The Topological Barrier: A Synchronization Abstraction for Regularly-Structured Parallel Applications*

Michael L. Scott and Maged M. Michael

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scott,michael}@cs.rochester.edu

keywords: barriers, synchronization, abstraction, communication topology

January 1996

Abstract

Barriers are a simple, widely-used technique for synchronization in parallel applications. In regularly-structured programs, however, barriers can overly-constrain execution by forcing synchronization among processes that do not really share data. The *topological barrier* preserves the simplicity of traditional barriers while performing the minimum amount of synchronization actually required by the application. Topological barriers can easily be retro-fitted into existing programs. The only new burden on the programmer is the construction of a pair of functions to count and enumerate the neighbors of a given process. We describe the topological barrier in pseudo-code and pictures, and illustrate its performance on a pair of applications.

*This work was supported in part by NSF grants nos. CDA-8822724 and CCR-9319445, and by ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

1 Introduction

Many scientific applications, particularly those involving the simulation of physical systems, display a highly regular structure, in which the elements of a large, multi-dimensional array are updated in an iterative fashion, based on the current values of nearby elements. In the typical shared-memory parallelization of such an application, each process is responsible for updates in a polygonal (usually rectilinear) sub-block of the array. During the course of one iteration, a process updates values in the interior of its block, communicates with its neighbors to update values on the periphery of its block, and then passes through a *barrier*, which prevents it from beginning the next iteration until all other processes have completed the current iteration.

If the new value of each array element indeed depends only on the values of nearby elements, then correctness requires synchronization only among neighbors. A general-purpose, all-process barrier is overkill: it forces mutually distant processes to synchronize with each other even though they share no data. Performance may suffer for several reasons:

- In the absence of special-purpose hardware, a barrier requires $O(\log p)$ serialized steps to synchronize p processes. If the number of neighbors of any given process is bounded by a constant, then it should be possible to synchronize among neighbors in $O(1)$ serialized steps.
- If work completes more quickly in some regions of the array than it does in other regions, then some earlier-arriving processes may be forced to wait at the barrier when they could be working on the next iteration.
- Because processes leave a barrier at roughly the same time, communication in barrier-based applications tends to be quite bursty, as newly-released processes all attempt to fetch remote data at once. The resulting contention for memory and network resources can increase the latency of communication dramatically.

Clearly nothing prevents the programmer from implementing the minimum amount of synchronization required by the application. Barriers are attractive, however, even if they over-synchronize, because they are so simple. As part of some recent experiments in software-managed cache coherence [3], we re-wrote a banded implementation of successive over-relaxation to use locks on boundary rows, rather than barriers, to synchronize between iterations. Performance did improve, but the changes required to the source were non-trivial: a single line of code in the original application (i.e. the call to the barrier) turned into 54 lines of lock acquisitions and releases in the newer version. The extra code is not subtle: just tedious. It could be incorporated easily in programs generated by parallelizing compilers [5]. For programs written by human beings, however, it is a major nuisance.

What we need for hand-written programs is a programming abstraction that preserves the simplicity of barriers from the programmer's point of view, while performing the minimum amount of synchronization necessary in a given application. We present such an abstraction in section 2. We call it a *topological barrier*; it exploits the sharing topology of the application. In section 3 we illustrate the performance advantage of topological barriers on a pair of applications. In section 4 we summarize conclusions.

2 The Algorithm

Pseudo-code for the topological barrier appears in figure 1. A pictorial representation of the barrier's data structures appears in figure 2. Each process has a private copy of a 4-field record

that represents the barrier. The first field is a serial number that counts the number of barrier episodes that have been completed so far. The only purpose of this counter is to provide a value that is different in each episode, and on which all processes agree. The counter can be as small as two bits; roll-over is harmless. The second field of the barrier record for process i indicates the number of neighbors of i; this is the number of elements in the *neighbors* and *flags* arrays. In a toroidal topology, every process would have the same number of neighbors; in a mesh the processes on the edges would have fewer.

The *neighbors* array for process i contains pointers to flag words in the *flags* arrays of i's neighbors; i's own flag words are pointed at by elements of the *neighbors* arrays of i's neighbors. To pass a barrier, process i (1) increments its copy of *serial_num*, (2) writes this value into the appropriate flag variables of its neighbors, and (3) waits for its neighbors to write the same value into its own flags. To minimize communication and contention, flag words should be local to the process that spins on them, either via explicit placement on a non-coherent shared-memory machine, or via automatic migration on a cache-coherent machine. To prevent a process from proceeding past a barrier and over-writing its neighbor's flag variables before that neighbor has had a chance to notice them, we alternate use of two different sets of flags in consecutive barrier episodes.

Gupta [2] has noted that in many applications there is work between iterations that is neither required by neighbors in the next iteration, nor dependent on the work of neighbors in the iteration just completed. Performance in such applications can often be improved by using a *fuzzy barrier*, which separates the arrival ("I'm here") and departure ("Are you all here too?") phases of the barrier into separate subroutines. We can create a fuzzy version of *topological_barrier* trivially, by breaking it between the third and fourth lines.

The initialization routine *top_bar_init* must be called concurrently by all processes in the program. It takes four arguments. The first two specify the (private) barrier record to be initialized and the total number of processes that will participate in barrier episodes. The last two arguments are formal subroutines that *top_bar_init* can use to determine the number and identity of the neighbors of a given process p. These two subroutines must be re-written for every sharing topology. A synchronization library that included topological barriers would presumably provide routines for common topologies (lines, rings, meshes, tori); application programmers can write others as required. Programmers can also declare and initialize more than one barrier record within a single application, to accommodate different sharing topologies in different phases of the computation, or to allow subsets of processes to synchronize among themselves. An example pair of routines, in this case for an $N \times N$ square mesh, appears in figure 3.

Initialization proceeds in two phases. In the first phase, each process (1) calls a user-provided routine to determine its number of neighbors, (2) allocates space to hold its *neighbors* and *flags* arrays, (3) calls another user-provided routine to temporarily fill the *neighbors* array with the process ids of its neighbors, and (4) writes pointers to these arrays, together with the count of neighbors, into a static shared array, where they can be seen by process 1. In the second phase of initialization, process 1 uses the information provided by the various other processes to initialize the pointers in all of the *neighbors* arrays. To separate the two phases, and to confirm that the second phase has finished, we assume the existence of a standard, all-process barrier.

To retro-fit an existing application to use topological barriers, the programmer must (1) obtain or write a suitable pair of topology routines (*num_neighbors* and *enumerate_neighbors*), (2) insert a call to *top_bar_init*, and (3) replace general barrier calls with calls to *topological_barrier*. These tasks are substantially easier than coding the required synchronization explicitly, in-line.

```
proc = 1..MAX_PROCS                 -- NB: all arrays are indexed 1..whatever.
parity = 0..1
top_bar = record
    serial_num : small integer      -- roll-over is harmless
    nn : integer                    -- number of neighbors
    neighbors : pointer to array of pointer to array [parity] of small integer
    flags : pointer to array of array [parity] of small integer
    -- Two sets of neighbor and flag variables, for alternate barrier episodes.

-- Variables of type top_bar should be declared private.  They should be initialized
-- by calling top_bar_init in all processes concurrently.  Note that top_bar_init
-- assumes the existence of a general all-process barrier called basic_barrier.

private self : proc

top_bar_init (var tb : top_bar; num_procs : integer
              num_neighbors (p : proc) : integer
              enumerate_neighbors (p : proc; var list : array of proc))

    -- The following array is used during initialization only,
    -- to communicate neighbor information among processes:
    static shared top_bar_info : array [proc] of record
        nn : integer
        neighbors : pointer to array of pointer to array [parity] of small integer
        flags : pointer to array of array [parity] of small integer
        next_flag : integer

    tb.serial_num := 0; tb.nn := num_neighbors (self)

    new[tb.nn] tb.neighbors         -- allocate nn x 1 array
    new[tb.nn][2] tb.flags          -- allocate nn x 2 array
        -- Shared.  On an NCC-NUMA machine, must be physically local to self.

    enumerate_neighbors (self, (array of proc) tb.neighbors^)
        -- Temporarily treat neighbors as an array of process ids,
        -- rather than pointers to flag variables.

    foreach i : integer in 1..tb.nn
        tb.flags^[i][0] := tb.flags^[i][1] := 0

    top_bar_info[self].nn := tb.nn;         top_bar_info[self].neighbors := tb.neighbors
    top_bar_info[self].flags := tb.flags;   top_bar_info[self].next_flag := 1

    basic_barrier()     -- synchronize all processes (top_bar_info is initialized)
    if self = 1
        foreach p : proc in 1..num_procs
            foreach i : integer in 1..top_bar_info[p].nn
                nb : proc := top_bar_info[p].neighbors^[i]
                top_bar_info[p].neighbors^[i] := &top_bar_info[nb].flags^[top_bar_info[nb].next_flag]
                top_bar_info[nb].next_flag++

    basic_barrier()     -- synchronize all processes (top_bar is initialized)

topological_barrier (var tb : top_bar)
    tb.serial_num++     -- roll-over is harmless
    foreach i : integer in 1..tb.nn
        tb.neighbors^[i]^[serial_num % 2] := tb.serial_num
    foreach i : integer in 1..tb.nn
        repeat /* spin */ until tb.flags^[i][serial_num % 2] = serial_num
```

Figure 1: Pseudo-code for the topological barrier. This code is the same for all applications, regardless of sharing topology.

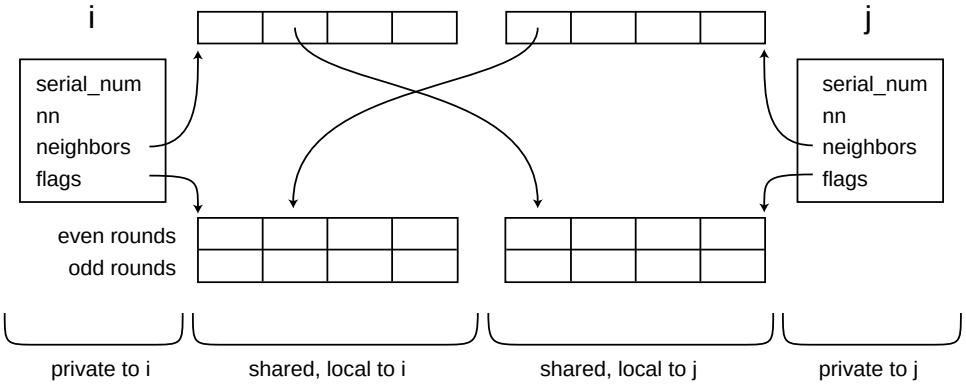

Figure 2: Pictorial representation of the topological barrier data structures. Process j is i's second neighbor; process i is j's first neighbor.

```
num_neighbors (p : proc)
    if p in {1, N, N*(N-1)+1, N*N}
        return 2
    if p < N or p > N*(N-1) or (p-1) % N in {0, N-1}
        return 3
    return 4

enumerate_neighbors (p : proc; var list : array of proc)
    if p = 1
        list[1] := 2; list[2] := N+1
    elsif p = N
        list[1] := N-1; list[2] := N*2
    elsif p = N*(N-1)+1
        list[1] := N*(N-2)+1; list[2] := N*(N-1)+2
    elsif p = N*N
        list[1] := N*(N-1); list[2] := N*N-1
    elsif p < N
        list[1] := p-1; list[2] := p+1; list[3] := p+N
    elsif (p-1) % N = 0
        list[1] := p-N; list[2] := p+N; list[3] := p+1
    elsif (p-1) % N = N-1
        list[1] := p-N; list[2] := p+N; list[3] := p-1
    elsif p > N*(N-1)
        list[1] := p-1; list[2] := p+1; list[3] := p-N
    else
        list[1] := p-1; list[2] := p+1; list[3] := p-N; list[3] := p+N
```

Figure 3: Example topology functions for a square $N \times N$ mesh ($N > 1$). Code of this sort needs to be written for each different sharing topology.

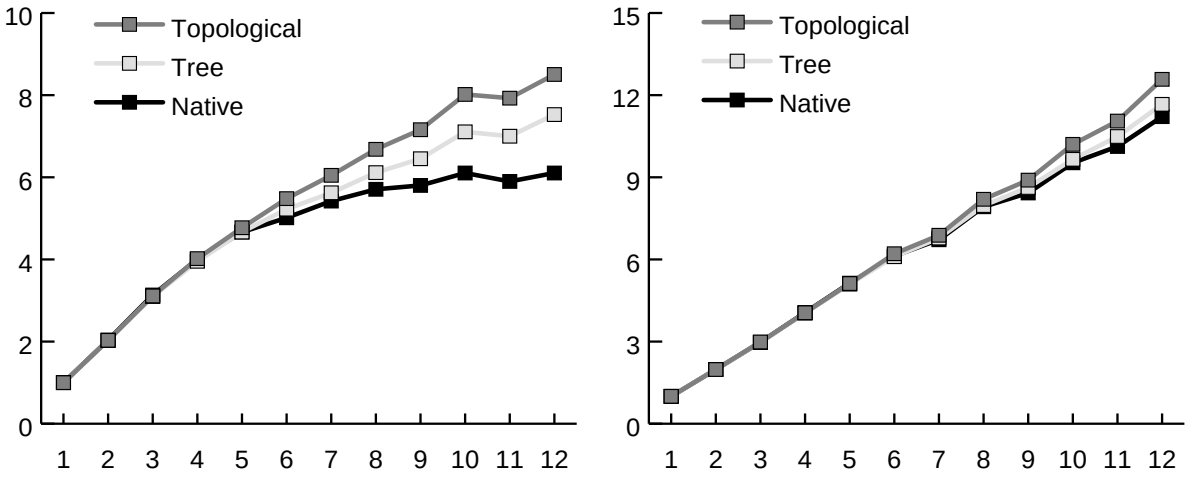

Figure 4: Speedup graphs for SOR (left) and Mgrid (right) on a 12-processor SGI Challenge. Absolute times on one processor are the same for all three barriers.

3 Experimental Results

To verify the usefulness of the topological barrier abstraction, we retro-fitted a pair of regularly-structured applications—SOR and Mgrid—and measured their performance on a 12-processor SGI Challenge machine. We ran the applications with the topological barrier, the native (SGI library) barrier, and a tree-based barrier known to provide excellent performance and to scale well to large machines [4]. Speedup graphs appear in figure 4. Absolute running times on one processor were essentially the same for all three barriers.

SOR computes the steady state temperature of a metal sheet using a banded parallelization of red-black successive over-relaxation. We used a small (100×100) grid, to keep the number of rows per processor small and highlight the impact of synchronization. Mgrid is a simplified shared-memory version of the multigrid kernel from the NAS Parallel Benchmarks [1]. It performs a more elaborate over-relaxation using multi-grid techniques to compute an approximate solution to the Poisson equation on the unit cube. We ran 10 iterations, with 100 relaxation steps in each iteration, and a grid size of $8 \times 8 \times 120$.

For SOR on 12 processors, absolute running time with the topological barrier was 28% faster than with the native barrier, and 11% faster than with the tree barrier. For Mgrid on 12 processors, running time with the topological barrier was 11% faster than with the native barrier, and 7% faster than with the tree barrier. For any given number of processors, the relative impact of barrier performance decreases with larger problem sizes, since each process does more work per iteration. Our small data sets may therefore overestimate the importance of barrier performance on small machines. Speculating about larger machines, we observe that increasing both the problem size and the number of processors, together, should increase the performance differences among barrier implementations, since the time to complete a general barrier is logarithmic in the number of processes, while the time to complete a topological barrier is essentially constant.

4 Conclusions

We have introduced topological barriers as a programming abstraction for neighbor-based busy-wait synchronization in regularly-structured, iterative, shared-memory parallel programs, and have illustrated its utility with performance results for a pair of applications on a 12-processor SGI machine.

The notion of neighbor-based synchronization is not new; our contribution is to make it easy. The only potentially tricky part is to obtain or write a pair of functions to count and enumerate the neighbors of a process. Given these functions, a topological barrier can be retro-fitted into an existing barrier-based application simply by adding a call to an initialization routine, and then calling a different barrier. The latter task can be achieved either by modifying the source or by changing the identity of the *barrier* library routine.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NASA Ames Research Center, January 1991.

[2] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Boston, MA, April 1989.

[3] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, San Jose, CA, February 1996. Earlier version available as "Distributed Shared Memory for New Generation Networks," TR 578, Computer Science Department, University of Rochester, March 1995.

[4] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[5] S. P. Midkiff and D. A. Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, C-36(12), December 1987.