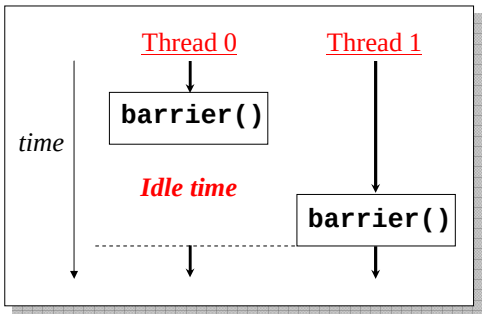
A Basic Problem: Long Latency Operations

Synchronization leads to idle time

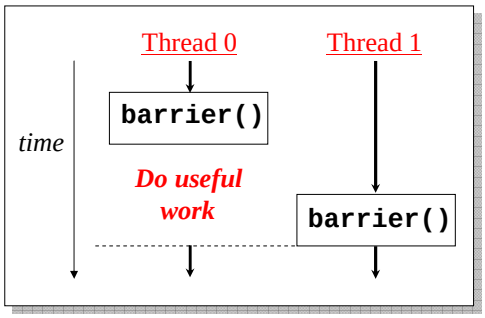
- Threads arrive at different times
- All except the last one has to wait



Hiding the Cost of Synchronization

Optimization

- Do useful work while waiting
- **Overlap synchronization latency with computation**



- **How do we accomplish this?**

Split-Phase Operations

Basic idea

- Separate **initiation** from **completion**
- No thread **completes** until all threads have **initiated**

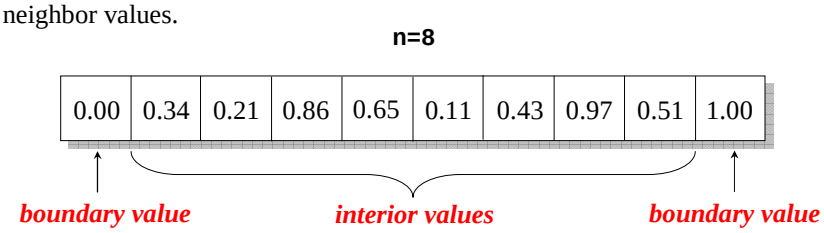
Example: Split-phase barrier

- Initiation: `barrier.arrive()`
- Completion: `barrier.wait()`

```
/* Initiate synchronization */
  barrier.arrive();

/* Do useful work */

/* Complete synchronization */
  barrier.wait();
```


Example: Relaxation

We start with an array of **n+2** values: **n** interior values and 2 boundary values. At each iteration, we replace each interior value with the average of its 2 neighbor values.



Relaxation on 2D and 3D arrays is used in practice to solve systems of differential equations such as the Navier-Stokes equations for fluid flow

Parallel Relaxation

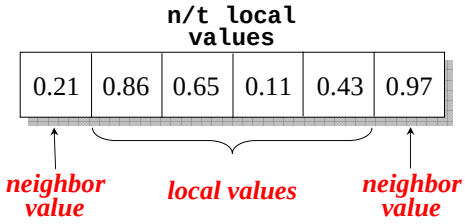
```
double *val, *new;          // Values array
int n;                      // Number of interior values
int t;                      // Number of threads
int iterations;             // Iterations to perform

thread_main(int self) {
  int n_per_thread = n / t;
  int start = self * n_per_thread;

  // For each iteration
  for (int i=0; i<iterations; i++) {
      // Update values
      for (int j=start; j<start+n_per_thread; j++)
         new[j] = (val[j-1] + val[j+1]) / 2.0;
      swap(new, val);
      barrier();            // Synchronize
  }
}
```

How do we use split-phase barriers to reduce synchronization costs?

Parallel Relaxation—A Closer Look

Consider an individual thread's work

- Each thread is responsible for updating `n/t` values at each iteration
- Each thread accesses **2** additional neighbor values:



  - The neighbor values are modified by the left and right neighbors, so the access and update of these neighbor values must be synchronized
  - The local values are only accessed by this thread, so they require no synchronization

Relaxation with Split-Phase Barriers

```
thread_main(int self) {
  ...
  // For each iteration
  for (int i=0; i<iterations; i++) {
      // Update neighbor values
      int j = start;
      val[j] = (val[j-1] + val[j+1]) / 2.0;
      j = start + n_per_thread –1;
      val[j] = (val[j-1] + val[j+1]) / 2.0;

      barrier.arrive();          // Initiate barrier

      // Update local values
      for (j=start+1; j<start+n_per_thread_-1; j++)
          new[j] = (val[j-1] + val[j+1]) / 2.0;
      swap(new, val);
      barrier.wait();            // Complete barrier
  }
}
```


Split-Phase Barrier

```
class Barrier {
  int nThreads;
  int count = 0;
  pthread_mutex_t lock;
  pthread_cond_t all_here;

public:
  Barrier (int t);
  ~Barrier (void);

  // Initiate barrier
  void arrive (void);

  // Check if done
  int done (void);

  // Block until done
  void wait (void);
};
```

```
int Barrier::done (void) {
  int rval;
  pthread_mutex_lock (&lock);

  // Done if the count is zero
  rval = !count;

  pthread_mutex_unlock (&lock);
  return rval;
};
```

Split-Phase Barrier

```
void Barrier::arrive (void) {
  pthread_mutex_lock (&lock);

  count++;              // Another thread has arrived
  // If last thread arrives, then wake up any waiters
  if (count==nThreads) {
     pthread_cond_broadcast (&all_here);
     count = 0;
  }
  pthread_mutex_unlock (&lock);
};
```

```
void Barrier::wait (void) {
  pthread_mutex_lock (&lock);

  // If not done, then wait
  if (count!=0)
     pthread_cond_wait (&all_here, &lock);

  pthread_mutex_unlock (&lock);
};
```

This code will not work!

What's wrong?

Deadlock

	Thread 0	Thread 1	count
			0
	`barrier.arrive()`		1
		`barrier.arrive()`	0
time		`barrier.wait()`	
		`barrier.arrive()`	1
	`barrier.wait()`		

Both threads are now in **`pthread_cond_wait()`**,
so we have deadlock

What caused the problem?
Two different instances of the barrier share the same value of **`count`**

Split-Phase Barrier—Revisited

Keep track of the current **phase**. The **`arrive()`** method returns the current phase, which is then passed into the **`done()`** and **`wait()`** methods

```
class Barrier {
  int nThreads;
  int count;
  int phase;     // current phase
  pthread_mutex_t lock;
  pthread_cond_t all_here;

public:
  Barrier (int t);
  ~Barrier (void);
  // Initiate barrier and return phase
  int arrive (void);
  // Check if phase p is done
  int done (int phase);
  // Block until phase p is done
  void wait (int phase);
};
```


Split-Phase Barrier—Revisited (cont)

```
int Barrier::arrive (void) {
  int p;
  pthread_mutex_lock (&lock);

  p = phase;       // Get phase
  count++;         // Another thread has arrived

  // If last thread to arrive,
  // then wake up any waiters and go to next phase
  if (count==nThreads) {
     pthread_cond_broadcast (&all_here);
     count = 0;
     phase = 1 - phase;
  }

  pthread_mutex_unlock (&lock);
  return p;
};
```

Split-Phase Barrier—Revisited (cont)

```
int Barrier::done (int p) {
  int rval;
  pthread_mutex_lock (&lock);

  // Done if phase has changed
  rval = (phase != p);

  pthread_mutex_unlock (&lock);
  return rval;
};
```

```
void Barrier::wait (int p) {
  pthread_mutex_lock (&lock);

  // If not done, then wait
  while (p==phase)
     pthread_cond_wait (&all_here, &lock);

  pthread_mutex_unlock (&lock);
};
```


Deadlock Problem Resolved

	Thread 0	Thread 1	count	phase
			0	0
time	barrier.arrive()		1	
		barrier.arrive()		
		barrier.wait(0)	0	1
		barrier.arrive()		
		barrier.wait(1)	1	
	barrier.wait(0)			

Deadlock does not occur: Thread 0's call to **`barrier.wait(0)`** returns without waiting.

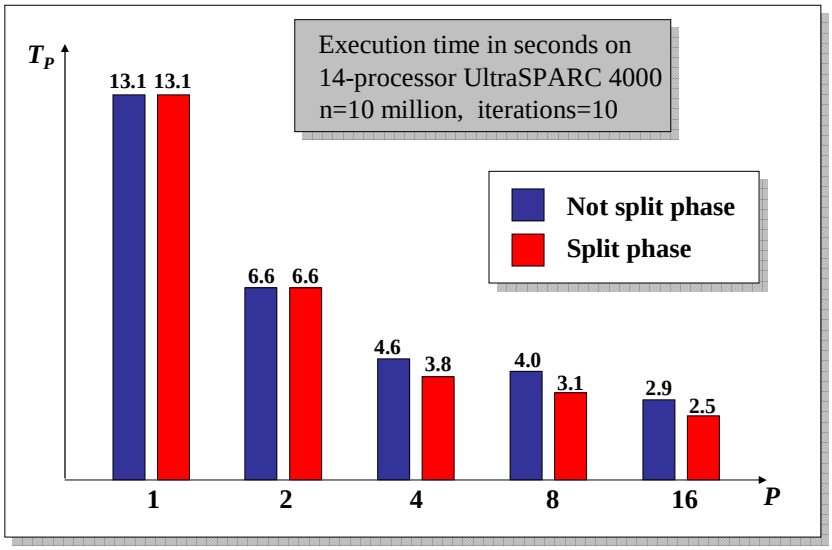
We can now distinguish between the first barrier and the second barrier

Relaxation Revisited

```
thread_main(int self) {
  int phase
  ...
  // For each iteration
  for (int i=0; i<iterations; i++) {
      // Update neighbor values
      int j = start;
      val[j] = (val[j-1] + val[j+1]) / 2.0;
      j = start + n_per_thread –1;
      val[j] = (val[j-1] + val[j+1]) / 2.0;

      phase = barrier.arrive(); // Inititate barrier

      // Update local values
      for (j=start+1; j<start+n_per_thread_-1; j++)
          new[j] = (val[j-1] + val[j+1]) / 2.0;
      swap(new, val);
      barrier.wait(phase);      // Complete barrier
  }
}
```

The logic of our code is more complex now

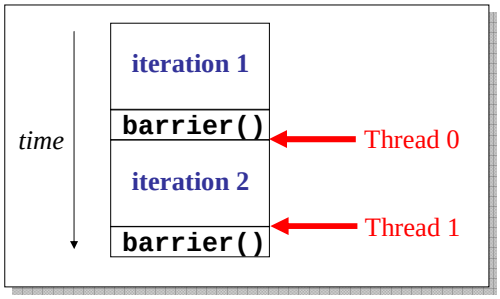
Relaxation Performance



Split Phase Operations

Relaxed synchronization

– With standard barriers, how far out of synch can two threads become?

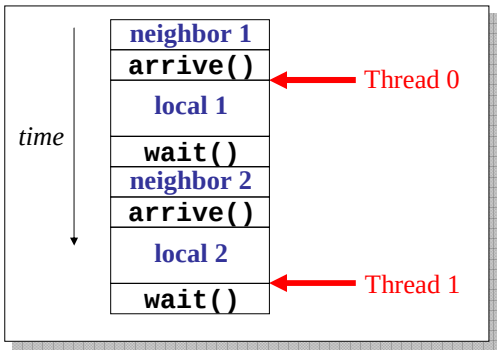


– With split-phase barriers, how far out of synch can two threads become?

Split Phase Operations (cont)

Relaxed synchronization

– About 1.5 iterations. Looser synchronization ⇒ improved performance



Split Phase Operations (cont)

Other examples of split-phase operations?

- Split-phase message receive
- Split-phase scans and reductions
- Asynchronous I/O
- Asynchronous RPC
- Speculative loads in the IA64
- . . .

} **Increase parallelism**