Barrier Synchronization Pattern

Rajesh K. Karmani * rkumar8@illinois.edu
Nicholas Chen * nchen@illinois.edu
Bor-Yiing Su ** subrian@eecs.berkeley.edu
Amin Shali * shali1@illinois.edu
Ralph Johnson * johnson@cs.uiuc.edu

* Computer Science Department
University of Illinois at Urbana-Champaign
** EECS Department
University of California, Berkeley

May 11, 2009

1 Problem

How does one synchronize concurrent UEs which are mutually dependent on each other across phases of a computation?

2 Context

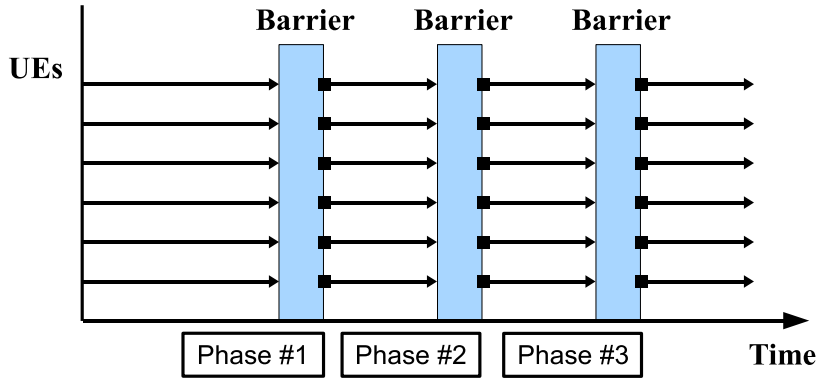
Parallel algorithms divide the work into multiple, concurrent tasks. These tasks or UEs may execute in parallel depending on the physical resources available. It is common for UEs to proceed in phases where the next phase cannot start until all UEs complete the previous phase. This is typically due to mutual dependency on the data written during the previous phase by concurrent UEs. Since UEs may execute at different speeds, there is a need for UEs to wait for one another before proceeding to the next phase.

Barriers are commonly used to enforce such waiting. Figure 1 illustrates how a barrier works. A UE executes its code until it reaches a barrier. Then it waits until all other UEs have reached that barrier before proceeding.

Consider the Barnes-Hut [BH86] N-body simulation algorithm. This is an iterative algorithm with well-defined phases: building the octree, calculating the forces between bodies, updating the positions and velocities of each body. One way to parallelize the algorithm is to have multiple UEs perform the three different phases. However, no UE can proceed to the next phase until all UEs complete executing the previous phase. After all, it does not make sense to update the position when some UEs are still calculating the forces between bodies. A barrier where **all** UEs wait for each other to reach the barrier before continuing with their respective computation, is called a *global barrier*.

We distinguish a *global barrier* from another kind of barrier called *local barrier*, where a **parent** task waits for all the **child** tasks to finish before it can continue.

Figure 1 Conceptually a barrier synchronizes all UEs due to mutually dependencies across phases of a computation



Consider the quicksort divide-and-conquer algorithm. The parent task divides the array into two and spawns child tasks to sort each half of the array. The child tasks subdivide their respective arrays into two halves and hand each half to their own child tasks and so on. A parent task must *wait* for both its child tasks to complete *before* continuing. Recursively, this argument applies to all the child tasks in the computation tree except the leaves.

Conceptually, after spawning the child tasks, the parent tasks enters a *barrier* where it waits for all the child tasks to finish before it can use the array.

Both kinds of barriers can be implemented using the other. In practice though, some problems like divide-and-conquer algorithms are naturally expressed using a local barrier, while many algorithms in scientific computing are candidates for global barriers. We provide examples and discuss usage implications in Section 6 to clarify the distinction.

A variant of the *barrier* is an implicit barrier. An implicit barrier is typically used to synchronize UEs at the end of a code block or parallel `for` loop.

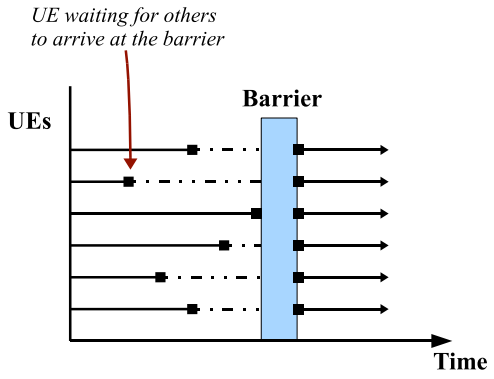
Implementing barrier synchronization can be quite complex, and may prove to be a performance bottleneck if the programmer is not careful [HS98]. Anyhow, a barrier is an expensive synchronization mechanism since the semantics of barrier require the computation to wait for the *slowest* UE to arrive before the rest can proceed. Figure 2 shows how barrier synchronization can cause performance degradation by making all UEs to wait for the slowest UE. When performance is a major concern, use barriers judiciously [Tse95, SMS96].

3 Forces

Dilemma of abstraction For some programs, barrier synchronization is stronger than the synchronization needed for correct execution of the program. The programmer may still be tempted to use a barrier abstraction due to its availability in the parallel programming environment and the resulting succinctness. In such scenarios, relaxed barrier synchronization schemes such as split barrier or custom synchronization schemes such as neighborhood synchronization may perform better.

Locality Locality of reference may be exploited across calls to a Barrier. This is commonly the case when barriers are used to synchronize UEs between different parallel iterations of

Figure 2 Barrier makes the fastest UE to wait for the slowest UE before it can proceed



a computation (See Section 6.2 for details). The placement and scheduling policy of the parallel development environment needs to be carefully reviewed in order to avoid performance degradation due to poor locality.

Know thy environment Parallel programming environments like OpenMP and Cilk provide implicit barrier semantics at the end of code blocks, parallel `for` loops etc. Due to the two forces discussed above, it is very important for the developers to be aware of any implicit barriers in their code.

4 Solution

Use the barrier abstraction if barrier synchronization is unavoidable (for program correctness or program maintenance or shortage of time) and one is provided by the parallel programming environment. Barrier synchronization is such a common pattern in parallel and concurrent programming that it is available as an abstraction in almost all parallel programming environments. Table 1 lists the ways in which barrier abstraction can be expressed in different environments.

Although the underlying architecture and the run-time placement of the UEs are important factors in the efficiency of barrier synchronization, the implementation of Barrier abstraction that is available in a programming environment can be considered to perform well for many programs.

For some programs though, a barrier abstraction may succinctly express the intent of the programmer but it could be a performance bottleneck. See Section 4.1 for a discussion on implementing barrier synchronization. Relaxed barrier synchronization schemes such as split barrier, topological barrier, or custom synchronization schemes such as neighbors synchronization or pairwise synchronization could be more efficient than a full barrier and yet sufficient for correct execution of the program. The caveat is that custom synchronization code is generally hard to write, debug, understand and maintain.

Deciding which barrier (local or global) to use can be quite tricky. Programs such as quicksort and other divide-and-conquer algorithms which have a tree-like task graph are naturally expressed using local barriers. On the other hand, many scientific and numerical applications, where computations proceeds in phases or iterations, are a natural fit for global barriers. Also, programs in which locality may be exploited across barriers may perform poorly, if they are implemented naïvely using local barriers.

Environment	Explicit Barrier	Implicit Barrier
MPI	`int MPI_Barrier(MPI_Comm comm)`	Some MPI Collective Communication constructs [a]
OpenMP	`#pragma omp barrier`	`for` directive
Charm++	`void contribute()`	None
CUDA	`__syncthreads()`	End of kernel function
Cilk	`sync` keyword	End of Cilk procedure
FJ Framework	`void join()`	`void coinvoke()`
Intel's TBB	`void wait_for_all()` and variants	`void parallel_for()`
Java Thread API	`void join()`	None
Java5 Concurrency API	`CyclicBarrier` and `CountDownLatch`	None

[a] According to Section 4.1 of the MPI: A Message-Passing Interface Standard report (version 1.1) the presence of an implicit barrier during a collective communication call is implementation specific. For correctness and portability, a programmer should not rely on the presence of an implicit barriers. On the other hand, for efficiency, a programmer has to account for the possibility that a particular implementation includes implicit barriers for those constructs.

Table 1: Barrier abstraction in parallel programming environments

Many environments also provide implicit barrier at the end of constructs such as parallel `for` loop or a code block. Programmers should be aware of any implicit barriers in their programs, specially given the implications of barriers on execution performance as discussed above.

4.1 Implementation

Barrier synchronization on distributed, message-passing systems such as MPI is commonly implemented using a tree-based approach [XMN92], having logarithmic cost in terms of messages and latency. On shared memory systems, the butterfly algorithm [Bro86] or its variant is commonly used. The algorithm is shown to have logarithmic scaling properties for large number of processors and avoid hot-spots associated with a tree-like approach. The butterfly approach has also been adapted to work for barrier synchronization on distributed nodes.

5 Invariants

Precondition A collection of concurrent UEs that need to be synchronized at a point in the program.

Invariant A UE that reaches the barrier does not continue until all other UEs corresponding to the same barrier hit the barrier.

Postcondition The blocked UEs continue their corresponding computation only after all the other UEs involved in the barrier reach the barrier point.

6 Examples

In this section, we present examples which show usage of different kinds of barriers i.e global, local and implicit barriers.

6.1 Quicksort with Cilk using local barrier

Listing 1 shows quicksort algorithm implemented in Cilk language [FLR98]. The program uses the built-in primitive `sync` for expressing a local barrier.

Listing 1 Quicksort Example Using `sync` in Cilk

```
1  void main (int[] A, int n) {
2    qsort(A, 0, n);
3    // 0 inclusive, n exlusive
4  }
5
6  cilk void qsort (int[n] A, int i, int j) {
7    if (j-i < 2) return;
8    int pivot = A[i]; //first element
9    int p_index = partition(A, pivot, i, j);
10
11   spawn qsort(A, i, p_index);
12   spawn qsort(A, p_index, j);
13
14   sync; // local barrier in Cilk
15
16   // print the sorted array
17   }
18 }
19
20 int partition(int[] A, int pivot, int i, int j) {
21   ...
22 }
```

6.2 Successive-over-relaxation in OpenMP using implicit local barrier

Successive over-relaxation is a kernel used in numerical method computation to speed up the convergence of Gauss-Seidel [Gau] method. In order to parallelize this kernel, a reordering is performed on the elements such that alternate elements can be "over-relaxed" in parallel, followed by the remaining elements. These two phases can be repeated until the desired convergence threshold is reached. This ordering is called the Red-Black ordering.

Listing 2 describes successive over-relaxation in OpenMP. The grid is divided row-wise among the UEs using the `parallel for` directive. An implicit barrier at the end of for loop synchronizes the parallel UEs.

A downside of using local barriers for synchronizing UEs across iterations or phases is that local barriers typically wait for the UEs to finish before the master UE can proceed. A new set of UEs is, hence created for the next iteration which may be placed on a processor different from the one executing the previous iteration. This results in poor locality and can hurt performance severely.

Listing 2 Successive over-relaxation Example Using `implicit barrier` in OpenMP

```
1  void main (int[][] A, int m, int n, int nIterations) {
2
3    for (int i = 0; i < nIterations; i++) {
4
5      //update red
6      #pragma omp parallel for
7      for (int j = 0; j < m; j++) {
8
9        for (int k = 0; k < n; k++) {
10         if ((j + k) % 2 == 0) //verify it is red
11           A[j][k] = 0.25 *
12           (A[j][k] + A[j][k] + A[j][k-1] + A[j][k+1]);
13       }
14     } // implicit barrier here
15
16     //update black
17     #pragma omp parallel for
18     for (int j = 0; j < m; j++) {
19
20       for (int k = 0; k < n; k++) {
21         if ((j + k) % 2 == 1) //verify it is black
22           A[j][k] = 0.25 *
23           (A[j-1][k] + A[j+1][k] + A[j][k-1] + A[j][k+1]);
24       }
25     } // implicit barrier here
26   }
27 }
```

Realizing this performance penalty, OpenMP, Intel's TBB provide constructs for declaring affinity of UEs to processors.

6.3 Successive over-relaxation in MPI using global barrier

Alternatively, a programmer can statically map the UEs to processors in MPI, and then employ global barriers to synchronize the UEs. This preserves locality of reference across the barrier synchronization. Listing 3 shows the successive over-relaxation program written in MPI that employs a global barrier.

6.4 Successive over-relaxation using neighbor synchronization

On carefully observing the successive-over-relaxation algorithm, one can notice that the parallel UEs access the data of their neighbors only. Hence, it is sufficient to synchronize each UE with its two neighbors only. Barrier synchronization is stronger than the synchronization required for correct semantics of the successive over-relaxation algorithm. Also, barrier synchronization is expensive in terms of overhead. Therefore, when performance is a concern, programs like successive-over-relaxation are implemented using custom synchronization schemes. An implementation of successive over-relaxation in Java using neighbor synchronization can be found in Java Grande Benchmarks [SBO01]. As discussed earlier, custom synchronization code is generally hard to write,

Listing 3 Successive over-relaxation Example Using a `global barrier` in MPI

```
8  int main(int argc, char* argv[]) {
9
10   int my_id;
11   int number_of_processors;
12
13   // SOR with row-wise agglomeration
14   //
15   // Initialize MPI and set up SPMD programs
16   //
17   MPI_Init(&argc,&argv);
18   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
19   MPI_Comm_size(MPI_COMM_WORLD, &number_of_processors);
20
21   if(my_id == MASTER_NODE) {
22           // broadcast A to the slave nodes
23   }
24   else {  // slave nodes perform the SOR computation
25
26     // receive the broadcast containing A from the master node
27
28     for (int i = 0; i < ITERATIONS; i++) {
29       //update red
30         for (int k = 1; k < n; k++) {
31          if ((my_id + k) % 2 == 0) //verify it is red
32            A[my_id][k] = 0.25 *
33              (A[my_id][k] + A[my_id][k] + A[my_id][k-1] + A[my_id][k+1]);
34          }
35
36       MPI_BARRIER(MPI_COMM_WORLD);
37
38       //update black
39         for (int k = 1; k < n; k++) {
40          if ((my_id + k) % 2 == 1) //verify it is black
41            A[my_id][k] = 0.25 *
42              (A[my_id][k] + A[my_id][k] + A[my_id][k-1] + A[my_id][k+1]);
43          }
44       MPI_BARRIER(MPI_COMM_WORLD);
45      }
46   }
47
48   MPI_Finalize();
49   return 0;
50 }
```

debug, understand and maintain.

6.5 Conway's Game of Life in CUDA using implicit global barrier

Conway's game of life is a cellular automaton first proposed by the British mathematician John Horton Conway in 1970. The game is a simulation on a two-dimensional grid of cells. Each cell starts off as either *alive* or *dead*. The state of the cell changes depending on the state of its each

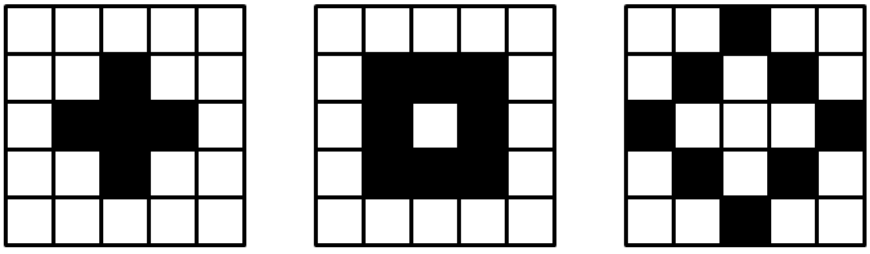
neighbors in the grid. At each time-step, we update the state of each cell according to the following four rules.

1. A live cell with fewer than two live neighbors dies due to underpopulation.
2. A live cell with more than three live neighbors dies due to overpopulation.
3. A live cell with two or three live neighbors survives to the next generation.
4. A dead cell with exactly three live neighbors becomes a live cell due to breeding.

Figure 3 shows an example of the Conway's Game of Life. The example starts with a cross and *evolves* through two time steps.

Listing 4 shows a parallel implementation of the game of life in CUDA. The program generates threads equal to the number of cells, and updates the status of each cell independently. Listing 5 shows how we update the status of each grid. Before proceeding to the next time step, it is necessary that all the grids have been updated. This requirement can be ensured by using a global barrier for all threads. CUDA provides an implicit global barrier at the end of execution of each kernel function as shown in listing 4.

Figure 3 2 time steps of a life game starting with the topology of a cross



7 Known Uses

SIMD computations proceed in lock-step fashion and therefore, have an implicit barrier.

8 Related Patterns

Collective Synchronization Barrier provides a way for synchronizing a set of UEs. It is a basic pattern for the Collective Synchronization [OPL] pattern.

Reduction In some parallel programming environments such as Charm++ [Cha], the Reduction abstraction is used with a `NULL` operator to effectively behave as a Barrier.

Rendezvous A Rendezvous is a special case of barrier synchronization in which only two UEs are involved. This distinction is sometimes not rigidly observed.

Listing 4 Routine for Game of Life simulation

```
42 void GameOfLife(int timePeriod, int width, int height, int* devInputArray,
43   int* devOutputArray)
44 {
45   dim3 blockDim(BLOCKX, BLOCKY);
46   dim3 gridDim((width+BLOCKX - 1)/BLOCKX, (height+BLOCKY- 1)/BLOCKY);
47
48   for (int i = 0; i < timePeriod; i++)
49   {
50     UpdateStatus<<<gridDim, blockDim>>>(width, height, devOutputArray);
51     //Implicit Global Barrier Here.
52     cudaMemcpy(devInputArray, devOutputArray, sizeof(int)*width*height,
53       cudaMemcpyDeviceToDevice);
54   }
55 }
```

Listing 5 Routine for updating the status of a grid

```
13 __global__ void UpdateStatus(int width, int height, int* devArrayOutput)
14 {
15   int x = IMUL(blockDim.x, blockIdx.x) + threadIdx.x;
16   int y = IMUL(blockDim.y, blockIdx.y) + threadIdx.y;
17   int id = y*width+x;
18   int count = 0;
19   for (int i = -1; i < 2; i++)
20   {
21     for (int j = -1; j < 2; j++)
22     {
23       int xnext = x + i;
24       int ynext = y + j;
25       if ((xnext >= 0 && xnext <=width) &&
26         (ynext >= 0 && ynext <=height) &&
27         (!(i == 0 && j == 0)))
28       {
29         if (tex1Dfetch(texStatus, ynext*width+xnext) == 1)
30           count++;
31       }
32     }
33   }
34
35   devArrayOutput[id] = 0;
36   if (count == 3)
37     devArrayOutput[id] = 1;
38   if (count == 2)
39     devArrayOutput[id] = tex1Dfetch(texStatus,id);
40 }
```

References

[BH86] J. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324:446–449, December 1986.

[Bro86] E.D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[Cha] Charm++ Parallel Programming Model. http://charm.cs.uiuc.edu/.

[FLR98] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[Gau] Gauss-Seidel Method. http://mathworld.wolfram.com/Gauss-SeidelMethod.html.

[HS98] JMD Hill and DB Skillicorn. Practical barrier synchronisation. In *Parallel and Distributed Processing, 1998. PDP'98. Proceedings of the Sixth Euromicro Workshop on*, pages 438–444, 1998.

[OPL] Berkeley Pattern Language for Parallel Programming. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.

[SBO01] LA Smith, JM Bull, and J. Obdrizalek. A parallel Java Grande benchmark suite. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 6–6, 2001.

[SMS96] M.L. Scott, M.M. Michael, and ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE. The Topological Barrier: A Synchronization Abstraction for Regularly-Structured Parallel Applications, 1996.

[Tse95] C.W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 144–155. ACM New York, NY, USA, 1995.

[XMN92] H. Xu, P.K. McKinley, and L.M. Ni. Efficient implementation of barrier synchronization in wormhole-routed hypercube multicomputers. *Journal of Parallel and Distributed Computing*, 16:172–172, 1992.