CISC 372: Parallel Computing
Threads, part 4: barrier and reduction implementations

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Barrier implementations

Last lecture, we saw a simple two-thread barrier.

Barrier implementations

Last lecture, we saw a simple two-thread barrier.

- two flags are used: `f1` and `f2`
  - `f1` is used by Thread 1 to send a signal to Thread 2 saying "I have arrived at barrier"
  - `f2` is used by Thread 2 to send a signal to Thread 1 saying "I have arrived at barrier"

Barrier implementations

Last lecture, we saw a simple two-thread barrier.

- two flags are used: `f1` and `f2`
  - `f1` is used by Thread 1 to send a signal to Thread 2 saying "I have arrived at barrier"
  - `f2` is used by Thread 2 to send a signal to Thread 1 saying "I have arrived at barrier"
- Thread 1
  1. raises `f1`
  2. lowers `f2`
- Thread 2
  1. lowers `f1`
  2. raises `f2`

Barrier implementations

Last lecture, we saw a simple two-thread barrier.

- two flags are used: `f1` and `f2`
  - `f1` is used by Thread 1 to send a signal to Thread 2 saying "I have arrived at barrier"
  - `f2` is used by Thread 2 to send a signal to Thread 1 saying "I have arrived at barrier"
- Thread 1
  1. raises `f1`
  2. lowers `f2`
- Thread 2
  1. lowers `f1`
  2. raises `f2`

Now let's generalize. . .

A counter barrier

A counter barrier

- a shared counter keeps track of number of threads in barrier

A counter barrier

- a shared counter keeps track of number of threads in barrier
- last thread to enter resets counter and signals all other threads to depart

A counter barrier

- a shared counter keeps track of number of threads in barrier
- last thread to enter resets counter and signals all other threads to depart
- one flag for each thread is used to transmit the departure signal

A counter barrier

- a shared counter keeps track of number of threads in barrier
- last thread to enter resets counter and signals all other threads to depart
- one flag for each thread is used to transmit the departure signal
- see `flag_barrier.c`

A counter barrier

- a shared counter keeps track of number of threads in barrier
- last thread to enter resets counter and signals all other threads to depart
- one flag for each thread is used to transmit the departure signal
- see `flag_barrier.c`

Analysis

- (+) only 1 flag per thread
- (-) contention on shared variable counter (and its lock)
- (-) $O(n)$ due to last thread's protocol

A coordinator barrier

A coordinator barrier

- use an additional "coordinator" thread
  - a special thread that is not part of the "team"
  - it keeps track of who is in the barrier
  - the n regular threads are "workers"

A coordinator barrier

- use an additional "coordinator" thread
  - a special thread that is not part of the "team"
  - it keeps track of who is in the barrier
  - the n regular threads are "workers"
- use two flags for each worker
  1. for worker to signal coordinator it has arrived
  2. for coordinator to signal worker it may leave

A coordinator barrier

- use an additional "coordinator" thread
  - a special thread that is not part of the "team"
  - it keeps track of who is in the barrier
  - the *n* regular threads are "workers"
- use two flags for each worker
  1. for worker to signal coordinator it has arrived
  2. for coordinator to signal worker it may leave
- worker barrier protocol
  1. worker signals coordinator "I have arrived"
  2. worker waits for departure signal from coordinator and lowers flag

A coordinator barrier

- use an additional "coordinator" thread
  - a special thread that is not part of the "team"
  - it keeps track of who is in the barrier
  - the *n* regular threads are "workers"
- use two flags for each worker
  1. for worker to signal coordinator it has arrived
  2. for coordinator to signal worker it may leave
- worker barrier protocol
  1. worker signals coordinator "I have arrived"
  2. worker waits for departure signal from coordinator and lowers flag
- coordinator protocol
  1. loop over workers:
     - wait for and lower arrival flag for each
  2. loop over workers:
     - send departure signal to each

A coordinator barrier

- use an additional "coordinator" thread
  - a special thread that is not part of the "team"
  - it keeps track of who is in the barrier
  - the *n* regular threads are "workers"
- use two flags for each worker
  1. for worker to signal coordinator it has arrived
  2. for coordinator to signal worker it may leave
- worker barrier protocol
  1. worker signals coordinator "I have arrived"
  2. worker waits for departure signal from coordinator and lowers flag
- coordinator protocol
  1. loop over workers:
     - wait for and lower arrival flag for each
  2. loop over workers:
     - send departure signal to each
- see `coordinator_barrier.c`

Analysis of coordinator barrier

- (+) avoids memory contention
- (-) requires an extra thread
- (-) $O(n)$: execution time is proportional to n
  - what if $n = 10^6$?

Combining binary tree barrier

Combining binary tree barrier

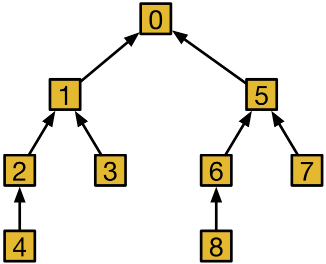
- combines actions of worker and coordinator
  - so each worker also coordinates

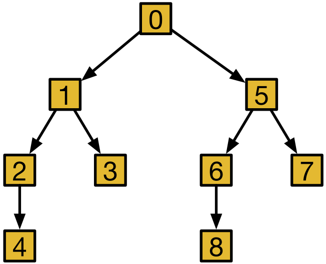
Combining binary tree barrier

- combines actions of worker and coordinator
  - so each worker also coordinates
- organizes workers in tree
  - exeuction time proportional to $\log(n)$

Combining binary tree barrier

- combines actions of worker and coordinator
  - so each worker also coordinates
- organizes workers in tree
  - exeuction time proportional to $\log(n)$
- flow of signals
  - send arrive signals up the tree
  - send depart signals down the tree
- sequence of events
  - worker waits for all children to arrive
  - then tells parent it has arrived
  - when root learns that its children have arrived
  - it knows all procs have arrived
  - then root tells its children to depart
  - when a worker is told to depart, it tells its children to depart, . . .

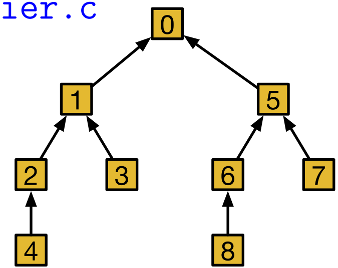


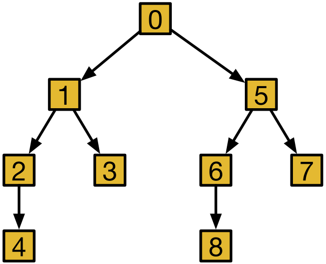


Combining binary tree barrier: `tree_barrier.c`

Combining binary tree barrier: `tree_barrier.c`

- protocol for leaf node *L*
  1. raise `arrive`[*L*]
  2. lower `depart`[*L*]
- protocol for interior node *I*
  1. lower `arrive`[*left*]
  2. lower `arrive`[*right*]
  3. raise `arrive`[*I*]
  4. lower `depart`[*I*]
  5. raise `depart`[*left*]
  6. raise `depart`[*right*]
- protocol for root node *R*
  1. lower `arrive`[*left*]
  2. lower `arrive`[*right*]
  3. raise `depart`[*left*]
  4. raise `depart`[*right*]





Analysis of combining tree barrier

- time is $O(\log(n))$
  - no loops $1..n$
  - each row in the tree can execute in parallel
- different procs execute different code
- leaf and root execute fewer instructions
  - could lead to inefficiency
- increases complexity

Symmetric barriers

Symmetric barriers

- in symmetric barriers
  - all procs execute same code

Symmetric barriers

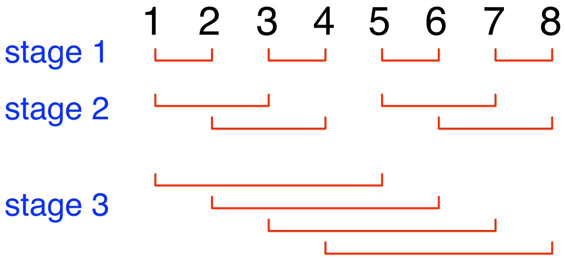
- in symmetric barriers
  - all procs execute same code
- common structure: solutions are constructed from pairs of 2-process barriers
  - Thread 1
    1. raises `f1`
    2. lowers `f2`
  - Thread 2
    1. lowers `f1`
    2. raises `f2`

Symmetric barriers

- need to choose interconnection scheme
  - sequence of 2-process barriers executed by each proc

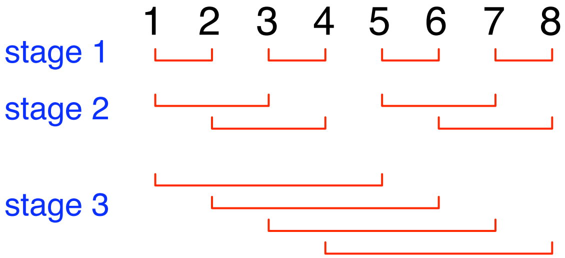
Symmetric barriers

- need to choose interconnection scheme
  - sequence of 2-process barriers executed by each proc
- Example: butterfly barrier
  - assume $n = 2^s$ is a power of 2



Symmetric barriers

- need to choose interconnection scheme
  - sequence of 2-process barriers executed by each proc
- Example: butterfly barrier
  - assume $n = 2^s$ is a power of 2



- butterfly: $O(\log(n))$ time and symmetric
- requires n to be power of 2

Dissemination Barrier: dissem_barrier.c

Dissemination Barrier: `dissem_barrier.c`

- butterfly requires *n* to be power of 2 or have exceptional code which breaks symmetry

Dissemination Barrier: `dissem_barrier.c`

- butterfly requires *n* to be power of 2 or have exceptional code which breaks symmetry
- dissemination barrier works for any *n*

Dissemination Barrier: `dissem_barrier.c`

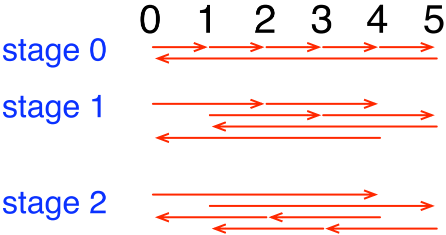
- butterfly requires n to be power of 2 or have exceptional code which breaks symmetry
- dissemination barrier works for any n
- uses cyclic order of threads

Dissemination Barrier: `dissem_barrier.c`

- butterfly requires n to be power of 2 or have exceptional code which breaks symmetry
- dissemination barrier works for any n
- uses cyclic order of threads
- two flags (`a` and `b`) for each thread, in each stage

Dissemination Barrier: `dissem_barrier.c`

- butterfly requires n to be power of 2 or have exceptional code which breaks symmetry
- dissemination barrier works for any n
- uses cyclic order of threads
- two flags (`a` and `b`) for each thread, in each stage
- in stage i each thread
  - synchs with thread 2^i to the right using `a` and `b` flags of that thread
  - synchs with thread 2^i to the left using its `a` and `b` flags
- stages: $0 \leq i < \lceil \log_2 n \rceil$



Dissemination barrier: code

```
for (int stage=0, i=1; stage<nstages; stage++, i*=2) {
  flag_raise(&bs->a[stage][(tid+i)%nthreads]);
  flag_lower(&bs->a[stage][tid]);
  flag_raise(&bs->b[stage][tid]);
  flag_lower(&bs->b[stage][(tid+i)%nthreads]);
}
```

Reductions

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier
  - each node has an associated value

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier
  - each node has an associated value
  - each node waits for arrival of its left and right child

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier
  - each node has an associated value
  - each node waits for arrival of its left and right child
  - then sets its value to sum of the value of the children

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier
  - each node has an associated value
  - each node waits for arrival of its left and right child
  - then sets its value to sum of the value of the children
  - then alerts its parent, . . .

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier
  - each node has an associated value
  - each node waits for arrival of its left and right child
  - then sets its value to sum of the value of the children
  - then alerts its parent, ...
  - root gets the global sum and can assign it to a global variable

Reductions

- each barrier algorithm can be extended to a reduction algorithm
  - for example, to sum elements of an array efficiently
- example: tree barrier
  - each node has an associated value
  - each node waits for arrival of its left and right child
  - then sets its value to sum of the value of the children
  - then alerts its parent, . . .
  - root gets the global sum and can assign it to a global variable
  - allows reduction without all threads contending for a single mutex

pthread_barrier

pthread_barrier

- Pthreads provides a barrier

pthread_barrier

- Pthreads provides a barrier
  - but this is an optional feature

pthread_barrier

- Pthreads provides a barrier
  - but this is an optional feature
  - not supported on all platforms (even those supporting Pthreads)

pthread_barrier

- Pthreads provides a barrier
  - but this is an optional feature
  - not supported on all platforms (even those supporting Pthreads)
  - for portable code: know how to write your own barrier
- `pthread_barrier_t` : type of a barrier object
- `pthread_barrier_init(...)`
  - `pthread_barrier_t *`
    - pointer to barrier object to initialize
  - `pthread_barrierattr_t *`
  - `unsigned int count`
    - number of threads that will participate in this barrier
- `pthread_barrier_destroy(pthread_barrier_t *)`
- `pthread_barrier_wait(pthread_barrier_t *)`