Combining reduction with synchronization barrier on multi-core processors

Aboul-karim Mohamed El Maarouf, Luc Giraud, Abdou Guermouche, Thomas Guignon

Combining reduction with synchronization barrier on multi-core processors

Aboul-Karim MOHAMED EL MAAROUF Luc GIRAUD Abdou GUERMOUCHE Thomas GUIGNON
IFP Energies nouvelles INRIA LABRI, Université de Bordeaux IFP Energies nouvelles
aboul-karim.mohamed-el-maarouf@ifpen.fr luc.giraud@inria.fr abdou.guermouche@labri.fr thomas.guignon@ifpen.fr

Abstract

With the rise of multi-core processors with a large number of cores, the need for shared memory reduction that performs efficiently on a large number of cores is more pressing. Efficient shared memory reduction on these multi-core processors will help share memory programs be more efficient. In this paper, we propose a reduction method combined with a barrier method that uses SIMD read/write instructions to combine barrier signaling and reduction value to minimize memory/cache traffic between cores, thereby reducing barrier latency. We compare different barriers and reduction methods on three multi-core processors and show that the proposed combining barrier/reduction methods are 4 and 3.5 times faster than respectively GCC 11.1 and Intel 21.2 OpenMP 4.5 reduction

Keywords

Barrier, Butterfly barrier, Reduction, Synchronization primitives, SIMD, Shared memory

I. Introduction

The Intel microprocessor with MIC (Many Integrated Core) architecture coupled with hyperthreading has up to 72 physical cores and can have up to 4 threads per core. This processor is one of the first many-core processors and represents the actual trend of the strong number of core growth in current microprocessors. As in modern architectures, an efficient reduction operation requires a good barrier hence a low latency to help reach high parallel scalability of scientific applications. Many studies propose novel algorithms taking into account the special features of microprocessor architectures. Among available synchronization mechanisms, barriers and reductions are certainly the most widely used. The barriers are parts of the code where threads expect each other. The reductions are mechanisms where threads will exchange values and apply associative operations on these last to get the same scalar value at the end. Usually, to perform a reduction operation we need a barrier that can be before and/or after the reduction operation which is mandatory even though it is time consuming. Alternatively, we can use the synchronized threads in the barrier to compute the reduction operation. In the following, we will be focusing on the second approach. As mentioned in [11] our barrier will go through three phases: the initialization, the incoming and the outgoing, which we will discuss further more. We will start by discussing the proposed barriers in the literature then the reduction operations.

The most well-known and perhaps the oldest is the Centralized barrier, its principle is that the threads increments a global counter at every entry and spin on a global flag until the last one comes and changes it. This method creates a lot of hot-spots during the execution of the code causing a serious degradation of performance as well as memory contentions, [15]. Hot-spots occur when a large number of processors try to access the same global variable simultaneously. At the end of the '80s, the Butterfly Algorithm [4] and Software Combining Tree Algorithm [20] were published to reduce hot-spots memory during the barrier. In his original version, the Butterfly Algorithm performs pairwise synchronization per step with a shared array of flags. It is known to be very efficient and commonly used when the number of threads is a power of two even though a non-power of two version was also proposed in [4]. The Software Combining Tree Algorithm is based on a binary tree, where the last thread reaching its node decrements the global counter of the node. This method greatly reduces the number of simultaneous accesses to a global counter, unlike the Centralized barrier, by using local counters to the nodes. But the threads must go down the tree until one reaches the root and then go up again to the leaves, see Figure 1. The Software Combining Tree barrier works well as the number of threads increases and for any number of threads. Based on [4; 9], the Dissemination and the Tournament barrier were presented in [10]. The Dissemination barrier is an improvement of the Butterflybarrier, it has the same performance as the Butterflybarrier for a number of threads equal to a power of two and scales very well for non-power of two. In a follow-up, [13] has reviewed four barriers from [4; 10; 20] and proposed a new tree-based barrier. By using a local variable for thread busy-waiting, they showed an improvement in reducing the barrier latency. However, their experimental results showed that the Dissemination barrier is the most efficient. Meanwhile, other studies and algorithms have appeared to improve and gain a few more percents [1; 8; 11; 14]. Nevertheless, a major move in the domain, for shared memory systems, was proposed by [7] with the multi-level tree barrier. This barrier is a Software Combining Tree barrier that handles more than two threads per node and can change the size of the group thread following the level in the tree. This approach is competitive because it uses SIMD instructions to wake up groups of threads at each level. It also shows a speedup of up to 2.84 times

regarding the OpenMP barrier with Intel compiler 2013 in the EPCC micro-benchmark [5].
The democratization of the SMT (Simultaneous Multi-Threading) technique in modern processor architectures is forcing developers to adapt their algorithms to take into account the evolution brought on the hardware side. In [17], a new algorithm named hybrid is introduced and consists in : synchronizing at first all the threads running in the same physical core with a Centralized barrier, then using a Dissemination barrier for inner-core synchronization. This algorithm showed 3 times less overhead than the OpenMP barrier with Intel compiler 2014. Another interesting approach is to synchronize threads according to their location in a Non-Uniform Access Memory (NUMA) node. It consists in : synchronizing of the threads on the same NUMA node, in each group the last thread reaching the root will then be designated to continue and synchronize with each other. This method has been developed in [21]. So we see that the recent methods that show good performance on a large number of threads proceed in two steps by mixing two different methods of synchronization. We use the same approach to design the Extended Butterfly barrier. Using this idea of threads group synchronization, we propose a hybrid barrier as [17] but with a Butterflybarrier for inter-group synchronization and a Centralized barrier inner-group synchronization. We chose the ButterflyAlgorithm because its communication patterns are essential for the operation reduction part. The operation to reduce values is common in parallel algorithms. It consists of applying an associative operation to a set of local values to get a scalar value or an array. The simplest case is to reduce values using operations such as sum or multiplication, max or min. In large systems with NUMA, these reduction operations are very critical because the communication between the different memory modules can be very expensive. Reduction combined with a barrier has widely been studied. In [6], authors proposed to combine an operation reduction with a multi-degree combining tree barrier. It exposes an acceleration up to 1.56 times over the reduction of ICC 2013 OpenMP 4.0. However, despite the effort to optimize communication and exploit specificities of different processors, the majority of articles do not address the subject of unused spaces in cache lines (payload) when operation reduction is combined with a synchronization barrier in shared memory. This payload is necessary to avoid false sharing during the barrier. An operation reduction combined with a Tournament barrier proposed in [18], uses this payload by transporting the reduction values and the synchronization flag on the same cache line. This article was the first to propose handling the payload cache line in barrier synchronization during the reduction operation combined with a barrier. This new method shows a speedup of up to 1.53 times over the default OpenMP reduction with compiler GCC 4.5 in the 312.swim_m SPEC OMP2001 benchmark. The same approach is adopted in our Extended Butterfly reduction. Also, we use read/write SIMD instructions to work with the whole cache line at once.
In this work, an algorithm of a reduction operation combined with a barrier for processors with a large number of threads is proposed. The adopted solution uses a modification of the Butterfly barrier called Extended Butterfly barrier and a method to transport the reduction values with the flag of synchronization, which can handle up to 7 double per thread. This method is simplified and optimized by using SIMD instructions to write the 8 double (synchronization flag and reduction values) atomically. The paper is structured as follows : we present the state of the art in barrier algorithms in Section II. Our new barrier will be described in Section III. Then, we will analyze the reduction methods in Section IV and the specificities of Extended Butterfly reduction. Finally, in Section V, the experimental results obtained on different multi-core system processors (Intel Skylake X (SKL), AMD Milan (MILAN) and Intel Knights Landing (KNL)) will also be presented.

II. Related work

In this section, we introduces several known barrier Algorithms. Their descriptions give an idea of their working principle.

A. Sense-reversing Centralized barrier

A naive Centralized barrier Algorithm used a global synchronization flag and a global counter. The first $\mathbf{P}-1$ threads reaching the barrier will increment the counter and move into busy-wait mode. The last thread will release them by toggling the global flag. Each thread will decrement the counter and the last thread will reinitialize the flag. The sense-reversing method improves the naive approach by using local synchronization flags, and each thread will spin on its own local flag. The algorithm can be described by these steps :

1) The first $\mathbf{P}-1$ threads will decrease the counter and spin until their local flag is equal to the global flag.
2) The last thread reaching the barrier will set the global counter to $\mathbf{P}$ and the global flag to the value of the local flag. This will release all busy-waiting threads and prepare the barrier for its next use.

This approach proposed in [13, Algorithm 7] reduces the number of atomic add operations compared to the naive Centralized barrier ([13, Algorithm 6]). The Sense-reversing Centralized barrier gives good performance for small numbers of threads but is not efficient for large numbers of threads.

B. Software Combining Tree barrier

"Divide and rule" is the main principle applied in this case. The threads are gathered into groups of at most k members. We will consider the case of k is at most 2. Within each node of the tree, a Centralized barrier is used, and the last thread to reach the node (colored in Figure 1) will be able to continue to the next node while the other will be waiting, and so on until

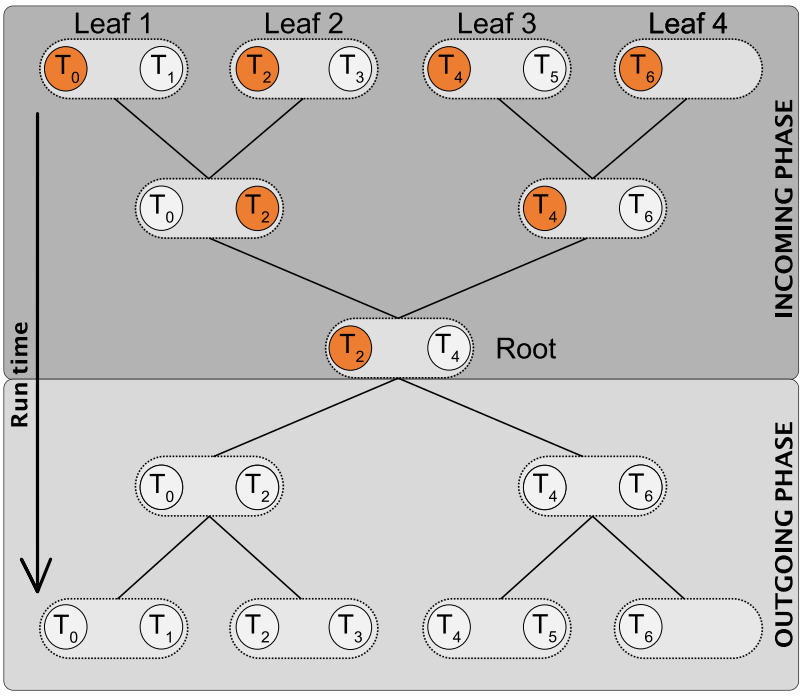


Figure 1: Example of communication schemes in Software Combining Tree barrier with 7 threads. The colored thread is the last to reach the node and the one which will continue to the next node.

the last thread reaches the root node and resets the counter. Then the threads will wake up from the root to the leaves (see the outgoing phase of Figure 1). The algorithm is presented in [13, Algorithm 8]. This approach reduces the serialization effect of the Sense-reversing Centralized barrier by spreading the atomic operations across the nodes. Unlike centralized barriers, this algorithm has logarithmic growth. More recently, a new version of the Software Combining Tree Algorithm in [7] allows for different group sizes from one level of the tree to another and uses a SIMD instruction to release the group in each node (Multi-Degree SIMD Combining Tree Barrier). The authors showed an acceleration up to 2.84 times compared to the OpenMP 4.0 barrier.

C. Linear Centralized barrier

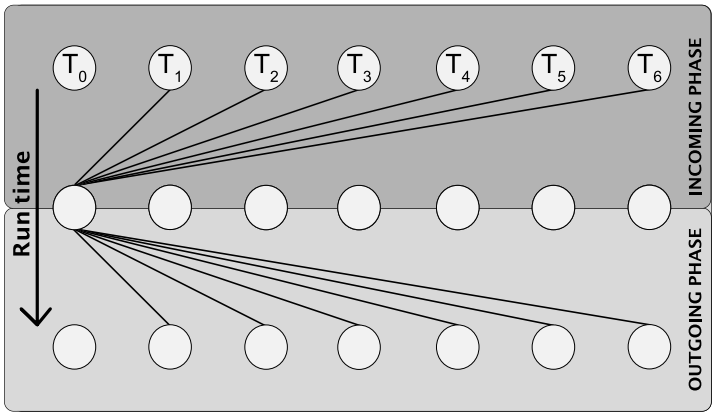


Figure 2: Example of communication scheme in the Linear Centralized barrier with 7 threads.

Threads coming into the barrier will signal their presence by reversing their synchronization flag, and will then go into a busy-wait mode. During the incoming phase, the master thread checks if all the threads have arrived at the barrier. The flags are then updated by the master during the outgoing phase to release the busy-waiting threads. Figure 2 illustrates how the information is handled by the master thread. This barrier contrasts with the Sense-reversing Centralized barrier because it

chooses a thread to check both the incoming and outgoing phases of the barrier. This way, atomic operations are not needed. This approach turned out to be very efficient for a small number of threads.

D. Butterfly barrier

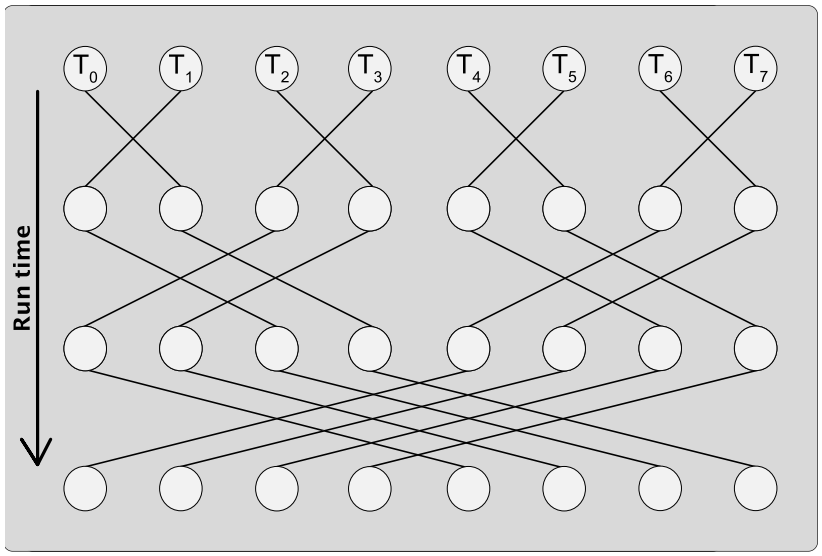


Figure 3: Example of communication schemes in Butterfly barrier with 8 threads.

This barrier has been widely commented on the literature [4; 11; 19]. Thread communication is performed two by two at each step (see Figure 3. The thread i at step r notifies its incoming to the thread $\left((i+2^{r-1})\%2^r\right)+s$, $i \in [\![0, \mathbf{P}-1]\!]$, $r \in [\![0, \log_2(\mathbf{P})]\!]$ and $s = (i-(i\%2^{r-1}))$ if $i \geqslant 2^r$ or $s = 0$ otherwise. This type of barrier only needs $\log_2(\mathbf{P})$ steps to finalize the synchronization. Its main constraint is that it is only efficient with a power of two threads. A version for any number of threads is proposed in [4]. For $\mathbf{P}$ threads, $\mathbf{P}$ not a power of two and if $\mathbf{T}$ is the next power of two greater than P, the method consists to simulate the presence of the $\mathbf{T}$-$\mathbf{P}$ threads missing by the threads participating in the barrier. This approach is not efficient for a large number of threads. The Extended Butterfly barrier proposes another way to synchronize a number of threads different to a power of two using this Butterflybarrier. For the remainder, the Butterflybarrier will refer only to the version with a number of threads equal to a power of two.

E. Dissemination barrier

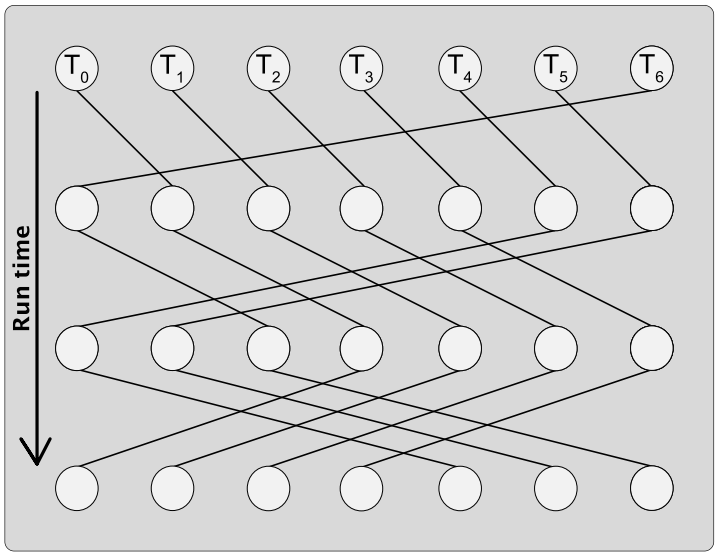


Figure 4: Example of communication schemes in the Dissemination barrier with 7 threads.

As the Butterfly barrier, the thread communication is performed two by two at each step (see Figure 4. Threads need $\lceil log_2(\mathbf{P}) \rceil$ steps to finalize the synchronization. At the barrier step r, the thread i will notify its thread partner $(i+2^r)\%\mathbf{P}$ with $i \in [\![0, \mathbf{P}-1]\!]$, $r \in [\![0, \lceil log_2(\mathbf{P}) \rceil]\!]$. It has been studied in [11; 13; 17]. This barrier works very well for any number of threads. However, the Dissemination barrier needs to create redundant communication paths if the number of threads is non-power of two to ensure that all threads have reached the step r before the next step. As the version of Butterflybarrier for non-power of two threads, the Dissemination Algorithm simulates the presence of the missing **T**-**P** threads by the threads participating in the barrier. Nevertheless, this barrier is one of the most efficient in our state of the art. It has the same logarithmic growth as the Butterflybarrier with an almost constant cost between two powers of two.

III. Extended Butterfly barrier

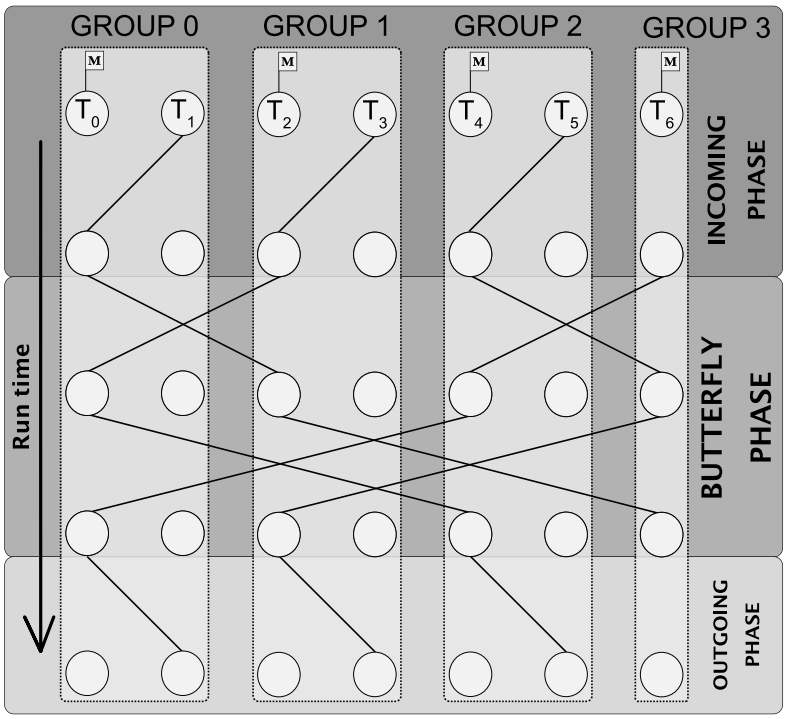


Figure 5: Example of communication schemes in Extended Butterfly barrier with 7 threads.

In this part, we will describe our barrier. During initialization, threads are statically assigned to a group led by a master. The master thread is the thread designated to perform the Butterflyphase. It is also in charge of waking up the other threads in the group at the end of this second phase. To minimize the cost of the synchronization in the groups, we decided to limit the group size to 2 threads. The total number of groups is equal to the previous power of two lesser than **P** *i.e* $nb\_group = 2^{(\lfloor \log_2 \mathbf{P} \rfloor)}$. As the incoming phase of the Linear Centralized barrier, each thread of each group will notify its master of its arrival at the barrier. Once this incoming phase is completed, the masters will then start a butterfly synchronization. At the end of this second phase, each master will be able to release its busy-waiting team. Assuming that the incoming and outgoing phases count for two steps in the barrier, then we will have exactly $\lfloor \log_2(\mathbf{P}) \rfloor + 2$ steps in the barrier. Figure 5 shows the communication patterns of the Extended Butterfly barrier for **P** = 7. If the number of participating threads is equal to a power of two, then creating groups is unnecessary. In this case, only a synchronization butterfly will be necessary, which will require $log_2(\mathbf{P})$. To perform a reduction operation combined with a barrier, the barrier must guarantee non-redundant communication paths between the threads. When the ancestors of the current thread are exactly equal to the threads participating in the barrier (including the current thread), then we qualify the communication as non-redundant. The thread ancestors are all the partners of the current threads. Figure 8 illustrates the ancestors of the thread T_0 for reduction operations with five threads with Extended Butterfly barrier. The non-redundancy property is satisfied for the Linear Centralized barrier, Software Combining Tree barrier, Butterflybarrier, and the Extended Butterfly barrier but not for the Dissemination barrier. This means that the Dissemination barrier is not suitable for the combining reduction operation.

IV. Reduction combined with a barrier

A reduction operation consists in applying an associative operation on each thread's private value to get a result that will be known by all. Let's take the case of a reduction operation with a sum, as shown in Algorithm 1. Each thread stores its

Algorithm 1 Simple master reduction

```
Require: global array : data_thread, global_red;
Require: processor private value : val_red;
1: data_thread[tid]=val_red;
2: barrier();
3: if tid==0 then
4:     global_red=0;
5:     for i=1; i<NTHREAD; ++i do
6:         global_red +=data_thread[i];
7:     end for
8: end if
9: barrier();
```

value in the vector `data_thread` and waits for the whole group to finish this step. Then, the threads will perform the sum on the values of the array. This naive reduction needs two synchronization barriers. Note that Algorithm 1 can be improved if each thread reduces the values into a local variable, then the master thread updates the global variable. In this case, the second barrier (line 1) is useless because all threads will have the result of the reduction operation in a local variable if they need to use it right away, otherwise the result will be stored in the global variable by the master thread for further use. As its name suggests, the reduction operation combined with a barrier is performed at the same time as the threads run through the barrier. This gives the advantage that only one barrier is needed to both synchronize the threads and reduce the values. For example, [6] uses the Software Combining Tree barrier to compute the reduction operation, and the authors in [18] use the Tournament barrier. However, during the thread synchronization, we have to be careful about the communication between threads, especially about the sharing of specific variables.

A. False sharing

A phenomenon that needs to be taken into account for the efficiency of barriers is the false sharing of the synchronization flags. This problem occurs when a shared data is modified and updated frequently by several cores. To comply with the spatial locality property, this cached data will be loaded on the same cache line. The false sharing hides behind the two principles of data locality (spatial and temporal). To avoid false sharing several techniques exist. Concerning the synchronization variables, they must be stored in memory aligned on a cache line boundary. This ensures that two synchronization variables will not share the same cache line.

State	Adress	Data cache							
...	...	...	...	...	...	...	...	...	...
E	0x...	flag sync	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...

Figure 6: Example of core's cache line with payload unused.

This payload has a small memory penalty as most of the cache line will not be used, see Figure 6. One of the main goals of this work is to use this available space on the cache lines to store the partial reduction. Figure 7 illustrates this idea.

State	Adress	Data cache							
...	...	...	...	...	...	...	...	...	...
E	0x...	flag sync	red val 1	red val 2	red val 3	red val 4	red val 5	red val 6	red val 7
...	...	...	...	...	...	...	...	...	...

Figure 7: Example of core's cache line transporting flag and reduction values (e.g., 8x64 bits real numbers).

In the literature, only [18] proposes this method : transporting reduction values on the same cache line as the synchronization flag. Their method is to use a "container" in each node to store both the flag and the variable to be reduced. Since the size of

the container is exactly the size of a cache line, the flag takes 1 bit and the remaining space is used for the reduction value. When the flag and partial reduction value fit the size of the container, the path referred to as "fast" is chosen. If the size of the reduction data-type exceeds the capacity of the container, then a new variable will be created to store the reduction value. This way is referred to as "slow". However, the challenge with this method is that if the flag and partial reduction value exceed the size of the container by 2 bits, then the solution found to use the "fast path" is to exclude the 2 bits of exponent from the partial reduction value. A packing function checks whether it is possible to neglect the exponent bits and then use the "fast path"; if not, the "slow path" will be used to reduce the values. For more details, we refer to [18]. In our method, we rely on SIMD read/write operations to transport flags and partial reduction values at the same time, which allows us to reduce up to 7 doubles for each thread.

1) During the reduction operation, we do not use a global variable to share the reduced values.
2) These values are local to each thread and will be updated as the threads progress in the barrier.
3) SIMD instructions are used by the current thread to atomically update the cache line data of its partner thread.

In addition, we take advantage of the MESI cache coherency protocol. When a thread T_1 modifies a data D owned by another thread T_2 then before using D, T_2 must update its entire cache line. In theory, when threads communicate in the barrier, they can exchange synchronization flags and reduction values without extra cost. But the read/write of the whole cache line must be done atomically.

B. SIMD write atomicity

In Intel and AMD's reports about their respective processor architectures, it is not clear about the atomicity of SIMD instructions. In [2], it is explained that load/store instructions of more than 8 bytes are not guaranteed to be atomic. Similarly, [3] informs users that, except for the list of instructions detailed in [12], "AVX and FMA instructions do not introduce any new guaranteed atomic memory operations". To ensure that the threads can see the change of the flag and the update of the values to be reduced, we need atomic SIMD write operations. By atomic, we mean that a SIMD write is performed in one non-uninterruptible step. So the threads that notice a change in the vector are guaranteed that the entire vector is updated. This atomicity is usually not guaranteed beyond 64 bits of write operations on x86 processors. Nevertheless, we make a simple test to check if we have some atomic behavior for SIMD writes. This test consists of a master thread performing a SIMD write with a synchronization flag at the beginning and some specific values. A reading thread will expect flag state changes and then checks specific values of the vector. If the reading thread does not see the specific values in the vector after the flag state change, the test is marked as failed. The threads are bound to two different cores of the same socket, and the test is performed multiple times for different cache lines. We show in Table I the failure ratio for 10^6 tests over 1000 cache lines for different x86 processors. We see that all processors seem to have atomic SIMD writes at least up to 128 bits. Ivy Bridge processor does not have atomic 256 bits SIMD writes, unlike the other processors. The Skylake-X and KNL processors seem to have SIMD write atomicity up to 512 bits which corresponds to an entire cache line.

Data size	128 bits	256 bits	512 bits
Ivy-Bridge (E5-2680 v2)	0.0	3E-9	Not supported
Rome (Epyc 7H12)	0.0	0.0	Not supported
Milan (Epyc 7763)	0.0	0.0	Not supported
SkyLake-X (Xeon Gold 6140)	0.0	0.0	0.0
KNL (Xeon Phi 7290)	0.0	0.0	0.0

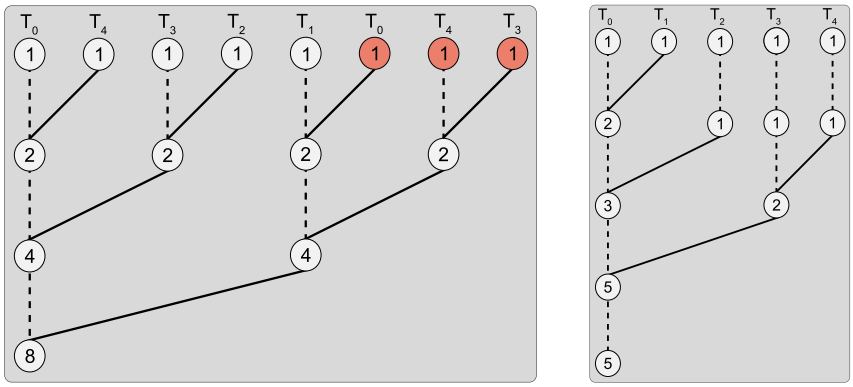
Table I: Failure ratio from one million tests over 1000 cache lines between 2 threads on 2 cores inside the same socket.

These results show a level of atomicity of load/store SIMD instructions that is not explicitly guaranteed in manufacturer documentation, for example on AMD MILAN. The level of atomicity limits the number of reduction values that we can "transport" at the same time with the synchronization flag. On Ivy-Brige we can transport up to 3 floats (3×32 bits + flag) while we can only transport 1 double (1×64 bits + flag). These observations are similar to those obtained by [16].

C. Effect of redundant communications in reduction

We have introduced in Section III the non-redundant property. Let's take an example (Figure 8) to show why redundant communications are a problem when a reduction operation and a barrier are used together.

Because of the redundancy of the communications, the value of T_0 at the end of the reduction operation combined with the Dissemination barrier is incorrect (Figure 8a). The reduction operation in this example should result in a value of 5 (each thread having an initial value of 1). However, the result is 8 because of the redundancy phenomenon. Figure 8a shows that T_0, T_3 and T_4 appear twice in the ancestors of T_0. This does not respect the definition of non-redundancy of communication between threads introduced earlier. In contrast, the ancestors of T_0 coming from the Extended Butterfly barrier (8b) are exactly equal to the set of threads participating in the barrier. We have the same observation for the other threads (T_1 to T_4). The property of non-redundancy is mandatory for a reduction operation combined with a barrier. However, for operations such as



(a) Dissemination barrier (b) Extended Butterfly barrier

Figure 8: Example of reduction operation with a sum involving 5 threads. We represent the ancestors of T_0 from Dissemination barrier (8a) and Extended Butterfly barrier (8b). The solid lines indicate to the communication between the partner threads and the dotted lines symbolize the thread waiting.

min/max, this property is not necessary. Indeed, the search for max or min does not take into account the frequency with which a thread value is encountered during the synchronization. So a reduction operation combined with the Dissemination barrier for the min/max search is possible. Nevertheless, our goal is to propose a reduction operation that can work with all operations supported by OpenMP reduction.

D. Algorithm

Algorithm 2 describes each step of the reduction operation with a sum and with AVX512 enabled. If the number of threads is a power of 2, all threads perform a reduction operation combined with the Butterflybarrier (Algorithm 4) automatically. In the case that AVX512 is enabled on an x86 processor, we will store the flag and the reduction values in a 512-bit container and then use the *_mm512_store* instruction to atomically write the entire container to the vector of the current partner thread. For AVX2 or non-AVX, the partner thread vector will be updated in 2 packets of 256 bits or 4 packets of 128 bits. Starting by storing the packets with the reduction values first and then finishing with the packet with the synchronization flag, the partner thread will see the change in its synchronization flag and will exit its busy-wait mode with the updated reduction values. To summarize, at each step of the synchronization :

1) The threads will send the container with the reduction values and the synchronization flag to its partner.
   If AVX512 is enabled, then the *_mm512_store* instruction is used.
   Else if AVX2 is enabled, then the *_mm256_store* instruction is used by sending two 256-bit packets, and the last packet will contain the synchronization flag.
   Else AVX is not enabled, then the *_mm128_store* instruction is used by sending four 128-bit packets, and the last packet will contain the synchronization flag.
2) The reduction operation will be performed by the threads.

In the next section, we will compare the performances of the reduction algorithms to the reduction of OpenMP 4.5.

V. Experimental results

In this section, we evaluate the behavior of our algorithms. We present and discuss the results of reduction operations. We first evaluate our methods using synthetic tests where we measure the average clock cycle spent in a barrier and reduction operation. In the second part, we compare the overheads of the Extended Butterfly reduction relative to OpenMP reduction using the EPCC micro-benchmark suite [5]. The Algorithm 6 describes how to compute the overheads of the barriers and reduction operations. The functions are called a predefined number of times in a loop, and the global number of cycles is measured. Then an average is calculated.

A. Test environment

The tests were performed on three different microprocessors : Intel Xeon Gold 6140 (SKL), Intel Xeon Phi 7290 (KNL) and AMD EPYC Milan 7763 (MILAN) (see table II for more details). The codes were compiled with GCC 11.1 and Intel 21.2 except on the MILAN processor. Due to compatibility problems between the Intel compiler binaries and MILAN, we compiled the codes only with GCC 11.1 on MILAN. The aim is to observe the behavior of our reduction operation on processors with a large number of physical cores. For the experiments, hyperthreading was not enabled and the tests were performed on one socket of

Algorithm 2 Extended Butterfly reduction with a sum for avx512.

```
1: Type enum {WAIT,GO}
2: Type union {
3: volatile avx512dVec simdVec
4: volatile double val[8]
5: volatile int64 flag[8]
6: } CacheLine byte_aligned(64)
7:
8: Type union {
9: avx512dVec simdVec
10: double val[8]
11: int64 flag[8]
12: } NvCacheLine
13:
14: Type struct {
15: Boolean isMaster;
16: int idGroup                                              ▷ id of the group
17: int numGroup                                             ▷ Total number of group
18: int sizeGroupe
19: CacheLine flag[2][⌊log2(P)⌋]
20: } GroupThread
21:
22: GroupThread groupOfThread[P ]
23: groupOfThread[:].flag[0][:].flag[0] = 1
24: groupOfThread[:].flag[1][:].flag[0] = 0
25: CacheLine flag[P ]
26: flag[:].flag[0] = GO
27: int nStep = log2(groupOfThread[tid].numGroup)            ▷ Number of step for the butterfly
28:
29: procedure ALL_REDUCE_EXTENDED_BUTTERFLY(int N, array val[])
30:     Input: array of double : val[N] ; int N ; ▷ The array of values to reduce and the number of values to reduce (at most
    equal to 7 for avx512 with double)
31:     Output: array of double : val[N] ;
32:
33:     thread private: int sense = 0
34:     thread private: int parity = 1
35:     thread private: int tid = omp_get_thread_num()
36:
37:     call REDUCE_LINEAR_CENTRALIZED(N, val, tid)
38:     if groupOfThread[tid].isMaster then
39:         call REDUCE_BUTTERFLY(N, val, tid)
40:     end if
41:     call BCAST_LINEAR_CENTRALIZED(N, val, tid)
42:     if parity == 1 then                                  ▷ Prepare the barrier for the next use
43:         sense = !sense
44:     end if
45:     parity = 1-parity
46: end procedure
```

the node. The KNL processor was configured on All2All cluster mode and flat memory mode. Concerning the binaries from the GCC compiler, the environment variable `OMP_PROC_BIND` is set to true and the `OMP_WAIT_POLICY` is set to `ACTIVE`. For the binaries from Intel compiler, the variable `KMP_AFFINITY` was set to `compact` with `granularity=fine` and the variables `KMP_PLAIN_*_PATTERN` were set to `hyper`. The Intel `hyper` configuration provides the best performance for Intel OpenMP reduction and barrier.

Algorithm 3 Incoming phase in Extended Butterfly barrier

```
1: procedure REDUCE_LINEAR_CENTRALIZED(N, val, tid)
2:     if groupOfThread[tid].isMaster then
3:         for i in members of group do
4:             Repeat until (flag[i].flag[0]==WAIT)
5:             for j in range(0,N-1) do                          ▷ Unroll a SIMD add
6:                 val[j]+=flag[i].val[j+1]
7:             end for
8:         end for
9:     else
10:        NvCacheLine localFlag;
11:        localFlag[i].flag[0]=WAIT
12:        for j in range(1,N) do                                ▷ Unroll a SIMD write
13:            localFlag.val[j] = val[j-1]
14:        end for
15:        flag[tid].simdVec = localFlag.simdVec
16:    end if
17: end procedure
```

Algorithm 4 Butterfly phase in Extended Butterfly barrier

```
1: procedure REDUCE_BUTTERFLY(N, val, tid)
2:     NvCacheLine localFlag
3:     for step in range(0,nStep-1) do
4:         int partner = getpartner(groupOfThread[tid].idGroup,step)   ▷ Compute the partner (see the section II-D)
5:         localFlag.flag[0] = sense
6:         for j in range(1,N) do                                ▷ Unroll a SIMD write
7:             localFlag.val[j] = val[j-1]
8:         end for
9:         groupOfThread[partner].flag[parity][step].simdVec = localFlag.simdVec   ▷ SIMD write instruction
10:        Repeat until (groupOfThread[tid].flag[parity][step].flag[0]==sense)
11:        for j in range(0,N-1) do                              ▷ Unroll a SIMD add
12:            val[j]+=groupOfThread[tid].flag[parity][step].val[j+1];
13:        end for
14:    end for
15: end procedure
```

Supercomputer	IFPEN (ENER)	IFPEN	TGCC* (TOPAZE)
Processor type	Intel Xeon Gold 6140 (SKL)	Intel Xeon Phi 7290 (KNL)	AMD EPYC Milan 7763 (MILAN)
# processor / node	2	1	2
# core / processor	18	72	64
Freq (GHz)	2.6	1.5	2.45
Compiler	Intel 2021.2	Intel 2021.2	GCC 11.1
OpenMP version	4.5	4.5	4.5

Table II: Information about the test environment.

B. Results of barriers and reductions

In this section, we present results from the synthetic test. Each algorithm (barrier or reduction) is called 10^4 times, and we report the average number of cycles per iteration in the loop. Figure 9 compares the different barriers introduced previously. The lower the values are, the better the performance is. We can see that three barriers stand out : the Dissemination barrier (in brown), the Linear Centralized barrier (in green), and the Extended Butterfly barrier (in blue). The Dissemination barrier is not usable for the reduction operation, but we use it as a reference (as the best barrier to our knowledge). The Extended Butterfly barrier is the best of the three, better than the Dissemination barrier. In section IV-C, we have seen that the Dissemination barrier cannot be used to perform a reduction operation with the method proposed here. Indeed, the redundant communications of some threads at some steps will distort the final result, and it is quite difficult to identify these communications. However, we

Algorithm 5 Outgoing phase in Extended Butterfly barrier

```
1:  procedure BCAST_LINEAR_CENTRALIZED(N, val, tid)
2:      if groupOfThread[tid].isMaster then
3:          NvCacheLine localFlag
4:          localFlag.flag[0] = GO
5:          for  j in range(1,N) do                          ▷ Unroll a SIMD write
6:              localFlag.val[j] = val[j-1]
7:          end for
8:          for i in members of group do
9:              flag[i].simdVec = localFlag.simdVec          ▷ SIMD write instruction
10:         end for
11:     else
12:         Repeat until (flag[tid].flag[0]==GO)
13:         for j in range(0,N-1) do                         ▷ Unroll a SIMD write
14:             val[j] = flag[tid].val[j+1]
15:         end for
16:     end if
17: end procedure
```

Algorithm 6 Synthetic test of reduction operation

```
1:  NTEST //Set the number of tests
2:  start = getclock();
3:  #pragma omp parallel
4:  {
5:  for j = 0; j < NTEST; ++j do
6:      #pragma omp for reduction(+:aaaa) schedule(static,1)
7:      for i=0; i<nthreads; ++i do
8:          val_red+=1;
9:      end for
10: end for
11: }
12: times =(getclock()-start); //get the clock cycle
13: times /=(double) NTEST;
```

have implemented a reduction operation combined with the Linear Centralized barrier.

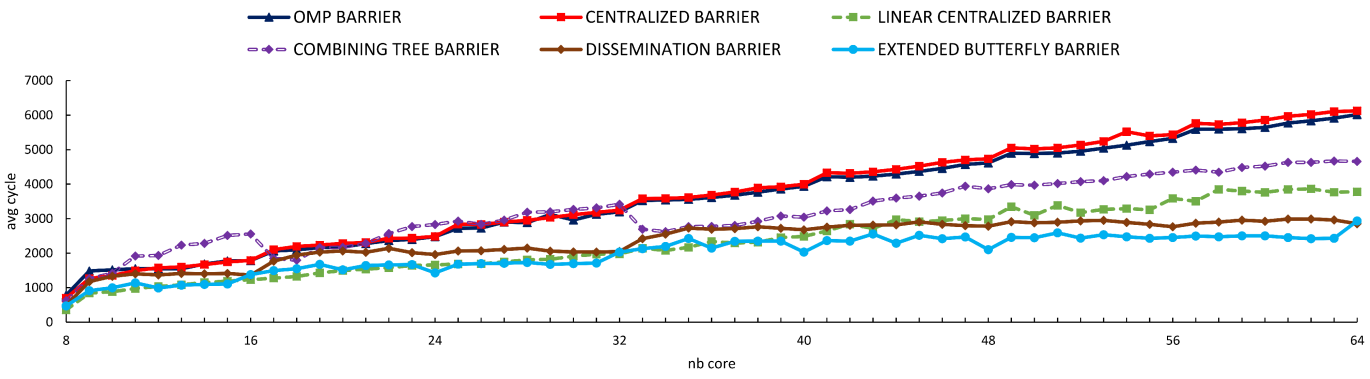


Figure 9: Comparative result of the barriers with the synthetic test using the GCC 11.1 compiler on MILAN processor.

Let's consider now the results of the Linear Centralized, Extended Butterfly and OpenMP barriers, which are represented by the hatched parts of the histograms in Figures (10,11,12). The barrier and reduction overheads have been measured separately. For a small number of threads (less than 30), the Linear Centralizedand Extended Butterfly barriers do not show a big difference on MILAN processor, even if the Linear Centralized barrier seems to perform better (especially for a number of threads less

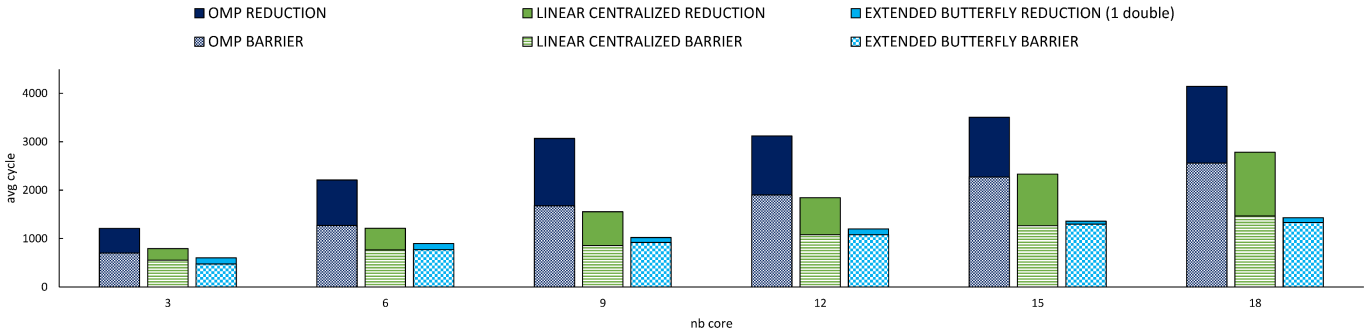


Figure 10: Comparative result of the reduction operations with the synthetic test using the Intel 21.2 compiler on SKL processor.

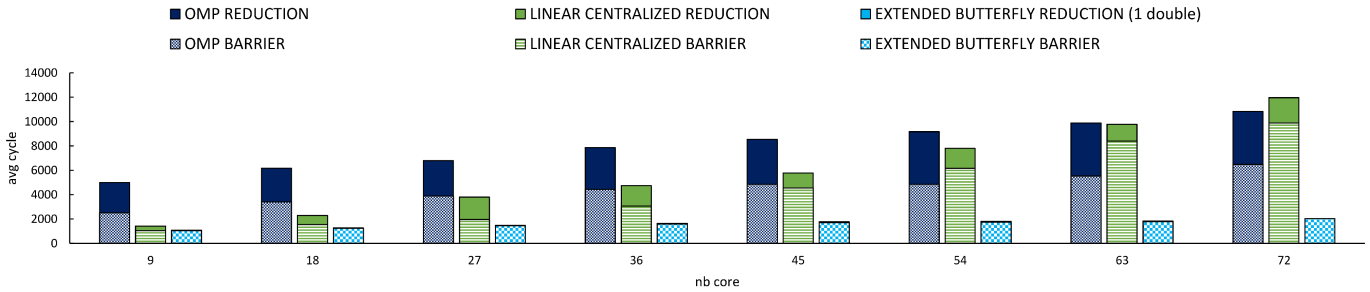


Figure 11: Comparative result of the reduction operations with the synthetic test using the Intel 21.2 compiler on KNL processor.

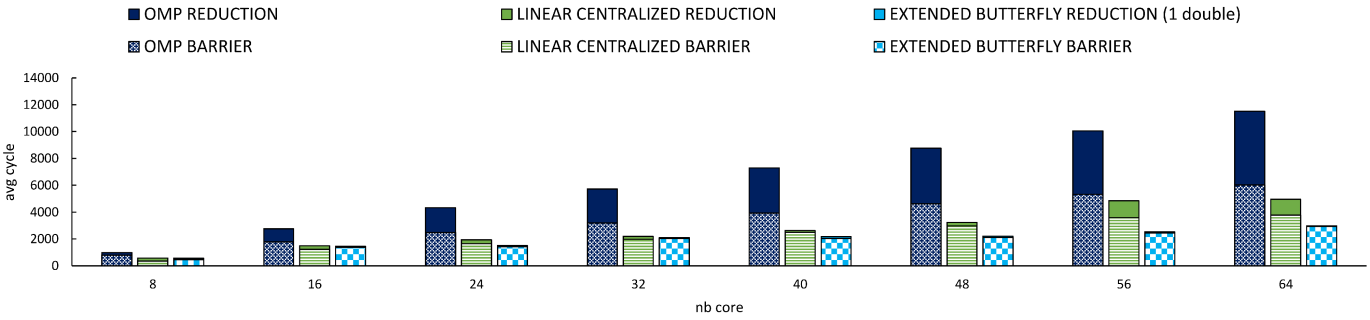


Figure 12: Comparative result of the reduction operations with the synthetic test using the GCC 11.1 compiler on MILAN processor.

than 16). However, these two barriers are both much faster than the barriers from OpenMP 4.5 with the GCC 11.1 and Intel 21.2 compiler. For a large number of threads, Extended Butterfly barrier has the best results (see Figure 11,12) far ahead of GCC 11.1/Intel 21.2 OpenMP barriers and Linear Centralized barrier. Now we focus on the performance of the reductions, which correspond to the histograms (hatched and darker parts) Figures (10,11,12). Similarly to the barrier case, our Extended Butterfly reduction outperforms the Intel 21.2/GCC 11.1 OpenMP `for reduction` and the reduction operation combined with the Linear Centralized barrier on the three processors. By combining the reduction operation with a barrier, the reduction inherits the same properties as the barrier on which it is based. Nevertheless, on the three processors we often see a very significant difference between the number of average cycle (directly related to the time spent) of the Linear Centralized barrier and reduction. The reason is that all operations are centralized and managed only by the master thread. Unlike the Extended Butterfly barrier where the operations are decentralized across each group. It makes all the difference. As can be seen in Figure 11, the reduction operation combined with the Linear Centralized barrier has a linear growth, which makes it not efficient for a large number of threads.

In the case where the reduction operation is performed on several values, i.e. an array with at the most 7×64 bits for the reduction values and 1×64 bits for the flag (see Figure 7). The average cycle consumed for 1, 3, and 7 double is shown in Figure 13. A minor additional cost appears between AVX/AVX2 and AVX512 (see Figure 13). However, the Extended Butterfly reduction cost stays below the 2000 average cycles on KNL processor, which is 5.5 times less than the cost of the

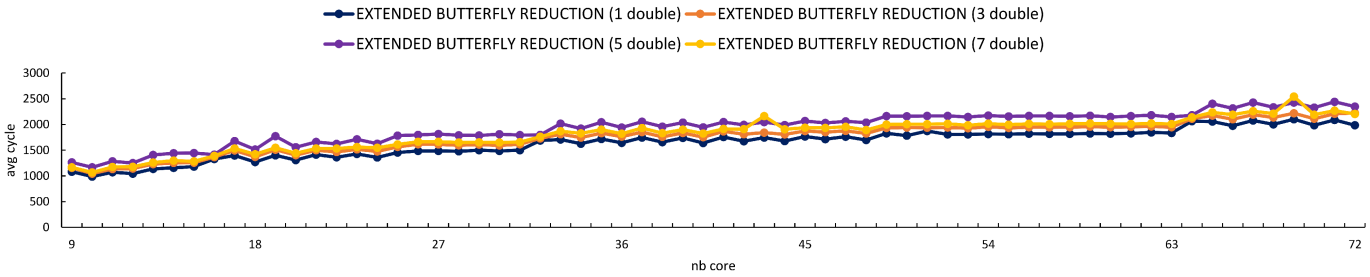


Figure 13: The synthetic tests of the reduction operation for 1, 3 and 7 values on KNL processor with the Intel 21.2 compiler.

Intel 21.2 OpenMP reduction (see Figure 11). In the next step, we decided to test our functions on a known benchmark like the EPCC microbenchmark.

C. EPCC Benchmark

The idea of the EPCC benchmark is to execute in loops, a large number of times, the OpenMP 4.5 directives to make the creation time of these directives significantly measurable. Algorithm 7 describes how the benchmark stresses reduction in the parallel section. Another function computes a reference time (`delaylength`) which takes into account the cost of the `delay()` call. The overhead measurement is obtained by looking at the difference between the actual parallel execution time and the reference execution time [5]. The computation of the barrier and reduction overheads is quite similar to our synthetic tests. However, the benchmark imposes a delay between each iteration in the inner loop to simulate a calculation before the reduction. This is more realistic than our synthetic test.

Algorithm 7 Reduction overhead with the EPCC benchmark

```
1:  for k=0; k<=OUTERLOOPS ; ++k do
2:      start = getclock();
3:      for j = 0; j < INNERLOOPS; ++j do
4:          #pragma omp parallel reduction(+:aaaa)
5:          {
6:          delay(delaylength);
7:          aaaa+=1;
8:          }
9:      end for
10:     times[k]=(getclock()-start)*1.0e6;
11:     times[k]/=(double) INNERLOOPS;
12: end for
```

Figure 14 shows the evolution according to the number of threads of the ratio between Extended Butterfly reduction and reduction operation of GCC 11.1 and Intel 21.2 OpenMP for respectively MILAN and SKL, KNL. So, the higher the speedups are, the lower the overheads are. The Extended Butterfly reduction has around 4 times less overhead than the GCC 11.1 OpenMP reduction. Moreover, the Extended Butterfly reduction performs around 2 times and 3.5 times less overhead than the Intel 21.2 OpenMP reduction respectively on SKL and KNL processors.

VI. Conclusion

Our goal was to have a reduction operation combined with an efficient barrier for a large number of threads. To achieve this objective, we needed a barrier that would be efficient for a large number of threads. We first studied and implemented various barriers from the literature. We then proposed a novel barrier (Extended Butterfly barrier) based on the Butterflybarrier and efficient for any number of threads. Our synthetic tests have shown that the Extended Butterfly barrier is the best for a large number of threads. It significantly outperforms the Intel 21.2 and GCC 11.1 OpenMP barriers. Then we proposed a method for the reduction of values combined with a barrier by transporting the values to be reduced with the synchronization flag. Thus, the reduction values are directly available in the thread cache line and are updated at each step of the barrier. This method has been used in the reduction operation combined with the Linear Centralized barrier and the Extended Butterfly reduction. We have optimized the reduction operation by using SIMD instructions for reading/writing cache lines containing the reduction values and the synchronization flag for each thread atomically. Figure 14 shows that Extended Butterfly reduction has in average 2 times and 3.5 times less overheads than Intel 21.2 OpenMP reduction on SKL and KNL. For example, with 72 cores on

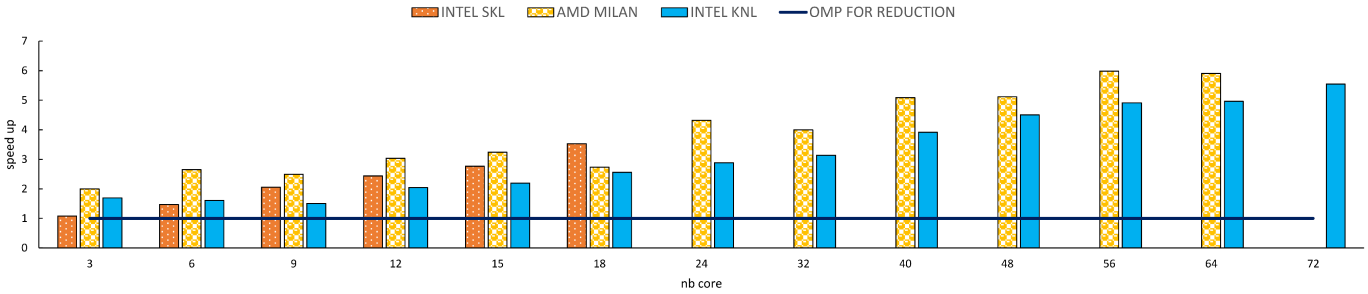


Figure 14: Representation of the overhead speed-ups of Extended Butterfly reduction relative to OpenMP 4.5 reduction on SKL, KNL and MILAN. Overheads computed with the EPCC benchmark.

KNL the Extended Butterfly reduction are 5.4 times faster than Intel 21.2 OpenMP reduction and with 64 cores on MILAN the Extended Butterfly reduction are 5.9 times faster than GCC 11.1 OpenMP reduction. As future work, we will continue testing on applicative benchmarks based on Krylov solvers such as CG or BiCGStab. It would be interesting to configure the Extended Butterfly reduction to manage hyperthreading. Furthermore, the processors use different core interconnect topologies (KNL and SKL are similar but not MILAN), so it may be interesting to investigate the impact on barriers and reduction operations. We foresee that with the increasing number of cores, the topology impact can be significant.

Statement: The authors have no conflicts of interest to declare. The data that support the findings of this study are openly available in github repository "Data Reductions and Barriers" at https://github.com/mohameak/Data-Reductions-and-Barriers.git

REFERENCES

[1] A. Aravind. Barrier Synchronization: Simplified, Generalized, and Solved Without Mutual Exclusion. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 773–782, Los Alamitos, CA, USA, may 2018. University of Northern British Columbia, Department of Computer Science, Prince George,BC,Canada, IEEE Computer Society. doi: 10.1109/IPDPSW.2018.00124. URL https://doi.ieeecomputersociety.org/10.1109/IPDPSW.2018.00124.

[2] AMD64 Architecture. AMD64 Architecture Programmer's Manual, Volumes 1-5, 40332, 24592, 24593, 24594, 26568, 26569. Technical Report Volumes 1-5, 2020. URL https://www.amd.com/system/files/TechDocs/40332.pdf.

[3] Intel Architecture. Intel® Architecture Instruction Set Extensions Programming Reference. Technical Report 319433-023, August 2015. URL https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[4] E. D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986. doi: 10.1007/BF01407877. URL https://doi.org/10.1007/BF01407877.

[5] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[6] D. Caballero. *SIMD@OpenMP : a programming model approach to leverage SIMD features*. PhD thesis, Universitat Politècnica de Catalunya, 2015.

[7] D. Caballero, A. Duran, and X. Martorell. An OpenMP* Barrier Using SIMD Instructions for Intel® Xeon Phi™ Coprocessor. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, pages 99–113, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40698-0.

[8] D. Grunwald and S. Vajracharya. Efficient barriers for distributed shared memory computers. pages 604 – 608. University of Colorado, Boulder, 05 1994. ISBN 0-8186-5602-6. doi: 10.1109/IPPS.1994.288242.

[9] Y. Han and R. Finkel. An optimal scheme for disseminating information. *Proceedings of 1988 International Conference on Parallel Processing, Chicago*, pages 198–203, 1988. URL http://h.web.umkc.edu/hanyij/html/research/icpp88.pdf.

[10] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988. doi: 10.1007/BF01379320. URL https://doi.org/10.1007/BF01379320.

[11] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. Chemnitzer Informatik Berichte. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.5369, 2004.

[12] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. Technical report, 2021. URL https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[13] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL https://doi.org/10.1145/103727.103729.

[14] R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Evolving OpenMP in an age of extreme parallelism*, pages 42–52, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02303-3.

[15] G. F. Pfister and V. Norton. "hot-spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10):943–948, oct 1985. ISSN 1557-9956. doi: 10.1109/TC.1985.6312198.

[16] Erik Rigtorp. Aligned AVX loads and stores are atomic, June 2020. URL https://rigtorp.se/isatomic/.

[17] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján. Effective barrier synchronization on intel xeon phi coprocessor. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 588–600, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48096-0.

[18] E. Speziale, A. di Biagio, and G. Agosta. An optimized reduction design to minimize atomic operations in shared memory multiprocessors. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1300–1309, 2011. doi: 10.1109/IPDPS.2011.271.

[19] O. Villa, G. Palermo, and C. Silvano. Efficiency and Scalability of Barrier Synchronization on NoC Based Many-Core Architectures. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'08, page 81–90, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605584690. doi: 10.1145/1450095.1450110. URL https://doi.org/10.1145/1450095.1450110.

[20] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. *Distributing Hot-Spot Addressing in Large-Scale Multiprocessors*, page 302–309. IEEE Computer Society Press, Washington, DC, USA, 1994. ISBN 0818661976. doi: 10.5555/201173.201223. URL https://dl.acm.org/doi/abs/10.5555/201173.201223.

[21] ZhengMing Yi, Fei Chen, and YiPing Yao. A barrier optimization framework for NUMA multi-core system. *Concurrency and Computation: Practice and Experience*, 32(5):e5527, 2020. doi: https://doi.org/10.1002/cpe.5527. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5527. e5527 cpe.5527.