Characterizing OpenMP Synchronization Implementations on ARMv8 Multi-Cores

Pengyu Wang[§], Wanrong Gao[§], Jianbin Fang[*], Chun Huang[*]
College of Computer Science
National University of Defense Technology
{pengyu_wang, gaowanrong, j.fang, chunhuang}@nudt.edu.cn

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

***Abstract*—Synchronization operations like barriers are frequently seen in parallel OpenMP programs, where an inefficient implementation can severely limit the application performance. While synchronization optimization has been heavily studied on traditional x86 architectures, there is no consensus on how synchronization can be best implemented on the ARMv8 multi-core CPUs. This paper presents a study of OpenMP synchronization implementation on two representative ARMv8 multi-core architectures, Phytium 2000+ and ThunderX2, by considering various OpenMP synchronization mechanisms offered by two mainstreamed OpenMP compilers, GCC and LLVM. Our evaluation compares the performance, overhead and scalability of both compiler implementations. We show that there is no "one-fits-for-all" synchronization mechanism, and the efficiency of a scheme varies across hardware architectures and thread parallelism. We then share our insights and discuss how OpenMP synchronization operations can be better optimized on emerging ARMv8 multi-cores, offering quantified results for future research directions.**

***Index Terms*—OpenMP, Scalability, Synchronization, Performance**

I. Introduction

Synchronization primitives are an essential part of parallel programming languages like OpenMP [1]. To avoid race conditions and ensure correct execution, concurrently running threads have to meet at specific synchronization points. Such synchronization operations are often implemented using locks or shared variables, for which all parallel processes sit idle to wait for the slowest peer.

An inefficient synchronization implementation can severely limit the application performance due to its overhead [2]. This is because contention for obtaining the lock or shared value and waiting delays can substantially degrade the performance of parallel applications. Synchronization can also harm the performance by increasing the bus traffic [3] or creating memory "hot-spots" [4]. This problem worsens on modern multi-cores where the growing number of processors means the synchronization interval decreases when a larger number of competing threads running on the system.

Synchronizations are required in a range of widely used parallel programming patterns, including `fork-join`, `exclusive accessing` and `producer-consumers`. Most OpenMP parallel constructors' implementation typically inserts one or more synchronization points to avoid race conditions among parallel running threads. In OpenMP, a `barrier` like synchronization is used for `fork-join` type of parallelism, including `parallel` and `reduction` region. Similarly, a `mutex` lock can be used to ensure exclusive resource access in OpenMP. This synchronization mechanism is often used together with a `critical` and `atomic` OpenMP directives.

The synchronization implementation varies depending on the OpenMP library vendors. For example, the GNU libgomp OpenMP library used by GCC chooses to use a centralized algorithm to implement the `barrier`, while LLVM adopts a tree-based algorithm. Most of these implementations were tuned on traditional x86 architectures and conventional multi-processors [5], but it remains unclear whether the existing implementations are still efficient on the emerging ARMv8 multi-cores. Given that ARMv8 based CPUs have become a strong contender in the high-performance computing market, it is interesting to know whether the implementation choices of mainstream OpenMP compilers remain effective on ARMv8 multi-cores. Having such information will inform future OpenMP implementation in particular and synchronization optimization in general on ARM HPC systems.

This paper studies OpenMP synchronization implementations on two representative ARMv8 multi-core processors, Phytium 2000+ and ThunderX2. We investigate the performance behaviours of both barrier-related synchronization constructs (including explicit `barrier` constructs and implicit barrier synchronization in work-sharing regions like `parallel for` and `reduction`) and mutex-based synchronization constructs (`critical` and `atomic` directives). We use EPCC benchmark [6] to quantify the barrier overhead of LLVM and GCC compilers. Our evaluation suggests that in addition to the main synchronization overhead, other overhead resulting from the multi-threading management and reduction operations can also have a significant impact on the application performance.

We empirically demonstrate that there is no "one-size-fits-all" synchronization implementation because the efficiency of OpenMP synchronization varies depending on the underlying hardware and the number of parallel threads. Our work evaluates two representatives of the barrier-related synchronization

[§]Equal contribution
[*]Corresponding author

implementations, building upon the tree-based [7] and the centralized algorithm [8], as well as mutex-based synchronization. Our results expose the scalability and performance bottlenecks of different synchronization constructs on two distinct ARMv8 multi-cores. This study thus offers quantified results for optimizing synchronization algorithms on ARMv8 multi-core systems in particular and future ARM HPC systems in general.

II. Setup

This section introduces the architecture features of Phytium 2000+ and ThunderX2, and then describes the experimental configurations and benchmarks.

A. Hardware Platforms

Phytium 2000+ integrates 64 ARMv8 compatible processing cores running at 2.2GHz. Each core has a private L1 cache of 32KB for data and instructions, respectively. Figure 1(a) shows that the cores are partitioned into eight `panels` to form a non-uniform memory access structure. There are two `clusters` in each panel. Each `cluster` contains four processing cores, a 2MB shared L2 cache and one directory control unit (DCU) used to maintain the directory-based cache coherency. The DCUs can access any memory control unit (MCU) according to the corresponding configurations. The `panels` are routed and communicated through the on-chip network interface. The communication latency and bandwidth vary according to the distances of different panels. The floating-point pipeline can combine and execute dual-channel floating-point SIMD instructions to achieve peak performance of 4 double-precision floating-point operations per cycle.

Incorporating a two-socket Vulcan system, ThunderX2 integrates 64 ARMv8 compatible cores (32 cores in a single socket). Each core is equipped with a 32KB L1 data cache, a 32KB L1 instruction cache and a 256KB L2 cache and operates at 2.2GHz in the normal mode, 2.5GHz in the Turbo mode. Figure 1(b) shows that 32 cores in a Vulcan socket share a distributed 32MB L3 cache. The two sockets are connected with a Cavium's coherent processor interconnect (CCPI2) and compose a 2-way SMP node. ThunderX2 also uses the CCPI2 to achieve cache coherence across the two sockets. ThunderX2 also supports 128-bit SIMD instructions.

B. Experimental Configurations

We use the EPCC benchmarks [6] to measure the overhead of the OpenMP constructs. It works by comparing the execution time of a serial code with the execution time of the code in the parallel zone with specific directives.

To minimize the noise of system environment, we modified the source code of the EPCC benchmarks. On the one hand, only one single construct is specified to measure its own overhead. On the other hand, 20 iterations for each directive are run to ensure the accuracy of overhead measurements. We use the environment variable `OMP_NUM_THREADS` to specify the number of threads, and `GOMP_CPU_AFFINITY` to pin each thread to a specified hardware core. Note that we use the `COMPACT` policy in this work, i.e., binding threads according to the core number. In this way can we analyze the performance behaviours from the perspective of processor architectures in a straightforward manner. We use GCC v8.3.0 and LLVM v10.0.1 on both platforms.

III. Results

This section shows the performance of various synchronization constructs (i.e., barrier-related and mutex-based). We enable the comparative analysis combined with different compiler implementations and processor architectures.

A. Barrier-related Synchronization

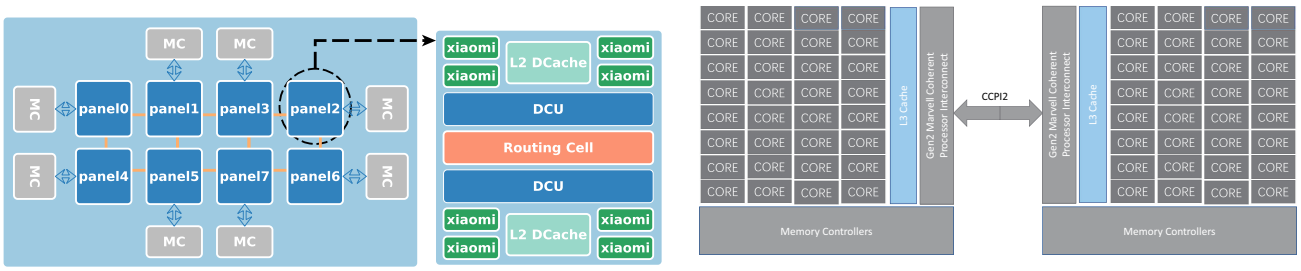
The most commonly used barrier-related synchronization construct is the `barrier` itself. We analyze the explicit barrier and implicit barrier embedded in work-sharing regions (`single` and `for`). We also discuss the overhead of work-sharing constructs through the comparison of `parallel` (`parallel for`). In addition, we measure the overhead of `reduction`, which resembles the barrier synchronization.

1) Barrier: We measure the barrier performance implemented in GCC and LLVM, respectively in Figure 2. We observe that the barrier overhead based on GCC grows linearly over the number of parallel threads.

By contrast, the overhead based on LLVM exhibits a logarithmic growth, which shows much better scalability. We also find that the different growth trends do not vary across processors. The different overhead behaviour between Phytium 2000+ and ThunderX2 is the parameters of linear and logarithm growth, which is determined by their architecture.

We explain the performance difference by analyzing the runtime implementation of `barrier` in GCC and LLVM. The sense-reversing centralized algorithm is used in `libgomp` of GCC to implement barrier synchronization [8], which essentially belongs to a linear algorithm. When taking a closer look at the implementation, we note that the algorithm uses two global shared variables, "counter" and "global_sense", to control synchronization. Each thread atomically decrements the counter to announce its arrival when it enters the barrier. Furthermore, it will spin for release if it is not the last arrival thread. When all threads reach the barrier, the last one reverses "global_sense" to wake up other threads. During the synchronization process, the atomic operation of multiple threads is to be executed in order, and the overhead is accumulated. Therefore, as the number of threads increases, the overhead of the `barrier` grows linearly.

Multiple barrier algorithms are implemented in `libomp` of LLVM [7], e.g., the linear barrier, the tree barrier, the hyper barrier, and the hierarchical barrier. Among them, the hyper algorithm is used by default. It constructs a hypercube-embedded tree to implement synchronization. Threads are divided into several groups. Once all the threads in the group reach the barrier, the parent thread of each group performs the next round of group synchronization until only one thread is left. It is actually a traversal of the tree. Each thread is assigned to a leaf node. In the hypercube-embedded tree, every



(a) Phytium 2000+ (b) ThunderX2

Fig. 1. An architectural overview of Phytium 2000+ and ThunderX2.

four threads are divided into a group, and the thread with the smallest number is set to the parent thread. Figure 3(a) shows the tree structure with 16 threads. Synchronization at each layer of the tree is performed in a parallel fashion. Theoretically, the overhead is proportional to the number of layers of the tree, thereby rendering a logarithmic growth.

When comparing the two processors, we see that the curve slope of ThunderX2 is much larger and the overhead is almost three times as large as that of Phytium 2000+ with 64 threads (42.479 μs versus 14.823 μs), which reflects the potential drawback of centralized barrier on the Vulcan socket. In a UMA Vulcan socket, the core accesses the flag value in the shared L3 cache causing severely busy waiting, which increasing the access latency. Thanks to the efficient DCU and the on-chip interconnection, the synchronization overhead on Phytium 2000+ is relatively small. The performance of LLVM's implementations on the two multi-core CPUs behaves similarly (see Figure 2(b)). Both overhead increase logarithmically, and the maximum numbers are similar. This is because the hyper-embedded tree can adapt to the underlying architectures.

2) Implicit Barrier: We measure the performance of the OpenMP directives including the implicit barrier synchronization. Such directives can be partitioned into two categories: one relates to the management of parallel work regions such as `parallel` and `parallel for`, and the other does not, such as `single` and `for`.

a) Single and For: Figure 4 shows the overhead of `single` and `for` implemented in GCC and LLVM on Phytium 2000+ and ThunderX2. We also compare their overhead with that of `barrier`. It is easy to find that their overhead is similar no matter which compiler and platform are used. There is an implicit synchronization point at the end of the work-sharing region of `single` and `for`. The overhead of these two directives is mainly from this synchronization. While `single` has extra overhead of controlling the single-thread entry of a parallel region with atomic operations, the overhead difference between `single` and `barrier` is evident in Figure 4(b). This is because the atomic operations on ThunderX2 are more expensive than that on Phytium 2000+.

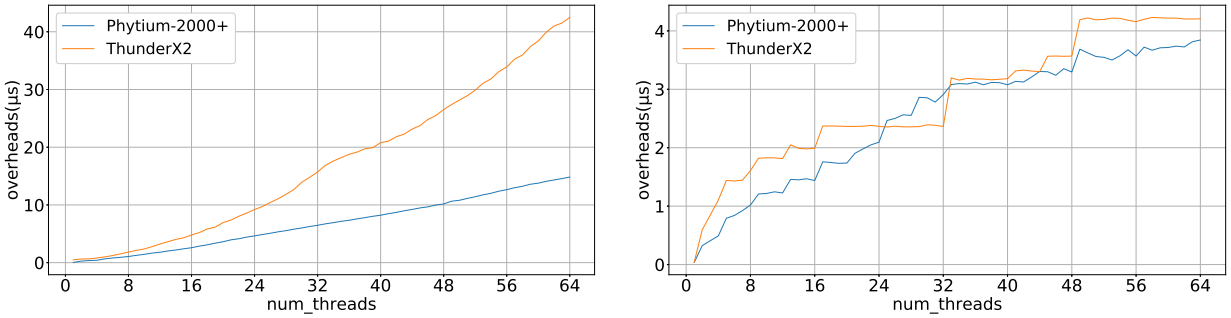
b) Parallel: The `parallel` directive has an implicit synchronization point in its working-share region. So its performance shows exactly the same trend as that of the `barrier` as shown in Figure 5. Nevertheless, it also has additional overhead for thread management. The master creates all required threads in the GCC implementation once starting a parallel region with the `pthread_create` API. In contrast, the LLVM compiler maintains a working thread pool to avoid repetitive thread creation and destruction.

On Phytium 2000+, the GCC thread management overhead does not change in the range of 8 to 64 threads, remaining around 1.8 μs. Since each panel manages its memory module. Threads mapped onto different panels can be created concurrently. However, the thread mapped onto the same panel should be created sequentially. Therefore, the management overhead grows in the range of 1 to 8 threads. While on ThunderX2, the two Vulcan sockets constitute the 2-way SMP architecture and all the 64 cores share the memory. Consequently, the GCC management overhead grows with the number of threads increasing (see Figure 5(b)). With 64 threads, the overhead of `parallel` is twice as much as that of `barrier`. In Contrast to GCC's management overhead, the overhead in the LLVM implementation (see 5) is smaller.

On ThunderX2, the GCC implementation yields lower overhead when using a few threads. When the number of threads is more than six, the LLVM implementations outperform GCC (see Figure 5(b)).

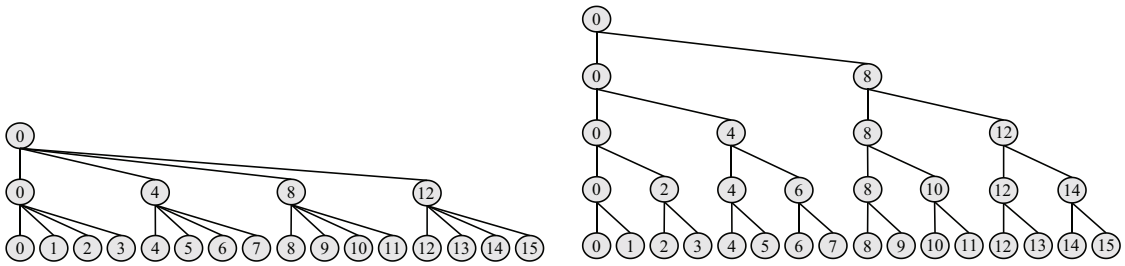
3) Reduction: The OpenMP reduction directive's overhead contains implicit synchronization, thread-management and reduction operations. Figure 6 depicts the reduction overhead on Phytium 2000+ and ThunderX2. As the number of threads increases, the growth trends of reduction overhead are similar to that of the barrier directive on Phytium 2000+ (see Figure 6(a)). For the same thread counts, the overhead of reduction is about 2.5 microseconds larger than that of barrier. We also observe that the overhead gap between reduction and barrier increases over the number of threads for LLVM. When using a few threads, GCC yields a smaller reduction overhead, but LLVM performs better when using more threads.

We dive into the source code of GCC and LLVM to explain the performance behaviours. For GCC's implementation, slave threads will store the private reduction variable in a shared array indexed by thread id when synchronizing. Until all threads finished their computation, the master thread starts to traverse the array to accomplish reduction. Our code analysis shows that the LLVM compiler implements the reduction



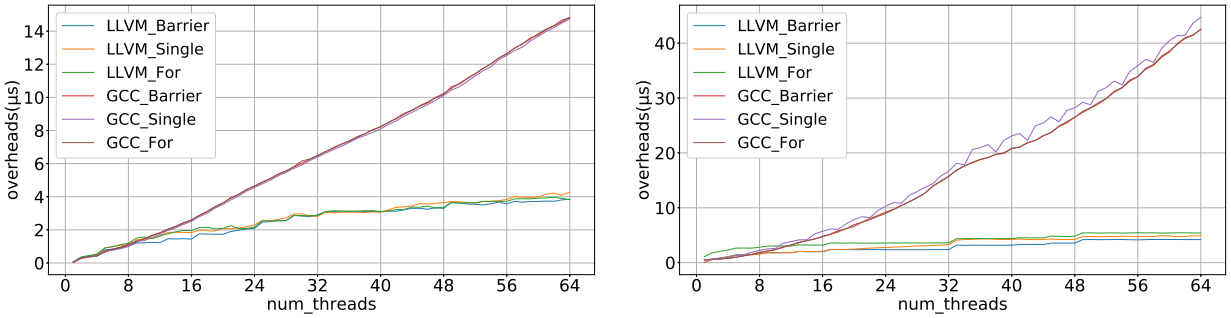
(a) GCC (b) LLVM

Fig. 2. The comparison of `barrier` overhead based two compiler implementation on Phytium 2000+ and ThunderX2.



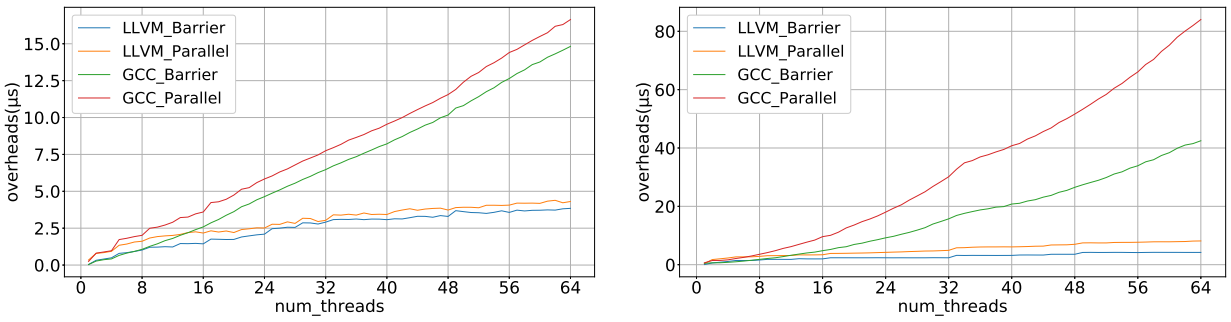
(a) Barrier (b) Reduction

Fig. 3. The hypercube-embedded tree used in barrier (fan-in=4) and reduction (fan-in=2) of the LLVM implementation with 16 threads.



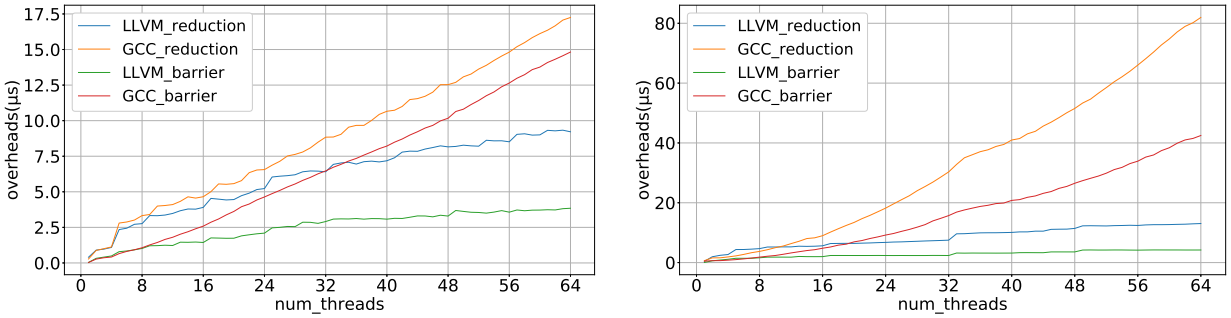
(a) Phytium 2000+ (b) ThunderX2

Fig. 4. The performance of `single` & `for` compared to `barrier`.



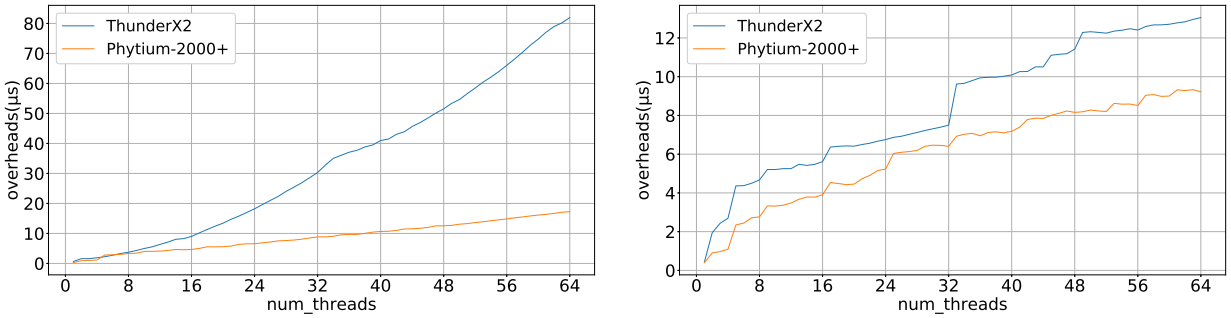
(a) Phytium 2000+ (b) ThunderX2

Fig. 5. The performance of `parallel` compared to `barrier`.



(a) Phytium 2000+ (b) ThunderX2

Fig. 6. The performance of `reduction` and comparison with `barrier`.



(a) GCC (b) LLVM

Fig. 7. The comparison of `reduction` overhead between two processors.

operation in three different ways: critical, atomic and tree. Our experimental results demonstrate that the hyper-embedded tree algorithm is actually used during execution. Different from the explicit barrier, the branch factor in the reduction operation decreases from 4 to 2, which reshapes the hyper-embedded tree form (illustrated in Figure 3(b)) and leads to performance degradation (see Figure 6). Like the parallel directive, the GCC compiler has a smaller overhead when using a few threads on ThunderX2. When the number of threads is larger than 10, the LLVM implementations perform better(see Figure 6(b)).

Figure 7(a) compares the GCC reduction overhead on Phytium 2000+ and ThunderX2. We use the curve-fitting approach to model the reduction overhead on Phytium 2000+ and ThunderX2 in Equation 1 and Equation 2 to intuitively distinguish the growth rate. Obviously, the GCC compiler yields a smaller overhead on ThunderX2. LLVM's reduction overhead on Phytium 2000+ and ThunderX2 are compared in Figure 7(b). The overhead on ThunderX2 is larger than that on Phytium 2000+. The two compilers' implementations both perform better on Phytium 2000+.

We conclude that when using over 32 threads, the overhead on the ThunderX2 increase sharply. We believe that this is because the additional cross-socket memory access overhead.

$$y = 0.249x + 0.786 \tag{1}$$

$$y = 0.0112x^2 + 0.591x - 1.538 \tag{2}$$

B. *Mutex-based Synchronization*

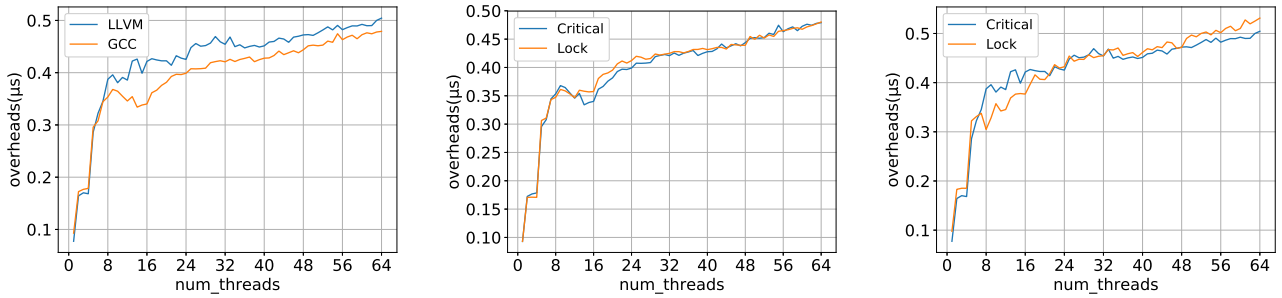
In this subsection, we focus on the mutex-based synchronization on Phytium 2000+ and ThunderX2.

1) Critical and Lock: The `critical` directive is a typical representative of mutex-based synchronization, which ensures that only one thread enters the critical section each time. It is implemented based on the locking and unlocking mechanism, mainly involving two functions omp_set_lock() and omp_unset_lock().

In Figure 8(b) and Figure 8(c), the overhead of `critical` and `lock` are compared on Phytium 2000+. Their similar performance indicates that the dominating overhead of `critical` comes from the usage of locking and unlocking. Figure 8(a) compares the overhead of `critical` based on GCC and LLVM. The overhead of the two compilers is almost the same when the number of threads is fewer than eight. While the overhead of LLVM is smaller than GCC when the number of threads over eight.

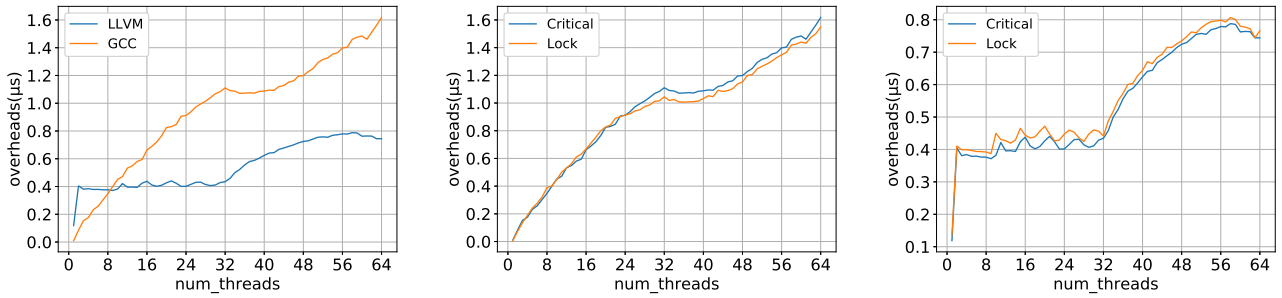
Figure 9(a) shows the `critical` overhead implemented by LLVM and GCC on ThunderX2. When using fewer than 8 threads, the GCC implementation outperforms that the LLVM implementation. on the contrary, the LLVM implementation will have a smaller overhead when launching more threads.

The `critical` implementation in LLVM is identical to the `critical` construct description in the OpenMP API Specification. The thread dispatches corresponding callbacks when it enters and/or exits from the critical region. When the



(a) Critical (b) GCC (c) LLVM

Fig. 8. The performance of `critical` and comparison with `lock` on Phytium 2000+.



(a) Critical (b) GCC (c) LLVM

Fig. 9. The performance of `critical` and comparison with `lock` on ThunderX2.

number of threads increases from 1 to 32, the `critical` overhead based on LLVM remains stable. This performance behaviour is left for future investigation, and we speculate that it relates to the underlying architectures. We demonstrate that the `critical` implementation in LLVM is more suitable for the UMA Vulcan architecture than GCC.

2) Atomic: The `atomic` directive specifies that operation to the variable must be atomic, which is the minimum mutex-based synchronization. It provides a smaller critical region than the `critical` directive. The two compilers (GCC and LLVM) use the same mechanism (i.e. the `__sync` builtins) to implement the `atomic` directive. Figure 10(a) compares the atomic overhead based on two compilers. Their overhead is approximately the same when using few threads. As the number of threads increase, GCC gradually expresses superiority. We compare the performance of `critical` and `atomic` based on two compilers in Figure 10(b) and Figure 10(c). It can be distinctly observed that the overhead of `atomic` is significantly lower than that of `critical` baesd on the same compiler. Since the `atomic` directive can utilize hardware operations to immensely reduce the implementation overhead.

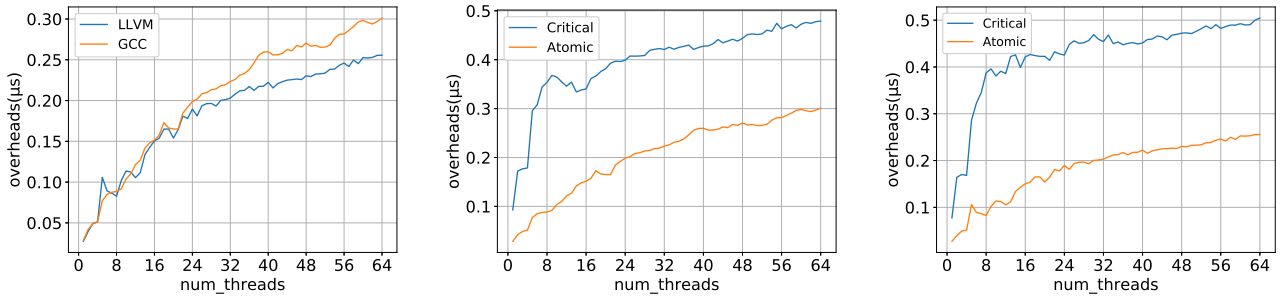
Figure 11(a) compares the atomic overhead implemented by GCC and LLVM. They approximately coincide all the time because the two compilers adopts the same implementation mechanism. As the number of threads increase from 0 to 32, the overhead of `atomic` delivers linear growth. It is because that the time of acquiring the atomic lock continuously increases in a UMA Vulcan socket. Figure 11(b) shows the overhead of `atomic` and `critical` implemented by GCC compiler. They exhibit similar growth due to the consistent function calls (`gomp_mutex_lock` and `gomp_mutex_unlock`). But the `atomic` directive adopts the cheaper atomic lock than default lock, which reduces the implementation overhead.

IV. Discussion

This section summarizes our findings on OpenMP implementations and discusses how the OpenMP synchronizations can be optimized for future directions.

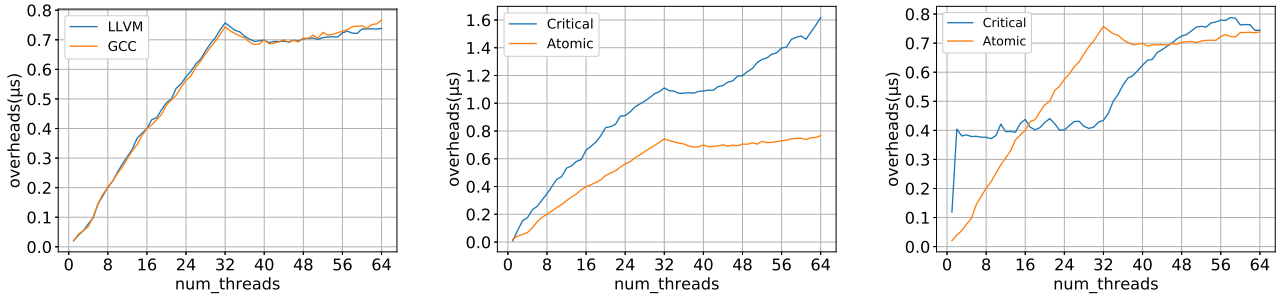
LLVM OpenMP outperforms GCC's on ARMv8 processors for larger thread counts. Overall, the LLVM implementation is more scalable than GCC on both Phytium 2000+ and ThunderX2. For the barrier-related synchronization, GCC uses a centralized barrier which brings "hot-spot" problem while LLVM adopts a hyper-embedded tree barrier. They show linear and logarithmic growth in overhead, respectively. The LLVM implementation yields a smaller overhead when using more threads than GCC, whereas the GCC compiler excels when using fewer threads. Thus, a hybrid barrier implementation for the underlying architecture is required. Furthermore, GCC incurs more thread management overhead than LLVM. This is because LLVM maintains a thread pool to avoid repetitively creating threads when starting a parallel region.

For the mutex-based synchronization, their performance is similar. We note that the performance of `critical` on GCC is even better than LLVM.



(a) Atomic (b) GCC (c) LLVM

Fig. 10. The performance of `atomic` and comparison with `critical` on Phytium 2000+.



(a) Atomic (b) GCC (c) LLVM

Fig. 11. The performance of `atomic` and comparison with `critical` on ThunderX2.

The overhead of OpenMP constructs varies across architectures. There are many differences between Phytium 2000+ and ThunderX2 in terms of the performance of OpenMP constructs. It mainly comes from the architectural disparities. Overall, the overhead on Phytium 2000+ is smaller.

In terms of the centralized barrier, the overhead growth on ThunderX2 is higher than that on Phytium 2000+. In the memory hierarchy of a UMA Vulcan socket, the shared L3 cache is utilized to implement the centralized barrier, which degrades the performance for busy waiting. On Phytium 2000+, the distributed directory control and the on-chip interconnection can provide efficient data accesses. Besides, the communication across the ThunderX2 sockets via CCPI2 incurs an overhead.

Adaptive optimizations for OpenMP implementations are required. On ThunderX2, the GCC implementation can yield better performance when using a few threads, whereas the LLVM implementations excel for a large number of threads. Neither GCC nor LLVM has provided such an adaptive OpenMP implementation based on different underlying architectures.

Besides, efficient synchronization algorithms such as tournament barrier [2], [5], [9], are shown to be scalable on the NUMA architectures, which is regarded as a promising candidate implementation for the ARMv8 processors. In a nutshell, an adaptive implementation is required for future OpenMP implementations.

V. Related Work

This section provides a brief introduction to the related work on measuring the overhead of OpenMP constructs, evaluating its scalability and synchronization.

Measurement of OpenMP overhead The most comprehensive benchmark for OpenMP constructs is EPCC OpenMP micro-benchmark suite [6], [10]. Its working principle is to subtract the execution time of serial code from the execution time of parallel code containing specific OpenMP directive to obtain the overhead of corresponding structure. This paper utilized the EPCC benchmark to the experimental datas. Fürlinger *et al.* [11] proposed a tool to evaluate the runtime characteristics of OpenMP applications. The tool defines the overhead of OpenMP structures into four categories according to the causes, which are derived from synchronous operations, unbalanced workloads, limited parallelism and thread management. The EPCC benchmark is well designed to capture the overhead of complex data environment. However, it does not directly measure the overhead of a single OpenMP directive, so it is more vulnerable to noise on account of environmental impact. Iwainsky *et al.* [12] expanded the EPCC benchmark to evaluate various categories of overhead implemented by OpenMP directives such as the minimum cost of last in first out when the thread reaches the fence, average cost, etc.

Scalability of OpenMP implementation Shirako *et al.* [13] propose two new synchronization constructs in the OpenMP programming model, thread-level phasers and iteration level

phasers to support various synchronization patterns such as point-to-point synchronizations and sub-group barriers with neighbor threads. Liao *et al.* [14] analyzed the performance of OpenMP structure on UltraSparc IV and Xeon processors to explore its performance supported by SMP technology. Since the SMP system and memory hierarchy are not considered in OpenMP design, the architecture and compilation strategy need to be reconsidered to predict the parallel speedup. Iwainsky *et al.* [12] applied automated performance modeling to analyze the scalability in OpenMP structures. They found that the OpenMP structure actually shows linear or superlinear growth instead of expected logarithmic or linear growth. Jammer *et al.* [15] compared the OpenMP synchronization implementation and overhead of LLVM and GCC [12]. They found that the LLVM compiler generally outperformed on Xeon processors, but the gcc compiler outperformed for a small number of threads. With regard to ARMv8 architecture, Michalowicz *et al.* [16] analyzed the performance of OpenMP applications with different compilers on the A64FX platform. But their evaluation was based on practical applications rather than specific directives.

Synchronization evaluation and optimization Ramachandra *et al.* [2] researched the impact of different synchronization algorithms to the overhead of OpenMP constructs. They found that for any OpenMP construct, there is no optimal implementation algorithm due to the different number of threads and architecture. In addition, there has been much work on the evaluation of synchronization algorithms [3], [9], [17]–[20]. Researches have shown that a given synchronization implementation depends on the number of launched threads, architecture, parallel applications and specific system workloads. Ma *et al.* [21] suggested removing redundant fences or implementing DOACROSS parallelism to reduce the overhead of synchronization in OpenMP programs. For non-uniform memory access multi-core systems, Zeng *et al.* [5] proposed a barrier optimization framework and two synchronization algorithms based on the framework. The experimental results on their three NUMA multi-core platforms show that the synchronization algorithm optimized by the framework is as good as the most advanced methods even provides better performance. Huang *et al.* [22] extended the implementation of barrier and reduction directives in OpenMP. Contrasted with the original OpenMP performance, the performance of the extended directives on SDSM system has been significantly optimized. In addition, these two extended directives are defined at the OpenMP instruction level, through which programmers can optimize program performance.

VI. Conclusion

We have presented a comprehensive study of OpenMP synchronization implementations on ARMv8 multi-cores. Our work targets GCC and LLVM by using the EPCC microbenchmark to evaluate the overhead of OpenMP constructs in terms of barrier-related and mutex-based synchronization on Phytium 2000+ and ThunderX2. We observe that the performance of OpenMP constructs varies with regard to synchronization algorithms and thread-management. The LLVM OpenMP compiler shows better performance than that of GCC for larger number of threads. Accordingly, for the reduction operations and parallel region managements, GCC incurs a larger overhead. Thus, the LLVM OpenMP implementation is regarded to be more scalable and efficient. When it comes to the mutex-based synchronization, their implementation overhead varies with the underlying architectures. For future work, we believe that a better OpenMP implementation has to adapt to processor architectures, input workloads and working contexts. Learning-based methods could be used to select the right configuration, e.g., the fan-in, the scheduling work granularity during runtime.

VII. Acknowledgements

This work is partially supported by the National Key Research and Development Program of China under Grant No. 2020YFA0709803, the National Natural Science Foundation of China under Grant Nos. 61972408 and 61872294.

References

[1] "The openmp api specification for parallel programming," OpenMP Home. https://www.openmp.org/, Tech. Rep.
[2] N. R. *et al.*, "Scalability evaluation of barrier algorithms for openmp," in *IWOMP 2009*.
[3] C. A. Lee, "Barrier synchronization over multistage interconnection networks," in *SPDP 1990*.
[4] G. F. Pfister and V. A. Norton, ""hot spot" contention and combining in multistage interconnection networks," in *ICPP'85*.
[5] Z. M. Yi, F. Chen, and Y. Y. Yao, "A barrier optimization framework for NUMA multi-core system," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 5, 2020.
[6] 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.
[7] "Llvm: Llvm openmp runtime library," the LLVM Project, 2018. https://openmp.llvm.org/Reference.pdf, Tech. Rep.
[8] "Gnu offloading and multi processing runtime library: The gnu openmp and openacc implementation," GNU libgomp, 2018. https://gcc.gnu.org/onlinedocs/gcc-8.2.0/libgomp.pdf, Tech. Rep.
[9] D. Grunwald and S. Vajracharya, "Efficient barriers for distributed shared memory computers," in *IPPS 1994*.
[10] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," 2002.
[11] K. Fürlinger and M. Gerndt, "Analyzing overheads and scalability characteristics of openmp applications," in *VECPAR 2006*.
[12] C. Iwainsky *et al.*, "How many threads will be too many? on the scalability of openmp implementations," in *Euro-Par 2015*.
[13] J. Shirako, K. Sharma, and V. Sarkar, "Unifying barrier and point-to-point synchronization in openmp with phasers," in *IWOMP 2011*.
[14] C. H. Liao *et al.*, "Evaluating openmp on chip multithreading platforms," in *IWOMP 2005*.
[15] T. Jammer *et al.*, "A comparison of the scalability of openmp implementations," in *Euro-Par 2020*.
[16] B. Michalowicz *et al.*, "Comparing the behavior of openmp implementations with various applications on two different fujitsu A64FX platforms," in *PEARC '21*.
[17] A. Rodchenko *et al.*, "Effective barrier synchronization on intel xeon phi coprocessor," in *Euro-Par 2015*.
[18] T. Hoefler *et al.*, "A Survey of Barrier Algorithms for Coarse Grained Supercomputers," *Chemnitzer Informatik Berichte*, vol. 04, no. 03, Dec. 2004.
[19] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems: a case-study with xeon phi," in *HPDC'13*.
[20] C. Ball and M. Bull, "Barrier synchronisation in java," 2008.
[21] H. T. M *et al.*, "Barrier optimization for openmp program," in *ACIS/SNPD/IWEA/WEACR 2009*.
[22] C. Huang and X. J. Yang, "Improve openmp performance by extending barrier and reduction constructs," in *ISHPC 2003*.