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—Bottleneck, desynchronization, parallel distributed computing, performance modeling, performance optimization, scalability, synchronization

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 multi-faceted. 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 article 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 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 $N_{memBW} = 20$ 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.

Table 1 shows an overview of a microbenchmark and the two proxy applications under investigation and which part of the parameter space we cover 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 have been addressed in [5].

**Terminology**

Computational wavefronts and spontaneous asynchronicity are rather new fields of research, so we provide in the

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

Fig. 1. Timeline view of the propagation of idle waves in the simple setting of regular compute (light blue bars) and communication (pink bars) phases of a parallel program. A long, one-off extra workload (dark blue bar) on the first MPI process and first iteration causes a delay (dark pink bars) at (a) rank 1 and, after another iteration, at rank 2, etc., (b) rank 1, 2, 3, and, after another iteration, at rank 4, 5, 6, etc. Communication is open chain (non-periodic) and bidirectional between (a) process $i$ and $i+1$ (b) process $i$ and $i+2$ after each execution phase. The program was simulated with one process per node to avoid the memory-bandwidth bottleneck for simplicity.

following a list of definitions of terms that have been introduced in our previous research:

- **Idle wave**: An idle wave emerges when a delay on an individual MPI process propagates across other MPI processes in multiple iterations due to inter-process dependencies; see Fig. 1. Irrespective of the origin of the delay (extra work, communication delays, system noise, etc.), it “ripples” through the system in a wave-like manner, hence the name.

- **Propagation speed of idle wave**: The propagation speed $v$ of an idle wave is the distance (number of processes) the delay travels per time unit, i.e., it has a unit of ranks per second. It depends on the communication properties in a time step (communication time $t_{comm}$, protocol quantifier $\sigma$, and quantifier for concurrency and topology $\kappa$) and the execution properties (runtime $t_{comp}$ between successive communications) of the MPI-parallel program [6]:

$$v = \frac{\sigma \times \kappa}{t_{comp} + t_{comm}}$$  \hspace{1cm} (1)

For the transmission of messages, if the messages are shorter than the eager threshold then $\sigma = 1$, otherwise we have $\sigma = 2$. The parameter $\kappa$ depends on the communication topology, i.e., which other processes each process communicates with, and how communications are grouped via calls to MPI_Waitall. In general, $\kappa$ is the sum over all communication distances, however it becomes equal to longest distance only if the outstanding non-blocking MPI requests of all communication partners are grouped in the same MPI_Waitall. In Fig. 1a, a delay takes 8 iterations to propagate through the system because the communication is next-neighbor only ($d = \pm 1$). In Fig. 1b, the delay propagates faster because a process communicates with its next and its next-to-next neighbor ($d = \pm 1, 2$).

- **Decay of idle wave**: Idle wave decay means that the propagating delay get shorter with each inter-process

1. This does not mean that there is no memory bandwidth contention; it just plays no role for desynchronization phenomena.
“hop.” Idle waves decay faster on noisy or hierarchical systems. The details of this behavior have been investigated in [7].

- **Desynchronization**: This describes a situation where MPI processes go “out of lockstep,” i.e., when processes do not arrive at the same point of a execution at the same time. MPI collectives do not always resynchronize processes (see also [6]).

- **Bottleneck evasion**: In the presence of bottlenecks such as memory bandwidth or communication bandwidth, the lock step mode of highly regular MPI applications is unstable, and desynchronization emerges automatically via natural system noise (or can be triggered by provoking idle waves) [5]. This has the effect that the resource bottleneck is used by fewer processes at a time.

- **Computational wavefront**: A computational wavefront is formed by all MPI processes that have arrived at the same iteration in the parallel program. If the program is in lock step (i.e., synchronized), the wavefront is a straight line (it is infinitely steep). In the presence of resource bottlenecks, nontrivial wavefronts form because the lock-step state is unstable (see Fig. 2). The development of a wavefront can be accelerated by the injection (deliberate or by system noise) of idle waves, in which case the wavefront is an “echo” of the disturbances (such as idle waves) after its eventual disappearance.

- **Slope of computational wavefront**: The slope of the computational wavefront is the ratio

\[
\varrho = \frac{\Delta}{t_{\text{slow}} - t_{\text{fast}}},
\]

where \(\Delta\) is the difference in ranks between the slowest and fastest process in a particular iteration, and \(t_{\text{slow}} (t_{\text{fast}})\) is the time at which these processes arrive there. The unit is ranks per second.

The slope in a computational wavefront determines the degree of desynchronization among processes. In complex scenarios the slope may not be constant across the whole range of processes.

### Relevance and Generalization of Desynchronization

Desynchronization is a very common phenomenon in parallel programs. As shown in previous work [5], [7], the presence of a resource bottleneck is decisive for the initial “regular” compute-communicate phase to become unstable over time 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 [8], [9] 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 [6]. 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 presented a validated analytic model for the idle wave propagation speed [6], [7], 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 idle 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
We derive guidelines for performance optimization. We investigate the role of idle waves as possible triggers for desynchronization 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. In contrast to Petrini et al. [13], where frequent synchronization causes a huge loss by forcing a delay on one process to propagate to all other processes within one time step, we regard noise as a means to get out of sync faster by removing synchronizations. Consequently, some noise (i.e., delay) on one process takes (potentially) many iterations to propagate to all other processes. Hoeffer 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 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. [6] 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 (redundant versus 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 [7]. 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 idle waves, like damping and speed, and the interaction with memory-bound characteristics of the application were ignored.

Afzal et al. [5], [6], [6], [7], [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 dynamics applies there. It turned out that on real cluster systems with their complex node-level topology, MPI-parallel memory-bound programs exhibit local, non-static idle 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 depends 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 done on code optimization, focusing on new data structures, efficient algorithms, and parallelization techniques, all of which require explicit programming [30] or implicit library or MPI-aware compiler techniques [31]. 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 article extends our prior work [5], [6], [7] on incorporating disturbances in parallel programs into first-principles analytic performance models. It investigates the influence of such effects on the performance of a microbenchmark and two parallel proxy applications to yield insight into the hardware-software interaction. The investigated parameter space for analysis is given in Table 1. Our paper makes the following relevant contributions:

- We derive guidelines for performance optimization by identifying and investigating desynchronization via bottleneck evasion, which can lead to automatic overlap of communication with computation in MPI-parallel programs.
- We investigate the role of idle waves as possible triggers for desynchronization and identify a strong correlation between idle wave speed and automatic communication-computation overlap. The latter is generally more effective with low idle wave speeds.
Using SpMV and ChebFD proxy applications, we demonstrate that performance improvement by more effective automatic communication overlap is facilitated by large communication overhead, a simple non-split communication concurrency scheme, and a “compact” communication topology, the latter of which can be affected by domain decomposition strategies allowing for slower idle waves.

We elaborate that depending on the underlying problem, forcing overlap by explicit programming may not even be required.

We show that 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.

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.

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

This section categorizes the most relevant characteristics for our experimental setup. System and application parameters and properties are described in Sec. 2.1 and Sec. 2.2, while Sec. 2.3 explains the desynchronization speedup metric.

2.1 HPC Platforms and Architectures

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

1) SuperMUC-NG\(^2\), an Omni-Path cluster with two Intel Xeon “Skylake SP” CPUs per node and 24 cores per CPU (hyper-threading enabled)

2) Meggie\(^3\), 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. It features a graphical user interface (GUI) to observe the status of all MPI processes at any point in a run. ITAC goes to great lengths to ensure that clocks across MPI processes are synchronized so that any observed patterns are not clock skew artifacts.\(^4\)\(^5\) It is impossible to tell whether a process is actually idling or receiving data. We consider the additional time beyond pure communication overhead spent in the MPI library as idleness. Idleness is usually caused by the process waiting for a message and means that a process does neither do any useful work nor does it actually communicate. Process-core affinity was enforced using the _I_MPI_PIN_PROCESSOR_LIST_ environment variable. We ignored the simultaneous multi-threading (SMT) feature and used only physical cores with compact mapping, i.e., the mapping of consecutive cores on a node to consecutive MPI processes. A change in the placement of MPI processes to the hardware will modify the communication structure of the code and thus the speed of idle waves and the slope of computational wavefronts; consequently, the code performance will be influenced \(^6\). 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 multi-threaded optimized release library, which supports asynchronous progress threads. Working sets for memory-bound cases were chosen large enough to not fit into the available cache, i.e., at least

4. \url{https://intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/environment-variable-reference/environment-variables-for-async-progress-control.html}
10× the size of all last-level cache (LLC), which is the L3 caches (non-inclusive victim L3 plus the L2 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 (in Gflops/s) to quantify the speedup caused by desynchronization via bottleneck evasion. Experimentally, we determine the quantity

\[
P_D = \frac{P_{\text{barrier-free}} - P_{\text{barrier}}}{P_{\text{barrier}}} \times 100 \, \% ,
\]

which measures the relative speedup of the program when eliminating an artificial barrier synchronization. It is “artificial” as it is unnecessary in the algorithm and we deliberately insert barriers in each iteration to observe the difference between synchronized and desynchronized runs. Here, \(P_{\text{barrier}}\) and \(P_{\text{barrier-free}}\) are the measured code performance with and without including the extra barriers, respectively. The value \(P_D\) quantifies relative speedup caused by desynchronization via bottleneck evasion, in the sense that a value of 0% means no gain in performance. In usual terms one would say that “the speedup is 1×.” A doubling of the performance (i.e., a “2× speedup”) would result in \(P_D = 100\%\), no matter whether communication or computation dominates; this is how the speedup percentages in the experimental results should be interpreted. To make this comparison useful, the actual barrier overhead (as measured by a microbenchmark) is subtracted from the overall runtime before calculating \(P_{\text{barrier}}\). To be more specific, measurements of isolated barrier duration (e.g., 57 μsec with 1280 processes on 64 Meggie nodes) show negligible variations of barrier time across processes on the time scale we investigate here. In addition, we made sure that compilers did not apply different code transformations when removing a barrier subroutine call. 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 microbenchmark and proxy applications with a focus on idle waves and the automatic overlap of 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:

```c
for (int i=0; i<arrayElements; i++) {
    A[i] = B[i] + s*C[i];
}
```

It allows a straightforward application of the Roofline model [32] to predict the theoretical memory-bound parallel performance limit on a ccNUMA domain as \(P = \frac{b_S}{B_C}\), where \(b_S\) is the domain memory bandwidth and \(B_C = 12\text{byte/flop}\) 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 (10⁶ 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 (\(\text{MPI}_{\text{Isend}}/\text{MPI}_{\text{Irecv}}\)) together with a final \(\text{MPI}_{\text{Waitall}}\) are used in all cases, i.e.:

```c
for (int j = 0; j < 2; j++) {
    MPI_Isend(., srcRank[j], ., &req[j+2]);
    MPI_Irecv(., dstRank[j], ., &req[1+j*2]);
}
MPI_Waitall(4, &req[0], &stt[0]);
```

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 16384 B to and from \(i \pm 1\) after each STREAM phase).

#### 3.1.1 Shape of Computational Wavefronts

Fig. 2 shows MPI traces of the MPI-augmented STREAM Triad on 40 processes (two nodes of Meggie) with different communication topologies. It shows the correlation between the communication structure (and thus the idle wave speed) and the slope of a computational wavefront: Systems with faster idle waves ((b) — (d)) exhibit steep computational wavefronts; the peculiar shape observed in (c) is governed by the two inherent communication distances. Details are discussed in [6]. 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 point, as was already pointed out in [5]. Blue bars with whiskers quantify the wall-time difference between the fastest and the slowest process on each of three NUMA domains to better visualize the constant slope of computational wavefront. In (a), on the fourth (top) domain, the slope is different since the full wavefront was not fully developed yet. In (b) one can observe the difference to next-pair communication \((d = \pm (1, 2))\): The developed desynchronized state (the computational wavefront) with \(d = \pm (1, 2)\) is about \(3\times\) steeper than the one with \(d = \pm 1\) in (a), i.e., it has a smaller amplitude. This is illustrated by the shorter blue bars in (b) compared to (a). This correlates with the higher idle wave speed for longer-distance communication scenarios [6]. This means that, for the change in communication structure from (a) to (b), the parameter \(k\) in (1) changes from \(k = 1\) to \(k = 1 + 2 = 3\) and leads to a three times higher speed of idle waves; see Fig. 1. 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

7. A saturation point is the minimum number of processes required to achieve the maximum memory bandwidth on the ccNUMA domain.
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 \( d = \pm (1, \ldots, 12) \) causes high-speed idle waves [6], which leads to steep computational wavefronts. Starting at a communication distance of at least \( d = \pm (1, \ldots, 8) \), 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 extra workloads. 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 extra workloads can. In the first and third graph of Fig. 3 we show the results of an experiment on the Meggie system, where a computational wavefront (marked with blue symbols at the 80th, the 100th, and the 200th iteration, respectively) is disturbed by an injection at rank 15, which runs on a ccNUMA domain different from the one where the lagger initially resides (ranks 0–9). To spark an idle wave, a series of floating-point divide operations are performed by rank 15 at time step 80, which can be observed as a “break” in the wave at step 80 and also via the blue bars in the second and forth graph of Fig. 3. For one-off extra workloads, 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 \((A\vec{x})\) 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 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 (\(colIndA[\] \)), 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/flop\(^8\) [33], [34].

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. The “local” and “remote” kernels describe the parts of one SpMV for which the right-hand side vector entries are either local to the process or need to be obtained from other processes, respectively. Since the sparse matrix and the vectors are distributed to processes in consecutive row blocks, the RHS entries corresponding to the quadratic block of a matrix that is centered on the diagonal are already present on the process. Matrix entries outside of this block (to the left and the right) require remote RHS vector entries. 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 [33].

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. 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.

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 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) [35]. 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 [34]). 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 versus 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. 4 and 5). 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 (ScaMaC) library.

### 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

---

**Listing 1: CRS based SPMVM kernel**

```c
1: double *valA[nzA], *b[nor], *x[nor];
2: int *colidxA[nzA], rowPtrA[nor + 1], tmp;
3: for (row = 0; row < nor; row++) {
4:     tmp = 0;
5:     for (idx = rowPtrA[row]; idx < rowPtrA[row+1] - 1; idx++) {
6:         tmp += valA[idx] * x[colidxA[idx]];
7:     }
8:     b[row] += tmp;
9: }
```

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

**Listing 2: SPLIT-WAIT mode**

```c
1: while (iter <= maxiter) {
2:     MPI_Irecv();
3:     MPI_Irecv();
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
```

**Listing 3: NON-SPLIT mode**

```c
1: while (iter <= maxiter) {
2:     MPI_Irecv();
3:     MPI_Irecv();
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
```

---

**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

---

**Fig. 4.** (a-b) The communication topology of the large matrices using periodic boundary conditions in the wider pe order (left/right bandwidth of 41385344) and slimmer 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.

---

9. The ScaMaC 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.
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. Since the measurements are not normally distributed, the minimum, maximum, and median values are reported instead of average and standard deviation [36]. 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 (pink in Table 4), 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., Fig. 5. (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).

### Table 4

| Program properties | 96-pe | 144-pe | 240-pe | 480-pe | 720-pe | 960-pe | 1296-pe |
|--------------------|-------|--------|--------|--------|--------|--------|---------|
| Exec min [ms]      | 39.58 | 18.64  | 9.67   | 4.28   | 3.13   | 2.44   | 1.82    |
| Exec max [ms]      | 64.92 | 45.73  | 27.55  | 13.49  | 10.53  | 8.25   | 6.24    |
| Exec median [ms]   | 53.55 | 35.74  | 18.51  | 9.05   | 6.53   | 5.06   | 3.85    |
| Comm min [ms]      | 20.7  | 15.38  | 6.72   | 5.62   | 3.57   | 4.03   | 2.51    |
| Comm max [ms]      | 38.73 | 29.49  | 21.87  | 13.48  | 9.83   | 7.39   | 5.52    |
| Comm median [ms]   | 29.48 | 24.56  | 16.17  | 14.75  | 11.28  | 9.78   | 7.99    |
| Mean P2P msg size [kB] | 2700  | 1460   | 957    | 480    | 302    | 213    | 153     |
| CER median         | 0.55  | 0.69   | 0.87   | 1.63   | 1.73   | 1.93   | 2.08    |

### Table 5

Key Specifications of Symmetric Sparse Matrices

| Matrix-order | Bandwidth | \( n_{\text{electrons}} = n_{\text{sites}} = n_{\text{phonons}} \) | \( n_r = n_c \) | \( n_{\text{nz}} \) | Size [GB] |
|--------------|-----------|-----------------|------------|----------------|---------|
| HHQ-large-pe | high      | 3 – 8 – 10      | 80988928   | 889816368     | 13      | 10.9    |
| HHQ-large-ep | low       | 3 – 8 – 10      | 80988928   | 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    |

\( ^{\frac{1}{2}} \) The described quantum system comprises \( n_{\text{electrons}} \) electrons on \( n_{\text{sites}} \) lattice sites coupled to \( n_{\text{phonons}} \) phonons.

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

\( ^{1} \) The inner loop length of the CRS SpMVM kernel \( n_{\text{nz}}(n = \frac{1}{2}) \) is the average number of non-zero entries in each row of the sparse matrix.

\( ^{*} \) Data set size is estimated by \( 12n_{\text{nz}} + 4n_r \) (eight byte per matrix entry and four byte for column indices).
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. Figs. 4c and 4d 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 Eq. (3)) caused by bottleneck evasion via desynchronization is shown in Fig. 4e. 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 lockstep 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 speed [5, 6]. 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. 4e. 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 black markers in Fig. 4. 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. 5 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. 5c), 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. 5d), 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 (light blue in Table 4) 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], [9]. 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-source\textsuperscript{10} implementation of ChebFD is built with the GHOST [37] 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 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-I vector operations. Non-blocking communication is performed via MPI\textunderscore Isend/MPI\textunderscore Irecv/MPI\textunderscore 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 $U^T$ 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\textunderscore 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 $n_b/n_s$ this scheme enables explicit effective overlap of computation on the current subblock (local\textunderscore kernel) with the communication needed for the next subblock. Details can be found in [9].

The polynomial filter degree (we use $n_p = 500$ here, which is a relevant value for practical applications) applies independently to all search vectors $n_s$. In the Algorithm 1, $n_{iter}$ 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 1) facilitates cache reuse. Moreover, the global reduction needed for computation of the polynomial filter coefficients $\tilde{\eta}_p$ and $\tilde{\mu}_p$ can be postponed until after the iteration loop, which eliminates all (possibly) synchronizing collectives; the communication topology is

| Matrix       | Traits\textsuperscript{\dagger} | Data-type | $n_s = n_b$ | $n_{tot}$ | $n_{mp}$ | Size [GB]\textsuperscript{\dagger}\textsuperscript{\ddagger} |
|--------------|----------------------------------|-----------|-------------|-----------|---------|-----------------|
| Topi-ENH     | $m_z - m_z, m_z - 128 - 128 - 64$ | complex   | 268435456   | 3487563776 | 13      | 70.8\textsuperscript{\dagger} |
| 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            |

\textsuperscript{\dagger} Traits for the Topi-ENH matrix represent the $m_z - m_z, m_z - 128 - 128 - 64$ while for the Spin matrices mark $n_{mp}$-disorder-seed.

\textsuperscript{\ddagger} Eight (sixteen) byte for double (complex double) precision numbers of matrix entries, and four-byte indexing for 32-bit integers are considered.

\textsuperscript{\ddagger} Sparse matrices comprising periodic boundary conditions with upper count are to the half of the lower.

\textsuperscript{\dagger} Available as part of the GHOST package at https://bitbucket.org/essex/ghost

Fig. 6. 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.
thus entirely determined by the matrix structure. The sparse matrices and the code balance of the algorithm (optimistically 11) (260/n_oh + 80)/146 byte/flop for double complex data and (48/n_oh + 40)/19 byte/flop for double precision real data) were thoroughly investigated in [2].

### 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-ENH) [38] and strong scaling for a spin system (Spin). For TOPI-ENH (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 × n_y × n_z = 128^3 × 64 and a global size of n_x m_x n_y m_y n_z m_z, where the n_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 × 128 × 128 × 64 × 13 = 273 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. 6 reveals interesting differences between the two matrices: Although increasing the block size from n_oh = 2 to n_oh = 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_oh = 2 and thus shows good scaling in this case (optimistic upper bandwidth limit at 14.8 Gflop/s), the TOPI-ENH matrix shows strong bandwidth saturation and achieves 90% of the optimistic maximum of 34.8 Gflop/s already at eight cores. At n_oh = 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], [8]. While this can be clearly observed in case of TOPI-ENH, 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 library.12 Table 6 illustrates the key specifications of both types of matrices.

### 3.3.3 Decomposition, Communication Schemes and Block Vector Sizes

#### TOPI-ENH Matrices.

Fig. 8 shows the weak scaling performance of ChebFD with the TOPI-ENH 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 W and 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 (see Table 7), as do the communication data paths used (intra- versus inter-node). Hence, we expect a significant dependence of the performance on the decomposition. Unlike in SpMV, 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. This is in accordance with our idle wave speed model (1): M3 is expected to have the slowest idle wave because of the direct-neighbor communication (see Fig. 1), while M7, M8, M4 are the ones which have large-distance communications. With explicit barriers, on the other hand, M7 is the worst configuration and M8 is

### Table 7

| n_oh | m_x m_y m_z | Communication distances (n_oh = 64) | Message sizes [kB] (n_oh = 64) |
|------|-------------|-------------------------------------|-------------------------------|
| M3   | 1-64m_oh   | ±1, ±20, ±19                        | 1050, 105, 0.128             |
| M2   | 1-64m_oh   | ±1, ±20, ±19                        | 1050, 105, 0.128             |
| M3   | 1-64m_oh   | ±1                               | 1050                          |
| M4   | 1-64m_oh   | ±1, ±20, ±19, ±159, ±19           | 1050, 105, 48, 8, 0.128       |
| M5   | 1-64m_oh   | ±1, ±20, ±19                        | 1050, 105, 0.128             |
| M6   | 1-64m_oh   | ±1, ±20, ±19, ±140, ±19, ±159, ±19 | 1050, 105, 48, 8, 0.128       |
| M7   | 2-64m_oh   | ±1, ±20, ±50, ±80, ±561, ±79, ±19  | 1050, 105, 48, 8, 0.128       |
| M8   | 2-64m_oh   | ±1, ±20, ±80, ±79, ±81             | 1050, 105, 48, 8, 0.128       |

11. “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.

12. The ESSEX_PHYSICS library is a open-source software of ESSEX project, available for download at https://bitbucket.org/essex/physics/src/master.
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. This rising idle wave speed order correlates well with the expected connection between idle wave speed and computational wavefront slope. This is visible in Fig. 7, which shows traces of 64-node runs on Meggie with the M3 and M4 configuration, respectively: With M3, although the time spent in MPI is larger per process on average (red color), performance is better because desynchronized wavefronts can develop across the whole system allowing bottleneck evasion and automatic communication overlap.

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 $\{\text{NON-SPLIT, PIPELINE, SPLIT-WAIT}\}$ 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, 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. Fig. 9 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 $\{0.05, 0.35, 0.5\}$...
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 \(T_{exec} \text{[ms]}, T_{comm} \text{[ms]}, \text{CER} = \{0.6, 0.41, 0.68\}, \{2.38, 1.86, 0.78\}, \{10.2, 7, 0.69\}\) for Spin26, Spin28 and Spin30, respectively. Similarly, the range of P2P message sizes is \(\{V_{min} \text{[kB]}, V_{max} \text{[kB]} \text{[red]}\} = \{16, 130\} \text{(Spin26)}, \{64, 501.5\} \text{(Spin28)} \) and \{32, 1939\} \text{(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. 9. 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 Summary and Future Work

In this section, using MPI-parallel synthetic benchmarks and application programs we investigated the consequences of desynchronization via bottlenecks in bulk-synchronous parallel code.

#### 4.1 Summary

Using a memory-bound microbenchmark we showed that there is a strong positive correlation between idle wave speed and the slope of a computational wavefront, indicating that automatic communication-computation overlap can be more effective in settings with low idle wave speeds. Using one-off extra workloads we also showed that a stable computational wavefront 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 Chebychev 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 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.

#### 4.2 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.

ACKNOWLEDGEMENT

We are indebted to LRZ Garching for granting CPU hours on SuperMUC-NG. We wish to thank Andreas Alvermann for his ScaMac library. For the Meggie system, the authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). The hardware is funded by the German Research Foundation (DFG).

REFERENCES

[1] J. D. McCalpin et al., “Memory bandwidth and machine balance in current high performance computers,” *IEEE Comput. Soc. Tech. Committee Comput. Architecture Newslett., vol. 2, no. 19-25, 1995.
[2] A. Pieper et al., “High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations,” *J. Comput. Phys.*, 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 Comput.*, vol. 40, no. 7, pp. 224–238, 2014.
[5] A. Afzal, G. Hager, and G. Wellein, “Desynchronization and workload completion 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. Berlin, Germany: Springer, 2020, pp. 391–411.
[6] 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. Berlin, Germany: Springer, 2021, pp. 351–371.
[7] A. Afzal, G. Hager, and G. Wellein, “Propagation and decay of injected one-off delays on clusters: A case study,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2019, pp. 1–10.
[8] M. Kreutzer, A. Pieper, G. Hager, G. Wellein, A. Alvermann, and F. Hefseke, “Performance engineering of the Kernel Polynomial Method on large-scale CPU-GPU systems,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 417–426.
[9] M. Kreutzer et al., “Chebyshev filter diagonalization on modern manycore processors and GPGPUs,” in *High Performance Computing, Bioinformatics, and Health Informatics*, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis Eds., Berlin, Germany: Springer, 2018, pp. 329–349.
[10] A. Afzal, G. Hager, and G. Wellein, “Analytic performance model for parallel overlapping memory-bound kernels,” *Concurrency Comput. Pract. Experience*, vol. 34, no. 10, Jan. 2022, Art. no. e6816.
[11] X. Zhao, M. Jahre, and L. Eeckhout, “HSM: A hybrid slowdown model for multitasking GPGPUs,” in *Proc. 25th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1371–1383.
[12] A. Afzal, G. Hager, and G. Wellein, “Delay flow mechanisms on clusters,” 2019. [Online]. Available: https://hpc.fau.de/files/2019/09/EuroMPI2019_AHW-Poster.pdf
[13] F. Petkini, D. J. Kerbyson, and S. Pakin, “The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q,” in *Proc. IEEE/ACM Conf. Supercomputing*, 2003, pp. 55–55.
[14] T. Jones et al., “Improving the scalability of parallel jobs by adding parallel awareness to the operating system,” in *Proc. IEEE/ACM Conf. Supercomputing*, 2003, Art. no. 10.
[15] P. Terry, A. Shan, and P. Hutton, “Improving application performance on HPC systems with process synchronisation,” *Linux J.*, vol. 2004, no. 127, pp. 68–71, 2004.
[16] R. Giosioa, F. Petrinia, K. Davis, and F. Lebaillif-Delamare, “Analysis of system overhead on parallel computers,” in *Proc. IEEE 4th Symp. Signal Process. Inf. Technol.*, 2004, pp. 387–390.
[17] D. Tsafiranis, V. Eftisioy, D. G. Fedelston, and G. Y. Yirick, “System noise, OS clock ticks, and fine-grained parallel applications,” in *Proc. 19th Annu. Int. Conf. Supercomputing*, 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 *Proc. IEEE Int. Conf. Cluster Comput.*, 2006, pp. 11–12.
[19] R. Gioiosa, F. Petrinia, F. A. Novotny, and G. Wellein, “Desynchronization and Characterizing application sensitivity to OS interference using kernel-level noise injection,” in *Proc. IEEE/ACM Conf. Supercomputing*, 2008, Art. no. 19.
[20] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, “A quantitative analysis of OS noise,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2008, Art. no. 19.
[21] H. Weisbach, B. Gerofi, B. Kocikoloski, H. Härting, 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., Berlin, Germany: Springer, 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 *Proc. IEEE/ACM Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2010, pp. 10–11.
[23] S. Markidis, J. Vencels, I. B. Peng, D. Akhmetova, E. Laure, and P. Henri, “Idle waves in high-performance computing,” *Phys. Rev. E*, vol. 91, no. 1, 2015, Art. no. 013306.
[24] M. Gamell et al., “Local recovery and failure masking for stencil-based applications at extreme scales,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.
[25] D. Böhme, M. Geimer, L. Arnold, F. Voigtländer, 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,” *2004, arXiv:cs/0402034*.
[27] A. Afzal, G. Hager, and G. Wellein, “Physical oscillator model for parallel distributed computing,” 2021, poster at ISC High Performance 2021.
[28] D. An Mey et al., “Score-P: A unified performance measurement system for petascale applications,” in *Competence in High Performance Computing*, Berlin, Germany: Springer, 2011, pp. 65–97.
[29] T.-T. Pham, M. Pester, and P. Couves, “Recurrent neural network for classifying of HPC applications,” in *Proc. Spring Simul. Conf.*, 2019, pp. 1–12.
[30] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, Boca Raton, FL, USA: CRC Press, 2010.
[31] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, “MPI-aware compiler optimizations for improving communication-computation overlap,” in *Proc. 23rd Int. Conf. Supercomputing*, 2009, pp. 316–325.
[32] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[33] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, “Influences of the root causes of wait states in large-scale parallel applications,” *ACM Trans. Parallel Comput.*, vol. 52, no. 4, pp. 65–76, 2009.
[34] M. Kreutzer, G. Hager, G. Wellein, F. Hefseke, 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 J. Sci. Comput.*, vol. 36, pp. C401–C423, 2014.
[35] H. Fehske, G. Wellein, G. Hager, A. Weiße, and A. Bishop, “A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units,” *SIAM J. Sci. Comput.*, vol. 36, pp. C401–C423, 2014.
[36] T. Hoefler and R. Belli, “A new algorithm for parallel computing systems: Twelve ways to tell the masses when reporting performance results,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.
Ayesha Afzal received the bachelor’s degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan and the master’s degree in computational engineering from Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany. She is currently working toward the PhD degree with the professorship for High Performance Computing with Erlangen National High Performance Computing Center (NHR@FAU), Germany. Her research interests lies with 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 received the doctorate (PhD) and the Habilitation degrees in computational physics from the University of Greifswald, Germany. He leads the Training & Support Division with Erlangen National High Performance Computing Center (NHR@FAU) and is an associate lecturer with the Institute of Physics with the University of Greifswald. His 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 (MSc) degree and a doctorate (PhD) degree in Physics from the University of Bayreuth, Germany. He is a professor with the Department of Computer Science with 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.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.