Efficient barrier synchronization for 2D meshed NoC-based many-core processors

Lovic Gauthier[†], Farhad Mehdipour[⋆], Koji Inoue[‡], Shinya Ueno[‡] and Hiroshi Sasaki[‡]

[†]Sytem LSI Research Center, Kyushu Univerity, Fukuoka, Japan 814-0001
[‡]Departement of Informatics, Kyushu University, Fukuoka, Japan 819-0395
[⋆]E-JUST Center, Kyushu University, Fukuoka, Japan 814-0001

{lovic,ueno,sasaki}@soc.ait.kyushu-u.ac.jp, {inoue}@ait.kyushu-u.ac.jp, farhad@ejust.kyushu-u.ac.jp

Abstract— Network-on-Chip (NoC) based many-cores are becoming popular due to their high scalability compared to traditional bus-based architectures. However they still lack software tailored to their specificities. In this paper we propose several techniques for tailoring and combining barrier synchronizations in order to take advantage of the 2D-meshed NoCs. Experimental results show that our combined barriers achieve often twice shorter delays than state of the art barriers.

I. Introduction

NoC-based many-core architectures are emerging as solutions for matching the increasing demand in processing power. However, legacy software does not scale well on such architectures. This is especially true for the synchronization primitives which are usually implemented assuming an identical communication delay between all the cores. In this paper, we explore the design of barriers, some of the most often used synchronization mechanisms for multi-threaded applications.

A barrier is a mechanism used for synchronizing a set of threads. It is a point in each thread's program (usually a call to a barrier function) which can be passed through only when the other threads of the set have reached this barrier. Performing a barrier requires to exchange several messages among the threads to synchronize. With an NoC-based architecture, this can take a long time due to the important communication delay between distant cores. Unfortunately, the traditional barriers are designed considering an identical delay for all the inter-thread messages which is not the case with an NoC.

In this context, the contribution of this paper is double. First, it is explained how three existing barrier mechanisms, namely *tree* [5], *butterfly* [3] and *dissemination* [5], can be tailored for shorter delays on a 2D meshed NoC-based architecture. Second, new barrier mechanisms are presented which combine the previously tailored ones for achieving even shorter delays.

The rest of the paper is organized as follows: the next section gives the assumptions we made about the architecture, the threads and the implementation of the barriers. Section III. presents some related works. Section IV. presents state of the art barrier mechanisms and studies their performances when used with a NoC-based many-core architecture. Sections V. and VI. present the proposed techniques for tailoring and combining barrier mechanisms. Finally, section VII. gives some experimental results and section VIII. concludes the paper.

II. Assumptions

First, the target architecture considered in this paper is a 2D meshed NoC-based many-core processor where each core is a MeP [15] having its own scratch-pad memory (SPM). Inter-core communication is assumed to be memory-mapped: each core can access the local memory of another core, but then, the data needs to transit through the NoC.

Second, the barriers considered in this paper are OpenMP-based [11]. Hence it is required that the threads to synchronize are identified when creating a barrier. It is also assumed that these creations are performed during non-critical phases, e.g., at compile or initialization time. Therefore important computations can be afforded for creating a barrier. Such an assumption is common in recent applications (cf. OpenCL [7]). Then, based on the fact that data transfers are usually more critical than synchronizations, is it assumed that the threads mapping cannot be changed when creating a barrier.

Third, the mechanism of a barrier is described in two parts. The first part, called the *implementation* in this paper, describes how the messages are sent and received by the threads. The second part, called the *protocol*, describes which thread sends/receives messages to/from which other thread. The focus of this paper is on the protocol only.

If the implementation and the protocol are functionally independent of each other, they are related regarding the performance of the barriers since a slow implementation gives more importance to the message processing delay with respect to the NoC delay. For the experiments, we considered two typical implementations: a fully software one based on busy-waits (cf. [10]), and a fast hardware-based one (cf. [16]). Since they are not the focus of the paper, we do not detail them here.

III. Related work

Barrier synchronizations have been a largely studied topic. Initial works explored the possible protocols [3, 5], while others explored various software implementations [10, 16]. Since barrier synchronizations can be important bottlenecks, several papers also proposed hardware implementations for them [1, 2, 9, 12, 14, 16], and others proposed optimizations of the application or the system software for reducing the impact of the barriers overhead or for supporting incomplete barrier mechanisms [4, 6, 8, 13].

Nevertheless, the protocol is crucial for the performance, especially with NoC-based architectures where the delay of a message depends on the relative position of the sender and the receiver. However, to our knowledge, only [16] studied the effect of NoC architectures on barriers. They considered the master-slave, the tree, the butterfly and the all-to-all barrier protocols but only for the cases where the number of threads is a power of two and their study overlooked the effects of the threads mapping on the synchronization delays.

IV. Motivation

A. *Basic barrier synchronization protocols*

In this paper a protocol is described as a succession of steps. For each step, some threads send and receive messages. By convention, a send operation is non-blocking and is executed at the beginning of its step whereas a receive operation is blocking and executed at the end of its step. In order to distinguish the threads from each other, they are numbered from 0 to $N-1$ (N being the number of the threads). Also, the messages contain the identity of their senders so that they can arrive out of order while keeping the validity of the protocols. In the figures of this paper, the cores are represented by small squares and the threads by numbers on the squares. Additionally, a message sent from one thread, t, to another, t', is represented by an arrow, and we say that they are *connected*. Then, if at the same step thread t' sends also a message to thread t the arrow is doubled and we say that they the *exchange* a pair of messages.

Among the multiple barrier protocols, we can cite the following basics [3, 5, 16] which are illustrated in Fig. 2:

All-to-all: it is a one-step protocol. Each thread sends one message to each of the other threads of the barrier and waits for all their messages. This is the most straight forward protocol but it requires a lot of messages.

Master-slave: it is a two-step protocol. A thread is selected to be the master, the others being the slaves. During the first step, each slave sends one message to the master. When the master has received all the slaves' messages, the second step starts where the master sends one message to each of the slaves.

Tree: it is a multi-step protocol with two phases. A tree is built upon the threads for determining the steps of the protocol. During the first phase, each thread waits for one message from each of its children then sends one message to its father. During the second phase, each thread waits for one message from its father, then send one message to each of its children.

Butterfly: it is a multi-step protocol inspired from the butterfly algorithm of the Fourier transform. When N is a power of two, there are $log_2(N)$ steps and concretely, during step i, thread t exchanges a message with thread $t\ xor\ 2^i$. However, when N is not a power of two, the butterfly cannot be implemented directly. A solution [3] is to build a butterfly for $np_2(N)$ threads[1]. The additional threads are virtual and mirrored backwardly over the real threads. This is illustrated in Fig. 1 where threads 5, 6 and 7 are virtual. During the $log_2\big(np_2(N)\big)$ steps, the exchanges of messages are performed as previously by the real threads, but they also exchange the messages of the virtual threads overlapping them. Incidentally, virtual threads imply sometimes operations which can be omitted, i.e., messages exchange of a thread with itself or multiple exchanges between two threads. Hence, it can be observed that the first step is exempted from extra messages.

Dissemination: it is a variant of the butterfly which does not need mirroring when N is not a power of two. As with the

[1]In this paper, np_2 stands for next higher power of two

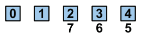


Fig. 1. Mirroring threads for the butterfly protocol

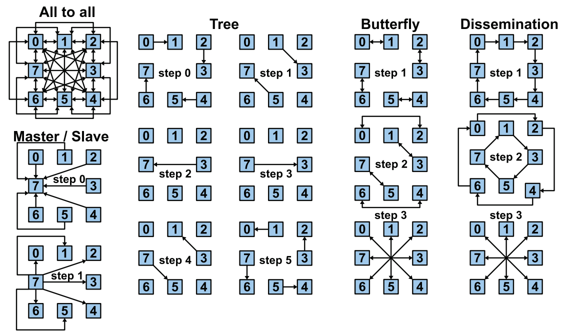


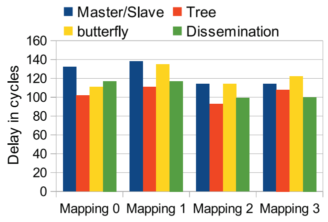
Fig. 2. Basic barrier synchronization protocols

butterfly, there are $log_2\big(np_2(N)\big)$ steps, but the threads are not paired. Instead, they are chained to their successor and predecessor according to the butterfly diagram. Concretely, for step i thread t sends a message to thread $(t+2^i)\ mod\ N$ then receives a message from thread $(t-2^i)\ mod\ N$.

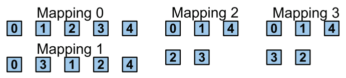
B. *Performance of the basic barrier protocols*

Fig. 3a shows the different delays in cycles for several protocols with four threads mapped according to the patterns given in Fig. 3b. It has been considered that the software implementation was used for the barriers. Not only the protocols perform differently from what was expected from legacy results [3, 5, 16] (dissemination and butterfly should have been the most efficient), but the order of the threads with a same global shape also influence strongly the results as it can be seen by comparing mappings 0 and 1 and mappings 2 and 3. This quick analysis strongly hints that shorter delays could be achieved if the specificities of the NoC and the mapping of the threads could be properly taken into account.

Furthermore, the delay depends on the characteristics of the target architecture (e.g., NoC delays, cores speed), making it hard to define a good protocol in general. Hence, we introduce two target independent metrics. The first one, called *message delay*, is the largest number of messages processings (one processing includes both a send and a receive) required for a thread to reach another thread. This metric depends only on the protocol and N, the number of threads to synchronize. The protocol with the shortest message delay is dissemination which completes after $log_2\big(np_2(N)\big)$ message processings. The second metric, called *hop delay*, is the largest number of hops required for a thread to reach another thread. This metric depends only on the protocol, N and the mapping of the threads. With it, a bound can be computed: for a barrier synchronization to be completed, each thread must reach, directly or not, each of the other threads, therefore the minimal hop delay is the Manhattan distance between the farthest threads. For



(a) Delays



(b) Mappings

Fig. 3. Delays of basic barriers on different mappings

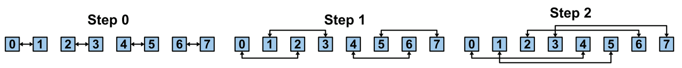


Fig. 4. Efficient butterfly with a power of two line

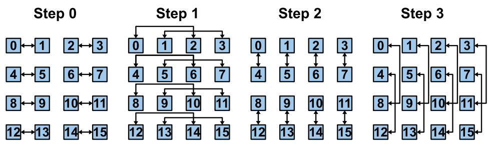


Fig. 5. Efficient butterfly with a power of two rectangle

instance, in Fig. 2 for dissemination, thread 0 requires first to send a message to thread 2 for reaching thread 6 which implies that the corresponding message delay is 2 and the hop delay is 6. These are also the message and hop delays of the full protocol since both are the maxima among all the threads. By contrast, for the master-slave case, thread 0 reaches thread 6 through the master, thread 7, which needs to process the messages of all the other threads before reaching 6 so that the message delay is 12 and the hop delay is 2.

V. TAILORING OF EXISTING PROTOCOLS

In this section, we present techniques for tailoring the most promising protocols, i.e., butterfly, dissemination and tree for various mappings of the threads in a 2D meshed NoC. By tailoring we mean modifying the numbering of the threads, modifying the mirroring (for butterfly) and modifying the arity of the nodes (for tree). Note that the numbering is just for the barrier protocols, it is not related to any other management of the threads. Simple shapes like lines and rectangles are frequent in practice due to the ease of describing and manipulating them by the programmers (cf. OpenCL [7]). Therefore, they have the priority when tailoring the protocols. It also worth noting that the proposed tailoring are also efficient when the shapes are sparse, for instance when one core out of two is used.

A. Tailoring butterfly

If N is a power of two:
This is a favorable case for the butterfly since the message delay is only $log_2(N)$. When the threads are mapped as a line or a rectangle, it is enough to number in topological XY order for achieving the optimal hop delay. With this order, the threads are sorted by comparing first their y coordinates, and then if equal, their x coordinates. Fig. 4 and 5 illustrate the butterfly with such a numbering. The other shapes are treated in the last paragraph of this section.

For a line the optimality can be demonstrated by summing the hop delay of each step: for the first step, it is 1 hop, for the second 2, for the third 4 and so on. Since there are $log_2(N)$ steps, the total hop delay is $\sum_{i=0}^{log_2(N)-1} 2^i = N - 1$ which is the length in hops of the line and therefore the optimal. For a rectangle shape, N being a power of two, the width W and the height H too are powers of two. With a topological numbering the lines are first synchronized independently of each other before the columns are, with respective hop delays of $W - 1$ and $H - 1$. Hence, the total hop delay is $W + H - 2$ which is also the diagonal length in hops of the rectangle (in the Manhattan sense) and the optimal.

If N is not a power of two:
When the number of the threads is not a power of two, a mirroring technique is used and the synchronization requires $log\big(np_2(N)\big) = log_2(N) + 1$ steps. At each step, the threads covered by virtual threads need to send and receive two messages instead of one (unless the extra message can be omitted as seen in section IV.A.). Consequently, the message delay is longer than $log_2(N) + 1$ and often reaches $2 * log_2(N) + 1$.

When the threads are mapped as a line shape as in Fig. 6, it can be verified that the XY topological numbering still achieves an optimal hop delay for the butterfly (but other protocols can achieve better). This is also the case for a rectangle shape provided the width W (or the height H with the YX topological numbering) is a power of 2. As it can be seen in Fig. 7, this numbering leads the protocol to synchronize the lines independently of each other with an optimal hop delay of $W - 1$. Then, the columns can also be synchronized independently of each other with a delay of $np2(H) - 1$ if the mirroring technique is applied column by column instead of globally. This way, the total hop delay is $W + np2(H) - 2$.

When, for a rectangle, neither W nor H are powers of two the topological numbering does not guarantee an optimal hop delay. For instance, if the width is odd, the first step will make the last thread of a line exchange a message with the first thread of the next line which requires a large number of hops whereas if the numbering is performed in zigzag as illustrated in Fig. 8, only one hop is necessary. From this observation, we propose a general heuristic which numbers each line in topological order as previously, but whose direction, left or right depends on the previous line. Specifically, if the last thread of a line exchanges more messages with its successors than its predecessors, the next line is numbered in the opposite direction, otherwise it is numbered in the same direction.

Other shapes:
For mapping shapes which are neither lines nor rectangles the previous heuristic, i.e., numbering the lines from the left to the right or the right to the left depending on the messages exchanged by their last threads, is still applicable.

B. Tailoring the dissemination protocol

The dissemination protocol is the most efficient in term of message delay since it requires only $log_2(np_2(N))$ messages processing for any value of N. However, it performs poorly in term of hop delay. This is due to the ring topology of this protocol which often implies messages between the most distant threads. This is why the heuristics proposed here numbers

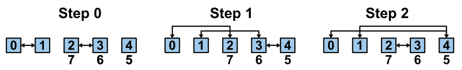


Fig. 6. Efficient butterfly on a non power of two line

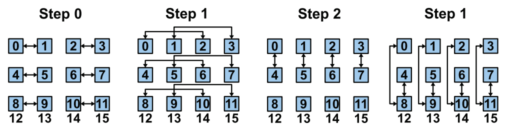


Fig. 7. Efficient butterfly on rectangle whose width is a power of two

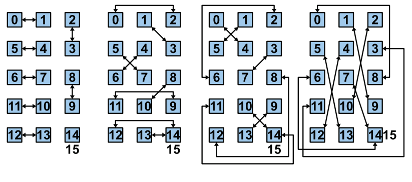
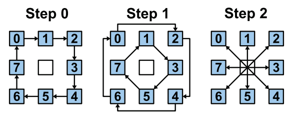
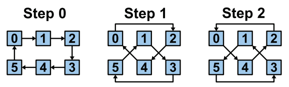


Fig. 8. Efficient butterfly on a rectangle whose sides are not a power of two

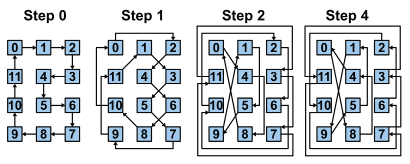


(a) Plain ring

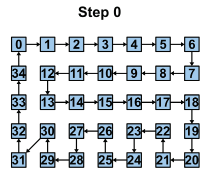


(b) 2 lines rectangle

Fig. 9. Efficient dissemination on rings

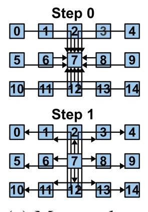


(a) One even side

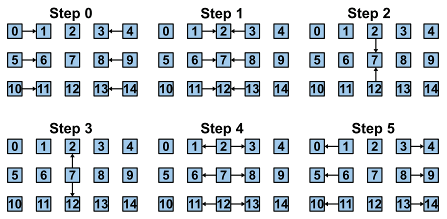


(b) No even side

Fig. 10. Efficient dissemination on rectangles

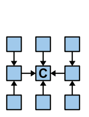


(a) Master-slave

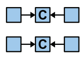


(b) Rectangle tree

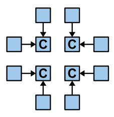
Fig. 12. Master slave tuning and conversion to tree



(a) odd-odd



(b) odd-even



(c) even-even

Fig. 13. Center depending on the parity of the height and the width

the threads so that each of them is as close as possible to its predecessors and successors in the ring of the dissemination.

For an actual ring shape or a two-line rectangle shape as shown in Fig. 9a and 9b, the ring numbering is immediate and it can be verified that the total hop delay is $N - 1$. This is better than for a topological numbering which would require this delay for the two first steps already.

For a rectangle shape where the width or the height is even, a ring can again be built as it can be seen in Fig. 10a. This numbering is close to the zigzag of butterfly, but the left column (apart from thread 0) is numbered last and from the bottom to the top in order to close the ring. For a rectangle shape with only odd sides, the zigzag falls on the wrong side for the last line, therefore we propose to stop the vertical zigzag two lines before and to perform a horizontal zigzag from here as in Fig. 10b (only the first step is given).

For a line shape, it is impossible to build a ring of direct neighboring threads. However, it is possible to reduce the connection between the last and the first thread by numbering from the left to the right one thread out of two and then numbering from the right to the left for the remaining threads as shown in Fig. 11. It can be verified that the total hop delay is $2 * N - 2$, which is better in practice than with a topological numbering.

When the threads are not mapped as a rectangle nor a line, the zigzag approach for dissemination can still be used but the left column is not straight any longer and the variable size of the lines might incur more delays.

C. Tailoring (deeply) the tree protocol

For tuning the tree protocol, let us first consider the master-slave one. With this protocol, it is enough to select the thread which is in the center of the mapping for being the master in order to achieve an optimal number of hops. Indeed, assuming this center corresponds to a single thread, it is located at half the Manhattan distance from the farthest threads. Since the protocol includes two steps, the total hop delay is equal to the Manhattan distance between the farthest threads. However, the NoC usage is very high since all the slaves send and receive messages with the same master so that the links are used several times as it can be seen in Fig. 12a.

This is why we propose to convert this master-slave protocol to a tree protocol as follows: each connection from a thread to the center (i.e., the master) is segmented over each NoC link

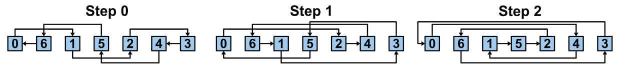


Fig. 11. Efficient dissemination on an odd line

encountered along the path. Then, the segments sharing a same link are merged so that the resulting protocol is executed as seen in Fig. 12. This new protocol is called *rectangle tree* and is indeed a tree protocol where the root is the center and the leaves are at the periphery of the shape. With it, each link is used at most once, while the hop delay is preserved.

There are still issues when the center of the mapping does not fall on a single thread. This happens first when the width or the height of the shape are even. We solve this by considering as root the threads adjacent to the center while synchronizing them with butterfly which ensures an optimal hop delay for the three possible cases. The first case is when the width and the height are both odd so that the root is a single thread as shown in Fig. 13a. In the second case, only one of the width or the height is odd. Hence the root is made of two threads as illustrated in Fig. 13b, and synchronizing them with butterfly requires 1 hop delay. In the last case, both the width and the height are even so that the root is made of four threads as in Fig. 13c. When butterfly is applied on them the hop delay is only 2. Finally, if the center falls on cores unused by the threads to synchronized, the closest thread is chosen instead, which allows a short hop delay but not the optimal. Apart from this last issue, rectangle tree achieves the optimal hop delay independently of the mapping of the threads and notably when they are distant from each other.

VI. Combining the protocols

Tailoring the protocols can improve their performances, but they are not efficient on all the shapes. In this section we propose two new techniques which combines those protocols in order to achieve good results with all sort of mappings. The first one, called *divide and conquer*, partitions the threads into several groups for efficient intra and inter-group synchronizations. While often efficient, this approach fails on certain cases as seen further, this is why a second technique is proposed which embeds divide and conquer within rectangle tree.

A. Divide and conquer

Base algorithm:
Let us assume that the number of threads is $N = F_0 * F_1$. They are first partitioned into F_1 groups of F_0 threads. The F_1 groups can then be synchronized independently of each other, before synchronizing them with each other. Since the groups have the same number of threads, this second phase is performed by synchronizing each thread of each group with one

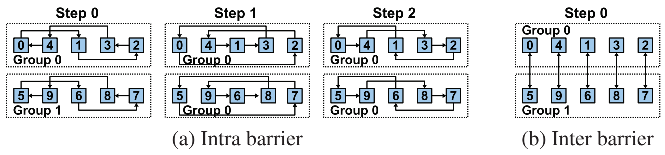


Fig. 14. Divide and conquer-based barrier synchronization protocol

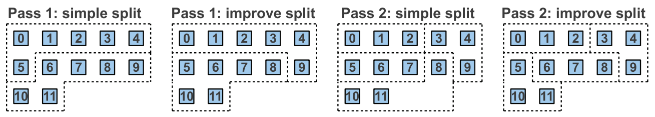


Fig. 15. Selecting groups

thread of each of the other groups. Both phases are illustrated in Fig. 14. For the inter-group phase, the distance between connected threads is reduced by synchronizing the first threads of each group together, the second together, and so on as it can be seen in Fig. 14b. The intra and inter-synchronizations can then be performed by applying recursively the same divide and conquer approach provided F_0 or F_1 can be decomposed into integer factors (i.e., they are not prime).

The recursion is stopped when prime numbers of threads are reached. From there, any of the previously described protocols can be used. In our implementation, both butterfly and dissemination are selected depending on an estimation of their delay computed by summing for each thread to thread path, the weighted message and hop delays. For instance, Fig. 14 shows a dissemination barrier for the intra-group part and a butterfly one for the inter-group part.

The goal for selecting the partition is to obtain groups which are as close to compact rectangles or lines as possible since they are easier to synchronize than other shapes, the compactness ensuring shorter hop delays. For this purpose, we propose the following recursive procedure (Fig. 15 illustrates it for two partitioning passes):

1. Pick F_i one prime factor of N. Since there are usually only a small number of different prime factors (e.g., at most 3 with N varying from 1 to 128), it is possible to try each one for dividing a set before recursing.
2. Partition the set of threads in F_i groups of N/F_i threads according to the topological order of the threads (cf. *simple split* in Fig. 15).
3. Make the groups closer to rectangle shapes by iteratively swapping pairs of corner threads between groups. For now, we use a mountain climbing approach which ends the iteration when no better shape is achieved. The shape is estimated by the rate between the number of its threads and the number of cores in the smallest rectangle including it. (cf. *improved split* in Fig. 15).
4. Retry from 1 for another prime until the best shapes are achieved according to 3.
5. Recurse to 1 on each group if N/F_i is not prime.

Limitations and benefits of the technique:
While promising, divide and conquer cannot be applied when the number of threads is prime. Moreover it is likely to achieve poorly for the cases where the number of synchronizing steps is higher than with a basic protocol. For instance, considering butterfly and dissemination, such happens when $np2(F_0) * np2(F_1) > np2(N)$ (it is still assumed that the N threads are partitioned into F_1 groups of F_0 elements). Indeed, $np2(N)$ is the number of steps required by the two previous protocols for synchronizing the N threads whereas divide and conquer requires $np2(F_0) * np2(F_1)$ steps if we assume that the intra and inter-synchronizations use butterfly or dissemination. Both limitations could be dealt with by adding virtual threads or by using free cores but the benefits are unsure. Instead, we propose a more general technique in the next section which prove to perform well on these cases too.

Apart from the previous cases, it can be shown that for line and rectangle shapes, the divide and conquer approach actually reduces the message and the hop delay for the butterfly protocol and reduces the hop delay for the dissemination. There is no room in this paper for demonstrating this result, but it can be understood intuitively: considering a case where $np_2(N) = np_2(F_0) * np_2(F_1)$, the message delay of dissemination is identical to divide and conquer (using dissemination for the inter and intra-group synchronizations), and improved for the case of butterfly since the first step is free of mirroring overhead (cf. section IV.A.) so that applying this last protocol several times on a partition is more efficient than globally once. The hop delay is improved too because, by construction, the most distant threads never exchange messages directly.

B. *Embedding divide and conquer*

The divide and conquer approach does not deal well with some specific numbers of threads. Moreover, it does not favor the strength of rectangle tree for optimizing the hop delay with distant threads. This is why we propose a final technique which embeds divide and conquer into the rectangle tree protocol presented in section V.C. Specifically, the technique embeds numerous threads in the root of the tree and synchronizes them with divide and conquer while the external threads are synchronized with rectangle tree. For that purpose, the following procedure is used:

1. Apply the first phase of rectangle tree.
2. Connect each thread reached at the last step of 1 to the closest thread located inside the root rectangle.
3. Synchronize the root using divide and conquer.
4. Generate the reverse connections of 2.
5. Apply the second phase of rectangle tree.

In order to adjust the number of threads for divide and conquer while benefiting from the advantage of rectangle tree for distant threads, the central root rectangle is defined iteratively: at first it is small as possible for applying the full rectangle tree, then it is grown progressively while applying divide and conquer inside it. This is repeated until the estimated delay (using weighted message and hop delays as explained previously) is not reduced any longer. It can be noticed that rectangle tree needs to be built and estimated only for the first iteration.

VII. Experiments

The experiments have been held on a NoC multi-core simulator designed internally. This simulator is cycle accurate for the NoC, but executes natively the programs of the cores for sake of speed. The execution time of the threads is taken into account through off-line annotation of their programs. Both the software and the hardware implementations of the barriers have been included into the threads management runtimes executed on the cores. They have also been annotated so that the delay for a single message processing is 12 cycles for the software barrier and 2 cycles for the hardware. Those delays has

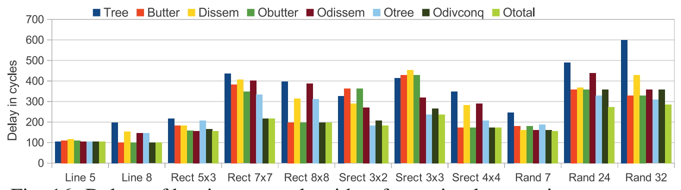


Fig. 16. Delays of barrier protocols with software implementation

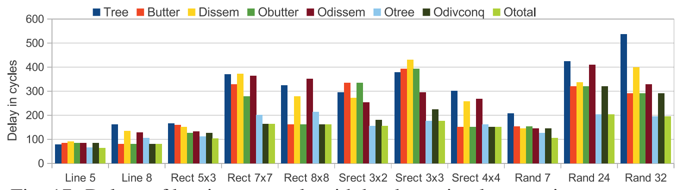


Fig. 17. Delays of barrier protocols with hardware implementation

been evaluated off line for the case of a MeP [15] processor, using the SPM for the messages buffers. For the hardware implementation, each barrier operation was reduced to a read or to a write to specific memory mapped registers. In order to experiment the barriers without interference, the application was made of identical threads, each of them generating one output identification message before barrier synchronizing with the other threads.

Results of the experiments are shown in Fig. 16 for the software implementation and Fig. 17 for the hardware one. In the figure the delays in cycles with various numbers of threads and mappings are given for the basic protocols, i.e, tree (*Tree*), butterfly (*Butter*) and dissemination (*Dissem*), our tuned version for them (respectively *Otree*, *Obutter*, *Odissem*), and our combined approaches, i.e., divide and conquer (*Odivconq*) and the final protocol which embeds divide and conquer into the rectangle tree protocol (*Ototal*). In order to obtain fair results, the threads have been numbered in XY topological order for the basic barrier protocols. On the horizontal axis, *line*, *rect*, *srec*, and *rand* respectively stand for line, rectangle, sparse rectangle and random shapes, and the figures represent the numbers of threads. In addition, Fig. 18 gives the corresponding message and hop delays for some of the shapes used in the experiments.

As it can be seen from the figures, our final protocol, Ototal, always achieved the shortest delays. For Rect 7x7, Srect 3x3 and Srect 3x3, these delays are about half the delay of the basic protocols for the software implementation and less than two-thirds for the hardware implementation. The second most efficient protocols are Otree and Odivconq which are complementary since they performed well on different cases. This is not surprising since Otree optimizes more the hop delay whereas Odivconq optimizes more the message delay. Yet, for the Rect 5x3 with the software implementation case, Odivconq actually performed a little worse than Obutter and Odissem. This was expected because $np_2(5*3) < np_2(5)*np_2(3)$ as explained in section VI.A. Obutter always achieves better or equal than Butter but the results for Odissem are more contrasted since for three cases it achieved worse than dissem, which illustrates the difficulty to optimize dissemination for all the cases.

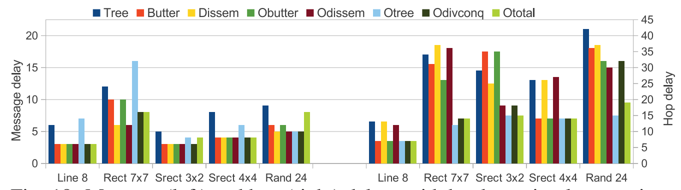


Fig. 18. Message (left) and hop (right) delays with hardware implementation

VIII. Conclusion

In this paper we explored the design of several barrier synchronization protocols for a NoC architecture. We showed that using a proper numbering of the threads and taking into account the shape of the mapping of the threads can significantly improve the performance of the synchronization. Then we showed how basic protocols could be tailored to be efficient on a NoC-based architecture and finally, we proposed a new divide and conquer based protocol and a new global combining protocol. For the experiments, this latter always performed better than the other protocols, and it was able to achieve half the delay of the (non-tailored) basic protocols for several cases.

As future work, we plan first to tailor more finely the protocols. Second, it could be interesting to use free cores in order to achieve better thread numbers or shapes. Third, it has been assumed that the mapping of the threads was already fixed before the barrier are created since the data communication overhead is usually more expensive than the synchronization one. Still, it could be valuable to consider a joint data communication and synchronization aware mapping of the threads. Last, the energy consumption of the various protocols is a major issue and is a base of our future work.

Acknowledgements

This research was supported in part by New Energy and Industrial Technology Development Organization.

References

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. Efficient and scalable barrier synchronization for many-core cmps. CF '10, pages 73–74, New York, NY, USA, 2010. ACM.

[2] C. J. Beckmann and C. D. Polychronopoulos. Fast barrier synchronization hardware. Supercomputing '90, pages 180–189, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[3] E. D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15:295–307, 1986. 10.1007/BF01407877.

[4] A. Darte and R. Schreiber. A linear-time algorithm for optimal barrier placement. PPoPP '05, pages 26–35, New York, NY, USA, 2005. ACM.

[5] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17:1–17, February 1988.

[6] M. Kandemir and S. W. Son. Reducing power through compiler-directed barrier synchronization elimination. ISLPED '06, pages 354–357, New York, NY, USA, 2006. ACM.

[7] Khronos group. OpenCL. `http://www.khronos.org/opencl`.

[8] L. Kontothanassis and R. W. Wisniewski. Using scheduler information to achieve optimal barrier synchronization performance. *SIGPLAN Not.*, 28:64–72, July 1993.

[9] A. Marongiu, L. Benini, and M. Kandemir. Lightweight barrier-based parallelization support for non-cache-coherent mpsoc platforms. CASES '07, pages 145–149, New York, NY, USA, 2007. ACM.

[10] R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin. Scalability evaluation of barrier algorithms for openmp. IWOMP '09, pages 42–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] OpenMP.org. OpenMP. `http://www.openmp.org`.

[12] V. Ramakrishnan, I. D. Scherson, and R. Subramanian. Efficient techniques for fast nested barrier synchronization. SPAA '95, pages 157–164, New York, NY, USA, 1995. ACM.

[13] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. *SIGPLAN Not.*, 42:369–386, October 2007.

[14] J. Sampson, R. González, J.-F. Collard, N. P. Jouppi, and M. Schlansker. Fast synchronization for chip multiprocessors. *SIGARCH Comput. Archit. News*, 33:64–69, November 2005.

[15] Toshiba. MeP processor. `http://www.semicon.toshiba.co.jp/eng/product/micro/mep/index.html`.

[16] O. Villa, G. Palermo, and C. Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. CASES '08, pages 81–90, New York, NY, USA, 2008. ACM.