# Empirical study of Amdahl's law on multicore processors

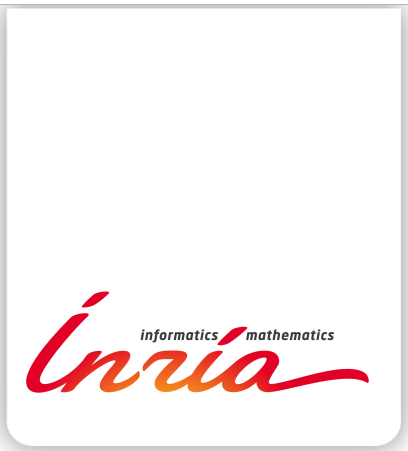
Carsten Bruns, Sid Touati


# Empirical study of Amdahl's law on multicore processors

**Carsten BRUNS, Sid TOUATI**

**RESEARCH REPORT**
**N° 9311**
May 2020 (2nd version)
Project-Team KAIROS

ISSN 0249-6399 ISRN INRIA/RR--9311--FR+ENG




# Empirical study of Amdahl's law on multicore processors

Carsten Bruns* , Sid Touati*

Project-Team KAIROS

Research Report n° 9311 — May 2020 (2nd version) — 59 pages

**Abstract:** Since several years, classical multiprocessor systems have evolved to multicores, which tightly integrate multiple CPU cores on a single die or package. This shift does not modify the fundamentals of parallel programming, but makes harder the understanding and the tuning of the performances of parallel applications. Multicores technology leads to sharing of microarchitectural resources between the individual cores, which Abel et al. [1] classified in storage and bandwidth resources. In this work, we empirically analyze the effects of such sharing on program performance, through repeatable experiments. We show that they can dominate scaling behavior, besides the effects described by Amdahl's law and synchronization or communication considerations. In addition to the classification of [1], we view the physical temperature and power budget also as a shared resource. It is a very important factor for performance nowadays, since DVFS over a wide range is needed to meet these constraints in multicores. Furthermore, we demonstrate that resource sharing not just leads a flat speedup curve with increasing thread count but can even cause slowdowns. Last, we propose a formal modeling of the performances to allow deeper analysis. Our work aims to gain a better understanding of performance limiting factors in high performance multicores, it shall serve as basis to avoid them and to find solutions to tune the parallel applications.

**Keywords:** Scalability of parallel applications, Multicore processors, Shared resources, Bandwidth saturation, Shared power and temperature budget, Dynamic frequency scaling, Amdahl's law, Program performance modeling, OpenMP, Benchmarking.

* Université Côte d'Azur, I3S-Inria laboratory

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

# Étude empirique de la loi d'Amdahl sur les processeurs multi-cœurs

**Résumé :** Cela fait plusieurs années que les systèmes multi-processeurs ont évolué vers des systèmes multi-cœurs. Cette évolution ne bouleverse pas les fondements de la programmation parallèle, mais rend plus difficile l'analyse et l'optimisation des performances des codes. La technologie multi-cœurs engendre un partage de ressources micro-architecturales entre les cœurs individuels, classifiées en ressources de stockage ou de bande passante d'après les travaux d'Abel et al [1]. Dans ce document, nous effectuons une analyse fine et empirique des effets de ce partage de ressources sur les performances. Nous montrons qu'ils dominent la scalabilité des temps d'exécution, au delà des considérations de synchronisation et de communication modélisées dans la loi d'Amdahl. En plus de la classification étudiée dans [1], nous regardons la température physique et la puissance électrique comme des ressources partagées. Elles deviennent des facteurs très importants pour les performances actuellement; la modulation de fréquence (DVFS) est utilisée dans presque tous les systèmes multi-cœeurs à hautes performances. Aussi, nous montrons que le partage de ressources micro-architecturales engendre non seulement une stagnation des accélérations (le *speedup* de l'application en fonction du nombre de threads parallèles sur cœurs physiques), mais parfois une dégradation (*slowdown*). En dernier, nous proposons une modélisation formelle des performances d'applications parallèles permettant une analyse plus fine. Notre travail permet une meilleure compréhension des facteurs limitants les performances des applications parallèles sur systèmes multi-cœurs, servant de base ensuite pour l'analyse et l'optimisation de ces performances.

**Mots-clés :** Scalabilité des applications parallèles, Processeurs multi-cœurs, Ressources partagées, Saturation de la bande passante, Budget partagé de puissance et température, Modulation dynamique de fréquence, Loi d'Amdahl, Modélisation de la performance des programmes, OpenMP, Test de performance.

# Contents

# 1 Introduction

The performance demands for computing systems are continuously increasing. Though, individual CPU cores cannot be improved a lot anymore since fundamental limits are already closely approached: clock frequencies cannot be increased further due to power constraints, the amount of Instruction-Level Parallelism (ILP) present in programs is well exploited and memory systems, composed of main memory and caches, barely improve. The only available solution is to use multiple cores in parallel, to benefit from coarse-grained parallelism provided by multithreaded applications or by a workload composed of a set of applications. Historical such multiprocessor systems use a separate die and package for each computation core. Nowadays, manufacturers are able to build multicore processors, also called Chip Multiprocessors (CMPs). Those integrate multiple cores closely together on a single die, or at least in the same package, thus they simplify the system design, reduce the cost and most important allow for more cores in a system. Multicore processors have become the state of the art in High-Performance Computing (HPC), servers, desktops, mobile phones and even start to be used in embedded systems. The current flagship processor of Intel for example includes 56 cores (Xeon Platinum 9282, 2 dies of 28 cores in a common package) and AMD released the Ryzen Epyc Rome with up to 64 cores (8 individual dies with 8 cores each plus an I/O die in a single package). For a precise definition of all terms related to processors, as we use them in this document, please also have a look at the Glossary (page 54).

This tight integration directly results in a way closer coupling between the individual cores. Some resources are usually shared between the cores for numerous reasons:

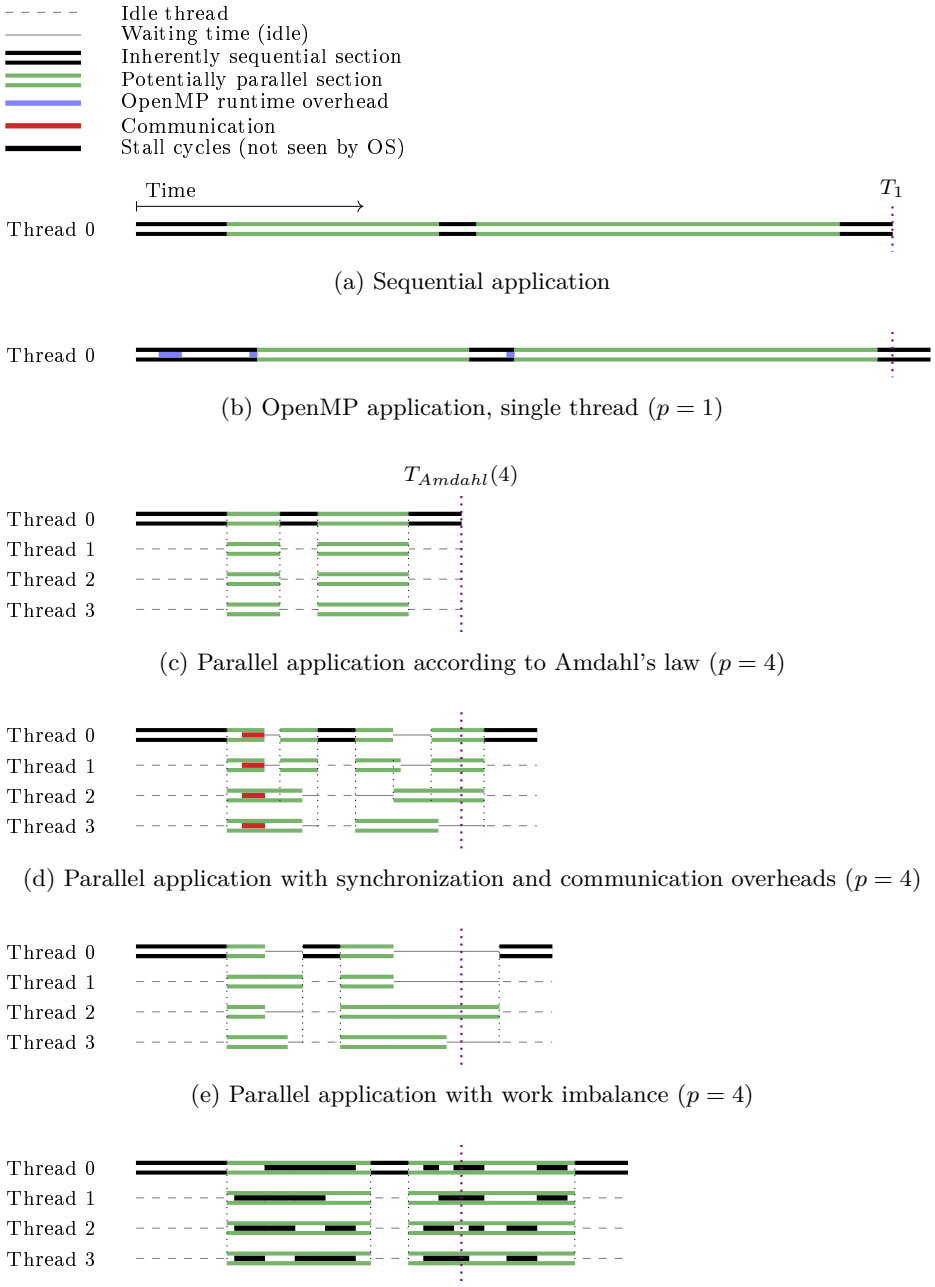
- to save chip area: e.g. shared interconnect/Network-on-Chip (NoC), which also impacts accesses to the memory controller, L3 cache, etc.;
- to dynamically use resources where they are needed, to redistribute them to cores that need them at the moment: e.g. shared caches;
- but also just due to the physical integration on a common die/package: e.g. power consumption, common cooling system.

Even though some of those can be avoided, they might be wanted. Chip designers would make a resource shared to save area only if they assume it does not degrade performance in most usage scenarios and using the freed area for other functionality improves the overall system performance in the end. Allowing to redistribute resources among cores (e.g. cache capacity) can likewise be beneficial when co-running applications have different demands. Nonetheless, under certain conditions, it might still degrade performance. In any case, sharing resources allows for interference between the cores. It consequently has strong implications for the parallel performance of the computing system and needs to be taken into account when optimizing for performance. Let us therefore first have a detailed look on the runtime of parallel applications.

## 1.1 Runtime of parallel applications

Figure 1 illustrates the runtime for different application types. A sequential program contains a single thread running on a single CPU core as in Figure 1a. It may contain code parts that could be parallelized (green) and parts that are inherently sequential (black, e.g. initialization code). When we use OpenMP to parallelize the program, the OpenMP runtime adds some overhead, even though it might be very small. Thus, when we use only a single thread as in Figure 1b, an OpenMP program version might be slightly slower than the purely sequential version.

If we now execute the application in parallel, the sections that could be parallized scale with the thread count $p$, whereas sequential sections still execute in the same time as before (Figure 1c). The overall runtime decreases, but can never get lower than the runtime of the sequential sections. Note that we here assume that always at least as many physical resources (cores) as threads are available




(a) Sequential application

(b) OpenMP application, single thread ($p = 1$)

(c) Parallel application according to Amdahl's law ($p = 4$)

(d) Parallel application with synchronization and communication overheads ($p = 4$)

(e) Parallel application with work imbalance ($p = 4$)

(f) Parallel application on a multicore processor with shared resource conflicts ($p = 4$)

Figure 1: Illustration of runtimes of parallel applications

and that all our threads are scheduled on one of those during the whole execution time, so that the threads actually run in parallel on the machine.

In a practical application as in Figure 1d, the parallel threads need to exchange data which costs additional time (red). Furthermore, synchronization between threads might be required, in addition to the implicit synchronization at the beginning and end of parallel sections. For example, one thread might need an intermediate result from another thread. This adds idle times to the execution in which

some threads wait for other threads to reach the common synchronization point (black vertical dotted lines). Moreover, in some cases it might not be possible to distribute the parallel work equally among all available threads. Some threads then have more computations to do than others and consequently need more time for a parallel section, like in the example case of Figure 1e. Threads with a small amount of work assigned experience additional idle times. All those phenomena are visible to the OS and therefore reported to the user.

The effects that interest us in this work are interferences between cores in multicore processors. When executed on a multicore, threads might compete for access to shared resources. Whenever such a resource is saturated, i.e. cannot serve all incoming accesses at the same time, CPU cores have to wait for it to get free and the operation takes longer to complete. As a result, the cores have to *stall* for some time as depicted in Figure 1f and the runtime of the parallel sections of the executed threads increases. Those stalls happen transparently in the underlying processor hardware, i.e. the OS does not see them and reports a high CPU usage with low idle times to the user. In a practical application, those effects will likely occur combined with communication/synchronization overheads and potentially also with work imbalance issues.

A good way to assess this kind of performance impacts is through the parallel scaling behavior of an application. The purely sequential program version (Figure 1a) serves as a baseline, as it potentially has lower overheads as the parallelized version with a single thread (Figure 1b). We then analyze the relative runtime of parallel executions compared to this base version, that is the speedup, when increasing the number of used threads and with that (active) cores. This way, we can see how much performance the additional resources (cores) really provide. Amdahl's law states a first limit to this kind of scaling due to the sequential program fraction, which does not profit from parallelization (Figure 1c). Further works consider overheads caused by synchronization and communication as illustrated in Figure 1d.

## 1.2 Contributions

In this work, we extend these limits with effects caused by the previously described close integration in multicore processors, i.e. due to shared resources as in Figure 1f. We show that those can be dominant factors for parallel scaling and thus limit the maximum performance. We do so with repeatable experiments. To allow the best possible understanding, we use a very well studied benchmark in HPC: different versions of matrix multiplication. Based on the gained data, we analyze different classes of resource sharing effects in CMPs, including their characteristic behavior and how they can reduce performance. With the knowledge we gain in this report, the effects can then easier be identified and possibly avoided by programmers and/or system designers. Furthermore, our observations also show that performance can even decrease when increasing the number of used cores with modern multicore processors. Last but not least, we provide an approach to formally model scalability in the presence of shared resources.

The rest of this report is structured as follows. In Section 2 we review the current state of the art for parallel scalability and resource sharing. Section 3 describes our experiments and hardware setup in detail. We present the resulting data in Section 4, including a detailed analysis of different classes of resource sharing. Section 5 provides the modeling part. Section 6 summarizes the results and concludes.

# 2 State of the art

Before explaining our experiments and results, let us review the current state of related parallel performance modeling. We briefly recall Amdahl's law and introduce current extensions to it. Then, we show related works containing measurement data with a behavior not well explained by Amdahl's law, motivating this work. Afterwards we talk about studies of interference between cores from the real-time community. They will help us to describe the effects we observe in our experiments. Last, we present works on scheduling for CMPs which face similar microarchitectural effects as the ones limiting multithreaded application scalability in our experiments.

## 2.1 Performance of parallel applications and Amdahl's law

Predicting runtime of code snippets or complete programs on modern x86-64 architectures is hard: even the Intel Architecture Code Analyzer (IACA) developed by the chip manufacturer, which has knowledge of all the internals of the hardware, achieves low accuracy [2]. However, often we are just interested in how a program scales when increasing parallelism, i.e. how its runtime behaves relative to the sequential version when using multiple computation resources.

Naturally, we expect that each additional resource augments performance, i.e. the speedup increases. Amdahl's law states a limit to this due to a part of a program which cannot be parallelized (e.g. initialization code). Thus, the overall runtime can never be lower than the time needed to execute this sequential part [3]. Let $\sigma \in [0, 1]$ be the fraction of execution time of this part when the program runs on a single resource. The total runtime $T_1$ can then be decomposed like in Figure 1a into the runtime of its sequential part $T_{seq}$ (black) and the time spent in its parallel part $T_{parallel}$ (green) as:

$$\begin{aligned} T_1 &= T_{seq} &&+ T_{parallel} \\ &= \sigma T_1 &&+ (1-\sigma)T_1 \end{aligned} \tag{1}$$

We now execute the code in parallel with $p$ threads. Each of those threads runs on a distinct physical processor (resource), or more precisely core. We then expect the runtime of the parallel part to scale (perfectly) and the time of the sequential part to stay constant, as in Figure 1c. A function $T_{Amdahl} \colon \mathbb{N}^* \to \mathbb{R}$ then describes the execution time for different values of $p$:

$$T_{Amdahl}(p) = \sigma T_1 + \frac{(1-\sigma)T_1}{p} \tag{2}$$

This results in a function $Speedup_{Amdahl} \colon \mathbb{N}^* \to \mathbb{R}$ representing the parallel acceleration:

$$\begin{aligned} Speedup_{Amdahl}(p) &= \frac{T_{Amdahl}(1)}{T_{Amdahl}(p)} = \frac{T_1}{\sigma T_1 + \frac{(1-\sigma)T_1}{p}} \\ &= \frac{1}{\sigma + \frac{(1-\sigma)}{p}} \end{aligned} \tag{3}$$

Note that $\lim_{p\to\infty}(Speedup_{Amdahl}(p)) = \frac{1}{\sigma}$. This shows how the maximum acceleration achievable by parallelization of an application is limited by the program's sequential fraction.

Even though Amdahl's law always stays an upper bound, we will see that it is not tight anymore and thus does not describe well the actual scaling for modern multicore processors. Other effects, not modeled in this simple formula, tend to dominate the behavior.

## 2.2 Extensions to Amdahl's law

The literature contains many reviews and extensions to Amdahl's law. We explain the three which are most relevant to our work in this section.

### 2.2.1 Trade-off between number of cores and core size

Hill and Marty [4] extend Amdahl's law for multicore chip design. Assume a baseline core which needs a certain amount of physical resources (e.g. transistors). The authors then consider a chip with a fixed total resource budget $b \in \mathbb{R}$ counted in multiples of this baseline core, i.e. the baseline core uses one resource unit. In other words, the chip contains a fixed number of resources which can be used by the designer, e.g. to implement up to $b$ baseline cores. Their work further assumes runtime performance of a single core to grow with the amount of resources $r \in \mathbb{R}$ used to implement it, which are again counted in multiples of the baseline core. A function $perf_{res}\colon \mathbb{R} \to \mathbb{R}$ describes the relative performance of a single core, depending on $r$, as a speedup compared to the baseline core. Hill and Marty are then interested in the trade-off between the performance of individual cores and the number of cores that fit on the chip. The authors use $perf_{res}(r) = \sqrt{r}$ as an example in their analysis. This means a core using four times the resources of the baseline core provides twice its performance. Thus, using bigger cores is beneficial for sequential code parts but performs worse for parallel sections as the number of cores $p = \lfloor \frac{b}{r} \rfloor$ gets smaller and the performance of each core grows only less than $r$. To simplify the modeling, the authors omit the floor function for the number of cores and use a continuous approximation instead, i.e. 4.5 cores on a chip are valid and we obtain simply $r = \frac{b}{p}$. Without restricting their model, we can still consider $p \in \mathbb{N}^*$ only. This results in the $Speedup_{Hill}$ function (adapted from [4]):

$$Speedup_{Hill}(p) = \frac{1}{\frac{\sigma}{perf_{res}(\frac{b}{p})} + \frac{(1-\sigma)}{perf_{res}(\frac{b}{p}) \times p}} = \frac{perf_{res}(\frac{b}{p})}{\sigma + \frac{(1-\sigma)}{p}} \tag{4}$$

They show that a small sequential fraction is still crucial for multicores to obtain good parallel speedups. More important, they highlight that for larger sequential fractions a smaller number of more powerful cores can be beneficial even if they offer lower maximum performance for parallel parts. They also consider Asymmetric Multiprocessor (AMP) cases where one core is more powerful than the others, as well as dynamic architectures which can rearrange their resources either for faster sequential execution or for more parallel cores that are less powerful. In contrast to this work, we are here concerned with effects occurring in a fixed hardware architecture. In particular, we cannot change the size of individual cores. We can just either use cores or let them idle, exactly as any end user of the hardware. Nevertheless, we will see that processors with Intel's Turbo Boost actually approach a dynamic AMP: the power available to the chip can be used dynamically either for faster execution of a single core or for more cores in parallel.

### 2.2.2 Adding communication and synchronization

In [5], Yavits et al. extend this model with inter-core communication and sequential to parallel data synchronization, similar to the phenomena shown in Figure 1d. They define $f_1\colon \mathbb{N}^* \to \mathbb{R}$ as the connectivity intensity depending on the number of used cores $p$. This function gives the time spent on communication, divided by the sequential runtime. Similarly, they represent the time spent on data synchronization relative to the sequential runtime and depending on $p$ as the synchronization intensity $f_2\colon \mathbb{N}^* \to \mathbb{R}$. As in the base model of Hill and Marty, $p = \frac{b}{r}$. The parallel speedup predicted by their model is then:

$$Speedup_{Yavits}(p) = \frac{perf_{res}(\frac{b}{p})}{\sigma + \frac{(1-\sigma)+f_1(p)}{p} + f_2(p)} \tag{5}$$

The authors argue that the impact of the two additionally modeled effects increases with parallelism and that even good parallelizable programs show better performance on a smaller number of powerful cores instead of many small cores when including those effects. Consequently, Amdahl's law predicts speedups significantly too high for tasks dominated by synchronization or communication not modeled by it. They

conclude that reducing the sequential fraction is only important when it is large and actually degrades performance scaling stronger than synchronization or communication effects. Otherwise, targeting those overheads might be more important. They also consider asymmetric core sizes like Hill and Marty.

In this work, we also show that other factors than the sequential fraction get more and more relevant for parallel scalability. However, we are interested in effects occuring in the hardware architecture of multicores, probably triggered by software behavior, whereas [5] focuses solely on workload properties (communication/synchronization). In addition, we present experiment data on real hardware opposed to simulations which can always only include the effects caused by features modeled in the simulator.

#### 2.2.3 Abstract model based on queuing theory

The Universal Scalability Law (USL) originally presented in [6] by Gunther and further analyzed in [7] is another model for parallel speedup which is based on queuing theory. It uses two abstract parameters $\sigma \in [0, 1]$, as in Amdahl's law, and $\kappa \in [0, \infty)$:

$$Speedup_{USL}(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)} = \frac{1}{\kappa(p-1) + \sigma + \frac{(1-\sigma)}{p}} \tag{6}$$

An interpretation of the parameters is that $\sigma$ accounts for contention effects and $\kappa$ for overheads to keep the system's parallel computing resources (e.g. caches) in a coherent state. Since contention behavior cause serialization in this model, the parameter $\sigma$ is equivalent to the serial fraction in Amdahl's law. USL thus simply adds a higher order term in the denominator of Equation (3) for coherency. This is similar to communication overhead in the model of Yavits et al. In a later work, the authors explain how their model can be used to guide performance tuning [8]. They claim that due to the foundation in basic queuing theory and the abstract nature of the parameters, contention and coherency effects on software as well as on hardware level can be described. Even though the model includes interesting ideas and can explain retrograde scaling (performance loss), it is not sufficient to explain well our experimental data. The hardware effects occurring in our experiments on multicore architectures behave in a more complex manner. Moreover, we want to establish a clear understanding of what are these effects and not just describe the resulting performance behavior with an abstract model.

### 2.3 Reported measurements beyond Amdahl's law

Many papers present scalability curves which cannot be well described by Amdahl's law. A superlinear speedup is reported in [9] for dense matrix multiplication. That is a scaling behavior better than linear with the number of used cores, so better than Amdahl's law with $\sigma = 0$. The same authors give reasons for such behavior in [10]. Non-persistent algorithms finish when one thread found a solution and thus might finish in less total instructions when run in parallel mode. Persistent algorithms (like matrix multiplication) can show constructive data sharing in shared caches. One core might implicitly prefetch data from memory for another core. The most common case they identify is an increased amount of available cache capacity when using more cores. Private caches of cores can only be used when executing on all cores in parallel, thus we do not only add computing power but also increase the total cache capacity when increasing the number of used cores. This is especially important for architectures with large private caches, like the Skylake processors we use: Intel increased the private L2 cache to 1 MiB from only 256 KiB in previous generations.

Didier et al. include a speedup curve which decreases with increasing thread count after a certain maximum [11], whereby they always use less or the same amount of threads as physical cores available. This is predicted by their timing model as well as occurring in their measurements. They argue that this happens due to memory access interference, where all accesses together saturate the available memory bandwidth. We come back to this explanation after presenting our results in the conclusion.

The performance evaluation of the Skylake architecture in [12] also contains measurements for matrix multiplication. Again the thread count is always smaller or equal to the core count. Their curve

contains steps, i.e. flat regions and increases only at certain thread counts, similar to what we observe in our data in Figure 5a (page 26). On the steps, performance sometimes even decreases for increasing thread count. The authors attribute all this behavior, especially the end of scaling after 16 used cores, to thermal throttling of the chip. After our experiments, we will see that this is only part of the cause.

Even the extensions to Amdahl's law shown in Section 2.2 cannot explain any of those behaviors well. Most of the speedup curves in the literature are not well analyzed and no or only a simple interpretation is provided. This motivates our work.

## 2.4 Interference analysis for real-time systems

Nowadays, even in the community of real-time systems, multicores are more and more used in order to meet increasing performance requirements. Safe upper bounds on the Worst Case Execution Time (WCET) are needed here, thus researchers started to study how cores interfere with each other through shared resources in multicore systems.

Wilhelm et al. give a first static timing analysis of caches and buses in [13]. They highlight issues in the timing analysis due to sharing of those resources in multicores. For example, to gain information on shared cache states all possible interleavings of accesses from the different cores need to be considered. As a consequence, the authors instead recommend to try to eliminate interferences on shared resources as much as possible in the hardware architecture design. In our context of HPC, this is obviously infeasible as e.g. shared caches are crucial for performance. Furthermore, Wilhelm et al. discuss that modern architectures are pipelined with complex features (e.g. out-of-order execution). They might thus exhibit timing anomalies as presented in [14] and formally defined in [15]. Simplified, those are situations where a local worst case does not cause the global worst case of execution time, e.g. a cache miss might in the end result in a shorter runtime compared to a hit due to following decisions of other processor features. Based on this, Wilhelm et al. [13] classify architectures as fully timing compositional (no timing anomalies), compositional with constant-bounded effects (timing anomaly effects can be bounded with a fixed number of penalty cycles) and noncompositional architectures (timing anomalies can have arbitrary effects). Most proposed static analysis for resource sharing architectures in the literature (e.g. [16] and [17]) are only feasible for (fully) compositional architectures. Penalties of individual effects can be viewed in isolation and then added together in this case. Otherwise, all effects have to be viewed together which results in exploding complexity. Hahn et al. [18] provide a clearer notion of this property. However, the Intel Skylake processor (x86-64) we use in this work has to be assumed noncompositional. In addition, many internals of the processors are secrets of Intel so that we cannot model them for a detailed timing analysis. Nonetheless, we can still derive an approximate parallel scaling behavior similar to the models of Section 2.2.

Measurement based approaches aiming to quantify interference through experiments complement the static analyses. In order to obtain the worst-case interference, Radojković et al. [19] use resource stressing benchmarks, i.e. they run a test code that makes extensive use of a specific shared resource. This might be very pessimistic as a real co-running application would stress the resources less. Furthermore, it might not be clear what is the worst-case interference for a noncompositional architecture, due to complex interactions. It might rarely occur in measurements. Also, as we see in our experimental data, only multiple cores together might be able to stress a shared resource fully.

A general overview over resource sharing is provided in [1]. The authors group shared resources into bandwidth resources (e.g. shared buses) and storage resources (e.g. shared cache capacity). Resources might be shared between all cores in the system, at chip level or even just between pairs of cores. We use this classification as a basis in this work.

## 2.5 Scheduling for multicore processors

Resource sharing and especially contention of resources has been studied in the context of co-scheduling, where a workload composed of multiple applications needs to be scheduled over time and on the set of

available cores of a system.

Antonopoulos et al. [20] observe that the memory bandwidth often limits performance for (bus-based) computing systems. They propose co-scheduling policies which measure the memory bus bandwidth usage of applications and select co-runners such that they use as much of the available bandwidth as possible in each scheduling quantum, but do not exceed its limit.

In [21], Knauerhase et al. focus on balancing the load between multiple Last Level Caches (LLCs). The researchers argue that the number of cache misses per cycle indicate cache interferences well, thus they use it as heuristic to guide their co-scheduling decisions.

Zhuravlev et al. [22] make a more detailed analysis of the impact of various shared resources when two applications run together - they study the DRAM controller, the Front Side Bus (FSB), the LLC and resources involved in prefetching (which also includes the memory controller and the FSB). Based on this, they discuss schemes to predict mutual degradation of co-running applications, including sophisticated strategies based on cache access patterns. The authors conclude that simply the number of LLC misses per cycle, as previously used by Knauerhase et al., serve as a good heuristic to estimate the degree of contention, because the misses highly correlate with DRAM, FSB and prefetch requests. The motivation for the heuristic thus differs from [21], where LLC contention is the optimization target.

The work of Bhadauria and McKee [23] considers multithreaded programs, their scheduler in addition decides on the number of threads to use for each application. In their solution, an application which does not scale well to high thread counts because it saturates a hardware resource can run with a lower thread count and another application with lower intensity on this resource is co-scheduled on the remaining cores. As heuristic to predict shared resource contention, they also use the LLC miss rate, and the occupancy of the (shared) data bus between processor chips in their target system.

Sasaki et al. [24] follow a similar approach but base their scheduling decisions only on the scalability of the programs when run in isolation, instead of the contention for resources caused by the possible schedules.

In [25] the authors consider modern NUMA machines where each processor chip manages part of the memory, instead a single memory accessed over a shared bus. Four factors are identified to be important: contention either for the LLC, memory controller or interconnect, as well as the latency of remote memory accesses. The previous solutions are found to not work well in such a situation because they frequently change the core on which a thread runs but the data stays in memory at the original node. This increases interconnect contention, contention for the memory controller at the remote node and introduces remote memory access latency. Only LLC contention is reduced by the co-scheduling algorithm on such a hardware architecture. The authors thus propose a new algorithm, which moves threads only when clearly beneficial and also migrates a certain amount of actively used memory pages with the thread to the new node. The load on the memory controller is better balanced and the chip interconnect has to handle only a few remote memory accesses.

In addition, works targeting Simultaneous Multithreading (SMT) processors exist, for example [26, 27, 28]. Those also try to maximize the symbiosis between co-running threads, but at the level of an individual physical core instead of at chip level. The aim is again to achieve a high, yet not overloading, usage of shared hardware resources.

Scalability of a parallel application, which is the topic of this work, can be seen as a special case of co-scheduling where all threads have identical characteristics. The threads of a single multithreaded application can in addition share memory regions, which allows for constructive cache sharing in this case. Still, similar microarchitectural resources will present bottlenecks and insights can be used in both disciplines.

The next section introduces our experiments and provides all details required to understand and reproduce our results shown afterwards in Section 4.

# 3 Experiment setup

Performance measurements highly depend on many details of the experiment design as well as the used hardware and software configuration. We thus detail here the most important parameters of our data collection environment needed to understand the obtained scalability curves in Section 4. We provide many additional details to allow reproduction of the results. As [29] shows, additional hidden factors might influence performance and add bias to our measurements. However, we are confident that the general conclusions are valid and can be recreated with the given information.

We repeat all our measurements $N$ times with identical parameters to capture variations which can be significant for parallel (OpenMP) applications as shown by Mazouz et al. [30], we use $N = 50$ executions. Our obtained data is thus not a single value but multiple observations of random variables of which we report the complete distributions instead of aggregating them into a single value like median or mean.

In this section, we first define important terms. Then, we describe the used hardware and software environment. The last part treats our methodology including the used test applications, parallel and sequential program versions, NUMA considerations and gives details on our way of measuring.

## 3.1 Definitions of used metrics and scalability

Let us first define some terms important for this work, to avoid any ambiguity.

**Experiment execution** With an experiment execution we mean a specific instance of running an application. In the context of this work, we can uniquely identify such an execution by the application $A$, the used thread count $p \in \mathbb{N}^*$ and the index of the measurement repetition $n \in \{0, 1, ..., N-1\} =: I$, so by the tuple $E = (A, p, n)$. The application $A$ in here includes the actual executed binary code as well as the runtime configuration (in particular OpenMP parameters), except for the used thread count $p$ which we denote explicitly to ease the definition of our metrics. We call the set of all applications $K$.

**Runtime** Each of our application contains multiple timed sections, in particular multiple calls of the benchmark function which Section 3.4.5 details further. Those timed sections are denoted by $s \in \mathbb{N}_0$. For one timed section $s$ of an experiment execution $E = (A, p, n)$, we distinguish three different times:

- $t_{real}(A, p, n, s)$ is the *real* time, i.e. the wall clock time, needed for the timed part of the application. We also refer to this as runtime.
- $t^i_{user}(A, p, n, s)$ denotes the time which thread $i$ of our application spent in *user* mode
- $t^i_{sys}(A, p, n, s)$ means the time that thread $i$ was running in *system* (kernel) mode

**Speedup** For each application, we also generate a corresponding purely sequential version. It serves as a baseline, i.e. we use it as a reference for the parallel versions. We detail this further in Section 3.4.3. All speedups presented in this work are the speedup in the real time between a specific experiment execution $E = (A, p, n)$ and the median runtime of all repetitions of this associated sequential program version. Since the sequential versions show small variability [30] and the base runtime only introduces a fixed scaling, using just the median of all sequential executions as a reference is acceptable. Let us denote $A_{seq}$ the sequential application belonging to an application $A$. The function $Speedup_{execution} \colon K \times \mathbb{N}^* \times I \times \mathbb{N}_0 \to \mathbb{R}$ then gives the speedup of a timed section $s$ for one specific execution:

$$Speedup_{execution}(A, p, n, s) = \frac{median\{t_{real}(A_{seq}, 1, i, s) \mid i \in I\}}{t_{real}(A, p, n, s)} \tag{7}$$

Similarly, we define the function $Speedup_{empirical}\colon \mathbb{N}^* \to \mathbb{R}^N$ which returns for a thread count $p$ the set of speedups obtained in the different repetitions. We consider a fixed application $A$ and a given timed section $s$, so those two become parameters instead of function variables:

$$\begin{aligned} Speedup_{empirical}(p) &= \{Speedup_{execution}(A,p,j,s) \mid j \in I\} \\ &= \{\frac{median\{t_{real}(A_{seq},1,i,s) \mid i \in I\}}{t_{real}(A,p,j,s)} \mid j \in I\} \end{aligned} \tag{8}$$

**CPU usage (per thread)** The CPU usage is given by a function $CPU\_usage_{execution}\colon K \times \mathbb{N}^* \times I \times \mathbb{N}_0 \to \mathbb{R}$ for one execution. We define it divided by the thread count $p$ to get a metric which is comparable for different thread counts:

$$CPU\_usage_{execution}(A,p,n,s) = \frac{\sum_{i=0}^{p-1}(t^i_{user}(A,p,n,s) + t^i_{sys}(A,p,n,s))}{p \times t_{real}(A,p,n,s)} \tag{9}$$

This is a value between 0 % and 100 %. For our experiments, the system time is small and negligible. Note that e.g. the `top` command in Linux uses a different notion and reports a total CPU usage which is not divided by the thread count, so a value between 0 % and $p \times 100$ % instead.

This metric allows us to monitor the fraction of the execution time during which the cores used by our application are actually doing computations, as seen by the OS and reported to users. Low values indicate that part of the cores are idle at some point during the application execution, e.g. because of a sequential program part or because they are waiting for another core to finish (synchronization). We show that the effects we study in this work are hidden from the OS, i.e. the reported CPU usage is high even though the cores actually spend a large amount of time waiting (stalling).

Analogous to the speedup in Equation (8), let us define a function $CPU\_usage_{empirical}\colon \mathbb{N}^* \to \mathbb{R}^N$ returning the set of all values for all repetitions of the same application $A$ in a given timed section $s$:

$$\begin{aligned} CPU\_usage_{empirical}(p) &= \{CPU\_usage_{execution}(A,p,j,s) \mid j \in I\} \\ &= \{\frac{\sum_{i=0}^{p-1}(t^i_{user}(A,p,j,s) + t^i_{sys}(A,p,j,s))}{p \times t_{real}(A,p,j,s)} \mid j \in I\} \end{aligned} \tag{10}$$

**Scalability** Under scalability we understand the behavior of the application and our machine when increasing the exploitation of coarse-grained data parallelism, i.e. increasing the number of parallel executed threads on the fixed hardware architecture. In particular, we mean the speedup of runtime while keeping the workload constant as in Amdahl's analysis (strong scaling, see Section 2.1) opposed to work size scaling proposed by Gustafson (weak scaling) [31]. We scale a single (OpenMP) application, with each parallel thread executing exactly the same code with the same properties (type of used instructions, cache usage, ...). This is very different from scalability of task level parallelism, e.g. of a web server for more users which might execute different tasks at the same time.

## 3.2 Machine description

The machine we use for all our experiments is a Dell Precision 7920 workstation. Figure 2 gives a simplified view on the characteristics of its architecture. It is equipped with an Intel Xeon Gold 6130 multicore processor in each of its two sockets. Each of those chips contains 16 physical cores, or 32 logical cores with Intel's SMT technology called Hyper-Threading. The whole machine thus has 32 physical or 64 logical cores. For a precise definition of all terms related to processors, as we use them in this document, please also have a look at the Glossary (page 54). The cores support AVX-512

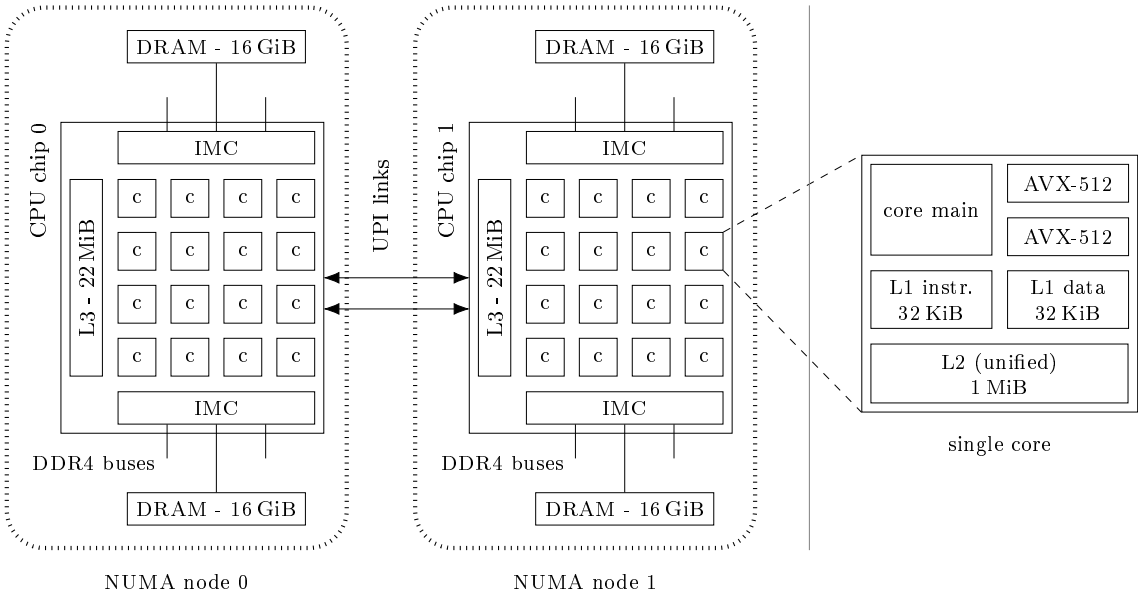



Figure 2: Architecture of the test machine

SIMD instructions for which they include two vector execution units. They have private L1 caches for instructions and data (32 KiB each), also private L2 caches (1 MiB, unified, inclusive) and a non-inclusive L3 cache shared between all cores of the chip (22 MiB, unified). Table 1 summarizes the most important properties of this cache hierarchy.

Two Integrated Memory Controllers (IMCs) per CPU chip offer three DDR4-DRAM channels each, so 12 channels in total. In our machine, only a single 16 GiB module (running at 2666 MT/s[1]) is connected to each IMC, leading to 64 GiB of total memory. A DDR4 bus is 64 bit wide, hence transmits 8 bytes in each transfer. All the four buses together in parallel thus result in a theoretical available peak memory bandwidth of $4 \times 2666\,\mathrm{MT/s}^1 \times 8\,\mathrm{bytes/T}^1 = 85.33\,\mathrm{GB/s}$ for our machine. The two chips are connected over two cache-coherent Intel Ultra Path Interconnect (UPI) links. Each CPU chip together with the RAM connected to it, however, forms a Non-Uniform Memory Access (NUMA) node. That implies that accesses to memory locations physically located in memory at the other node are slower than accesses to local memory locations.

The CPUs use a base frequency of 2.1 GHz. Our aim is to generate reproducible results. We thus deactivate the Dynamic Voltage and Frequency Scaling (DVFS) features of the processors in our experiments as far as possible. Since those mechanisms still have an impact on our experiment results, the next section describes them as far as required to follow our argumentation during the data analysis.

[1] *T* denotes data *transfer* operations on a communication bus. For a bus using Double Data Rate (DDR), two transfers occur per clock cycle - one on the rising and one on the falling edge of the clock signal. The amount of data transmitted per transfer is equal to the bus width. $1\,\mathrm{MT} = 10^6\,\mathrm{T}$.

Table 1: Cache hierarchy of the Skylake architecture [32]

| | Content | Size | Topology | Inclusion | Associativity | Line size | Latency (cycles) |
|---|---|---|---|---|---|---|---|
| **L1 instr.** | instructions | 32 KiB | private | - | 8-way | 64 B | |
| **L1 data** | data | 32 KiB | private | - | 8-way | 64 B | 4-5 |
| **L2** | unified | 1 MiB | private | inclusive | 16-way | 64 B | 14 |
| **L3** | unified | 1.375 MiB/core | shared | non-inclusive | 11-way | 64 B | 50-70 |

### 3.2.1 DVFS in Intel processors

Intel uses complex DVFS mechanisms in their modern processors to limit power consumption and keep the heat dissipation in a reasonable range. Sleep states (C-states) allow idle cores to consume less power by turning off parts of the cores, generally the deeper the sleep state is the less power is consumed but the longer is the wake-up time. The OS controls these and explicitly requests the desired C-state. When a core is in running mode, so in C0 state, it can use different P-states (performance states), mapping to different operating frequencies and voltages. Those thus offer various performance levels at corresponding power consumptions. Under normal conditions, e.g. a cooling system as the specified Thermal Design Power (TDP) of the chip, the CPU should be able to maintain all these frequencies steadily on all cores. Turbo Boost (2.0) extends the performance levels above the nominal (rated) clock frequency. The CPU might not be able to keep those frequencies over a long time and the concretely achievable frequency depends on the actual conditions, often referred to as *dynamic overclocking*.

The most important of those conditions are power consumption and temperature. Latter is measured constantly and in case the chip or individual cores get too warm lower frequencies are selected. Power drawn by the chip should never be larger than what the power supply can provide. For short times (peaks) a relatively high consumption might be acceptable, Intel calls this power limit 3 (PL3). Steadily the supply can provide a bit less power which imposes the next limit (PL2). In the long term, no more power than the cooling system can handle should be consumed (PL1), to prevent running into the overheating case. The power consumption is thus monitored and core frequencies are also throttled if any of these limits is exceeded [33]. In order to not always run into those dynamic limits, the maximum frequency is also limited by two static and deterministic factors: the type of the executed workload and the number of active cores. Here, the type of workload simply means whether or not vector instructions are used and with which vector width. Intel calls those modes license levels and the cores determine their level by the actual density of the different vector instructions (none, AVX2, AVX-512) in the instruction stream, i.e. a single AVX-512 instruction will not cause the highest level. Also, some simple instructions (e.g. shuffle) might belong to a lower level than the more complex ones (e.g. Fused Multiply Add (FMA)). Table 2 gives the allowed maximum frequencies for the CPU in our machine. *Base* here indicates the guaranteed long term frequency. The number of active cores is determined by cores being either in C0 or in C1/C1E state [34]. Just cores in higher sleep states are considered idle. Only the license mode of each individual core matters for its own maximum frequency, i.e. also if 15 cores run none vector code, the $16^{\text{th}}$ core is limited to 1.9 GHz if it uses AVX-512. On the other hand, if 15 cores run an AVX-512 load and the last core uses no vector instructions, it can clock up to 2.8 GHz.

In many cases the advantage of the larger vector width of AVX-512 vanishes, as a consequence of the lower allowed frequencies due to higher power consumption. In addition, directly following code not using vector instructions anymore can be slowed down as Gottschlag and Bellosa show in [37]. This is likely the reason why compilers use AVX-512 instructions conservatively and only when explicitly told to do so through compiler flags - we come back to this in Section 3.4.2.

The importance of dynamic throttling and static frequency limits varies depending on the processor model. A low power laptop processor (low TDP) will frequently suffer dynamic throttling whereas for a (well cooled) server or workstation processor, like in our experiments, the static limits are sufficient in most cases. We verified that the actual temperature during our benchmarks is far below critical values in all cases, such that no dynamic throttling occurs and the frequency behavior is predictable.

Table 2: Maximum clock frequencies in MHz of the Xeon Gold 6130 [35, 36]

| **License** | | | **Active cores** | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **level** | **Instructions** | **Base** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 0 | non vector | 2100 | 3700 | | 3500 | | 3400 | | | | 3100 | | | | 2800 | | | |
| 1 | AVX2 | 1700 | 3600 | | 3400 | | 3100 | | | | 2600 | | | | 2400 | | | |
| 2 | AVX-512 | 1300 | 3500 | | 3100 | | 2400 | | | | 2100 | | | | 1900 | | | |

Historically, P-states were managed by the OS and frequencies in the Turbo Boost range were automatically used by the hardware when the highest (non-turbo) P-state was chosen. In recent processors, including the here used Skylake chip, Intel completely moved the control over P-states to the hardware with the Speed Shift technology (Hardware Managed P-states, HWP). The OS can however still provide an allowed range of states, now also explicitly including the turbo states. The presented overview is simplified and additional mechanisms, as package level C-states, complement the described ones.

As mentioned we deactivate DVFS through Turbo Boost and P-states in our experiments to generate reproducible results. Please note that the cores might still adapt their clock frequencies under certain conditions as we see later in our results. On the other hand, we directly allow C-states during the experiments as (used) cores should not enter any sleep state in our high load experiments anyway and this configuration is closer to an usual scenario. We discuss the impact of this choice in Section 4.2.5.

## 3.3 Software environment

Our experiment applications run under a Linux Mint 19 (Tara) OS, with Linux kernel 4.15.0. To reduce interference from other processes during our measurements, we boot the OS in a custom `systemd` target which minimizes the load on the (idle) system. Only a minimum set of required services as well as `SSH` are running in this mode. Especially, no graphical environment is started.

We use GCC 8.2.0, ICC 19.0.0.117 and Clang/LLVM 7.0.1 to compile our C++ test codes. Applications using OpenMP link the default OpenMP runtimes in the same versions as the compilers - for GCC to `libgomp`, for ICC and Clang to `libiomp5` respectively `libomp5`, which use the same source for the implementations. Experiments with Intel's MKL use it in version 2019.0.0.

## 3.4 Methodology

As explained, we repeat each measurement $N$ times and thus obtain distributions of random variables for our metrics of interest instead of just a single value. Whenever we claim a difference between two settings, we checked this statistically with the methods provided in [38]. Nonetheless, we employ several methods to reduce variability to a minimum. These include minimal background services, explicitly binding threads to physical cores, as well as controlling NUMA memory allocation. The following sections detail those further, but first we introduce the applications we use for our experiments.

### 3.4.1 Test applications

The applications we measure in our experiments perform one of the most common and basic examples for parallel computing: matrix multiplication. This problem is a well studied benchmark in HPC. It concentrates in a small kernel many code performance optimization problems and hardware-software interaction phenomena, including instruction scheduling, register allocation, vectorization, loop nest transformation for cache optimization and loop parallelization. Finding the best of all these possible transformations still remains a challenging problem. This single application is thus versatile in itself, i.e. it can show many different microarchitectural behaviors, depending on the actual used machine code implementation. The task is further embarrassingly parallel, i.e. it can be easily decomposed into parallel tasks which are independent of each other and thus do not require any synchronization or communication. This is done by computing a subblock of the overall result matrix in each task. Just one implicit synchronization at the end of the operation is present, as the whole multiplication is only finished when all independent subtasks are completed (persistent algorithm). Thus, the overall runtime is determined by the last finishing task. Consequently, the algorithm is expected to scale very well with increasing parallel execution capabilities.

We analyze three different implementations to compute $M_C = \beta M_A \times M_B + \gamma M_C$ where $M_A$, $M_B$ and $M_C$ are matrices of sizes $D_N \times D_P$, $D_P \times D_M$ and $D_N \times D_M$. $\beta$ and $\gamma$ are scalars. Our first two implementations only take care of the special case where $\beta = \gamma = 1$ but can easily be generalized, however, this does not add value to our scalability measurements so that we decided to keep the code as simple as possible. In all experiments we use as matrix sizes $D_N = D_M = 4096$ and $D_P = 3072$ to get sufficiently large matrices but still allow the execution of all our experiments in a reasonable time. The data type of individual matrix elements is either float (4 bytes in memory) or double (8 bytes), depending on the experiment. Consequently, for float the matrices $M_A$ and $M_B$ occupy 48 MiB of memory and $M_C$ has a size of 64 MiB. In the case of double, the matrices are twice as large so 96 MiB, respectively 128 MiB. Please note that with these sizes the matrices cannot be fully present in the cache hierarchy of our test machine. We now describe each of the the three implementations in detail.

**a) *Simple*** The *simple* implementation is a straightforward code with three nested loops as shown in Listing 1. Only the order of the loops - which are interchangeable - is optimized so that accesses to the matrix data follow cache lines and have the best possible data locality behavior for the matrices which are stored in row-major order.

Listing 1: *Simple* matrix multiplication implementation

```
1 for (int i = 0; i < D_N; i++) {
2   for (int k = 0; k < D_P; k++) {
3     for (int j = 0; j < D_M; j++) { //AVX-512 vectorized
4       M_C[i][j] += M_A[i][k] * M_B[k][j];
5 } } }
```

**b) *Tiling*** The code of the *tiling* implementation, presented in Listing 2, explicitly implements tiling on all three loops using additional loops in the nest. This improves data locality and thus makes efficient use of the processor's caches. An additional loop at the innermost level (line 9), which gets unrolled, helps the compiler to further reduce data access times by re-using values in processor registers. It is basically another level of tiling on the k-loop. The block sizes, as defined by the code listing, are $BS_k^{float} = 32$, $BS_i^{float} = 32$, $BS_j^{float} = 1024$ and $BS_{k\_reg}^{float} = 4$ for float and $BS_k^{double} = 16$, $BS_i^{double} = 16$, $BS_j^{double} = 512$ and $BS_{k\_reg}^{double} = 4$ for double. Those values were chosen such that the data accessed by the different loop levels fits well the sizes of the different cache levels of our machine. We provide a detailed justification in Appendix A.

Listing 2: *Tiling* matrix multiplication implementation

```
1  for (int i0 = 0; i0 < D_N; i0 += BS_i) {
2    for (int k0 = 0; k0 < D_P; k0 += BS_k) {
3      for (int j0 = 0; j0 < D_M; j0 += BS_j) {
4
5        for (int k2 = k0; k2 < k0 + BS_k; k2 += BS_k_reg) {
6          for (int i = i0; i < i0 + BS_i; i++) {
7            for (int j = j0; j < j0 + BS_j; j++) { //AVX-512 vectorized
8
9              for (int k = k2; k < k2 + BS_k_reg; k++) { //re-use register data
10               M_C[i][j] += M_A[i][k] * M_B[k][j];
11             }
12       } } }
13 } } }
```

**c) *MKL*** Our *Math Kernel Library (MKL)* version uses the library from Intel to do the computation of the matrix product. It therefore calls the `cblas_sgemm` function, or respectively `cblas_dgemm` when

using double as data type. The actual implementation is therefore a secret of Intel and not public. However, we reverse-engineered some parts of it needed to explain our observations.

### 3.4.2 Compilation details

To compile our implementations, we use three different compilers: GCC, ICC and Clang. We pass the options `-O3 -std=c++17 -march=skylake-avx512` in all cases. This allows the compilers to use features specific to the Skylake microarchitecture of the processors in our test machine, especially it allows the compilers to use AVX-512 vector instructions. It also enables the highest optimization level composed of a set of optimizations. For all compilers, these include the loop transformations of automatic vectorization and unrolling which are relevant for our benchmarks. ICC also enables loop blocking and indeed applies it for our *simple* implementation. Note that also GCC and Clang provide similar functionality through their (experimental) polyhedral frameworks `Graphite` and `Polly`, however they have to be enabled explicitly.

All our experiment implementation were successfully SIMD vectorized by all three compilers. In all cases vectorization is applied to the innermost loop after unrolling, i.e. in the j-loop. Even though by specifying the CPU microarchitecture we enabled AVX-512 instructions, GCC and ICC are conservative with using the maximum vector width of 512 bits and fall back to 256 bits wide vectors instead. The rationale for this is that AVX-512 instructions are in many cases not worth to use and smaller vectors lead to better performance, especially because the processor is allowed lower maximum frequencies when using the full vector width, in order to limit power consumption and heat dissipation (see Section 3.2.1). For our matrix multiplication example, however, we want to make use of AVX-512. We thus force 512 bit vectors for GCC (`-mprefer-vector-width=512`) and ICC (`-qopt-zmm-usage=high`), Clang by default uses the full vector width. This allows each core to do 32 float FMAs or 16 double FMAs in parallel on their two AVX execution units introduced in Section 3.2, leading to high computation throughput but also the demand for fast data fetching. The arrays containing the matrix data are aligned to 64 bytes (512 bit) for aligned accesses by the vector instructions and best caching behavior, one cache line is also 64 bytes on our machine.

In addition, we add the `-ffast-math` flag for GCC and Clang which allows floating point calculations which are not compliant to the IEEE Standard for Floating-Point Arithmetic (IEEE 754) and might cause changes in the results, e.g. by changing the associativity of a floating point multiplication. ICC already allows similar optimizations with the `-O3` flag. Obviously, for experiments with OpenMP, the `-fopenmp` flag is also necessary. The linker is always instructed to link only the required libraries for each experiment (of OpenMP, MKL, `PAPI`).

### 3.4.3 Parallel and sequential versions, thread mapping

We create parallel and sequential versions of our codes. The parallel version makes use of (coarse-grained) data parallelism through OpenMP with a certain number of threads that we vary between measurements. In all our experiments, this thread count is smaller or equal to the number of physical cores in the system. For our two loop based implementations (*simple* and *tiling*), we always distribute the iterations of the outermost loop among the threads. In contrary, the sequential version runs in a single thread, i.e. it only uses a single CPU core. We do not link OpenMP in this case to avoid any overhead (compare Figure 1b vs. Figure 1a). However, note that fine-grained data-level parallelism is still used through SIMD execution (AVX-512) also in this version. It serves as a baseline case to which we relate reported speedups. MKL also provides sequential and parallel library versions to which we link our binaries accordingly.

Controlling thread affinity, i.e. binding threads to specific cores for execution, is crucial to reduce variability in program execution times [39]. Without fixing the affinity, the OS kernel is free to choose any mapping of threads to cores and, even worse, migrate threads from one core to another during execution. As some mappings might be superior in terms of performance compared to others, these

two phenomena lead to differences in runtimes. The kernels decisions depend on many internal factors, the influence is thus random from our point of view and we observe it as variability in the data.

The OpenMP specification since version 4.0 includes thread affinity control. We only present a simplified overview of the mechanisms. For more details, please refer to the specification [40]. In OpenMP terminology, a *place* defines a location to which threads can be mapped. It is a set of processors, in particular in our context a set of logical CPU cores as seen by the OS. The *place list* is the list of all the existing places. It is ordered, thus we can identify each place uniquely by its position in the list. Let us call the $(i+1)$-th place in this list $place_i$.

In our experiments, we tell OpenMP to create one place per physical core in the place list. This means that each place contains all the logical cores associated with one physical core. The place list is ordered by NUMA nodes, i.e. it includes first all places of NUMA node 0, followed by the places of node 1, and so on. Logical cores are also enumerated, so we can represent them by their numbers. Consider a machine with $m$ physical cores with each two logical cores as ours (2-way SMT). The numbers of all first logical cores are again grouped by NUMA nodes. Furthermore, the two logical cores mapping to the same physical core are the logical core $i$ and the logical core $i+m$, for $i \in \{0, 1, ..., m-1\}$. The $i$-th place in this setting is then given by a set of two (logical) core numbers:

$$place_i = \{i,\ i+m\} \quad \text{for } i \in \{0, 1, ..., m-1\} \tag{11}$$

As example we obtain $place_3 = \{3, 35\}$ for our machine, meaning that the fourth place in the place list contains the logical cores 3 and 35, both mapping to the same physical core. Since parallel regions might be nested in an OpenMP program, not all places of the place list might be available for use in a parallel region, but only a subset which is termed a *place partition*. However, in our test applications the place partition is identical to the place list.

A *policy* then defines how the threads are actually mapped to the available places in the place partition:

- The *master* policy assigns each thread to the same place as the master thread reaching the parallel region, which in our applications is always mapped to $place_0$.
- With *close*, threads are bound to adjacent places, starting at the encountering master thread. Figure 3a illustrates this policy for our machine with $m = 32$ physical cores when mapping $p = 20$ threads. Places available in the place partition are indicated below the cores and used cores are highlighted (filled, red).
- The *spread* policy in contrary aims to create a sparse distribution of the threads among the available places. Therefore, the place partition list is divided into smaller lists of adjacent places. Every place is included in exactly one of those subsets, which are called *subpartitions*. The number of subpartitions created is equal to the number of threads to be mapped. Then, one thread is assigned to the first place of each subpartition. When as above $m$ places are available in the place partition list and $p$ threads are to be mapped, for $p \leq m$ naturally each of those subpartitions will contain either $\lfloor m/p \rfloor$ or $\lceil m/p \rceil$ places, so their length might differ by one place. However, the specification lets open which of the subpartitions should be the smaller ones and which should be the bigger ones.

The OpenMP runtime of GCC (`gomp`) implements the spread policy in the simplest way possible: it first places the bigger subpartitions and then the smaller ones. We show this behavior in Figure 3b, where we also indicate the generated subpartitions with solid (blue) borders.

We can see that this is problematic in a NUMA system like ours, where cores are physically located on different NUMA nodes. In such a case, `gomp's` strategy of partitioning might lead to an imbalance between the nodes. For our example of Figure 3b, `gomp's` implementation leads to 8 threads assigned to cores on node 0 and 12 threads to node 1, so a strong imbalance. More general, if $p = \frac{m}{2}$ both nodes will have the same amount of threads assigned. If we increase the number of threads now, however, new

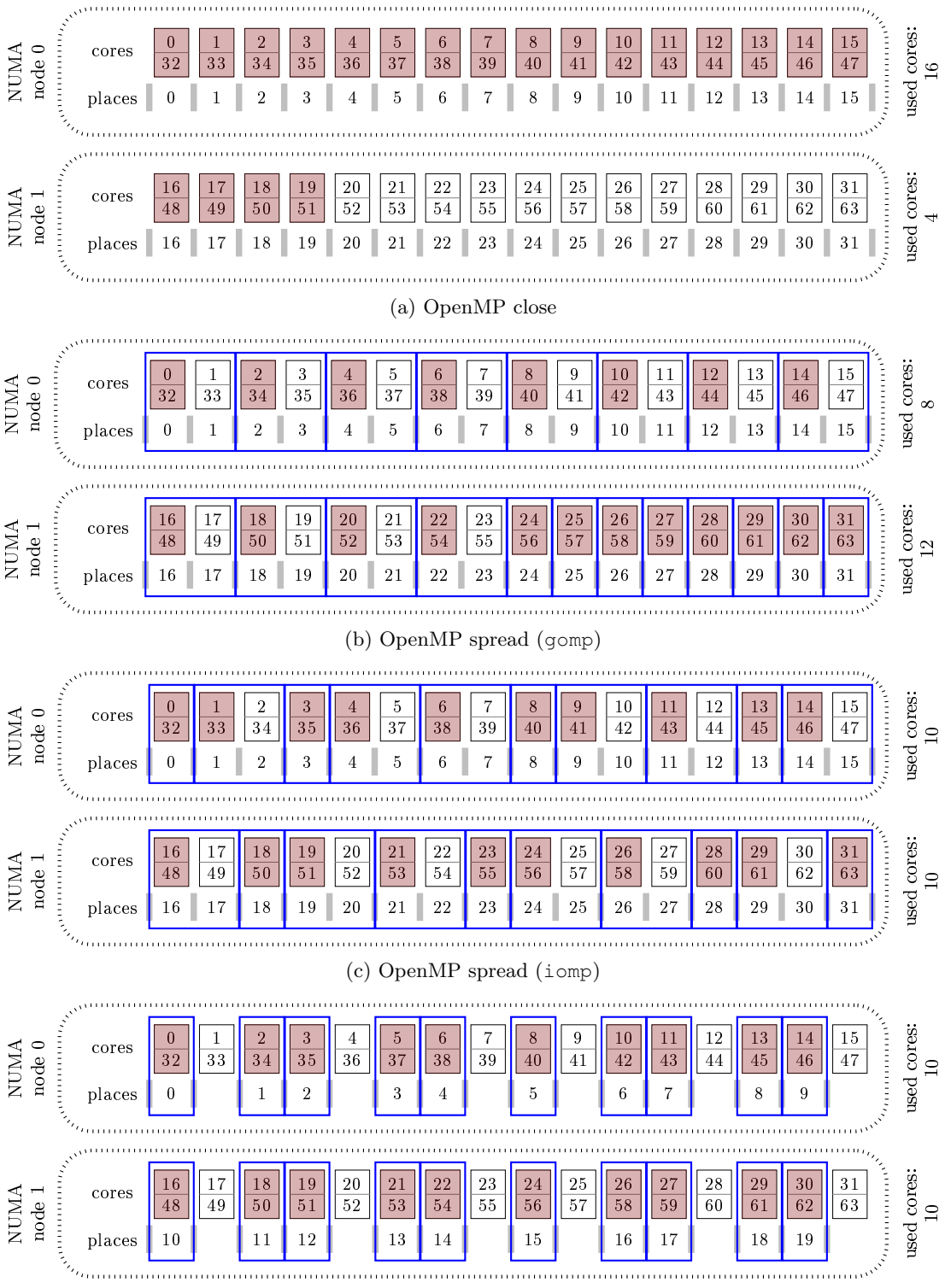



Figure 3: Thread mapping schemes, $m = 32$ on two NUMA nodes and $p = 20$

threads will always be added on node 1 until all its cores are used: partitions will have either length two or one and `gomp` first creates all subpartitions of size two.

The OpenMP runtime of ICC/Clang implements a more sophisticated partitioning scheme that assigns the same amount of threads to all NUMA nodes, except for one thread difference for odd thread counts. We see how this improves the situation in our example case in Figure 3c. Thus, we also implemented an affinity mapping scheme that assures the property of balance between NUMA nodes. To keep the implementation simple, we solely change the place partition. We provide to the OpenMP runtime a list containing one place for each physical core which we want to use for one of the $p$ threads, instead of all cores present in the system as before. Since we give only exactly as many places as threads in our application, this results in a fixed mapping, independent of the spread or close policy of OpenMP- both will map one thread to each of the places. Let $nint\colon \mathbb{R} \to \mathbb{Z}$ be the function that returns the nearest integer to its input, with round to the next even integer for ties. Each place contains again the two logical cores of one physical core, however, this time we define:

$$place_i = \{nint(i \times \frac{m}{p}),\ nint(i \times \frac{m}{p}) + m\} \quad \text{for } i \in \{0, 1, ..., p-1\} \tag{12}$$

This spreads the threads by maximizing the distance between the used core numbers, but as desired still ensures balance between NUMA nodes, as we clearly see in Figure 3d. We thus refer to this policy as *balanced* in this report.

In the literature, more complex affinity mapping schemes are studied (e.g. [41, 42]). Those are out of scope for this work, as we only want to remove thread mapping variability and obtain a balanced NUMA load distribution. Our balanced mapping is sufficient for these aims. The sequential version also forces the Linux scheduler to always run it on the same core (core 0) through the `sched_setaffinity` system call.

Note that we let Hyper-Threading activated but always just assign a single thread of our application to each physical core. This allows the OS to schedule other processes on logical cores mapping to a physical core used by our application but is only relevant when we use (almost) all physical cores.

### 3.4.4 NUMA memory allocation

Our test machine has two NUMA nodes as shown in Figure 2. Data in the main memory can thus be physically located in those two different locations. Allocation of memory happens in granularity of memory pages of usually 4 KiB size. By default, the Linux kernel applies the *local* policy: memory is allocated on the node from which it is first accessed. This is problematic when data initialization happens in a sequential part of the code, since all memory gets allocated on a single node, but the actual computation happens in parallel on multiple nodes [43]. The Linux kernel (since version 3.15) solves this by monitoring accesses and migrating pages to other nodes if judged beneficial by a heuristic. This is also useful when threads are not bound to cores and might be scheduled on different nodes over time. However, it introduces another non-deterministic factor and thus runtime variability, depending on when and how page migrations happen.

To avoid this, we use the *bind* and *interleaved* policies. The first allows memory to be allocated only on a subset of nodes and not to migrate afterwards. We only allow node 0 in this case. We also fix the memory allocation for the sequential program version, even though no migrations should happen as every access to the data comes from the same node. The second policy allocates pages in a round-robin fashion to an allowed set of nodes. Here, we allow both NUMA nodes, i.e. new pages are assigned alternating between the two nodes. Note that if we bind all data to a single node only half the memory bandwidth is available compared to the interleaved case.

### 3.4.5 Measurement method

A shell script controls the repetitions of an experiment instead of running the same function multiple times in the C++ code. This is important to make the executions independent of each other: each

repetition is done in a new process, such that e.g. caches and page tables are flushed. In addition, we give the system one second time to settle between each execution. The application is only compiled once for each configuration before its first repetition. Similarly, we also launch all the different experiments from one shell script such that their processes inherit the environment variables of the same parent process. Our experiment design thus ensures that all experiments run with exactly the same environment variables.

We measure all metrics directly in the C++ code. The benchmark function is called twice in each execution, as Intel's MKL library initializes internal buffers during the first call (in a single thread) and thus the first matrix multiplication is significantly slower than subsequent calls [44]. Both calls are measured individually, from the start of the function until its end. Especially, this also does not include the initialization of the matrices and program loading. We keep the data for each call as a separate data series, those are the different timed sections $s$ of an execution which we mention in the formal definitions in Section 3.1. Our code gets the *real* time through the `gettimeofday()` function and *system* and *user* times through the `getrusage()` system call. We obtain additional metrics for the same program parts through hardware performance counters with `PAPI` [45], such as actual core clock frequencies or cache miss counts. We report the sum of those counter values over all cores, or receptively over all cores of one NUMA node. To measure the memory access bandwidth, we use the Memory Bandwidth Monitoring (MBM) part of Intel's Resource Director Technology (RDT) included in the Skylake architecture. The `PQoS` library allows us to easily integrate this into our application. It requires access to Model-Specific Registers (MSRs). To facility our measurements, we modified the corresponding Linux kernel module to allow these as non-root user without the usually required `CAP_SYS_RAWIO` capability. However, this opens large security issues and is only reasonable for our system dedicated to measurements.

In the next section, we present the data resulting from the experiments described here. We report different effects that limit the applications scalability and analyze them in detail.

# 4 Empirical scalability analysis

We run the experiments described in Section 3 for different:

- implementations (*simple*, *tiling*, *MKL*)
- compilers (GCC, ICC, Clang)
- data types (float, double)
- NUMA memory allocation schemes (bind, interleaved)
- thread affinity policies (spread, balanced)

This results in 72 different configurations, which are all different applications $A$ in the sense of our definitions of Section 3.1. Since we want data for all thread counts $p$ smaller or equal to the number of available physical cores $m = 32$ in our machine, each of those requires 33 different experiments (32 threads counts and a sequential version). With 50 repetitions for each, we run a total of 118 800 executions. In the following we thus focus on interesting cases only. We denote configurations in the form `(implementation, compiler, data type, memory allocation scheme, affinity policy)`, e.g. `(simple, GCC, float, bind, spread)`. We use a `*` as placeholder when a parameter can be any of the possible options.

## 4.1 Observing Amdahl's law in practice

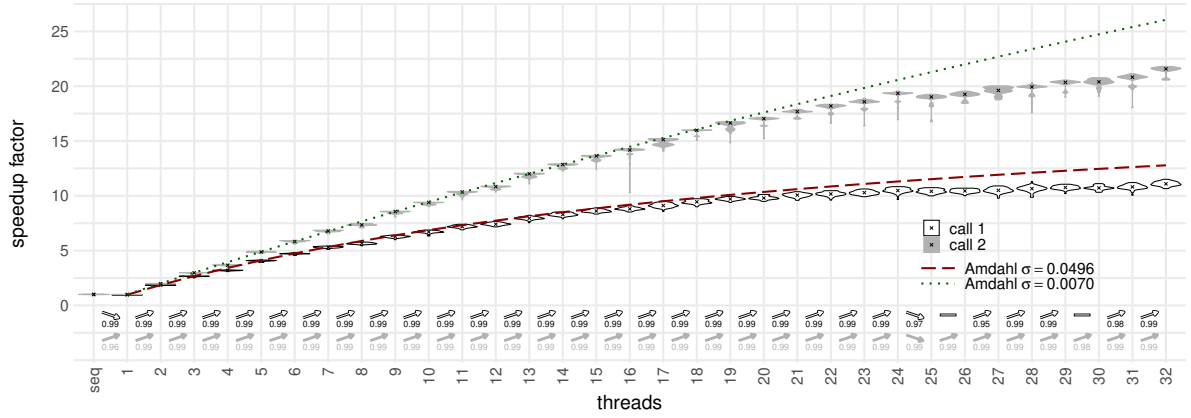
Our experiment data shows very different scaling behaviors, depending on the experiment configuration. In most of the cases the scaling is thereby dominated by effects that are not modeled in Amdahl's law. We analyze those in detail in Section 4.2. Thus, even though Amdahl's law stays an upper bound of the scalability curve, it does not describe well the behavior of our application on the test machine.

Only when no other limits are present, we can observe Amdahl's performance limit caused by the sequential fraction of a program. This is for example the case when using `(MKL, *, float, interleaved, balanced)`. All compilers show similar behavior since MKL is a precompiled library and the thread to core mapping is identical for both OpenMP runtime implementations (`gomp` and `iomp`), because the configuration uses our own balanced policy.

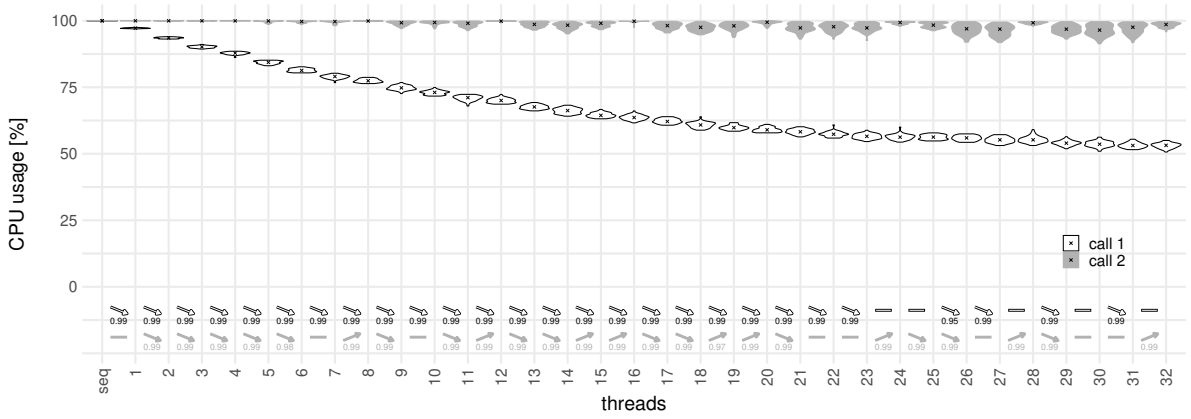
Figure 4 shows the obtained scaling behavior as violin plots for GCC. This type of plot is similar to a boxplot but in addition the sides of the boxes are rotated plots of the estimated probability density distribution, so that the resulting shapes resemble violins. Each individual violin in the plot represents all repetitions of one experiment with a certain thread count, i.e. one violin shows the output of Equation (8) for one input value in Figure 4a and the output of Equation (10) in Figure 4b. We mark the median values of the repetitions but omit quartiles, to avoid overloading the figures. Recall that we execute the benchmark function twice in each repetition. Therefore, we draw two series of violins, one for the first function call and one for the second call.

Arrows below the plot indicate if there is a statistically significant increase or decrease in the median of the plotted metric from one thread count to the next, again independently for the two data series. Numbers next to those indicate the maximum confidence level for which this trend could be proven. We use the protocol developed in [38] to obtain this information, which in its core uses a Wilcoxon-Mann-Whitney test but ensures that all hypotheses of the statistical test are met, meaning that the test is valid for our case and produces the desired result. The black (not filled) arrow between *seq* and *1* in Figure 4a shows for example that the median of the speedup for the first function call is smaller for the parallel version using a single thread than for the sequential (base) version, which could be proven with a confidence level of 0.99. We use the same type of plot throughout this report.

The speedup curves (Figure 4a) for both function calls have a behavior well described by Amdahl's law until 20 threads. Then, another effects decreases the performance compared to the simple bound,




(a) speedup




(b) CPU usage

Figure 4: Amdahl scaling - (`MKL, GCC, float, interleaved, balanced`)

which we analyze in Section 4.2.5. We fit Amdahl's law to the maxima of the data between 1 and 20 threads, excluding thread counts divisible by 4 (we explain why in Section 4.2.2), such that the value chosen as estimation for the sequential fraction $\sigma$ minimizes the Mean Squared Error (MSE). Figure 4a includes curves showing the result: $\sigma_1 = 0.0496$ for the first call (red dashed curve) and $\sigma_2 = 0.0070$ for the second call (green dotted curve). This clearly shows how the longer sequential part of MKL in the first function call, caused by the initialization of internal buffers in a single thread, limits the scalability in this case. Figure 4b also illustrates this through the CPU usage (per thread). Since in a sequential part only one thread is active and the others are waiting, the average CPU usage over all threads decreases when adding more threads.

In most cases, however, other effects than purely the sequential program fraction dominate scalability on modern multicore processors. We describe and analyze those in the next section.

## 4.2 Effects dominating scalability

Let us now look at factors other than the one described by Amdahl's law that lower the performance increase achievable by using more computing cores in parallel. The following subsections treat the different effects we observe in our experiments one by one. We start with considerations on the work

distribution. Then, we show an example where the thread count is actually influencing the behavior of the algorithm. Last, we describe three different cases of resource sharing: bandwidth resource saturation, storage resource conflicts as well as a shared power and temperature budget. In this section, we report and plot data for the second function call only, as this case excludes sequential initialization phases which are just executed once per program run, in particular the internal initialization of MKL as shown in the previous section. The smaller sequential fraction allows us to better observe and analyze the effects occurring in the parallel section of the program.

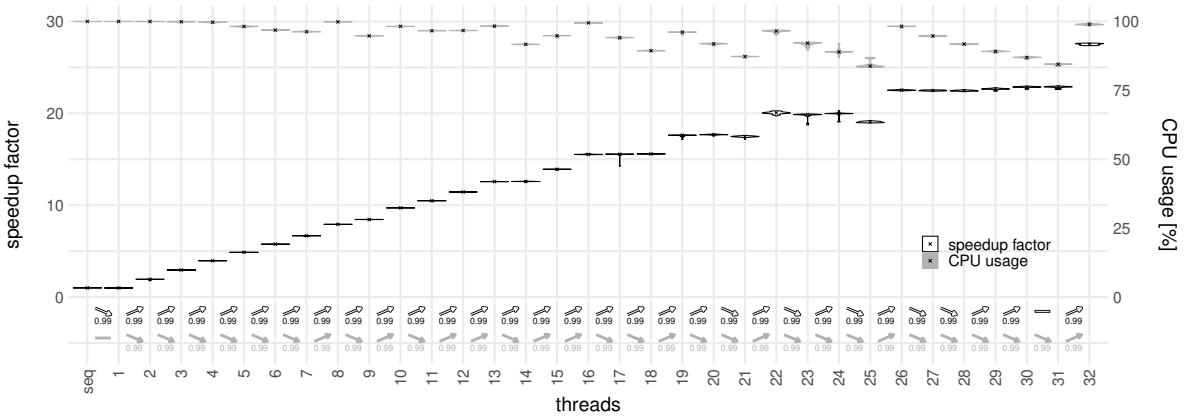
### 4.2.1 Work distribution

A well known factor limiting parallel speedup is unbalanced work distribution among the available computing resources, as the overall performance is determined by the last finishing computation like shown in Figure 1e. We observe four different types of this issue.

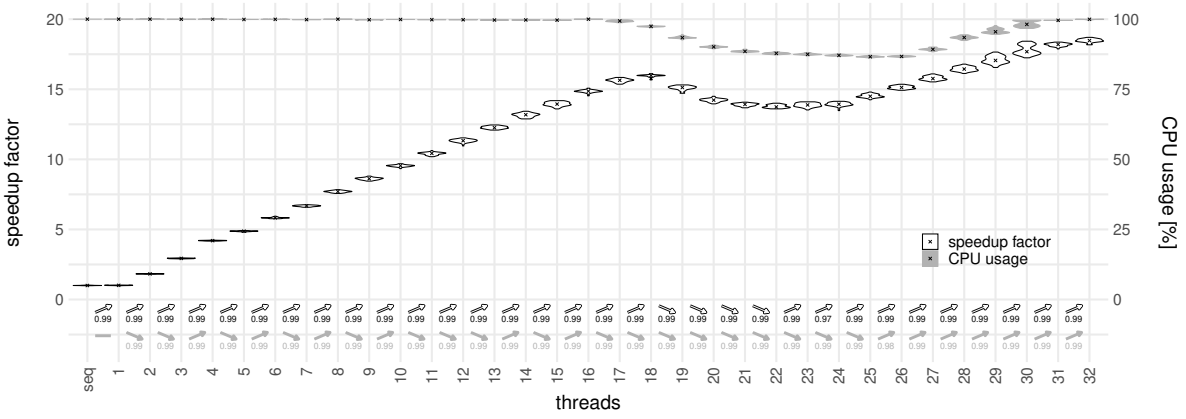
**a) Few work chunks** The simplest such case can be seen in Figure 5a for `(tiling, ICC, float, interleaved, balanced)`. We again plot two series of violins, but instead of showing data for the first and second function call, this time we plot two different metrics, both for the second call. In particular, we show the obtained speedup and the CPU usage. The speedup curve shows clear steps, it stays constant for some thread count increases and only improves at certain values. This is caused by the limited amount of work chunks provided. The loop we parallelize with OpenMP only has $D_N/BS_i^{float} = 4096/32 = 128$ iterations, which is small compared to the number of threads we use. Thus, the work cannot be divided equally among all threads and we see a performance increase only when the maximum number of loop iterations assigned to a thread decreases. For example, for 16,17 and 18 threads the worst-case thread will always have 8 loop iterations assigned. Only for 19 threads all workers have to handle 7 or less loop iterations and we see a performance increase. On the other hand, note that this is not harmful if the chosen number of threads divides the amount of work chunks well, here the performance is not influenced. The curve shows a slowdown from 24 to 25 threads due to a change in clock frequency, which we detail later.

**b) Global chip imbalance** In the second case, computing resource belong to two higher level groups, the two processor chips, which contain resources shared only between cores inside these groups. Figure 5b visualizes the results for `(simple, GCC, float, interleaved, spread)`. The used spread affinity policy of `gomp` does not take care well of this grouping as described in Section 3.4.3 and shown in the example case of Figure 3b. Recall that when increasing the thread count further than 16, `gomp's` implementation of spread binds all new threads on node 1 until all cores of this node are used for a thread count of 24 (16 threads on cores of node 1 vs. 8 threads on node 0). In this experiment, the L3 access bandwidth of a chip gets saturated when more than 10 cores are active. We scrutinize this further in Section 4.2.3. As each thread and with this core gets the same amount of work assigned, from 16 threads on a larger fraction of the overall task is processed on node 1. However, as soon as the L3 bandwidth bound is reached, the chip's overall throughput does not increase anymore (or just very slow). The execution time consequently increases with more work assigned to that chip. From 24 threads on, parts of the work start to be moved back to node 0 and we see the speedups recover in the plot. Even though such a situation can be mitigated by a careful affinity mapping, it can easily occur if the implementation is not done by an expert and uses the affinity policy provided by OpenMP.

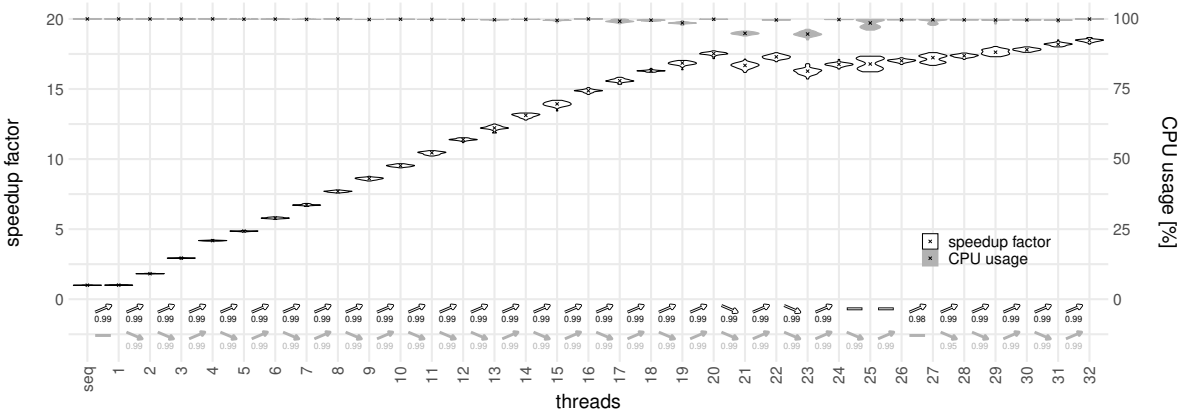
**c) Local chip imbalance** In Figure 5c we do exactly this change and switch to our balanced affinity policy, i.e. the global imbalance is resolved but there is still a difference of one used core for odd thread counts between the CPU chips. This results in slowdowns when changing from an even thread count to an odd one when using 20 or more threads, i.e. having reached L3 bandwidth saturation. Here again, more work is shifted to one chip while its throughput is close to saturation.




(a) small number of work chunks
`(tiling, ICC, float, interleaved, balanced)`

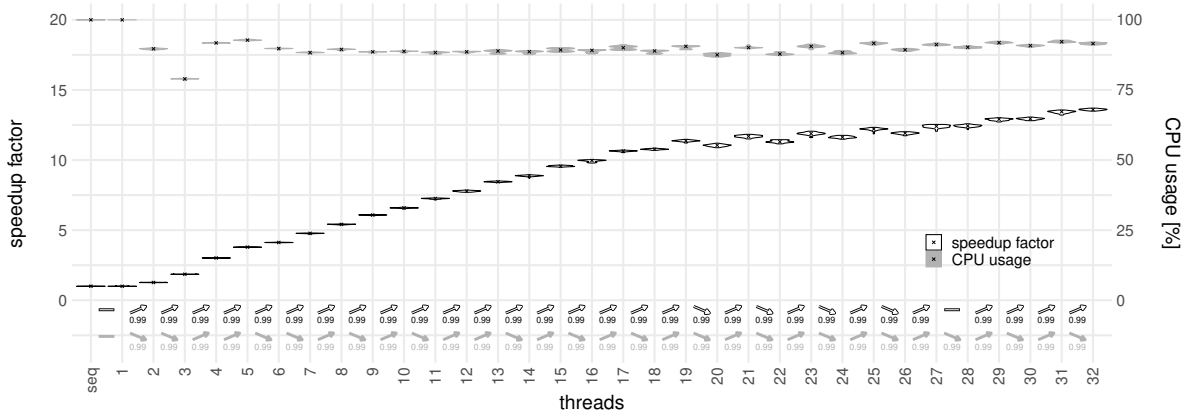



(b) chip imbalance (global)
`(simple, GCC, float, interleaved, spread)`




(c) chip imbalance (local)
`(simple, GCC, float, interleaved, balanced)`

Figure 5: Unbalanced work distribution




(d) NUMA access time
`(simple, GCC, float, bind, balanced)`

Figure 5: Unbalanced work distribution

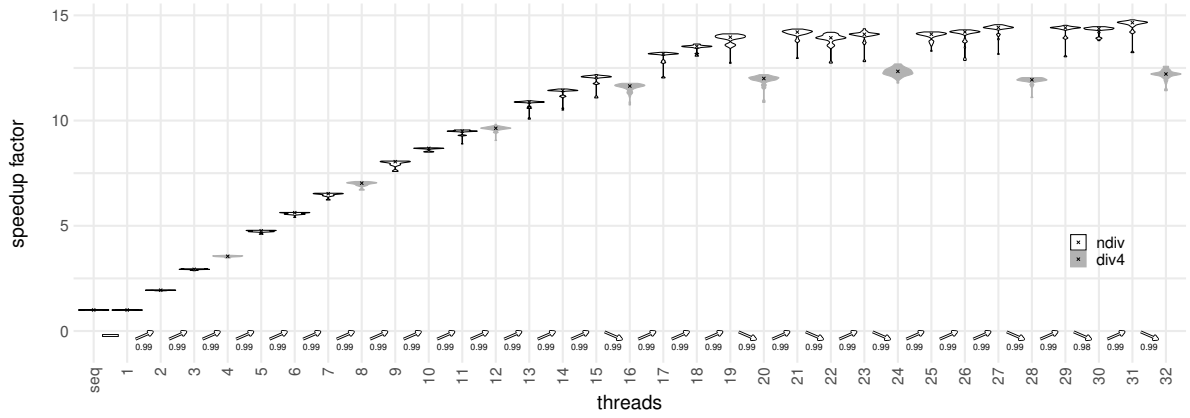
**d) NUMA access time** The last case of Figure 5d goes further by allocating memory only on NUMA node 0. Now, the two chips in addition have different properties in form of unequal memory access times. Because cores on node 1 have the slower access to remote NUMA memory locations compared to the local accesses of cores on node 0, threads located on node 1 need longer to finish. Therefore, when using two threads only, i.e. one thread on each node, the thread located on node 0 is idle for some time in the end and the CPU usage is rather low. For a thread count of three, the two threads located on node 0 finish early, waiting for the last thread on node 1. Hence, in this case two out of three threads need to wait and the CPU usage drops even further.

When we reach L3 bandwidth saturation (20 or more threads), we observe that in this situation, opposed to the case of c), even thread counts generally perform worse than odd thread counts. For equal work distributions between the nodes, the remote threads on node 1 are slower and dominate the runtime. If now some of the work is shifted to node 0, the runtimes of both nodes get better equilibrated. Cores on node 1 have to wait for slow remote memory accesses, whereas cores on node 0 have to stall longer for L3 accesses as more cores compete for the bandwidth on this chip. In summary, in this trade-off between increased L3 stress on node 0 vs. longer NUMA access times of node 1, an intentional imbalance of used cores between the nodes leads to a better execution time balance. This might not be chosen by a programmer's intuition. Of course, this issue only arises because of the used memory allocation scheme. However, as explained in Section 3.4.4, this might be a common situation since often memory is allocated by a single core in a sequential code fraction. With the default allocation policy it is consequently located on a single node - at least until the kernel potentially decides to move it.

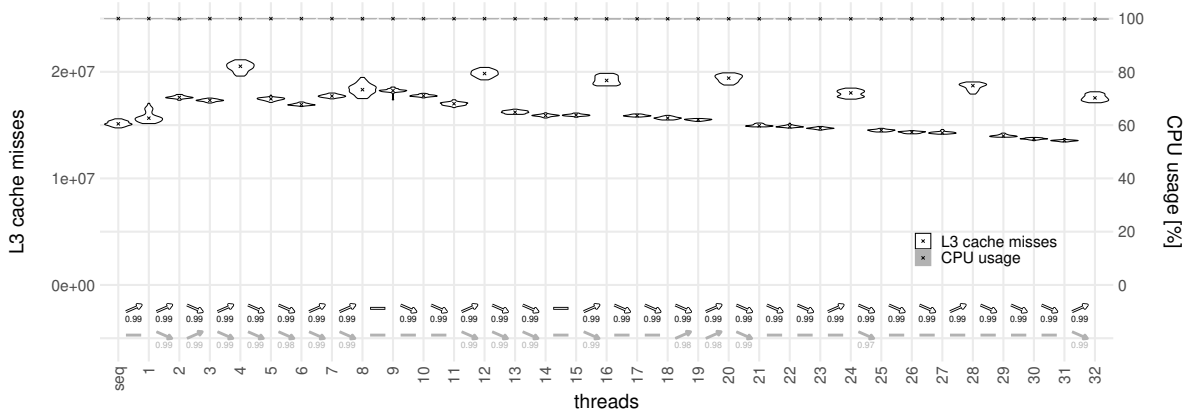
Please note that all these problems can be seen in the CPU usage metric, as threads are idle when having no more work to do. A good (static) work distribution scheme can solve them, or dynamic load balancing techniques can be used. The latter might however introduce additional runtime variations.

### 4.2.2 Thread count as implicit input

In Figure 6a we plot the speedups for `(MKL, ICC, float, bind, spread)`. We draw thread counts divisible by four in gray to highlight that they behave differently than the remaining ones. Let us call those two cases `div4` and `ndiv` from now on. In particular, we see that the application shows significantly worse performance for `div4` than for `ndiv`. Note that in Figure 6b the second series of violins tells us that the CPU usage is always high, also in those slower performing cases.




(a) speedup




(b) L3 cache misses and CPU usage

Figure 6: Thread count as implicit input - `(MKL, ICC, float, bind, spread)`

Our reverse engineering of MKL showed what causes the effect: MKL uses tiling with different block sizes depending on the thread count, with two different cases. Those blocks are then also mapped to threads in different ways. As a result, we can distinguish two different calculation schemes with different properties, one for `div4` and one for `ndiv`. In Figure 6b we plot the sum of L3 misses over all cores measured through the `LONGEST_LAT_CACHE.MISS` hardware performance counter [46]. It clearly shows that `ndiv` has a better cache re-use, its number of L3 misses is significantly lower than for the scheme of `div4`. We remark that this is also caused by the Dynamic Re-Reference Interval Prediction (DRRIP) cache replacement policy used by Skylake [47, 48] instead of classical Least Recently Used (LRU). Otherwise, both calculation schemes, the one for `div4` and the one for `ndiv`, could not keep any parts of the working set in the caches. The overall curve of L3 misses is overlaid by a general decreasing tendency caused by more total L2 cache available with an increasing number of used cores and thus reduced stress on the (non-inclusive) L3 cache.

Since each L3 miss causes a memory access, the scheme of `ndiv` with less L3 misses performs better in the memory bandwidth limited situation. When enough memory bandwidth is available, however, the tiling scheme chosen by MKL for `div4` is beneficial. MKL can completely hide the time needed for the additional memory accesses through extensive prefetching (through software instructions) and does not suffer anymore from the slight decrease in cache re-use. Intel's library thus makes a bad choice because we limited the memory bandwidth through memory allocation on a single node only.

In summary, we can say that MKL implicitly uses the thread count as an input parameter and the calculation scheme is different for two cases, `div4` and `ndiv`, leading to slower execution when increasing the thread count e.g. from 27 to 28 threads.

### 4.2.3 Shared bandwidth resource saturation

The Intel Skylake and other multicore architectures include many bandwidth resources shared among the CPU cores, for example:

- memory access bandwidth
- UPI links for inter-socket communication
- access bandwidth of the shared L3 cache
- bandwidth of the chip's mesh interconnect network

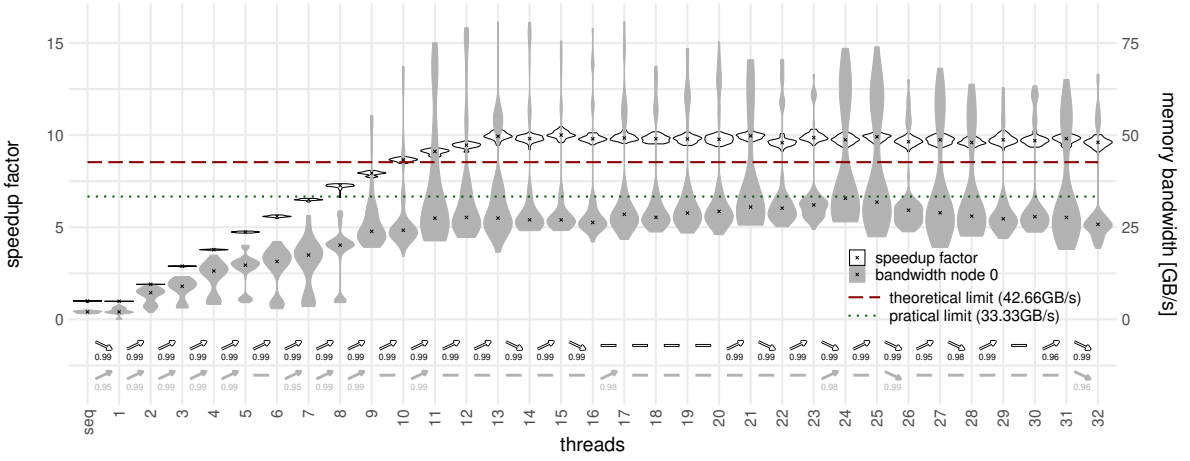
We analyze two of those which show to limit performance scalability in our experiments: shared memory access bandwidth and shared L3 bandwidth.

**a) Shared memory access bandwidth** As our machine is a shared memory architecture, all CPU cores access the same physical main memory connected to the memory controllers of the individual NUMA nodes. The available bandwidth of their busses is thus shared among all cores of the system. Figure 7 shows data for `(MKL, ICC, double, bind, balanced)`. We bind memory allocation to node 0 only, thus just the memory bandwidth of the IMCs of a single NUMA node of our machine is used, i.e. half of the theoretical available peak memory transfer bandwidth calculated in Section 3.2. The theoretical available memory bandwidth in this experiment is thus 42.66 GB/s. The practically achievable access throughput is of course lower due to interleaving of read and write accesses, DRAM refresh cycles and other problems. A Skylake system with two CPU chips simliar to ours is evaluated in [12], however, using all memory channels of the processors (each running at a speed of 2666 MT/s[2] as in our system). The paper reports a maximum measured memory bandwidth of around 100 GB/s when using a single chip of their machine, i.e. for a theoretical transfer rate of 128 GB/s. Assuming linear scaling with the number of active memory channels, in practice we can thus expect around 33.33 GB/s of maximum memory throughput in our experiment.

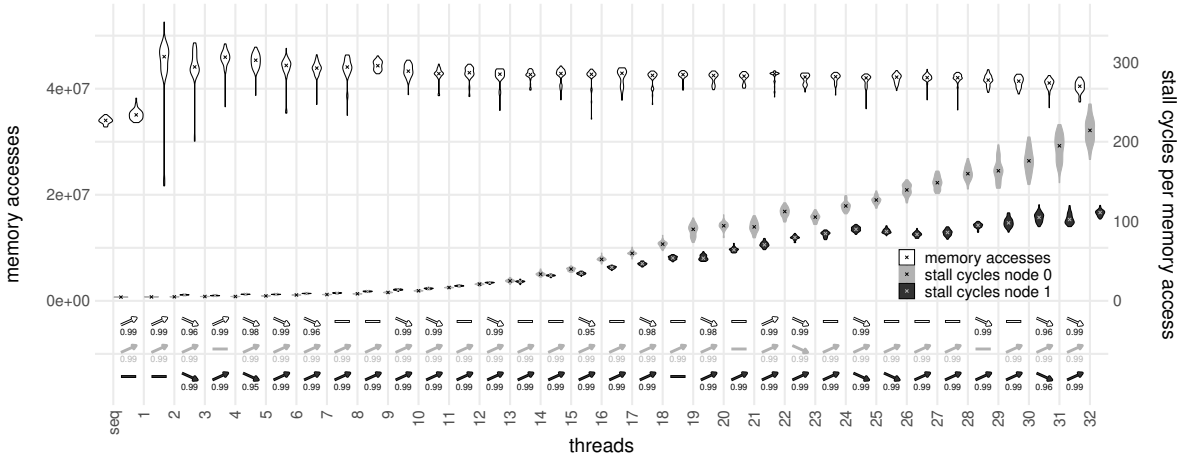
In Figure 7a we plot the achieved speedups together with the memory bandwidth used on NUMA node 0. The figure also includes the theoretical (red dashed) and practical (green dotted) bandwidth limits as calculated before. We observe that the curve of speedups sharply stops increasing at 13 threads and then stays almost constant. It also shows that the used memory bandwidth of node 0 gets close to its practical limit at the same moment. We obtain this bandwidth by starting measurements (approximately) 10 ms after the start of the matrix multiplication repeatedly for intervals of 10 ms and keeping the maximum value of those for each execution. This allows to obtain the peak bandwidth and also avoids including start and end phases of the computation in the bandwidth value which is averaged over the whole measurement duration. The large variation of the data is explained by the measurement interval still falling into different parts of the executions. In particular, for thread counts up to 8, we see that in some cases we still measured the start phase with much lower memory intensity. On the other hand, some executions for bigger thread counts even report bandwidths higher than the theoretical maximum. This might be caused by data arriving from buffers inside the chip, e.g. internal to the IMCs or located in the NoC, or inaccurate reporting of Intel's MBM hardware as we measure for very short time intervals only.

To further investigate the consequences of the memory bandwidth saturation, in Figure 7b we draw the number of memory accesses, gathered through the number of L3 misses in total (sum over all

[2] $T$ denotes data *transfers* (see also Footnote 1, page 14)




(a) speedup and used bandwidth




(b) memory accesses and stall cycles per access

Figure 7: Shared memory bandwidth saturation - `(MKL, ICC, double, bind, balanced)`

cores). This is again obtained through the `LONGEST_LAT_CACHE.MISS` hardware performance counter. The memory accesses show a slowly decreasing behavior from 3 threads on. As in Section 4.2.2, this is caused by adding L2 cache capacity such that more data can remain in the overall cache hierarchy. Consequently, less misses in the last level cache (L3) occur. In the beginning of the curve, other effects not important for our observation dominate, e.g. starting to use both NUMA nodes. The important fact is that the number of memory accesses stays constant or even decreases.

The second and third curve in Figure 7b shows for how many cycles cores have to stall for each memory access (in average) on the two NUMA nodes. We get this metric through the quotient of the performance counters `CYCLE_ACTIVITY.STALLS_MEM_ANY` and `LONGEST_LAT_CACHE.MISS`. Note that we intentionally use the counter for any memory access instead of `STALLS_L3_MISS`. Latter does not count stall cycles caused by instructions accessing memory addresses for which a (software) prefetch was already issued but which did not yet actually finish fetching the data. A possible pipeline stall however is always caused by the actual memory load instruction and never by the prefetch instruction. As a prefetch is already in flight, the access of the load instruction is counted as a hit in L3 or even in a higher cache level. Thus, the cycles are not attributed to the L3-miss stall counter, even though the

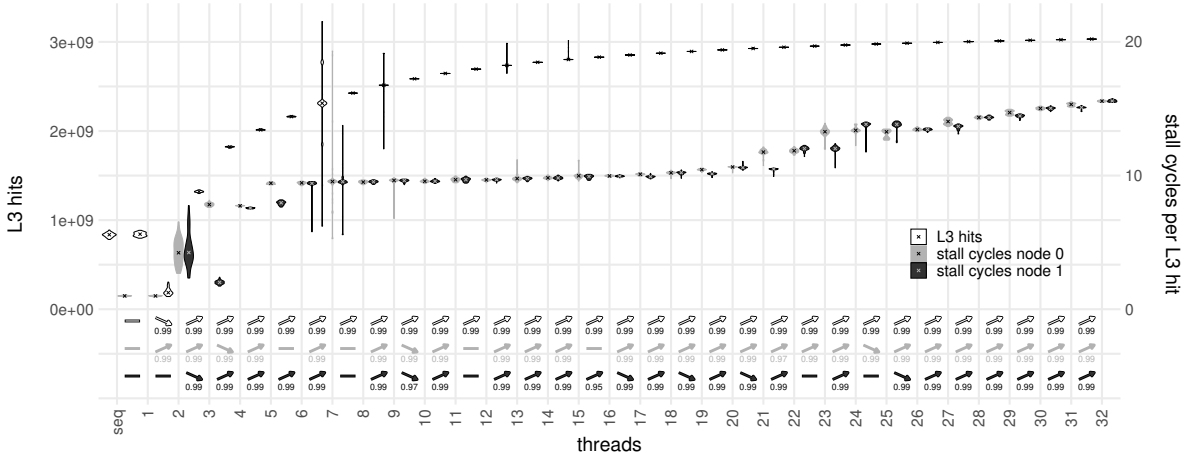



Figure 8: Shared L3 bandwidth saturation - `(simple, GCC, float, interleaved, balanced)`

stall indeed waits for data from main memory. MKL extensively uses prefetching through software instructions - such that this is a commonly occurring case.

In addition to our reverse-engineering, the resulting curve of stall cycles per memory access (Figure 7b) makes this advanced data prefetching of MKL obvious: For low thread counts, MKL's memory accesses incur almost no stall cycles. The usually high memory latency is almost completely hidden because the data is already transferred to one of the caches through prefetching when needed in a computation. When increasing the number of threads, however, Figure 7a shows us that the sum of the shared bandwidth reaches its limit. Now, each individual core gets less and less bandwidth to use but still needs to fetch the same amount of data per computation. As a logical consequence, the data fetch needs longer and does not arrive in time anymore: the cores need to stall. The more threads we add, the worse this situation gets and the stall cycles for memory accesses of all cores increase. Adding new threads thus does not add performance anymore, as the workload and available bandwidth are both shared equally between all used cores.

With the used memory binding, node 0 only experiences local and node 1 only does remote accesses. Interestingly, the stall cycles increase faster for node 0 than for node 1, i.e. remote accesses experience lower delays than local ones during bandwidth saturation. For low thread counts on the other hand, we see remote accesses causing more stalls than local ones as expected. This hints that Intel prioritizes remote NUMA accesses in the memory controllers.

**b) Shared L3 bandwidth** In Section 4.2.1 we claim that the experiments in Figures 5b to 5d reach L3 bandwidth saturation. We want to explain this in detail now, without work imbalance considerations. Let us thus focus on even thread counts of Figure 5c. The same analysis applies for odd thread counts and the other figures. Analogous to the memory bandwidth saturation case where the bandwidth to access main memory is shared among cores, here the bandwidth to access the L3 cache is shared. However, this time each chip has its individual L3 cache, so the bandwidth resource is only shared between all cores of one chip. We only care about accesses that are actually served with success by the L3, i.e. L3 hits. The first series of violins of Figure 8 visualizes those for our experiment. We get the metric by subtracting L3 misses (`LONGEST_LAT_CACHE.MISS`) from all L3 references which include L2 misses as well as prefetches (`LONGEST_LAT_CACHE.REFERENCES`). Even though we see in the plot that the L3 hits increase with the thread count, for higher thread counts this increase is small.

We now want to study the penalty in time caused by each of the L3 hits, in other words we are interested in how many clock cycles are needed for an individual data access that can be served by the L3. Be aware that we are again talking about stall cycles and not the actual time to access the L3

cache. Out-of-order execution and other features can hide a part of the access time which is around 50-70 cycles for the L3 of Skylake [32] when the cache does not have to serve other requests in parallel. However, only when the core has to stall, the cycles are actually penalty time and not spent for useful work. We thus compute the stall cycles incurred by each L3 hit (in average) as:

$$\text{stall cycles per L3 hit} = \frac{\text{total L3 hit stall cycles}}{\text{L3 hits}} = \frac{\text{stall cycles L2 miss} - \text{stalls cycles L3 miss}}{\text{L3 references} - \text{L3 misses}}$$
$$= \frac{\texttt{CYCLE\_ACTIVITY.STALLS\_L2\_MISS} - \texttt{CYCLE\_ACTIVITY.STALLS\_L3\_MISS}}{\texttt{LONGEST\_LAT\_CACHE.REFERENCES} - \texttt{LONGEST\_LAT\_CACHE.MISS}}$$

Figure 8 also contains two series of violins showing this metric individually for the two chips. From 3 to 10 used cores of a chip we can see an almost constant penalty of around 10 stall cycles for each data access that hits in L3. Then, a sudden increase can be seen which corresponds to the drop at 21 threads in the speedup curve (Figure 5c), as the performance of all cores of node 0 decreases at this point. The stall cycles continue to increase further with additional added active cores - the bandwidth for L3 cache accesses reached its limit. We remark that these cycles also includes the latency of the chip's mesh interconnect, i.e. the parallel accesses might saturate the interconnection network and not the actual L3 cache logic. However, in both cases, for us this is the maximal bandwidth available for accesses to the L3 cache.

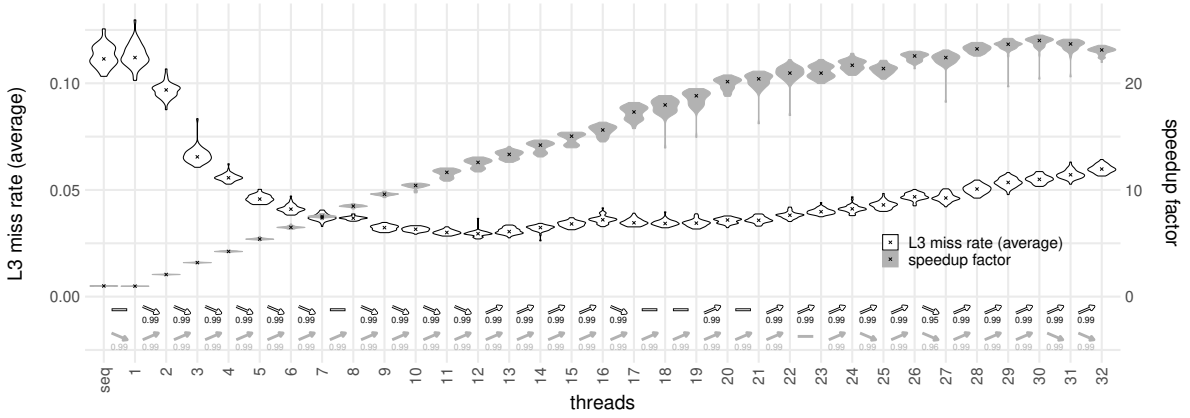
In summary we see that saturation of a shared bandwidth resource can limit scalability to a strict boundary in our two example cases of the main memory access bandwidth and the shared L3 cache access bandwidth.

#### 4.2.4 Shared storage resource conflicts

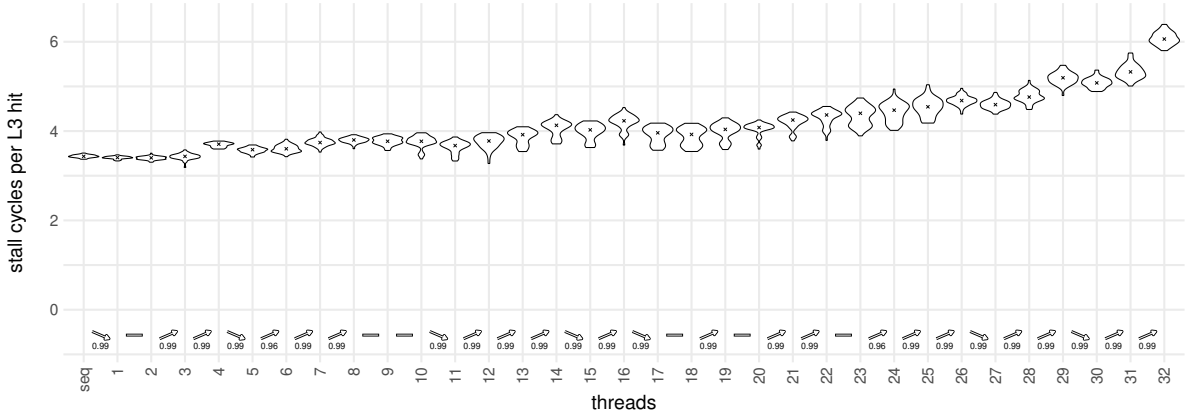
Storage resources are all resources that can maintain state information over time and thus keep data. This is a clear contrast to the bandwidth resources of the previous section. In this work, we are mainly interested in on-chip storage resources since they likely introduce stronger interferences. The amount of storage is limited and in case of a resource shared between multiple cores they compete for the available capacity. Storing information of one core might thus need to evict data of another core. In general, a storage resource can be any component of the chip which keeps information over multiple CPU cycles, including load or store buffers, Translation Lookaside Buffers (TLBs) or even the state of a branch predictor. However, here we only consider the most relevant case: caches, in particular the shared L3 of our system (see Figure 2).

Figure 9 shows the cache behavior for `(simple, ICC, float, interleaved, balanced)`. The average L3 miss rate of both nodes is plotted in Figure 9a, obtained by the quotient between the `LONGEST_LAT_CACHE.MISS` and the `LONGEST_LAT_CACHE.REFERENCES` counters. The same figure also includes a series of violins for the achieved speedups. The L3 miss rate first decreases until 12 threads. This is explained by the fact that using more cores also adds more cache in form of the private L2 caches of the additional cores, which cannot be used otherwise. Remember that for Skylake the L3 is non-inclusive and the L2 is quite large (1 MiB) compared to previous processor generations (Table 1). Hence, the total available cache capacity increases significantly and larger parts of the working set can stay in the caches, such that the L3 miss rate decreases. This effect of additional cache capacity is especially relevant for small thread counts, as the relative increase is larger. Moreover, cores might access the same data and thus constructively share the cache.

For higher thread counts on the other hand, the cores start competing for the L3 cache. We observe increases in the miss rate (Figure 9a) from 13 to 16 threads and again from 21 threads on. Latter is especially interesting: node 0 has 11 used cores at this count and Skylake's L3 cache is 11-way set associative. Consequently, for more than 11 used cores per chip, not all cores can place a (distinct) cache line in the same cache set at the same time. We can thus suspect that there is high interference on




(a) L3 miss rate and speedup




(b) L3 hit latency

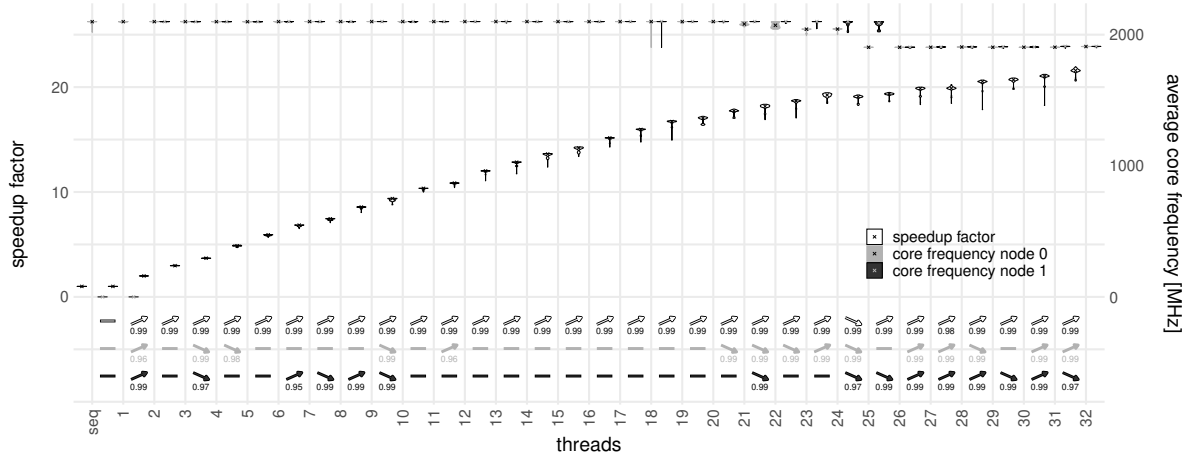
Figure 9: Storage resource conflicts - `(simple, ICC, float, interleaved, balanced)`

some of the cache sets, even though the overall L3 capacity might not be fully used. In other words, we likely observe many *conflict misses* caused by multiple cores sharing the common cache. Competition for the overall capacity of the L3 cache might also occur causing *capacity misses*, maybe even more common. In particular, cases where the threads work on distinct working sets, i.e. not like in our case on a fixed common dataset that is divided among the threads, might experience such conflicts.

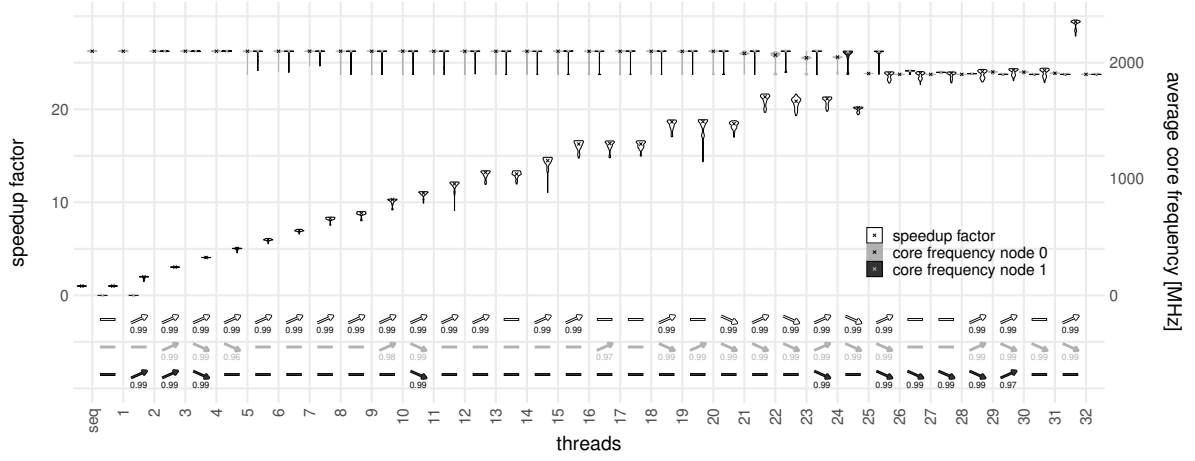
Figure 9b shows us the stall cycles imposed by each memory access that hits in L3, like we define it in Section 4.2.3. The metric increases, i.e. the application also reaches L3 access bandwidth saturation in addition to the storage resource conflict. Both those effects together cause the speedup curve not only to flatten but even to decrease for 31 and 32 used threads.

### 4.2.5 Shared power and temperature budget

In the previous sections, we see in Figure 5a a performance decrease for 25 threads compared to 24 threads and in Figure 4a the scaling curve starts to deviate from Amdahl's law from the same thread count onwards. Figure 10 shows configurations similar to those two. We show the data for Clang here to provide more diverse data, but the same phenomenon occurs for the other compilers (GCC and ICC). This time, we plot the obtained speedup curve together with the clock frequencies at which the cores run, in average over the whole experiment execution time and over all cores of a NUMA node.




(a) *MKL*




(b) *tiling*

Figure 10: Shared power/temperature budget - `(*, Clang, float, interleaved, balanced)`

We observe that the frequency is stable at 2.1 GHz for low thread counts. It drops to 1.9 GHz when a node reaches 13 used cores, i.e. at 25 threads for node 0 and at 26 threads for node 1. Note that for node 0 there is an intermediate region and the decrease gradually starts at 11 used cores. Obviously, this frequency decrease directly causes the performance per core to decrease and thus the slowdown followed by a lowered slope for MKL, as well as the slowdown for the *tiling* implementation. Figure 10b shows a few outliers between 5 and 25 threads, Figure 10a only for 18 threads.

Recall from Section 3.2 that we disabled Turbo Boost and P-states for our experiments. So why do we still observe different clock frequencies in Figure 10? Without Turbo Boost, the cores run at the *base* frequency. Our code makes heavy use of AVX-512 instructions, so it triggers Intel's license level 2, meaning that very power hungry instruction are executed. However, also in this mode, the base frequency of license level 0 of 2.1 GHz (compare Table 2) is used. It is higher than the guaranteed base frequency of AVX-512 of 1.3 GHz. Limits thus apply and in the table of allowed frequencies we see that for more than 12 active cores the maximum frequency is 1.9 GHz, independent of any actual conditions. This is lower than the used base frequency and corresponds exactly to the observed frequency steps.

We can interpret this as Turbo Boost being not completely disabled for license levels other than 0, or as vector instructions causing downclocking.

The frequency of node 0 starts decreasing already at 11 used cores because the OS is scheduling other processes than our test application, which use one or also easily two additional cores. Those are remaining kernel processes which run even though we reduced the amount of background services (Section 3.3). They do not consume a significant amount of CPU time but are enough to trigger sleeping cores to wake-up to the active state which then causes the interference with our application. If we are close to the boundary of 13 active cores, all our cores have to reduce their frequency during those times in which the additional cores are active and in average over the whole experiment execution time our cores use an intermediate frequency. Remember that the license level on these other cores does not matter for the frequency selection of the cores running our experiment. Similarly, the frequency outliers are explained by other processes interfering and keeping additional cores active, during the full time of some experiment executions.

We note that C-states here actually have an influence on the performance, though not directly since the cores used by our application do not enter in any sleep state during our experiments. Instead, activated C-states allow unused cores to enter a sleep state (deeper than C1) and use the freed power budget for the active cores, i.e. those can run at the higher frequency of 2.1 GHz in some cases. If we would have disabled C-states, all cores would always be considered active and the frequency limited to 1.9 GHz in every case.

If we had enabled Turbo Boost in our experiments, even more frequency steps would be present in a huge range from 3.5 GHz to 1.9 GHz. Many more slowdowns or at least worse scaling behaviors could be seen.

As with the end of Dennard scaling the power wall gets more and more dominant in chip design [49], Intel extends the DVFS features in each chip generation. Power and thermal management thus have an increasing impact on performance, which can be expected even stronger in future generations. In multicore processors, the individual cores are not independent but coupled through the chip's total budget. Cores compete for this resource - a high consumption on one core introduces interference on other cores. We can consequently see the *power and temperature budget of the chip as a shared resource*.

Therefore, we extend the classification of shared resources into bandwidth and storage resources of [1] with shared power and temperature resources. We treat temperature and power together as they have similar characteristics and are closely related. Heat dissipation is in the end caused by power consumption and thus the integration over time of it, compensated by a cooling system. Frequency and voltage scaling are only the means to control the usage of this shared resource.

Such a coupling also exists in classical multi-processor systems, e.g. through the case temperature or the maximum current the power supply can deliver to all chips. However, here the interference is many orders of magnitude lower and, with proper system dimensioning, usually not influencing the performance. It is the close integration of many computing elements (on a single chip) which causes these effects to matter.

## 4.3 Summary: Reasons for performance decreases

The previous sections show that many effects can limit performance scaling, i.e. the speedup curve stops increasing. In some cases, we can even see performance to decrease while increasing the thread count. This observation is counterintuitive and most scalability models cannot predict slowdowns, or only very limited (e.g. USL [6]) and not sufficiently for the cases presented in this work.

In all cases presented here, slowdowns are caused by a combination of two or more effects. Not just the strongest effect limits the scalability but they add up. The first effect saturates the speedups, then another effect decreases performance per core and thus causes the overall performance to decrease:

- Figures 4 and 10a: Amdahl bound, then the core frequency decreases (shared power/temperature budget) from 24 to 25 threads.

- Figures 5a and 10b: Bad work distribution scheme which cannot make use of additional cores, then the core frequency decreases from 24 to 25 threads.
- Figure 5b: Shared L3 access bandwidth (per chip) saturated, then adding imbalance in the amount of used cores between the chips for more than 16 threads. This also means assigning a larger fraction of the overall work to one chip compared to the other. As a consequence, more L3 accesses are needed on that node which reduces the overall performance as the L3 bandwidth is already saturated. Work gets re-balanced from 24 threads on.
- Figure 5c: As in Figure 5b, L3 bandwidth saturated and imbalance in the amount of work assigned to the nodes. However, the imbalance is only local for odd thread counts, thus slowdowns when changing from even to odd counts (20 $\rightarrow$ 21 and 22 $\rightarrow$ 23 threads).
- Figure 5d: As in Figure 5c, but even thread counts perform worse because of unequal memory access times of the nodes. Node 0 has faster memory access times which compensates a higher L3 access delay, thus assigning a slightly larger fraction of work to that node performs better overall. Slowdowns thus occur when going from odd to even thread counts (19 $\rightarrow$ 20, 21 $\rightarrow$ 22, 23 $\rightarrow$ 24 and 25 $\rightarrow$ 26 threads).
- Figure 6: Shared memory bandwidth saturated, then MKL chooses a calculation scheme needing more memory accesses for thread counts divisible by 4.
- Figure 9: L3 access bandwidth saturated, together with increasing competition for shared L3 cache ways which causes a higher cache miss rate for all cores.

In each of those, at least one effect involved is due to a shared resource. It occurs in the hardware micro-architecture and is completely hidden from the OS and therefore from the user. However, the importance to take this influence of the hardware architecture into account when analyzing performance scaling is apparent with our presented data. In the next section we will therefore present an attempt to formally model performance scalability when shared resources are present in the system.

# 5 Modeling scalability in the presence of shared resources

The previous chapter shows that Amdahl's law alone does not provide a good bound for the performance scalability of parallel applications. Since matrix multiplication is embarrassingly parallel, adding communication and synchronization as in the models of Section 2.2 does not help, they are also small and negligible. We thus develop a generalized model for performance scalability and then show how to apply it to our empirical data.

## 5.1 Generalized performance scalability model

The effects we want to model are due to sharing of resources between cores as illustrated in Figure 1f (page 5) and observed in Section 4.2. These directly influence the execution time of the parallel part of the program, usually degrading it from the perfect scaling with increasing thread count $p$ which Amdahl's law expects. On the other hand, the sequential part remains untouched by these effects as it only executes on a single core, i.e. all resources are exclusively available to one core as in a purely sequential program execution. Please note that there might still be a small influence of the parallel execution on the sequential part. One example is the clock frequency which might remain reduced for a short time after a parallel section since the additional cores used here, but not in the following sequential part, need a certain time to enter their sleep states. Thus, they can still lower the frequency allowed for the core used in the sequential part. We suspect those effects to be small and neglect them.

Consequently, we now assume that the parallel part scales with a more complex function $\alpha(p, \phi)$, where $\phi$ is a tuple of (at least) the executed program, its input, as well as the hardware architecture of the executing machine. The sequential fraction stays constant exactly as Amdahl also assumes it. As we want to model the scaling behavior of a fixed program with fixed input on a certain given machine, let us denote $\phi$ as a fixed parameter of the function from now on. We then obtain a simpler function $\alpha_\phi \colon \mathbb{N}^* \to \mathbb{R}$ and Equation (2) from page 7 gets:

$$T_{SharedRes}(p) = \sigma T_1 + \frac{(1-\sigma)T_1}{\alpha_\phi(p)} \tag{13}$$

We defined our empirical speedups in relation to a purely sequential program version, we have to do analogously here. Let $S_1 = \frac{T_{A_{seq}}}{T_1}$ denote the ratio between the runtimes of the single-threaded but parallelized version $T_1$ and the purely sequential version $T_{A_{seq}}$. The speedup formula of Equation (3) then becomes

$$\boxed{Speedup_{SharedRes}(p) = \frac{S_1}{\sigma + \frac{(1-\sigma)}{\alpha_\phi(p)}}.} \tag{14}$$

In the context of our experiments, we estimate that

$$\begin{aligned} S_1 &= aggr\{Speedup_{empirical}(1)\} \\ &= \frac{median\{t_{real}(A_{seq}, 1, i, s) \mid i \in I\}}{aggr\{t_{real}(A, 1, j, s) \mid j \in I\}} \end{aligned} \tag{15}$$

where *aggr* is an aggregation function which is chosen depending on what we want to predict, in our case either the *median* or the *maximum* of the speedup.

Amdahl's law is the special case where $\alpha_\phi(p) = p$, i.e. the parallel part scales linearly with increasing processor count. In the following section, we try to further quantify the arbitrary function $\alpha_\phi(p)$.

## 5.2 Modeling specific resource sharing effects

We now want to detail $\alpha_\phi(p)$ for two kinds of resource sharing effects which we observe in our experiments: a shared power/temperature budget (Section 4.2.5) limiting the clock frequency of cores and the

saturation of a shared bandwidth resource (Section 4.2.3). The modeling as presented in this section is valid for all cases dominated by solely those effects. We do not provide models for shared storage resources and effects not related to resources sharing (e.g. work distribution).

Throughout this section, we add the model names as superscripts to $\alpha_\phi$ to distinguish the different proposed scaling functions for the parallel program part: $\alpha_\phi^{freq}$ for the `freq` model of Section 5.2.1 and $\alpha_\phi^{bw\_freq}$ for the `bw_freq` model presented in Section 5.2.2.

### 5.2.1 Modeling a shared power/temperature budget

The cores in a multicore chip share a common limited power and temperature budget. As explained in Section 3.2.1, frequency scaling (DVFS) is used to prevent exceeding these limits even when all cores are active. Often, and in particular for our machine and experiments, static limits are sufficient and no dynamic throttling occurs. We can thus simply model the effect of frequency scaling instead of the power and temperature budget causing it.

A system might, as in the case of our test machine, be composed of multiple CPU chips. We define *numchips* as the count of those, and $D = \{0, 1, ..., numchips - 1\}$ as the set of all chips in the system. Each of the chips independently performs frequency scaling. We thus first describe the behavior of a single chip identified by $d \in D$, and combine the individual components afterwards to obtain the system level behavior.

Let $maxfreq\colon \mathbb{N}_0 \to \mathbb{R}$ denote the function that returns the maximum clock frequency of cores, depending on the number of active cores $c_{chip} \in \mathbb{N}_0$ in the overall chip. This function is the mathematical representation of the information shown in Table 2 (page 15) for our test machine. During the execution of an application, the number of active cores $c_{chip}$ is composed of the cores used by the application itself and additional cores kept in the active state by the OS or other processes running in the background. The number of cores used by the application $c_{app}(d, p) \in \mathbb{N}_0$ is different for each chip $d$, but otherwise only depends on the used thread count $p$. On the other hand, from our point of view on the execution, the amount of cores that are kept busy due to interfering processes is random. Furthermore, it varies during the time of the experiment $T_{app}$. Let $\Omega$ be the set of possible outcomes of the underlying random phenomenon. A discrete random variable $C_{interf}(d, t)\colon \Omega \to \mathbb{N}_0$ then describes the observed number of additional active cores at a specific time $t \in [0, T_{app}]$, on the chip $d$. Further, the collection of the random variables for all instances of the continuous time $t$ together forms a stochastic process $\{C_{interf}(d, t)\}_{t\in[0,T_{app}]}$.

In the modeling, we consider only the average influence of the stochastic process over the whole experiment execution time. We thus compute the time average of the varying clock frequency, i.e. the frequency of a constant clock which would show the same number of clock ticks when measuring over the whole execution time:

$$Freq_{tavg}(d, c_{app}) = \frac{1}{T_{app}} \int_0^{T_{app}} maxfreq(c_{app} + C_{interf}(d, t))\, dt \tag{16}$$

Be aware that $Freq_{tavg}$ is again a random variable. We only take the time average of a function applied to the stochastic process $C_{interf}$ for a single outcome, and not the average over different outcomes. In other words, each outcome of the stochastic process shows a different interference and thus we see a different frequency in average over the experiment time. We apply an aggregation function $aggr$ chosen depending on what we want to predict, e.g. taking the *median*, to get a single value:

$$\begin{aligned} freq_{chip}(d, c_{app}) &= aggr\{Freq_{tavg}(d, c_{app})\} \\ &= aggr\{\frac{1}{T_{app}} \int_0^{T_{app}} maxfreq(c_{app} + C_{interf}(d, t))\, dt\} \end{aligned} \tag{17}$$

We now need to combine the obtained frequencies of all chips in the system to get a system level modeling. The overall application runtime is determined by the last finishing thread. Recall that we

assume the workload to be balanced among the threads, as work imbalance issues have already been studied well and our focus in this work is to show that microarchitectural effects degrade scalability even for perfectly balanced applications. We can thus expect that a thread finishes last which is processed by a core located on the chip running at the lowest frequency. Therefore, we have to consider the minimum over the clock frequencies of all chips:

$$freq_{system}(p) = min\{freq_{chip}(d, c_{app}(d,p)) \mid d \in D\} \tag{18}$$

To get a scaling factor which we can use in the general model of Section 5.1, we relate this clock frequency to $freq_{system}(1)$, which is the clock frequency when our application uses a single core. It has to share the power and temperature budget only with cores which are active due to interfering processes. This leads:

$$\boxed{\alpha_{\phi}^{freq}(p) = p \times \frac{freq_{system}(p)}{freq_{system}(1)}} \tag{19}$$

Equation (19) and Equation (14) together form our model `freq`. To give a better understanding, we will now derive the particular functions for our test machine.

**Example of our test machine**

We estimate $freq_{chip}$ of Equation (17) with the data from our empirical measurements. We could measure samples of the unknown stochastic process $\{C_{interf}(d,t)\}_{t\in[0,T_{app}]}$ and then compute $Freq_{tavg}$ for each of those observations following Equation (16). However, we can also directly measure $Freq_{tavg}$. Therefore, we count the actual clock ticks of the cores, as well as the ticks of a fixed-frequency reference clock, during the whole experiment execution time. We can then easily compute the core's frequency through the ratio of the two counts, which corresponds to the ratio of their clock's frequencies. The resulting frequency is then already the average over the execution time, so one observation of the random variable $Freq_{tavg}$. Since we run many experiments with many repetitions, we get a large sample of observations. We take the median of all these and round to the nearest 10 MHz step to obtain an approximation of $freq_{chip}$:

$$freq_{chip}(0, c_{app}) = \begin{cases} 2100 & \text{if } c_{app} \leq 10 \\ 2070 & \text{if } c_{app} = 11 \\ 2040 & \text{if } c_{app} = 12 \\ 1900 & \text{if } c_{app} \geq 13 \end{cases} \qquad freq_{chip}(1, c_{app}) = \begin{cases} 2100 & \text{if } c_{app} \leq 12 \\ 1900 & \text{if } c_{app} \geq 13 \end{cases} \tag{20}$$

For the CPU chips of our machine, in the configuration used for our experiments, we have:

$$maxfreq(c_{chip}) = \begin{cases} 2100 & \text{if } c_{chip} \leq 12 \\ 1900 & \text{if } c_{chip} \geq 13 \end{cases} \tag{21}$$

We can see that $freq_{chip}(1, c_{app}) = maxfreq(c_{app})$. This means that $c_{chip} = c_{app}$ on chip 1, i.e. the interference is negligible and the frequency scaling is solely determined by the cores used by our application. However, on chip 0 we observe a very different, significant, interference component.

For a machine with two CPU chips, using our balanced affinity policy, we can explicitly write the number of cores kept active by the applications on each of the chips:

$$\begin{aligned} c_{app}(0,p) &= \lceil p/2 \rceil \\ c_{app}(1,p) &= \lfloor p/2 \rfloor \end{aligned} \tag{22}$$

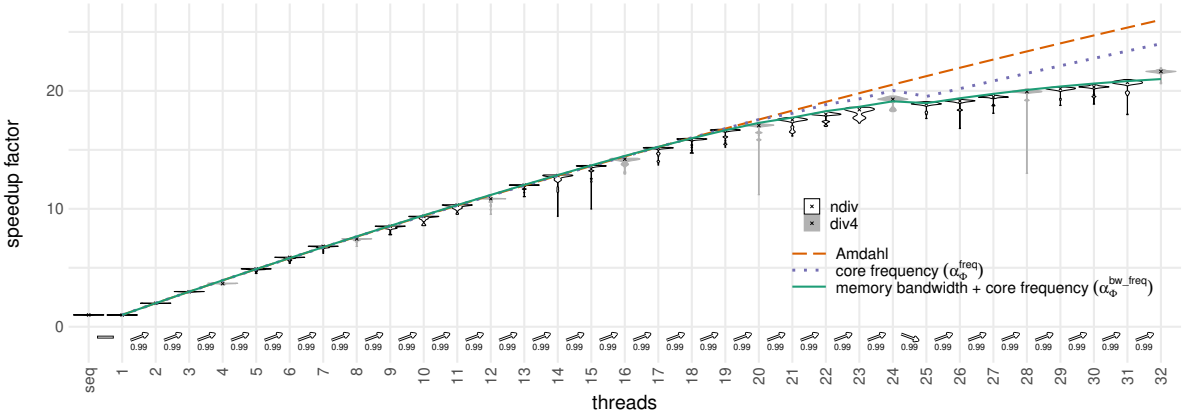



Figure 11: Scalability models - `(MKL, ICC, float, interleaved, balanced)`

For odd thread counts our applications use more cores on chip 0 than on chip 1, such that we get that $freq_{chip}(0, c_{app}(0,p)) \leq freq_{chip}(1, c_{app}(1,p)) \ \forall\, p \in \mathbb{N}_0$. The minimum over the chips is thus equal to the values of chip 0 and we obtain:

$$\begin{aligned} freq_{system}(p) &= min\{freq_{chip}(0, c_{app}(0,p)), freq_{chip}(1, c_{app}(1,p))\} \\ &= freq_{chip}(0, c_{app}(0,p)) \end{aligned} \tag{23}$$

Finally, we retrieve as scaling factor:

$$\begin{aligned} \alpha_\phi^{freq}(p) &= p \times \frac{freq_{chip}(0, \lceil p/2 \rceil)}{freq_{chip}(0,1)} \\ &= \frac{p}{2100} \times \begin{cases} 2100 & \text{if } p \leq 20 \\ 2070 & \text{if } 21 \leq p \leq 22 \\ 2040 & \text{if } 23 \leq p \leq 24 \\ 1900 & \text{if } p \geq 25 \end{cases} \end{aligned} \tag{24}$$

We fit our model to the case seen in Section 4.1 and Section 4.2.5 (Figure 4 and Figure 10a), that is the configuration `(MKL, *, float, interleaved, balanced)`. This results in the blue (dotted) bound in Figure 11, compared to the Amdahl bound plotted in dashed orange (both $\sigma = 0.0077$, fitted until 20 threads without `div4`). We note that the model's prediction follows well the slowdown from 24 to 25 threads. However, for thread counts higher than 20, the bound is still far from the actual measured speedups. A closer analysis of the application shows us why. First, we use MKL which behaves as in Section 4.2.2, that is it uses two different algorithms depending on the thread count, even though here we do not see it as clear in the speedup data as in the previous case. Second, both those cases get bandwidth bound by a shared resource. Thereby, MKL's algorithm for thread counts divisible by 4 (`div4`, grey violins in Figure 11) gets bound by the bandwidth of the UPI links whereas the other case (`ndiv`, black in the plot) starts to show congestion for accesses to the main memory.

### 5.2.2 Modeling shared bandwidth saturation

We now aim to describe bandwidth saturation. Therefore, we model the shared bandwidth resource as a queueing system, similar to Gunther [7]. We only model a single bandwidth resource but with a general model which can be applied to the various bandwidth resources found in a system. Each such

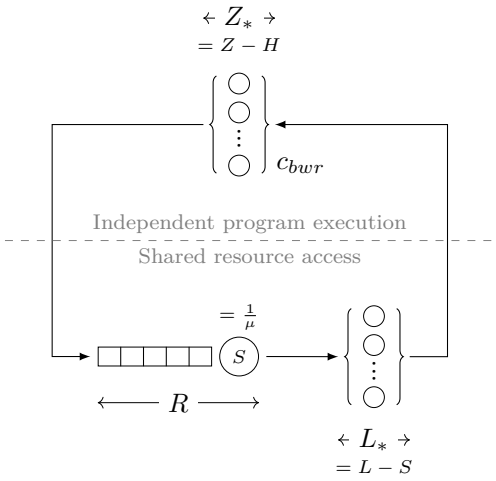



Figure 12: Scheme of our model for shared bandwidth resources

resource will impose its own bound and the overall system's performance is determined by the lowest of these, assuming that different bandwidth resources do not influence each others behavior. Figure 12 shows the scheme of our model. Circles represent possible states of the customers in the queueing system, whereby the machine's cores are the customers. States can be occupied by at maximum one core. Arrows show the possible transitions between these states. For the actual queue we use boxes instead of circles to depict the waiting states, with implicit transitions between adjacent states. Each core either executes independently program parts which do not use the shared resource (upper part of the scheme), or issues a request to the shared resource which means it has to enter the queue (lower part). A mapping function $c_{act\_bwr} \colon \mathbb{N}^* \to \mathbb{N}_0$ reflects how many of the $p$ threads execute on cores which access the limiting shared resource. Our model requires at least one core doing so, we thus further define $c_{bwr} \colon \mathbb{N}^* \to \mathbb{N}^*$ as $c_{bwr}(p) = max(1, c_{act\_bwr}(p))$ and use this function in our equations instead. We will see that for a single core no congestion occurs in our model (a single core will always obtain direct service in the queueing system), such that this is not an issue but allows to have a valid reference for a single thread in any case. All cores can be in the upper part of the scheme at the same time as there are $c_{bwr}$ parallel states. The resource offers a single server to process requests at a service rate of $\mu \in \mathbb{R}$, so in an average time of $S = \frac{1}{\mu} \in \mathbb{R}$. The total time spent at this stage, as a function of the number of threads $p$, is the residence time $R \colon \mathbb{N}^* \to \mathbb{R}$, including queueing time. The shared server component of our model is then followed by an additional delay part which multiple requests (at least $c_{bwr}$) can enter at the same time, to account for the overall access latency. Let $L \in \mathbb{R}$ be the total latency of a shared resource access without any congestion. The average time spent at this stage $L_* \in \mathbb{R}$ is then given as $L_* = L - S$. These two components allow to describe a pipelined resource where only one stage represents the actual bottleneck. Think for example of a main memory access. The access is composed of many steps causing the overall latency, but we can imagine that only the transmission on the DRAM bus limits, as it is the bandwidth of this bus for which cores actually compete.

When the access is completed, the core enters again the independent parallel execution region of our model. It takes a core an average time of $Z \colon \mathbb{N}^* \to \mathbb{R}$ to complete all instructions between two accesses to the shared resource. This time scales with the clock frequency of the cores. With the DVFS mechanisms modeled in the previous section, it therefore depends on the number of active cores. Let $Z_1$ denote this time when only a single core is active. We then have:

$$Z(p) = Z_1 \times \frac{freq_{system}(1)}{freq_{system}(p)} \tag{25}$$

Modern processor architectures, including the Skylake processor in our machine, use techniques like out-of-order execution and data prefetching. A core can thus issue a request to a shared resource, i.e. enter the queue, but still continue doing useful work. Let $H$ be a function which gives us the time for which the core still executes instructions after issuing the request. This time depends on the overall time for the shared resource access which is $T_b = R(p) + L_*$. Frequency scaling will also affect $H$, such that it further depends on $p$. The function is thus defined as $H\colon \mathbb{R} \times \mathbb{N}^* \to \mathbb{R}$. We develop the exact relation later. Already treated instructions do not need to be executed after the shared resource access anymore. The effective time spent in independent program execution, denoted by $Z_*$, is consequently reduced by the value of $H$:

$$Z_*(p) = Z(p) - H(R(p) + L_*, p) \tag{26}$$

The queueing system embedded in our model is $\mathrm{M/M/1}/c_{bwr}/c_{bwr}$ in Kendall's notation. Note that the queueing system's quantities introduced until now represent mean values of Poisson processes. The actual arrival rate at the queue is:

$$\lambda_q(p) = \frac{1}{Z_*(p) + L_*} = \frac{1}{Z(p) - H(R(p) + L_*, p) + L_*} \tag{27}$$

With the service rate $\mu$, we get as mean residence time in the queuing system at steady-state [50]:

$$R(p) = \frac{1}{\mu} \frac{c_{bwr}(p)}{(1 - B(\frac{\mu}{\lambda_q(p)}, c_{bwr}(p)))} - \frac{1}{\lambda_q(p)} \tag{28}$$

where $B\colon \mathbb{R} \times \mathbb{N}^* \to \mathbb{R}$ is the *Erlang B formula* defined as:

$$B(E, i) = \frac{\frac{E^i}{i!}}{\sum_{j=0}^{i} \frac{E^j}{j!}} \tag{29}$$

Note that replacing $\lambda_q$ with Equation (27) in Equation (28) leads an implicit equation for the residence time. We compute solutions numerically when fitting the model to our experimental data.

Let us now find a function for $H(T_b, p)$ which describes the behavior of out-of-order execution. We develop it step-by-step to allow the reader to understand the reasoning behind this modeling. We first establish the function $H_p(T_b) = H(T_b, p)$ for which $p$ is a constant parameter. When the time to access the shared resource $T_b$ is short, the core can likely hide the full latency, i.e. we want that the derivative converges to one for small $T_b$:

$$\lim_{T_b \to -\infty} H'_p(T_b) = 1 \tag{30}$$

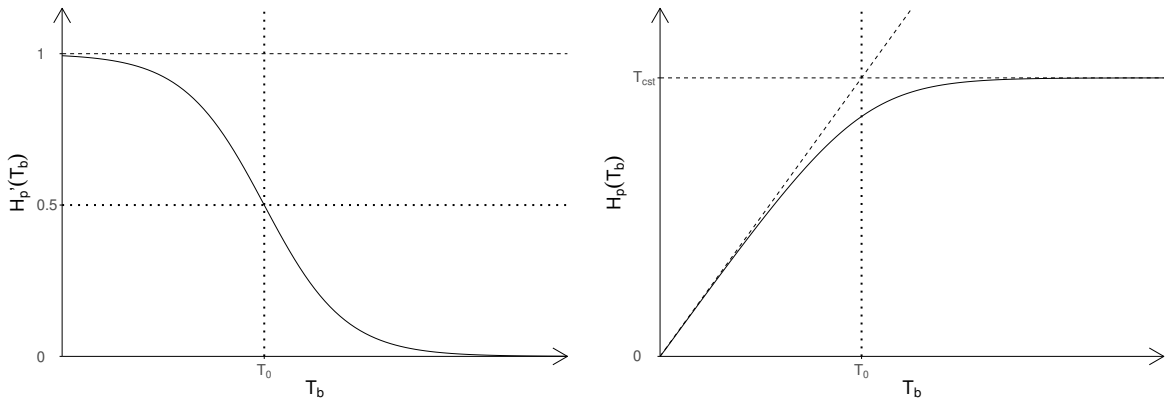
For larger $T_b$, however, at some point the core cannot find any instructions anymore which are ready to be executed and it has to stall. $H_p$ saturates at a maximum $H_{max}$, representing an asymptote to our function. We consequently want that:

$$\lim_{T_b \to \infty} H'_p(T_b) = 0 \tag{31}$$

These two constraints on $H'_p(T_b)$ characterize an inverted *sigmoid* curve. The simplest function with such a curve is the (inverted) *logistic function* which we will use in our model for $H'_p(T_b)$. We get:

$$H'_p(T_b) = \frac{1}{1 + e^{-k(T_0 - T_b)}} \tag{32}$$

This function and its parameters are illustrated in Figure 13a. The parameter $T_0 \in \mathbb{R}$ shifts the curve along $T_b$ and thus describes at which point saturation occurs. The second parameter, $k \in \mathbb{R}$,




(a) $H'_p(T_b)$ (b) $H_p(T_b)$

Figure 13: Illustration of the $H_p$ function

characterizes the steepness in the intermediate region between the two limit behaviors, i.e. how fast the curve changes from one limit to the other. We integrate $H'_p$ and get a function which is often called *softplus* function, again inverted and with integration constant $T_{cst}$:

$$H_p(T_b) = -\frac{1}{k} log(1 + e^{k(T_0 - T_b)}) + T_{cst} \tag{33}$$

Figure 13b shows this function, including its limit behaviors. From the constraint of the desired asymptotic behavior we obtain $T_{cst} = H_{max}$. We obviously need $H_p(0) = 0$. This leads:

$$T_0 = \frac{1}{k} log(e^{kH_{max}} - 1) \tag{34}$$

Analogous to $Z$ in Equation (25), $H_{max}$ changes with the clock frequency of the cores:

$$H_{max}(p) = H_1 \times \frac{freq_{system}(1)}{freq_{system}(p)} \tag{35}$$

Our function for $H$ is thus given by:

$$H(T_b, p) = H_{max}(p) - \frac{1}{k} log(1 + e^{k(T_0(p) - T_b)}) \tag{36}$$

Our model intuitively requires $Z_*(p) \geq 0$. This is equivalent to $H_{max}(p) \leq Z(p)$. From Equations (25) and (35), we see that our parameters thus need to satisfy $H_1 \leq Z_1$.

Let $W(p) \in \mathbb{R}$ denote the number of accesses to the shared resource which are required in total by our application. $T_q$ is the overall time of execution of the parallel program part. The throughput in our model, i.e. the amount of requests to the resource which are treated per time, is then $X_q(p) = \frac{W(p)}{T_q(p)}$. By applying Little's law to each station of our model, we get a relation between the throughput and the total number of customers in the system which is $c_{bwr}(p)$:

$$c_{bwr}(p) = X_q(p) \times R(p) + X_q(p) \times L_* + X_q(p) \times Z_*(p) \tag{37}$$

Which is equivalent to:

$$\begin{aligned} X_q(p) &= \frac{c_{bwr}(p)}{R(p) + L_* + Z_*(p)} = \frac{c_{bwr}(p)}{R(p) + L_* + Z(p) - H(R(p) + L_*, p)} \\ &= \frac{c_{bwr}(p)}{R(p) + \frac{1}{\lambda_q(p)}} = \mu(1 - B(\frac{\mu}{\lambda_q(p)}, c_{bwr}(p))) \end{aligned} \tag{38}$$

We keep the workload constant when scaling as we do not increase the size of our matrices. We assume the computations to be equally distributed among all threads. The total number of accesses $W(p)$ to the shared resource is then proportional to the fraction of threads accessing the resource:

$$W(p) = W_1 \times \frac{c_{bwr}(p)}{p} \tag{39}$$

The parallel speedup in queueing time (not in overall program time), i.e. the scaling factor $\alpha_\phi$ for the parallel part of our generalized model of Equation (14), equals to:

$$\alpha_\phi^{bw\_freq}(p) = \frac{T_q(1)}{T_q(p)} = \frac{W(1)}{W(p)} \times \frac{W(p)}{T_q(p)} \times \frac{T_q(1)}{W(1)} = \frac{p}{c_{bwr}(p)} \times \frac{X_q(p)}{X_q(1)} \tag{40a}$$

$$= \frac{p(R(1) + L_* + Z(1) - H(R(1) + L_*, 1))}{R(p) + L_* + Z(p) - H(R(p) + L_*, p)} \tag{40b}$$

$$= \frac{p}{c_{bwr}(p)}(1 + \frac{\mu}{\lambda_q(1)})(1 - B(\frac{\mu}{\lambda_q(p)}, c_{bwr}(p))) \tag{40c}$$

Equations (14) and (40) together with the definitions of Equations (25), (27), (28), (35) and (36) form our model `bw_freq`. It has six parameters $\sigma$, $\mu$, $Z_1$, $L_*$, $H_1$ and $k$. In the rest of this section we show that the model incorporates the `freq` model and that its speedup is invariant to scaling of the absolute time.

**Absence of a limiting shared bandwidth resource**

From Equation (40b) we can easily see that `bw_freq` includes the `freq` model as a special case, namely when the shared bandwidth resource does not limit the system's performance. This case is approached when the time spent in independent program execution $Z$ is largely superior to the time spent for accesses to the shared resource $L$, i.e. if $Z \gg L = S + L_*$. A small service time $S$ and a large inter-arrival time $Z$ implies that the residence time $R(p)$ in the queueing system is also small (close to $S$). For small $T_b = R(p) + L_*$, by construction of the asymptotic behavior of the function $H$ we have:

$$H(R(p) + L_*, p) \approx R(p) + L_* \tag{41}$$

Substituting Equation (41) in Equation (40b) shows us that in this case $\alpha_\phi^{bw\_freq}$ approaches $\alpha_\phi^{freq}$:

$$\alpha_\phi^{bw\_freq}(p) \approx p \times \frac{Z(1)}{Z(p)} = p \times \frac{freq_{system}(p)}{freq_{system}(1)} = \alpha_\phi^{freq}(p) \tag{42}$$

**Invariance of the speedup to time scaling**

We now show that the parallel speedup in our model is invariant to scaling of the values of its parameters. Therefore, we scale all parameters for the parallel program section of our model, which are all either times or rates per time, by an arbitrary time scaling factor $\eta$:

$$Z_1^+ = \eta Z_1 \qquad \mu^+ = \frac{\mu}{\eta}$$
$$L_*^+ = \eta L_* \qquad k^+ = \frac{k}{\eta} \tag{43}$$
$$H_1^+ = \eta H_1$$

We thereby indicate with a "$^+$" the new scaled parameter values. We use the same notation in the following for all derived quantities. Substituting in Equation (25) leads:

$$Z(p) = \frac{Z_1^+}{\eta} \times \frac{freq_{system}(1)}{freq_{system}(p)} = \frac{Z^+(p)}{\eta} \tag{44}$$

Analogous, Equation (35) gives:

$$H_{max}(p) = \frac{H_1^+}{\eta} \times \frac{freq_{system}(1)}{freq_{system}(p)} = \frac{H_{max}^+(p)}{\eta} \tag{45}$$

From Equation (34) we get:

$$T_0(p) = \frac{1}{\eta k^+} log(e^{\eta k^+ \frac{H_{max}^+(p)}{\eta}} - 1) = \frac{T_0^+(p)}{\eta} \tag{46}$$

Equation (36) becomes:

$$H(T_b, p) = \frac{H_{max}^+(p)}{\eta} - \frac{1}{\eta k^+} log(1 + e^{\eta k^+ (\frac{T_0^+(p)}{\eta} - T_b)}) = \frac{H^+(\eta T_b, p)}{\eta} \tag{47}$$

From Equation (28), assuming $\lambda_q(p) = \eta \lambda_q^+(p)$ which we proof afterwards, we get:

$$R(p) = \frac{1}{\eta \mu^+} \frac{c_{bwr}(p)}{(1 - B(\frac{\eta \mu^+}{\eta \lambda_q^+(p)}, c_{bwr}(p)))} - \frac{1}{\eta \lambda_q^+(p)} = \frac{R^+(p)}{\eta} \tag{48}$$

Equation (27) together with the previous results confirms our assumption on $\lambda_q^+(p)$:

$$\lambda_q(p) = \frac{1}{\frac{Z^+(p)}{\eta} - \frac{H^+(\eta(\frac{R^+(p)}{\eta} + \frac{L_*^+}{\eta}), p)}{\eta} + \frac{L_*^+}{\eta}} = \eta \lambda_q^+(p) \tag{49}$$

Finally, in Equation (40c) we see that parallel scaling factor $\alpha_\phi^{bw\_freq}$ is invariant to $\eta$:

$$\alpha_\phi^{bw\_freq}(p) = \frac{p}{c_{bwr}(p)}(1 + \frac{\eta \mu^+}{\eta \lambda_q^+(1)})(1 - B(\frac{\eta \mu^+}{\eta \lambda_q^+(p)}, c_{bwr}(p))) = \alpha_\phi^{bw\_freq^+}(p) \tag{50}$$

The overall parallel speedup of Equation (14) is thus also invariant to $\eta$. When fitting the model to experimental data to obtain its parameter values, we must thus fix one of the parameters and can only retrieve relative parameter values for the remaining ones but not their absolute quantities. In the following, we set $Z_1^+ = 1$, i.e. we use the scaling $\eta = \frac{1}{Z_1}$.

The green (solid) curve in Figure 11 is the result of fitting our model to the maxima of our data in the memory bound case. We use $c_{bwr}(p) = p$, i.e. all cores access a common shared bandwidth resource. Even though physically four memory controllers with a total of four used DDR4-DRAM channels exist, we can summarize them as a single shared bandwidth resource in this case, as the data allocation is interleaved among all of them. Accesses from all cores thus stress all the physical resources equally. We again exclude thread counts divisible by 4 (`div4`) which use a different algorithm and are instead UPI bound, as explained before. The obtained parameters for this example case are $\sigma = 0.0073$, $Z_1^+ = 1$ (fixed), $\mu^+ = 25.127$, $L_*^+ = 0.1126$, $H_1^+ = 0.2130$ and $k^+ = 108.86$. We graphically see that it describes well the experimental data. In the next section we further investigate the model's capability to describe our observed scalability curves. We check the model fitting more rigorously and for all our experiments.

### 5.2.3 Evaluation of the models

We now analyze how well the models of Section 5.2.1 (`freq`) and Section 5.2.2 (`bw_freq`), as well as Amdahl's law, can explain the data we obtained in our different experiments. Therefore, we fit the models to the *median* values observed in the repetitions of our experiments, such that the chosen parameters minimize the Mean Squared Error (MSE). Opposed to the fittings in previous sections, we now use data of all thread counts in the fitting process for all the models. This allows us to compare how well each model can explain the overall observed data. The models provide us with upper scalability bounds. They can thus just give predictions with low errors when the scalability is solely determined by the effects included in a model. Note that our fitting methodology does not search for a strict bound but for the closest fit to the data, with the lowest MSE. A more meaningful fit for Amdahl's law is for example obtained when proceeding as in the previous sections, i.e. when fitting to the observed maxima and only using thread counts which can be explained by Amdahl's law, so only thread counts smaller or equal to 20 to avoid including the frequency scaling. Though, here we want to compare the models and see what is the lowest MSE they can achieve for our data. For *MKL* we still exclude thread counts divisible by four (`div4`) from the fitting, but also from the evaluation of the error metric, as MKL uses a fundamentally different computation scheme in those cases. For Amdahl's law and the `freq` model we only have to determine a single parameter $\sigma$, whereas for the `bw_freq` model we need to fix $Z_1^+ = 1$ and find the five parameters $\sigma$, $\mu^+$, $L_*^+$, $H_1^+$ and $k^+$. We assume all cores access to one common shared bandwidth resource and therefore use $c_{bwr}(p) = p$ for the `bw_freq` model.

Table 3 shows the Root Mean Squared Error (RMSE) between the actual speedup and the speedup value predicted by each of the three fitted models (Amdahl's law, `freq`, `bw_freq`). We can distinguish three categories by comparing the `bw_freq` model with the `freq` model and Amdahl's law:

1. No improvement, i.e. `bw_freq` is not better in explaining the data as `freq`. An example is `(tiling, GCC, float, bind, balanced)`.

2. Improvement but still a large RMSE. Bandwidth saturation is likely present but not the only effect dominating the scalability, for example in the case of `(simple, ICC, float, interleaved, balanced)`. We know from Section 4.2.4 that this experiment is L3 bandwidth saturated but also suffers from storage resource conflicts on the L3 cache. We assume this category when the RMSE of `bw_freq` is less than $\frac{2}{3}$ of the one of `freq` or Amdahl's law, but the absolute RMSE value is still larger than 0.4. In Table 3 these cases are printed in italic.

3. Good fit, `bw_freq` can explain well the scalability curve. This is the case for all MKL experiments which are indeed bandwidth saturated. Furthermore, the experiments `(simple, GCC, float, interleaved, balanced)` and `(simple, Clang, float, interleaved, balanced)` are part of this category, which are the L3 bandwidth saturated cases from Section 4.2.3. We define this category again by an RMSE of `bw_freq` less than $\frac{2}{3}$ of the one of `freq` or Amdahl's law, but now also the absolute RMSE has to be smaller than 0.4. The table shows them in bold.

This classification is only meant as an illustration of the very different results and therefore the classification criteria are rather arbitrarily chosen. Nevertheless, it allows us to see that the `bw_freq` model can explain well the data of all bandwidth saturated experiments, and we can even identify if bandwidth saturation is important for the parallel scalability of an experiment.

Table 3: Achieved RMSE with the different scalability models (balanced affinity policy)

| Implementation | Call | Type | Memory allocation | Compiler | RMSE Amdahl | RMSE freq | RMSE bw_freq |
|---|---|---|---|---|---|---|---|
| simple | 2 | float | bind | GCC | 0.4550 | 0.4488 | 0.44348 |
| *simple* | *2* | *float* | *bind* | *ICC* | *1.4013* | *1.0387* | *0.55193* |
| simple | 2 | float | bind | Clang | 0.8554 | 0.8523 | 0.85126 |
| **simple** | **2** | **float** | **interleaved** | **GCC** | **1.4850** | **1.1853** | **0.39838** |
| *simple* | *2* | *float* | *interleaved* | *ICC* | *1.4941* | *1.1070* | *0.52571* |
| **simple** | **2** | **float** | **interleaved** | **Clang** | **0.7808** | **0.5629** | **0.32976** |
| simple | 2 | double | bind | GCC | 0.5453 | 0.4761 | 0.46840 |
| *simple* | *2* | *double* | *bind* | *ICC* | *1.3328* | *1.1283* | *0.48844* |
| simple | 2 | double | bind | Clang | 1.0300 | 0.9499 | 0.80134 |
| **simple** | **2** | **double** | **interleaved** | **GCC** | **1.9451** | **1.6213** | **0.33475** |
| simple | 2 | double | interleaved | ICC | 0.6208 | 0.6277 | 0.54290 |
| **simple** | **2** | **double** | **interleaved** | **Clang** | **1.3621** | **0.9574** | **0.19216** |
| tiling | 2 | float | bind | GCC | 0.7744 | 0.6081 | 0.60788 |
| tiling | 2 | float | bind | ICC | 0.6306 | 0.5695 | 0.55324 |
| tiling | 2 | float | bind | Clang | 0.7520 | 0.6105 | 0.60956 |
| tiling | 2 | float | interleaved | GCC | 0.9913 | 0.8103 | 0.80252 |
| tiling | 2 | float | interleaved | ICC | 0.8398 | 0.7542 | 0.74864 |
| tiling | 2 | float | interleaved | Clang | 0.9278 | 0.7804 | 0.77834 |
| tiling | 2 | double | bind | GCC | 0.4871 | 0.2524 | 0.24316 |
| tiling | 2 | double | bind | ICC | 0.3487 | 0.3376 | 0.25151 |
| tiling | 2 | double | bind | Clang | 0.4311 | 0.2594 | 0.23754 |
| tiling | 2 | double | interleaved | GCC | 0.8673 | 0.5190 | 0.54855 |
| tiling | 2 | double | interleaved | ICC | 0.6872 | 0.4534 | 0.45181 |
| tiling | 2 | double | interleaved | Clang | 0.7407 | 0.4576 | 0.45325 |
| **MKL** | **1** | **float** | **bind** | **GCC** | **0.4642** | **0.3768** | **0.04700** |
| **MKL** | **1** | **float** | **bind** | **ICC** | **0.4365** | **0.3572** | **0.03891** |
| **MKL** | **1** | **float** | **bind** | **Clang** | **0.4360** | **0.3545** | **0.04635** |
| **MKL** | **1** | **float** | **interleaved** | **GCC** | **0.3051** | **0.1912** | **0.03572** |
| **MKL** | **1** | **float** | **interleaved** | **ICC** | **0.2600** | **0.1575** | **0.03117** |
| **MKL** | **1** | **float** | **interleaved** | **Clang** | **0.2547** | **0.1502** | **0.02599** |
| **MKL** | **1** | **double** | **bind** | **GCC** | **0.9421** | **0.8729** | **0.08479** |
| **MKL** | **1** | **double** | **bind** | **ICC** | **0.8925** | **0.8274** | **0.07518** |
| **MKL** | **1** | **double** | **bind** | **Clang** | **0.9042** | **0.8372** | **0.08393** |
| **MKL** | **1** | **double** | **interleaved** | **GCC** | **0.5873** | **0.4291** | **0.14606** |
| **MKL** | **1** | **double** | **interleaved** | **ICC** | **0.6068** | **0.4517** | **0.11842** |
| **MKL** | **1** | **double** | **interleaved** | **Clang** | **0.5888** | **0.4340** | **0.12344** |
| **MKL** | **2** | **float** | **bind** | **GCC** | **1.0854** | **0.8735** | **0.14764** |
| **MKL** | **2** | **float** | **bind** | **ICC** | **1.0856** | **0.8785** | **0.27498** |
| **MKL** | **2** | **float** | **bind** | **Clang** | **1.0810** | **0.8703** | **0.27915** |
| **MKL** | **2** | **float** | **interleaved** | **GCC** | **0.7865** | **0.4643** | **0.05391** |
| **MKL** | **2** | **float** | **interleaved** | **ICC** | **0.7770** | **0.4647** | **0.05657** |
| **MKL** | **2** | **float** | **interleaved** | **Clang** | **0.7674** | **0.4428** | **0.04649** |
| **MKL** | **2** | **double** | **bind** | **GCC** | **1.2954** | **1.1990** | **0.10940** |
| **MKL** | **2** | **double** | **bind** | **ICC** | **1.2710** | **1.1749** | **0.10088** |
| **MKL** | **2** | **double** | **bind** | **Clang** | **1.2916** | **1.1937** | **0.11317** |
| **MKL** | **2** | **double** | **interleaved** | **GCC** | **1.0902** | **0.8170** | **0.25456** |
| **MKL** | **2** | **double** | **interleaved** | **ICC** | **1.1370** | **0.8581** | **0.21978** |
| **MKL** | **2** | **double** | **interleaved** | **Clang** | **1.1002** | **0.8243** | **0.22840** |

# 6 Discussion and conclusion

In this work, we show different factors dominating performance scalability and their characteristic speedup curves, from Amdahl's sequential fraction over work distribution and algorithmic considerations, up to shared resources. Latter are especially interesting from our point of view since intense sharing of hardware resources between cores is a consequence of the tight integration in modern CMPs. They might limit performance of the truly parallel fraction of a program and also go alongside communication and synchronization considerations: all our experiments use matrix multiplication which is embarrassingly parallel with no communication and a very small sequential fraction. From the experiments we can conclude that this sharing of resources can dominate scalability in many cases and thus needs to be taken into account when optimizing performance, be it by tuning code or by changing the hardware design. In this work, we only analyze interference between threads of the same application which all run the same code. Nevertheless, the same kind of interference can also happen between different processes, or threads of different types in an application. In the following section we summarize our results and draw conclusions from them.

## 6.1 Results and contributions

Our experiment data shows many cases where an increase in the thread count, and with that core count, causes a lowered performance. In all cases these are the combination of two or more effects. One effect causes saturation of performance while the other one decreases performance per core, so that the overall performance decreases - even if the reported CPU usage is constantly high. It seems to the OS and user that the hardware is fully utilized, though the chip's microarchitecture is actually badly used.

The general principle of co-scheduling solutions is to schedule processes together that complement each other in their resource usage, i.e. applications causing high contention on a resource together with processes using this resource only rarely. In this work, we analyze shared resources in a system with modern CMPs with high core counts. Contrary to the findings of published co-scheduling works, we observe that contention for many other resources than the main memory bandwidth can represent important performance limiting factors. Our insights can be used to improve the heuristics of co-scheduling approaches.

We extend the classification of shared resources of [1] in bandwidth and storage resources with a shared power and temperature budget. Since chips are hitting the power wall, sharing of the maximum power consumption and heat dissipation between all cores of a die is getting more and more important and processor vendors integrate strategies like Intel's Turbo Boost to cope with this problem. Those strongly affect performance, in particular parallel scalability. There are two possibilities to look at features like Turbo Boost. The marketing name suggests us that performance is increased for low numbers of active cores compared to the baseline. Charles et al. [51] and Annavaram et al. [52] thus interpret that such technologies can accelerate sequential code parts and with this mitigate Amdahl's law to some extend. However, all cores are full-fledged cores, i.e. each core of the chip contains exactly the same hardware resources. Each core can run at the maximum Turbo Boost frequency, when the other cores of the chip are idle, and thus is capable to deliver the same high performance as any other core of the system. Cores only have to slow down when other cores on the chip interfere and also consume high power. This situation is very similar to e.g. memory bandwidth shared between cores: a single active core can use all the resource by itself whereas in parallel execution cores have to share and probably get their data slower. Hence, we argue to also see Turbo Boost the same way, i.e. as reducing performance in the parallel case due to a common shared budget. Of course this does not mean that it is a feature which lowers the chip's maximum achievable performance. On the contrary, since the overall budget is a fixed design constraint, less cores could be integrated per chip without such a technology. We just want to highlight that a modern multicore processor with $m$ cores cannot deliver the performance of $m$ times using only a single core of the same chip (which has the same power/temperature budget). Scalability worse than linear is intrinsic to such a design, except

for special cases of applications that can compensate the decreasing per core performance by another effect, e.g. an improved cache behavior (as superlinear speedup cases). Applications that already do not parallelize very well can perform worse for high thread counts compared to lower thread counts. We remark that this is very close to an Asymmetric Multiprocessor (AMP) with a core for sequential execution which is more powerful than the cores for parallel code phases. The resource of power and temperature is dynamically redistributed either to a single core or to be shared by multiple cores.

Furthermore, as we observe that the bound imposed by Amdahl's law is often far from the actual measured scalability behavior, we also provide a general formula to describe scalability in the presence of resource sharing effects with Equation (14). In its general form, it can model any scaling limit of the parallel program part. We provide detailed models for (deterministic) frequency scaling in Equation (19) and bandwidth saturation in Equation (40) which makes the model useful in practice.

In the next section, let us come back to the explanations for performance scalability behavior found in the literature which we present in Section 2.3 and review them with the knowledge of our results.

## 6.2 Discussion of previous work

The authors of [11] explain their performance loss with increasing core count after reaching a maximum by memory bandwidth saturation. They indicate that other increasing overheads, which notably exist in their model, still lead to a quasi-linearly scaling (see right of Figure 10 in [11], green vs. red curve) and are thus not the cause. However, bandwidth saturation alone also would only cause the scalability curve to flatten and not to decrease. Only the combination of bandwidth saturation with the other increasing overheads causes the performance decrease.

Hammond et al. present a speedup curve which stays constant for most increases in the thread count (flat regions) and only increases at certain steps [12]. The curve also shows some slowdowns. They accredit all the behavior to thermal throttling. Assuming a properly cooled system, we know from Section 4.2.5 that this would cause a differently shaped curve. For performance to stay approximately identical for a range of core counts as in their flat regions, the overall computation power provided by all used cores needs to stay constant. This again would require the core frequency to decrease a bit for each core count increase in order to compensate for the additional added core. Though, analogous to Table 2, their processor's frequency follows a deterministic behavior, containing frequency reductions only at certain core counts. Instead, we identify that their benchmark uses a small number of work chunks to be calculated in parallel on the available cores (as in Figure 5a). From the position of the steps in the presented data, we can determine those to be likely 48 chunks for their Skylake experiment (AVX-512) and 64 chunks for their Haswell experiment (AVX2). Just the slowdowns which occur e.g. from 8 to 9 threads and from 16 to 17 threads are actually due to downclocking. They correspond exactly with deterministic scaling steps of the Xeon Platinum 8160 processor used in their experiment (3.0 GHz to 2.6 GHz and 2.3 GHz to 2.1 GHz).

This analysis of two example cases shows that our work provides a better understanding of scalability limits in modern multicores, allowing to identify performance issues. Obviously, there are still a few open paths to continue this work.

## 6.3 Future work

Our scope in this work is to show how resource sharing can affect performance nowadays. It remains to apply the same methods we use to more complex real-world applications instead of matrix multiplication. Those likely contain different phases with different characteristics. However, each of those phases individually might be bounded by one of the different shared resources we observe here. Our modeling is also only verified on the matrix multiplication case. We as well plan to validate it on more complex parallel applications.

Our modeling needs to be extended to cover all cases of shared resources which can limit scalability, in particular also shared storage resources. We would like to better establish the relation of model

parameters to actual machine parameters, such as e.g. the main memory access bandwidth, and application properties, like its memory access intensity. Then it would be possible to actually predict scalability and assess how certain changes would affect it, for example what happens when improving (software) prefetching or increasing the available memory access bandwidth. It could also allow to estimate how co-running processes behave on a multicore, especially how they interfere and thus lower each others performance.

Furthermore, the possible impact of resource sharing in multicores on compiler optimizations should be studied. Compilers usually employ some sort of performance model to predict which code version reaches the highest performance when running on a specific target machine. A compiler translating a parallel program should not only be guided by the architecture of a single core but should also evaluate the influence of resource sharing effects as we present them in this work. Our models can serve as a starting point for such performance predictions.

We do not show data for executions with Turbo Boost enabled, even though this is the usual operation case. Those might be interesting as they will show even stronger frequency scaling behavior due to the power and temperature sharing.

An analysis of SMT (Hyper-Threading) might as well be interesting in our context. The different hardware threads per core then share (almost) all resources of a single physical core, i.e. we have an extreme case of resource sharing. Nonetheless, the expectations are different: as almost no additional hardware is added, we would never assume performance to double like when adding a complete core. Instead, SMT aims to hide latency imposed by e.g. resources shared between cores (like memory bandwidth, ...) as the second thread might be able to do useful work during this time. Consequently, this technique might allow the performance to continue scaling slightly further before reaching the performance limits due to shared resources which we analyzed in this work.

# A Block sizes of the tiling implementation

The block sizes which we use for the tiling algorithm (Listing 2) given in Section 3.4.1 were determined in two steps. First, we chose the values such that the subblocks of the matrices accessed by the different loop levels fit well with the cache sizes of our machine. We then experimentally tuned those values to obtain the maximal performance, however, only for a single example case (GCC, one specific thread count). This means that in some configurations other values might lead to higher performance, but as the aim of this work is not to develop the fastest matrix multiplication implementation, this is not of importance in our context.

Each loop iterates one of the indices used to address matrix elements, over a sub-range of the full matrix dimension. All iterations of one loop together, including the contained inner loops, therefore access a subblock of each of the matrices. To simplify the description, let us refer to the loop in line $i$ in Listing 2 as *Loop* $i$. For our analysis, we also need to take into account the compiler since it might change the loop structure and with this the sizes of data blocks accessed by the different loop levels. We thus inspected the generated assembly code and could identify that all compilers (GCC, ICC, Clang) performed two important transformations on our code:

- Loop 9 gets fully unrolled and
- Loop 7 is vectorized afterwards with 512 bit width (AVX-512).

Let us now look at the different block sizes. The purpose of $BS_{k\_reg}$ is to re-use data in processor registers, in particular the values of elements of $M_C$. Note that Loop 9 always uses the same element of $M_C$ and needs to load it before each computation and store it again after the computation. If we can keep it in a register, we just need to load it once, do $BS_{k\_reg}$ computations and then store it again once afterwards. Furthermore, Loop 9 is unrolled and Loop 7 then gets vectorized. This means that each of the (unrolled) computations of Loop 9 does not operate just on a single value but on a vector of values (16 floats or 8 doubles). Each load and store is consequently 64 byte long - a whole cache line. We thus want $BS_{k\_reg}$ to be as large as possible. Nevertheless, we also want to keep the values of $M_A$ in registers over the iterations of Loop 7 as also here always the same values are needed. If we choose $BS_{k\_reg}$ too large, not enough vector registers are available for this purpose. $BS_{k\_reg} = 4$ experimentally showed to be the optimal choice.

Table 4 shows the data size of each matrix accessed by the different loop levels. For each loop, we mark the associated block size variable. We indicate for Loop 9 the accessed data sizes as the source code suggests and the actual values of the real executed binary code generated by the compilers (vectorized). An arrow to the left indicates that a loop works on the same data as the next inner loop, i.e. all its iterations re-use the same data. Those cases are of special interest to us as we want the re-used data to stay in the fastest possible cache level, thus we have to adjust their sizes according to the cache capacities of our machine. From this we can conclude the desired block sizes, for float:

- We already chose $BS_{k\_reg}^{float} = 4$.
- From Loop 6/7, to fit the data of $M_B$ in the L1 cache we need $4\,\mathrm{B} \times BS_{k\_reg}^{float} \times BS_j^{float} = 16\,\mathrm{KiB}$, thus we obtain $BS_j^{float} = 1024$. We only use half the cache capacity to let space for other data.
- From Loop 2/3, to fit the data of $M_C$ in L2 we want $4\,\mathrm{B} \times D_M \times BS_i^{float} = 512\,\mathrm{KiB}$, thus we get $BS_i^{float} = 32$. We again only use half the cache capacity to let space for other data.
- From Loop 3/5, to fit the data of $M_A$ in L1 we must satisfy $4\,\mathrm{B} \times BS_i^{float} \times BS_k^{float} = 4\,\mathrm{KiB}$, thus we need $BS_k^{float} = 32$. We already used 16 KiB of L1 before. By using an additional 4 KiB, enough space is left for data which is only used once, such that it does not conflict with the data we intend to keep.

The same argumentation holds for the double datatype with the respective values.

Table 4: Accessed data sizes by the loop levels of the tiling algorithm

(a) float

| | **Loop** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 9 (source) $BS^{float}_{k\_reg}$ = 4 | 9 (binary) $BS^{float}_{k\_reg}$ = 4 | 7 $BS^{float}_{j}$ = 1024 | 6 $BS^{float}_{i}$ = 32 | 5 $BS^{float}_{k}$ = 32 | 3 $D_M$ = 4096 | 2 $D_P$ = 3072 | 1 $D_N$ = 4096 |
| $\mathbf{M_A(i,k)}$ | 16 B | 16 B | $\overleftarrow{registers}$ | 512 B | **4 KiB** | $\overleftarrow{L1}$ | 384 KiB | 48 MiB |
| $\mathbf{M_B(k,j)}$ | 16 B | 256 B | **16 KiB** | $\overleftarrow{L1}$ | 128 KiB | 512 KiB | 48 MiB | $\overleftarrow{L3+L2(part)}$ |
| $\mathbf{M_C(i,j)}$ | 4 B | 64 B | 4 KiB | 128 KiB | $\overleftarrow{L2}$ | **512 KiB** | $\overleftarrow{L2}$ | 64 MiB |

(b) double

| | **Loop** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 9 (source) $BS^{double}_{k\_reg}$ = 4 | 9 (binary) $BS^{double}_{k\_reg}$ = 4 | 7 $BS^{double}_{j}$ = 512 | 6 $BS^{double}_{i}$ = 16 | 5 $BS^{double}_{k}$ = 16 | 3 $D_M$ = 4096 | 2 $D_P$ = 3072 | 1 $D_N$ = 4096 |
| $\mathbf{M_A(i,k)}$ | 32 B | 32 B | $\overleftarrow{registers}$ | 512 B | **2 KiB** | $\overleftarrow{L1}$ | 384 KiB | 96 MiB |
| $\mathbf{M_B(k,j)}$ | 32 B | 256 B | **16 KiB** | $\overleftarrow{L1}$ | 64 KiB | 512 KiB | 96 MiB | $\overleftarrow{L3+L2(part)}$ |
| $\mathbf{M_C(i,j)}$ | 8 B | 64 B | 4 KiB | 64 KiB | $\overleftarrow{L2}$ | **512 KiB** | $\overleftarrow{L2}$ | 128 MiB |

# List of Figures



# List of Tables



# Listings

## Glossary: Processors

**Processor** A processor or processing unit is a general term for any type of entity that performs operations on data in a computing system. It does not specify further the purpose of the entity (general purpose main processor, graphics processor, etc.). Depending on the context, it can refer either to a whole processor *chip* or to a single processor *core*.

**CPU** The Central Processing Unit (CPU) is the main *processor* of the computing system. Other specialized processors might exists in the system, e.g. a Graphics Processing Unit (GPU). In this work we are only concerned with CPUs. The term CPU is also ambiguous as it can refer to a complete CPU *chip* or to an individual CPU *core*.

**Core** A core is an individual processing unit (or *processor*) of the system. It can independently execute a stream of instructions. A computing system might contain multiple CPU cores, either in separate *chips* or integrated into a single *chip* (multicore).

**Physical core** A physical core is an independent processing *core* as it exists in the actual hardware implementation.

**Logical core** A logical core is a processing *core* as it is seen from outside the hardware by the OS. Multiple logical cores might be implemented by the same *physical core*, e.g. always two logical cores are realized by one physical core in our test machine. In other words, a single *physical core* can execute multiple software threads at the same time, which is called Simultaneous Multithreading (SMT).

**Chip** A chip is a physical device containing an integrated electric circuit in a chip *package*. In some contexts the term chip is also used as synonym for a single die only, i.e. one package can contain multiple chips and not just one. However, our notion of a chip means everything, in particular all dies, integrated together in one common package. Chips thus consist of a single or multiple *dies* to realize the required functionality. Evidently, a CPU chip is a chip implementing a CPU. It might include just one *core* (single-core) or multiple CPU *cores* (multicore). Latter case is also called a Chip Multiprocessor (CMP) and is the focus of this work.

**Die** A die is a block of semiconductor material containing an electric circuit. One or multiple dies together in a common *package* form a *chip*.

**Package** A package is the supporting case of a *chip* which protects the *die(s)* from physical damage and which allows to connect the chip to its surrounding circuitry.

**Socket** A socket is a physical connector which holds and connects a CPU *chip* in its *package* to the surrounding circuitry. The term is often used to denote the CPU chip itself, however, this is not precise. For example, nowadays, in many cases the CPU chips are soldered directly to the mainboard, so no physical socket exists anymore. As we focus on effects caused by the integration of multiple cores into a single physical device in this work, we use the more accurate term *chip* throughout this document.

## Acronyms

**AMP** Asymmetric Multiprocessor. 8, 49

**CMP** Chip Multiprocessor. 4, 6, 7, 48, 54

**CPU** Central Processing Unit. 4, 14, 39, 54

**DDR** Double Data Rate. 14

**DRRIP** Dynamic Re-Reference Interval Prediction. 28

**DVFS** Dynamic Voltage and Frequency Scaling. 1, 2, 14–16, 35, 38, 41

**FMA** Fused Multiply Add. 15, 18

**FSB** Front Side Bus. 11

**GPU** Graphics Processing Unit. 54

**HPC** High-Performance Computing. 4, 6, 16

**HWP** Hardware Managed P-states. 16

**IACA** Intel Architecture Code Analyzer. 7

**ILP** Instruction-Level Parallelism. 4

**IMC** Integrated Memory Controller. 14, 29

**LLC** Last Level Cache. 11

**LRU** Least Recently Used. 28

**MBM** Memory Bandwidth Monitoring. 22, 29

**MKL** Math Kernel Library. 17

**MSE** Mean Squared Error. 24, 46

**MSR** Model-Specific Register. 22

**NoC** Network-on-Chip. 4, 29

**NUMA** Non-Uniform Memory Access. 14

**OS** Operating System. 6, 13, 15, 16, 18, 19, 21, 35, 48, 54

**RDT** Resource Director Technology. 22

**RMSE** Root Mean Squared Error. 46, 47, 53

**SIMD** Single instruction, multiple data. 14, 18

**SMT** Simultaneous Multithreading. 11, 13, 19, 50, 54

**TDP** Thermal Design Power. 15

**TLB** Translation Lookaside Buffer. 32

**UPI** Intel Ultra Path Interconnect. 14, 40, 45

**USL** Universal Scalability Law. 9, 35

**WCET** Worst Case Execution Time. 10

# References

[1] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *Proceedings of the 24th international conference on Concurrency Theory*, pages 25–43. Springer, 2013.

[2] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 4505–4515. PMLR, 2019.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[4] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[5] Leonid Yavits, Amir Morad, and Ran Ginosar. The effect of communication and synchronization on Amdahl's law in multicore systems. *Parallel Computing*, 40(1):1–16, 2014.

[6] Neil J Gunther. A simple capacity model of massively parallel transaction systems. In *19th International Computer Measurement Group Conference*, pages 1035–1035. Computer Measurement Group Inc, 1993.

[7] Neil J Gunther. A general theory of computational scalability based on rational functions. *arXiv preprint arXiv:0808.1431*, 2008.

[8] Neil J Gunther, Shanti Subramanyam, and Stefan Parvu. A Methodology for Optimizing Multithreaded System Scalability on Multicores. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multicore and Many-core Computing Systems*, pages 363–384. John Wiley and Sons Inc., 2017.

[9] Sasko Ristov and Marjan Gusev. Superlinear speedup for matrix multiplication. In *Proceedings of the 34th International Conference on Information Technology Interfaces*, pages 499–504. IEEE, 2012.

[10] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. Superlinear speedup in HPC systems: Why and when? In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, pages 889–898. IEEE, 2016.

[11] Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, and Amaury Graillat. Efficient parallelization of large-scale hard real-time applications. Research Report RR-9180, INRIA Paris, Jun 2018.

[12] Simon Hammond, Courtenay Vaughan, and Clay Hughes. Evaluating the Intel Skylake Xeon processor for HPC workloads. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 342–349. IEEE, 2018.

[13] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[14] Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21. IEEE, 1999.

[15] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[16] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224. IEEE, 2010.

[17] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[18] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *ACM SIGBED Review*, 12(1):28–36, 2015.

[19] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):34, 2012.

[20] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs. In *Proceedings of the 11th International Conference on High Performance Computing*, HiPC'04, pages 286–296, Berlin, Heidelberg, 2004. Springer.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008.

[22] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. *SIGARCH Computer Architecture News*, 38(1):129–142, 2010.

[23] Major Bhadauria and Sally A McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 189–199. ACM, 2010.

[24] Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-Based Manycore Partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 107–116, New York, USA, 2012. ACM.

[25] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, USA, 2011. USENIX Association.

[26] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 234–244, New York, USA, 2000. Association for Computing Machinery.

[27] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 123–132. IEEE, 2013.

[28] Stijn Eyerman and Lieven Eeckhout. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 91–102, New York, USA, 2010. Association for Computing Machinery.

[29] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM SIGARCH Computer Architecture News*, 37(1):265–276, 2009.

[30] Abdelhafid Mazouz, Denis Barthou, et al. Study of variations of native program execution times on multi-core architectures. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 919–924. IEEE, 2010.

[31] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[32] Wikichip. Skylake (server) - Microarchitectures - Intel. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`. Last accessed: 23 Jul 2019.

[33] Efraim Rotem. Intel architecture, code name Skylake deep dive: A new architecture to manage power performance and energy efficiency. Presentation at *Intel Developer Forum (IDF15)*, 2015.

[34] Akhilesh Kumar and Malay Trivedi. Intel Xeon scalable processor architecture deep dive. Presentation at *Intel Press Workshops June 2017*, 2017.

[35] Wikichip. Xeon Gold 6130 - Intel. `https://en.wikichip.org/wiki/intel/xeon_gold/6130`. Last accessed: 27 Jun 2019.

[36] Intel Corporation. *Intel® Xeon® Processor Scalable Family Specification Update*, September 2019. Reference Number: 336065-010US.

[37] Mathias Gottschlag and Frank Bellosa. Mechanism to Mitigate AVX-Induced Frequency Reduction. *arXiv preprint arXiv:1901.04982*, 2018.

[38] Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. The Speedup-Test: a statistical methodology for programme speedup analysis and computation. *Concurrency and computation: practice and experience*, 25(10):1410–1426, 2013.

[39] Abdelhafid Mazouz, Sid Touati, and Denis Barthou. Analysing the variability of OpenMP programs performances on multicore architectures. In *Fourth workshop on programmability issues for heterogeneous multicores (MULTIPROG-2011)*, 2011.

[40] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, November 2015. Version 4.5.

[41] Abdelhafid Mazouz, Denis Barthou, et al. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In *2011 International Conference on High Performance Computing & Simulation*, pages 273–279. IEEE, 2011.

[42] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. autopin - automated optimization of thread-to-core pinning on multicore systems. In *Transactions on high-performance embedded architectures and compilers III*, pages 219–235. Springer, 2011.

[43] Ghassan Almaless and Franck Wajsburt. On the scalability of image and signal processing parallel applications on emerging cc-NUMA many-cores. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*, pages 1–8. IEEE, 2012.

[44] H. Ying, Murat Efe Guney, and S. Shane. Tips to Measure the Performance of Matrix Multiplication Using Intel MKL. `https://software.intel.com/en-us/articles/a-simple-example-to-measure-the-performance-of-an-intel-mkl-function`, December 2017. Last accessed: 21 Jun 2019.

[45] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[46] Intel Corporation. *Intel® 64 and IA32 Architectures Performance Monitoring Events*, December 2017. Revision 1.0, Document Number: 335279-001.

[47] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement`, Jan 2013. Last accessed: 26 Jul 2019.

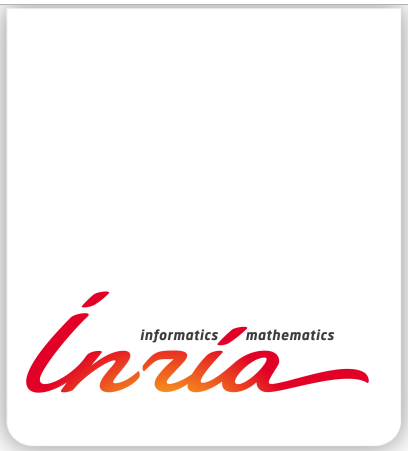
[48] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, 38(3):60–71, 2010.

[49] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013. ISBN-13: 978-0124077263.

[50] János Sztrik. *Basic Queueing Theory*. GlobeEdit, 2016. ISBN-13: 978-3639734713.

[51] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel® Core™ i7 Turbo Boost feature. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 188–197. IEEE, 2009.

[52] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's law through EPI throttling. *ACM SIGARCH Computer Architecture News*, 33(2):298–309, 2005.




**RESEARCH CENTRE**
**SOPHIA ANTIPOLIS – MÉDITERRANÉE**

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399