Characterizing and Mitigating Performance Variability in Parallel Applications on Modern HPC multicore Systems

Characterizing and Mitigating Performance Variability in Parallel Applications on Modern HPC multicore Systems

Minyu Cui
Chalmers University of Technology and University of Gothenburg
Göteborg, Sweden
minyu@chalmers.se

Miquel Pericas
Chalmers University of Technology and University of Gothenburg
Göteborg, Sweden
miquelp@chalmers.se

Abstract

In high-performance computing (HPC), OpenMP has become the de facto programming model for shared-memory systems. However, running OpenMP-based parallel applications on multicore systems often faces the challenge of performance variability, particularly as core counts increase in modern HPC clusters. Factors spanning from Operating Systems (OS) and hardware feature to OpenMP implementation can significantly impact performance stability. This paper evaluates execution time variability across five multicore systems from multiple HPC clusters, covering two different ISAs and using five OpenMP benchmarks and a real-world mini-app compiled with both gcc and llvm/clang. We analyze the effects of various factors such as thread-pinning, OpenMP runtime implementations, OpenMP scalability, simultaneous multithreading (SMT), core resource reservation, frequency scaling, and platform-specific features such as hybrid architecture core configurations. Our findings highlight the complex interplay of these factors in performance variability and propose lightweight mitigation strategies to enhance the stability of OpenMP programs for developers and system users.

CCS Concepts

• **Computing methodologies** → *Massively parallel algorithms*; • **Computer systems organization** → **Multicore architectures**.

Keywords

HPC, performance variability, mitigation strategies, OpenMP, simultaneous multithreading, resource reservation, multicore systems, hybrid architecture.

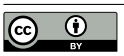
ACM Reference Format:
Minyu Cui and Miquel Pericas. 2025. Characterizing and Mitigating Performance Variability in Parallel Applications on Modern HPC multicore Systems. In *22nd ACM International Conference on Computing Frontiers (CF '25), May 28–30, 2025, Cagliari, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3719276.3725184

1 Introduction

Optimization efforts in HPC usually focus on the performance of parallel applications, with performance variability often addressed only as an afterthought. However, in parallel computing with OpenMP, performance stability is crucial for consistent and repeatable results, especially on large-scale multicore HPC clusters. If one or more threads encounter resource contention, OS interrupts, or non-application work during the execution of parallel applications, the execution time will likely be impacted in hard to predict ways. This problem has gained more attention with the rise of large-scale multicore platforms on modern HPC clusters. Performance stability can be influenced by OS features, hardware configurations, and OpenMP implementation. A major source of execution time variability is OS noise, caused by OS daemons and system processes[2, 6, 21] that interfere with the user application. Resource contention further contributes to variability, as shown in previous studies on shared network [3] and I/O interference [27]. On SMT-enabled systems, over-subscribing physical cores without reserving cores for OS leads to performance instability [14, 16], while hardware factors such as dynamic voltage scaling [17] also add variability. In OpenMP, thread migration can improve core utilization but often increases variability due to data migration overhead.

Thread-pinning has effectively stabilized performance [10, 18–20], although it does not always optimize execution time. Some studies suggest that letting the OS manage thread placement can enhance speed [19], but at the cost of stability. A common strategy to mitigate OS noise is to reserve core resources. Systems like Cray XE implement core specialization [24], reserving one or more cores for OS and service threads. LUMI [15] dedicates 12.5% of the cores in the LUMI-G partitions to the OS for low-noise operation. With modern systems featuring increasing core counts, a critical question arises: What is the optimal number of cores to reserve while avoiding core resource wastage (Q1)? On SMT systems, reserving threads for OS has been shown to reduce performance perturbations [16]. Hybrid architectures such as Intel AlderLake, which has both performance cores (P-cores) and efficient cores (E-cores), are less common in HPC due to load-balancing challenges caused by asymmetry. Little is known about how the combined use of these cores affects performance stability. This raises an intriguing question: What is the optimal number of enabled E-cores for OS tasks to minimize interference and enhance system efficiency (Q2)?

Prior studies have explored individually thread pinning, SMT, frequency variation, or focused on specific platforms like Intel Xeon Phi [2, 5]. None has synthesized their combined impact on performance stability across modern machines. What happens to performance stability when these effects work together (Q3)? Given the increasing complexity of modern systems, understanding how these effects interact is essential. To the best of our knowledge,



CF '25, Cagliari, Italy

ACM ISBN 979-8-4007-1528-0/25/05
https://doi.org/10.1145/3719276.3725184

no existing research directly addresses the three outlined questions (Q1-Q3). Furthermore, most previous studies focus on a single architecture or a narrow range of homogeneous systems, limiting the generalizability of their findings across diverse hardware platforms. Given the wide range of modern architectures, from traditional homogeneous systems to hybrid designs like Intel AlderLake, there is a critical need to evaluate performance variability in parallel applications across modern HPC architectures to ensure broader applicability. This work addresses questions Q1-Q3 by evaluating performance variability across diverse platforms, including traditional homogeneous systems and hybrid designs. Our key contributions include:

- **Comprehensive experimentation:** We conducted extensive experiments using five OpenMP microbenchmarks and a real-world mini-app (Lulesh) across five platforms including two distinct instruction set architectures (ISA) and four vendors: AMD, Intel, Fujitsu, and AWS.
- **Insights into performance variability:** We identified key factors influencing stability without relying on kernel modules or platform-dependent performance counters. We proposed effective lightweight, user-level mitigation strategies, including thread placement control, OpenMP implementation, core resource reservation for the OS, and isolation from co-running processes. Our findings also highlight the impact of frequency scaling, particularly on AMD Zen2 and Intel Xeon platforms, a factor beyond user control. These insights shed light on OpenMP programming-level recommendations, for system user and OpenMP code developers.
- **Case studies on resource reservation for OS:** We examined optimal core reservation on SMT platform and evaluated E-core allocations on Intel AlderLake to minimize OS interference and enhance stability of user workloads, while avoiding core over-reservation to ensure efficient usage.

The remainder of the paper is organized as follows. Section 2 presents the experimental environment. Section 3 outlines our methodology for evaluating performance variability, Section 4 presents experimental results and mitigation strategies. Section 5 provides recommendations, followed by related work in Section 6 and conclusion in Section 7.

2 Experimental Environment

This section outlines the experimental environment, including hardware platforms and compiler versions, which are summarized in Table 1, and the OpenMP benchmarks and mini-app used in this paper.

2.1 Hardware platforms

To ensure broad applicability within HPC community, we adopted five diverse multicore platforms.

AMD Zen2 (PDC Center): An HPE Cray EX supercomputer with dual AMD EPYC Zen2 2.25GHz 64-core processors, totaling 128 physical cores and 256 hardware cores/threads (hyperthreading). It has eight NUMA nodes (16 physical cores each) and operates at a maximum frequency of 3.4GHz. Additionally, each socket functions as a quad-NUMA domain. It runs SUSE Linux Enterprise Server 15 SP3 with the Linux kernel version 5.3.18-150300.59.76_11.0.53-cray_shasta_c.

Intel AlderLake (local workstation): A hybrid design with 8 performance cores (P-cores, supporting hyperthreading) and 8 efficient cores (E-cores) in a single NUMA domain. The cores can operate with a maximum frequency of 3.2GHz. It runs Ubuntu 22.04.5 LTS with the Linux kernel version 5.19.0-32-generic.

Intel Xeon (C3SE Cluster): This platform integrates two Intel Xeon Gold 6130 2.1GHz 16-core processors per node, totaling 32 cores across two NUMA nodes. The cores can operate at a maximum frequency of 3.7GHz. It runs Rocky Linux release 8.9 (Green Obsidian), with the Linux kernel version 4.18.0-513.11.1.el8_9.0.1.x86_64.

Fujitsu A64FX (Barcelona Supercomputing Center): Powered by ARM processors manufactured by Fujitsu A64FX CPU (Armv8.2-A + SVE), this platform features 48 cores across four NUMA nodes (each with 12 cores), along with 12 "assistant cores", which are dedicated to handling OS activities and are not visible to the user. The cores operate at a maximum frequency of 2.20 GHz. It runs Red Hat Enterprise Linux Server 8.1 (Ootpa) with the Linux kernel version 4.18.0-147.3.1.el8_1.aarch64.

AWS Graviton3 (EC2 C7g instances): This platform features the latest AWS Graviton3 processors with 64 CPU cores in a single NUMA node. It runs Amazon Linux release 2 (Karoo) with the Linux kernel version 5.10.167-147.601.amzn2.aarch64.

We used both gcc and llvm compilers, along with their respective OpenMP libraries, across all platforms. On Graviton3, the ARM Compiler for Linux (ACFL), based on llvm, was used. Compiler versions are provided in Table 1.

2.2 OpenMP benchmarks and application

Depending on their characteristics and relevance to the goal of this paper, we evaluated various OpenMP benchmarks and a real-world mini-app, including both compute- and memory-bound workloads. *BabelStream* [7] is a widely used memory-bound benchmark to measure sustainable memory throughput, via simple operations (kernels) such as `copy`, `add`, `mul`, `triad`, and `dot` products. The EPCC microbenchmark suite [4, 13, 26] includes *schedbench*, *syncbench*, *taskbench* and *arraybench*, focusing on loop scheduling, synchronization, task scheduling, and array-based operations. *Lulesh* [11, 12] incorporates both compute- and memory-bound kernels, serving as a compact, full-featured mini-app. The insights learned from the mini-app exploration can be directly applied to full-scale applications. These workloads assess how CPU cores, SMT implementation, and thread placements affect performance variability, offering insights applicable to real-world parallel computing. All workloads were tested on the five platforms, except *taskbench*, *arraybench* and *lulesh* on A64FX, due to expired access.

3 Experimental Design

We design our benchmarking methodology to characterize performance variability while ensuring reproducible, unbiased measurements. Experiments were conducted in isolation on a single node within production, site-managed clusters, without privileged access to modify the execution environment. Instead of trace analysis, we ran each experiment 10 times under identical conditions to account for variability and apply statistical analysis to ensure the robustness of our results. We collected execution times across 10 runs

Table 1: Parameters of the hardware platforms

Platforms	AMD Zen2	Intel AlderLake	Intel Xeon	Fujitsu A64FX	Graviton3
Architecture	x86_64	x86_64	x86_64	ARM aarch64	ARM aarch64
CPU model	AMD EPYC Zen2	Intel i9-12900K	Intel Xeon Gold 6130	Fujitsu A64FX	ARM Graviton3
Number of CPU cores	128, 2-way SMT	8, 2-way HT	32, no SMT	48, no SMT	64, no SMT
Number of NUMA nodes	8	1	2	4	1
gcc version	gcc v12.2.0	gcc v11.4.0	gcc v12.2.0	gcc v11.1	gcc v12.3.0
llvm/clang version	clang v15.0.6	clang v14.0.0	clang v15.0.6	clang v16.0.6	armclang (llvm v17.0.0)

and analyze variations and the trends to derive reliable conclusions about performance stability.

3.1 Thread-pinning

We use three OpenMP environment variables to control thread placement: `OMP_NUM_THREADS`, `OMP_PLACES` and `OMP_PROC_BIND`. `OMP_NUM_THREADS` defines the number of threads. `OMP_PLACES` defines specific cores or hardware threads for OpenMP threads to be placed. `OMP_PROC_BIND` determines the thread pinning policy, specifying how threads are assigned to the places. Used together, these variables enable fine-grained control over thread affinity. We set `OMP_PROC_BIND` as *close* to ensure that the threads remain near their parent thread's location [9, 22].

3.2 Using Simultaneous Multithreading (SMT)

Among the platforms tested, only AMD Zen2 and AlderLake feature SMT. To assess its impact on performance variability, we design three experiment configurations: **(i) ST** - using only the first hardware thread per core, reserving the second thread for OS activities to absorb OS noise and shield user workload executions from system interference; **(ii) MT** - Using both hardware threads per core for user workloads; **(iii) STb** - Like ST, but with the second thread **b**usy running a background task during user workload executions.

3.3 Frequency logging on a dedicated core

To analyze the impact of frequency scaling on performance variability, we collected core frequencies during experiments, using a Python script running on a dedicated core. The script accessed the Linux CPUFreq sysfs interface to track frequency levels across all cores. No other applications were executed concurrently except for the specified ones, minimizing interference from the frequency logger and background activities, and ensuring minimal impact on benchmark execution.

4 Experimental Results

This section analyzes experimental findings, focusing on factors contributing to performance variability and corresponding mitigation strategies. Each runtime configuration was tested 10 times to capture execution time variability, along with any variability inherent in EPCC micro-benchmarks, which performed 30 repetitions. Due to space limitations, we highlight representative results, emphasizing key impacts on variability. To quantify this, we report the average (Avg.) execution time over 10 runs. Execution time variability is assessed by normalizing the minimum (min.) and the maximum (max.) execution times against the average time of each run for the benchmarks reporting both. This allows us to examine how the normalized minimum and maximum execution times (Norm. exe. time) fluctuate relative to the average execution time in each run. We report compiler results default to gcc, unless notable differences arise. In all boxplots, the red line indicates the *median*, and the blue line indicates the *mean*. We first evaluate two commonly used mitigation strategies. We then provide our specific observations on performance variability and offer corresponding mitigation suggestions.

4.1 The commonly used mitigation methods

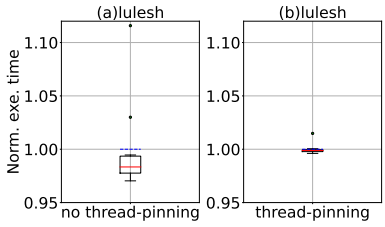


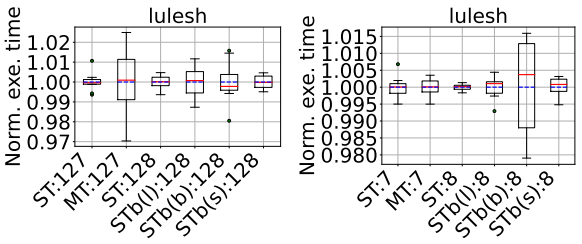
Figure 1: Lower variability after thread-pinning for *lulesh* on AlderLake running with 8 threads, using gcc for compilation.

4.1.1 The effect of thread-pinning. We first compare results with and without thread-pinning, conducted in the ST configuration, as described in Section 3. Overall, thread-pinning significantly improves performance stability across all platforms, except Graviton3. For example, on AlderLake (disabling 8 E-cores), Figure 1 shows that letting the OS to manage thread placement (left subfigure) by default results in substantial execution time variability due to dynamic thread migration. OS-level workload balancing aims to optimize core usage by moving threads between cores during execution, but this can lead to instability if application characteristics and thread placement policies are misaligned. In contrast, thread-pinning (right subfigure) mitigates this variability effectively. Taking the standard deviation as a metric (as in the rest of this paper), performance variability is reduced by 87.8% after thread-pinning. Similarly, improvements were observed across other benchmarks and the mini-app, though the extent of improvement varied. For Graviton3, thread-pinning had negligible impact, likely due to the virtualized cloud environment, where the cloud hypervisor (Nitro) abstracts CPU core management and presents vCPUs to the virtual machine, making thread-pinning either ineffective or unnecessary. For consistency, thread-pinning was applied in all subsequent experiments.

4.1.2 Reserving additional hardware resources for OS using SMT. Previous studies suggest that reserving additional hardware cores or threads for OS activities can reduce performance variability.

Table 2: Avg. execution time (s) of *lulesh* in Figure 1

Platforms	gcc		clang	
	pin	no pin	pin	no pin
AlderLake	11.201	11.474	12.440	14.807



(a) AMD Zen2 **(b) AlderLake**

Figure 2: Higher variability for *lulesh* due to SMT and co-running processes, using gcc for compilation.

Table 3: Avg. execution time (s) of *lulesh* in Figure 2

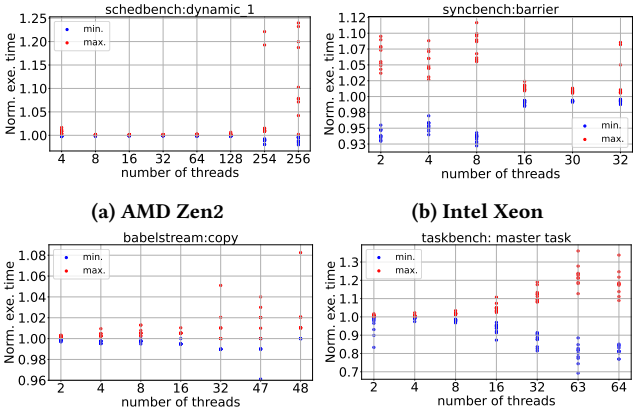
AMD	ST:127	MT:127	ST:128	STb(l)	STb(b)	STb(s)
Zen2	9.082	17.232	8.999	12.777	17.049	9.219
Alder	ST:7	MT:7	ST:8	STb(l)	STb(b)	STb(s)
Lake	11.769	11.387	14.415	18.374	33.542	15.865

Our experiments on AMD Zen2 and AlderLake, the only platforms supporting SMT, confirmed this. To explore this further, we tested how using these threads for user applications or background tasks affects performance stability. We used *lulesh* (as the user application with compute- and memory-bound kernels) to illustrate our findings in Figures 2. The first hardware thread of each physical cores was dedicated to the user application, while the second thread was either reserved for the OS (ST), assigned to the user application (MT), or busy with background tasks: STb(l) (running *lulesh*), STb(b) (running *BabaelStream*), or STb(s) (running *schedbench*). The number after the colon in Figures 2 denotes the total number of used physical cores.

When comparing the ST and MT tests, the ST configuration achieves better performance stability by reserving the second thread per core for OS activities, thereby absorbing OS noise, especially on AMD Zen2 (73.7% of variability is reduced in Figure 2a). Conversely, the MT configuration, which uses both hardware threads for the user application, negatively impacts performance stability. We also explored the impact of co-running processes on performance variability. As expected, the ST configuration outperforms all STb configurations in terms of stability, as shown in the right four columns of each subfigure in Figure 2. The STb configurations show some performance perturbations, primarily due to interference from OS activities and co-running processes on the same cores as the user application (even not on the same OpenMP places). Among the STb cases, STb(l) and STb(b) exhibit the highest variability, potentially caused by memory contention, as both background tasks are memory-bound.

While the two methods above effectively reduce variability, they do not eliminate it entirely. We continue to explore other factors that contribute to variability and examine how the studied effects interact to reduce performance variability (answering Q3), while also introducing additional mitigation strategies.

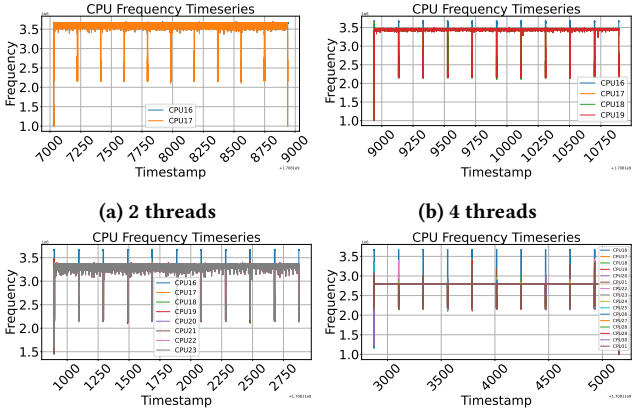
4.2 The effects of OpenMP scalability and platform characteristics



(a) AMD Zen2 **(b) Intel Xeon**

(c) A64FX **(d) Graviton3**

Figure 3: Scalability of variability for *schedbench*, *syncbench*, *babelstream* and *taskbench* as the number of threads increases, using gcc for compilation.



(a) 2 threads **(b) 4 threads**

(c) 8 threads **(d) 16 threads**

Figure 4: CPU frequencies of cores used to run *syncbench* in Figure 3b on Intel Xeon.

This experiment evaluates the impact of OpenMP scalability and explores how increasing thread counts influence performance stability in Figure 3. All tests were performed in ST configuration, except for AMD Zen2 when the thread count exceeded 128. As expected, increasing thread count generally adds to execution time variability across all tested platforms except AlderLake. This effect is particularly pronounced at higher thread count (> 128 on AMD Zen2 in Figure 3a, > 30 on Intel Xeon in Figure 3b, ≥ 32 on A64FX

Table 4: Avg. execution time in Figure 3

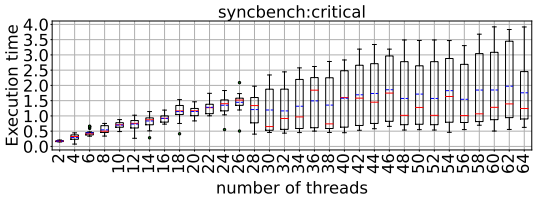
	thread count/execution time		
AMD Zen2(ms)	4/8/16	32/64/128	254/256
	126.6/125.0/125.2	124.5/129.7/122.5	137.3/134.5
Intel Xeon(µs)	2/4/8	16/30/32	-
	0.81/0.85/0.86	1.14/6.71/2.45	-
A64FX (s)	2/4/8	16/32/47	48
	∼0.0065	∼0.0065	0.00097
Gravi ton3(µs)	2/4/8	16/32/63	64
	2.0/4.0/15.4	80.8/670.1/3230.4	3570.8

in Figure 3c, and ≥ 32 on Graviton3 in Figure 3d). Notably, when utilizing all cores/hardware threads for benchmark execution, e.g. 256 threads (128 physical cores) on AMD Zen2, 32 threads on Intel Xeon and 64 threads on Graviton3, performance stability deteriorates markedly. This instability stems from resource depletion solely for user programs, leaving no capacity for OS activities, which interferes with benchmark execution. Recent discussions on resource isolation in [6, 8] echo these findings. To mitigate this interference, we spared at least one core/hardware thread for the OS during subsequent tests. On A64FX, the performance variability increased slightly with higher thread counts (e.g. 47 or 48 threads), as shown in Figure 3c. However, this variability is less pronounced compared to other platforms, likely due to 12 off-line cores reserved for system operations. On AlderLake, we did not observe a significant increase in variability. This is due to its limited core count (8 P-cores with all E-cores disabled during this experiment to minimize the impact of the hybrid architecture).

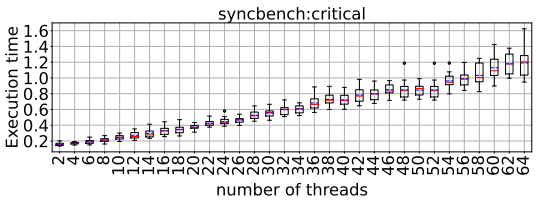
Figure 3 also reveals irregular results that challenge the above conclusion. On AMD Zen2, we observed relatively high variability with 4 threads (Figures 3a) compared to 8 threads, while on Intel Xeon, relatively high variability occurred with 2, 4, and 8 threads compared to 16 threads (Figures 3b). This variability is attributed to frequency variation. For example, Figure 4 shows the core frequencies during *syncbench* test on Intel Xeon, revealing higher frequency fluctuations with 2 (Figure 4a), 4 (Figure 4b) and 8 (Figure 4c) threads, but not with 16 (Figure 4d) threads. Performance fluctuations on both AMD Zen2 and Intel Xeon likely stem from the Turbo mechanism, which temporarily increases clock speeds beyond the base frequency when operating below the Thermal Design Power (TDP) and thus introduces variability due to dynamic clock speed adjustments. With fewer active cores, reduced power consumption and thermal output enable higher core frequencies, contributing to these fluctuations. We do not observe frequency variation on A64FX, and we lack access to core frequency data for Graviton3 due to system limitations.

4.3 The effect of OpenMP implementation

When running *syncbench* on Graviton3 compiled with gcc and setting `OMP_WAIT_POLICY` to *passive*, we noticed significant execution time variability in two synchronization kernels, `critical` and `lock/unlock`, as the thread count increased. Using `critical` as an example, shown in Figure 5, switching `OMP_WAIT_POLICY` from *passive* (Figure 5a) to *active* (Figure 5b) dramatically reduced this variability (86.5% on average and up to 95.3%). This indicates how different algorithms in the OpenMP runtime can affect the execution time variability. The `OMP_WAIT_POLICY` environment variable



(a) `OMP_WAIT_POLICY`=passive



(b) `OMP_WAIT_POLICY`=active

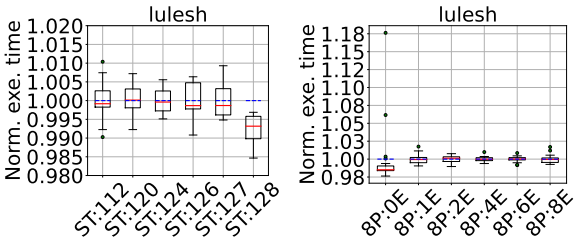
Figure 5: Setting `OMP_WAIT_POLICY` to *active* when compiling *syncbench* (`critical`) with gcc on Graviton3 reduces the variability.

hints at the OpenMP implementation about the desired behavior of waiting threads. For libgomp (the gcc OpenMP runtime library), the *passive* policy instructs waiting threads to immediately yield the CPU upon failing to acquire a lock, allowing the OS to schedule other processes. This leads to excessive voluntary context switches, which introduce direct overhead and cause inconsistent wake-up times in thread resumption, further exacerbating performance variability in synchronization scenarios. In contrast, the *active* policy directs waiting threads to spin for a specified duration before yielding the CPU. This busy-waiting mechanism reduced OS preemption and eliminates excessive (voluntary) context switches, thereby reducing variability in Figure 5b.

However, the observed variability largely disappeared at a certain point during our experimental analysis. We speculate that this was due to an update to the system, which was probably causing the problem. However, this remains a hypothesis, as there is no way to analyze it further. Interestingly, we did not observe the same behavior on other platforms, or when using the armclang compiler on Graviton3. Therefore, we conclude that the observed effect is likely a combination of synchronization algorithms and platform-specific characteristics.

4.4 The effect of resource reservation for the OS

Reserving physical core resources for OS activities can often yield a notable reduction in performance variability. However, determining the optimal amount of reserved resources is crucial to avoid core over-reservation, which can lead to unnecessary resource wastage. To address this, we conducted a case study on AMD Zen2 to explore the impact of adjusting the number of reserved cores (Q1). Additionally, for hybrid architectures such as Intel AlderLake, which features both P- and E-cores, we performed a case study to identify the optimal number of enabled E-cores (Q2).



(a) AMD Zen2 **(b) AlderLake**

Figure 6: Variability in *lulesh* with varying reserved core counts on AMD Zen2 and varying enabled E-core counts on AlderLake, using gcc for compilation.

4.4.1 Avoiding core over-reservation with SMT on AMD Zen2. As we do not have privileged access to fine-tune the execution environment on AMD Zen2, and enabling SMT does not negatively affect performance stability as long as the additional hardware threads are reserved for the OS (as discussed earlier), we kept SMT enabled by default on AMD Zen2 and used only one thread of the cores at most, i.e. ST configuration, to run the user application. Performance variability behavior when reducing the number of reserved physical cores from 16 (ST:112) to 0 (ST:128) is shown in Figure 6a, after removing an outlier for the ST:128 test. For AMD Zen2 platform examined in this study, we find that reserving 4 physical cores (ST:124) may achieve the best performance stability without significant loss of absolute performance (seen in Table 5), effectively addressing question Q1. This observation suggests that core resources can be utilized more efficiently especially in large-scale multicore systems, while avoiding core over-reservation such as 16 (ST:112) and 8 (ST:120) cores in this experiment.

4.4.2 Determining the optimal number of enabled E-cores on AlderLake. In the earlier experiments, all E-cores were disabled to eliminate the impact of the asymmetric architecture on AlderLake. In this experiment, we investigate the execution time variability for *lulesh* with varying E-core counts, as shown in Figure 6b, to evaluate the performance implications of enabling E-core in this hybrid architecture. In Figure 6b, we used the 8 P-cores to run the application while enabling 0 (8P:0E), 1 (8P:1E), 2 (8P:2E), 4 (8P:4E), 6 (8P:6E) and 8 (8P:8E) E-cores each at boot time. We observe that enabling some E-cores can reduce the variations in execution time, while also slightly reduce execution time (seen in Table 5). For example, even with 1 E-core (8P:1E), the results were more stable compared to enabling no E-core (8P:0E) (85.2% of variability is reduced). For AlderLake tested in this study, enabling 2 (8P:2E) or 4 (8P:4E) E-cores impressively mitigated the execution time variability (as well as slightly improved absolute performance), effectively answering question Q2. Through experimenting with different E-core configurations, we aim to understand the trade-offs between high performance and system efficiency, providing insights into optimal core configurations for hybrid architectures.

Table 5: Avg. execution time (s) of *lulesh* in Figure 6

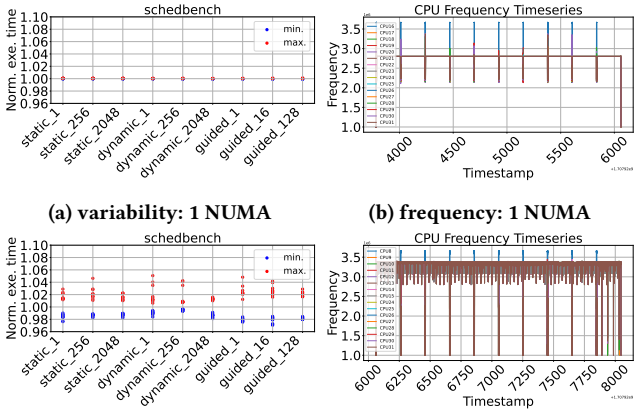
AMD	ST:112	ST:120	ST:124	ST:126	ST:127	ST:128
Zen2	7.67	7.87	7.97	7.99	8.03	8.23
Alder	8P0E	8P1E	8P2E	8P4E	8P6E	8P8E
Lake	10.21	10.02	9.98	9.95	9.94	9.93

Table 6: Avg. execution time (µs) of *schedbench* (x represents all the values in x-axis) in Figure 7

	static_x	dynamic_x	guided_x
1 NUMA	~163615	~163463	~163908
2 NUMA	~143215	~141634	~141994

4.5 The effect of frequency scaling

In addition to performance variability caused by frequency fluctuations discussed in Section 4.2, we observed greater variability when the same thread count was launched across multiple NUMA nodes (≥ 5 NUMA nodes on AMD Zen2 and > 1 NUMA node on Intel Xeon). This increased variability is largely attributed to frequency scaling, which is beyond the user's control. For example, on Intel Xeon with two NUMA nodes, Figures 7a and 7c compare performance variability for *schedbench* using 16 threads on a single NUMA node versus across both nodes. When both NUMA nodes were used, the variability increased significantly (Figures 7c), corresponding to greater frequency fluctuations (Figure 7d). Notably, performance variability on Intel Xeon (0.96 ~ 1.10 in Figure 7c) is broader on average than on AMD Zen2 (0.994 ~ 1.005, data not shown), aligning with higher frequency fluctuations recorded on Intel Xeon (Figure 7d) compared to AMD Zen2 (data not shown).



(a) variability: 1 NUMA **(b) frequency: 1 NUMA**

(c) variability: 2 NUMA **(d) frequency: 2 NUMA**

Figure 7: Variability in *schedbench* and the corresponding core frequencies on Intel Xeon, when launching 16 thread on a single NUMA node versus across both NUMA nodes.

4.6 The effect of mitigation strategies on absolute execution time

This paper focuses primarily on performance variability and the effectiveness of mitigation strategies. Our proposed mitigation methods further enhance performance stability by exploring the interplay of effects studied together, effectively answering Q3, while

we also briefly summarize their impact on absolute performance (execution time). Thread-pinning significantly reduces variability (Figure 1), but does not consistently improve or degrade execution time, as its effect depends on the experiment configuration (Table 2 shows the reduction in execution time for *lulesh*), including thread count, platform, compiler, and application characteristics. Allocating additional hardware resources to the OS via SMT generally enhances both performance (reduced execution time) and stability, seen in Figure 2 and Table 3. On Intel Xeon and Graviton3, increasing thread counts tends to negatively impact both execution time and variability, shown in Figure 3 and Table 4. Reducing reserved cores for the OS slightly increases execution time (Table 5), the optimal number of reserved cores to achieve best performance stability depends on the platform and workload (Figure 6). Lastly, frequency fluctuation introduces variability (Figure 7), although a higher frequency generally reduces execution time (Table 6).

5 Recommendations

Based on our findings, in addition to common recommendations such as thread-pinning, core resource reservation, and addressing frequency fluctuations (which is beyond our control), we propose the following strategies to mitigate execution time variability for OpenMP-based applications on multicore systems.

Avoid core over-reservation: Considering Q1, although core reservation reduces performance variability, excessive reservation of CPU cores does not provide additional benefits and can lead to inefficient resource utilization.

Avoid co-running processes: Avoiding co-running processes when executing user programs on platforms with SMT is important for maintaining performance stability. Even background codes executed in different threads than user programs can interfere with their executions.

Enable appropriate E-cores in hybrid architectures: Considering Q2, in hybrid architectures such as Intel AlderLake, enabling an optimal number of E-cores while keeping P-cores fully engaged for critical user workloads allows the system to efficiently manage background tasks without significantly impacting the performance of user workloads.

Best practices for performance stability in multicore systems: We recommend not to use too few or the full number of cores in the node. This strategy reduces variability by avoiding (a) core boosting (too few active cores) and (b) frequency reduction (too many cores leading to thermal issues).

6 Related work

As modern multicore systems incorporate increasing number of cores, factors such as OS noise and frequency variations amplify performance variability, particularly in parallel computing environments. Studying these effects and developing mitigation strategies are essential. This section reviews current research efforts aimed at reducing performance variability or providing insight into its mitigation.

Several studies have explored thread affinity mechanism to enhance performance stability of parallel applications on multicore machines [10, 18–20]. The authors in [18] demonstrated that OpenMP applications experience high execution time variability when thread placement is managed by the OS, whether running alone or in parallel with other independent processes. The later works compared multiple thread pinning strategies on three multicore machines [19] and proposed dynamic thread-pinning policy for phase-based OpenMP programs [20]. However, these results are largely limited to systems with a small scale of cores, with most tests conducted on machines with up to 8 cores, except for one study involving 96 cores. Recent work [10] introduced a practical OpenMP runtime system supporting both flat and nested parallelism. While thread-to-CPU binding was evaluated in terms of execution times, performance variability was not a primary focus. Explicitly specifying the affinity setting is generally recommended. Core source reservation techniques have also been proposed to reduce performance variability by isolating user applications from OS functions. For instance, the core specialization feature [24], implemented on Cray XE systems, reserved one or more cores for OS and service threads. Similarly, the LUMI supercomputer reserved the first core of each core compute complex (CCX) to operate in low-noise mode [1].

On systems with SMT, performance variation often arises when both threads of the physical cores are utilized. Studies have observed worse variability when running parallel applications on both threads per core compared to using a single thread per core [14, 16]. Tuning core frequencies has been extensively studied as a means of managing performance variability. Dynamic voltage scaling, for example, has been shown to induce execution time variability [17]. The authors established a correlation between performance variability and core operating frequency [16], observing that hardware-enforced power limits further magnified these effects. Even in thermally stable states, frequency variation can cause significant performance variability [23]. OS noise in NUMA architectures could exacerbate run-to-run performance variability, as highlighted in [25]. The authors proposed a performance-stable NUMA management scheme to improve performance stability.

7 Conclusion

This study investigates performance variability in parallel applications using OpenMP through experiments on five OpenMP benchmarks and a real-world mini-app, compiled using gcc and llvm/-clang, across four production clusters and a local workstation. Our findings reveal the complex interplay of factors influencing performance variability, including thread-pinning, OpenMP runtime implementations, SMT, core resource reservation, frequency fluctuations, and hardware-specific characteristics like E-cores usage in a hybrid architecture. We demonstrated effective mitigation strategies, such as applying thread-pinning, reserving core resources for OS activities to absorb noise while avoiding core over-reservation, and optimizing E-core usage in hybrid architectures. However, some performance perturbations, particularly frequency fluctuations, remain beyond our control. Future work will extend this evaluation to include more benchmarks, larger-scale parallel programs, and other programming models. We also aim to analyze OS noise sources and explore advanced methods to further mitigate or eliminate performance variability.

Acknowledgments

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No. 800928 and Specific Grant Agreement No. 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme, and from the Swedish Research Council, among others. Additionally, this work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048. The computations were enabled by resources provided by Chalmers e-Commons at Chalmers. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The authors thank the Barcelona Supercomputing Center, Spain for providing access to Mare Nostrum 4 ARM-CTE partition.

References

[1] Andrey Alekseenko, Szilárd Páll, and Erik Lindahl. 2024. GROMACS on AMD GPU-Based HPC Platforms: Using SYCL for Performance and Portability. *ArXiv* abs/2405.01420 (2024). https://api.semanticscholar.org/CorpusID:269502644

[2] Roberto Camacho Barranco and Patricia J. Teller. 2016. Analysis of the Execution Time Variation of OpenMP-based Applications on the Intel Xeon Phi. https://api.semanticscholar.org/CorpusID:1550490

[3] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. 2013. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[4] J. Mark Bull. 1999. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *In Proceedings of First European Workshop on OpenMP*. 99–105. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.8780

[5] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-run Variability on Xeon Phi based Cray XC Systems. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[6] Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. 2023. Operating System Noise in the Linux Kernel. *IEEE Trans. Comput.* 72, 1 (2023), 196–207.

[7] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream. *Int. J. Comput. Sci. Eng.* 17, 3 (jan 2018), 247–262.

[8] Balazs Gerofi, Kohei Tarumizu, Lei Zhang, and all. 2021. Linux vs. Lightweight Multi-Kernels for High Performance Computing: Experiences at Pre-Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. New York, NY, USA.

[9] HPCWiki. 2022. Binding/Pinning. https://hpc-wiki.info/hpc/Binding/Pinning.

[10] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. 2019. BOLT: Optimizing OpenMP parallel regions with user-level threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 29–42.

[11] Ian Karlin. 2012. LULESH Programming Model and Performance Ports Overview. https://api.semanticscholar.org/CorpusID:58454153

[12] Ian Karlin, James R. McGraw, Jeff Keasler, and Bert Still. 2013. Tuning the LULESH Mini-app for Current and Future Hardware. https://api.semanticscholar.org/CorpusID:64395756

[13] James LaGrone, Ayodunni Aribuki, and Barbara Mary Chapman. 2011. A Set of Microbenchmarks for Measuring OpenMP Task Overheads. https://api.semanticscholar.org/CorpusID:8375037

[14] Edgar A Leon, Ian Karlin, and Adam T Moody. 2016. System noise revisited: Enabling application scalability and reproducibility with SMT. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 596–607.

[15] LUMI. 2023. The low-noise mode on LUMI-G. https://lumi-supercomputer.github.io/LUMI-training-materials/User-Updates/Update-202308/lumig-lownoise/.

[16] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. 2017. An Empirical Survey of Performance and Energy Efficiency Variation on Intel Processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing* (Denver, CO, USA) *(E2SC'17)*. Association for Computing Machinery, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/3149412.3149421

[17] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. 2010. Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study. In *Research Report*. inria 00514548v2.

[18] Abdelhafid Mazouz, Sid Ahmed Ali Touati, and Denis Barthou. 2011. Analysing the Variability of OpenMP Programs Performances on Multicore Architectures. https://api.semanticscholar.org/CorpusID:55852616

[19] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. 2011. 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*. 273–279. https://doi.org/10.1109/HPCSim.2011.5999834

[20] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. 2013. Dynamic Thread Pinning for Phase-Based OpenMP Programs. In *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–64.

[21] Aroon Nataraj, Alan Morris, Allen D. Malony, Matthew Sottile, and Pete Beckman. 2007. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Reno, Nevada) *(SC '07)*. Association for Computing Machinery, New York, NY, USA, Article 29, 12 pages. https://doi.org/10.1145/1362622.1362662

[22] OpenMP. 2018. OpenMP-API-Specification. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[23] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, and Wei Wang. 2013. Openmp and mpi application energy measurement variation. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*. 1–8.

[24] Howard Porter Jr. Pritchard, Duncan Roweth, Dave Henseler, and Paul Cassella. 2012. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. https://api.semanticscholar.org/CorpusID:53474239

[25] Jaehyun Song, Minwoo Ahn, Gyusun Lee, Euiseong Seo, and Jinkyu Jeong. 2021. A Performance-Stable NUMA Management Scheme for Linux-Based HPC Systems. 9 (2021), 52987–53002.

[26] Pengyu Wang, Wanrong Gao, Jianbin Fang, Chun Huang, and Zheng Wang. 2021. Characterizing OpenMP Synchronization Implementations on ARMv8 Multi-Cores. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 669–676.

[27] Li Xu, Thomas Lux, Tyler Chang, and all. 2021. Prediction of high-performance computing input/output variability and its application to optimization for system configurations. *Quality Engineering* 33, 2 (2021), 318–334.