Analysis and Characterization of Performance Variability for OpenMP Runtime

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

Nikela Papadopoulou
Chalmers University of Technology
Gothenburg, Sweden
nikela@chalmers.se

Miquel Pericàs
Chalmers University of Technology
Gothenburg, Sweden
miquelp@chalmers.se

arXiv:2311.05267v1 [cs.DC] 9 Nov 2023

ABSTRACT

In the high performance computing (HPC) domain, performance variability is a major scalability issue for parallel computing applications with heavy synchronization and communication. In this paper, we present an experimental performance analysis of OpenMP benchmarks regarding the variation of execution time, and determine the potential factors causing performance variability. Our work offers some understanding of performance distributions and directions for future work on how to mitigate variability for OpenMP-based applications. Two representative OpenMP benchmarks from the EPCC OpenMP micro-benchmark suite and BabelStream are run across two x86 multicore platforms featuring up to 256 threads. From the obtained results, we characterize and explain the execution time variability as a function of thread-pinning, simultaneous multithreading (SMT) and core frequency variation.

KEYWORDS

performance variability, OpenMP, parallel computing, thread-pinning, simultaneous multithreading

1 INTRODUCTION

Parallel applications executing on shared-memory systems in the HPC world usually follow the single-program multiple-data (SPMD) model, typically implemented with OpenMP. OpenMP, the de facto programming model for SPMD, spawns multiple threads when encountering a `#pragma omp parallel` clause. Each thread is then executed on one core/hardware thread of the system to execute the parallel region, and commonly all threads synchronize at the end of the execution of the parallel region to compute the final result. Some system-specific activities, such as operating system (OS) daemons and interrupt processing, can cause preemption or interrupt handling to one or multiple of the threads, causing the execution of the parallel work to be delayed and the execution time to be dominantly determined by the slowest thread, while the others wait for synchronization, leading to a waste of resources like time and energy. Also, due to the randomness of the delay, it will in turn generate performance variability for the runtime of the parallel application. Performance variability has become an important limiter to the scalability in parallel computing [13]. With the complexity of modern hardware architecture features increasing, variability has become an increasingly challenging issue for improving the efficiency of parallel computing [19].

Performance variability or run-to-run variations of applications owing to multiple components in the system can become an obstacle for the development of parallel applications in several ways, like performance debugging or quantifying the effects of system software and compilers changes [2]. There have been various efforts to identify the potential causes of variability and in turn find solutions to reduce the possibility of variability occurrence, aiming at obtaining performance stability of parallel application executions. Most studies on variability have focused on MPI [9, 11, 16, 23], as the explicit and synchronizing nature of message passing communication, and the large scale of applications using this programming model, make MPI applications more sensitive to noise, which in turn leads to load imbalance and ultimately performance degradation. Evaluating the impact of noise and the occurring performance variability in shared memory models such as OpenMP has received comparatively less attention. However, as the core count of modern CPUs increases, shared memory parallel applications using OpenMP are likely to be also impacted by OS noise.

Several strategies to optimize the performance of parallel programs with OpenMP have been proposed, studied, and have influenced the state-of-practice in execution. A common strategy is thread pinning [30], which can improve application performance by keeping threads bound to a specific core and avoiding expensive memory accesses. The authors in [21] have studied several thread-pinning strategies to improve the performance of OpenMP programs, while in a later work [22], they have proposed dynamic thread-pinning for phase-based OpenMP programs with multiple parallel regions. Their study is limited to a small scale of core/thread counts. However, they identified thread pinning as a critical factor to performance variability [20]. The effective usage of simultaneous multithreading, the architectural mechanism that supports several hardware threads per physical core, has also been shown to improve the performance of MPI and MPI+OpenMP applications [16]. Finally, tuning the core frequencies for performance is also important, and well-studied in the literature relevant to developing dynamic energy-efficiency techniques [4, 17, 18], as even in steady-state, frequency variation can cause high variability of the performance [25].

This work focuses on characterizing the performance variability of OpenMP on modern CPUs. Motivated by the scale of recent, modern multi-core systems, we conduct an extensive study of the impact of common performance-optimizing strategies on the performance of OpenMP applications, in an

effort to further understand and pinpoint sources of performance variability in OpenMP. We use two micro-benchmarks from the EPCC benchmark suite [14] which focus on the performance of common OpenMP constructs, such as `parallel for` and synchronization, and the BabelStream benchmark [7], which assesses the memory bandwidth, and execute them on two different systems, using different variability-reducing strategies. In particular, we analyze thread-pinning, which can help in revealing the performance degradation related to unbound threads in parallel applications. We additionally explore simultaneous multithreading to show how it can affect the performance of OpenMP benchmark executions. Finally, during benchmark execution, we record the frequencies of all cores, to examine whether frequency variation exists, and how it can affect the variability of execution time.

Our study is conducted on two production clusters hosted by two different academic institutions. We do not have privileged access to these systems and therefore cannot control the node setup or operating system knobs and we cannot trace kernel-level events. We, therefore, rely on a statistical analysis of the observed execution times, repeatedly running every benchmark with multiple iterations of the kernels of interest. By studying the possible sources and characterizing the performance variability, we can categorize the sources of performance variability and find efficient solutions to mitigate a particular class of variability in future work.

The rest of the paper is organized as follows. We introduce related work in Section 2. Section 3 provides an overview of the proposed methodology to characterize the performance variability. We present our experimental setup in Section 4 and experimental results in Section 5. A conclusion of this work follows in Section 6.

2 RELATED WORK

Performance variability of parallel applications has been well reported on modern systems in multiple works. At the extra-application level, there can be multiple reasons for unpredictable performance, with operating system noise (also referred to as *OS jitter*) being one of the most common reasons. Several works [6, 9, 23, 28] study the impact of operating system activities on performance, looking primarily at large-scale parallel applications with MPI. A recent work [27] demonstrates that OS noise on non-uniform memory access (NUMA) architectures can cause high run-to-run performance variability. As the number of cores/processors on modern systems grows, OS noise can become a more significant factor of performance variability, as a small amount of perturbation can be greatly amplified in parallel computing. It is therefore important to study the impact of OS noise on performance variability and find solutions to mitigate it.

Aside from the operating system, performance variability can arise from contention and interference on shared resources. Bhatele et al. [2] show that sharing network resources on HPC systems is a primary source of performance variability. Xu et al. [31] show that interference on the I/O subsystem affects the performance of parallel applications. On systems with simultaneous multithreading, performance degradation can occur from oversubscription of the physical cores [16]. Another source of variability is manufacturing variability [11], which leads to performance heterogeneity. The power variation from manufacturing variability can affect the performance stability of HPC applications, as it translates to CPU frequency variation [26].

As there is increasing evidence for performance variability of parallel applications, several techniques and tools have been proposed to measure and characterize performance variability in recent works. In particular, for OS noise, Pradipta et al. [5] develop a tool to monitor and evaluate the impact of OS noise on Linux-based systems through fine-grained kernel instrumentation. Gioiosa et al. [10] extend *Oprofile*, a Linux kernel-level tool to characterize the sources of OS noise. Morari et al. [23] extend the Linux tool *LTTng* to build *LTTng-Noise*, a tracing tool. De Oliveira et al. [6] develop the *osnoise* tracer, which analyzes noise activities via kernel instrumentation. A more generic technique to measure performance variability and statistically characterize performance distributions has been proposed by Kocoloski et al. [13], to assist in system parameter design such as power-capping.

A limited number of works have focused on analyzing the performance variability of OpenMP programs. Camacho et al. [1] show that thread binding can reduce execution time variation in OpenMP applications, and Mazouz et al. [20] study the effects of thread binding, OS jitter, and hardware-related sources (memory-access related sources, concurrent jobs, asymmetry between cores, dynamic voltage scaling and device temperature) on execution time variation in OpenMP.In our work, we also focus on analyzing and characterizing performance variability in OpenMP. We exclude interference from other applications and run our benchmarks in isolation. Additionally, as we do not have privileged access to the platforms in study, we exclude operating system knobs from our techniques and only observe the impact of operating system noise on OpenMP, with a statistical analysis of results.

3 METHODOLOGY

In this section, we describe the methodology followed to characterize performance variability in OpenMP. We note that we always execute benchmarks in isolation on a single node, eliminating the case of variability from application interference. We perform our experiments on production, site-managed clusters, therefore we do not have privileged access that would allow us to tune the execution environment. Instead of detailed trace analysis, we rely on multiple experiments and statistical analysis of the results. The following paragraphs describe the strategies we apply to detect the sources and impact of performance variability.

Thread pinning: By default, we let the operating system decide the thread placement on cores, as the default setup for `OMP_PROC_BIND` is set to false. In this case (before thread-pinning), the threads may migrate between cores during the execution of parallel programs to improve work balance. In modern multi-core architectures, exploitation of locality is essential to efficiently run parallel programs [12]. OpenMP supports users with fine-grained thread affinity control through

thread pinning. A group of *places*, corresponding to a group of hardware threads, can be defined and OpenMP threads can be bound to specific places (therefore hardware threads) by setting a pinning policy. To achieve this, some OpenMP-related environment variables are used to specify the OpenMP settings in this paper, i.e., `OMP_NUM_THREADS` is used to define the number of threads, `OMP_PLACES` and `OMP_PROC_BIND` work together to pin each thread to a specific core. The thread affinity policy is set as *close*, which implies that worker threads are close to the main thread in contiguous partitions [24, 30].

Using Simultaneous Multithreading: Simultaneous multithreading mechanism is implemented on one of the two platforms included in our experimental setup, Dardel (see Section 4 for details), where each core has two hardware threads (also referred to as logical cores). We evaluate two configurations in our experiments. The first one is single-threaded, denoted as **ST**, in which at most one hardware thread per physical core is used to run the benchmark. In this case, the additional hardware thread of the core is reserved for operating system activities to absorb noise and isolate the benchmark running from the system interference. In the second configuration, both hardware threads of the core are utilized to run our benchmarks. We refer to this configuration as **MT**. We collect the results under the above two configurations and compare the performance variability of the execution of the benchmarks.

Frequency logging on a separate core: During the execution of benchmarks, a background Python script is run on a separate core to collect the frequencies of all cores. By doing this, we try to avoid interference from the frequency logger and benchmark running on the same core and guarantee that the execution of benchmarks is influenced as little as possible by other background activities. In the next section, we showcase the performance variability that can be related to the frequency variations.

4 EXPERIMENTAL SETUP

4.1 Hardware platforms

We use two different hardware platforms for our experiments. The first platform, *Dardel*, is an HPE Cray EX supercomputer located at the PDC Center for High-Performance Computing in Sweden. Each node of Dardel integrates two AMD EPYC Zen2 2.25GHz 64-core processors, accommodating two hardware threads per core. From the operating system's view, there are a total of 128 cores and 256 hardware threads/logical cores. The cores are organized in 8 NUMA domains of 16 cores each, with each socket behaving as a quad-NUMA domain. The maximum frequency of each core is 3.4GHz. The system runs the SUSE Linux Enterprise Server 15 SP3 OS, with Linux kernel version 5.3.18-150300.59.76_11.0.53-cray_shasta_c. We use `gcc v7.5.0` as the compiler.

The second platform, *Vera*, is a cluster located at C3SE Center for Scientific and Technical Computing at Chalmers University of Technology in Sweden. Each node of Vera integrates two Intel Xeon Gold 6130 2.1GHz 16-core processors, with a total of 32 cores. Each socket corresponds to a NUMA domain, with a total of 2 NUMA domains on the node. The maximum frequency of each core is 3.7GHz. The system runs Rocky Linux release 8.7, with Linux kernel version 4.18.0. We use `gcc v8.5.0` as the compiler.

Table 1: Parameters of the EPCC OpenMP micro-benchmarks

EPCC micro-benchmark	schedbench	syncbench
outer repetitions	100	100
delay time(μs)	15	0.1
test time(μs)	1000	1000
itersperthr	8192	-

4.2 OpenMP benchmarks

We use three different OpenMP benchmarks for our evaluation of performance variability in OpenMP. We draw two benchmarks from the EPCC OpenMP micro-benchmark suite [15, 29], one of the most comprehensive suites for OpenMP constructs, which provides measurements of the overhead incurred from an OpenMP construct by comparing the execution time of parallel code against this of serial code. We use *schedbench*, the benchmark focusing on the `parallel for` construct with different schedules, and *syncbench*, the benchmark which evaluates all the different available synchronization methods in OpenMP. The benchmarks can be run with different parameters. We present the parameters used for the two benchmarks in our evaluation in Table 1. The third benchmark is *BabelStream* [8], a common benchmark to measure memory bandwidth by executing simple vector kernels, including copy, add, multiplication, triad, and dot product. It has been used in previous work [11] to evaluate performance variability in a power-limited environment. We use the default parameters and an array size of 2^{25} for *BabelStream* in our evaluation.

We have executed a large set of experiments with the three benchmarks on the two hardware platforms described above. For every runtime configuration, we run each experiment 10 times, to collect run-to-run performance variability, in addition to any variability reported by the EPCC benchmarks themselves, which also execute 100 repetitions of each micro-benchmark. Due to page limitations, we only highlight those experimental results that show statistically significant performance variability and can shed light on the potential sources of this variability. In particular, we execute *schedbench* with three different schedules, namely `static`, `dynamic` and `guided` and various different chunk sizes [3], and present the results for specific schedules with the chunk size equal to 1. e.g., `static` or `dynamic` schedule with chunk size equal to 1, labelled as `static_1` and `dynamic_1` respectively. From *syncbench*, we select the `reduction` clause as the most representative of synchronization methods in OpenMP. For *BabelStream*, in every single run, we collect the minimum, average, and maximum execution time for each kernel and then normalize the minimum and maximum execution time to the average execution time. The run-to-run variations of execution time are depicted by comparing the normalized minimum and maximum execution times among 10 runs for every vector operation kernel respectively.

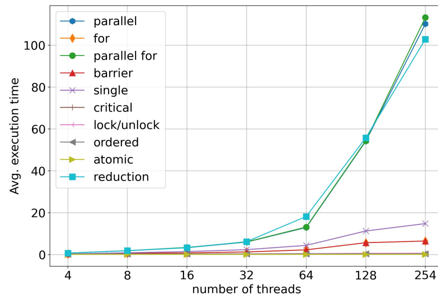
Table 2: Higher execution time (µs) for *schedbench* (`dynamic_1`).

run #	Dardel		Vera	
	4 threads	254 threads	4 threads	30 threads
1	124020.18	154277.48	136485.84	164672.22
2	124062.15	154162.74	136562.99	164642.42
3	123989.57	154159.44	136621.25	164662.94
4	123949.12	153999.99	136509.92	164665.88
5	124016.11	154206.72	136386.38	164652.30
6	123917.89	154044.95	136479.29	164573.27
7	123885.41	154222.61	136513.89	164699.42
8	123902.87	154182.23	136448.17	164754.22
9	123935.31	168835.06	136645.71	164717.21
10	124023.24	154065.79	136743.01	164757.37

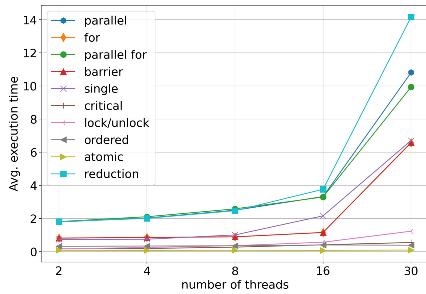
5 EXPERIMENTAL RESULTS

5.1 OpenMP scalability

We begin our evaluation by examining the scalability of the three OpenMP benchmarks, to understand the trend of the average (Avg.) execution time in Table 2 and Figures 1 and 2, and examine whether higher thread counts have higher performance variability in Figure 3. In Figure 3, the minimum and maximum execution times are normalized to the average execution time for each run respectively, and run each benchmark 10 times. We employ thread pinning for all the experiments, and make use of SMT, where available.



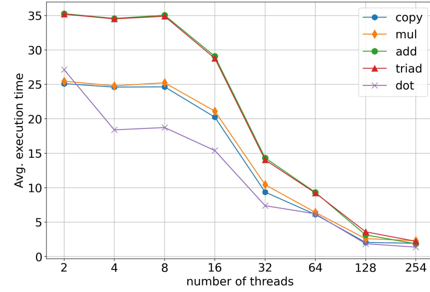
(a) 4-254 threads on Dardel



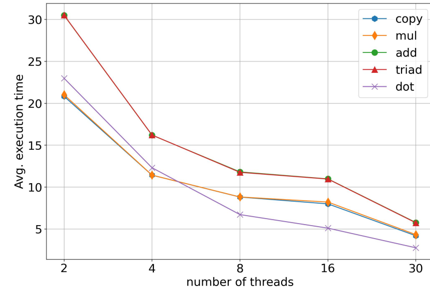
(b) 2-30 threads on Vera

Figure 1: Execution time (µs) when increasing the number of HW threads in *syncbench* on Dardel and Vera.

Regarding the scalability of execution time, we observe that the execution time increases as we spawn additional OpenMP threads, for *schedbench* in Table 2 and for *syncbench* in Figure 1, showing the average execution time for all 10 runs. For *syncbench*, we additionally observe a sharp increase in the execution time when we start utilizing the second socket on both systems (30 threads with 2 NUMA domains for Vera, and on 128 threads with 2 quad-NUMA domains for Dardel), as well as when we utilize the logical cores, in addition to the physical cores, on Dardel (254 threads). Also, we additionally highlight that the micro-benchmark corresponding to the `reduction` clause is the most time-consuming among the synchronization micro-benchmarks. We finally showcase the scalability of *BabelStream* in Figure 2, observing that the execution time of *BabelStream* reduces when launching more parallel threads, as expected, on both Dardel and Vera.



(a) 2-254 threads on Dardel



(b) 2-30 threads on Vera

Figure 2: Execution time (ms) for *BabelStream* when increasing the number of HW threads on Dardel and Vera.

Regarding the scalability of performance variability, higher thread counts add to performance variability for *syncbench* and *BabelStream* in Figure 3, especially when the thread count is high ($\geq$ 128 HW threads/OpenMP threads on Dardel and $\geq$ 30 on Vera), while it is not as pronounced for *schedbench*, as seen in the first column of Figure 3. It is worth pointing out that when all cores/HW threads were used for this scalability experiment, we observed a significantly worse performance behavior. To avoid this, on both systems, we spare 2 cores/HW threads, using 30 out of the 32 cores on Vera and 254 out of the 256 hardware threads on Dardel. We highlight that we observe both higher run-to-run variability, and also high variability between the 100 repetitions of the micro-benchmark for *schedbench* and *syncbench*. We argue that this is due to

operating system activities, which interfere with the benchmark execution when no spare cores/resources are left for them, causing more noise from the view of the user's application execution. This observation is in accordance with recent works [6, 9], which argue for resource isolation required for system activities, in order to reduce the OS noise.

5.2 The effect of thread pinning

In this part, we evaluate the effect of thread pinning on reducing performance variability. We showcase our results from Dardel in Figure 4, which is in accordance with our evaluation on Vera, although due to the larger scale of Dardel nodes, performance variability is more pronounced on this platform. The first column of Figure 4 shows average (Avg.) execution times of *schedbench* for 10 runs. The run-to-run variations of the average execution time do not completely disappear but can be reduced after the threads are pinned in Figure 4d, where only run # 9 has a higher execution time, compared to Figure 4a, where runs #(2,8,9,10) take longer time to finish. We believe that thread-pinning plays an important role in removing run-to-run variability and improving performance stability. The benefits of thread pinning are much more pronounced in the case of *syncbench*, as synchronization primitives are more susceptible to noise and even slight increases in the execution time of one thread can propagate throughout the operation. Figure 4b shows a high run-to-run variability for the reduction micro-benchmark of *syncbench*, on 128 physical cores of Dardel, resulting in more than 3 orders of magnitude of differences in the execution time of the micro-benchmark. Contrarily, after pinning, we achieve a much higher performance stability for the micro-benchmark, as shown in Figure 4e. We note that the y-axes of the two subfigures have different scales. The run-to-run variability is almost eliminated after pinning, while the execution time variations between the 100 repetitions of the benchmark are also largely reduced for certain runs, e.g. runs #(2, 3). Our observations for *BabelStream* are similar. Figure 4c shows high run-to-run variability for all the five kernels of the benchmark before thread pinning, as there is a difference of up to 6× between the minimum and maximum execution times between 10 different runs. After pinning, in Figure 4f, we observe less run-to-run variability, especially for the `copy` and `mul` kernels.

As load balancing at the OS-level can be affected by thread-pinning, it is promising to jointly consider pinning policy and application characteristics. In a nutshell, thread pinning is particularly beneficial for reducing run-to-run variability and improving performance stability of OpenMP applications, especially for memory-bound applications, as evidenced by *BabelStream* and synchronization-sensitive applications, as evidenced by *syncbench*. In the remainder of our evaluation, we use thread pinning for all our experiments.

5.3 The effect of SMT

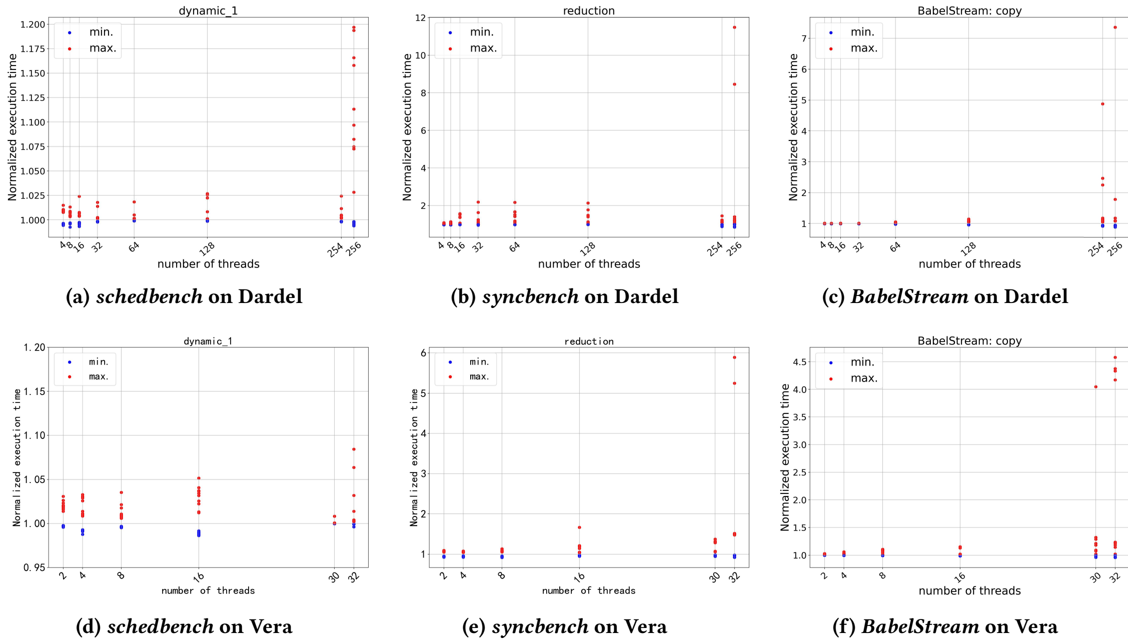
We examine the effect of simultaneous multithreading on Dardel, as Vera does not support SMT. We compare the performance variability of our benchmarks in Figure 5, for the **ST** case, where we use only physical cores of Dardel, e.g.. 32/64/128 cores and OpenMP threads, and the **MT** case, where we use both two hardware threads of 16/32/64 physical cores of Dardel, i.e. 32/64/128 HW threads and OpenMP threads. We note that the use of SMT is usually decided by the developer/user, based on application properties, e.g. the compute-boundedness or memory-boundedness of the application. However, in our evaluation, we regard SMT only as a potential source of performance variability, examining cases where we use the same number of threads.

For *schedbench* in Figure 5a and Figure 5d, even though some run-to-run variability exists under the **ST** configuration, we observe a very high variability among the 100 outer repetitions of the benchmark, for each single run under the **MT** configuration. Regarding *syncbench*, we compare the run-to-run variations of execution times and adopt the coefficient of variation (CV), i.e. the ratio of the standard deviation to the average (lower is better), for every run, as the metric to measure the performance variability of execution time in Figure 5b and Figure 5e. The performance stability is significantly affected in a negative way when leveraging SMT especially for some synchronization directives such as `for`,`single`,`ordered` and `reduction`, as the CV values of all 10 runs show high variances in Figure 5e. For most synchronization cases, the **ST** configuration exhibits better performance stability in Figure 5b, by leaving the second hardware thread free, potentially available for OS activities, whereas higher performance variability, including run-to-run variations and the variations among the 100 outer repetitions for each single run can be seen with the **MT** configuration. Similarly, we compare the performance variability of the normalized minimum and maximum times for all 10 runs for *BabelStream* in Figure 5c and Figure 5f under the **ST** and **MT** configurations respectively. *BabelStream* also does not benefit from using hardware threads.

The above observations reveal that leaving the second thread in SMT implementation for system activities results in better performance stability, while the **MT** configuration makes the executing benchmark experience more SMT interference. This impact of additional hardware thread resources reserved for operating system, i.e. **ST** configuration, varies with benchmark characteristics and scale. For example, **ST** does not outperform **MT** much for BabelStream when only a few threads are used. Overall, leaving the additional thread resources implemented by SMT mechanism for OS activities can be a promising way to achieve performance stability for the OpenMP runtime.

5.4 The effect of frequency variation

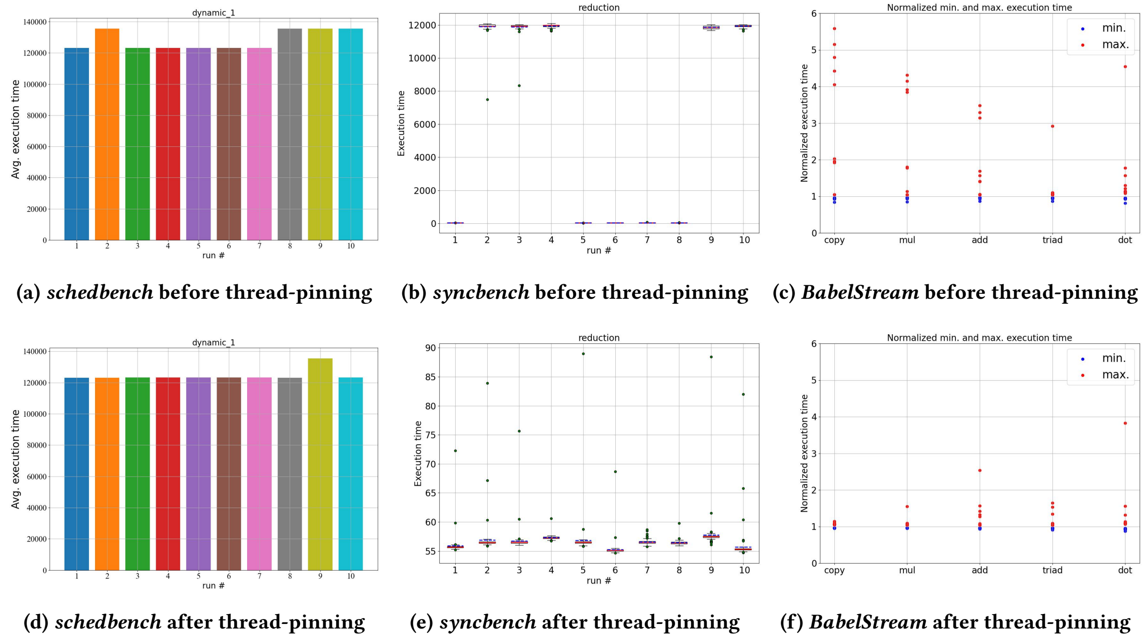
We finally examine the effect of frequency variation on the performance variability of OpenMP, by continuously logging the frequency levels of all cores (read through the sysfs interface of the Linux CPUFreq), through a Python script executing on a separate core. Although the default governor on Vera is set to performance, boosting all the core frequencies to the maximum, for some of our experiments, we have observed performance variability, especially across NUMA nodes, which can be justified by frequency variations. For some of the experiments done on Vera when using same cores but from same NUMA node or cross NUMA nodes, we observed the different behaviours of performance variability that can be potentially



(a) ***schedbench*** **on Dardel** (b) ***syncbench*** **on Dardel** (c) ***BabelStream*** **on Dardel**

(d) ***schedbench*** **on Vera** (e) ***syncbench*** **on Vera** (f) ***BabelStream*** **on Vera**

Figure 3: Scalability of performance variability of normalized execution time in *schedbench*, *syncbench*, and *BabelStream* when increasing the number of used HW threads on Dardel and Vera.

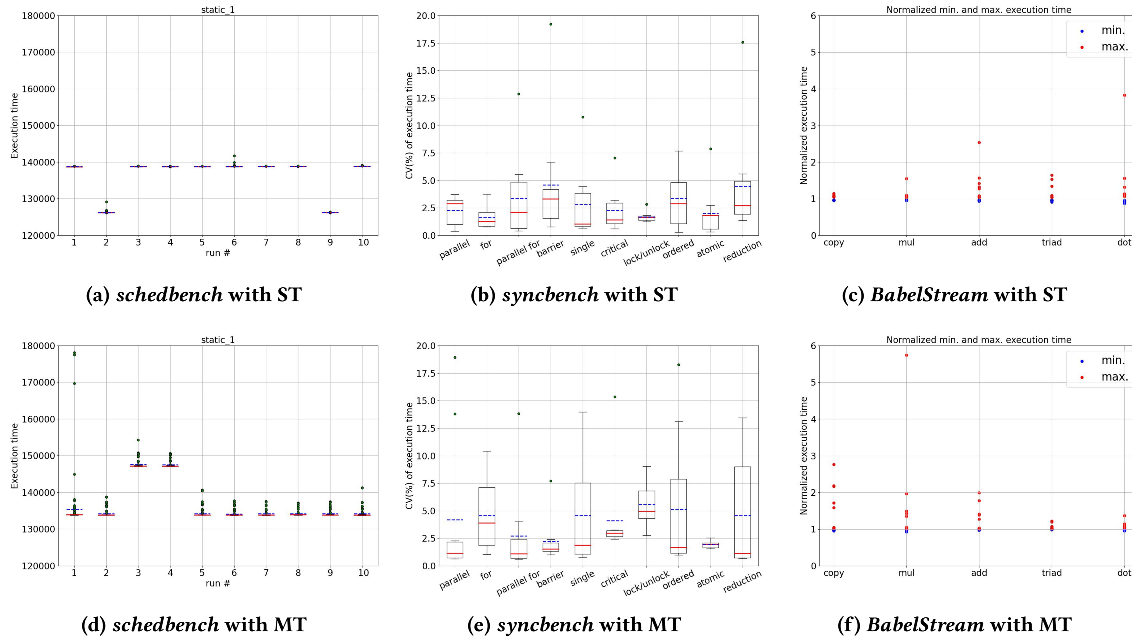


(a) ***schedbench*** **before thread-pinning** (b) ***syncbench*** **before thread-pinning** (c) ***BabelStream*** **before thread-pinning**

(d) ***schedbench*** **after thread-pinning** (e) ***syncbench*** **after thread-pinning** (f) ***BabelStream*** **after thread-pinning**

Figure 4: Lower variability of execution time(μs) after thread-pinning in *schedbench* (first column) when using 16 threads, and in *syncbench* (second column) and in *BabelStream* (third column) when using 128 threads on Dardel.

explained by the variation of frequency. Figure 6 depicts the relation between variability of execution time for *schedbench* and frequency variation. Figure 6c, where we use cores across NUMA nodes, shows higher performance variability, both between different runs, and among the 100 outer repetitions, compared to Figure 6a, where we use the same number of cores on a single NUMA node. Figure 6b and Figure 6d depict the behavior of frequency for these two groups of experiments respectively. The brown region in Figure 6d during all 10 runs indicates more frequent frequency variation, compared to Figure 6b. As higher frequency levels used to run the benchmark, dictated by the performance governor, positively influence the execution



(a) *schedbench* with ST

(b) *syncbench* with ST

(c) *BabelStream* with ST

(d) *schedbench* with MT

(e) *syncbench* with MT

(f) *BabelStream* with MT

Figure 5: Higher variability of execution time (µs) in *schedbench* (first column, executed with 128 threads) and *syncbench* (second column, executed with 32 threads) and *BabelStream* (third column, executed with 128 threads) due to SMT implementation on Dardel.

time, this explains the above observation that the execution times vary more, depending on the frequency variation.

We made a similar observation for *syncbench* in Figure 7, where Figure 7c exhibits more variations for both run-to-run executions and outer repetitions for a single run compared to Figure 7a. The same effect of frequency variation can be seen in the grey region in Figure 7d. We note that on Dardel, we have not observed an obvious trend between performance variability and frequency variations, as Dardel exhibits less frequency variation compared to Vera.

6 CONCLUSION

This paper aims to characterize the performance variability of OpenMP benchmarks and analyze the potential sources and impact of the performance variability based on an experimental study. We have tested two OpenMP benchmarks from the EPCC OpenMP micro-benchmark suite and BabelStream, on two platforms, assessing the impact of thread pinning, SMT, and core frequency variation on performance variability. Our experimental results have illustrated that performance variability exists in OpenMP, both within a benchmark and between different runs, and can be reduced considerably by applying thread-pinning, leaving the additional hardware threads implemented by SMT for OS activities, but can be negatively affected by frequency variation during execution, which is beyond the control of the user.

For future work, we aim to extend our characterization to other benchmarks such as FP-intensive or cache-intensive benchmarks and larger OpenMP applications on other platforms. We also wish to pinpoint the exact sources of OS noise and their impact on OpenMP applications, in order to design strategies to mitigate or eliminate performance variability.

ACKNOWLEDGMENTS

The experiments were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) at PDC partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The experiments were furthermore enabled by resources provided by Chalmers e-Commons at Chalmers. 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.

REFERENCES

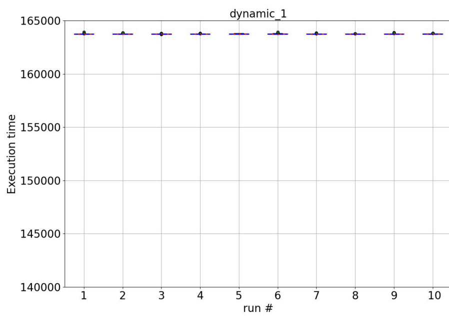
[1] Roberto Camacho Barranco and Patricia J Teller. 2016. Analysis of the Execution Time Variation of OpenMP-based Applications on the Intel Xeon Phi. (2016).

[2] 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.

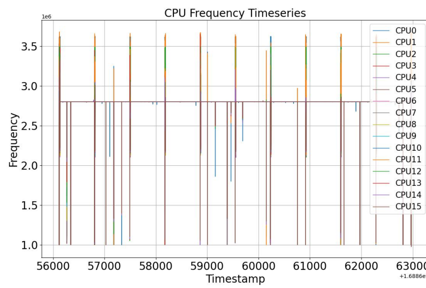
[3] Jonathan Mark Bull and James Clerk Maxwell. 2007. Measuring Synchronisation and Scheduling Overheads in OpenMP. https://api.semanticscholar.org/CorpusID:18396931

[4] Jing Chen, Madhavan Manivannan, Bhavishya Goel, Mustafa Abduljabbar, and Miquel Pericàs. 2022. STEER: Asymmetry-aware Energy Efficient Task Scheduler for Cluster-based Multicore Architectures. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 326–335.

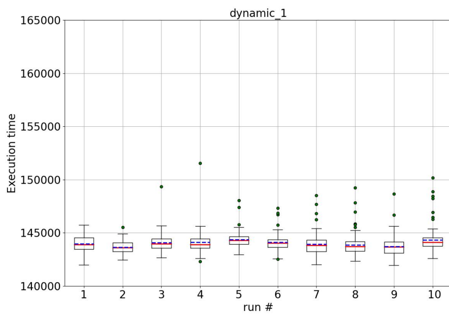
[5] Pradipta De, Ravi Kothari, and Vijay Mann. 2007. Identifying sources of Operating System Jitter through fine-grained kernel instrumentation. In



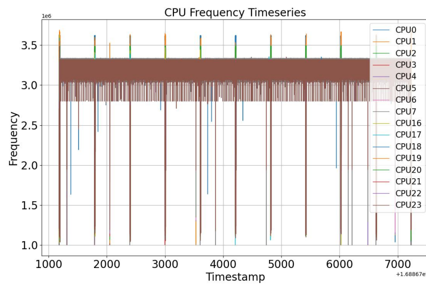
(a) 16 cores from one NUMA node



(b) frequency

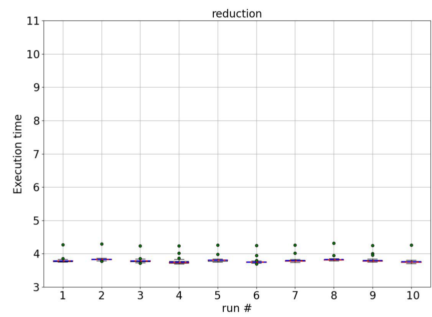


(c) 16 cores from two NUMA nodes

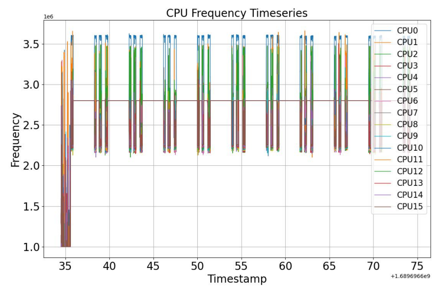


(d) frequency

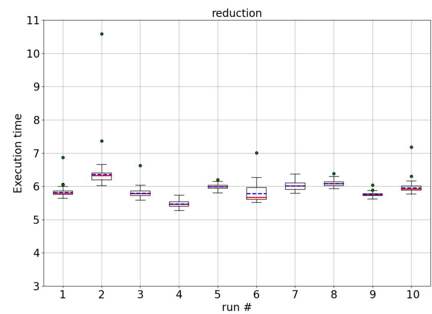
Figure 6: Higher variability of execution time(μs) in *schedbench* due to frequency variation on Vera.



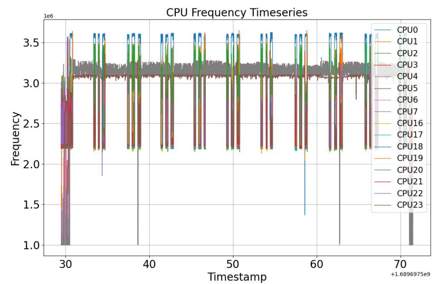
(a) 16 cores from one NUMA node



(b) frequency



(c) 16 cores from two NUMA nodes



(d) frequency

Figure 7: Higher execution time(μs) in *syncbench* micro-benchmark due to frequency variation on Vera.

2007 IEEE International Conference on Cluster Computing*. 331–340. https://doi.org/10.1109/CLUSTR.2007.4629247

[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. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.

[8] 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.

[9] 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.

[10] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. 2004. Analysis of system overhead on parallel computers. In *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology, 2004*. 387–390. https://doi.org/10.1109/ISSPIT.2004.1433800

[11] Yuichi Inadomi, Tapasya Patki, Koji Inoue, and all. 2015. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of SC 2015 (International Conference for High Performance Computing, Networking, Storage and Analysis, SC)*.

[12] 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.

[13] Brian Kocoloski and John Lange. 2018. Varbench: An Experimental Framework to Measure and Characterize Performance Variability. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. New York, NY, USA.

[14] James LaGrone, Ayodunni Aribuki, and Barbara Chapman. 2011. A set of microbenchmarks for measuring OpenMP task overheads. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. Citeseer, 1.

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

[16] 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.

[17] Dong Li, Bronis R De Supinski, Martin Schulz, Dimitrios S Nikolopoulos, and Kirk W Cameron. 2012. Strategies for energy-efficient resource management of hybrid programming models. *IEEE Transactions on parallel and distributed Systems* 24, 1 (2012), 144–157.

[18] Aniruddha Marathe, Peter E Bailey, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. 2015. A run-time system for power-constrained HPC applications. In *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings 30*. Springer, 394–408.

[19] 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

[20] Abdelhafid Mazouz, Sid Touati, and Denis Barthou. 2011. Analysing the variability of openMP programs performances on multicore architectures. In *Fourth workshop on programmability issues for heterogeneous multicores (MULTIPROG-2011)*. 14.

[21] 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. *2011 International Conference on High Performance Computing & Simulation* (2011), 273–279. https://api.semanticscholar.org/CorpusID:6499273

[22] 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.

[23] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. 2011. A quantitative analysis of OS noise. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 852–863.

[24] Open MP. 2018. *OpenMP-API-Specification.* Retrieved March 23, 2022 from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[25] 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.

[26] Barry Rountree, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal, and Martin Schulz. 2012. Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 947–953.

[27] 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.

[28] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. 2005. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. Association for Computing Machinery, 303–312.

[29] 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.

[30] HPC Wiki. 2022. *Binding/Pinning*. Retrieved March 23, 2022 from https://hpc-wiki.info/hpc/Binding/Pinning

[31] 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.