Barrier (computer science)

From Wikipedia, the free encyclopedia

This article **requires immediate attention** for reasons given on the talk page. After reading the **talk page**, if you can help, please **edit this page**.

In parallel computing, a **barrier** is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Many collective routines and directive-based parallel languages impose implicit barriers. For example, a parallel *do* loop in Fortran with OpenMP will not be allowed to continue on any thread until the last iteration is completed. This is in case the program relies on the result of the loop immediately after its completion. In message passing, any global communication (such as reduction or scatter) may imply a barrier.

Contents [hide]

Implementation [edit]

The basic barrier has mainly two variables, one of which records the pass/stop state of the barrier, the other of which keeps the total number of threads that have entered in the barrier. The barrier state was initialized to be "stop" by the first threads coming into the barrier. Whenever a thread enters, based on the number of threads already in the barrier, only if it is the last one, the thread sets the barrier state to be "pass" so that all the threads can get out of the barrier. On the other hand, when the incoming thread is not the last one, it is trapped in the barrier and keeps testing if the barrier state has changed from "stop" to "pass", and it gets out only when the barrier state changes to "pass". The following C++ code demonstrates this procedure.[1][2]

```
struct barrier_type
```

```
{
    // how many processors have entered the barrier
    // initialize to 0
    int arrive_counter;
    // how many processors have exited the barrier
    // initialize to P
    int leave_counter;
    int flag;
    std::mutex lock;
};

// barrier for p processors
void barrier(barrier_type* b, int p)
{
    b->lock.lock();
    if (b->leave_counter != p)
    {
        b->lock.unlock();
        while (b->leave_counter != p); // wait for all to leave before clearing
        b->lock.lock();
    }
    if (b->arrive_counter == 0) // no other threads in barrier
    {
        b->flag = 0; // first arriver clears flag
    }
    b->arrive_counter++;
    int arrived = b->arrive_counter;
    b->lock.unlock();
    if (arrived == p) // last arriver sets flag
    {
        b->arrive_counter = 0;
        b->leave_counter = 1;
        b->flag = 1;
    }
    else
    {
        while (b->flag == 0); // wait for flag
        b->lock.lock();
        b->leave_counter++;
        b->lock.unlock();
    }
}
```

The potential problems are as follows:

1. When sequential barriers using the same pass/block state variable are implemented, a deadlock could happen in the first barrier whenever a thread reaches the second and there are still some threads have not got out of the first barrier.
2. Due to all the threads repeatedly accessing the global variable for pass/stop, the communication traffic is rather high, which decreases the scalability.

The following Sense-Reversal Centralized Barrier is designed to resolve the first problem. And

the second problem can be resolved by regrouping the threads and using multi-level barrier, e.g. Combining Tree Barrier. Also hardware implementations may have the advantage of higher scalability.

Sense-Reversal Centralized Barrier [edit]

A Sense-Reversal Centralized Barrier solves the potential deadlock problem arising when sequential barriers are used. Instead of using the same value to represent pass/stop, sequential barriers use opposite values for pass/stop state. For example, if barrier 1 uses 0 to stop the threads, barrier 2 will use 1 to stop threads and barrier 3 will use 0 to stop threads again and so on.[3] The following C++ code demonstrates this.[1][4][2]

```
struct barrier_type
{
    int counter; // initialize to 0
    int flag; // initialize to 0
    std::mutex lock;
};

int local_sense = 0; // private per processor

// barrier for p processors
void barrier(barrier_type* b, int p)
{
    local_sense = (b->flag == 0) ? 1 : 0;
    b->lock.lock();
    b->counter++;
    int arrived = b->counter;
    if (arrived == p) // last arriver sets flag
    {
        b->lock.unlock();
        b->counter = 0;
        // memory fence to ensure that the change to counter
        // is seen before the change to flag
        b->flag = local_sense;
    }
    else
    {
        b->lock.unlock();
        while (b->flag != local_sense); // wait for flag
    }
}
```

Combining Tree Barrier [edit]

A Combining Tree Barrier is a hierarchical way of implementing barrier to resolve the scalability by avoiding the case that all threads spinning on a same location.[3]

In k-Tree Barrier, all threads are equally divided into subgroups of k threads and a first-round synchronizations are done within these subgroups. Once all subgroups have done their

synchronizations, the first thread in each subgroup enters the second level for further synchronization. In the second level, like in the first level, the threads form new subgroups of k threads and synchronize within groups, sending out one thread in each subgroup to next level and so on. Eventually, in the final level there is only one subgroup to be synchronized. After the final-level synchronization, the releasing signal is transmitted to upper levels and all threads get past the barrier.[4][5]

Hardware Barrier Implementation [edit]

The hardware barrier uses hardware to implement the above basic barrier model.[1]

The simplest hardware implementation uses dedicated wires to transmit signal to implement barrier. This dedicated wire performs OR/AND operation to act as the pass/block flags and thread counter. For small systems, such a model works and communication speed is not a major concern. In large multiprocessor systems this hardware design can make barrier implementation have high latency. The network connection among processors is one implementation to lower the latency, which is analogous to Combining Tree Barrier.[6]

See also [edit]

- Fork–join model
- Rendezvous (Plan 9)
- Memory barrier

References [edit]

1. ^ a b c Solihin, Yan (2015-01-01). *Fundamentals of Parallel Multicore Architecture* (1st ed.). Chapman & Hall/CRC. ISBN 1482211181.
2. ^ a b "Implementing Barriers". Carnegie Mellon University.
3. ^ a b Culler, David (1998). *Parallel Computer Architecture, A Hardware/Software Approach*. ISBN 978-1558603431.
4. ^ a b Nanjegowda, Ramachandra; Hernandez, Oscar; Chapman, Barbara; Jin, Haoqiang H. (2009-06-03). Müller, Matthias S.; Supinski, Bronis R. de; Chapman, Barbara M., eds. *Evolving OpenMP in an Age of Extreme Parallelism*. Lecture Notes in Computer Science. Springer Berlin Heidelberg. pp. 42–52. doi:10.1007/978-3-642-02303-3_4. ISBN 9783642022845.
5. ^ Nikolopoulos, Dimitrios S.; Papatheodorou, Theodore S. (1999-01-01). "A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000". *Proceedings of the 13th International Conference on Supercomputing*. ICS '99. New York, NY, USA: ACM: 319–328. doi:10.1145/305138.305209. ISBN 158113164X.
6. ^ N.R. Adiga, et al. An Overview of the BlueGene/L Supercomputer. *Proceedings of the Conference on High Performance Networking and Computing,* 2002.

V · T · E	Parallel computing [hide]
General	Distributed computing · Parallel computing · Massively parallel · Cloud computing · High-performance computing · Multiprocessing · Manycore processor · GPGPU · Computer network · Systolic array
Levels	Bit · Instruction · Thread · Task · Data · Memory · Loop · Pipeline

Multithreading	Temporal · Simultaneous (SMT) · Speculative (SpMT) · Preemptive · Cooperative · Clustered Multi-Thread (CMT) · Hardware scout
Theory	PRAM model · Analysis of parallel algorithms · Amdahl's law · Gustafson's law · Cost efficiency · Karp–Flatt metric · Slowdown · Speedup
Elements	Process · Thread · Fiber · Instruction window
Coordination	Multiprocessing · Memory coherency · Cache coherency · Cache invalidation · **Barrier** · Synchronization · Application checkpointing
Programming	Stream processing · Dataflow programming · Models (Implicit parallelism · Explicit parallelism · Concurrency) · Non-blocking algorithm
Hardware	Flynn's taxonomy (SISD · SIMD · SIMT · MISD · MIMD) · Dataflow architecture · Pipelined processor · Superscalar processor · Vector processor · Multiprocessor (symmetric · asymmetric) · Memory (shared · distributed · distributed shared · UMA · NUMA · COMA) · Massively parallel computer · Computer cluster · Grid computer
APIs	Ateji PX · Boost.Thread · Chapel · Charm++ · Cilk · Coarray Fortran · CUDA · Dryad · C++ AMP · Global Arrays · MPI · OpenMP · OpenCL · OpenHMPP · OpenACC · TPL · PLINQ · PVM · POSIX Threads · RaftLib · UPC · TBB · ZPL
Problems	Deadlock · Livelock · Deterministic algorithm · Embarrassingly parallel · Parallel slowdown · Race condition · Software lockout · Scalability · Starvation
Category: parallel computing · Media related to Parallel computing at Wikimedia Commons	

[1]

1. ^ http://blogs.sourceallies.com/2012/03/parallel-programming-with-barrier-synchronization/

Categories: Parallel computing