Barrier comparison

BENBIHI Assia

Abstract

This report presents several barriers studied in class in a particular implementation so as to illustrate their performance and their already discussed characteristics. In particular, it recall the logic of sense reverse centralized and tree barrier and Open-MP experiments show the overhead performance due to non-optimized spinning and synchronization in the centralized barrier. It then recalls the design of tournament barrier and MCS barriers in the context of distributed machines for which MPI experiments show the importance of having barrier with statically determined spin location and the relevant choice of Mellor-Crummey and Scott to design two separate tree for its barrier and their respective fanin. This way, this report goes deep into barrier implementation challenges and gives a personal perspective to classic barriers.

1. Introduction

Synchronization issues occur as soon as a computation wants to benefit from parallelization. This technique has the obvious advantage to divide work among several computation units which intuitively seems more efficient. However, as in every team work, this strategy is *really* efficient if and only if the team communicates well. In computer science, this communication is ensured by synchronization mechanism such as locks and barriers. Locks allow different team members to interact with a shared memory variable without it being incoherent for the other members. Barriers allow a team to go through the work at the same global speed by making each member of the team wait for the other at this barrier. Both mechanisms seems to be necessary structure for team work but they may become a “drag” if not implemented well. This comes from the inter-team member communication needed for them to perform correctly. An intuitive example would be about spin lock: if each member anarchically and recurrently tries to acquire the lock, many of the tries will result in a fail and the CPU would have scheduled them for nothing. However, if a software designer can implement a rule to acquire the lock, form the simplest one as a spin delay on the lock or a hierarchical structure as the MCS-lock, it would avoid the CPU running unnecessary instructions and gain performance. The same goes for barriers that will be studied here. This report first starts by recalling the design and theoretical evaluation of complexity for the sense reverse centralized and tree barrier, the tournament and MCS barrier. It then exposes the performance measurements and results which illustrates the statements with experiments run both on a shared memory machine and a distributed memory machine.

2. Barriers

This section presents the tested barriers and how they were designed. Pseudo codes are at the end of this document.

2.1 Sense reverse centralized barrier

In such barrier, the overhead comes from high contention on shared variables and/or non-statically determined spin location.

The centralized barrier is the most intuitive one but as it mimics well the concept of barrier one human may have, it does not perform well on computers. We illustrate the algorithm in the figure below. Each barrier has a count initialized with the number of processes running. When a process arrived at the barrier, it decrements the count and wait for all the other processes to arrive. To do so, it wait for the counter to be reset to P which is done by the last processor.

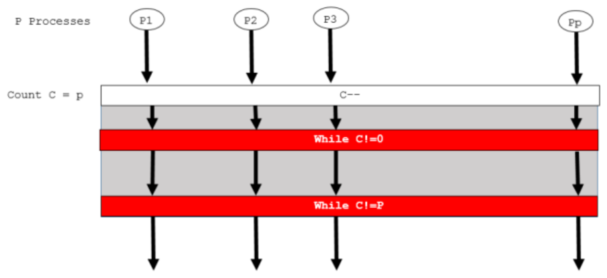


Figure 1 : Centralized Barrier

The obvious drawback of this barrier is that it requires two loops:

- One is for the threads to wait for the last one which set the count to 0.
- The second is to prevent the waiting threads to resume before the last thread resets the barrier.

To avoid these two spinning episodes, the sense reverse centralized barrier uses a sense variable on which the processes who have already arrived spin while waiting for the last process to arrive. As in the previous barrier, each process decrements the count but then wait for the sense reverse to be flopped. This is done by the last process only after je has reset the barrier count. This way, we get rid of the additional spin episode and maintain equity between the processes.

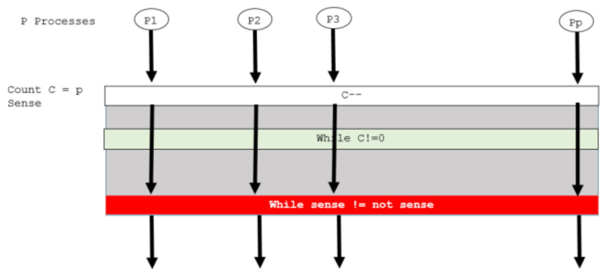


Figure 2 : Sense reversed centralized Barrier

However, the sense reversed barrier has also drawbacks.

- It makes all the processes spin on a unique sense variable.
- It makes all the processes modify one shared variable.

If the processes run on a cache coherent machine, the synchronization overhead induced may be ignored if each process spins on local copy of the sense but there is still the overhead due to high interconnection traffic due to memory updates messages (write invalidate/write update) sent to each process and due to the contention to access shared data structure. Time performance measurements show that even in cache coherent machine, time to get access to the critical part of the barrier, i.e. decrementing the count composes the main part of the barrier overhead. If the machine is a non-cache coherent, then there exists contention not only to access the shared counter but also for each process to spin. This contention goes also with high interconnection

traffic as each process try to acquire the variable in “anarchical” order that is all at the same time even if the variable is not available.
A first approach to reduce interconnection traffic is to try to order processes traffic by changing the basic spin episode to spin with delay. There is still to the type of delay to determine which is done experimentally. According to the results, all spinning give equivalent results in term of both global time achievement and spinning contention but the linear back off seems slightly better which goes in the same direction as the results of the MCS paper. The numerical value of the delay is left to the designer appreciation depending on his OS structure: in these experiments, the delay is relevant is it less than $1.10^{-11}s$.

2.2 Sense reverse Tree Barrier

To reduce these unnecessary contention, the sense reverse barrier can also go through a “divide and conquer strategy” implemented as a binary tree structure.

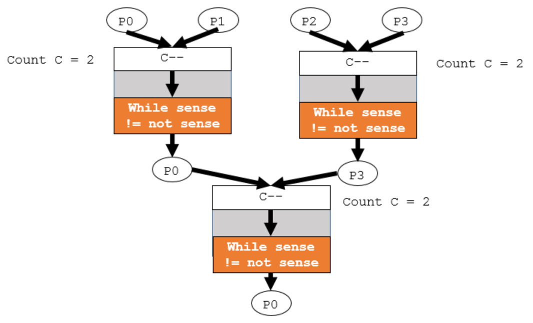
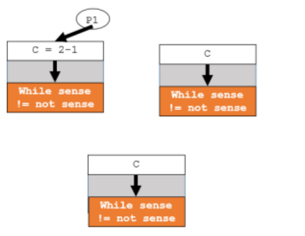


Figure 3 : Tree sense reversed centralized Barrier

This can be viewed as there were sub-barriers in the barrier. Each process arrives at the base of the tree and is assigned a leaf in the three. Each node is micro-barrier with its own count set to 2 and sense variable.

To illustrate the code, we illustrate the steps of algorithm on the simple case of 4 processes with following order arrival: P1-P2-P3-P0



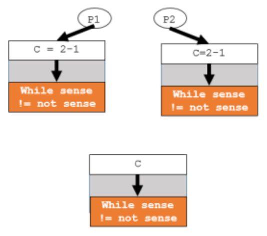
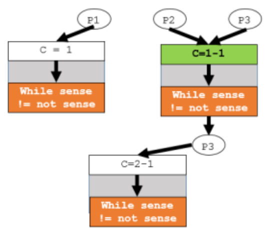


Figure 4.a : Example Tree sense reversed centralized Barrier Figure 4.b : Example Tree sense reversed centralized Barrier



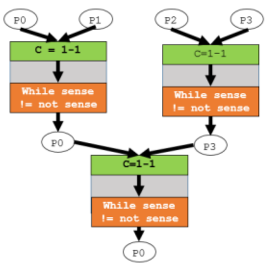
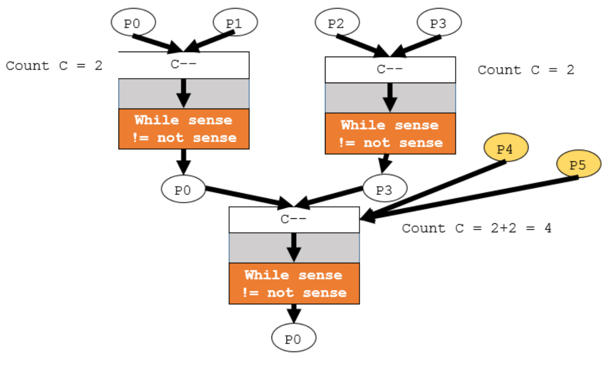


Figure 4.c : Example Tree sense reversed centralized Barrier Figure 4.d : Example Tree sense reversed centralized Barrier

This has the advantage of assigning local variable to modify and spin on to pairs of processes instead of the whole set of processes. This reduces contention and time performance as the experiments on tree barrier show.

Even if the implemented tree is a binary tree, this barrier can still be used with a number of process that is not a number of 2. One may wonder if the performance then changes for better or for worse. This depends on the implementation and here a two examples of how the barrier can be modified. Let $P = 2^N + k$ the number of processes.

- Method 1: The binary tree structure is built for the first 2^N processes and the last k processes directly point to the root of the tree which counter is now$2 + k$.
- Method 2: The binary tree is built with $2^{1+\log_2 P}$ with empty nodes for the $2^{1+\log_2 P} - P$ left nodes.



P=6 Figure 5: Tree sense reversed centralized Barrier

Even if the second method required more memory and more programming efforts, it is better from a synchronization point of view since it keeps the advantages of distributing contention while the first one reduces this effort by making the root be a hot spot for the k left processes. We evaluate the performances evolution with odd and even number of processes and observe that this criteria does not have specific effect of the barrier performance but the time achievement is not linear with the number of process.

Still the tree barrier has a major drawback that is making the process spin on dynamically located variables. It is possible to counter this drawback with better algorithm such as the tournament and MCS barrier. These two barriers will be studied in the context of a distributed system.

2.3Tournament barrier

This barrier is an update of the previous tree barrier so as to make each process spin on a static location. This is allowed by setting in advance which process is going to spin on each sub-barrier. For example, it can be set that P1 will always spin waiting for P0 to wake him up, P3 wait for P2 … The following figure illustrates an example of the algorithm with four processes.

The algorithm is divided into two phases: arrival and wake up. Each phase is itself divided into “rounds” that is levels of tree, and for each round, each process has s static a pre-defined role: “winner” when it is the one which goes up whichever the order of arrival, “loser” when it spins on the static location.

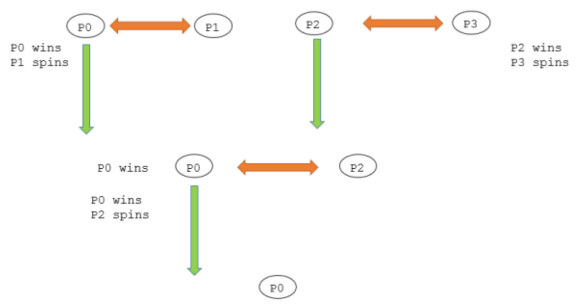


Figure 6 : Arrival phase tournament

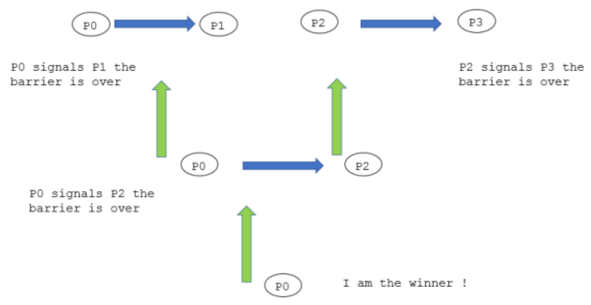

P0
P1
P2
P3
P0 signals P1 the barrier is over
P2 signals P3 the barrier is over
P0
P2
P0 signals P2 the barrier is over
P0
I am the winner !

Figure 7 : Wake up phase tournament

- Arrival phase, if the winner arrives first, he waits for the loser to arrive to go up and the loser will wait for the end of the "tournament" that is when the last round ends at the top of three.
- Wake-up phase, each loser waits for its opponent to signal him that the tournament ended. For each round, each winner signals its opponent. The barrier end when all losers are waken up.

In this example, P0 wins over P1 and P2 over P3. If P2 arrives before P3; P2 is going to wait for P3 to arrive before going up to the next round.

This barrier can either be implemented for shared memory machines using shared data structures or for distributed machines using message passing. This project studies the second option using MPI API. We compute the number of passed message to get a first overview of the complexity of network transactions. Let $\mathrm{h} = \log_2 P$ be the number of rounds. In each arrival round i, there are 2^{h-i} messages and in each wake-up round, there are 2^{i-1}messages which leads $\vartheta(2^i)$ messages. There are h rounds and$\sum_1^h 2^{h-i} = 2^h \cong P$. The number of message transactions is linear with the number of process.

Even though this barrier achieves to overcome the previous challenges of efficient communication without contention and latency, MCS have designed another tree barrier which shows better performance results thanks to a particular tree structure.

2.4 MCS barrier

This barrier is an evolution of the previous barrier where the arrival and wake up have distinct tree. The arrival tree is a four-array tree and the wake up tree is a two-array tree.

- Arrival tree : Each process is associated to a node in the tree and have two data structures associated with it.
    - hasChild Boolean vector of size 4. The ith entry is 1 if the node has an ith child and 0 elseway. This lets the node know how many child he has.

- childVector integer vector of size 4. The ith entry is 1 is the ith child has arrived and 0 else way.

These entries are changed by the node's children during the arrival phase child process: the child has a pointer to its specific entry where it signals its arrival.

- Wake up tree: Each process is also associated with a node and has one data structure.
  - childPointer vector of size two. Each entry points toward the nodes child it will have to wake up (if the process has one).

The algorithm has the following phases detailed for the following example.

- Arrival phase: P1 and P2 wait for their child to arrive by checking their respective NC vector. Each child signals its arrival through its specific data structure in the NC vector. Once all of their child have arrived, P1 and P2 signal that they have all arrived to P0.

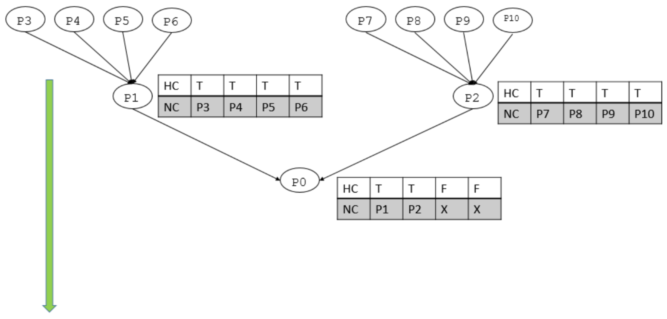


Figure 8: Arrival tree MCS

- Wake up phase: It starts with P0 waking up its two child node and each of this once does the same with its children until there are no more processes to wake up.

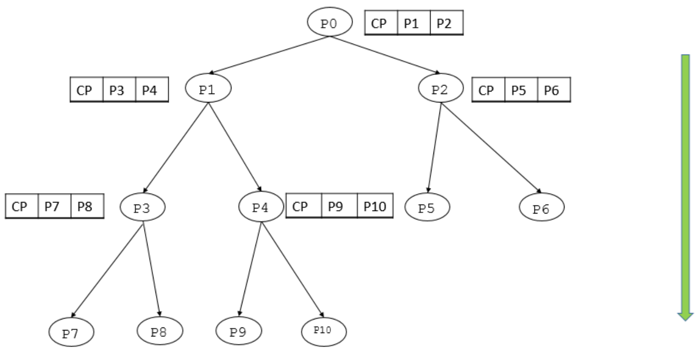


Figure 9: Wake up tree MCS

In a similar as the previous complexity computation, it can be shown that the number of network transaction needed is linear with the number of processor Let $\mathrm{h} = \log_4 P$ be the number of rounds. In each arrival round i, there are 4^{h-i} messages and in each wake-up round, there are 4^{i-1}messages that leads $\vartheta(4^i)$. There are h rounds and$\sum_1^h 4^{h-i} = 4^h \cong P$ message transactions.

3. Performance measurements

The previous barriers were implemented in two different context structure. The centralized and tree sense reverse barriers implementation are meant for shared memory machines where the tournament and MCS barrier are meant for distributed systems. They still can be used by shared memory machines even if this is not their main application. The first two are implemented via OpenMP while the two last are implemented with MPI.

Depending on the barrier, we measure global achievement time, spin contention by counting the spinning and observe the influence of parameters such as the type and value of delay, the number of threads, processes.

3.1 Hardware description

The experiments are run on two specific nodes of the Jinx cluster:

- A four-core node equipped with two 4-core Intel Nehalem processors.
- A six-core node equipped with two 6-core Intel Westmere processors.

These processors ensure that the machine is cache coherent and are designed with particularly wide bandwidth regarding the network transactions and is optimized so as to avoid bottlenecks. Hence, we do not expect to obviously observe performance drawback because of synchronization delay or data structure access contention overhead. That is why for this kind of observation, it is preferred to run experiments on a local machine that will suffer more from the barrier overhead. The local machine has one 2-core processor and is cache coherent.

3.2 Measurement techniques

This part explicits how measurements are taken on each experiment. Two different methods are used for MP barriers and MPI barriers respectively because of the difference of implementation. All data are collected through the program in csv files and then analysed through MATLAB.

3.2.1 MP Barriers

This part describes the goals and methods of the experiments on MP barriers: confirm the importance of contention management and efficient spinning.

Global time achievement: This is measured to get a global view of the barrier performance. It measures the time a group of threads takes to go through the barrier. This is done by setting the number of threads and making them go through the barrier NUM_BARRIERS times. Each thread measures its own achievement time with `omp_get_wtime()`. The test adds these local time and average it over the number of barriers.

Contention time for accessing the count: This evaluates the overhead due to threads sharing the count data structure. This is measured by the difference between the time when the thread arrives to the barrier and the time it signals its arrival by decrementing the count. Once again, the measurement is local to each thread and then summed and averaged.

Synchronization overhead: This measurement may seem useless since it is already known that both machine are cache coherent. Measurement of synchronization time are done by taking the time before and after a critical action. The returned values are all null which means that they are not high enough for the timer to detect them at its number granularity. That is why, this measurement is not taken into account.

Contention for sense spinning: This evaluates how much scheduling is wasted on useless spinning, especially when the spin structure is shared. Since the machines are cache coherent, the experience is made on the number on spinning. The more spinning there is, the more overhead it would create on a non-cache coherent machine. Even though, this overhead does not exist in our particular machines, the experience still illustrates spinning overhead concept.

Dynamically located spinning structure overhead: This experience tries to observe the effects of non-statically allocated data structures. Since the machine are cache coherent, it can not be expected to get speaking results. Still, it records each achievement time of each thread on several successive barriers and measure the variance of this time over the barriers for one thread. A high variance can be interpreted as the fact that the spin location change a lot. In a non -ache coherent machine, this leads some threads to have to spin on remote location which increases the latency.

The spinning overhead and global time achievement are measured for the default barrier and with spin-with-delay barrier.

3.2.2 MPI Barriers

These experiments serve to illustrate the relevancy of the 4-array arrival tree in the MCS barrier by comparing both global time achievement, arrival time and wake-up time for the tournament and MCS barrier.

Global time achievement: This is measured locally for each process by taking the time before going through the NUM_THREADS barriers and after the barrier. It uses the MPI instruction `MPI_Wtime`(). Once all the processors have gone over the barrier, the MASTER process adds up at the local time and average it over the number of the barrier.

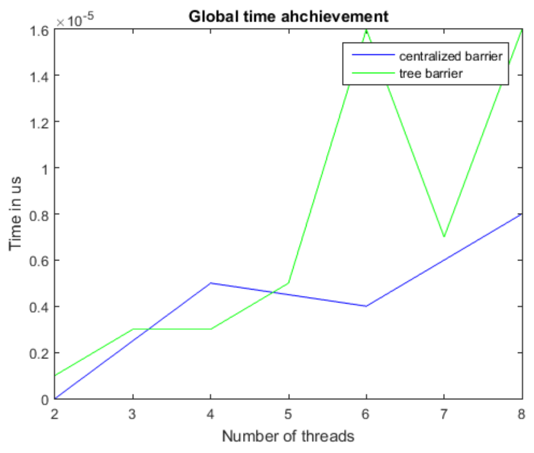
Arrival time and wake up time: This is measured inside by each process over all the barriers. Each time a process go through the barrier, it measures the time at its arrival and the time at which the arrival ends and the time at which the wake up phase ends. The relevant subsraction then give the arrival time and the wake up time. Once all threads have gone over the barrier, the MASTER process adds up all the times and average them over the number of barriers.

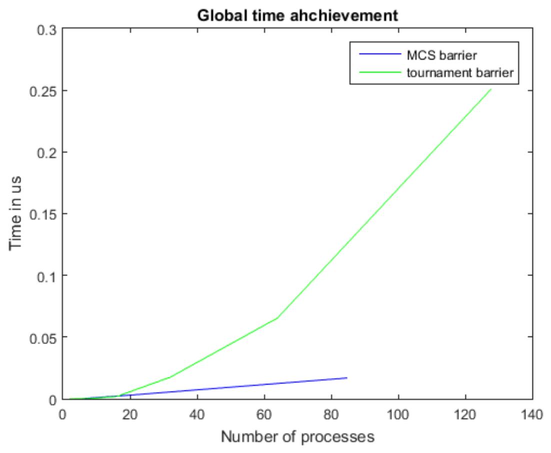
4. Results

This part presents numerical results and analysis of the previously described experiments. For all experience we compare MP barriers between them and MPI barriers between them.

First comparison is the global achievement time of the barriers. We observe that the optimized implementation of the tree barrier is not obviously better that the centralized barrier. This is due to machine structure that compensates for the centralized barrier drawback. This can also come from the implementation of the tree barrier that requires a tree structure. Even though the implementation takes care to precompute every necessary value and store it in an array to reduce the access time, there may be implementation overhead that are explicitly visible and explain this non conclusive graph. It may also be that such samples is not enough to draw a conclusion and that the experiments should run more threads as for MPI barriers.

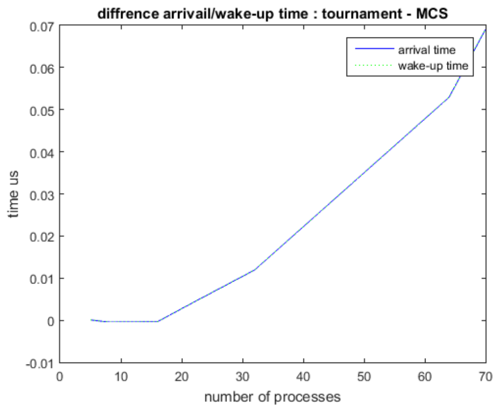
Indeed results are more explicit for MPI barriers. We see the MCS time achievement which seems to be and below the tournament achievement which increase faster than it. This tend to draw the conclusion that the more processes there are, the more outperformed the tournament barrier is compared to the MCS barrier which was to be expected since the arrival tree is optimized.





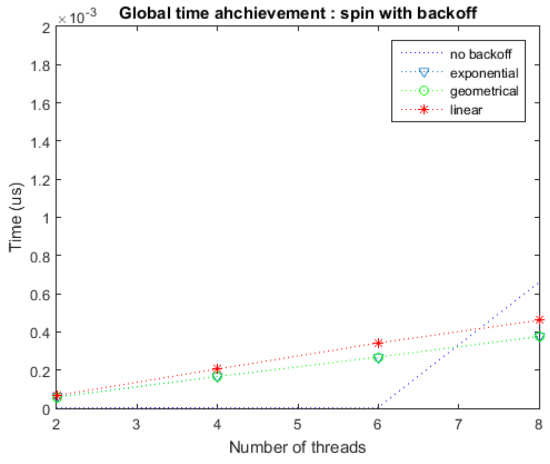
Regarding the centralized barrier measurement on contention arrival time were not conclusive enough to show the drawback of the shared data structure.

More especially for the MPI barriers, the following graph plots the difference between the arrival time and wake-up time between the MCS and the tournament barrier. As expected, of the arrival time difference it is positive because of the 4-aray arrival tree of the MCS barrier. We observe a very similar curve for the wake-up tome difference (that is why we cannot distinguish them in the graph) which was expected since each node wakes up two processes at the time in the MCX wake up tree while a node only wakes up its opponent in the wake-up tree of the tournament barrier.

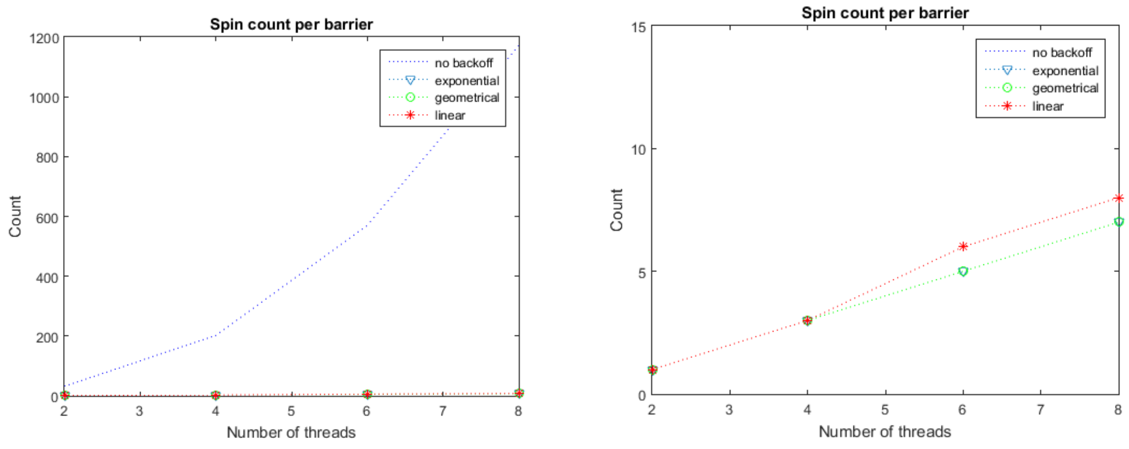


The next experiments show the critical role of the spinning in a barrier and how its setting may completely change the barrier performance. First, the relevant time-spin must be set. Given that the default barrier is achieved in an order of time of microseconds, we set the delay to 10^{-5}ns. Then, the type of delay must be chosen. This is done by observing the effect on global time achievement and spinning contention for each type of delay.

First it is noticed that for small value of threads (2 to 4) the spin adds achievement time rather than making the barrier more efficient. Then in cache coherent machine, spin_with_delay barrier may not be relevant for small-scale synchronization. However, the time difference is not big enough to compensate for the gain obtained when the number of threads exceeds 6. The achievement time seems to grow linearly but with a lower coefficient when the barrier implements a spin with delay. The graph also shows different achievement times for different spin types. According to these first values, the best spin are the exponential and geometrical delay. Still, it is said in the MCS that the best spin is supposed to be the linear one so we may question this result regarding that there are few sampling.



The results are more obvious regarding the spinning efficiency. The left graph plots the number of spinning of all barriers. It shows a high difference between the default barrier spinning and the delay spinning which are far more efficient. The right graph is a zoom on the delay barrier spinning to determine the best spinning strategy regarding the spinning contention.

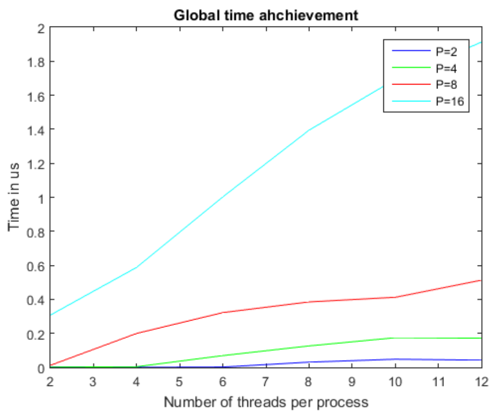


These graph show that best spinning strategy is once again either the exponential or geometrical strategy and once again show that the number of samples may not be enough to get trustful result given that we know that the best spinning strategy in this case is the linear one.

Regarding the observation of the consequences of dynamically allocated spin location, the computation of the variance of achievement time of one thread over several barrier was not conclusive at all since we get a maximum ratio of the variance over the data of 0.0001% which cannot be studied to draw any conclusion.

The experiments also compare the MPI barriers with a combined OpenMP-MPI barrier built from the centralized sense reverse barrier and the tournament barrier. The comparison is made with equal number of process/threads on both barriers.

First we observe the global time achievement of several settings for the combined barriers. The number of processes is in the set {2,4,8,16} and the number of threads in {2,4,6,8,19,12}.



We observe that the order of global achievement time is higher that when the MPI barriers run alone even for the same number of threads/process. For example, the combined barrier has an achievement time of 0.4us for 8 processes times 10 threads that is 80 process/threads whereas the MPI barriers do not exceed 0.15us. We may suspect this drawback to come from the MP barrier drawback since it is the centralized barrier. Hence, it is not that surprising but it shows once again that the centralized barrier is not an efficient tool for synchronization.

5. Conclusion

This project has led to the implementation of two OpenMP and two MPI barriers. Even though the experiments were not conclusive enough regarding the efficiency of set the centralized spin system with a delay, they obviously showed the benefits from the point of view of contention. The OpenMP tree barrier has been implemented as an evolution of the OpenMP centralized barrier which suffers too much unnecessary spin and contention. However the setting of the experiments did not permit to get meaningful time achievement difference between these two implementation.
The MPI experiments were more conclusive. Comparing arrival and achievement time between the MCS and tournament barrier have confirm the theory of the MCS paper stating that the MCS barrier is more efficient and especially its 4-array arrival tree. However, one must be careful with which thread synchronization he uses any of these two process synchronization barrier. As we saw, a bad thread barrier can lead to unnecessary drawback even if the process barrier is the most efficient one.

6. Implementation

This part sums up the algorithm used to run the tests in the particular case of this project. The algorithm are in a large part inspired by the algorithms provided by the MCS paper [1] algorithms. Since they were written for shared memory machines only in this paper, the tournament and MCS barrier are adapted to the distributed system machines. Instead of managing shared memory data structure, process communicate between each other with messages. Once each of this barrier has been implemented, the centralized sense barrier and the MCS are combined to create an OpenMP-MPI barrier for shared memory machines in distributed systems. These implementation may not be the optimum but it still allowed to run experiments and get relevant results.

Sense Centralized Barrier

```
shared count : integer := NUM_THREADS
shared sense : integer := 0
processor private local_sense : integer := 0

procedure SRbarrier

     local_sense := !local_sense
     if fetch_and_decrement(&count) = 1
          count:=NUM_THREQDS
          sense:=local_sense
     else
          repeat until sense = local_sense
```

Sense Tree Barrier

```
Type node =
     id : integer //rank
     k : integer //fanin of the node, in this case k:=2
     count : integer := k
     locksense : integer //initially 1
     parent : ^node //pointer to the parent node

Shared nodes : array [O…P-1] of node //tree
processor private sense : integer := 0
processor private mynode : ^node //specific leaf for process

procedure combining_barrier
     combining_barrier_aux(mynode)
     sense := not sense

procedure combing_aux_barrier(nodepointer: ^node)
     with nodepointer^ do
          if fetch_and_dec(&count) = 1
               if parent!=nil
                    combining_barrier_aux(parent)
               count:=k
```

```
                locksense := !locksense
           repreqt until locksense = sense
```

Tournament Barrier

```
Procedure tournament(my_rank,tree)
     for each round
          If I am a winner
               I wait for the loser’s signal
          If I am a loser
               I signal the winner
               I wait for the winner’s signal
```

MCS Barrier

The tree structure is implemented through nodes. The value for each parent and children are computed before the barrier implementation so that we are sure to measure only barrier actions and not barriers calculations.

- The arrival tree is made of Node1 tree nodes. Each node records the rank of the process and the parent it has to signal.
- The wake up tree is made of Node2 tree nodes. Each node records the rank of the process and its two children in the wake-up tree.

```
Type node2 =
     id : integer //rank
     child : integer vector[2] //vector of children rank in
                               // wake up tree

Type node1 =
     id : integer //rank
     child : integer //parent’s rank

Type level =
     first : integer //rank of the most left node
     last : integer //rank of the most right node +1

Procedure mcs (my_rank, tree)
//arrival phase
     if my_rank is the father of child j
          wait for message from child j
     if my rank is child of i
          send a message to i
          wait for a message from i

//wake up phase
     if I am wakener
          send a message to my two children
```