

sync-barriers

barriers

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU

one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

barriers

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU

one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

barriers API

`barrier.Initialize(NumberOfThreads)`

`barrier.Wait()` — return after all threads have waited

idea: multiple threads perform computations in parallel

threads wait for *all other threads* to call `Wait()`

barrier: waiting for finish

```
barrier.Initialize(2);
```

Thread 0:

```
partial_mins[0] =  
    /* min of first  
       50M elems */;
```

```
barrier.Wait();
```

```
total_min = min(  
    partial_mins[0],  
    partial_mins[1]  
);
```

Thread 1:

```
partial_mins[1] =  
    /* min of last  
       50M elems */  
barrier.Wait();
```

barriers: reuse

```
results[0][0] = getInitial(0);  
barrier.Wait();
```

```
results[1][0] =  
    computeFrom(0,  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][0] =  
    computeFrom(0,  
        results[1][0],  
        results[1][1]  
    );
```

```
results[0][1] = getInitial(1);  
barrier.Wait();
```

```
results[1][1] =  
    computeFrom(1,  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][1] =  
    computeFrom(1,  
        results[1][0],  
        results[1][1]  
    );
```

barriers: reuse

```
results[0][0] = getInitial(0);  
barrier.Wait();
```

```
results[1][0] =  
    computeFrom(0,  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][0] =  
    computeFrom(0,  
        results[1][0],  
        results[1][1]  
    );
```

```
results[0][1] = getInitial(1);  
barrier.Wait();
```

```
results[1][1] =  
    computeFrom(1,  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][1] =  
    computeFrom(1,  
        results[1][0],  
        results[1][1]  
    );
```

barriers: reuse

```
results[0][0] = getInitial(0);  
barrier.Wait();
```

```
results[1][0] =  
    computeFrom(0,  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][0] =  
    computeFrom(0,  
        results[1][0],  
        results[1][1]  
    );
```

```
results[0][1] = getInitial(1);  
barrier.Wait();
```

```
results[1][1] =  
    computeFrom(1,  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][1] =  
    computeFrom(1,  
        results[1][0],  
        results[1][1]  
    );
```

pthread barriers

```
pthread_barrier_t barrier;  
pthread_barrier_init(  
    &barrier,  
    NULL /* attributes */,  
    numberOfThreads  
);  
...  
...  
pthread_barrier_wait(&barrier);
```

exercise

```
pthread_barrier_t barrier; int x = 0, y = 0;
void thread_one() {
    y = 10;
    pthread_barrier_wait(&barrier);
    y = x + y;
    pthread_barrier_wait(&barrier);
    pthread_barrier_wait(&barrier);
    printf("%d %d\n", x, y);
}
void thread_two() {
    x = 20;
    pthread_barrier_wait(&barrier);
    pthread_barrier_wait(&barrier);
    x = x + y;
    pthread_barrier_wait(&barrier);
}
```

output? (if both run at once, barrier set for 2 threads)

life homework (pseudocode)

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {  
    for (int y = 0; y < size; ++y) {  
        for (int x = 0; x < size; ++x) {  
            to_grid(x, y) = computeValue(from_grid, x, y);  
        }  
    }  
    swap(from_grid, to_grid);  
}
```

life homework

compute grid of values for time t from grid for time $t - 1$ parallel version: produce parts of grid in different threads use barriers to finish time t before going to time $t + 1$

Backup slides

life homework even/odd

naive way has an operation that needs locking:

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {  
    ... compute to_grid ...  
    swap(from_grid, to_grid);  
}
```

but this alternative needs less locking:

```
Grid grids[2];  
for (int time = 0; time < MAX_ITERATIONS; ++time) {  
    from_grid = &grids[time % 2];  
    to_grid = &grids[(time % 2) + 1];  
    ... compute to_grid ...  
}
```

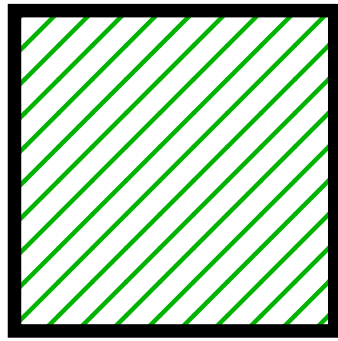
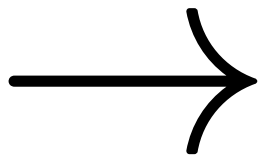
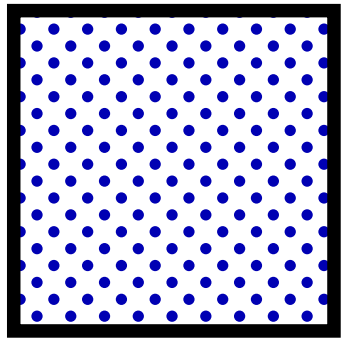
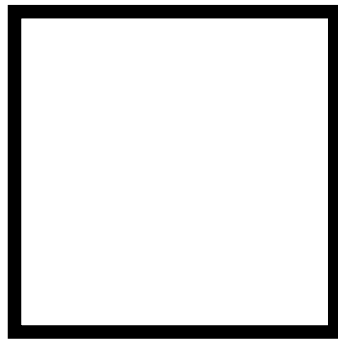
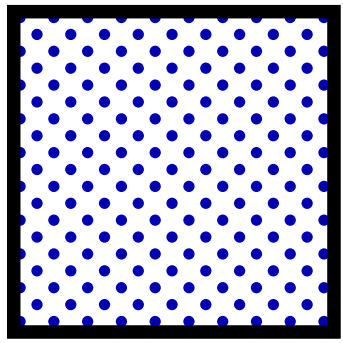
life homework even/odd

naive way has an operation that needs locking:

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {  
    ... compute to_grid ...  
    swap(from_grid, to_grid);  
}
```

but this alternative needs less locking:

```
Grid grids[2];  
for (int time = 0; time < MAX_ITERATIONS; ++time) {  
    from_grid = &grids[time % 2];  
    to_grid = &grids[(time % 2) + 1];  
    ... compute to_grid ...  
}
```



swap

