n_b = 32$) for simplicity. In the implementation, a single, fused, MPI-parallel CHEBFD() kernel (marked in the algorithm) facilitates cache reuse. Moreover, the global reduction needed for computation of the polynomial filter coefficients $\vec{\eta}_p$ and $\vec{\mu}_p$ can be postponed until after the iteration loop, which eliminates all (possibly) synchronizing collectives; the communication topology is thus entirely determined by the matrix structure. The sparse matrices and the code balance of the algorithm (optimistically⁹ (260/ n_b + 80)/146 B/flop for double complex data and (48/ n_b + 40)/19 B/flop for double precision real data) were thoroughly investigated in [2].

⁹ “Optimistically” means here that one assumes the minimum possible data transfer, i.e., each data element is only loaded once and then reused from cache as often as necessary.

TABLE 7: (a) ChebFD message profile of topological insulator (TOPI-EHN) matrices of same problem sizes. It encompasses nine domain configurations (M1–M9) with n_{nodes} that facilitates a good mixture of eager and rendezvous messages. (b)–(c) Communication matrices of Spin matrices of multiple sizes.

| | m_x - m_y - m_z | Communication distances ($n_{\text{nodes}} = 64$) | Message sizes [kB] ($n_{\text{nodes}} = 64$) |
|----|---|---|--|
| M1 | 1-1- n_{nodes} | $\pm 1, -20, -19$ | 1050, 105, 0.128 |
| M2 | 1- n_{nodes} -1 | $\pm 1, \pm 20, \pm 19$ | 1050, 48, 8 |
| M3 | n_{nodes} -1-1 | ± 1 | 1050 |
| M4 | $1-\sqrt{n_{\text{nodes}}}-\sqrt{n_{\text{nodes}}}$ | $\pm 1, -20, \pm 160, \pm 159, -19$ | 1050, 105, 48, 8, 0.128 |
| M5 | $\sqrt{n_{\text{nodes}}}-1-\sqrt{n_{\text{nodes}}}$ | $\pm 1, -20, -19$ | 1050, 105, 0.128 |
| M6 | $\sqrt{n_{\text{nodes}}}-\sqrt{n_{\text{nodes}}}-1$ | $\pm 1, -20, -140, -19, -141$ | 1050, 48, 48, 8, 8 |
| M7 | $2-4-\sqrt{n_{\text{nodes}}}$ | $\pm 1, -20, -480, -160, -481, -159, -19$ | 1050, 105, 48, 48, 8, 8, 0.128 |
| M8 | $2-\sqrt{n_{\text{nodes}}}-4$ | $\pm 1, -20, -560, -80, -561, -79, -19$ | 1050, 105, 48, 48, 8, 8, 0.128 |
| M9 | $\sqrt{n_{\text{nodes}}}-2-4$ | $\pm 1, -20, -80, -79, -81, -19$ | 1050, 105, 96, 8, 8, 0.128 |

(a) TOPI-EHN



(b) SPIN26/30

(c) SPIN28

3.3.2 Test matrices

Two types of test matrices and scaling scenarios were considered for ChebFD: weak scaling for a topological insulator problem (TOPI-EHN) [35] and strong scaling for a spin system (*Spin*). For TOPI-EHN (see Table 6), the Hamiltonian matrix emerges from a three-dimensional mesh with four degrees of freedom (DOFs) per mesh point and thus exhibits a rather regular, stencil-like structure. In order to study different communication topologies, we chose a local (per-node) problem size of $n_x \times n_y \times n_z = 128^2 \times 64$ and a global size of $n_x m_x n_y m_y n_z m_z$, where the m_i are the number of nodes in each Cartesian dimension. Each matrix row has 13 complex double-precision nonzero entries, leading to a matrix size of $(16 + 4) B \times 128 \times 128 \times 64 \times 13 = 273 \text{ MB}$, which is much larger than the available LLC on the benchmark platforms. Periodic boundary conditions in the x and y directions lead to outlying diagonals in the matrix corners.

The real-valued SPIN matrix is used in three different, fixed sizes (see Table 6) and has a structure that leads to more communication overhead and a larger impact of memory latency on the node-level performance. The socket-level performance scaling data (using OpenMP) in Fig. 5 reveals interesting differences between the two matrices: Although increasing the block size from $n_b = 2$ to $n_b = 32$ improves the performance in both cases as expected from the reduced code balance, the impact on the scaling behavior is different: While the SPIN-26 matrix starts off with very low performance on a single thread with $n_b = 2$ and thus shows good scaling in this case (optimistic upper bandwidth limit at 14.8 Gflop/s), the TOPI-EHN matrix shows strong bandwidth saturation and achieves 90% of the optimistic maximum of 34.8 Gflop/s already at eight cores. At $n_b = 32$, the code balance is strongly reduced in both cases, so one expects weaker saturation as the pressure on the memory interface is lowered and in-cache effects become more prominent [2], [7]. While this can be clearly observed in case of TOPI-EHN, saturation actually becomes stronger for SPIN-26. This is rooted in a better utilization of the memory interface and a lower impact of latency due to the vector blocking technique.

All matrices are generated on the fly using the ESSEX_PHYSICS library¹⁰. Table 6 illustrates the key specifications of both types of matrices.

10. The ESSEX_PHYSICS library is an open-source software of ESSEX project, available for download at <https://bitbucket.org/essex/physics/src/master/>.

3.3.3 Decomposition, communication schemes and block vector sizes

TOPI-EHN matrices: Figure 6 shows the weak scaling performance of ChebFD with the TOPI-EHN case in nine domain decomposition variants and three communication modes. While the bars show the observed performance with barrier-free code, the red line denotes the performance with an explicit barrier added at each new polynomial degree p (specifically when the \vec{W} and \vec{V} vectors are swapped). As expected, the pipelined mode exhibits no noticeable performance hit from the barrier because the vector swap occurs between the p and k loops. The communication topology depends on the domain decomposition, as do the communication data paths used (intra- vs. inter-node). Hence, we expect a significant dependence of the performance on the decomposition. Unlike in Sp-MVM, where the matrix data dominates the code balance, ChebFD has a much stronger dependence on the vector data; consequently, the split-wait variant shows the worst performance among all schemes and decompositions due to extra memory traffic. Favorable configurations for the sync-free code are (in order of descending performance): {M3, M2, M6, M5, M1, M9, M7, M8, M4}, which is also roughly in rising idle wave speed order. With explicit barriers, on the other hand, M7 is the worst configuration and M8 is among the top performers. Configuration M3 uses a one-dimensional decomposition and thus has an unfavorable communication pattern. However, it features only direct next-neighbor communication, which leads to the slowest possible idle wave speed and thus the highest potential for desynchronization.

M4 is the worst case among all studied decomposition variants, and its CER is even larger than that of M3. For instance, at the blocking vector size of 32, we observe with the synchronized non-split variant $\{T_{exec} [\text{ms}], T_{comm} [\text{ms}], \text{CER}\} = \{214.6, 30, 0.14\}$ for M3 and $\{258.6, 43, 0.17\}$ for M4. Thus, the configuration M4 spends more time communicating, while M3 has the slowest idle wave; both affect the overall performance. In {NON-SPLIT, PIPELINE, SPLIT-WAIT} mode, the performance increment between the worst M4 and best M3 corner cases is {11.1, 9, 6.9}% with $n_b = 2$ and {6.1, 5.1, 2.8}% with $n_b = 32$ at 1280 MPI processes on Meggie. The $n_b = 32$ cases exhibit a stronger slowdown from the non-split to the split version due to the dominant data traffic from the vector blocks. The pipeline version is always better than non-split in the synchronized scenario,

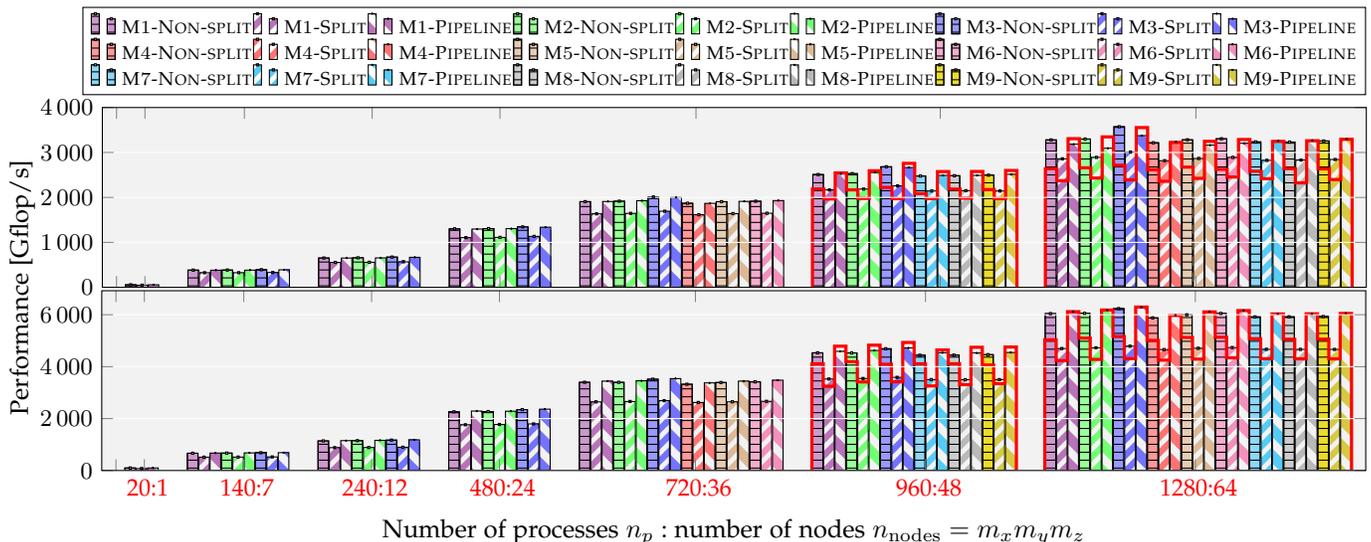


Fig. 6: Weak scaling performance of ChebFD with TOPI-ENH matrices in nine domain decompositions (M1–M9 in different colors) with three communication schemes (NON-SPLIT in horizontal lines, SPLIT-WAIT in north east hatch lines, PIPELINE in north west hatch lines) on the Meggie cluster. The number of MPI processes and compute nodes are shown on the x -axis. Top: block vector size of $n_b = 2$, bottom: $n_b = 32$. The red line indicates performance with a barrier in each p iteration (see text).

while in the naturally desynchronized case, the non-split version is generally on par or even better for the more saturating case. Contrarily, the non-split version suffers a higher performance hit at $n_b = 2$ with synchronizing barriers in place.

Key takeaway: The domain decomposition that allows lower idle wave speeds is better suited for automatic communication overlap.

SPIN matrices: Figure 7 shows strong scaling results for the SPIN matrices. Due to the working set size, SPIN-28 can only be used on 36 nodes and more, and SPIN-30 requires 64 nodes at least. Aggregate LLC sizes of $\{0.05, 0.35, 0.6, 1.2, 1.8, 2.4, 3.2\}$ GB are available for $\{60, 120, 240, 480, 720, 960, 1280\}$ MPI processes on the Meggie system. As a consequence, contrary to the SPIN28 and SPIN30 matrices, ChebFD with the SPIN26 matrix starts to be cache bound from 720 processes up.

In NON-SPLIT mode and on 64 nodes on Meggie, we observe $\{T_{exec} [\text{ms}], T_{comm} [\text{ms}], \text{CER}\} = \{0.6, 0.41, 0.68\}$, $\{2.38, 1.86, 0.78\}$ and $\{10.2, 7, 0.69\}$ for SPIN26, SPIN28 and SPIN30, respectively. Similarly, the range of P2P message sizes is $\{V_{min} [\text{B}], V_{max} [\text{kB}]\} = \{16, 130\}$ (SPIN26), $\{64, 501.5\}$ (SPIN28) and $\{32, 1939\}$ (SPIN30). These matrices cause significantly more communication overhead than TOPI-ENH, which leads to more opportunity for desynchronized execution and communication overlap in the NON-SPLIT case. The case with $n_b = 32$ shows stronger socket-level saturation here, so it has a higher potential for desynchronization than $n_b = 2$, which can be observed in the data in Fig. 7. There is no prominent advantage of explicit overlap (PIPELINE). In fact, the speedup from removing the barrier synchronization in the non-split version grows with increasing communication volume along the strong scaling curve and at certain points it becomes competitive with the

PIPELINE version. Also, the barrier-free NON-SPLIT variant is consistently worse for $n_b = 2$, as expected.

Key takeaway: Overlapping via explicit programming techniques may not be necessary for strongly bandwidth-saturating code with large (but not dominant) communication overhead due to the presence of natural overlap by desynchronization.

4 OUTLOOK AND FUTURE WORK

Using MPI-parallel synthetic benchmarks and application programs we investigated the consequences of desynchronization via bottlenecks in bulk-synchronous parallel code. Using a memory-bound microbenchmark we showed that there is a strong positive correlation between idle wave speed and the slope of a computational wave, indicating that automatic communication-computation overlap can be more effective in settings with low idle wave speeds. Using one-off idle injections we also showed that a stable computational wave can be shifted along the MPI rank dimension without losing its basic properties.

For back-to-back sparse matrix-vector multiplications, we demonstrated that speedup by automatic desynchronization is facilitated by large communication overhead, slow idle wave speed, and a simple non-split communication scheme. Our investigation of a Chebyshev filter diagonalization application showed that a more compact communication topology, which can be affected by domain decomposition strategies that allow for slower idle waves, enables more effective communication overlap. Depending on the underlying problem (and thus the sparse matrix structure), forcing overlap by explicit programming may not even be required.

We took great care to separate the effects of overhead reduction via elimination of collective communication from

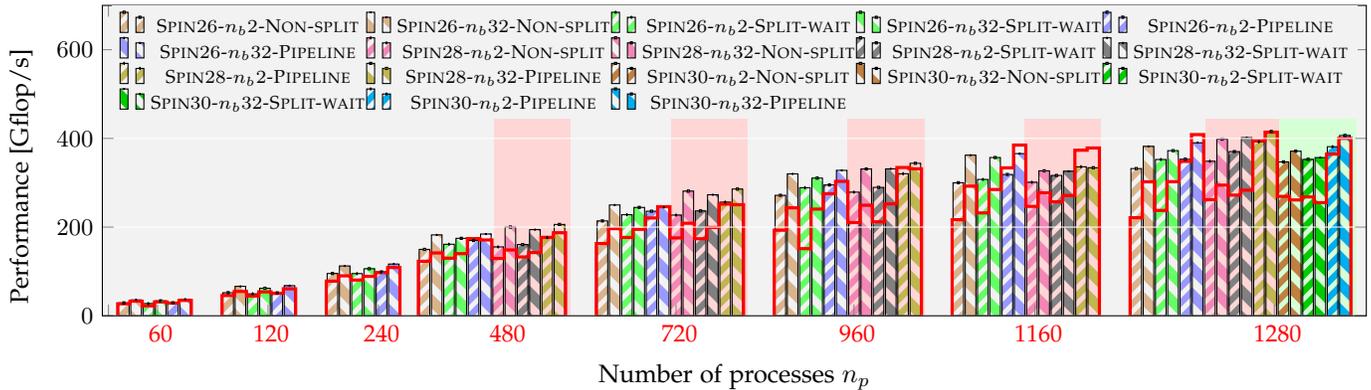


Fig. 7: Strong scaling performance of ChebFD with SPIN matrices of diverse sizes (SPIN26 in gray, SPIN28 in red, SPIN30 in green background) using various communication schemes (NON-SPLIT, SPLIT-WAIT, PIPELINE in different colors) with block vector sizes of $n_b = 2$ (north east hatch lines) and $n_b = 32$ (north west hatch lines) on the Meggie system. The red line indicates performance with a barrier in each p iteration.

the actual benefit of desynchronization. Overall, our results show that bottleneck evasion by desynchronization can be regarded as a performance optimization technique, and that forcing a parallel program into lock-step may be the wrong course of action in some settings.

Future work: In this work we have only considered memory bandwidth as the relevant bottleneck in desynchronization phenomena, but we have reason to assume that other bottlenecks, such as the compute node network injection bandwidth or the network topology can effect similar behavior in parallel codes. In future work we will explore this option further. In addition we have as yet no rigorous proof of instability for bottleneck-bound programs, which is why we work towards an analytic description of desynchronization processes that goes beyond idle wave speed.

In order to study out-of-lockstep behavior in more detail, we are working on a message passing and threading simulator that can simulate large-scale applications while taking the socket-level properties of code into account. It can explore parallel program dynamics further in a controlled environment, saving resources and time on real systems and allowing for advanced architectural exploration.

ACKNOWLEDGMENTS

This work was supported by KONWIHR, the Bavarian Competence Network for Scientific High Performance Computing in Bavaria, under the project name “OMI4papps.” We are indebted to LRZ Garching for granting CPU hours on SuperMUC-NG. We wish to thank Andreas Alvermann for his ScaMaC library and the admin team at NHR@FAU for excellent technical support on Meggie system.

REFERENCES

- [1] J. D. McCalpin *et al.*, “Memory bandwidth and machine balance in current high performance computers,” *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19–25, 1995.
- [2] A. Pieper, M. Kreutzer, A. Alvermann, M. Galgon, H. Fehske, G. Hager, B. Lang, and G. Wellein, “High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations,” *Journal of Computational Physics*, vol. 325, pp. 226–243, 2016.
- [3] E. C. Carson, “Communication-avoiding krylov subspace methods in theory and practice,” Ph.D. dissertation, UC Berkeley, PZ, Italy, 2015.
- [4] P. Ghysels and W. Vanroose, “Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm,” *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014, 7th Workshop on Parallel Matrix Algorithms and Applications. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819113000719>
- [5] A. Afzal, G. Hager, and G. Wellein, “Desynchronization and Wave Pattern Formation in MPI-Parallel and Hybrid Memory-Bound Programs,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., vol. 12151 LNCS. Cham: Springer International Publishing, 2020, pp. 391–411, cRIS-Team Scopus Importer:2020-07-10.
- [6] —, “Propagation and Decay of Injected One-Off Delays on Clusters: A Case Study,” in *Proceedings - IEEE International Conference on Cluster Computing, ICC*, vol. 2019-September. Institute of Electrical and Electronics Engineers Inc., 2019, cRIS-Team Scopus Importer:2019-11-29.
- [7] M. Kreutzer, A. Pieper, G. Hager, G. Wellein, A. Alvermann, and H. Fehske, “Performance engineering of the Kernel Polynomial Method on large-scale CPU-GPU systems,” in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 417–426.
- [8] M. Kreutzer, D. Ernst, A. R. Bishop, H. Fehske, G. Hager, K. Nakajima, and G. Wellein, “Chebyshev filter diagonalization on modern manycore processors and GPGUs,” in *High Performance Computing*, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, Eds. Cham: Springer International Publishing, 2018, pp. 329–349.
- [9] A. Afzal, G. Hager, and G. Wellein, “Analytic Modeling of Idle Waves in Parallel Programs: Communication, Cluster Topology, and Noise Impact,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds., vol. 12728 LNCS. Springer Science and Business Media Deutschland GmbH, 2021, pp. 351–371, cRIS-Team Scopus Importer:2021-08-20.
- [10] —, “Analytic performance model for parallel overlapping memory-bound kernels,” *Concurrency and Computation: Practice and Experience*, Jan 2022. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1002/cpe.6816>
- [11] X. Zhao, M. Jahre, and L. Eeckhout, “Hsm: A hybrid slowdown model for multitasking gpus,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020, pp. 1371–1385.
- [12] A. Afzal, G. Hager, and G. Wellein, “Delay flow mechanisms on clusters,” poster at EuroMPI 2019, September 10–13, 2019, Zurich, Switzerland. [Online]. Available: https://hpc.fau.de/files/2019/09/EuroMPI2019_AHW-Poster.pdf
- [13] F. Petrini, D. J. Kerbyson, and S. Pakin, “The case of the missing su-

- percomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 55–55.
- [14] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson *et al.*, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003, pp. 10–10.
- [15] P. Terry, A. Shan, and P. Huttunen, "Improving application performance on HPC systems with process synchronization." *Linux Journal*, no. 127, pp. 68–71, 2004.
- [16] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology*. IEEE, 2004, pp. 387–390.
- [17] D. Tsafir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 303–312.
- [18] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Cluster Computing, 2006 IEEE International Conference on*. IEEE, 2006, pp. 1–12.
- [19] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 19.
- [20] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 852–863.
- [21] H. Weisbach, B. Gerofi, B. Kocoloski, H. Härtig, and Y. Ishikawa, "Hardware performance variation: A comparative study using lightweight kernels," in *High Performance Computing*, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, Eds. Cham: Springer International Publishing, 2018, pp. 246–265.
- [22] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.
- [23] S. Markidis, J. Vencels, I. B. Peng, D. Akhmetova, E. Laure, and P. Henri, "Idle waves in high-performance computing," *Physical Review E*, vol. 91, no. 1, p. 013306, 2015.
- [24] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar, "Local recovery and failure masking for stencil-based applications at extreme scales," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.
- [25] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf, "Identifying the root causes of wait states in large-scale parallel applications," *ACM Trans. Parallel Comput.*, vol. 3, no. 2, pp. 11:1–11:24, Jul. 2016.
- [26] A. Kolakowska and M. A. Novotny, "Desynchronization and speedup in an asynchronous conservative parallel update protocol," *arXiv preprint cs/0409032*, 2004.
- [27] A. Afzal, G. Hager, and G. Wellein, "Physical Oscillator Model for Parallel Distributed Computing," 2021, poster at ISC High Performance 2021. [Online]. Available: https://www.researchgate.net/publication/354208484_Physical_Oscillator_Model_for_Parallel_Distributed_Computing?_sg=IVYEm3XW4E92Isf55nIWTxkXYhgPb13cA2Zi3zbsaP-YcPn2zMsmYHnIURLA_eADEZioHeF0aYKglpaCYNIPew0H4GtDTB14Q-E_avYu.W3TTE5Nzo7hjNz45wagExeFeJB8qTBJQe764hOdV9pK0lbGaFk1mulggkEpWAVEETapO_wOS8AKyPVJ1-4g0WA
- [28] D. an Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel *et al.*, "Score-p: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*. Springer, 2011, pp. 85–97.
- [29] T.-T. Pham, M. Pister, and P. Couvée, "Recurrent neural network for classifying of hpc applications," in *2019 Spring Simulation Conference (SpringSim)*, 2019, pp. 1–12.
- [30] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.
- [31] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Toward realistic performance bounds for implicit cfd codes," in *Proceedings of parallel CFD*, vol. 99. Citeseer, 1999, pp. 233–240.
- [32] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM Journal on Scientific Computing*, vol. 36, pp. C401–C423, 2014. [Online]. Available: <http://epubs.siam.org/doi/abs/10.1137/130930352>
- [33] H. Fehske, G. Wellein, G. Hager, A. Weiße, and A. Bishop, "Quantum lattice dynamical effects on single-particle excitations in one-dimensional mott and peierls insulators," *Physical Review B*, vol. 69, no. 16, p. 165115, 2004.
- [34] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein, "GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems," *International Journal of Parallel Programming*, vol. 45, pp. 1046–1072, 2017.
- [35] M. Sitte, A. Rosch, E. Altman, and L. Fritz, "Topological insulators in magnetic fields: Quantum Hall effect and edge channels with a nonquantized theta term," *Physical review letters*, vol. 108, no. 12, p. 126807, 2012.



Ayesha Afzal holds a Master's degree in Computational Engineering from Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, followed by a Bachelor's degree in Electrical Engineering from the University of Engineering and Technology, Lahore, Pakistan. She is working toward the Ph.D. degree at the professorship for High Performance Computing at Erlangen National High Performance Computing Center (NHR@FAU), Germany. Her PhD research lies at the intersection of analytic performance models, performance tools and parallel simulation frameworks, with a focus on first-principles performance modeling of distributed-memory parallel programs in high-performance computing.



Georg Hager holds a doctorate (Ph.D.) and a Habilitation degree in Computational Physics from the University of Greifswald, Germany. He leads the Training & Support Division at Erlangen National High Performance Computing Center (NHR@FAU) and is an associate lecturer at the Institute of Physics at the University of Greifswald. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes on chip and system levels, and the modeling of out-of-lockstep behavior in large-scale parallel codes.



Gerhard Wellein received the Diploma (M.Sc.) degree and a doctorate (Ph.D.) degree in Physics from the University of Bayreuth, Germany. He is a Professor at the Department of Computer Science at Friedrich-Alexander-Universität Erlangen-Nürnberg and heads the Erlangen National Center for High-Performance Computing (NHR@FAU). His research interests focus on performance modeling and performance engineering, architecture-specific code optimization, and hardware-efficient building blocks for sparse linear algebra and stencil solvers.