A Survey of Barrier Algorithms for Coarse Grained Supercomputers

Technical University of Chemnitz

Torsten Hoefler, Torsten Mehlan, Frank Mietke, Wolfgang Rehm
{htor, tome, mief, rehm}@informatik.tu-chemnitz.de

Abstract

There are several different algorithms available to perform a synchronization of multiple processors. Some of them support only shared memory architectures or very fine grained supercomputers. This work gives an overview about all currently known algorithms which are suitable for distributed shared memory architectures and message passing based computer systems (loosely coupled or coarse grained supercomputers). No absolute decision can be made for choosing a barrier algorithm for a machine. Several architectural aspects have to be taken into account. The overview about known barrier algorithms given in this work is mostly targeted to implementors of libraries supporting collective communication (such as MPI).

Contents

1 Introduction **2**
1.1 Related Work . . . 2
1.2 Document Organization . . . 2

2 Algorithms **2**
2.1 Algorithms Performing Phase 3 . . . 2
2.1.1 Central Counter . . . 3
2.1.2 Combining Tree . . . 4
2.1.3 Tournament . . . 7
2.1.4 f-way Tournament . . . 8
2.1.5 MCS . . . 9
2.1.6 BST . . . 10
2.2 Algorithms Omitting Phase 3 . . . 13
2.2.1 Butterfly . . . 13
2.2.2 Pairwise Exchange With Recursive Doubling . . . 14
2.2.3 Dissemination . . . 16

A Appendix **18**
A.1 Pseudocode Semantics . . . 18
A.1.1 General Constructs . . . 18
A.1.2 Conditional Constructs . . . 19
A.1.3 Loops . . . 20

1 Introduction

Several barrier algorithms are currently available. To make a proper decision which to take or to improve for reaching the best performance with InfiniBand™all of them have to be investigated. This short paper is intended to give an overview about all published algorithms. All Algorithms are listed in Chapter 2. Each description is devided into two parts, first the description and second the conclusion. The description gives a slight idea about the working principle for better clarity and proposes a reference to the original papers for further information.

The best way to understand each algorithm is to read the description in combination with the given graphical representation. To gain further knowledge about the algorithms especially on message passing based systems, the reader is encouraged to retrace the proposed pseudo-code.

1.1 Related Work

After careful research only one paper which compares more than two different barrier algorithms for their suitability for a special system was found. This paper [1] is for the thread based shared memory model in Java. Thus it analyzes the behavior of the different algorithms only for the shared memory approach. However, this work expands the comparison regarding to the investigated algorithms and to the more general message passing model. There are also some papers about general barrier techniques for special machine architectures [2].

1.2 Document Organization

This document follows the usual rules for scientific articles. The self-defined pseudo-language which is used to describe the algorithms in detail is explained in the appendix (A.1).

2 Algorithms

This chapter introduces all currently known barrier algorithms. Each algorithm can be split up logically into three phases. The algorithm is initializated in phase 1 (e.g. reserving shared objects or calculating ranks). So it has to be done only once during initialization or reconfiguration (processors enter or leave) of each communicator. Phase 2, also called "Check-in-Phase" has to be done on each node every time when it calls MPI_Barrier. All nodes communicate with each other until one or all nodes know that every node reached it's MPI_Barrier call. A barrier-identifier is often used to distinct between different MPI_Barrier calls to avoid race conditions when one processor enters the next barrier before all other processors left the last barrier - this is called x in the following chapter. Each barrier number is used once per communicator and incremented for each barrier starting initially with 1. The third and last phase can be referred as "Notification-Phase" and is only needed when not all processors know that the barrier is reached by all other processors. The typical case is that one processor knows the barrier is reached by all and it has to notify all remaining processors. The difference in phase 3 leads to a distinction between two types of algorithms. The first type performs phase 3 as described above (see Section 2.1 on page 2) and the second type omittes it completely (see Section 2.2 on page 13).

2.1 Algorithms Performing Phase 3

Phase 3, as described in Section 2 on page 2 can efficiently be implemented as a broadcast (e.g. MPI_Bcast). This operation could especially benefit from hardware broad- or multicast capabilities which perform (ideally) in $O(1)$. If this is not capable with the underlying architecture[1],

[1] regardless if it's provided by hardware or software

standard bcast algorithms could be used, which usually scale with $O(log_2(n))$ for 1 byte messages. The time which is necessary to perform a broadcast from one to n nodes is modelled as $t_{bc}(n)$ regardless of the implementation and architectural details mentioned above.

2.1.1 Central Counter

2.1.1.1 Description

This algorithm is quite simple and straightforward. But because of it's obvious simplicity and the naive prove for correctness it is implemented quite often. Especially the atomic "fetch-and-Φ"[2] Operation is frequently mentioned related to this barrier. This approach is investigated for the fetch-and-increment[3]operation in [4] and [5]. One node holds an integer value which is used as central barrier counter. This integer starts with 0 and is increased by each node once (after it entered the barrier) until the node count p is reached. The last node sends a message to all other nodes to wake them up.

This barrier consists of the two parts counting and notification. Both parts can be optimized independently. Optimized algorithms for counting and broadcasting a message are evaluated later. We assume the easiest case in the following pseudo-code (see listing 1) and graphical representation (see figure 1).

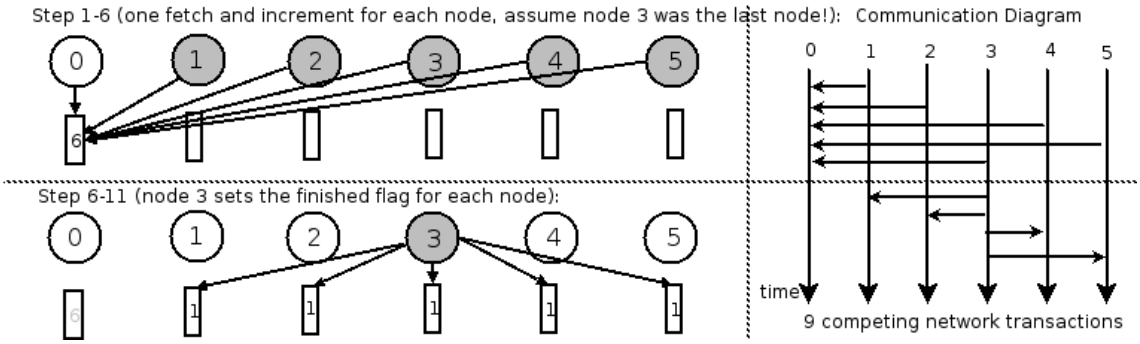

Figure 1: a central counter barrier between 6 nodes

2.1.1.2 Conclusion

As the algorithm splits up into two phases for each MPI_Barrier call, each phase is analyzed apart. Phase one is critical, because the shared counter is altered by each node. This memory location is called a hot-spot (see [3]). $O(p-1)$ competing network transfers are needed to implement the counter. These operations have to be atomic on the target to prevent lost-update problems, resulting in deadlocks. Phase two is also critical, because one node has to inform all other nodes. Regarding to 2.1, the possibilities to perform this broadcast are not mentioned here. Thus, the overall amount of competing network operations can be seen as $(O(p-1+t_{bc}(n-1))$. The memory usage per node is constant with $O(1)$ byte per node.

[2] "fetch-and-Φ" is a conceptional term for a collection of atomic operations which change and return a single value in memory - e.g. fetch-and-add, fetch-and-swap, fetch-and-inc, ...

[3] the fetch-and-increment operation takes a value to increment from it's caller, increments it's memory value and returns the new value to the caller (some implementations may return the value before incrementing)

Listing 1: Central Counter in Pseudo-Code

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
if rank == 0 then
   // it's my counter
   reserve ctr with 1 entries as shared
   set ctr = 1
else
   reserve flag with 1 entries as shared
   set flag = 0
ifend

// phase 2/3 - central barrier
set x = x + 1;
if rank == 0 then
   wait until ctr == p
else
   set localctr = fetch and increment ctr on node 0
   if localctr == p then
      set flag in all nodes to x
   ifend

   wait until flag >= x
ifend
```

2.1.2 Combining Tree

2.1.2.1 Description

The combining tree barrier was introduced by Yew, Tzeng and Lawrie in [7]. It uses a tree to speed up the central counter barrier. It divides the nodes into subgroups with n members, which synchronize among each other with a simple shared counter. Every first node of each group spins[4] its local counter which is shared to all others until all nodes reach the barrier ($counter == n$). When all nodes in the subgroup reached the barrier, all first nodes form a new group and synchronize among each other. This is repeated until only one group is left and has finished the synchronization. The first node informs all other nodes about the barrier end. Yew reported a group-count (n) of 4 to achieve the best results. A graphical example as well as pseudocode for this algorithm can be found in figure 2 and listing 2.

2.1.2.2 Conclusion

The combining tree barrier reduces hot spots in memory and network contention. The number of required steps is naively seen lowered to $O(log_n(p) + t_{bc}(n))$[5] (the more correct equation will be given after modelling the target InfiniBand™network). $O(2)$ byte memory is used per node.

[4] check the counter frequently

[5] this equation is only valid for a fan-out of n - e.g. in a mesh topology, it has to be seen as a naive approximation for all other cases

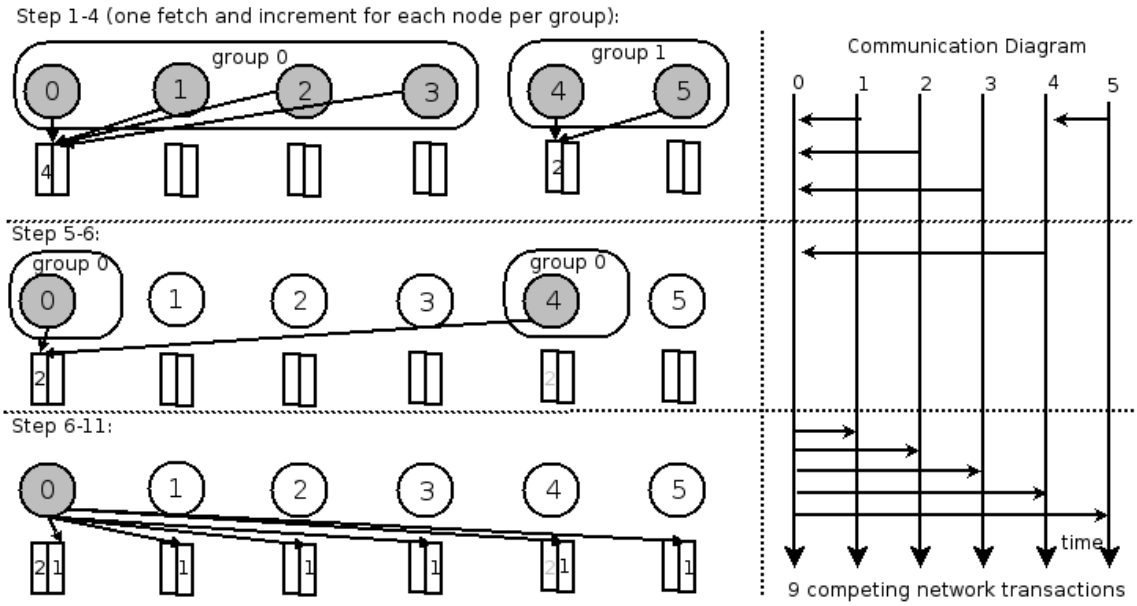

Figure 2: a combining tree barrier between 6 nodes

Listing 2: Pseudo Code for Combining Tree Algorithm

```
set p = number of participating processors
set n = nodes per group // parameter
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter

reserve ctr with 1 entries as shared
set ctr = 1
reserve flag with 1 entries as shared
set flag = 0

set round = 0 // actual round
set relnodeid = 0 // relative nodeid (only active nodes)

// phase 2 - barrier
set x = x + 1;
repeat
   set round = round + 1
   set relnodeid = rank / (n^(round-1))
   set grpnum = relnodeid div n // group number?
   set grprank = relnodeid mod n // my rank in group

   // I am out of the game, when I have no natural number as relnodeid
   if round(relnodeid) != relnodeid then
      wait until flag >= x
   ifend

   if grprank == 0 then
      wait until ctr == n
   else
      set ctr = fetch and increment ctr on node rank-grprank*n^(round-1)
      wait until flag >= x
   ifend
until round == log(n)(p) or flag >= x

// phase 3
if rank == 0 then
   set flag in all other nodes to x
ifend
```

2.1.3 Tournament

2.1.3.1 Description

The Tournament Barrier, proposed by Hengsen et al. in [9] is mostly suitable for shared memory multiprocessors because it benefits from several caching mechanisms. Nevertheless, the algorithm is analyzed here. As in the Butterfly (see chapter 2.2.1) and the Dissemination Barrier (see chapter 2.2.3), different rounds (s) are used. The algorithm is similar to a tournament game. Two nodes play in each round against each other. The winner is known in advance and waits until the looser arrives. The winners play against each other in the next round. The overall winner (the champion) notifies all others about the end of the barrier. A graphical and pseudo-code representation can be found in figure 3 and listing 3.

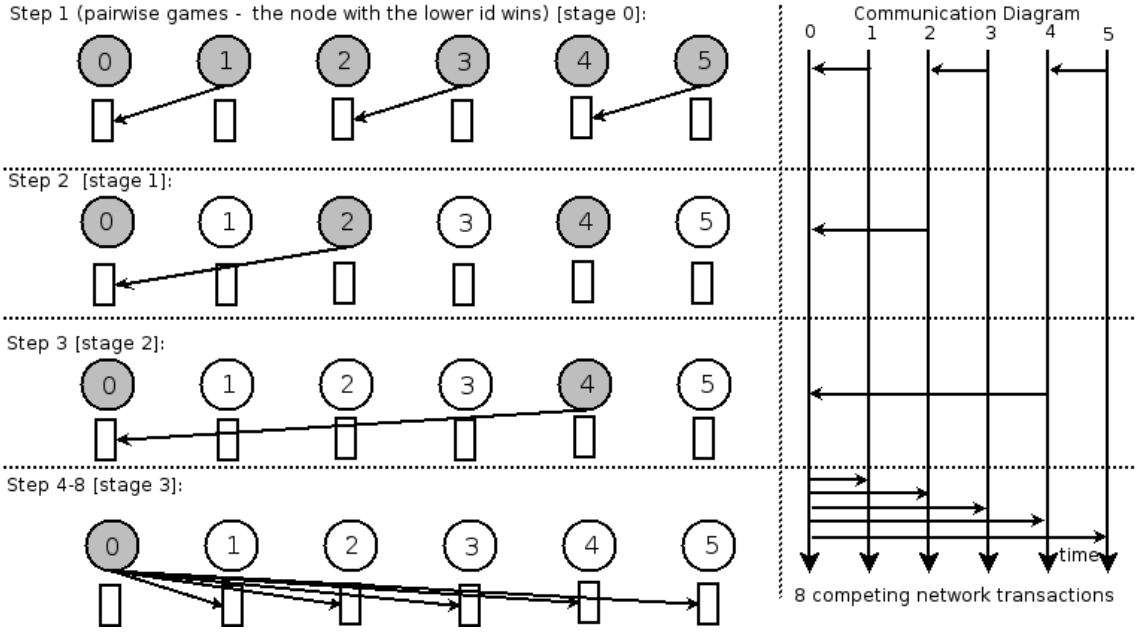

Figure 3: example for the tournament barrier with 6 nodes

2.1.3.2 Conclusion

The algorithm is also subdivided into two parts. Part one (the game) scales with $log_2(p)$ and uses $O(1)$ byte of memory. Part two scales as mentioned in chapter 2.1 with $t_{bc}(n-1)$. Thus the entire complexity can be estimated with $O(log_2(p) + t_{bc}(n-1))$.

Listing 3: Pseudo Code for Tournament Barrier

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
reserve flag with 1 entries as shared
set flag = 0

// phase 2 - done for every barrier
set true = 1
set false = 0
set round = -1
// repeat log(p) times
repeat
   set round = round + 1
   set peer = rank xor 2^round

   // I have no partner -> next round ...
   if peer > p then
      continue
   ifend

   // I am the winner
   if rank > peer then
      wait until flag == true
      set flag = false
   else
      set flag on peer = true
      wait until flag == true
   ifend
until round > ld(p)

// phase 3 - node 0 ever wins
if rank == 0 then
   set flag in all other nodes to true
ifend
```

2.1.4 f-way Tournament

2.1.4.1 Description

The f-way Tournament Barrier bases on the same principle as the tournament barrier (Section 2.1.3). It was proposed by Grunwald et al. in 1993 [13]. The most important difference is that more than two processors are competing in one game. A graphical representation can be found in figure 4. The pseudo-code is nearly identical to the tournament barrier (see listing 3), only with more than two nodes.

2.1.4.2 Conclusion

This Barrier is suitable for special network topologies with a fan-out of more than one (e.g. torus networks). But should not scale better on standard central switching-based networks. The algorithm scales theoretically (with a fan-out of f in each node) with $log_f(p)$ network transactions and $O(1)$ bytes memory per node, but should be practically limited by the network infrastructure which serializes and enqueues concurrent requests.

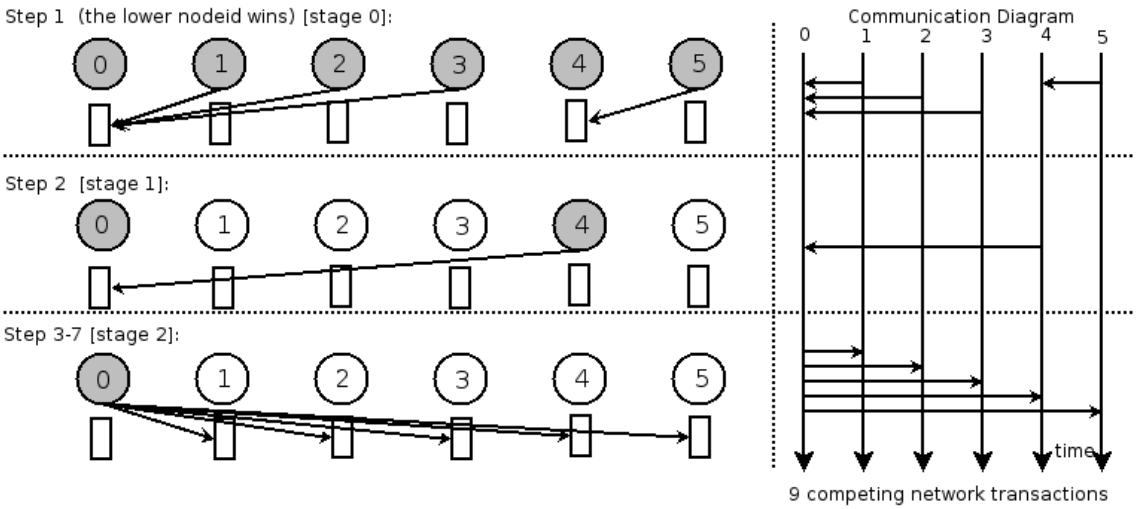

Figure 4: example for the f-way tournament algorithm between 6 nodes

2.1.5 MCS

2.1.5.1 Description

The MCS Tree Barrier was proposed by Mellor-Crummey and Scott in 1991 ([10], [11] and [12]). It uses also a tree structure and is quite similar to the Combining Tree barrier (Chapter 2.1.2). Each node is assigned to a tree node. The resulting n-ary tree consists of all nodes, each node has an array of n flags. All, but the top node write to their parent's node flag when all child nodes wrote the flag to them. All nodes, which have no children start with the array initialized with true. When the last node's flag array is completely filled, the last node notifies the others.

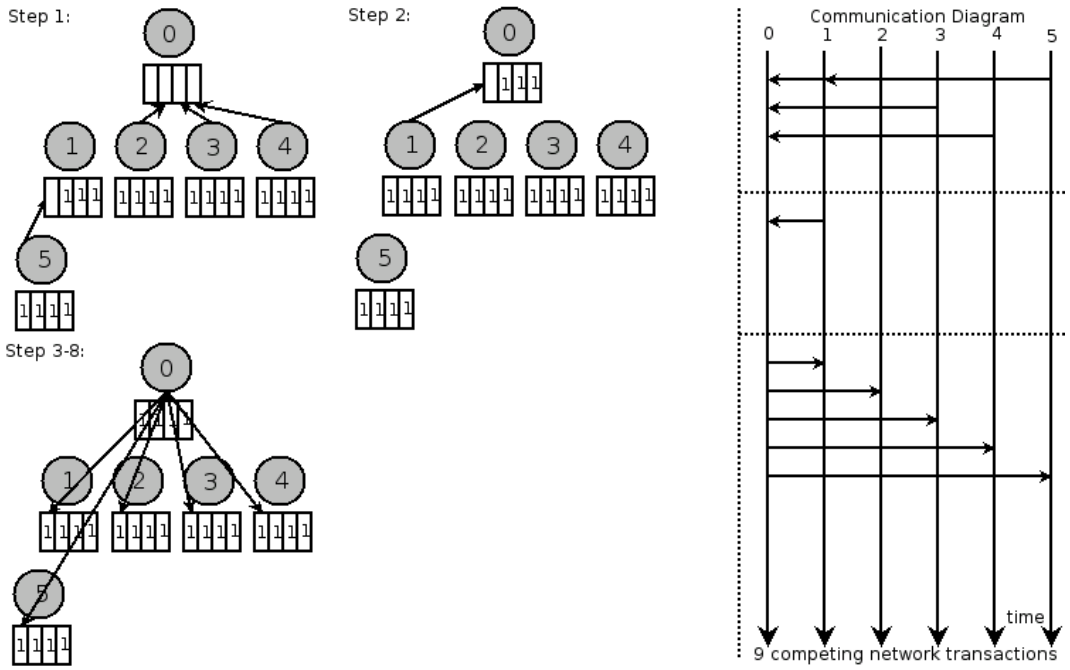

Figure 5: example of the MCS Tree algorithm between 6 nodes

Listing 4: Pseudo Code for the MCS Barrier

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id
set n = number of childnodes

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
reserve array with n+1 entries as shared
  // -> array[n] acts as barrier_reached flag

// phase 2 - done for every barrier
set x = x + 1

// initialize my flags (flag == 1 if no child is present)
for j in 0..n-1 do
   if p >= (rank * n) + 1 + j then
      set array[j] = 0
   else
      set array[j] = 1
   ifend
forend

set array[n] = 0
repeat
   set parent = (rank-1) div n
   set slot = (rank-1) mod n

   if sum(array[0..n-1]) == 4 then
      if rank == 0 then
         set array[n] = 1
      else
         set array[slot] in parent to 1
      endif
   endif
until array[n] == 1

// phase 3
if rank == 0 then
   set array[n] in all nodes to 1
ifend
```

2.1.5.2 Conclusion

The MCS-Barrier uses a tree structure with a fan-out of n to improve the barrier performance to $O(log_n(p))$ concurrent network transactions (only if the network offers a fan-out of n) and $O(p)$ bytes of shared memory per node in the first part. The second notification part depends as usual on the underlying network architecture and scales with $t_{bc}(p-1)$ competing network transactions.

2.1.6 BST

2.1.6.1 Description

The Binomial Spanning Tree (BST) Barrier was proposed by Tzeng et al. in 1997 - [14]. It uses

a binomial tree structure which reduces the network contention by its principle. The working principle is quite similar to the MCS Barrier (2.1.5) - every processor is assigned to one tree-node and waits until all children reached their barrier (they notify their parent) and then notifies its own parent. A binomial tree is built up recursively, the whole tree of step $j-1$ is appended to the root node in step j. The principle is shown in figure 6.

This special characteristic is used to avoid contention on single nodes.[6] To manage the processor-to-tree-node assignment, the following numbering scheme is used:

- each node is numbered in binary digits (from 0 to $p-1$)
- each node calculates it's parent by resetting the leftmost "1" in it's own id to "0"
- each node calculates it's children by adding 2^i to it's own id where $i = \{i \in N \wedge log_2(id) < i < \lceil log_2(p) \rceil \wedge id + 2^i < p\}$

A numbered binomial tree with 6 nodes is shown in figure 7. Pseudo code for the algorithm can be found in listing 5.

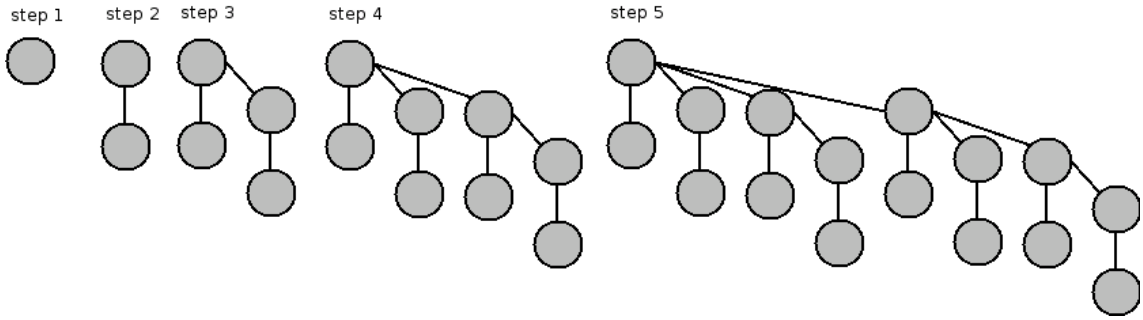

Figure 6: example for building a binomial tree

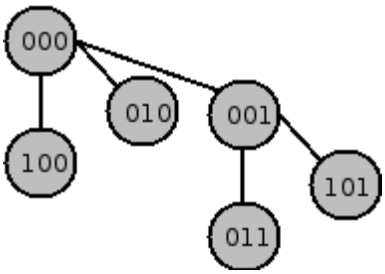

Figure 7: a numbered binomial tree with 6 nodes (each processor is assigned to one tree node)

2.1.6.2 Conclusion

The binomial spanning tree barrier minimizes the concurrency at the root node. One child of the root node finishes each round. The root node has typically $\lceil log_2(p) \rceil$ children, so that the root node knows after $\lceil log_2(p) \rceil$ steps that all nodes reached the barrier. So the time for check in scales with $O(\lceil log_2(p) \rceil)$ The notification of all nodes scales with $t_{bc}(p-1)$. The memory required scales with $O(log_2(p))$ bytes.

[6] due to the distribution of nodes in a binomial spanning tree, each network link is utilized at most once per round if p is a power of two - for all other node-counts, each link is utilized at most twice

Listing 5: Pseudo Code for BST Barrier

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1: initialisation
set x = 0 // the barrier counter
reserve array with p entries as shared // could be shortened to ld(p)

// set all array entries to '1'
for j in 0..p-1 do
   set array[j] = 1
forend

// determine parent (reset leftmost '1')
set j = 1
while j <= rank do
   set j = j * 2
whileend

set parent = rank - j/2

// determine children - unset their array entries
for j=0..ceil(ld(p))-1 do
   // ld(0) is not defined ... take all entries for root node
   if rank == 0 or j > ld(rank) then
      set k = rank + 2^j
      // only for rank + 2^j < p
      if k < p then
         array[k] = 0
      ifend
   ifend
forend

// phase 2: check in phase
// wait until all children reached their barrier
for j in 0..p-1 do
   wait until array[rank] == 1
forend

if rank != 0 then
   set array[rank] in node parent to 1
ifend

// phase 3: release phase
// use array[0] as finished indicator, because node 0 is the root -
// nobody has it as child node
if rank == 0 then
   set array[0] in all nodes 0;
else
   wait until array[0] == 0
ifend
```

2.2 Algorithms Omitting Phase 3

2.2.1 Butterfly

2.2.1.1 Description

The Butterfly Barrier was proposed by Brooks in 1986 [8]. The original algorithm uses a single shared array of flags (shared memory) and performs several stages of pairwise synchronization. The used algorithm can be described easily in the following way:

1. wait until previous stages finished (until my flag is false)
2. set my flag to true (I am currently synchronizing)
3. wait for the partner's flag to become true (the partner is ready)
4. set the partners flag to false (done)

After the initial synchronization finished the whole process is repeated $w = log_2(p)$ times, each time is called a stage. The stages (s) are numbered ascending, the very first stage starts with 0. Each node p_i synchronizes in each stage with node p_j where $j = i\ XOR\ 2^s$ (see figure 8). This method only works for $p = 2^x;\ x \in N$ (p = power of two). For all other number of nodes, the necessary pairs are represented virtually by the other nodes (e.g. to synchronize 6 nodes, 2 additional virtual nodes are necessary). Thus this algorithm performing worst with any number of nodes, slightly bigger than a power of two.

The array mentioned above has to have the dimensions $p * log_2(p)$. One column per processor and one row for each round.

This implementation does not scale very well on a message passing based system (because of the shared array). After applying all modifications to ensure scalable operation on message passing based systems, the algorithm looks very similar to the Pairwise Exchange 2.2.2. Thus, this work does not propose a pseude code.

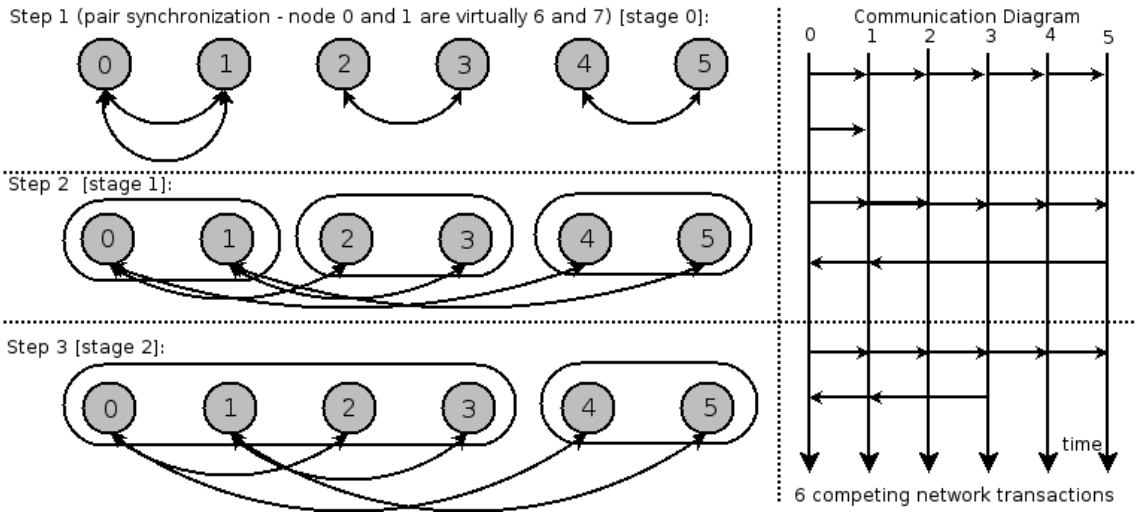

Figure 8: the butterfly algorithm - the shared array was left out due to the clearness

2.2.1.2 Conclusion

The barrier's competing network operations scale best with processor numbers which are a power of two with $O(log_2(p))$. The worst case is when the processor number is slightly higher than a power of two with $O(2 * log_2(p))$ because half of the processors must synchronize twice. The used shared array of flags memory scales with $O(p * log_2(p))$ in size. Due to the above mentioned problems, the Pairwise Exchange Barrier (Chapter 2.2.2) should be implemented in message passing based systems.

2.2.2 Pairwise Exchange With Recursive Doubling

2.2.2.1 Description

This algorithm was proposed in [6] and will be discussed in the following section.

The first part of the pairwise exchange algorithm is, that all nodes group theirselfs in pairs (node 0 and node 1 for each pair). The barrier-identifier, described in chapter 2 is used to avoid several race conditions.

In the first part, all nodes write their value of x to the corresponding peer. When a node 0 and node 1 of each pair received the correct barrier value[7] from the peer they continue and enter the next stage. Each group peers with another group of two processors and each member of the group writes the barrier number to it's corresponding peer in the other group. This procedure is recursively repeated until all nodes form one big group. So this algorithm uses $\lfloor log_2 p \rfloor$ network write operations per node.

Thus this works only for power of two nodes. For all other node counts p, the biggest power of two $y = 2^z$ is calculated which is smaller than p. This creates two groups, the group with y nodes (group a) and the remaining nodes (group b). Every single node in group b pairs with another node in group a. When a node of group b reaches the barrier it writes the barrier number to it's peer node in the group a. Each of this nodes in group a waits until it receives the barrier number from the second's group partner before it starts the normal pairwise exchange algorithm. When the barrier is finished, each peer node in group a notifies its partner that the barrier is finished. This extension for non power of two node counts increases the latency to $\lfloor log_2 p \rfloor + 2$ network write operations.

Figure 9 gives a graphical explanation for a barrier with 6 nodes. After step 4, node 0 has all necessary information (that all nodes entered the barrier already) - node 1,2 and 4 communicated directly with node 0 and the other nodes finished before node 1,2 or 4.

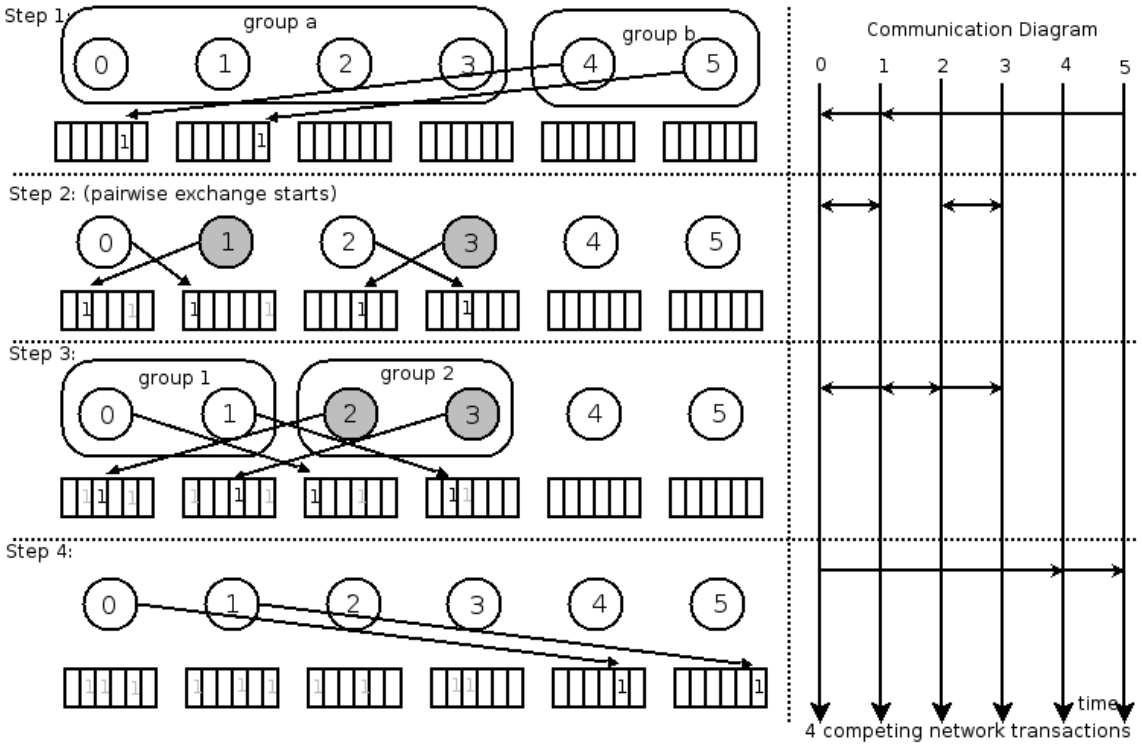

Figure 9: example for the pairwise exchange algorithm between 6 nodes

[7] the currently active barrier number or each number higher than this

Listing 6: Pseudocode for the pairwise exchange barrier

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
reserve array with p entries as shared
for i in 0..p-1 do
   set array[i] = 0
forend
set x = 0 // the barrier counter
y = 2^floor(ld(p)) // the 2^z count

// barrier - done for every barrier
set x = x + 1
if rank >= y then
   // I am in group b, my partner is node i-y in group a
   set array[rank] in node rank-y to x
   // wait for notificatin from partner
   wait until array[rank] >= x
else
   // I am in group a
   if p-y > rank then
      // I have a partner in group b
      // wait for partner
      wait until array[rank+y] >= x
   ifend

   // the pairwise exchange algorithm
   set round = -1

   // repeat log(p) times
   repeat
      set round = round + 1

      set peer = rank XOR 2^round

      set array[rank] in node peer to x
      wait until array[peer] >= x
   until round == log(y)

   if p-y > rank then
      // I have a partner in group a
      // notify partner
      set arr[rank+y] in node rank+y to x
   ifend
ifend
```

2.2.2.2 Conclusion

The algorithms uses $O(\lfloor log_2 p \rfloor + 2)$ network writes and $O(p)$ bytes memory per node. It can be used to exploit the advantages of an RDMA architecture efficiently.

2.2.3 Dissemination

2.2.3.1 Description

The Dissemination Barrier, introduced by Hengsen, Finkel and Manber in 1988 [9], is mostly an improvement of the Butterfly Barrier for non power of two processor counts. It uses the same pairwise synchronization but with other partners. In each round s each processor p_i synchronizes with p_j where $j = i + 2^s \ mod \ p$. Each processor is waiting for the cyclically next to set its flag and for his own flag set by a circular previous processor. The algorithm is the same as used in the butterfly barrier but with different partners.

The implementation with a central shared array does not scale very well on a message passing based system. Thus this work proposes a more suitable solution for message passing systems.

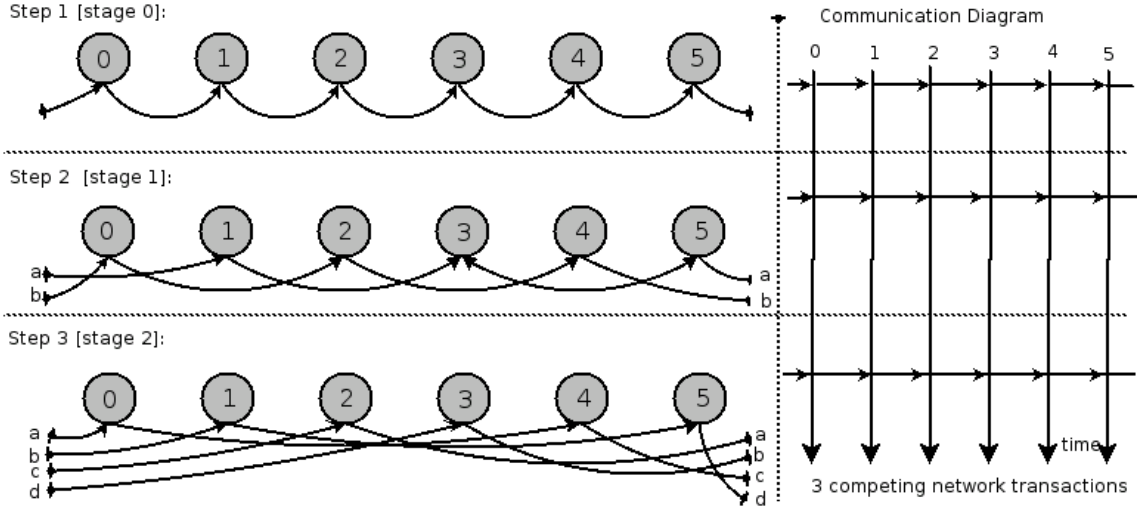

Figure 10: Dissemination Barrier with 6 processors

2.2.3.2 Conclusion

The Dissemination Barrier scales better as the butterfly barrier also for non power of two processor counts with $O(\lceil log_2(p) \rceil)$ competing network transactions. The algorithm uses $O(p)$ bytes of memory per node.

Listing 7: Pseudocode for the Dissemination Barrier

```
// parameters (given by environment)
set p = number of participating processors
set rank = my local id

// phase 1 - initialization (only once)
set x = 0 // the barrier counter
reserve array with p entries as shared
for i in 0..p-1 do
   set array[i] = 0
forend

// barrier - done for every barrier
set round = -1
set x = x + 1
// repeat log(p) times
repeat
   set round = round + 1

   set sendpeer = rank + 2^round mod p
   set recvpeer = rank - 2^round mod p

   set array[rank] in node sendpeer to x
   wait until array[recvpeer] >= x
until round >= log(p)-1
```

References

[1] CARWYN BALL, MARK BULL: *Barrier Synchronization in Java*

[2] ANJA FELDMANN, THOMAS GROSS, DAVID OHÀLLARON, THOMAS M. STRICKER: *Subset Barrier Synchronization on a Private-Memory Parallel System*

[3] G.F. PFISTER, V.A. NORTON: *"Hot Spot" contention and combining in multistage interconnection networks*

[4] ERIC FREUDENTHAL, ALLAN GOTTLIEB: *Process Coordination with Fetch-and-Increment*

[5] JAMES R. GOODMAN, MARY K. VERNON, PHILIP J. WOEST: *Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors*

[6] RINKA GUPTA, VINOD TIPPARAJU, JARE NIEPLOCHA, DHABALESWAR PANDA: *Efficient Barrier using Remote Memory Operations on VIA-Based Clusters.*

[7] P.C. YEW, N.F. TZENG, D.H. LAWRIE: *Distributing Hot Spot Addressing in Large Scale Multiprocessors*

[8] EUGENE D. BROOKS: *The Butterfly Barrier*

[9] DEBRA HENGSEN, RAPHAEL FINKEL, UDI MANBER: *Two Algorithms for Barrier Synchronization*

[10] JOHN MELLOR-CRUMMEY, MICHAEL SCOTT: *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*

[11] JOHN MELLOR-CRUMMEY, MICHAEL SCOTT: *Synchronization Without Contention*

[12] JOHN MELLOR-CRUMMEY, MICHAEL SCOTT: *Fast, Contention-Free Combining Tree Barriers for Shared Memory Multiprocessors*

[13] DIRK GRUNWALD, SUVAS VAJRACHARYA: *Efficient Barriers for Distributed Shared Memory Computers*

[14] NIAN-FENG TZENG, ANGKUL KONGMUNVATTANA: *Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer*

A Appendix

A.1 Pseudocode Semantics

The semantics for each language construct used in the pseudocode sections is described in the following. Some operations are specially designed for message passing systems.

A.1.1 General Constructs

A.1.1.1 Instruction Blocks

Syntax:

```
<instructions>
```

An instruction block consists of one or more instructions, each instruction can be one of the successional described commands.

A.1.1.2 Comparative Instructions

Syntax:

```
<var1> == <var2>
<var1> >= <var2>
<var1> <= <var2>
<var1> != <var2>
```

A comparative instruction compares two or more (logical combined) variables. The used comparison signs (==, >=, <=, ! =) have the same meaning as the according sign the well known programming language Pascal. Each condition returns true or false.

A.1.1.3 The Floor Function

Syntax:

```
floor(var)
```

Rounds var down to the nearest integer.

A.1.1.4 The Ceil Function

Syntax:

```
ceil(var)
```

Rounds var up to the nearest integer.

A.1.1.5 Set Constructs

Syntax:

```
set <var1> = <var2>
```

Registers variable var1 (if this is not already done) and sets it to var2. Var2 can also be an equation.

A.1.1.6 Set Remote Constructs

Syntax:

```
set <var1> in node <nodeid> to <value>
```

Sets variable var1 in the given node to a new value.

A.1.1.7 Broadcast Set Constructs

Syntax:

```
set <var1> in all nodes to <value>
```

Sets variable var1 all node to the given value. The most suitable implementation would be a broadcast.

A.1.1.8 Register Shared Variables

Syntax:

```
reserve <var> with <amount> entries as shared
```

To access a local variable from another node, this has to be registered as shared. The reserve command shares the named array with the given count of entries to all other nodes.

A.1.1.9 Fetch And Add

Syntax:

```
set <lvar> = fetch and increment <rvar> on node <nodeid>
```

Returns the result of a fetch-and-add operation on variable rvar in the given node into the local variable lvar. The operation is system-wide atomic - no other node can interrupt it. It is assumed that the target node returns the value of its local variable after modifying it.

A.1.1.10 Logarithmic Expressions

Syntax:

```
log(n)(x)
ld(x)
```

The first expression means $log_n(x)$, the second $log_2(x)$.

A.1.2 Conditional Constructs

A.1.2.1 If

Syntax:

```
if <condition> then
   <instructions>
else
   <instructions>
ifend
```

The if clause is used to test the condition for its boolean result. It executes the first instruction block if the condition returns true and the second instruction block in the other case.

A.1.2.2 Wait

Syntax:

```
wait until <condition>
```

The wait clause is not passed by the instruction pointer until the condition returns true.

A.1.3 Loops

A.1.3.1 For

Syntax:

```
for <var> in <range> do
   <instructions>
forend
```

The for loop counts the variable in the given range (e.g. 0..10 means 0 up to 10) and executes the instructions for each single value of the variable. The variable is accessible inside the for loop with its current value.

A.1.3.2 Repeat

Syntax:

```
repeat
   <instructions>
until <condition>
```

Repeat keeps executing all instructions until the condition returns true.

A.1.3.3 While

Syntax:

```
while <condition> do
   <instructions>
whileend
```

The instructions are executed until the condition returns false.