Array 19 (2023) 100311

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/locate/array

Performance evaluation on work-stealing featured parallel programs on asymmetric performance multicore processors☆

Adnan

Department of Informatics, Universitas Hasanuddin., Jl. Poros Malino, Kab. Gowa Sulawesi Selatan, Indonesia

ARTICLE INFO

Keywords:
Amdahl's law
Speedup factor
Asymmetric performance
Multicore
Work stealing

ABSTRACT

The speed difference between high-performance CPUs and energy-efficient CPUs, which are found in asymmetric performance multicore processors, affects the current form of Amdahl's law equation. This paper proposes two updates to that equation based on the performance evaluation results of a simple parallel pi program written with OpenCilk. Performance evaluation was done by measuring execution time and instructions per cycle (IPC). The performance evaluation of the parallel program executed on the Intel Core i5 1240P processor did not indicate decreased performance due to asymmetric performance. Instead, the program with efficient work-stealing advantages from OpenCilk performed well. In the case of using the execution time of the P-CPU as a reference to obtain speedup, the evaluation results in a sublinear speedup. Conversely, in the case of using the execution time of the E-CPU as a reference, the evaluation results in a superlinear speedup. This paper proposes two updates to Amdahl's law equation based on these two evaluation results.

1. Introduction

Since 2021, Intel has introduced two generation of asymmetric performance multicore processors. The architecture is similar to the Little-Big architecture [1]. Asymmetric performance multicore processors integrate two different types of core processors. The first type is high-performance cores (P-Cores), designed to leverage single-threaded performance by boosting their frequency clock. They are also capable of hiding latency by simultaneous multithreading. The second type is energy-efficient cores (E-Cores), designed to leverage performance efficiency through multithreaded programming. Both E-Cores and P-Cores share the same instruction set, so they can perform in parallel to execute a multithreaded process.

The emersion of the Intel asymmetric performance multicore products will further enliven the use of AMPs architectures for broader applications. For better application and the best practices in software development, it is necessary to understand their performance behavior well. For simple writing purposes in this paper, we rewrite the CPU of P-Core as P-CPU and CPU of E-Core as E-CPU.

This paper discusses two issues related to the emergence of AMPs architecture. The first issue is the implications of AMPs on Amdahl's law equation, and the related issue is the problem of unfair work distribution in AMPs.

When it comes to AMPs, it is necessary to revisit the equation form of Amdahl's law [2]. The primary reason for this is that the equation, as shown in Eq. (1), assumes that all N processors have the same and constant speed. The equation only has two independent variables, namely N and f, which represent the number of processors and the parallel fraction, respectively. However, with AMPs, there are dynamic and static variations in CPU speed. Dynamic variations in CPU speed refer to changes in the speed of CPUs caused by changes in their clock frequency. These changes are temporal and occur on processors. This cause is also known as dynamic voltage-frequency scaling. Static speed variations refer to variations in speed among a number of N processors spatially. Static speed variations occur on CPUs operated at different clock rates and/or CPUs with different microarchitectures.

$$S(f,N) = \frac{T_S}{T_P} = \frac{1}{1-f+\frac{f}{N}} \tag{1}$$

In the context of discussing AMPs, the speedup factor expresses the ratio of speed between faster processors and slower processors. Even though the idea of the speedup factor is nothing new, and regardless of how we derive its value, it is appropriate and a novelty to incorporate the speedup factor into the equation form of Amdahl's law. specifically how the speedup factor allows for sublinear and superlinear speedup from the perspective of Amdahl's law.

Amdahl's law is a tool used to estimate the speedup of parallel programs on parallel computers. However, the Amdahl's law equation makes certain assumptions that are not in line with the latest developments. Several related studies have attempted to extend Amdahl's equations. The first work that extends Amdahl's law and Gustafson's law

☆ In this work we demonstrate performance of OpenCilk on the Single ISA Hybrid Multicore Architecture.

E-mail address: adnan@unhas.ac.id.

https://doi.org/10.1016/j.array.2023.100311
Received 5 May 2023; Received in revised form 29 June 2023; Accepted 17 July 2023
Available online 20 July 2023

simultaneously is presented in [3]. The result is the memory-bounded speedup model.

The memory-bounded speedup in [4] can initially explain the phenomenon of how superlinear speedup can occur in cases involving problems with large sizes on a computer cluster. With a large problem size, a single node may run slowly due to memory limitations. By adding a number of nodes, the program speed per node becomes faster. Memory-bounded speedup can also explain how superlinear speedup can occur on symmetric multicore processors [5,6]. If the memory capacity increases with the increase in the number of nodes on a computer cluster, whereas in a symmetric multicore, the capacity of cache memory increases with the increase in the number of CPU cores used. From here, investigating whether compute-bounded workloads can encounter superlinear speedup events is an interesting question to investigate.

In [7] Hill and Marty proposed a performance model for Asymmetric Multicore processors. In their proposal, they assumed the use of processors with asymmetric multicore architecture, consisting of r base equivalent cores and one powerful core with perf(r) capabilities. In their extension, the authors proposed to accelerate the sequential fraction with a powerful core, while the parallel fraction was accelerated by some base cores. However, is it always useful for the fastest CPU to execute serial fractions?

To enhance comprehension of the design of various types of many-core processors, revised versions of Amdahl's law model are presented in [8]. These modifications are based on the performance model outlined in [7]. The revised model introduces a variable Sc, which represents the normalized performance of the efficient core relative to the powerful core.

A re-evaluation of Amdahl's law has also been conducted and presented in [9]. The paper assumes that all BCE resources are composed of a number of powerful cores, resulting in a symmetric performance multicore model. This assumption expects to enhance the performance gain. The extension model is then built using these assumptions.

The normal form heterogeneity model is presented in [10]. It appears that the normal heterogeneity model can accurately model superlinear speedups, but it may not be effective in modeling cases of sublinear speedups. Sublinear speedups are bound to occur if the fastest CPU is used for the serial execution, which serves as the baseline.

In general, the existing speedup law cannot accurately describe the performance behavior of AMPs. Specifically to the AMPs, superlinear speedup can occur if code that was previously executed by a slow processor core is then enhanced by one or more faster processor cores. Conversely, sublinear speedup can occur in cases where serial execution time is obtained from serial execution using the fastest CPU. This research gap is the basis for this research.

For the sake of simplicity, this paper will only discuss the static variation of CPU speed among the CPUs of AMPs. The main analysis is based on the asymmetric performance that statically exists among the CPUs of AMPs.

Not only is there variation in speed in AMPs that undermines the applicability of Amdahl's law, but idleness is also not factored into the equation, leading to deviations in speedup prediction. Moreover, the difference in speed between P-CPUs and E-CPUs can lead to an inequitable workload distribution. In the case of regular workloads with static scheduling, P-CPUs may remain idle for longer periods compared to E-CPUs, resulting in the execution time being bound by the slowest CPU. Therefore, this paper proposes a sub-problem of unfair workload distribution as an example case, which is addressed through work stealing.

Several studies have attempted to address the issue of unfair workload distribution by ensuring that the P-CPUs obtain more workload through static and dynamic scheduling. However, potential issues still exist. Static scheduling requires adjustments to the task grain size for different cases to achieve optimal performance, while dynamic fine-grain task scheduling provides load balancing at the cost of additional works [11].

Although work-stealing has been proposed as a solution to overcome the problem of unfair workload distribution in dynamic heterogeneous distributed computers, shared memory work-stealing on AMPs has not been investigated to the best of our knowledge. Distributed work-stealing in distributed computers has different characteristics than work-stealing in shared memory multicore architectures such as AMPs. Specifically, work-stealing operations that use communication channels contribute to overhead. On the other hand, there is no communication for work-stealing on AMPs, which means that it does not contribute to overhead. Therefore, investigating the performance of work-stealing on AMPs is still an interesting topic to discuss.

To address the issue of unfair workload distribution among AMPs, we utilize OpenCilk, which possesses work-stealing capabilities and non-blocking multithreading. OpenCilk is an ideal choice for shared memory system architectures like AMPs. It operates on lazy task creation and employs THE protocol, resulting in minimal overhead during task creation and stealing. By utilizing work-stealing and non-blocking multithreading, OpenCilk anticipate that faster processors will handle a greater workload compared to slower processors. Consequently, we expect to achieve superior load balance, utilization, and speedup of AMPs.

The rest of this paper is organized as follows. In Section 2, we present some discussions on related works. Mainly we applied OpenCilk and make comparisons on other papers. Section 3 is to discuss detailed experimental works, tools and configurations. We present and discuss our results in Section 4 and make some comparison with the results of recent papers in Section 4. We conclude this work in Section 5.

2. Related works

The effects of asymmetric performance of a Multicore processor were first studied in [12]. The study approximated asymmetric performance by controlling the frequency of each core of Intel Xeon multiprocessors. A similar idea was presented in [13], which simulated asymmetric performance by controlling the frequency and experimented with X10 work-stealing. However, our work differs as we applied OpenCilk to AMPs, which resulted in lower task creation and steal overheads and different results. We also excluded the frequency variation factor in one of our experiment scenarios by setting the frequency of all CPUs to the same frequency of 2.57 GHz on real asymmetric multicore CPUs. Additionally, we proposed two updates to the equation form of Amdahl's law based on our study.

The authors of [14] presented research on the performance evaluation of heterogeneous multicore processors on multi-threaded workloads. In the paper, they demonstrated that heterogeneous multicore is more efficient than homogeneous multicore when handling a multi-threaded workload. The authors deduced efficiency from the weighted speedup, which is calculated by dividing the total IPC of all running threads by the total IPC of the single slowest core. In this case, the authors conducted IPC measurements in a constant time window.

The authors of [15] presented a study on the obligatory nature of AMPs for achieving the highest area or power efficiency. This research suggests that heterogeneous multicore chips will be the most efficient design to handle broad-type application workloads with varying characteristics.

Attention has been drawn to the importance of workload fairness in scheduling for single-ISA heterogeneous multicore processors in a study presented in [16]. The study has demonstrated that equal-progress scheduling enhances performance on multit-threaded workloads. As the two types of cores on the AMPs have asymmetrical performance, a similar effect to the load imbalance scenario will occur during barrier synchronization. Equal-progress scheduling has served as an OS layer load balancer between a high-performance and efficient core.

This work demonstrated a solution to the unfairness problem in multi-threaded computations, which differs from [16]. Other authors present research that also makes attention to fair scheduling in [17].

All the works mentioned above focused on the OS-level fairly scheduling for the multi-programmed environment. However, our work tested the fair work distribution among the E-CPUs and P-CPUs in a multi-threaded program. Our point of view regarding fairness is that we argue that the faster CPUs should endure more works of parallel fraction than the slower ones as other works presented Speedup and Power Scaling Models in [10] and AID in [18]. As authors indicated in [19], unbalanced task distribution causes poor performance.

The OS-level mechanism ensures that each kernel-level thread receives a fair share of CPU time while working in the multiprocessing domain. On the other hand, the user-level mechanism allows a multithreaded application to efficiently utilize multiple CPU/core time and improve parallelism. To achieve this, programming for multicore systems uses multithreaded programming with a one-to-one mapping approach, where each user-level thread maps to a kernel-level thread. However, if a user-level thread becomes idle (for instance, while waiting for other threads due to load imbalance during computation), the CPU time it receives is wasted. In this case, idle threads do not contribute to the performance of multithreaded programs. Unfortunately, the kernel does not support the management of user-level threads [20]. Thus we need a user-level load-balancing mechanism such as work-stealing for multithreaded applications on AMPs.

Even more so in the era of multicore and manycore, parallel programs tend to utilize fine-grain parallelism to take advantage of the abundant parallelism of computing units. Fine-grain parallelism necessitates minimizing scheduling costs, as it is well-known that thread scheduling at the user task level is much lower than at the OS level.

From the perspective of a slower processor, an unfair distribution of work can be considered as an excessive workload. In the context of work stealing, there may be benefits for faster processors to prioritize stealing tasks from slower processors. For this purpose, victim selections [21–23] can be applied. Combining victim selection and stealing half or multiple items work stealing [24–26] may reduce the workload of slow victims and decrease the overhead of the thief.

A heterogeneous asymmetric multicore chip has the potential to offer more efficient performance than symmetric multicore, as presented by the authors in [7]. Although the authors wrote a retrospective paper in [27] regarding their previous model, they made a relevant recommendation that researchers should continue investigating this asymmetric multicore chip, including dealing with scheduling and overheads. Our work in this paper presents performance evaluation results where we assigned threads to the efficient core CPUs first and enhanced execution with the higher performance core.

Some of the papers we identified [7,28,29], based on Amdahl's law, suggest accelerating the critical path using the fastest CPU. Similar ideas to assign the fastest CPU to the critical path tasks are presented in [30,31].

Regarding our proposal to integrate the speedup factor into Amdahl's law equation, there is a need to estimate the speedup factor. Predicting this factor is a challenging task [32]; however, it is highly useful in determining the required type and number of CPUs for AMPs.

In this work, we demonstrated that OpenCilk [33,34], which was Cilk [35], overcame the problem of unfair work distribution as OpenCilk is capable of load-balancing by work-stealing [36]. When a worker is out of a task from its ready queue, it selects another worker as the victim and then steals a task from the victim's ready queue. Not only is the load-balancing capability the advantage of OpenCilk, but as the available parallelism is significantly more than the number of processors [37], OpenCilk workers tend to run on the fast clone. Furthermore, efficient reducer hyperobject implementation [38] features OpenCilk. All these features result in many workers being busy with work, less stealing operations, and finally, it has a low total overhead.

In this work, we also demonstrated that OpenCilk might perform a superlinear speedup, in which case the parallel program pinned the threads to E-CPUs first, and then P-CPUs give an assist. OpenCilk also might perform a sub-linear speedup, in which case the parallel program pinned the threads to the P-CPU first, and then E-CPUs give an assist. For these cases, we proposed an update to Amdahl's law equation.

3. Material and experimental configuration

In our experiment, we collected performance data from the Intel Core i5 1240P. The processor has four P-CPUs and two E-CPU clusters, each cluster consisting of 4 cores. Both P-CPU clusters are implemented with the Golden Cove microarchitecture, while the E-CPUs are implemented with the Gracemont microarchitecture. Therefore, the Intel Core i5 1240P is an asymmetric performance multicore processor.

Each of the P-CPU cores is equipped with 1280 KB of L2 cache memory, while all cores in each E-CPU cluster share 2048 KB of L2 cache memory. At a higher level of the cache memory hierarchy, all cores of both P-CPU and E-CPU share 12 MB of L3 cache memory. At the lowest level of the cache memory hierarchy, each P-CPU gets 48 KB L1 for data and 32 KB L1 for instructions. As for the E-CPU clusters, each of them gets 64 KB L1 for instructions and 32 KB L1 for data.

P-CPUs are capable of two-ways simultaneously multithreading, while E-CPUs do not. Therefore, the processor provides a total of 16 logical CPUs. As shown in Table 1, P-CPUs are numbered from 0 to 7, while E-CPUs are numbered from 8 to 15. P-CPUs have a maximum clock speed of 4.4 GHz, while E-CPUs have a maximum clock speed of 3.3 GHz.

3.1. Experimental scenarios and configuration

In this work, experiments were conducted in a number of scenarios. The scenarios indicate the order of adding CPUs in such a way as to see the difference in scalability between E-CPU first and P-CPU first. E-CPU first means that threads occupy E-CPU first until none are left and then P-CPU. Conversely, P-CPU first means that threads occupy P-CPU first until none are left and then E-CPU. To achieve that goal, the benchmark program utilized the Linux CPU affinity library [39] to determine which CPU to use.

The first scenario involves executing the benchmark solely on the physical performance cores while scaling up the number of threads to the maximum number of available physical CPUs. In this scenario, the P-CPUs operate in non-hyperthreading mode, and the CPU mask is set to [0, 2, 4, 6].

The second scenario entails running the program on the E-CPUs exclusively. For the efficient core only scenario, the CPU mask is set to [8, 9, 10, 11, 12, 13, 14, 15].

The third scenario is the hybrid scenario, which involves executing the program on both P-CPUs and E-CPUs. In this scenario, threads are scheduled to the P-CPUs first before the E-CPUs, and the CPU mask is set to [0, 2, 4, 6, 8, 9, 10, 11, 12, 13, 14, 15].

In the fourth scenario, called SMT, all threads alternate between populating the physical and logical P-CPUs. The CPU mask is set to [0, 1, 2, 3, 4, 5, 6, 7]. The fifth scenario is similar to the fourth, but it assigns threads to the physical CPU first before its logical pair. In this scenario, the CPU mask is set to [0, 2, 4, 6, 1, 3, 5]. The sixth scenario is the hybrid SMT, which is similar to the hybrid scenario, but all threads are assigned to all P-CPUs first before E-CPUs. In this scenario, P-CPUs work in hyperthreading mode. The last scenario is the hybrid SMT physical first scenario, which utilizes both physical P-CPUs and E-CPUs before remaining logical P-CPUs. Table 2 summarizes all the scenarios. In our execution platform, the ID number from 0–7 identifies all P-CPUs, and the ID number starts from 8 to 15 and identifies all E-CPUs. To determine which CPU program schedules threads, this experiment uses the Linux thread affinity library (sched.h). To factor out the effect of clock-speed variation on performance in some other experiments, we set all CPUs to a clock frequency of 2.57 GHz.

Table 1
Clock speed of CPUs on Intel Core i51240P.

CPU	Core	Max f	Min f	Measured	Type
0–1	0	4.4 GHz	0.4 GHz	2.1 GHz	P-CPU,HT
2–3	1	4.4 GHz	0.4 GHz	2.1 GHz	P-CPU,HT
4–5	2	4.4 GHz	0.4 GHz	2.1 GHz	P-CPU,HT
6–7	3	4.4 GHz	0.4 GHz	2.1 GHz	P-CPU,HT
8–15	4–11	3.3 GHz	0.4 GHz	2.1 GHz	E-CPU

Table 2
Experiment names and its CPU scheduling order.

Scenario name	# threads	CPU set mask
Performance core	1–4	[0, 2, 4, 6]
Efficient core	1–8	[8, 9, 10, 11, 12, 13, 14, 15]
Hybrid mode	1–12	[0, 2, 4, 6] [8, 9, 10, 11, 12, 13, 14, 15]
Hybrid	1–12	[8, 9, 10, 11, 12, 13, 14, 15]
E-CPUs First		[0, 2, 4, 6]
SMT mode	1–8	[0, 1, 2, 3, 4, 5, 6, 7]
SMT PF mode	1–8	[0, 2, 4, 6, 1, 3, 5, 7]
Hybrid SMT	1–16	[0, 1, 2, , 3, 4, 5, 6, 7 8, 9, 10, 11, 12, 13, 14, 15]
Hybrid SMT PF	1–16	[0, 2, 4, 6, 8, 9, 10, 11 12, 13, 14, 15, 1, 3, 5, 7]

Table 3
Software setup and configurations.

Software	Version	Compiler flags
Cilk Compiler	Clang 14	-fopencilk -O3
Serial Compiler	Clang 14	-O3
OS	Ubuntu 22.04	default configuration

3.2. Benchmark software

Regarding the benchmark software, we have written an OpenCilk implementation for numerical integration to estimate the value of pi. The program is highly parallel, and therefore, we can assume that the parallel fraction f is equal to 1. The numerical integration method used is the rectangle rule with a problem size of 2 billion iterations. Table 3 provides a summary of the software environment setup for this evaluation.

Creating a parallel version of a serial program using OpenCilk is easy. Creating a parallel program with OpenCilk only requires adding the necessary keywords. The basic keyword, Cilk_spawn, is the first keyword to be used. This keyword can be used for both regular and irregular parallelism. An example of regular parallelism is Fibonacci recursion, while an example of irregular parallelism is a program to solve the n queen problem.

One can use the keyword "cilk_spawn" to create a child parallel task that can run in parallel with its parent task. Then, "cilk_sync" can be used to make the parent task wait for all child tasks to complete. Additionally, one can use the keyword "cilk_scope" to define a structured block of codes that contains a number of parallelized tasks. "Cilk_spawn" can be nested within a serial loop to create multiple parallel tasks. If the number of iterations can be predetermined, as is typical in a "for" loop construct, it is highly recommended to transform the loop using the divide and conquer method.

For regular cases of parallelism, such as parallel loops, one can use the keyword "cilk_for". "cilk_for" applies a recursive divide and conquer algorithm. In this study, the parallel pi program is implemented using the cilk_for keyword. To overcome the race condition problem that arises in read–modify–write operations on shared variables in parallel, a reducer hyperobject is used.

The serial program implementation includes a parameter to choose the CPU on which to schedule, while the OpenCilk implementation of the parallel program includes a parameter that the program requires to pick off an operating mode, as described in Section 3.1. The program also has the second parameter to specify the number of threads involved in execution. The mode option determines the CPU mask and indirectly determines the order of the assigned CPUs when the program increases the number of threads. For example, two options, which select the CPU mask [8:15,0,2,4,6] and nine threads, will make eight threads occupy E-CPUs first and then one thread occupy a P-CPU. Then increasing the number of threads to ten will make eight threads occupy E-CPUs first, and two threads occupy two P-CPUs with ID 0 and 2. In addition, our experiments collect performance data repeatedly with Python script. Python script also sets an environment variable CILK_NWORKERS.

Table 4
CPU clock frequency setting and serial execution time.

CPU	Frequency	Execution time
P-CPU	2.57 GHz	3.134 s
	3.30 GHz	2.604 s
	3.50 GHz	2.454 s
	4.40 GHz	1.955 s
E-CPU	2.57 GHz	6.619 s
	3.30 GHz	5.030 s

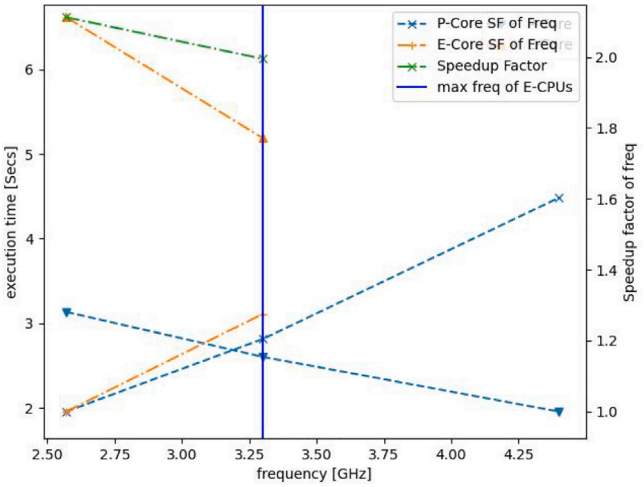

Fig. 1. Serial execution time and Speedup Factor measurement vs Clock Speed.

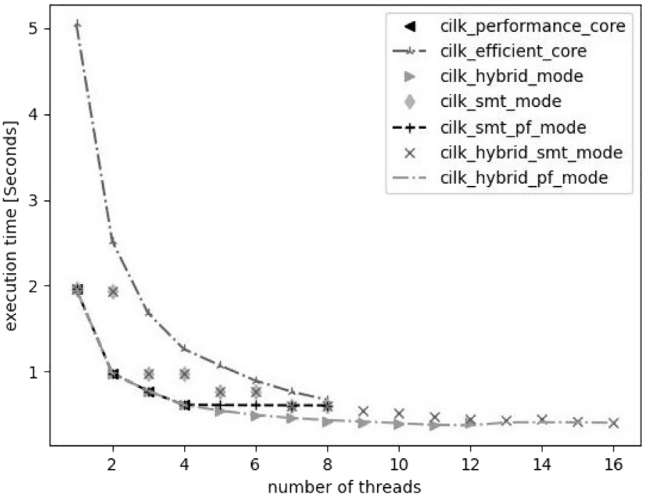

Fig. 2. Parallel programs execution time in various scenarios.

4. Results and discussion

The results presented in Fig. 1 and Table 4 indicate that the P-CPU to E-CPU speedup ratio is greater than their clock speed ratio. This

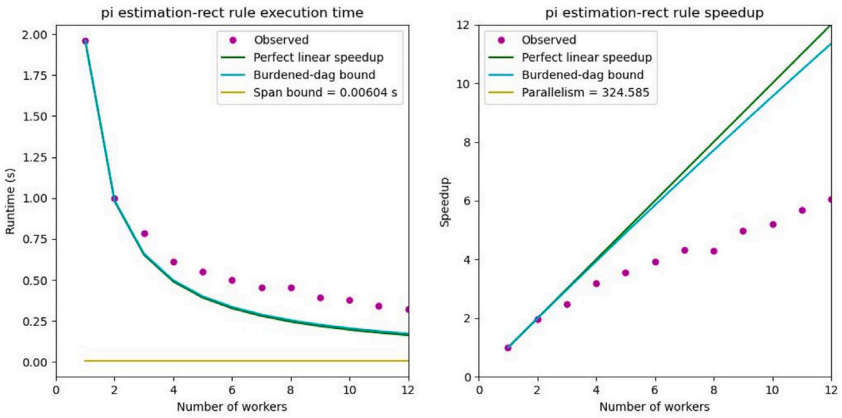

Fig. 3. Scalability analysis of the hybrid mode of P-CPUs first. Clock dropped from 4.4 GHz to 3.3 GHz since nworkes equaled 3.

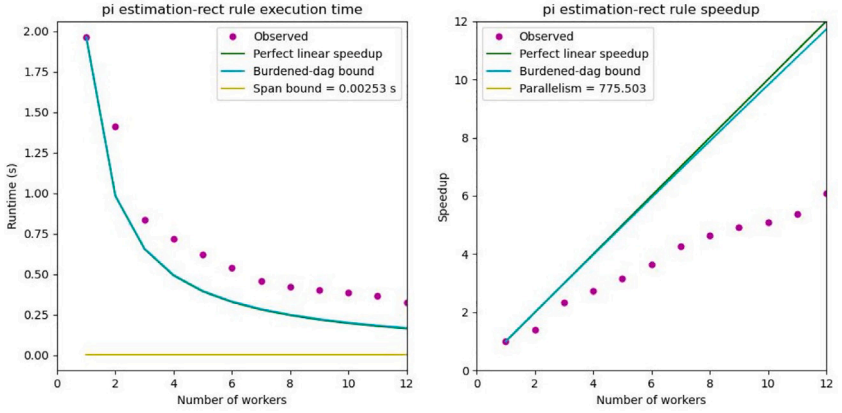

Fig. 4. Scalability analysis of the hybrid mode in which P-CPUs and E-CPUs interleave.

Table 5
CPU clock frequency measured by CPU Power and Parallel execution time.

No.	Scenario	# threads	Freq	Execution time
1	P-CPU	4	3.52 GHz	0.613 s
2	P-CPU	4	2.57 GHz	0.859 s
3.	E-CPU	8	3.11 GHz	0.672 s
4.	Hybrid	4 (P-CPUs)	3.52 GHz	0.323 s
		8 (E-CPUs)	3.11 GHz	

suggests that the superior microarchitecture of P-CPUs has contributed to better performance. Even when we reduced the clock speed to match that of E-CPUs, P-CPU still resulted in better execution time. At the same frequency of 2.57 GHz, P-CPU is twice as fast as E-CPUs. Therefore, we can take the P-CPU to E-CPU speedup factor, denoted as sf, to be 2.

The performance evaluation results in Fig. 2 show the execution time of various configurations. "Cilk_perf_core" indicates the execution time using a number of physical CPUs (P-CPUs). "Cilk_efficient_core" indicates the execution time using a number of efficient CPUs (E-CPUs). "Cilk_hybrid _mode" indicates the execution time using a combination of P-CPUs and E-CPUs. "Cilk_smt_mode" indicates the execution time using a number of P-CPUs with simultaneous multithreading capability. "Cilk_smt_pf" indicates the execution time using a physical P-CPU and a number of threads from the same core (SMT mode). "Cilk_hybrid_smt" indicates the execution time using all available threads from P-CPUs (SMT) and enhanced by a number of cores from E-CPUs.

Fig. 2 presents the results of the performance evaluation. The figure demonstrates that P-CPU performs faster than E-CPU. Interestingly, the time of pinning eight threads on E-CPUs was almost comparable to pinning four workers on P-CPUs. The hybrid mode, which utilizes both P-CPUs and E-CPUs, slightly improved performance in asymmetric performance scenarios. However, due to a slowdown in clock speed, performance did not scale linearly when the number of workers exceeded two. Execution time increased slightly when the number of workers equaled 12. This increase was not caused by critical path overhead but rather by increased temperature in the CPUs after continuous use. In the simultaneous multithreading mode (annotated with an SMT label), using more than two threads on the same core did not improve performance.

Dynamic voltage and frequency scaling is widely used to balance performance needs and power consumption. The operating point of the voltage and frequency core is known as the p-state. The lower the p-state, the higher the frequency and voltage, and vice versa. Depending on the workload, software, hardware, or a combination of both can select the most appropriate p-state. A limited number of cores can operate at the lower p-state, as long as they remain within the permissible operating limits such as the number of active cores, power consumption, and temperature. If the requirements for the number of active cores or temperature are no longer met, the active governor chooses to use a higher p-state. Therefore, a single-threaded program can run at a lower p-state than a multithreaded program. In this case, the change in the p-state from low to high when scaling the number of CPUs affects the speedup measurement.

Table 5 shows working frequencies and parallel execution times, which provide evidence of decreased clock frequency while the CPUs ran in multithreading and power-saving mode as we increased the number of workers.

In this paragraph, we will discuss and analyze the performance scalability generated by using the cilkscale tool [40]. Cilkscale first

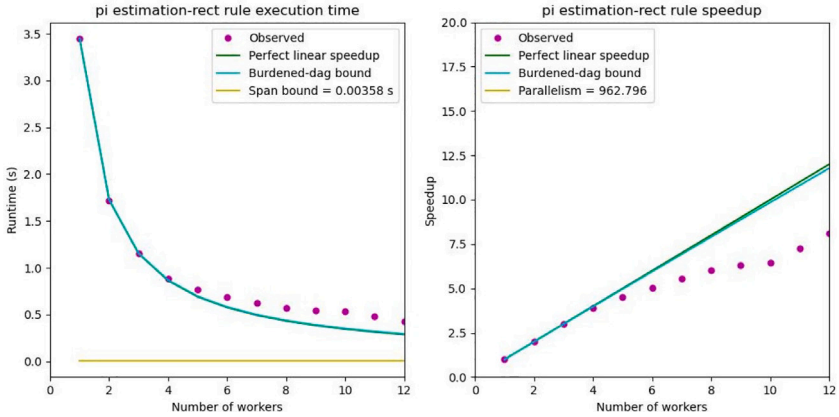

Fig. 5. Scalability analysis of the hybrid mode of P-CPUs first. Clock @ 2.57 GHz.

executes both serial and parallel programs, and then performs metric measurements such as serial execution time (T_1) and Span (T_∞). Parallelism is calculated by dividing T_1 by T_∞. In some of the figures presenting scalability visualizations, the parallelism horizon line may not be visible if its value falls beyond the y-axis [41].

Fig. 3 presents the scalability analysis of scenario 3 on AMPs. Until the number of threads equaled two on two P-CPUs, the speedup indicated linear scalability. However, scaling the number of CPUs with more than two P-CPUs resulted in a deceleration of speedup due to slowdowns in clock frequency. Further scaling of the number of CPUs to more than four CPUs switched to the asymmetric performance mode. The scalability in the asymmetric performance mode was sublinear due to the performance gap compared to the P-CPU as its reference. This phenomenon is clearly observed when we alternate both the P-CPUs and E-CPUs, as shown in Fig. 4. However, alternating P-CPUs and E-CPUs did not scale performance linearly after the number of workers equals or greater than 2.

According to the work law [42], the execution time Tp on P processors has a lower bound of as in Eq. (2):

$$T_P \geq \frac{T_1}{P} \tag{2}$$

However, in the case of the AMPs, which may provide at least two versions of T_1, there will be at least two versions of T_P. The value of T_p depends on the clock speed, the number of CPUs P, and the T_1 used as its sequential execution time of reference. Moreover, clock speed can decrease with the increase in the number of processors.

To simplify the problem, this paper eliminates the clock speed factor. Thus, our problem is confined to the static variation of speed among the P processors. Let us observe the performance of the same program at a lower clock speed and P-CPU as a base point, as shown in Fig. 5. Eq. (2) still holds as the CPUs had symmetric performance. In Fig. 5, T_1 equals 3.449, which differs from T_1 in Fig. 3. In Fig. 5, there was no slowdown in clock speed, and it scaled the performance of P-CPU linearly. As the experiments enhanced the number of CPUs that switched to the asymmetric mode, speedup rate indicated a sublinear due to the lower speed of E-CPUs. Thus, for the AMPs, we rewrite Eq. (2) as Eq. (3), where P is the number of P-CPUs and E is the number of E-CPUs. The constant of sf is due to our experiment, which showed that E-CPUs are empirically slower, with sf times folds than the P-CPU.

$$T_P \geq \frac{sf \times T_1}{sf \times P + E} \tag{3}$$

By alternating between the P-CPUs and E-CPUs and operating them at a clock frequency of 2.57 GHz, a similar pattern was observed as shown in the previous figure (Fig. 6). Eq. (4) expresses the total speedup, including a serial fraction. In the equation for the total speedup, sf represents a speedup factor that only enhances the parallel fraction and not the serial fraction for the scenario discussed.

$$Speedup(f, P, E, sf) = \frac{1}{1 - f + \frac{sf \times f}{sf \times P + E}} \tag{4}$$

In the best practices for evaluating performance scalability, it is appropriate to derive speedup based on the fastest execution time of the sequential version of the same solution. In cases of asymmetric performance with multicore processors, the shortest execution time is obtained from the execution run of the fastest CPU, i.e., P-CPU. However, what if we evaluate scalability starting with the slower CPU (E-CPU)? This paper argues that using T_1 from E-CPU as the reference is appropriate. This approach makes sense and is precise as long as the performance evaluation is still based on the same optimized code.

Fig. 7 presents the scalability of the asymmetric performance mode, starting with the use of E-CPUs before scaling up with the P-CPUs. All CPUs were set back to their default high clock speed. As P-CPUs were faster than E-CPUs by a factor of sf, each additional P-CPU increased the speedup by sf. Since E-CPUs scaled performance linearly, additional P-CPUs contributed to a superlinear speedup.

Setting the clock frequency at a lower rate (2.57 GHz) causes both work (T_1) and span (T_∞) to increase, as shown in Fig. 8. In this case, parallelism remained sufficient, resulting in linear speedup of the performance of E-CPUs. Enhancing the execution with extra P-CPUs resulted in superlinear speedup, as predicted by Eq. (5). Now, T_P can be predicted using Eq. (5), where T_1 is the work of E-CPUs. As long as parallelism is sufficient, the critical-path overhead is small enough to contribute to T_P.

$$T_P \geq \frac{T_1}{sf \times P + E} \tag{5}$$

$$Speedup(f, P, E, sf) = \frac{1}{\frac{1-f}{sf} + \frac{f}{sf \times P + E}} \tag{6}$$

If we assign one P-CPU to the serial fraction and assign the E-CPUs first to the parallel fraction before the P-CPUs, we obtain a total speedup according to Amdahl's law, as shown in Eq. (6). The term sf represents the speedup factor that indicates the relative speed of the P-CPUs to the E-CPUs. This paper presents evidence that efficient work-stealing techniques can overcome the problem of unfair load distribution on AMPs. The evidence shows that faster processors do not lose performance even in asymmetrical mode. Table 6 shows that the P-CPUs did not experience a nominal decrease in IPC when P-CPU and E-CPU cooperated in asymmetric mode. Except for CPU 11, E-CPUs also did not experience nominal changes in IPC. Work-stealing appears to keep the P-CPUs' nominal IPC the same as when they executed the same program without the E-CPUs.

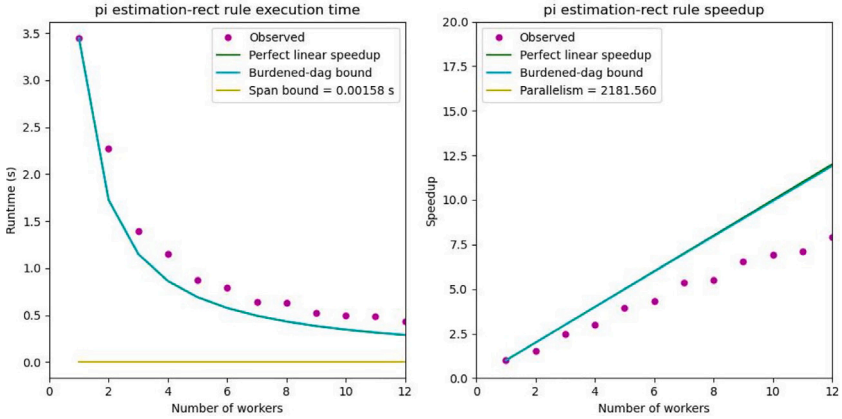

Fig. 6. Scalability analysis of the hybrid mode in whch P-E CPUs interleave. Clock @2.57 GHz.

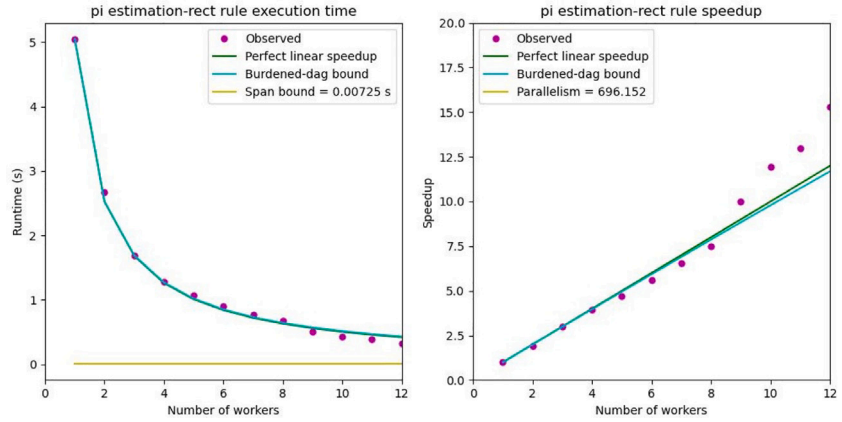

Fig. 7. Scalability analysis of the E-CPUs first as indicated in line no 4. Table 5.

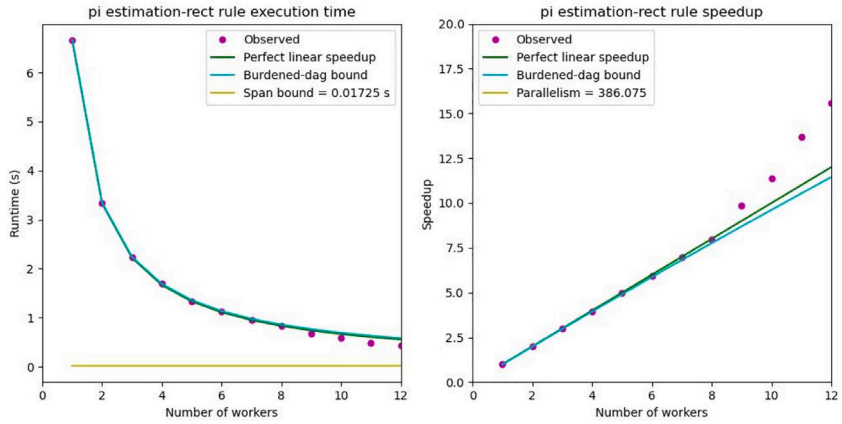

Fig. 8. Scalability analysis of the E-CPUs first. Clock @ 2.57 GHz.

Table 7 presents some comparison between recent related papers and the proposed method. The contribution of this paper is two updates of Amdahl's equation and the interpretations. The first equation suggests that the performance of multithreaded programs can lead to sublinear speedups, which is supported by our experimental results. Using 4 P-CPUs and 8 E-CPUs, we obtained a sublinear speedup of 6–7.5 times. The second equation suggests that the performance of multithreaded programs can lead to superlinear speedups, which is also supported by our experimental results. Using 8 E-CPUs and 4 P-CPUs, we obtained a superlinear speedup of 15 times.

Our works provide evidence that work-stealing strategy performs well on AMPs. We propose to apply work-stealing to solve the problem of asymmetric performance. If work stealing were unable to overcome the problem, then applied work stealing should result in the P-Cores being more idle and experiencing a decrease in the nominal IPC value when operating in asymmetric mode rather than operating exclusively. However, this is not the case, as the evidence suggests otherwise. To state it more clearly, we found that there is no performance loss due to asymmetric performance on the Intel Core i5 1240P, and there is no decrease in nominal IPC on the P-Cores.

Table 6
Nominal IPC for each CPU of P-CPUs and E-CPUs.

	Nominal IPC		
	P-CPU only	E-CPU Only	Asymmetric mode
CPU 0	4.58	–	4.61
CPU 2	4.59		4.63
CPU 4	4.57		4.64
CPU 6	4.60		4.61
CPU 8		2.09	2.11
CPU 9		2.09	2.11
CPU 10		2.09	2.11
CPU 11		2.09	2.11
CPU 12		2.09	2.11
CPU 13		2.09	2.11
CPU 14		2.09	2.11
CPU 15		2.10	1.42

Table 7
Qualitative and quantitative comparison of recent papers.

Criterion	[32]	[43]	Proposed
Contribution	SF prediction	Flexi-AID in libgomp (OpenMP)	two updates on Amdahl's equation
Performance - metrics	normalized - speedup & SF	normalized - Speedup	absolute - Speedup
Results	Predicted SF = 1.97	Comparable or better 6% - 60%	Sublinear & superlinear 6.05 – 15

A recent paper presented some results of performance evaluation on flexible-AID [43]. The results show that the flexi-AID shows comparable performance on most cases presented and only shows significant improvement on three cases. In the study, the authors compared normalized performance of flexible-AID to the SB's normalized performance.

In our study, we have found that the speedup factor is approximately 2. This finding can be confirmed by dividing the nominal IPC value of the P-Core by the nominal IPC of the E-Core in Table 6. Moreover, a recent paper on evaluating Intel Thread Director [32] has also reported an estimated speedup factor of 1.97.

5. Conclusion

Amdahl's Law is still relevant. What we need to do is to consider the number of processors along with their speedup factor, and incorporate them into the Amdahl's Law equation. This will result in a suitable extension model for the environment of asymmetric performance multicore processors, where the values of P and E are combined into an appropriate number of processors.

We evaluated an example of an asymmetric multicore processor by using a compute-intensive parallel program written in the OpenCilk language. Our findings indicate that the OpenCilk parallel program for rectangle integration performs well on the Core i5-1240P. Furthermore, we observed no decrease in IPC's nominal value in the faster CPUs, indicating that OpenCilk effectively resolves the asymmetric performance problem in shared memory multicore processors.

When we assign the P-CPUs first, there is a small slowdown in the speedup. However, the slowdown is due to the drop in clock frequency as we increase the number of P-CPUs for the program. When we stay with the same scenario and increase the number of CPUs with the E-CPUs, the speedup rate reduces by a factor of 0.5. This is because we use the performance of the P-CPU as a base point, whereas the performance of the E-CPU is two times slower than the P-CPU. On the contrary, when we assign the E-CPU first before the P-CPU, the program performance exhibits a linear speedup. By scaling up the program execution with the P-CPUs, the performance evaluation shows a superlinear speedup. Even though the two cases show different speedups, in the end, both cases have the same performance when they use the same total number of P-CPUs and E-CPUs, which equals 12.

CRediT authorship contribution statement

Adnan: Conceptualization, Methodology, Software, Investigation, Validation, Writing – original draft, Writing – review & editing, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] Cho H-D, Engineer PDP, Chung K, Kim T. Benefits of the big. LITTLE architecture. EETimes 2012. Feb.
[2] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference. AFIPS '67 (Spring), New York, NY, USA: Association for Computing Machinery; 1967, p. 483–5.
[3] Sun X-H, Ni LM. Another view on parallel speedup. In: Proceedings of the 1990 ACM/IEEE conference on supercomputing. 1990, p. 324–33.
[4] Sun X-H, Ni LM. Scalable problems and memory-bounded speedup. J Parallel Distrib Comput 1993;19(1):27–37.
[5] Che H, Nguyen M. Amdahl's law for multithreaded multicore processors. J Parallel Distrib Comput 2014;74(10):3056–69.
[6] Yan B, Regueiro RA. Superlinear speedup phenomenon in parallel 3D discrete element method (DEM) simulations of complex-shaped particles. Parallel Comput 2018;75:61–87.
[7] Hill MD, Marty MR. Amdahl's law in the multicore era. Computer 2008;41(7):33–8.
[8] Woo DH, Lee H-HS. Extending Amdahl's law for energy-efficient computing in the many-core era. Computer 2008;41(12):24–31.
[9] Sun X-H, Chen Y. Reevaluating Amdahl's law in the multicore era. J Parallel Distrib Comput 2010;70(2):183–8.
[10] Rafiev A, Al-Hayanni MA, Xia F, Shafik R, Romanovsky A, Yakovlev A. Speedup and power scaling models for heterogeneous many-core systems. IEEE Trans Multi-Scale Comput Syst 2018;4(3):436–49.
[11] Ciorba FM, Iwainsky C, Buder P. OpenMP loop scheduling revisited: Making a case for more schedules. In: Evolving OpenMP for evolving architectures: 14th international workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings 14. Springer; 2018, p. 21–36.
[12] Kumar R, Tullsen DM, Jouppi NP. Core architecture optimization for heterogeneous chip multiprocessors. In: 2006 International conference on parallel architectures and compilation techniques. 2006, p. 23–32.
[13] Sheridan B, Fineman JT. A case for distributed work-stealing in regular applications. In: Proceedings of the 6th ACM SIGPLAN workshop on X10. X10 2016, New York, NY, USA: Association for Computing Machinery; 2016, p. 32–3.
[14] Kumar R, Tullsen D, Ranganathan P, Jouppi N, Farkas K. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings. 31st annual international symposium on computer architecture, 2004. 2004, p. 64–75.
[15] Balakrishnan S, Rajwar R, Upton M, Lai K. The impact of performance asymmetry in emerging multicore architectures. In: 32nd International symposium on computer architecture. 2005, p. 506–17.
[16] Van Craeynest K, Akram S, Heirman W, Jaleel A, Eeckhout L. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In: Proceedings of the 22nd international conference on parallel architectures and compilation techniques. 2013, p. 177–87.
[17] Saez JC, Pousa A, Castro F, Chaver D, Prieto-Matias M. Towards completely fair scheduling on asymmetric single-ISA multicore processors. J Parallel Distrib Comput 2017;102:115–31.
[18] Saez JC, Castro F, Prieto-Matias M. Enabling performance portability of data-parallel openmp applications on asymmetric multicore processors. In: 49th International conference on parallel processing - ICPP. New York, NY, USA: Association for Computing Machinery; 2020.

[19] Wang M, Ding S, Cao T, Liu Y, Xu F. AsyMo: Scalable and efficient deep-learning inference on asymmetric mobile CPUs. In: Proceedings of the 27th annual international conference on mobile computing and networking. New York, NY, USA: Association for Computing Machinery; 2021, p. 215–28.
[20] Silberschatz A, Galvin PB, Gagne G. Operating system concepts, 10e abridged print companion. John Wiley & Sons; 2018.
[21] Perarnau S, Sato M. Victim selection and distributed work stealing performance: A case study. In: 2014 IEEE 28th international parallel and distributed processing symposium. IEEE; 2014, p. 659–68.
[22] Nakashima R, Yoritaka H, Yasugi M, Hiraishi T, Umatani S. Extending a work-stealing framework with priorities and weights. In: 2019 IEEE/ACM 9th workshop on irregular applications: architectures and algorithms. 2019, p. 9–16.
[23] Varisteas G, Brorsson M. DVS: Deterministic victim selection to ImprovePerformance in work-stealing schedulers. In: MULTIPROG 2014: Programmability issues for heterogeneous multicores. 2014.
[24] Hendler D, Shavit N. Non-blocking steal-half work queues. In: Proceedings of the twenty-first annual symposium on principles of distributed computing. 2002, p. 280–9.
[25] Dinan J, Larkins DB, Sadayappan P, Krishnamoorthy S, Nieplocha J. Scalable work stealing. In: Proceedings of the conference on high performance computing networking, storage and analysis. 2009, p. 1–11.
[26] Adnan, Sato M. Dynamic multiple work stealing strategy for flexible load balancing. IEICE Trans Inform Syst 2012;95(6):1565–76.
[27] Hill MD, Marty MR. Retrospective on Amdahl's law in the multicore era. Computer 2017;50(06):12–4.
[28] Joao JA, Suleman MA, Mutlu O, Patt YN. Bottleneck identification and scheduling in multithreaded applications. ACM SIGARCH Comput Archit News 2012;40(1):223–34.
[29] Jibaja I, Cao T, Blackburn SM, McKinley KS. Portable performance on asymmetric multicore processors. In: Proceedings of the 2016 international symposium on code generation and optimization. 2016, p. 24–35.
[30] Chronaki K, Rico A, Badia RM, Ayguadé E, Labarta J, Valero M. Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on international conference on supercomputing. 2015, p. 329–38.
[31] Chronaki K, Rico A, Casas M, Moretó M, Badia RM, Ayguadé E, et al. Task scheduling techniques for asymmetric multi-core systems. IEEE Trans Parallel Distrib Syst 2016;28(7):2074–87.
[32] Saez JC, Prieto-Matias M. Evaluation of the intel thread director technology on an alder lake processor. In: Proceedings of the 13th ACM SIGOPS Asia-Pacific workshop on systems. 2022, p. 61–7.
[33] Schardl TB, Lee I-TA, Leiserson CE. Brief announcement: Open cilk. In: Proceedings of the 30th on symposium on parallelism in algorithms and architectures. New York, NY, USA: Association for Computing Machinery; 2018, p. 351–3.
[34] Schardl TB, Lee I-TA. OpenCilk: A modular and extensible software infrastructure for fast task-parallel code. In: Proceedings of the 28th ACM SIGPLAN annual symposium on principles and practice of parallel programming. 2023, p. 189–203.
[35] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: An efficient multithreaded runtime system. In: Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming. New York, NY, USA: Association for Computing Machinery; 1995, p. 207–16.
[36] Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. J ACM 1999;46(5):720–48.
[37] Frigo M, Leiserson CE, Randall KH. The implementation of the cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 conference on programming language design and implementation. 1998, p. 212–23.
[38] Frigo M, Halpern P, Leiserson CE, Lewin-Berlin S. Reducers and other cilk++ hyperobjects. In: Proceedings of the twenty-first annual symposium on parallelism in algorithms and architectures. New York, NY, USA: Association for Computing Machinery; 2009, p. 79–90.
[39] Love R. Kernel korner: CPU affinity. Linux J 2003;2003(111):8.
[40] Schardl TB, Kuszmaul BC, Lee I-TA, Leiserson WM, Leiserson CE. The cilkprof scalability profiler. In: Proceedings of the 27th ACM symposium on parallelism in algorithms and architectures. 2015, p. 89–100.
[41] Iliopoulos A-S. Cilkscale reference. 2022. [Accessed on 22 June 2023].
[42] He Y, Leiserson CE, Leiserson WM. The cilkview scalability analyzer. In: Proceedings of the twenty-second annual ACM symposium on parallelism in algorithms and architectures. New York, NY, USA: Association for Computing Machinery; 2010, p. 145–56.
[43] Bilbao C, Saez JC, Prieto-Matias M. Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake. Concurr Comput: Pract Exper 2023;e7814.

Adnan is a senior lecturer at Universitas Hasanudin in Indonesia. He serves as Head of Parallel Computing & IoT at the Informatics Department Universitas Hasanuddin. His research interests include performance engineering and parallel programming with multicore.