Another synchronisation primitive is *barriers* (occasionally called *rendezvous*)
Another synchronisation primitive is barriers (occasionally called rendezvous)

A barrier stops threads from continuing until some required number of threads have all hit the barrier; then they can all continue together
Another synchronisation primitive is *barriers* (occasionally called *rendezvous*)

A barrier stops threads from continuing until some required number of threads have all hit the barrier; then they can all continue together

This allows us to synchronise parts of the program: recall supersteps
Suppose we have a list of numbers we want to square then add in pairs

```c
for (i = 0; i < 100; i++) {
    v[i] = v[i] * v[i];
}
for (i = 0; i < 100; i++) {
    s[i] = v[i] + v[99-i];
}
```
Suppose we have a list of numbers we want to square then add in pairs

```c
for (i = 0; i < 100; i++) {
    v[i] = v[i]*v[i];
}
```

```c
for (i = 0; i < 100; i++) {
    s[i] = v[i] + v[99-i];
}
```

We can parallelise this by having (say) 4 threads; each thread squares a block of values; then they add a block of values
Concurrency Primitives

Barriers

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>[0]^2</td>
<td>v[25]^2</td>
<td>v[50]^2</td>
<td>v[75]^2</td>
</tr>
<tr>
<td>v</td>
<td>[1]^2</td>
<td>v[26]^2</td>
<td>v[51]^2</td>
<td>v[76]^2</td>
</tr>
<tr>
<td>v</td>
<td>[2]^2</td>
<td>v[27]^2</td>
<td>v[52]^2</td>
<td>v[77]^2</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>v</td>
<td>[24]^2</td>
<td>v[49]^2</td>
<td>v[74]^2</td>
<td>v[99]^2</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
Concurrenc Primitives

Barriers

1
for (i = 0; i < 25; i++) {
    v[i] = v[i] * v[i];
}

2
for (j = 25; j < 50; j++) {
    v[j] = v[j] * v[j];
}

3...
for (i = 0; i < 25; i++) {
    s[i] = v[i] + v[99 - i];
}
for (j = 25; j < 50; j++) {
    s[j] = v[j] + v[99 - j];
}

Again, the above might work sometimes, or many times, but it is buggy.
Again, the above might work sometimes, or many times, but it is buggy
The problem here is again that the threads may not all be running at the same speed: perhaps one thread is interrupted and suspended by the OS; or memory access is not uniform speed; or many other factors
Concurrency Primitives

Barriers

The problem here is again that the threads may not all be running at the same speed: perhaps one thread is interrupted and suspended by the OS; or memory access is not uniform speed; or many other factors.

So we can’t rely on all the threads finishing their squares at precisely the same time: one thread might finish its block and start adding using values not yet finished squaring.
The problem here is again that the threads may not all be running at the same speed: perhaps one thread is interrupted and suspended by the OS; or memory access is not uniform speed; or many other factors.

So we can’t rely on all the threads finishing their squares at precisely the same time: one thread might finish its block and start adding using values not yet finished squaring.

Another synchronisation problem.
This is how we get the wrong answer: again just because the lines of code for the adds follows the lines of code for the squares make us believe every add happens after every square.
Concurrentey Primitives

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v[0]^2$</td>
<td>$v[25]^2$</td>
<td>$v[50]^2$</td>
<td></td>
</tr>
<tr>
<td>$v[1]^2$</td>
<td>$v[26]^2$</td>
<td>$v[51]^2$</td>
<td></td>
</tr>
<tr>
<td>$v[2]^2$</td>
<td>$v[27]^2$</td>
<td>$v[52]^2$</td>
<td>$v[75]^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>$v[76]^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>$v[75] + v[24]$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$v[98] + v[1]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$v[99] + v[0]$</td>
</tr>
</tbody>
</table>

This is how we get the wrong answer: again just because the lines of code for the adds follows the lines of code for the squares make us believe every add happens after every square
Concurrent Primitives

Barriers

We need to synchronise all the threads at the end of the squares before allowing them to continue with the adds

\[ b = \text{make\_barrier}(4); \]

\(<\text{parallel squares}> <\text{parallel squares}> <\text{parallel squares}> \ldots \]

\( \text{barrier\_wait}(b); \text{barrier\_wait}(b); \text{barrier\_wait}(b); \ldots \)

\(<\text{parallel adds}> <\text{parallel adds}> <\text{parallel adds}> \ldots \)

Only when all 4 threads have reached the barrier can they all proceed
Barriers are good for the superstep style of programming
Concurrency Primitives

Barriers

Barriers are good for the superstep style of programming

But beware: as a barrier synchronises many threads, there is potentially a lot of waiting going on: we can’t progress faster than the slowest thread
Barriers are good for the superstep style of programming

But beware: as a barrier synchronises many threads, there is potentially a lot of waiting going on: we can’t progress faster than the slowest thread

Thus barriers are best when all the threads are doing roughly the same amount of work
Concurrency Primitives
POSIX Barriers

#include <pthread.h>
pthread_barrier_t barrier;
int pthread_barrier_init(
    pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr,
    unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);

A barrier can be reused immediately after it has released its threads; it has a fixed value of $n$ set when it is initialised.
Concurrency Primitives
POSIX Barriers

#include <pthread.h>

pthread_barrier_t barrier;

int pthread_barrier_init(
    pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr,
    unsigned count);

int pthread_barrier_destroy(pthread_barrier_t *barrier);

int pthread_barrier_wait(pthread_barrier_t *barrier);

A barrier can be reused immediately after it has released its threads; it has a fixed value of \( n \) set when it is initialised.

Exercise. Have a look at the return value from pthread_barrier_wait.
Exercise. Fix the $\text{count1}/\text{count2}$ problem with barriers.

Exercise. Both semaphores and barriers are about synchronisation. Think about how you might implement barriers using semaphores.

Exercise. Think about how you might implement semaphores using barriers.