2011 IEEE 17th International Conference on Parallel and Distributed Systems

Reflex Barrier: A Scalable Network-Based Synchronization Barrier

Ahmad Anbar, Olivier Serres and Tarek El-Ghazawi
NSF Center for High-Performance Reconfigurable Computing (CHREC),
Department of Electrical and Computer Engineering,
The George Washington University
{anbar, serres, tarek}@gwu.edu

***Abstract*—**
High-performance computing is witnessing the proliferation of multi-core processors in parallel architectures, and the trend is expected to increase further with the emerging many-core technology, leading to hundreds of processing cores within each compute node in the near future. Along side with this trend, it is also clear that total number of cores within the whole system is increasing. To be able to harvest the fruits of this massive parallelism, inter-process synchronization and communication should be as lightweight as they can be, and should be relying on as limited involvement as possible of the participating processors/cores. The synchronization algorithms that target shared memory processors are expected not to be able to scale on many-cores as they rely on atomics, locks, and/or cache coherence protocols, which all should be very costly operations on many-cores. In the same time, some many-core architectures provide user space networks on chip (NoCs) that operate similar to regular networks. In this paper, we are introducing the Reflex barrier, a new synchronization barrier algorithm that relies on fundamental networking concepts. As the barrier relies on the characteristics of the network, it requires very little intervention from the participating processors/cores. The algorithm can also be implemented as split phase, which furnish an opportunity to reduce the synchronization cost. We implemented the algorithm using Unified Parallel C (UPC), MPI and pThreads. We tested our implementation on TILE64, a 64-core processor. The performance of the Reflex barrier is also analyzed and compared to other algorithms using performance models.

***Index Terms*—Reflex barrier, Synchronization barrier, Many-cores, Distributed memory barrier, Many-core clusters**

I. Introduction

The trend of exploiting faster clock speeds and instruction-level parallelism in microprocessors has subsided due to unsustainable thermal and power overheads. And since Moore's law is still holding, that is the number of transistor on a integrated circuit doubles every two years [1], the current trend is adding more processing cores on the same chip to utilize the added transistors. While adding more cores to the chip allows for achieving higher degrees of parallelism, it also brings the universal overheads of parallel programs such as communication and synchronization. Although these cores usually share the main memory and may be one or more levels of the cache hierarchy, as the number of cores gets bigger, we would expect that relying on shared memory to do synchronization and communication will not be scaling.

The main problem that will face the shared memory synchronization barriers algorithms that most of them rely on the cache coherence protocols to achieve high performance. That is making threads spin on a shared variable in their local caches. When other threads update that shared variable, the cache coherence protocol will propagate the new value to the spinning thread. Without this cache coherence, the shared memory synchronization barrier algorithms will not be able to scale. The biggest challenge for the many-core architecture now is how to implement a scalable cache coherence. Apparently, there is no ideal solution for this problem even using the state of the art technologies. Processors manufacturers dealt with this problem in one of two ways, giving up cache coherence as in Intel's Single-chip Cloud Computer SCC [2] or they relied on some techniques like homing memory pages to cores, such as in Tilera's TILE64 architectures [3]. Unfortunately, these techniques will not help scaling the synchronization algorithms as will be explained in detail later. Another major problem with the current synchronization and collective communication operations algorithms is that they require heavy involvement of the participating processors, i.e. rely on recursive doubling or tree based algorithms to communicate data or propagate synchronization signals between participating processors. This sets a strong limitation on their performance and thus their scalability.

One architectural characteristic that is expected to be supported in many-core architectures is the presence of the user-space messaging networks. These networks-on-chip (NoCs) allow the user-space messaging between the cores without the need to do an operating system call. These NoCs will allow the communication between cores without relying on shared memory and without the need to rely on a messaging mechanism built on top of shared memory. These NoCs are very attractive as they are usually using the concepts from the traditional interconnects. It is already proven in [4] that relying on messaging rather than relying on shared memory in on-chip communication is the way to exploit the powers of many core architectures.

In this paper we are introducing the Reflex barrier algorithm. The Reflex barrier is a synchronization barrier that does not rely on shared memory. Instead, it relies on some common properties of the interconnection network. These network properties, which are required to be able to implement the Reflex barrier, can be commonly found in many of the available interconnects such as InfiniBand [5], and it is also available on some of the NoCs. We believe that a user-space network with these properties will be available in a lot of many-core architectures, if not all of them.

We implemented this algorithm on the TILE64 processor, and evaluated and compared its performance to other synchronization barrier algorithms. We also extended our evaluation and comparisons to a larger number of cores using performance models. To do the performance models evaluations, we used the LogGP model after adding an extra parameter to account for the per hop latency.

The rest of this paper is organized as follows. Section II discusses the related work on barrier implementations. Section III introduces the Reflex barrier algorithm and discusses the set of interconnection network characteristics required to implement the Reflex barrier. The implementation details and its results are discussed in section IV. Section V discusses the results and the performance model evaluation of the algorithm. Finally, Section VI concludes the paper and show the possible future directions.

II. Related Work

A lot of algorithms and techniques have been proposed for parallel tasks synchronization. Most of these are targeted to shared memory architectures. In [6], Tang and Yew presented a centralized barrier. It relies on a central shared variable seen by all the contributing thread to the barrier. The variable is initialized by the count of the contributing processors. Each thread that arrives at the barrier, acquire a lock, decrements the shared variable, releases the lock and spins on another shared variable to be modified. When the last thread arrives, after it decrements the shared variable, it finds out that the value is zero and then modifies the other variable that the rest of the threads are spinning on, waiting for its modification. The centralized barrier main problem is the presence of a critical section in the barrier logic that will considerably limit its scalability. Also, the algorithm relies on the shared state of the cache coherence protocols to achieve good performance, as without it, multiple threads spinning on the same variable will dramatically affect the performance.

Another widely used barrier is the dissemination barrier, which was introduced by Hensgen, Finkel and Manber in [7]. Dissemination barrier takes $\lceil log_2 P \rceil$ of signaling rounds to complete. In round k, processor i signals processor $\left(i + 2^k\right) mod\, P$. As stated, the dissemination barrier algorithm is widely used because it is considered to be one of the most scalable algorithms as shown in [8]. Another advantage of the dissemination barrier is that the signaling between the processors can be implemented using any kind of messaging, so it can be implemented without using shared memory.

A distributed memory synchronization mechanisms called rendezvous and multirendezvous was proposed by Gupta and Panda in [9]. The mechanisms was also trying to overload the wormhole-based flow control interconnection networks to provide synchronization primitives similar to the Reflex barrier. It also reduces the number of needed synchronization steps. As one multirendezvous can be used to make each thread participating in it aware of the arrival of other participating threads. The main idea of the rendezvous mechanism, is to flood the network between the sender and receiver with a number of flits that cannot be hold in the intermediate buffers along the path between the sender and the receiver. The problem with this technique rises when the distance between the participating nodes is very large. It would take a long time for the sender to fill all the intermediate buffers and it will also take a long time for the receiver to consuming all the flits to make the network ready for the next barrier. And although not clearly stated, the multirendezvous technique relies on special hardware support, that allows the intermediate nodes to control the forwarding of the message to the ones after.

A main drawback in the above algorithms is the inability to implement them as split-phase barriers. A split-phase barrier is a synchronization barrier with separate notification and waiting phases. This allows the participating threads to notify the other threads that they reached a certain synchronization point, then they can still progress by doing more useful computations. Later, they can wait for other threads until they reach the synchronization point. As described in [4] the split-phase synchronizations and communication is key to achieve high performance by allowing overlapping communication and computation. This becomes more important when following a programing paradigm that supports one-sided communications such as the Partitioned Global Address Space (PGAS) programming model [10] or other shared memory model. It is also useful to note that a full barrier can easily be implemented by the split phase constructs, by calling the notify construct followed by the wait construct on all threads.

Hoare and Dietz introduced the Aggregate Function Networks (AFNs) in [11]. The AFNs aim at offloading the synchronization and collective communication operations from the participating processors to the networking hardware. Thereby, they require a specialized networking components, i.e. special networking interfaces and switches, to be able to implement these operations. Although the Reflex barrier also relies on the networking properties, these properties can be found in many intra- and inter-node interconnection architectures.

III. Reflex Barrier

A. Algorithm Description

The name of the Reflex barrier is attributed to its analogy to the spinal reflex arc [12]. This characteristic allows reflex actions to occur relatively quickly by activating spinal motor neurons without the delay of routing signals through the brain. The main purpose of the reflex arc is to provide a means for immediate withdrawal from dangerous stimuli, like touching a very hot surface. In this case, the spinal cord process the signal and take forward an action to the muscles to retract without waiting for the brain to analyse the situation and take an action. Similarly, the Reflex barrier is relying on the interconnect -

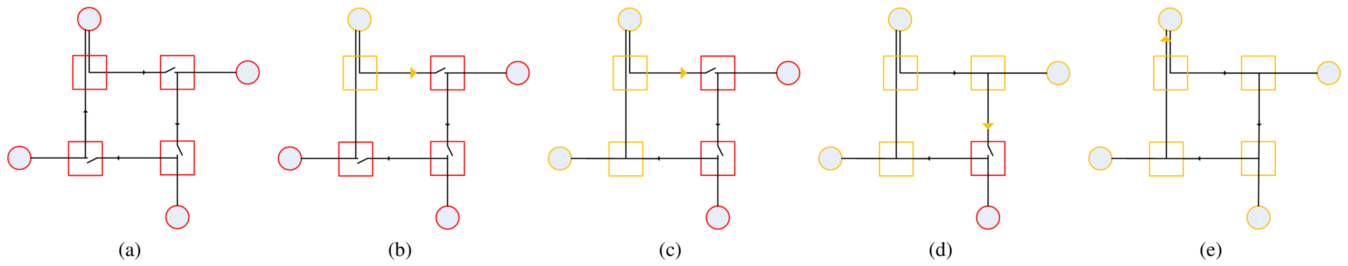


Fig. 1: Reflex barrier notify phase (a) Initial Setup (b) After root arrival and notify signal is out (c) After another processor arrival (d) After a processor arrival and the notify signal forwarded (e) Notify signal reaches the root after all processors pass the notify step

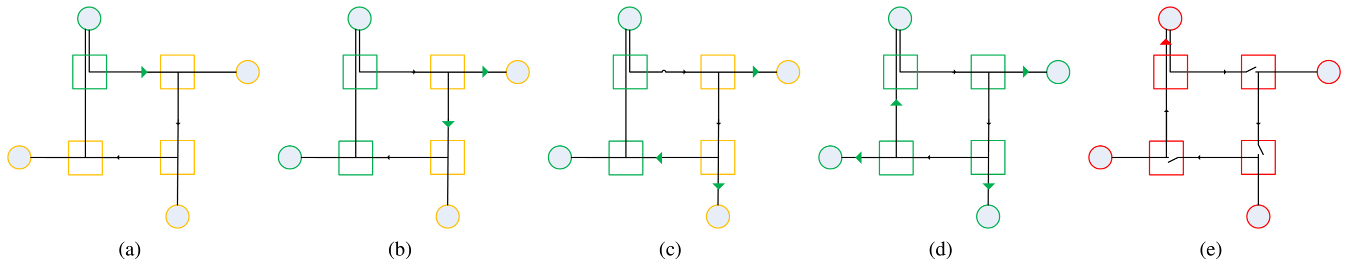


Fig. 2: Reflex barrier wait phase (a) The root reaches the wait phase and sends the wake up signal (b) Another processor reaches the wait phase (c) The propagation of the wake up signal (d) All processors arrived at the wait phase (e) The root reset the switches back to off state

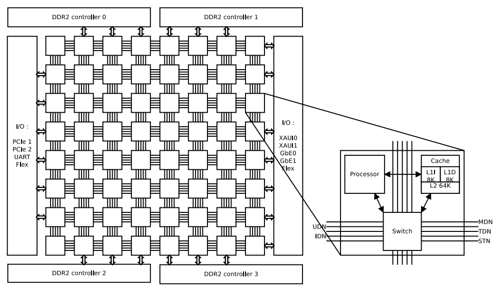


Fig. 3: TILE64 architecture diagram [3]

the spinal cord - to forward the actions without relying much on the involvement of the participating processors - the brain.

To introduce the Reflex barrier, we will start by describing an abstraction of the barrier steps. Then we will point out the necessary networking characteristics that are needed to implement this barrier. Then the algorithm will be presented in terms of these characteristics. The Reflex barrier can be implemented as a split phase barrier with two phases: a notification phase and a wait phase.

Figure 1 (a) shows the initial status of the participating processors. The circles represents the participating processors and the squares represent the switches. Every switch is associated with one of the processors. Every processor has one inbound port and one outbound port connecting it to the associated switch. Also, every switch has several inbound ports and several outbound ports. One inbound port and one outbound port of a switch are connected to the associated processor. The other inbound and outbound ports are used to connect the switch to its surrounding switches. A switch can only forward messages, i.e. it cannot consume or originate network traffic. A switch can forward messages from its associated processor or from any of its surrounding switches. A switch can forward the same incoming message to multiple output ports. To be able to implement the Reflex barrier, the switches are connected to each other to form a ring. The ring is broken at one of the switches and one of the processors is connected in series to the ring. This processors is designated as the barrier root, in the figures it is the top left processor. The root's behavior in the notify and wait phases is different than the rest of the processors. Also the way it is connected to the ring is different. The other processors are just connected to the ring by a link from their switches. Except for the root switch, all other switches initially are inactive, and not passing signals to the rest of the ring or the attached processor. When a signal reaches an inactive switch it is blocked at the switch till it becomes active and forwards it.

- **The Notify Phase:** Figure 1 shows different steps along the progress of the notify phase. Assuming a weak consistency model, the first step in the notify phase on all the processors should be issuing a memory fence instruction. This is to make sure that all the memory

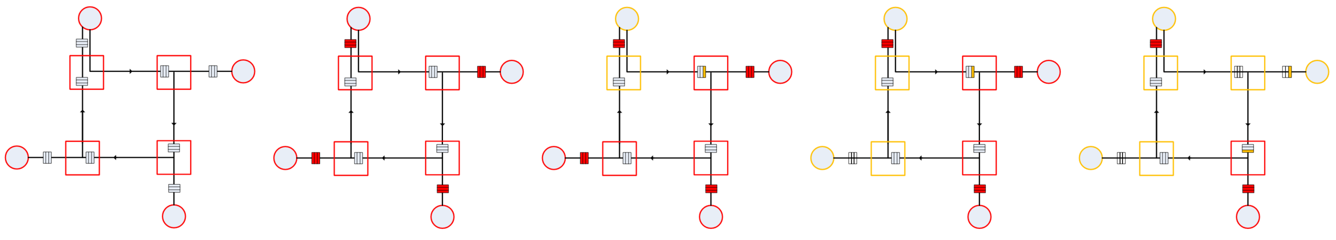

(a) (b) (c) (d) (e)

Fig. 4: Reflex barrier notify phase (a) Initial Setup (b) After *Init* step done by root (c) After root arrival and sending the notify signal (d) After another processor arrival (e) After a processor arrival and the notify signal forwarded

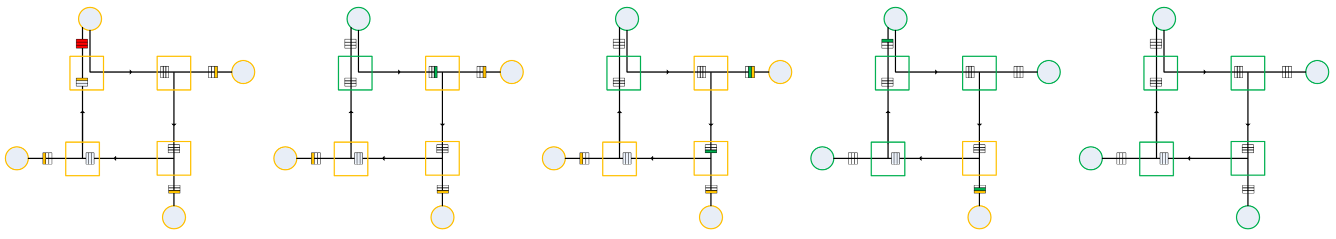

(a) (b) (c) (d) (e)

Fig. 5: Reflex barrier wait phase (a) The root reaches the wait phase and sends the wake up signal (b) Another processor reaches the wait phase (c) The propagation of the wake up signal (d) All processors arrived at the wait phase (e) The root reset the switches back to off state

references before the notify phase are completed. As mentioned before, the root behavior in the notify and the wait phases is different than other processors. When reaching the notify point, the root issue a memory fence instruction, and then it sends a signal on the ring as shown in figure 1 (b). The yellow color depicts that a processor arrived at the notification phase, and the yellow triangle depicts the notification signal. Since the other processors did not arrive at the barrier yet, the notification signal is blocked. By this the root has completed its notify logic. For the other participating processors, the notify phase begins by issuing the memory fence instruction as well. Then each processor will activate its switch allowing it to forward incoming signals as it appears in figure 1 (c) to (e). This completes the notify phase on the rest of the processors. After completing its notify phase, a processor is able to continue computing until it needs to synchronize with other processors, at which time it starts the wait phase. It is clear that adding one extra processor will only cost one extra network hop latency. It is expected, on a many-core chip, that this latency will be in the terms of nanoseconds. So, adding a thousand cores will only increase the latency of a few microseconds.

- **The Wait Phase:** Figure 2 illustrates the wait phase of the reflex barrier. The wait phase at the root again is different than at the other processors. When the root reaches the wait phase, it waits for the signal it sent in the notify phase to come back to it at the other end of the ring. When it receives it, it becomes aware that all the processors on the ring has reached their notify phase, as the signal would not be propagated to the other end unless all the switches are activated by their attached processors. At that time it sends another release signal on the ring as in figure 2a. The other processors when they reach the wait phase they just wait for the release signal sent by the root as in figures 2 (b) to (d). The root then deactivate the switches to prepare for the next barrier episode as in figure 2e.

It is also notable that a full barrier is equivalent to calling the notify logic directly followed by the wait logic on all threads.

B. Required Network Characteristics

The barrier logic presented so far can make use of regular network characteristics to be able to perform the tasks described above. These characteristics can be found in many interconnects, so there is no need to have special hardware support. These features are commonly found in many on-chip networks as well as inter-node networks. From the above discussion, we can list three needed features to be able to setup the required infrastructure for the Reflex barrier. The first feature is to be able to control the routing on the network. The second is to have a multicast support on the switches. The third needed feature is to have a mechanism that allow the switches to block or forward incoming traffic. These characteristics are described in more details in [13].

Algorithm 1 Reflex Barrier

```
Init:
  // Init has to be executed one and only one time by the root
  if $root$ then
    SEND msg[buffer size]
  end if
End Init

Notify:
  Memory Fence
  if $root$ then
    SEND msg[1]
  else // other threads
    RECV msg[buffer size]
  end if
End Notify

Wait:
  if $root$ then
    RECV msg[buffer size + 1]
    SEND msg[buffer size + 1]
    RECV msg[1]
  else // other threads
    RECV msg[1]
  end if
End Wait
```

The first required feature, that is controlling the routing, is needed to be able to setup the ring shaped topology that all the processor are attached to it. This feature is actually present in an on-chip network like the static network on the Tile based architectures, which will be described in more details later. It is also present in a inter-node network like InfiniBand. In InfiniBand, the routing is dependant on the forwarding tables at each switch. These routing table can be modified by the subnet manager to achieve the desired setup. This feature is used to setup the ring like topology described earlier.

The second feature is the ability to multicast. This feature is needed as when a signal arrives at a switch, it needs to be forwarded to the attached processor and to the next processor on the ring. Again this feature is present on many-core on chip network like the Tile static network, and an inter-node level interconnect as InfiniBand. This feature allows that switches to be setup to forward incoming message along the ring and to their associated cores at the same time.

The last important feature needed is the ability to block network traffic on a switch until the associated core arrives at the notify phase. The key to achieve this feature is to have a wormhole or virtual cut-through flow control mechanisms with a credit based buffer management. The wormhole and virtual cut-through are buffered, flit based flow control mechanisms, which are different than bufferless and packet based mechanisms. The buffers at the switches handle messages in units of flits. The credit based buffer management is a concept related to flit based mechanisms. It is the backpressure mechanism that informs the sender or an intermediate node along the message route that the next switch buffers are full and thus it cannot forward more flits. The basic idea that each output port at a switch starts with a credit that is equal to the buffer capacity. When a switch sends or forwards a flit to this given port, it decrements its credit by one. When this credit reaches zero, a switch blocks all incoming traffic for that output port as it knows that there is no available buffer space at the next switch. In case of multicast, i.e. the incoming traffic message should be forwarded to more than one output port, the forwarding stops on all ports if one of the output ports credit reaches zero. When the buffer space becomes available on the next switch, which happens when it forwards some of the flits in its buffer, it sends credit back to the previous switch to inform it that it may forward more flits. We used this behavior to implement how switches are turned on and off. So to turn off a switch, we fill the output buffer of the associated core. Thus all the incoming network traffic will be blocked because of the zero credit of the output buffer at the associated core. Intuitively, to turn on a switch, a core just has to read the flits in its buffer. This will allow later traffic to be propagated along the ring.

IV. Implementation

As we were mainly targeting many core architectures with this barrier implementation, we used a TILE64 - a 64-core processor - as a testbed.

A. TILE64

A TILE64 board comes with 4 GB of memory. The TILE64 processor features 64 identical 32-bits processor cores (tiles) interconnected with Tileras iMesh on-chip network [14]. Each tile consists of a complete, full-featured 3-way VLIW processor running at 700MHz as well as a 8KB of L1 data cache, 8 KB of L1 instruction cache, a 64 KB private L2 cache and a non-blocking switch that connect the tiles into the mesh. Four on chip memory controllers connect the tiles to on-board DDR2 memories. Figure 3 shows the architecture diagram of the TILE64 processor.

The iMesh consists of five separate networks [14], two of which are hardware controlled that are used to handle memory and cache requests. The three remaining networks are under the control of the user. After studying the features offered by each network we found that the static network (STN) is the one that meets our requirements. STN is meant to be used for scalar messaging between tiles. But since it is based on the general networking concepts described above, we were able to use it to implement the Reflex barrier. STN routes are configured by the user. A tile switch can be configured to forward or multicast incoming traffic to its processing core and/or any of its surrounding switches. The cost of one hop on STN is one cycle, that is 1-2 nanoseconds. This means adding an extra processor to the processors contributing to a Reflex barrier, which corresponds to an additional hop on STN, will just cost additional 1-2 nanoseconds. The routing is fixed as long as it is not reconfigured by the user.

The user dynamic network (UDN) is also a user space messaging network. The main differences between UDN and STN is that UDN is not a scalar network and the routing in UDN is not controlled by the user.

B. Algorithm

Algorithm 1 describes the steps of a Reflex barrier. Also figures 4 and 5 illustrate the barrier at different stages. The small rectangles represent the buffers at the switches or at the cores. The first step that needs to be done is the *Init* step. In this step the root sends a message with length that is equal to the input buffer of the cores, to block later networking traffic, i.e. turn off the switches. This is illustrated in figure 4 (a) and (b). The *Init* step must be done by the root before any barrier calls can proceed. If another thread arrives at the barrier before the root completes the *Init* step, it will just block there until it is completed. So this does not affect the functionality of the algorithm.

The *Notify* phase is illustrated in figure 4 (c) to (d). As shown in Algorithm 1, the first step is to issue a memory fence to ensure that all the past memory references are completed. Afterwards, the root should send a single flit message through the network. All the other threads should only empty their buffers. Doing this, they allow the message sent by the root to propagate through the network. After completing their notify steps, participating threads can proceed with their computation. This allows for overlapping computation and communication.

When the threads need to make sure that other threads arrived at the notify step, they should start the *wait* logic. The *wait* logic is illustrated in figure 5 (a) to (e). The *wait* logic on the root is to make sure of the arrival of the signal it sent on the notify step. To be able to do that it needs to read in a message of length equals to the incoming buffer + 1. That is, it should read the same message used to block the other switches, and an extra flit sent in the notify phase. When it receive that extra flit, the root knows that the rest of the threads have already arrived at the notify step. Accordingly, it sends another release message. It also prepares the network for the next barrier episode by sending a message with size equals to the size of the receive buffers at the other cores. To avoid input buffer overflow at the root, it should also receive the release message. The other threads only need to receive the release message in the *wait* step and they can proceed past the barrier after they do.

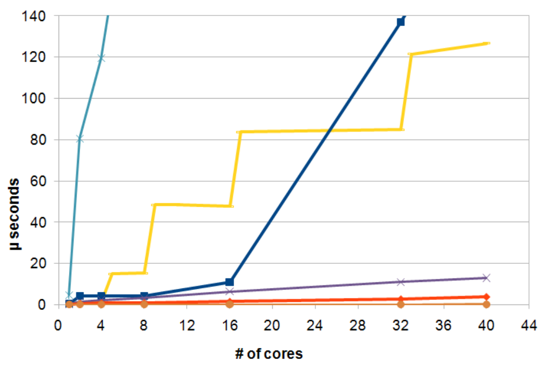
V. Results and Discussion

As described in the previous section, the Reflex barrier was implemented on the TILE64 platform. To be sure that the programming model does not have any effect on the barrier performance, we implemented the barrier using pthreads, UPC and MPI. UPC is an explicit parallel extension of the C language supporting the PGAS programming paradigm [15], [16]. These three libraries/languages was chosen to represent shared memory, PGAS, and message passing programming models respectively. The three implementations gave identical results. To compare the performance of the reflex barrier, we implemented the Centralized barrier and the Dissemination barrier. The high level description of these algorithms was given in section II. We also compared the performance to the two barriers implementations available from Tilera, the Tile Sync Barrier and the Tile Spin Barrier. In the Tile Sync Barrier, the threads sleeps if they arrive at the barrier and other threads are still on their way. Later, the sleeping threads wake up and check for other threads arrival. As expected, the performance of this barrier is very poor. Alternatively , the Tile Spin Barrier does not make the threads that arrive at the barrier sleep. Instead they busy wait until all other threads arrive. Although the performance of this barrier is much better than the Tile Sync Barrier, it consumes a lot of memory bandwidth. If the application that utilizes this barrier is performing a lot of memory operations just before the barrier call, the performance is expected to degrade even more.

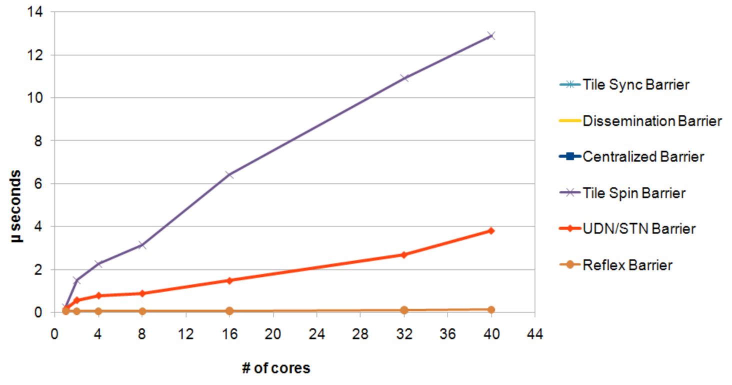
The first experiment we conducted is having a loop that calls the barrier on each thread. We measured the total time to complete the loop and divide the time by the number of iterations to get the time consumed by one barrier episode. We conducted this experiment for the different barrier implementations. Figure 6 shows the obtained results. It is clear from the results of the poor performance of the Tile Sync Barrier, the centralized barrier, and the dissemination barrier. The results is also shown for the Tile Spin Barrier, the Reflex barrier, and the UDN/STN barrier. The UDN/STN barrier is simple barrier in which all threads send a message on User Dynamic Network (UDN) to the root when they arrive at the barrier. When the root receive messages from all threads, it sends a release broadcast signal through STN. We implemented this barrier when we were trying to get a feeling of the scalability of the network-based barriers. The results shows that the network based barriers were performing much better than the other barrier algorithms based on shared memory. To better show the scalability of the network-based barriers, especially the Reflex barrier, we are showing figure 6 (b) which is a 10x zoomed version of figure 6 (a). From this, we can conclude that to achieve better barrier scalability, we should not rely on shared memory. The results shows the perfect scalability of the Reflex barrier, and really good scalability of the UDN/STN barrier as compared to other shared memory based algorithms.

To achieve more performance and consume less power, we argue that many-core architecture should have dedicated synchronization networks. These dedicated networks could be only 1-bit wide, and they should have the smallest possible buffer size.

According to [17], in practice the threads never arrive at the barrier at the same time. In the above experiment, the threads almost hit the barrier at the time. Figure 7 (a) illustrates the situation in which it is mostly likely that the threads will be released almost at the same time if they arrived at the same time. We are also expecting that the difference between the threads arrival and threads release is minimal, i.e. the

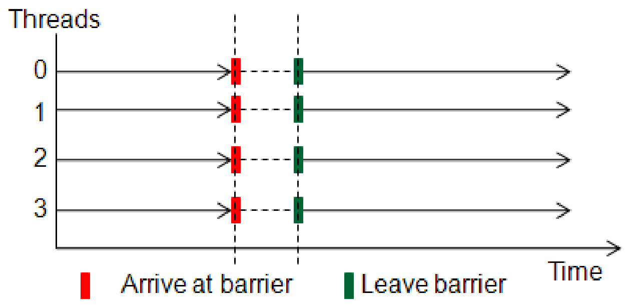


(a)

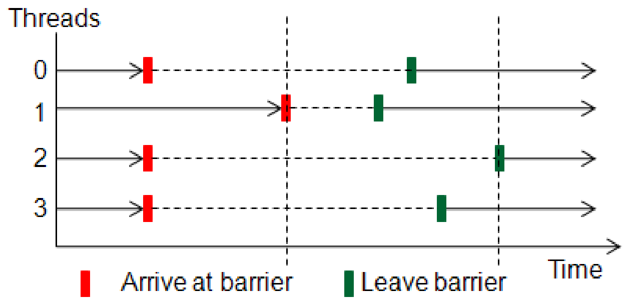


(b)

Fig. 6: Barriers performance on the TILE64 (a) all algorithms (b) zoomed



(a)



(b)

Fig. 7: Threads arrival patterns (a) together (b) jittered

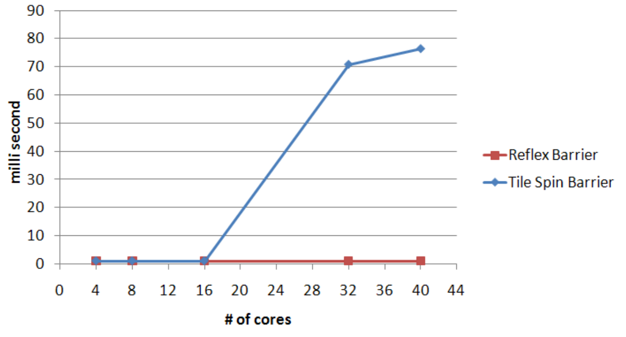


Fig. 8: Effect of jittered arrival on barriers performance

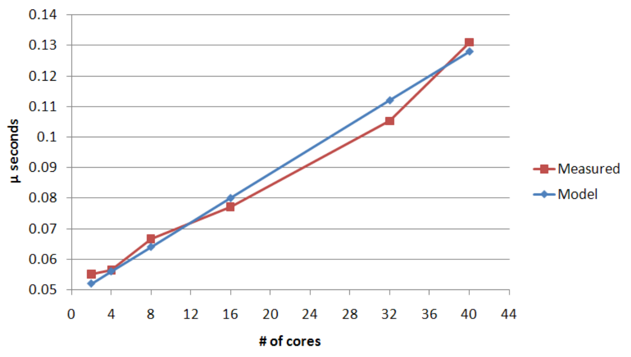


Fig. 9: Modeled and actual performance of reflex barrier on TILE64

difference between the vertical dotted lines is small. To get more practical feeling of the barrier performance, we designed another experiment in which we forced a jitter in the arrival of one of the threads as in figure 7 (b). We are expecting that the will be bigger difference between the last thread arrival and the last thread release from the barrier. The results for comparing the performance of the Reflex barrier versus the Tile Spin Barrier in the second experiment is given in figure 8. We can see that both algorithms were performing well till 16 cores. After that the Tile Spin Barrier scalability dropped severely while the Reflex barrier continued to scale.

We were also interested to study the scalability of the barrier on inter-node level using a high-speed interconnection network like InfiniBand [5]. We used the LogGP [18] performance model to estimate the behavior on InfiniBand. We first verified the accuracy of the model on our on-chip implementation

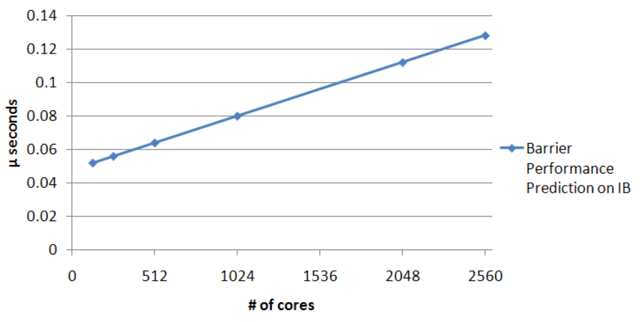


Fig. 10: Modeled performance of reflex barrier on IB

of the barrier. The verification results are given in figure 9. From the curves we can see that the model gives a good approximation of the real performance. We used the values given by [19] for the model parameters when estimating the InfiniBand performance. The estimated performance of the barrier on InfiniBand network is given in figure 10. We can see that the barrier is scaling linearly on InfiniBand. The slope of the line is very small, so we are expecting that although the barrier time grows linearly the algorithm will be scaling for large number of nodes.

VI. Conclusions and Future Work

The transition to multi- and many-core processors unleashed new ways of reaching performance gains. It also added a big burden on the programmers to be able to exploit these gains. Treating the many-core processors as regular SMPs will just be wasting a lot of the chips capabilities. Thus on many-core chips programmers need new scalable techniques to implement the same mechanisms that exist on SMPs. One of these mechanisms is barrier synchronization. In this paper, we introduced a novel approach to implement synchronization barrier for many-core chips. The approach is named Reflex barrier. Reflex barrier relies on existing networking characteristics to be able to achieve synchronization between parallel threads. We implemented the Reflex barrier on TILE64 - a 64-core processor. The results showed that relying on regular shared memory synchronization mechanisms will not be scalable on many-core systems. The Reflex barrier outperformed all the other barriers algorithms, even the ones available in the TILE64 native libraries.

For future directions we are evaluating the extension of the Reflex concept to other collective communication operations, such as the broadcast, reductions, scatter, ... We are also interested in evaluation of the Reflex barrier on inter-node network such as InfiniBand.

Acknowledgment

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

References

[1] G. E. Moore, “Readings in computer architecture,” M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming more components onto integrated circuits, pp. 56–59. [Online]. Available: http://portal.acm.org/citation.cfm?id=333067.333074

[2] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, “Light-weight communications on intel’s single-chip cloud computer processor,” *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945033

[3] Tilera Corporation, “www.tilera.com.”

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, “The multikernel: A new os architecture for scalable multicore systems,” in *SOSP ’09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 29–44.

[5] InfiniBand Trade Association, “www.infinibandta.org.”

[6] P. Tang and P.-C. Yew, “Processor self-scheduling for multiple-nested parallel loops.” in *ICPP’86*, 1986, pp. 528–535.

[7] D. Hensgen, R. Finkel, and U. Manber, “Two algorithms for barrier synchronization,” *Int. J. Parallel Program.*, vol. 17, pp. 1–17, February 1988. [Online]. Available: http://portal.acm.org/citation.cfm?id=54616.54617

[8] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, February 1991. [Online]. Available: http://doi.acm.org/10.1145/103727.103729

[9] S. Gupta and D. Panda, “Barrier synchronization in distributed-memory multiprocessors using rendezvous primitives,” in *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, apr 1993, pp. 501 –505.

[10] Partitioned Global Address Space, “www.pgas.org.”

[11] R. Hoare, , R. R. Hoare, and H. G. Dietz, “A case for aggregate networks,” in *Proc. Eighth IEEE Symp. on Parallel and Distributed Processing*, 1996, pp. 306–313.

[12] Wikipedia, “Reflex arc — Wikipedia, the free encyclopedia,” 2011, [Online; accessed 10-June-2011]. [Online]. Available: http://en.wikipedia.org/wiki/Reflex_arc

[13] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[14] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, “On-chip interconnection architecture of the tile processor,” *Micro, IEEE*, vol. 27, no. 5, pp. 15 –31, sept.-oct. 2007.

[15] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*, May 2005.

[16] O. Serres, A. Anbar, S. G. Merchant, A. Kayi, and T. El-Ghazawi, “Address translation optimization for Unified Parallel C multi-dimensional arrays,” in *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) in IEEE IPDPS Workshops (IPDPSW)*. IEEE, 2011.

[17] A. Faraj, P. Patarasuk, and X. Yuan, “A study of process arrival patterns for mpi collective operations,” in *Proceedings of the 21st annual international conference on Supercomputing*, ser. ICS ’07. New York, NY, USA: ACM, 2007, pp. 168–179. [Online]. Available: http://doi.acm.org/10.1145/1274971.1274996

[18] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, “Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation,” 1995.

[19] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, “Logfp - a model for small messages in infiniband,” in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006, p. 6 pp.