
Proceedings Paper:
Gao, W, Fang, J, Huang, C et al. (2021) Optimizing Barrier Synchronization on ARMv8 Many-Core Architectures. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER). 2021 IEEE International Conference on Cluster Computing (CLUSTER), 07-10 Sep 2021, Online. IEEE, pp. 542-552. ISBN: 978-1-7281-9666-4. ISSN: 1552-5244. EISSN: 2168-9253.

https://doi.org/10.1109/Cluster48925.2021.00044

Optimizing Barrier Synchronization on ARMv8 Many-Core Architectures

Wanrong Gao, Jianbin Fang, Chun Huang, Chuanfu Xu
College of Computer Science
National University of Defense Technology, China
{gaowanrong, j.fang, chunhuang, xuchuanfu}@nudt.edu.cn

Zheng Wang
School of Computing
University of Leeds, United Kingdom
z.wang5@leeds.ac.uk

***Abstract*—Synchronization operations are commonly seen in OpenMP programs where a parallel construct often works with an explicit or implicit barrier operation. While OpenMP synchronization has been extensively studied on the traditional x86 CPU architectures, there is little work on understanding OpenMP barrier synchronization operations on ARMv8 high-performance many-cores. This paper presents the first comprehensive performance study on OpenMP barrier implementations on emerging ARMv8-based many-cores. We evaluate seven representative barrier algorithms on three distinct ARMv8 architectures: Phytium 2000+, ThunderX2, and Kunpeng920. We empirically show that the existing synchronization implementations exhibit poor scalability on ARMv8 architectures compared to the x86 counterpart. We then propose various optimization strategies for improving these widely used synchronization algorithms on each platform. We showcase that our optimizations yield 12.6x performance improvement over the GCC implementation and 4.7x improvement over the LLVM implementation, translating to 1.6x improvement over the state-of-the-art best-performing algorithm. We share our experience and practical insights on optimizing OpenMP synchronization operations on emerging ARMv8 multi-core CPU architectures.**

***Index Terms*—Barrier synchronization, ARMv8 many-cores**

I. Introduction

Synchronization is a fundamental operation for parallel programs. Barriers are essential to ensure that no data races occur among concurrently running threads during parallel execution. Hence, a synchronization barrier is often explicitly or implicitly inserted at the end of a parallel region of an OpenMP program to synchronize parallel threads.

Depending on how often synchronizations are performed and the amount of computation given to a parallel worker, the achieved performance can be significantly limited due to the barrier synchronization overhead [1], [2]. Executing a barrier requires all threads to be idle while waiting for the slowest peer. This waiting overhead grows quickly as the number of parallel threads increases [3] - such an overhead could be significant on modern many-core processors as partitioning the computation across more processors means the interval between barriers decreases.

There is extensive work in optimizing barrier synchronizations, with various algorithms proposed in the past [4]–[7]. Indeed, optimizing OpenMP barrier synchronization is heavily studied on conventional shared memory architectures [6], [8]–[10]. However, it is still unclear if existing barrier algorithms remain effective on the new ARMv8 multi-cores. As the ARMv8-based multi-cores are quickly emerging as a promising high-performance computing hardware design, it is essential to revisit the efficiency of barrier synchronization algorithms on the ARMv8-based multi-core systems.

This paper presents the first comprehensive study on barrier synchronization performance on ARMv8 multi-cores. We empirically demonstrate that existing OpenMP barrier implementations are ineffective on ARMv8 architectures. We show that the barrier overhead is several times larger on ARMv8 many-cores compared to the x86 counterparts. Such inefficiency calls for new optimization strategies for barrier synchronization on ARMv8 many-cores. Our study evaluates seven mainstream barrier synchronization implementations [11]–[15] on three representative ARMv8 many-core processors: Phytium 2000+ [16], ThunderX2 [17] and Kunpeng920 [18].

We then analyze the root causes of the ineffectiveness of these barrier implementations. We found that the inter-core communication latency on ARMv8 has a significant impact on the performance of barrier implementations. Existing barrier algorithms typically use a hierarchical tree structure to synchronize parallel threads. However, the tree topology used by current implementations is ill-suited for the ARMv8 processor-core organization that typically groups processor cores into clusters. Since the communication latency within and across clusters can vary significantly, parallel OpenMP threads running on different processor clusters can have vastly different synchronization latency that cannot be ignored. While some barrier implementations have considered the non-uniform memory access (NUMA) pattern across CPUs, they are not optimized for on-chip NUMA communications introduced by ARMv8 many-cores. By ignoring the architecture characteristics, existing barrier algorithms can unnecessarily prolong the waiting time of synchronization and increase the thread contention, leading to synchronization inefficiency.

In light of these observations, we propose new optimizations to improve OpenMP barriers on ARMv8 many-cores. To this end, we extend the granularity of the arrival flag (a variable used by parallel threads to signal their arrival to a barrier) to minimize the impact of OpenMP thread contention by increasing the thread granularity. To match the processor-core latency across core clusters, we revise the synchronization tree to develop a new, better on-chip NUMA-aware synchroniza-

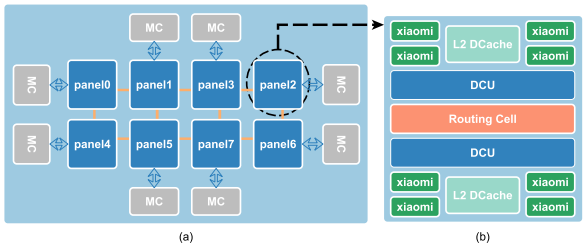


Figure 1. A high-level view of Phytium 2000+. Its 64 processor cores are groups into eight panels (a), where each panel contains eight ARMv8 cores (b)

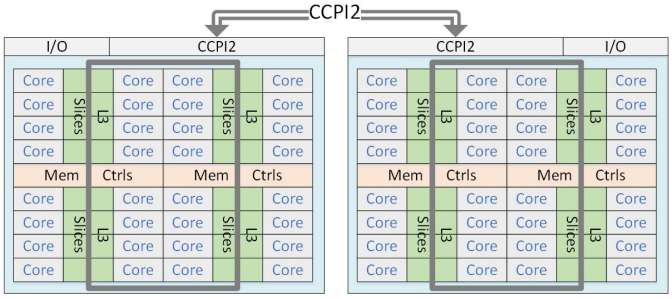


Figure 2. The dual-socket ThunderX2 ARMv8 system.

tion structure for ARMv8. Our design goal is to reduce the expensive remote accesses across processor core clusters.

We show that our new implementation, on average, improves the OpenMP barrier implementation of GCC and LLVM by 12.6x and 4.7x respectively on our evaluation platforms. The results translate to a 1.6x improvement over the best-performing state-of-the-art barrier algorithm. As such, our new implementation represents the most efficient barrier implementation on ARMv8 many-cores seen to date.

This paper makes the following contributions:

- It provides the first comprehensive study of OpenMP barrier efficiency, identifying the limitations of the current barrier implementations on ARMv8 many-cores, and outlining optimization opportunities (Section IV);
- It shows how analytical methods can be developed to analyze and optimize barrier implementations (Section III);
- It presents new barrier optimizations on ARMv8 processors, giving considerable performance improvement over existing approaches (Section V).

II. Background and Motivation

This section provides a description of three ARMv8 many-core architectures, gives an overview of mainstream barrier algorithms and introduces the motivation of our work.

A. ARMv8 Many-Core Architectures

Our work targets three representative ARMv8 many-core architectures, described as follows.

1) Phytium 2000+: Figure 1 gives a high-level view of Phytium 2000+. This processor has 64 ARMv8 compatible processing cores at 2.2 GHz, organized into eight panels. Each panel has eight cores connected through a memory control unit (MCU). Every four cores within a panel form a core group and share a 2MB L2 cache, and each core has a private L1 cache of 32KB for data and instructions. Processor cores within a core group have the same core-to-core communication latency. Inter-core communications outside the core group are more expensive compared to communications within the core group, and the latency varies depending on the distance between the core groups. Table I of Section III summarizes the core-to-core communication latency.

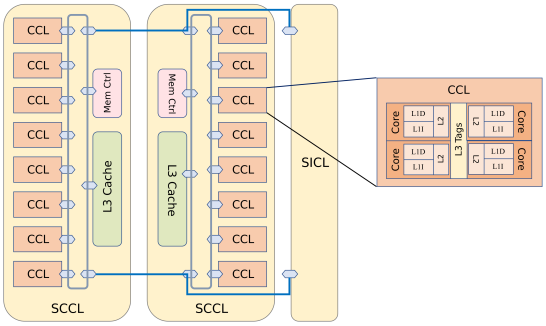


Figure 3. A high-level overview of the KP920 architecture.

2) ThunderX2: Figure 2 shows a typical two-socket ThunderX2 system with 2x 32-core ARMv8 processors at 2.5 GHz. The two processors are connected through a 2nd-generation Cavium's Coherent Processor Interconnect (CCPI2). Each processor core has a 32KB data cache, a 32KB instruction cache, and a 256KB L2 cache. All the cores within a socket share a 32MB last level cache (LLC) arranged as 1MB slices via a dual-ring on-chip bus. The LLC is exclusive, storing the evicted L2 cachelines. Processor cores within the same socket can communicate with each other with a uniform latency of around 24ns. This latency increases to over 140ns for cross-socket processor core communications.

3) Kunpeng 920: Figure 3 shows the 64-core Kunpeng 920 (KP) many-core processor at 2.6 Ghz. Processor cores on KP are organized as two super CPU clusters (SCCL). The processor cores are connected to a super IO cluster (SICL) similar to the Intel uncore component. Each SCCL has its own memory controllers, working as a NUMA node. Within an SCCL, there are eight CPU clusters (CCLs), where each CLL has four cores. Each processor core has 64KB private L1 instruction and data caches as well as 512KB of private L2 cache. All the 64 cores of the chip share a 64MB LLC that is equally partitioned among two SCCLs. Furthermore, every four cores within a CCL are associated with an L3 cache tag partition. Because of the memory hierarchy and core affinity, the inter-core communication latency varies across CCLs and SCCls, as shown in Table III of Section III.

B. Overview of Barrier Algorithms

A barrier synchronization typically includes three phases. The first is the *Arrival-Phase*, where all threads reach the barrier. The threads signal their arrival by modifying one or several shared variables (or semaphores). Then, the threads enter into the *Notification-Phase*, where the last arrival thread notify all other threads that the synchronization has been completed. This operation is typically realized by modifying a flag variable shared among threads. In the final, *Re-initialization-Phase*, all the flags will be reset for reusing.

Our work considers the following representative barrier synchronization algorithms:

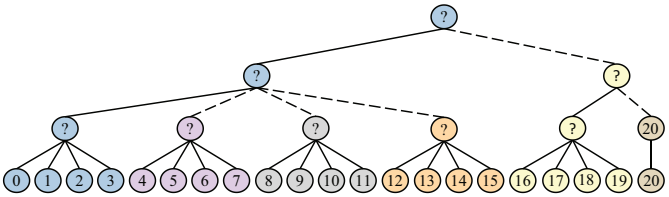
1) Sense: The sense-reversing centralized barrier is a centralized implementation. In the Arrival-Phase, each thread atomically decrements a shared integer (i.e., counter) when it enters the barrier. The initial value of the counter is the number of threads P. When its value becomes 0, it means that all threads have reached the barrier. The algorithm uses a thread-local variable (i.e., sense) and a global variable (or sense) to perform the wake-up process. The local senses are initialized to the opposite value of the global sense (e.g., if the global sense is initialized to 1, the local sense will be initialized to 0 or vice versa). Each thread spins on its local sense when its value differs from that of the global sense. The last arrived thread then informs the other threads by reversing the global sense. All the threads then reverse their local senses before leaving the barrier for the next synchronization. This global wake-up scheme is widely used to implement the Notification-Phase. The GCC OpenMP library adopts this barrier algorithm [19].

2) Tree-based algorithms: In a centralized barrier, all threads write and read the same memory location of the counter, leading to a hot-spot of memory accesses. This causes memory contention in the interconnection network [20], leading to poor scalability. To mitigate the contention issue, researchers proposed several tree-based synchronization algorithms.

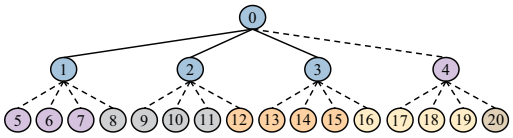
Software combined tree barrier. The classical work presented in [12] constructs a tree of multiple hot spots. Parallel threads are divided into several groups where threads within a group share a counter like the centralized barrier. However, counters across thread groups are stored at different memory locations to avoid a single hot spot. The topology of arrival tree with fan-in[1] of 4 is shown in Figure 4(a).

MCS tree barrier. This algorithm, proposed by Mellor-Crummey *et al.* [13], works by constructing a P-node tree in the Arrival-Phase. Each thread is assigned as a tree node instead of a leaf node as in the combined tree. The topology of the arrival tree with a fan-in of 4 is shown in Figure 4(b).

[1]In a tree, the *fan-in* of a node is the number of children the node has. This term is known as fanout in B+ Tree. In the context of a combined tree barrier, the fan-in is the number of threads in a tree group. It is called fan-in in a barrier tree as thread synchronization is performed bottom-up (i.e., growing from the bottom to the top).



(a) The combined tree barrier



(b) The MCS tree

Figure 4. An arrival tree of the combined tree barrier (a) and the MCS tree barrier (b) using 20 threads as an example. Different processor cores are in different colors on the node. The solid line indicates an operation within a processor core, and a dashed one means operations across core cluster(s). The node marked by a question mark means that the parent node is the last node reached in each group, determined at runtime.

Tournament barrier. The algorithm is similar to a tournament game [14]. A pair of threads play in each round against each other. The winner waits until the loser arrives. The winners play against each other in the next round. The overall winner (the champion) notifies all others about the end of the barrier. In essence, the tournament barrier can be seen as a bottom-up static combined tree with a fan-in of 2. The algorithm adopts global wake-up in the Notification-Phase.

Static and dynamic f-way tournament barriers. Built upon the tournament concept, an f-way tournament barrier [15] converts pairwise synchronization to group synchronization with f threads in each round. The grouping allows us to reduce the critical path length of the tournament. This algorithm is equivalent to a tree with a fan-in of f. The value of f varies across different levels to keep the synchronization tree as balanced as possible. In a *static* f-way tournament barrier, the winner of each group is pre-determined. This is different from a *dynamic* f-way tournament barrier, where the winner is dynamically decided during runtime.

3) Dissemination barrier: The dissemination barrier [14] has no Notification-Phase. This algorithm requires $\lceil \log_2 P \rceil$ rounds of pairwise communication between P threads. In round j, thread i informs thread $(i+2^j)\ mod\ P$ its arrival and waits for the notification from $(i-2^j)\ mod\ P$. Threads signal each other by writing flags. A thread can collect the arrival information of itself and its partners in all previous rounds in each round. In the last round, each thread would know the arrival of all threads.

C. Motivation

Our work is motivated by an observation that the OpenMP barrier on ARMv8 multi-cores is much more expensive than that of the x86 counterparts. As an example, consider Figure 5 that compares the barrier implementation of GCC 8.2.1 and LLVM 10.0.1 on a 32-core Intel Xeon Golden processor at 2.1 GHz and the three ARMv8 processors targeting in this

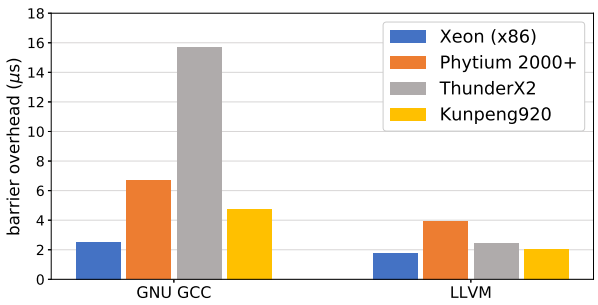


Figure 5. OpenMP barrier overhead (μs) for the GCC and LLVM implementations on various architectures using 32 parallel threads.

work - all have a similar CPU clock frequency (see Section II-A). All the experiments were conducted using the EPCC OpenMP micro-benchmark suite [21] with 32 parallel threads. In the experiments, we run the micro-benchmarks 20 times and report the average performance.

On the Intel platform, the barrier synchronization process takes around $2\mu s$ to complete using 32 threads. By contrast, this overhead can increase to $16\mu s$ using the GCC OpenMP implementation on ThunderX2. This translates to an $8\times$ slowdown on the GCC OpenMP implementation compared to the Intel platform. We also observe a similar trend in the LLVM implementation, albeit the overhead is less significant than GCC on ARMv8. Our work aims to improve the barrier synchronization performance on the ARMv8 platform.

III. Modeling Barrier Cache Performance

To understand the inefficiency of the current barrier implementation, we start by modeling the barrier implementation's performance characteristics. We achieve this by developing a simple yet effective analytical method to model the memory access overhead incurring during parallel synchronization.

As barrier implementations use a number of variables stored in the memory, the barrier synchronization of a parallel thread will generate a number of accesses to (1) the local cache on the processor core where the thread runs (intra-core communications) as well as (2) remote caches across processor cores and cluster (i.e., inter-core communications). As we have outlined in Section II-B, each thread notifies its arrival and waits for the release notification through writing and reading shared data hosted (or to be fetched into) the cachelines. Therefore, our analytical model aims to use the intra-core and inter-core communication latency to model the synchronization overhead.

A. Characterizing Core-to-Core Communication Latency

Our micro-benchmark runs with two threads: one thread places the data, and the other thread accesses the data. We pin each thread to a processor core to ensure that the data is placed in the cache local to the thread. We vary the thread to core affinity to measure the intra- and inter-core communication latency.

Grouping notations. Tables I to III list the measurement on our targeting platforms. As the core-core communication varies depending on the distance and locations between two cores, we group the core-core communications latencies into layers denoted as L_i. Intuitively, as cores in L_0 are in the same core cluster (e.g., a core group in Phytium 2000+), they have a low communication latency. We use N_c to represent the number of threads in each *logical core cluster* found through our measures. The number is 4, 32 and 4 on Phytium 2000+, ThunderX2, and Kunpeng920, respectively. The tables also list the communication latency of local cache access, denoted as ϵ. We use the measurement to derive our performance analysis, detailed in the following subsection.

B. Modeling Cache Load and Store Operations

We use the measured core-to-core latencies in Tables I to III to model cache load and store operations involved in barrier synchronization. To this end, we build analytic models based on the assumption that barriers will incur multiple times within the program execution (which is a typical scenario). Under such scenarios, the variables used for synchronization will reside in the cache as they are repeatedly accessed. There are mainly four types of operations during a barrier synchronization: R_L (local read), R_R (remote read), W_L (local write), and W_R (remote write), described as follows.

Load operations. We use R_L to denote reading a data copy from the local cache. Such a load overhead is small, $O_{R_L} = \epsilon$. We use R_R to denote the operation for loading a data copy from a remote cache. The overhead of this operation varies, depending on the layer the core-core communication belongs to, i.e., $O_{R_R} = L_i$; see Tables I to III.

Store operations. The ARMv8 architecture adopts the write-invalidate coherence protocol for cache consistency. This protocol incurs additional RFO (read-for-ownership) overhead during a store operation. We model the cost of sending RFO to each copy is $\alpha_i L_i$, where α_i is a layer-specific weight and $0 \leq \alpha_i \leq 1$. The local (W_L) and remote store (W_R) overhead can be modeled as $O_{W_L} = O_{R_L} + O_{RFO} = n\alpha_i L_i$ and $O_{W_R} = O_{R_R} + O_{RFO} = (1 + n\alpha_i)L_i$, respectively. Here, n denotes the number of shared copies held by other cores.

IV. Evaluating Barrier Algorithms

This section first gives a brief performance overview of the OpenMP barrier implementations in GCC and LLVM. It then provides an in-depth comparison of seven mainstream barriers algorithms on three ARMv8 many-core platforms.

A. Barrier Performance in GCC and LLVM

GNU GCC and LLVM are regarded as the most widely used open-source compiler infrastructures. Thus, we measure the overhead of the barrier primitive for the GNU and LLVM OpenMP implementations on the three ARMv8 multi-cores. We use GCC 8.2.1 and clang 10.0.1 as the target compiler for all evaluation platforms. We use the EPCC micro-benchmarks to measure the barrier overhead using 1 to 64 threads. We run each benchmark 20 times and report the average performance.

Table I
CORE-TO-CORE LATENCIES(NS) ON PHYTIUM 2000+

	ϵ (local)	L_0 (within a core group)	L_1 (within a panel)	L_2 (panel 0-1)	L_3 (panel 0-2)
latency(ns)	1.8	9.1	42.3	54.1	76.3
	L_4 (panel 0-3)	L_5 (panel 0-4)	L_6(panel 0-5)	L_7 (panel 0-6)	L_8 (panel 0-7)
latency(ns)	65.6	61.4	72.7	95.5	84.5

Table II
CORE-TO-CORE LATENCIES(NS) ON THUNDERX2

	ϵ (local)	L_0 (within a sockect)	L_1 (across sockect)
latency(ns)	1.2	24	140.7

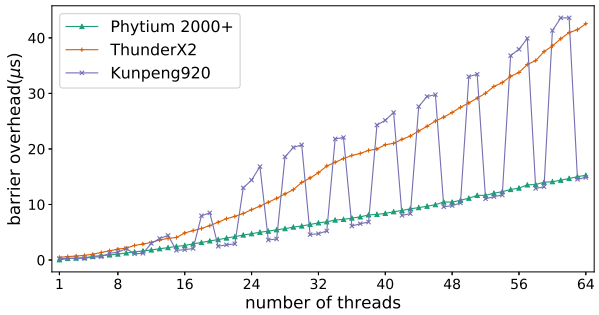
Table III
CORE-TO-CORE LATENCIES(NS) ON KUNPENG920

	ϵ (local)	L_0 (within CCL)
latency(ns)	1.15	14.2
	L_1 (within a SCCL)	L_2 (across SCCL)
latency(ns)	44.2	75

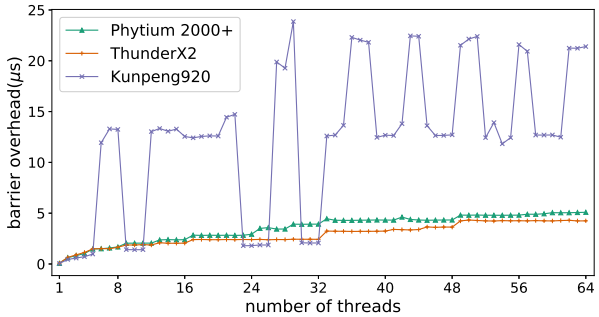
We found the noise across runs to be small, less than 2%. During our evaluation, each thread is pinned to a distinct physical core. Note that we use the same evaluation setting throughout the paper.

Figure 6(a) shows that the barrier overhead for GNU GCC implementation increases over the number of threads for both Phytium 2000+ and ThunderX2. By diving into the code, we know that the GCC OpenMP implementation adopts a sense-reversing centralized barrier algorithm. On Kunpeng920, the barrier performance fluctuates dramatically over threads. Overall, Phytium 2000+ performs the best in GCC OpenMP among the three ARMv8 many-cores.

By contrast, the LLVM OpenMP shows different performance behaviors from the GCC OpenMP in Figure 6(b). Because LLVM uses a tree-based algorithm (i.e., a hypercube-embedded tree), the barrier performance has been improved significantly. The overhead using 64 threads on Phytium 2000+ and ThunderX2 is reduced by 3x and 10x, respectively, compared to that of the GCC OpenMP. We also note see that the barrier overhead on Kunpeng920 has been reduced, but the performance numbers look unstable.



(a) GNU GCC



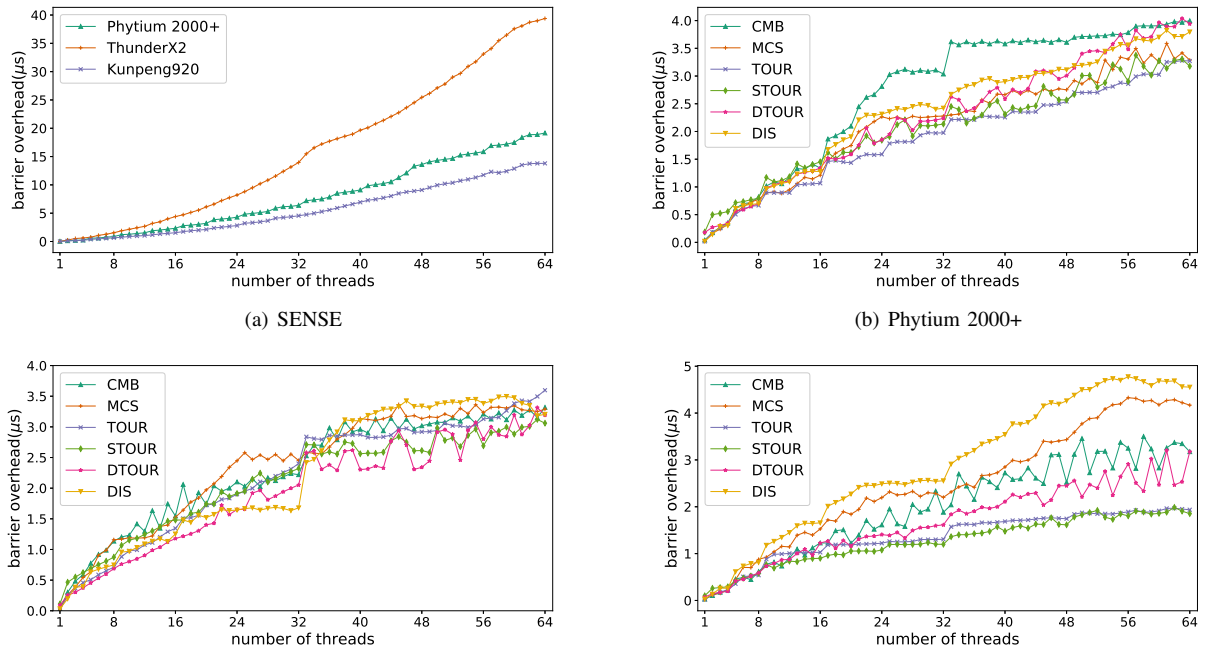
(b) LLVM

Figure 6. The barrier performance of the GNU GCC and LLVM OpenMP implementation on three ARMv8 multi-cores.

B. Performance of Current Barrier Algorithms

Setup. We implement the seven barrier algorithms described in Section II-B, including the sense-reversing centralized barrier (SENSE), dissemination barrier (DIS), software combined tree barrier with a fan-in of 2 (CMB), MCS tree barrier (MCS), tournament barrier (TOUR), static f-way tournament barrier (STOUR) and dynamic f-way tournament barrier (DTOUR). We use the C programming language to implement the algorithms. We develop a micro-benchmark using the OpenMP parallel pragma to parallelize the code and run the micro-benchmark using each barrier algorithm. We compile the program using GCC v8.2.1. We also pin the threads to a physical core to reduce the noise of measurements. Figure 7 shows the scalability of each barrier algorithm when running the micro-benchmark using 1 to 64 threads. To aid clarity, we separate the results of SENSE from others in Figure 7(a), as this algorithm is much more expensive than others.

SENSE. This algorithm is used to implement the barrier primitive in the GCC OpenMP library (i.e., GNU libgomp). As shown in Figure 7(a), the overhead of this algorithm grows linearly as the number of parallel threads increases. This algorithm gives poor scalability as it uses a global shared variable to coordinate the synchronization of parallel threads. Because multiple threads try to load and store to the same memory location multiple times, multiple cache controllers local to the active cores will attempt to prefetch the data simultaneously. It increases the contention of the network controller, whose overhead grows quickly as the number of processor cores increase. This algorithm also manifests high overhead on ThunderX2 than the other two platforms. We also note that our implementation has similar runtime compared



(a) SENSE (b) Phytium 2000+

(c) ThunderX2 (d) Kunpeng920

Figure 7. Overhead comparison between different barrier algorithms on the evaluated ARMv8 platforms.

to the native implementation of libgomp, suggesting that our implementations are effective. Later in other experiments, we compare directly to the libgomp barrier implementation.

CMB & MCS. The synchronization tree of MCS has fewer levels than the combined tree (CMB). As a result, MCS group more parallel threads at a tree-level compared to the combined tree. However, this strategy is ineffective when the number of parallel threads is large because threads at a synchronization point can run across the processor core cluster (Figure 4). In other words, it leads to more L_i $(i > 0)$ communications (Section III-B) than the combined tree and the tournament barrier in the Arrival-Phase. This can be observed from Figures 7(c) and 7(d), where the overhead of MCS becomes higher than CMB when using more than 8 threads. While MCS gives similar performance over the tournament barrier (TOUR) on Phytium 2000+ and ThunderX2 due to the use of the binary-tree wake-up scheme at the Arrival-Phase, it has a significantly higher overhead than the tournament barrier on Kunpeng920. The results show that the tree-based barriers do not give a portable performance on our evaluation platform.

DIS. The overhead of the dissemination barrier (DIS) has a spike using 2, 4, 8, 16, and 32 threads. Each tipping point occurs when the critical path length on its synchronization hierarchy increases. While DIS performs well in a distributed environment [22], [23], it gives poor scalability on the core-cache organizations of ARMv8. The poor performance is because of two reasons. First, the concurrent memory accesses for setting flags during pairwise communications (Section II-B) increase the contention of the on-chip network of the many-core processor. Secondly, when the number of threads exceeds the number of cores in a cluster, N_c, the dissemination algorithm introduces remote memory access for L_i $(i > 0)$ core-core communications in each round. By contrast, the tree-based algorithms like MCS and STOUR only incur such overhead in the last few layers (synchronization rounds), which have lower overhead than DIS.

TOUR, STOUR & DTOUR. Both the static (STOUR) and dynamic (DTOUR) f-way tournament barriers are variants of the tournament barrier (TOUR). Compared with TOUR with a fixed fan-in of 2, the overhead of STOUR and DTOUR fluctuate more because they have a variable fan-in for each round. We see that these three algorithms perform well on all three ARMv8 processors. This is because their tree structures are suited for the hierarchical core-cache organization. The static algorithms, TOUR, and STOUR, perform best on Phytium 2000+ and Kunpeng920. The dynamic tournament on ThunderX2 performs better than the static alternative. Based on the observation, we choose to use the static tournament barrier as our starting point to design our optimization strategies.

V. Our Barrier Optimization

A. Implementation Baseline

We choose the static f-way tournament algorithm (STOUR) as the starting point to improve because it gives the best overall performance during our initial evaluation (Section IV). Another reason is that its arrival tree structure matches the hierarchical core-cache organization on our targeting ARMv8 many-cores. It allows us to use a suitable thread grouping

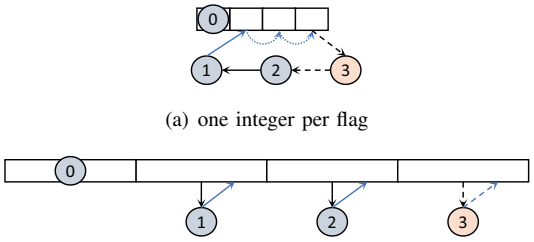
strategy (Section III-A) to maximize the chance of mapping the synchronization threads within the same core cluster during each synchronization round to reduce the expensive cross-cluster communications. Meanwhile, the advantage of using a static barrier algorithm is that this implementation does not have the overhead introduced by atomic instructions of a dynamic scheme. Our implementation focuses on minimizing the arrival and notification phases, which dominate the overhead of STOUR (Section IV).

B. Optimizing the Arrival-Phase

Optimization goals. At each synchronization point of STOUR, a thread (node) indicates its arrival by setting a flag shared with its parent thread. The parent thread continuously polls the arrival flags of all child threads to check if all children threads have reached the barrier. Therefore, the number of children (i.e., fan-in) and the number of bytes of the arrival flag occupies can significantly affect the barrier performance. Intuitively, the first parameter determines the number of concurrent threads that participate in a children-parent synchronization, and using a suitable number can reduce the expensive cross-core-cluster communication. Moreover, choosing an appropriate arrival flag size can avoid multiple flags being cached in the same cache line, which leads to cache conflict when multiple processor cores try to load/store the same cache line. Our optimization aims to find the optimal settings for these two parameters.

1) Determining the arrival flag size: The source publication of STOUR [15] uses a 32-bit arrival flag, which leads to a fan-in value, f, of 2 or 8. This allows the flags used by the children nodes and their parent nodes to be packed into a single cache line for most cache designs, as depicted in Figure 8(a). Here, the parent node has three children (1, 2, and 3), leading to a fan-in of 4 (including the parent node itself). The advantage of this approach is that the parent node needs just one R_R operation (Section III-B) to check the arrival of all its children. However, this strategy is ineffective on ARMv8 for the following reasons. First, it forces all children to write into the cache line when signaling their arrival, for which the write operations must perform in sequential. The sequential write thus limits the write performance. Secondly, since a cache line in ARMv8 can hold up to $16\times$ 32-bit flags, the arrival flags used by different parent nodes can reside in the same cache line. This will, in turn, introduces mutual interference among sub-trees of the synchronization tree. In such a scenario, a store issued by children can lead to an invalid cache when a parent of other sub-trees (that runs on a different processor core) polls it children. Thirdly, as the order of the write operations is nondeterministic, the same cache line may have to be moved back and forth among the children nodes. This can have a detrimental effect when a child node does not reside in the same core cluster as its siblings because two remote W_R operations will incur.

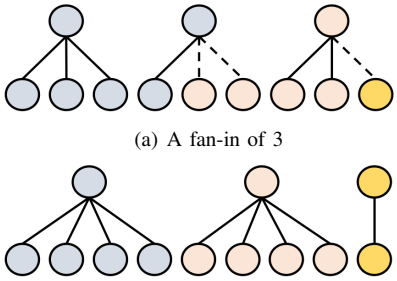
We mitigate the issue by representing the flag of each child node with a cache line. The cache line holds 16 bytes on Phytium 2000+ and ThunderX2 and 32 bytes on Kunpeng920.



(a) one integer per flag

(b) one cacheline per flag

Figure 8. Read and write operations at a barrier point with different size of the arrival flag. Node 0 is the parent of node 1-3. Node 3 is not in the same core cluster with other nodes. The solid line represents local operations, the dashed one represents remote operations (across NUMA node). Black line means write operations and blue one means read operations.



(a) A fan-in of 3

(b) A fan-in of 4

Figure 9. The arrival tree with different fan-in on Phytium 2000+. Nodes of the same color belong to the same core cluster. The solid line represents L_0, and the dashed line represents L_1.

In this way, the write operation of each child can run in parallel. The number of memory operations at one synchronization point is $W_R + (f-1) \times R_R$ in the best case and $(f-1) \times W_R + (f-1) \times R_R$ in the worst case. This technique reduces the number of W_R from $f-1$ to 1 in the best case. Thus, we can eliminate mutual interference between subtrees for better parallelism at each level of the tree.

2) Selecting a suitable fan-in: In the origin algorithm, the fan-in varies across different levels of the arrival tree. The main idea is to calculate a fan-in that makes the tree as balanced as possible based on the number of threads participating in each level of synchronization. But we find the fan-in that maintains tree balanced may destroy the grouping effect of the tree on our platforms, resulting in more inter-core cacheline movements L_i $(i > 0)$. Taking Phytium 2000+ as an example, choosing a fan-in of 3 can get a balanced tree when using 9 threads (Figure 9(a)). But compared with a tree with the fan-in of 4 (Figure 9(b)), it involves more remote cacheline movements (L_1). Because the number of cores in a core cluster, N_c, is 4 or 32, we recommend fixing fan-in to be a power of 2, aiming to avoid remote cacheline movements with L_i $(i > 0)$.

The following-up question is to select a suitable fan-in. The key is to weigh the length of the critical path and the synchronization cost of each layer. We calculate the optimal fan-in by modeling the overall cost in the Arrival-Phase. Our

model is built upon two assumptions. First, since the arrival flag of each node has only one copy in its parent node, we assume O_{W_R} is $(1+\alpha_i) \times L_i$. Second, we focus on the best case, where the number of memory operations is $W_R + (f-1) \times R_R$ at a barrier point. Based on the two assumptions, we can obtain the total cost of the Arrival-Phase in (1). Then, we calculate the f, which minimizes $T(f)$ by taking the derivative of $T(f)$ with respect to f:

$$\begin{aligned} T(f) &= \lceil log_f P \rceil (O_{W_R} + (f-1)O_{R_R}) \\ &= \lceil log_f P \rceil ((1+\alpha_i)L_i + (f-1)L_i) \\ &= \lceil log_f P \rceil (f+1)L_i \end{aligned} \tag{1}$$

$$T^{'}(f) = \frac{L_i \ln P((\ln f - 1)f - \alpha_i)}{f \ln^2 f} \tag{2}$$

According to (2), $T^{'}(f) = 0$ when $(\ln f - 1)f = \alpha_i$. Because $(\ln f - 1)f$ is monotonically increasing and $0 \leq \alpha_i \leq 1$, we can get $2.718 \leq f \leq 3.591$. Thus, $f = 3$ or $f = 4$ may be the optimal solution. Given that f prefers a power of 2, we select $f = 4$.

C. Optimizing the Notification-Phase

In the Notification-Phase, the root node wakes up other threads which are spinning locally. This process can be viewed as a broadcast operation. The two most commonly used methods are (1) using a global variable and (2) using a binary tree to perform the wake-up process. These two notification methods are suitable for different architectures.

Global wake-up. The root node sets a wake-up flag called "global sense" to trigger other threads to stop spinning and leave the barrier. The whole process includes a W_L and $(P-1) \times R_R$. Significantly, each of the other P-1 nodes has a copy of the wake-up flag. So the overhead of W_L is $(P-1)\alpha_i L_i$. The overhead for other P-1 threads to poll the same cacheline can be modeled as $O_{R_R} + c(P-1)$ [24], where c is a coefficient used to indicate possible contention caused by several readers moving the same cacheline. This contention coefficient depends on different processors, and it can be zero. The total overhead of global wake-up for P threads is T_{global}.

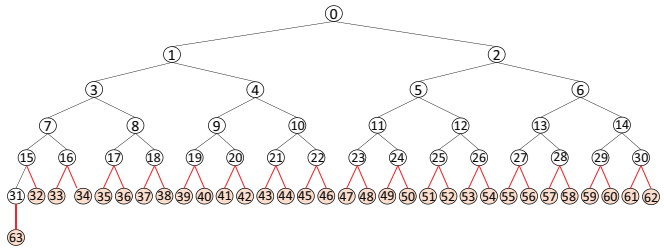
$$\begin{aligned} T_{global} &= (P-1)\alpha_i L_i + O_{R_R} + c(P-1) \\ &= ((P-1)\alpha_i + 1)L_i + c(P-1) \end{aligned} \tag{3}$$

Binary tree wake-up. This wake-up process is spread on a binary tree. In the binary tree, each node n connects to at most two children, that is, node $2n+1$ ($n < \lceil \frac{P-1}{2} \rceil$) and node $2n+2$ ($n < \lceil \frac{P-2}{2} \rceil$). For P threads, the binary tree has $\lceil log_2(P+1) \rceil$ levels. Each parent node writes the local wake-up flags of its child nodes sequentially. At each wake-up point, two W_L and two R_R are required. The copy of each wake-up flag only exists in the child node. So the overhead of W_L is $\alpha_i L_i$. Two R_R can be performed concurrently because they access different cachelines. The total cost of binary tree wake-up for P threads is T_{tree}.

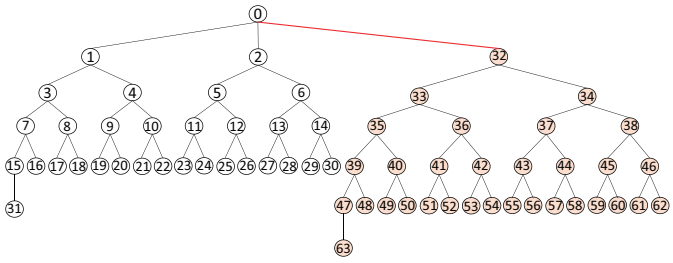
$$\begin{aligned} T_{tree} &= \lceil log_2(P+1) \rceil (\alpha_i L_i + L_i) \\ &= \lceil log_2(P+1) \rceil (\alpha_i + 1)L_i \end{aligned} \tag{4}$$

Since α_i and c will have different values on different processors, the performance of the two wake-up methods varies according to processors. Our empirical results show that the global wake-up is suitable for Kunpeng920, while the binary tree wake-up is suitable for Phytium 2000+ and ThunderX2.

NUMA-aware tree wake-up. The binary tree is not totally suitable for the hierarchical core-cache organization. Taking ThunderX2 as an example (Figure 10(a)), the binary tree generates too many remote accesses with L_i ($i > 0$), which account for half of the total number of remote accesses. We propose a new NUMA-aware tree topology to reduce the number of remote accesses with L_i ($i > 0$).



(a) binary tree topology



(b) NUMA-aware tree topology

Figure 10. Two kinds of wake-up tree topology on ThunderX2. The red line means remote access with L_i ($i > 0$).

In our NUMA-aware tree, the nodes are divided into two categories: the master node and the slave node. The master node refers to the first node in each NUMA node (core unit), and the other nodes are slave nodes. During the wake-up process, the number of children of the master node is no longer two but four, including two other master nodes and two slave nodes. The slave node still has only two child nodes. The children (n_{child}) of a node (n) can be calculated as (5), where N_c is the number of cores in a core cluster.

$$n_{child} = \begin{cases} 2n + N_c, & n \mid N_c \ and \ n < \lceil \frac{P-N_c}{2} \rceil \\ 2n + 2N_c, & n \mid N_c \ and \ n < \lceil \frac{P-2N_c}{2} \rceil \\ 2n + 1, & n \nmid N_c \ and \ n < \lceil \frac{N_c-1}{2} \rceil \\ 2n + 2, & n \nmid N_c \ and \ n < \lceil \frac{N_c-2}{2} \rceil \end{cases} \tag{5}$$

The NUMA-aware tree topology is shown in Figure 10(b). By changing the tree structure, while keeping the number of levels of the tree unchanged, we can reduce the number of remote accesses with L_i ($i > 0$). Although there is an extra overhead for the master node to wake up additional

child nodes, reducing remote access with L_i ($i > 0$) can still improve performance.

VI. Performance Results

This section presents how our optimized barrier performs on the three ARMv8 many-core architectures.

A. Optimizing the Arrival-Phase

We compare the barrier overhead of the original static f-way tournament and its two variants on three ARMv8 processors in Figure 11. Comparing to "static f-way" and "padding static f-way" methods in the figure, representing each arrival flag with a cacheline is beneficial in terms of performance. As shown in Figure 11(c), the performance improvement reaches a speedup of up to 1.35x on Kunpeng920. This is due to the fact that, a cacheline on Kunpeng920 holds 64 bytes, and using a 4-byte flag will incur more conflicts than Phytium 2000+ and ThunderX2. Overall, the synchronization overhead increases over the number of threads participating in the synchronization. We also observe that when f varies, the barrier overhead fluctuates significantly. Even using fewer threads leads to a larger synchronization overhead. But we have not observed such a fluctuation when using a fan-in of 4. The "padding static 4-way" performs consistently better than the "padding static f-way", proving the preference of using a fixed fan-in.

We also compare the overhead of static f-way tournament barrier with different fan-in using 64 threads in Figure 13. The best performance is observed with a fan-in of 4 on all three platforms. This is in line with our model result.

B. Optimizing the Notification-Phase

Figure 12 compares the barrier performance using different wake-up methods in the Notification-Phase on the three processors. The results indicate that the binary tree wake-up performs better on Phytium 2000+ and ThunderX2, while the global wake-up is better on Kunpeng920. This is because thread contention on Kunpeng920 has relatively little impact on barrier performance. We also see that the "global" lines meet with the "binary tree" when using fewer than 16 threads on Phytium 2000+, 8 threads on ThunderX2, and 16 threads on Kunpeng920. In other words, when the number of threads is small, T_{global} and T_{tree} are equal.

We compare the performance of the "binary tree" and the "NUMA-aware tree" on Phytium 2000+ and ThunderX2, showing that the latter is more scalable. The two algorithms have the same overhead within 16 threads and 32 threads on Phytium 2000+ and ThunderX2, respectively. When the number of threads is less than the number of cores in a core cluster, N_c, the NUMA-aware tree is equivalent to the binary tree. For Phytium 2000+, the overhead of waking up additional child nodes can be offset by the performance improvement by adjusting the tree structure when the number of threads is 4 to 16.

Table IV
Performance improvement delivered by our optimized algorithm

	Phytium 2000+	ThunderX2	Kunpeng920	Geomean
GCC	8x	23x	11x	12.6x
LLVM	2.7x	2.5x	9x	4.7x
state-of-the-art	1.7x	1.8x	1.4x	1.6x

C. Overall Performance

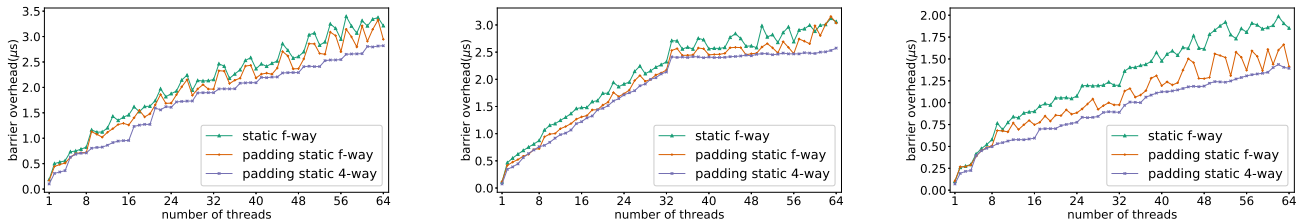
We compare our optimized barrier algorithm with the GNU GCC implementation, the LLVM implementation, and the current best-performing barrier algorithm. The overhead is measured with 64 threads on the three platforms. Table IV shows the speedup of the optimized algorithm compared to the other implementations. We see that our optimized barrier runs, on average, 12.6x and 4.7x faster than the GCC OpenMP and the LLVM OpenMP barrier implementations, respectively. Further, our optimized barrier outperforms the state-of-the-art barrier implementation by 1.6x on average. To conclude, we confirm that our optimized barrier implementation is efficient and scalable on the three ARMv8 platforms.

VII. Related Work

There has been a large body of studies on the performance evaluation and optimization of barrier synchronization.

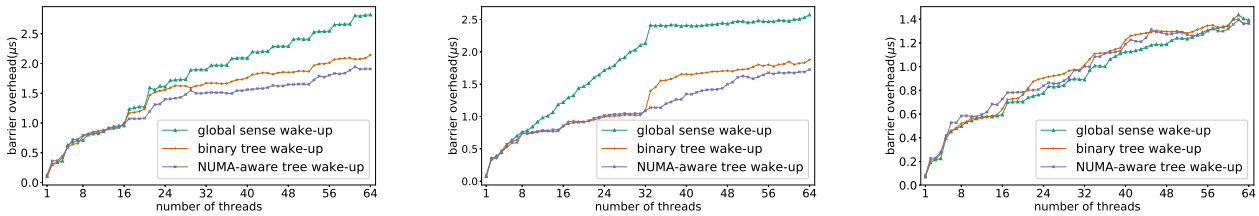
Performance evaluation of barrier algorithms. Nanjegowda *et al.* [25] show how different barrier implementations impact the overheads of OpenMP constructs. They find no single optimal algorithm for all the OpenMP constructs with different numbers of threads and on different platforms. But in most cases, the tournament and dissemination algorithms will have a better performance. Rodchenko *et al.* [8] evaluate typical algorithms on the Intel Xeon Phi coprocessor and have reached similar conclusions. Hoefler *et al.* [22] compare typical barrier algorithms and conclude that the dissemination algorithm is the most promising algorithm for networked clusters. Ramos *et al.* [23] model the dissemination algorithm based on their cache communication model. But the built performance model is based on the message-passing paradigm, which may not be suitable for the shared memory architectures. Ball *et al.* [26] compare several barrier implementations on the Sun Fire 6800 machine. The experimental results show the static f-way tournament barrier can achieve the best performance. Lee *et al.* [27] point out that the multistage network capacity cannot meet the high bandwidth requirements of the dissemination algorithm. When the number of threads increases, the advantage of the dissemination algorithm decreases. We have observed the same issue on the ARMv8 platforms. In this work, our focus is on performance analysis of barrier implementations on the ARMv8 many-core architectures.

Performance optimization of barrier algorithms. Researchers have developed a large number of more efficient and scalable algorithms. Hoefler *et al.* propose the n-way dissemination algorithm [4] based on the inherent hardware



(a) Phytium 2000+ (b) ThunderX2 (c) Kunpeng920

Figure 11. Overhead comparison between static f-way (original), padding static f-way (fill per flag to an entire cache line) and padding static 4-way (fan-in is 4) tournament algorithm.



(a) Phytium 2000+ (b) ThunderX2 (c) Kunpeng920

Figure 12. Overhead comparison between three wake-up methods including global sense, the binary tree and the NUMA-aware tree.

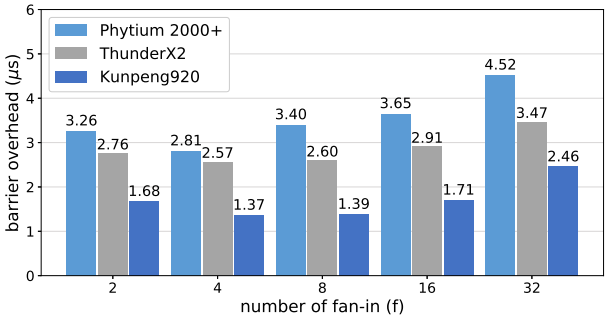


Figure 13. Overhead of static f-way tournament barrier with different fan-in on the three ARMv8 processors.

parallelism inside the InfiniBand network, which can speed up the barrier by 40%. Rodchenko *et al.* propose a hybrid barrier synchronization method based on their evaluation results. This method partitions the process of barrier synchronization into two phases: intra-core and inter-core, using a sense-reversing centralized barrier for the former and using a dissemination barrier for the latter. They have also used the SIMD instructions to optimize barrier, which is proposed by Caballero *et al.* [5]. But the optimizations above have not considered the thread interference and the NUMA effects.

Optimization specifics for shared-memory architectures. Sudheer *et al.* [6] develop an efficient barrier implementation based on the k-ary tree algorithm. As a matter of fact, the k-ary tree they mentioned is an MCS tree. They focus on the impacting factors of hardware prefetching and memory subsystem. They have also considered optimization techniques related to the degree of the tree and the flag representation. But they ignore that the MCS tree topology is ill-suited for the hierarchical inter-core organization. Aravind *et al.* [7] propose a new ring barrier algorithm that can employ minimal remote memory reference, facilitating a larger degree of parallelism. In this work, we focus on optimizing the static f-way tournament barrier algorithm by addressing the issue of thread interference and the NUMA organization.

VIII. Conclusions

We have presented the first comprehensive study of barrier synchronization performance on ARMv8 many-core systems. Our work is motivated by the observation that the widely used OpenMP barrier primitive implementations in GCC and LLVM are significantly more expensive on ARMv8 multi-cores than on the Intel platform. We use analytical methods to model the cache load and store operations incurred during barrier synchronization using micro-benchmarks. We evaluate the performance of seven representative synchronization algorithms, showing that the GCC and LLVM implementations are ineffective in exploiting the processor hierarchy of ARMv8 many-cores. We then present new optimizations to improve a static f-way tournament baseline. Experimental results show that our new implementation, on average, outperforms the OpenMP library of GCC and LLVM by 12.6x and 4.7x, respectively. This translates to a 1.6x speedup over the best-performing current barrier algorithm.

Acknowledgment

This work was partially funded by the National Key Research and Development Program of China under Grant No. 2017YFB0202003, the National Natural Science Foundation of China under Grant agreements 61972408 and 61872294. For any correspondence, please contact Jianbin Fang (Email: j.fang@nudt.edu.cn).

REFERENCES

[1] D. Marinescu and J. Rice, "On the effects of synchronization in parallel computing," Tech. Rep. 88-750, Department of Computer Science, Purdue University, 1988.

[2] H. Davis and J. L. Hennessy, "Characterizing the synchronization behavior of parallel programs," in *Proceedings of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems, New Haven, Connecticut, USA, July 19-21, 1988* (R. L. Wexelblat, ed.), pp. 198–211, ACM, 1988.

[3] C. Tseng, "Compiler optimizations for eliminating barrier synchronization," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995* (J. Ferrante, D. A. Padua, and R. L. Wexelblat, eds.), pp. 144–155, ACM, 1995.

[4] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "Fast barrier synchronization for infiniband/spl trade/," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, IEEE, 2006.

[5] D. Caballero, A. Duran, and X. Martorell, "An openmp* barrier using SIMD instructions for intel® xeon phitm coprocessor," in *OpenMP in the Era of Low Power Devices and Accelerators - 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings* (A. P. Rendell, B. M. Chapman, and M. S. Müller, eds.), vol. 8122 of *Lecture Notes in Computer Science*, pp. 99–113, Springer, 2013.

[6] C. D. Sudheer and A. Srinivasan, "Efficient barrier implementation on the POWER8 processor," in *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pp. 165–173, IEEE Computer Society, 2015.

[7] A. Aravind, "Barrier synchronization: Simplified, generalized, and solved without mutual exclusion," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, pp. 773–782, IEEE Computer Society, 2018.

[8] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján, "Effective barrier synchronization on intel xeon phi coprocessor," in *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings* (J. L. Träff, S. Hunold, and F. Versaci, eds.), vol. 9233 of *Lecture Notes in Computer Science*, pp. 588–600, Springer, 2015.

[9] T. Jammer, C. Iwainsky, and C. H. Bischof, "A comparison of the scalability of openmp implementations," in *Euro-Par 2020: Parallel Processing - 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24-28, 2020, Proceedings* (M. Malawski and K. Rzadca, eds.), vol. 12247 of *Lecture Notes in Computer Science*, pp. 83–97, Springer, 2020.

[10] Z. Yi, F. Chen, and Y. Yao, "A barrier optimization framework for NUMA multi-core system," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 5, 2020.

[11] A. Osterhaug, *Guide to parallel programming on Sequent computer systems*. 1989.

[12] P. Yew, N. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 388–395, 1987.

[13] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[14] D. Hensgen, R. A. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 1–17, 1988.

[15] D. Grunwald and S. Vajracharya, "Efficient barriers for distributed shared memory computers," in *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994* (H. J. Siegel, ed.), pp. 604–608, IEEE Computer Society, 1994.

[16] J. Fang, X. Liao, C. Huang, and D. Dong, "Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+," *J. Comput. Sci. Technol.*, vol. 36, no. 1, pp. 33–43, 2021.

[17] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "A performance analysis of the first generation of hpc-optimized arm processors," *Concurr. Comput. Pract. Exp.*, vol. 31, no. 16, 2019.

[18] HiSilicon, "Kunpeng 920." https://www.hisilicon.com/en/products/Kunpeng/Huawei$\%$20Kunpeng$\%$20920, 2000.

[19] "Gnu offloading and multi processing runtime library: The gnu openmp and openacc implementation," tech. rep., GNU libgomp, 2018. https://gcc.gnu.org/onlinedocs/gcc-8.2.0/libgomp.pdf.

[20] G. F. Pfister and V. A. Norton, ""hot spot" contention and combining in multistage interconnection networks," in *International Conference on Parallel Processing, ICPP'85, University Park, PA, USA, August 1985*, pp. 790–797, IEEE Computer Society Press, 1985.

[21] J. M. Bull and D. O'Neill, "A microbenchmark suite for openmp 2.0," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 41–48, 2001.

[22] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "A Survey of Barrier Algorithms for Coarse Grained Supercomputers," *Chemnitzer Informatik Berichte*, vol. 04, Dec. 2004.

[23] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems: a case-study with xeon phi," in *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013* (M. Parashar, J. B. Weissman, D. H. J. Epema, and R. J. O. Figueiredo, eds.), pp. 97–108, ACM, 2013.

[24] S. Ramos and T. Hoefler, "Cache line aware algorithm design for cache-coherent architectures," *IEEE Trans. Parallel Distributed Syst.*, vol. 27, no. 10, pp. 2824–2837, 2016.

[25] R. C. Nanjegowda, O. R. Hernandez, B. M. Chapman, and H. Jin, "Scalability evaluation of barrier algorithms for openmp," in *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP, IWOMP 2009, Dresden, Germany, June 3-5, 2009, Proceedings* (M. S. Müller, B. R. de Supinski, and B. M. Chapman, eds.), vol. 5568 of *Lecture Notes in Computer Science*, pp. 42–52, Springer, 2009.

[26] C. Ball and M. Bull, "Barrier synchronisation in java," 2008.

[27] C. A. Lee, "Barrier synchronization over multistage interconnection networks," in *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, SPDP 1990, Dallas, Texas, USA, December 9-13, 1990*, pp. 130–135, IEEE Computer Society, 1990.