An analytic performance model for overlapping execution of memory-bound loop kernels on multicore CPUs

Ayesha Afzal, Georg Hager, Gerhard Wellein
Department of High Performance Computing
Erlangen Regional Computing Center
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
{ayesha.afzal,georg.hager,gerhard.wellein}@fau.de

Abstract—Complex applications running on multicore processors show a rich performance phenomenology. The growing number of cores per ccNUMA domain complicates performance analysis of memory-bound code since system noise, load imbalance, or task-based programming models can lead to thread desynchronization. Hence, the simplifying assumption that all cores execute the same loop can not be upheld. Motivated by observations on plain and modified versions of the HPCG benchmark, we construct a performance model of execution of memory-bound loop kernels. It can predict the memory bandwidth share per kernel on a memory contention domain depending on the number of active cores and which other workload the kernel is paired with. The only code features required are the single-thread cache line access frequency per kernel, which is directly related to the single-thread memory bandwidth, and its saturated bandwidth. It can either be measured directly or predicted using the Execution-Cache-Memory (ECM) performance model. The computational intensity of the kernels and the detailed structure of the code is of no significance. We validate our model on Intel Broadwell, Intel Cascade Lake, and AMD Rome processors pairing various streaming and stencil kernels. The error in predicting the bandwidth share per kernel is less than 8%.

I. INTRODUCTION

With the number of cores and the peak performance of modern multicore chips still growing, the memory bandwidth bottleneck is becoming more severe. Many algorithms in computational science are based on building blocks that show memory-bound behavior, i.e., whose performance does not scale but saturate with respect to the number of active cores when running on a memory contention domain (usually a ccNUMA domain). Performance saturation due to bandwidth saturation should be regarded as a sign that a code is fast enough to address an architectural bottleneck, so it is not a bad thing in general. It also opens a clear optimization path via the reduction of data transfers.

Beyond the simple saturation pattern, however, memory-bound code exhibits other, more interesting phenomenology. As was shown in [2], bulk-synchronous barrier-free MPI programs can show desynchronization on a contention domain, i.e., processes move away from the initial lockstep state into a state where execution of loops kernels overlaps with communication or idleness, leading to automatic communication hiding. This can be provoked by a deliberate injection of delays, but it can also occur automatically by natural system noise and small load imbalances.

Although it is often observed in programs that have a significant communication overhead, desynchronization is not limited to this scenario. In fact, barrier-free code (and also modern, task-based programming models) allows for concurrent execution of code with very different characteristics on a contention domain. In this paper we investigate a specific scenario: concurrent execution of two different loop kernels on $n$ cores each. We construct a performance model that can describe accurately the memory bandwidth share that each kernel attains. The model is thus able to predict whether desynchronization of back-to-back parallel kernels and the ensuing overlap will speed up or slow down the execution of threads compared to a purely homogeneous execution. It can also predict if the mutual overlap will amplify or reduce the desynchronization effect.

This paper is organized as follows. The rest of this section covers related work and a motivational example (HPCG). In Sect. II we describe the experimental setup and methodology, and Sect. III defines code and performance metrics. The performance model is developed in IV and validated in Sect. V, where we also discuss the connection to desynchronization and close the loop to the initial HPCG example. We give a summary and an outlook to future work in Sect. VI.

A. Motivation and related work

This investigation is based on prior work by Afzal et al. [1, 3, 2], who studied idle wave propagation and computational wave formation in parallel programs on a phenomenological level. An analytic model of overlapping computational kernels and communication time on a contention domain was lacking, however. Alappat et al. [4] observed the consequences of desynchronization in the context of the well-known HPCG benchmark.\footnote{https://www.hpcg-benchmark.org/} While validating their Roofline model for the MPI-parallel HPCG, they observed that the DDOT2 (dot product $s += a[i]*b[i]$) kernels were in fact faster than what the local memory bandwidth would allow for. The authors attributed this to process desynchronization during the preceding sparse matrix-vector multiplication (SpMV) kernel.

Abstract—Complex applications running on multicore processors show a rich performance phenomenology. The growing number of cores per ccNUMA domain complicates performance analysis of memory-bound code since system noise, load imbalance, or task-based programming models can lead to thread desynchronization. Hence, the simplifying assumption that all cores execute the same loop can not be upheld. Motivated by observations on plain and modified versions of the HPCG benchmark, we construct a performance model of execution of memory-bound loop kernels. It can predict the memory bandwidth share per kernel on a memory contention domain depending on the number of active cores and which other workload the kernel is paired with. The only code features required are the single-thread cache line access frequency per kernel, which is directly related to the single-thread memory bandwidth, and its saturated bandwidth. It can either be measured directly or predicted using the Execution-Cache-Memory (ECM) performance model. The computational intensity of the kernels and the detailed structure of the code is of no significance. We validate our model on Intel Broadwell, Intel Cascade Lake, and AMD Rome processors pairing various streaming and stencil kernels. The error in predicting the bandwidth share per kernel is less than 8%.

I. INTRODUCTION

With the number of cores and the peak performance of modern multicore chips still growing, the memory bandwidth bottleneck is becoming more severe. Many algorithms in computational science are based on building blocks that show memory-bound behavior, i.e., whose performance does not scale but saturate with respect to the number of active cores when running on a memory contention domain (usually a ccNUMA domain). Performance saturation due to bandwidth saturation should be regarded as a sign that a code is fast enough to address an architectural bottleneck, so it is not a bad thing in general. It also opens a clear optimization path via the reduction of data transfers.

Beyond the simple saturation pattern, however, memory-bound code exhibits other, more interesting phenomenology. As was shown in [2], bulk-synchronous barrier-free MPI programs can show desynchronization on a contention domain, i.e., processes move away from the initial lockstep state into a state where execution of loops kernels overlaps with communication or idleness, leading to automatic communication hiding. This can be provoked by a deliberate injection of delays, but it can also occur automatically by natural system noise and small load imbalances.

Although it is often observed in programs that have a significant communication overhead, desynchronization is not limited to this scenario. In fact, barrier-free code (and also modern, task-based programming models) allows for concurrent execution of code with very different characteristics on a contention domain. In this paper we investigate a specific scenario: concurrent execution of two different loop kernels on $n$ cores each. We construct a performance model that can describe accurately the memory bandwidth share that each kernel attains. The model is thus able to predict whether desynchronization of back-to-back parallel kernels and the ensuing overlap will speed up or slow down the execution of threads compared to a purely homogeneous execution. It can also predict if the mutual overlap will amplify or reduce the desynchronization effect.

This paper is organized as follows. The rest of this section covers related work and a motivational example (HPCG). In Sect. II we describe the experimental setup and methodology, and Sect. III defines code and performance metrics. The performance model is developed in IV and validated in Sect. V, where we also discuss the connection to desynchronization and close the loop to the initial HPCG example. We give a summary and an outlook to future work in Sect. VI.

A. Motivation and related work

This investigation is based on prior work by Afzal et al. [1, 3, 2], who studied idle wave propagation and computational wave formation in parallel programs on a phenomenological level. An analytic model of overlapping computational kernels and communication time on a contention domain was lacking, however. Alappat et al. [4] observed the consequences of desynchronization in the context of the well-known HPCG benchmark.\footnote{https://www.hpcg-benchmark.org/} While validating their Roofline model for the MPI-parallel HPCG, they observed that the DDOT2 (dot product $s += a[i]*b[i]$) kernels were in fact faster than what the local memory bandwidth would allow for. The authors attributed this to process desynchronization during the preceding sparse matrix-vector multiplication (SpMV) kernel.
and assumed that MPI processes that started the DDOT2 kernel early could benefit from immediate cache reuse, which was backed by a measured computational intensity that was higher than expected. However, it was unclear whether cache reuse was the only reason for the observed elevated DDOT2 performance.

The HPCG algorithm comprises the following kernels: six of BLAS-1 type (two DDOT1, one DDOT2 [s+a[i]·a[i:j]] and three DAXPY [a[i:j]=s·b[i:j]] and one SpMV. Additionally, the multigrid preconditioner compromises five kernels: restriction, prolongation, SpMV, and two symmetric Gauss-Seidel routines (SymGS) as pre- and post-smoothers, each with a forward and a backward sweep, for coarsening and refinement.

HPCG has two types of MPI communication: a global collective operation (MPI_Allreduce) after each vector dot product and nonblocking point-to-point communication in SpMV and SymGS.

Figure 1(a) (top) shows a snippet from a timeline diagram of the HPCG benchmark (problem size 160^3 per process) on the nine cores of one ccNUMA domain of an Intel Broadwell processor (see Sect. II for hardware details and software setup). The colors encode the SymGS kernel (orange), the DDOT2 kernel (pink), and the MPI_Allreduce call (green). Note that this is only a part of a single iteration; all effects described here occur in every iteration of the algorithm.

The runtime of the SymGS kernel is about 20 times longer than that of DDOT2 here, so the clearly visible desynchronization effect has a minor influence on the SymGS performance. However, the execution of DDOT2 is strongly out of sync across processes. “Early” ranks still overlap with SymGS running on other cores, while “late” ranks overlap with waiting time (i.e., idleness) in MPI_Allreduce. The former must compete for memory bandwidth with SymGS and thus take longer to execute, while the latter can utilize more bandwidth per core, making them execute faster. This can be seen better in the lower part of Fig. 1(a), where we have sorted the MPI ranks according to the starting time of the DDOT2 kernel from early (bottom) to late (top). The late starters clearly take less time than the early starters, although the latter are those with cache reuse potential. In Fig. 1(b), the experiment was repeated on a 20-core Intel Cascade Lake processor. The effect is less pronounced here since the DDOT2 runtime is now significantly longer than the desynchronization time scale. There is hence less opportunity for DDOT2 to overlap with MPI_Allreduce.

Figure 1(c) shows the runtime of the DDOT2 kernel per rank in both experiments, using the same sorting as in (a) and (b). The runtimes are monotonically decreasing in both cases, showing clearly that late starters have better performance.

In summary, the desynchronization of the SymGS kernel, in itself a negligible effect, leads to some cores executing the DDOT2 kernel faster due to overlapping with idleness in the MPI_Allreduce function. This is the real reason for the “faster-than-light” performance observed in [4].

In the broader context of desynchronized execution of barrier-free bulk-synchronous code, the general question arises how

\[ a_1. \text{Identical kernels: } \text{Exec}_B := \text{Exec}_A \]
\[ a_2. \text{Different kernels: } \text{Exec}_A \neq \text{Exec}_B \]

Fig. 2: Characterization of concurrency scenarios for bandwidth-limited computational kernels in parallel programs. “P0” and “P1” denote multi-threaded processes or groups of MPI processes within a contention domain. Each thread or process in a group executes the same code on different data. (a1, a2) Overlapping memory-bound loop kernels of different (or same) characteristics, (b1, b2) overlapping loop kernel execution with latency- or bandwidth-bound communication, and (c) overlapping loop kernel execution with idleness or code that addresses nonshared or scalable resources.
We also executed a single iteration of the benchmark only.

This allows for desynchronized states to survive for a long time, the spread in endpoints of DDOT2 execution overlaps with a subsequent SpMV and its significant MPIWait time, the spread in endpoints of DDOT2 execution is smaller than the spread in their starting points. This is plausible because, as described earlier, early starters of DDOT2 take longer while late starters are faster. The effect can be quantified by a negative skewness parameter of the accumulated DDOT2 time distribution of \(-0.27\) ms. The second DDOT2, however, is sandwiched between SpMV and DAXPY, so any overlap with idleness is ruled out. Instead, the interaction with the low-intensity DAXPY code seems to boost the desynchronization effect, i.e., the tail end of the accumulated DDOT2 time distribution is longer than the front end (skewness 0.42 ms). The last DDOT, which is actually a vector norm computation (DDOT1), shows a similar characteristic with an even larger positive skewness of 1.0 ms.

The skewness parameterizes the asymmetry in the distribution of a random variable, but the timeline data from the HPCG is not normally distributed even if the skewness is zero. However, we can use the skewness here to identify an important property of barrier-free parallel loop kernel execution: Negative skewness indicates resynchronization, while positive skewness indicates desynchronization. Which of the two occurs is obviously a function of which other workloads the kernel is embedded in. Note that the described effects in HPCG, even in our synchronization-free variant, are small in absolute numbers since the HPCG runtime is dominated by the multigrid and SpMV kernels. However, our observations open interesting questions towards the onset and mitigation of desynchronization effects in barrier-free memory-bound MPI programs. The goal of this paper is to establish a performance model for loop kernels of different characteristics running concurrently on a contention domain that can describe this scenario with sufficient accuracy. As an additional benefit, such a model can also predict the dynamics of task-based programming models on memory-bound code.

**B. Contributions**

This paper makes the following new contributions. (i) We identify the influence factors that govern the bandwidth share for individual threads in a “hybrid execution” setting, specifically two (groups of) threads running loops with different characteristics on a memory bandwidth contention domain: the “memory request fraction” to the memory interface and the saturated memory bandwidth of the different kernels. (ii) We employ ideas from the ECM performance model to predict the bandwidth share of different kernels on a contention domain. (iii) We validate the model for a set of 30 kernel pairings on four x86 architectures. Despite significant differences in hardware...
TABLE I: Key hardware and software specifications of systems.

| Systems             | BDW-1                              | BDW-2                              | CLX                                | Rome                               |
|---------------------|------------------------------------|------------------------------------|------------------------------------|------------------------------------|
| Processor           | Broadwell EP                        | Broadwell EP                        | Cascade Lake SP                    | Zen                                |
| Model               | Intel Xeon E5-2630 v4               | Intel Xeon E5-2607 v4               | Intel Xeon Gold 6248               | ARM Epyc 7451                      |
| Base clock speed    | 2.2 GHz                             | 2.3 GHz                             | 2.5 GHz                            | 2.35 GHz                           |
| Cores per NUMA domain (SMT) | 10 (20)                             | 18 (36)                             | 20 (40)                            | 8 (16)                             |
| Private L1 size     | 32 (×10) KiB                        | 32 (×18) KiB                        | 32 (×20) KiB                       | 32 (×8) KiB                        |
| Private L2 size     | 256 (×10) KiB                       | 256 (×18) KiB                       | 1048 (×20) KiB                     | 512 (×8) KiB                       |
| Shared LLC size     | 25 MiB (10×2.5 MiB)                 | 45 MiB (18×2.5 MiB)                 | 27.5 MiB (20×1.375 MiB)            | 8 MiB (per 4 cores)                |
| Memory per node (type) | 64 GB (DDR4)                        | 384 GB (DDR4)                       | 128 GB (DDR4)                      | 128 GB (DDR4)                      |
| LD/ST throughput (SIMD) | 2/1 (AVX2/FMA3)                    | 2/1 (AVX2/FMA3)                    | 2/1 (AVX-512/FMA3)                | 2/1 (AVX2/FMA3)                    |
| L1 := L2 bandwidth  | 64 B/cy                             | 64 B/cy                             | 64 B/cy                            | 64 B/cy                            |
| L2 := LLC bandwidth | 32 B/cy                             | 32 B/cy                             | 16×16 B/cy                         | 32×32 B/cy                         |
| LLC organization    | Inclusive                           | Inclusive                           | Exclusive                          | Exclusive                          |
| Victim caches       | No                                  | No                                  | LLC                                | LLC                                |
| El. transfers       | Non-overlapping                     | Non-overlapping                     | Non-overlapping                    | Overlapping                        |
| Theor. MEM bandwidth| 68.3 GB/s                           | 76.8 GB/s                           | 140.8 GB/s                         | 170.6 GB/s                         |

II. HARDWARE AND SOFTWARE SETUP, EXPERIMENTAL METHODOLOGY

To ensure the broad applicability of our results, we conducted all experiments on one ccNUMA domain of four different processors from Intel and AMD (see Table I for details on the hardware and software setup). Cluster-on-Die (CoD) and Sub-NUMA Clustering (SNC) were turned off on Intel CPUs (except for the experiments with HPCG on BDW in Sect. I-A), NPS4 mode (i.e., four ccNUMA domains per socket) was active on AMD Rome, and SMT threads were ignored, i.e., each application thread was assigned its own physical core. We always employed the widest SIMD instruction set supported by the architecture (AVX2 except on CLX where it was AVX-512) and made sure that the compiler used standard (non-streaming) stores. The clock speed (core and uncore) was fixed to the base value via the likwid-setFrequencies tool from the LIKWID [9, 5] tool suite.

The transparent huge pages (THP) setting was put to “always,” and the NUMA balancing feature was turned off in order to reduce the performance impact from these settings [4]. All prefetching mechanisms in the hardware were enabled.

In order to run two groups of threads of different size and control thread-core affinity we used the likwid-mpirun tool. Working sets were chosen large enough to not fit into any cache (i.e., at least 10× the last-level cache size), and data sharing across overlapping kernels was ruled out to eliminate the possibility of cache reuse.

Runtime traces were visualized using Intel trace analyzer and collector (ITAC)². All timings were taken using the C++ high-resolution Chrono clock. Individual kernel executions were repeated at least 15 times to even out variations in runtime. Memory bandwidths were measured by taking the ratio of data volume and wall-clock time.

The HPCG benchmark results presented in the introduction were obtained with version 3.1, a problem size of 160³ (2.7 GB per rank), and a configured runtime of 1800 seconds. In addition to the standard compiler flags, we used -DPHCNG_NO_OPENMP for pure MPI, -DPHCNG_CONTIGUOUS_ARRAYS for contiguous data layout, and -trace -tcoll -tcollect-filter func.txt -qopt-report to enable trace data collection and analysis of selected user functions (configured in func.txt).

Further details can be obtained from the example code we put up for download at http://tiny.cc/ISPASS-OBSS.

² https://software.intel.com/en-us/trace-analyzer

Figure 4 visualizes the parameter space covered in this work with respect to numbers of threads sharing a contention domain. We only combine two distinct kernels, either filling the domain with respect to numbers of threads sharing a contention domain. The HPCG benchmark results presented in the introduction were obtained with version 3.1, a problem size of 160³ (2.7 GB per rank), and a configured runtime of 1800 seconds. In addition to the standard compiler flags, we used -DPHCNG_NO_OPENMP for pure MPI, -DPHCNG_CONTIGUOUS_ARRAYS for contiguous data layout, and -trace -tcoll -tcollect-filter func.txt -qopt-report to enable trace data collection and analysis of selected user functions (configured in func.txt).

Further details can be obtained from the example code we put up for download at http://tiny.cc/ISPASS-OBSS.
TABLE II: Key specifications of computational kernels used in the experiments. All arrays and scalars hold double-precision floating point data. In all cases with write access, standard stores and write-back caches are assumed so that write-allocate transfers cannot be avoided. Note that data transfers and code balance pertain to the L3 cache for the stencil codes.

### III. CODE AND PERFORMANCE METRICS

The computational intensity of a loop is the ratio of arithmetic operations (“true work,” e.g., flops) to the number of bytes that must be transferred over a data bottleneck in order to do the work. It is the main parameter that goes into the Roofline model [10], which assumes only two hardware bottlenecks: memory bandwidth and peak computational performance. Its inverse, the code balance, appears at first sight to be a reasonable metric to quantify the “hunger” of a loop for data. However, from the point of view of refined performance models such as the ECM model [8, 7], arithmetic instructions are an overlapping component of code execution. This means that the execution time of an executing loop does not depend on the actual number of flops but on the time spent moving data through the memory hierarchy unless the arithmetic work becomes dominant. The performance of the loop in flops does depend on the number of flops, however, but this is not the observable metric we are concerned with here.

We cannot give a full introduction to the ECM model here; instead, we describe the components that are relevant for our observable metric we are concerned with here.

- $T_{OL}$: In-core execution time of loop instructions that are not contributing to $T_{L1Reg}$
- $T_{MReg}$: Time for all required data transfers over the memory interface assuming that the full saturated memory bandwidth can be utilized
- $\{T_i\}$ Times for data transfers over all relevant data paths in the cache hierarchy, i.e., L2-L3 and L1-L2 in a three-level cache hierarchy

The ECM machine model makes assumptions about how these contributions have to be put together to arrive at a single-core runtime contribution. For example, on all Intel server CPUs to date, only loads count towards $T_{L1Reg}$, and the runtime is

$$T_{ECM} = \max \left( T_{OL}, T_{Mem} + \sum_{i=1}^{n} T_i + T_{L1Reg} \right),$$

i.e., data transfers are nonoverlapping while all instructions executed (except loads) in the core overlap with everything. The ECM application model provides information about the actual code (instructions and their dependencies) and data transfer volumes. It is usually obtained by static code analysis and assumptions about data transfers. In case of pure streaming loop kernels without temporal reuse, it is straightforward to calculate the data volumes. For stencils, a layer condition (LC) analysis [8] provides this information. The ECM model
provides us with a resource-centric point of view on memory bandwidth utilization. The fraction of time the serial code occupies the memory interface is

$$f = \frac{T_{\text{Mem}}}{T_{\text{ECM}}}.$$  \hfill (2)

On a largely nonoverlapping memory hierarchy like in Intel server CPUs, the memory request fraction \( f \) is significantly smaller than one even for streaming kernels with no temporal locality; on AMD Rome, which has strongly overlapping characteristics, it is often close to one in this case. The value of \( f \) quantifies how much of the shared resource (the memory interface) can be utilized by a single core. In most memory-bound loops, \( f \) does not change if the number of flops (and hence the code balance) changes because the data transfers dominate in (1). This is why code balance is not a good metric for quantifying the need of a loop for data. In contrast to \( f \), it does not take machine characteristics like the overlapping behavior of the cache hierarchy into account.

The value of \( f \) can be determined by using performance counters to measure the memory bandwidth drawn when executing a loop kernel in single-threaded mode. It can also be predicted by the analytic ECM model if the required knowledge about the hardware and code (see above) is available. We use the first option here and determine \( f \) per kernel as

$$f = \frac{b_{\text{meas}}}{b_s},$$  \hfill (3)

where \( b_{\text{meas}} \) is the measured single-threaded memory bandwidth and \( b_s \) is the saturated (full-domain) memory bandwidth of the kernel.

The ECM model also predicts the scaling behavior of loops across cores on a contention domain. We use a simplified version of the recursive model presented in [6]. It assumes that, to together with their basic properties such as number of elements sharing requires to take the differences in \( b \) time at least on Intel CPUs, an accurate model of bandwidth data transfer only accounts for a fraction (\( f \)) of the execution time at least on Intel CPUs, an accurate model of bandwidth sharing requires to take the differences in \( b \) into account.

The maximum memory bandwidth \( b_s \) is not the same for all loop kernels. As a general rule on all x86 CPUs, read-only kernels achieve a somewhat (5%–15%) higher saturated bandwidth than kernels with write streams. While this effect is of minor importance in single-core modeling since the memory data transfer only accounts for a fraction (\( f \)) of the execution time at least on Intel CPUs, an accurate model of bandwidth sharing requires to take the differences in \( b \) into account.

Table II lists the loop kernels used for our experiments together with their basic properties such as number of elements transferred over the memory interface per iteration (except for the stencil, where we show transfers to/from the L3 cache), their (memory or L3) code balance, their \( f \) values, and their saturated bandwidths. We include two 2d stencil algorithms with different characteristics. For each we select two different grid sizes that exhibit layer conditions (LC) fulfilled or broken at the L2 cache; if the LC is fulfilled, the data transfers between L3 and L2 are reduced because reuse across the outer stencil dimension is possible at the L2 cache. This happens when three consecutive rows of the source grid fit into L2. For the stencil we thus list the code balance in L3 instead of in memory, because the in-memory code balance is the same regardless of the LC at L2. However, the intra-cache data transfers between L2 and L3 make a significant difference for the memory request fraction \( f \). See [8] for an in-depth coverage of layer conditions.

IV. ANALYTICAL MODEL

The central quantities in our bandwidth sharing model are the kernel’s memory request fraction \( f \) and the saturated bandwidth \( b_s \). As an example we look at a ten-core contention domain with six cores running kernel I \( (f^I) \) and four cores running kernel II \( (f^II) \) as depicted in Fig. 5. Each core issues requests at a certain fraction of the maximum rate; this fraction is \( f^I \) or \( f^II \), respectively. In the model, this is also the rate at which requests are queued to be serviced by the memory interface. A kernel with higher \( f \) will be able to queue more requests. Since we are dealing with a fully populated contention domain, the requests from the ten cores compete for bandwidth, which is limited by a value that depends (though weakly) on the code characteristics. This variation is phenomenological input to the model. In Table II we listed the saturation bandwidths \( b_s \) for all kernels at homogeneous execution (no mixing). When two different kernels overlap, this value changes, and we assume that the overlapped saturated bandwidth is a weighted mean of homogeneous bandwidths:

$$b(n^I_s, n^II_s) = \frac{n^I_s \times b^I_s + n^II_s \times b^II_s}{n^I_s + n^II_s}.$$  \hfill (4)

Here, \( b^I_s \) and \( b^II_s \) are the saturated bandwidths for the two kernels in homogeneous execution, and \( n^I_s \) and \( n^II_s \) are the respective numbers of threads, where \( n_s = n^I_s + n^II_s \).

As shown in Fig. 5, where we chose \( n^I_s = 6 \) and \( n^II_s = 4 \), we now assume that group I of cores gets a share of all requests, and thus a share of the bandwidth, proportional to

$$\alpha^I = \frac{n^I_s \times f^I}{n^I_s \times f^I + n^II_s \times f^II} \quad \text{and} \quad \alpha^II = 1 - \alpha^I.$$  \hfill (5)
In the homogeneous case \( f^I = f^{II} \), the share of bandwidth is solely determined by the number of threads in each group, but different request fractions modify this simple linear behavior. The bandwidth obtained by group I is thus \( \alpha^I \times b(n_1^I, n_2^I) \).

This model, although we have derived it starting from the assumption that the ccNUMA domain is fully populated, can also be applied to the nonsaturated case. In the following section we will validate the model using measurements in the parameter space depicted in Fig. 4.

V. MODEL VALIDATION AND DISCUSSION

In this section we first present measurements of selected scenarios (i.e., kernel pairings) on the four architectures and compare with the model (5), distinguishing between the fully populated domain case and the case where the number of cores goes up till saturation. We then extend the view to more pairings and finally present an overview of the modeling error for all cases.

In Fig. 6 we show three different kernel pairing scenarios per architecture using fully populated ccNUMA domains (i.e., 10, 18, 20, and 8 cores on the four architectures (a)–(d)), covering the orange dots in Fig. 4. The top panel in each column shows a stacked graph of the bandwidth share of kernel I (DCOPY, green) versus kernel II (DDOT2, hatched red) as the number of threads on kernel I is increased and the number of threads on kernel II is reduced. The top line in the graph is the measured overall memory bandwidth. Since DCOPY has a higher \( f \) than DDOT2 (see Table II), we expect from the model (5) that DCOPY will get a higher share of the bandwidth as the number of DCOPY threads goes up. This is observed as the upward “bend” of the separator between the two regions of the graph. The overall memory bandwidth goes down because the saturation bandwidth of DCOPY is smaller than that of DDOT2 (which is a read-only kernel that gets a little more bandwidth out of the memory interface). This behavior is quite universal across all four architectures for this kernel pairing.

The second panel in each column is a direct comparison between the modeled (lines) and observed (symbols) memory bandwidth per core for the DCOPY+DDOT2 case. Since we chose a zoomed-in y axis, the general downward trend stemming from the decline in saturated bandwidth is now more pronounced. Although not directly visible from the mathematical description, this change is just as important for the observed bandwidth as the difference in \( f \). Obviously, the model describes the per-core bandwidth quite accurately.

The third and fourth panels in each column show per-core bandwidth for two other pairings: JacobiL3-v1+DDOT1 and STREAM+JacobiL2-v1. Especially in the latter case the saturated bandwidths of the two kernels are very similar, so the downward trend is weaker and also the bandwidth difference per core is smaller. Nevertheless the model also provides an accurate prediction here.

Figure 7 shows the same cases as above but for symmetrical thread scaling, running the same number of threads per kernel (blue dots in Fig. 4). There is good agreement between model and measurement along the bandwidth saturation curve as well, but some peculiarities are worth noting. The notable breakdown in bandwidth for DCOPY near saturation on BDW-1 is reminiscent of the overall scaling behavior of this kernel on this architecture, which shows a maximum bandwidth already before the domain is full. While the BDW CPUs and especially Rome show a rather strong decline in per-thread bandwidth already at two threads per kernel, CLX scales well from two to four, which reflects the fact that its single-core bandwidth is low.
compared to its saturated bandwidth. Rome is special in terms of scaling since all kernels can almost saturate the memory bandwidth already with one thread due to its overlapping cache hierarchy. A more complete coverage of scaling results for more pairings can be found at http://tiny.cc/ISPASS-OBS.

In Fig. 8 we give a concise overview of the relative modeling error in per-core bandwidth across 30 kernel pairings along the bandwidth scaling curve, i.e., for symmetrical pairing. In 75% of all cases the error is below 5%, and the maximum overall error is 8%. In view of the significant differences in architectural details across the four CPUs (inclusive vs. victim LLC, shared vs. segmented LLC, overlapping vs. serializing caches) we consider this an exceptional result.

Figure 9 shows an overview of 32 kernel pairings (including self pairings) of vecSUM, DDOT2, DDOT3, DCOPY, Schoenauer, DAXPY, DSCAL, JacobiL2-v1, JacobiL3-v1, and TRIAD with others using an equal share of threads on a full contention domain. Each row of bars (a)–(d) refers to one architecture. Each bar represents the relative bandwidth share of the first kernel in the pair (e.g., vecSUM in vecSUM+DDOT2) as opposed to the homogeneous situation (e.g., vecSUM paired with itself). All bars in a group are thus normalized to the first bar.

The first observation is that the patterns observed in each group of bars are quite consistent across architectures: Whether bandwidth is gained or lost with respect to the homogeneous situation depends on the ratio of $f$ values of the kernels being greater or smaller than one. Across the Intel CPUs, this criterion is independent of the architecture. The CLX CPU is still special in the sense that the differences in bandwidth are smaller overall. One might be tempted to attribute this to the fact that it needs more cores to saturate the memory bandwidth than the BDW variants, i.e., it is “more scalable.” In other words, the memory transfers need less time relative to the other contributions in the single-core ECM model, which cause generally smaller request fractions $f$ as can be seen in Table II. However, a global reduction factor in $f$ cancels out in the model (5), hence this cannot be the reason for the weaker bandwidth variations. The
we observed how back-to-back compute kernels overlapping (a)–(d). The height of each bar is the relative gain or loss in bandwidth of the first kernel in the pair when paired with the variations on CLX. The AMD Rome CPU shows different bulk-synchronous barrier-free programs. desynchronization and resynchronization in memory-bound bottlenecks; it can also predict qualitatively the dynamics of a quantitative description of performance differences on shared causes resynchronization. Our model is thus not only good for shown in Fig. 3(a) on the trailing edge of the DDOT2 execution, is amplified in such a situation. Overlapping with idleness, as DDOT2 up kernel to DDOT2 was DAXPY, with $f_{\text{DAXPY}} = 0.315$ and $f_{\text{DDOT2}} = 0.252$. This large difference directly leads to the positive observed skewness and means that the desynchronization is amplified in such a situation. Overlapping with idleness, as shown in Fig. 3(a) on the trailing edge of the DDOT2 execution, causes resynchronization. Our model is thus not only good for a quantitative description of performance differences on shared bottlenecks; it can also predict qualitatively the dynamics of desynchronization and resynchronization in memory-bound bulk-synchronous barrier-free programs.

VI. CONCLUSION AND OUTLOOK

Starting from observations of desynchronized kernel execution on the MPI-parallel HPCG benchmark we motivated the need for an analytical model of bandwidth sharing between groups of threads executing different kernel pairs with different characteristics on a bandwidth contention domain. Based on the principles of the Execution-Cache-Memory (ECM) performance model, we constructed and validated such a model on four current x86 server processors from Intel and AMD. We could show that the major influence factors for bandwidth sharing between two groups of threads are each kernel’s memory request frequency, which is directly related to its single-threaded memory bandwidth, and its saturated bandwidth on the domain. Across a variety of pairings of kernels with and without cache reuse, the observed error in predicted per-core bandwidth was never larger than 8%, and lower than 5% for 75% of all cases. Apart from the quantification of bandwidth shares, the model is also able to predict how desynchronization across MPI processes gets amplified or mitigated depending on which back-to-back kernels are overlapped with each other in MPI-parallel barrier-free bulk-synchronous programs.

The set of loop kernels chosen here is a reasonable cross section with a spectrum of properties. Since the memory request fraction and the saturated memory bandwidth are the only relevant parameters in our model, it should be applicable also for more complex kernels (e.g., with more concurrent data streams or with dominant in-core execution). It should also be useful in modeling the performance of task-parallel code where the synchronized, data-parallel execution of threads may be
more the exception than the rule. A validation of the model on more processor architectures, e.g., Power- or Arm-based CPUs, is certainly in order. Finally, our work enables the development of a new kind of MPI simulation technique that can take node-level bottlenecks into account much more accurately than previously possible. We leave these investigations for future work.

ACKNOWLEDGMENTS

This work was supported by KONWIHR, the Bavarian Competence Network for Scientific High Performance Computing in Bavaria, under project name “OMI4papps.”

REFERENCES

[1] A. Afzal, G. Hager, and G. Wellein. Delay flow mechanisms on clusters. URL: https://hpc.fau.de/files/2019/09/EuroMPI2019_AHW-Poster.pdf. Poster at EuroMPI 2019, September 10–13, 2019, Zurich, Switzerland.

[2] A. Afzal, G. Hager, and G. Wellein. Desynchronization and wave pattern formation in MPI-parallel and hybrid memory-bound programs. In P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, editors, High Performance Computing, pages 391–411, Cham. Springer International Publishing, 2020. ISBN: 978-3-030-50743-5. DOI: 10.1007/978-3-030-50743-5_20.

[3] A. Afzal, G. Hager, and G. Wellein. Propagation and decay of injected one-off delays on clusters: A case study. In 2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019, pages 1–10, 2019. DOI: 10.1109/CLUSTER.2019.8890995.

[4] C. L. Alappat et al. Understanding HPC benchmark performance on Intel Broadwell and Cascade Lake processors. In P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, editors, High Performance Computing, pages 412–433, Cham. Springer International Publishing, 2020. ISBN: 978-3-030-50743-5. DOI: https://doi.org/10.1007/978-3-030-50743-5_21.

[5] J. Eitzinger and T. Röhl. LIKWID web pages. https://github.com/RRZE-HPC/likwid.

[6] J. Hofmann, G. Hager, and D. Fey. On the accuracy and usefulness of analytic energy models for contemporary multicore processors. In R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, editors, High Performance Computing, pages 22–43, Cham. Springer International Publishing, 2018. ISBN: 978-3-319-92040-5. DOI: 10.1007/978-3-319-92040-5_2.

[7] J. Hofmann et al. Bridging the architecture gap: abstracting performance-relevant properties of modern server processors. Supercomputing Frontiers and Innovations, 7(2), 2020. ISSN: 2313-8734. DOI: 10.14529/jsfi200204.

[8] H. Stengel, J. Treibig, G. Hager, and G. Wellein. Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. In Proceedings of the 29th ACM International Conference on Supercomputing, ICS ’15, Newport Beach, CA. ACM, 2015. DOI: 10.1145/2751205.2751240.

[9] J. Treibig, G. Hager, and G. Wellein. LIKWID: a lightweight performance-oriented tool suite for multicore environments. In 2012 41st International Conference on Parallel Processing Workshops, pages 207–216, Los Alamitos, CA, USA. IEEE Computer Society. Sept. 2010. DOI: 10.1109/ICPPW.2010.38.

[10] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 52(4):65–76, 2009. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.