



- **OpenMP-parallel raytracer: Parallelize outer loop**

```
#pragma omp parallel private(tile)
{
    tile=(char*)malloc(tilesizetilesizesizeof(char))

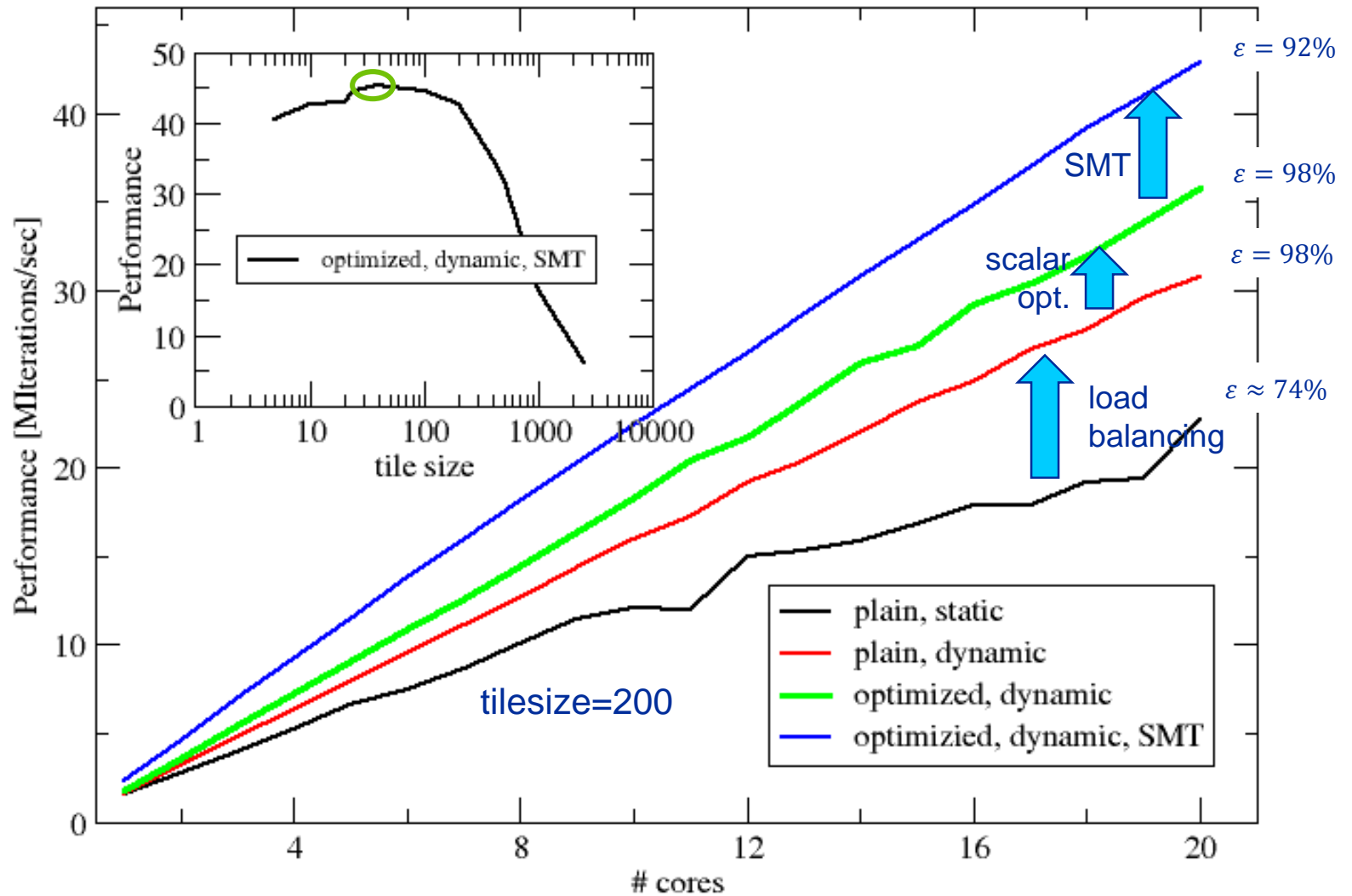
#pragma omp for private(xc,i) schedule(runtime) collapse(2)
    for(yc=0; yc<ytiles; yc++)
        for(xc=0; xc<xtiles; xc++) {
            /* calc one tile */
            calc_tile(size, xc*tilesizetilesizesizeof(char), yc*tilesizetilesizesizeof(char), tilesizetilesizesizeof(char), tile);
            /* copy to picture buffer */
            for(i=0; i<tilesizetilesizesizeof(char); i++) {
                tilebase=yc*tilesizetilesizesizeof(char)*xtiles+xc*tilesizetilesizesizeof(char);
                memcpy((void*)(picture+tilebase+i*tilesizetilesizesizeof(char)),
                    (void*)(tile+i*tilesizetilesizesizeof(char)),
                    tilesizetilesizesizeof(char));
            }
        }
    }
}
```



- **Avoiding FP divides in `shade()` and `calc_tile()` yields about 15% speedup**

```
r = 1./sqrt(nx * nx + ny * ny + nz * nz);
nx *= r; ny *= r; nz *= r;
...
r = sqrt(ldx * ldx + ldy * ldy + ldz * ldz);
rr = 1./r;
ldx *= rr; ldy *= rr; ldz *= rr;
...
r = 1./sqrt(dx * dx + dy * dy + dz * dz);
c = 100 * shade(2.1, 1.3, 1.7, dx * r, dy * r, dz * r, 0);
```

OpenMP-parallel ray tracer performance on Ivy Bridge 2.2 GHz (Emmy)



Is it memory bound?



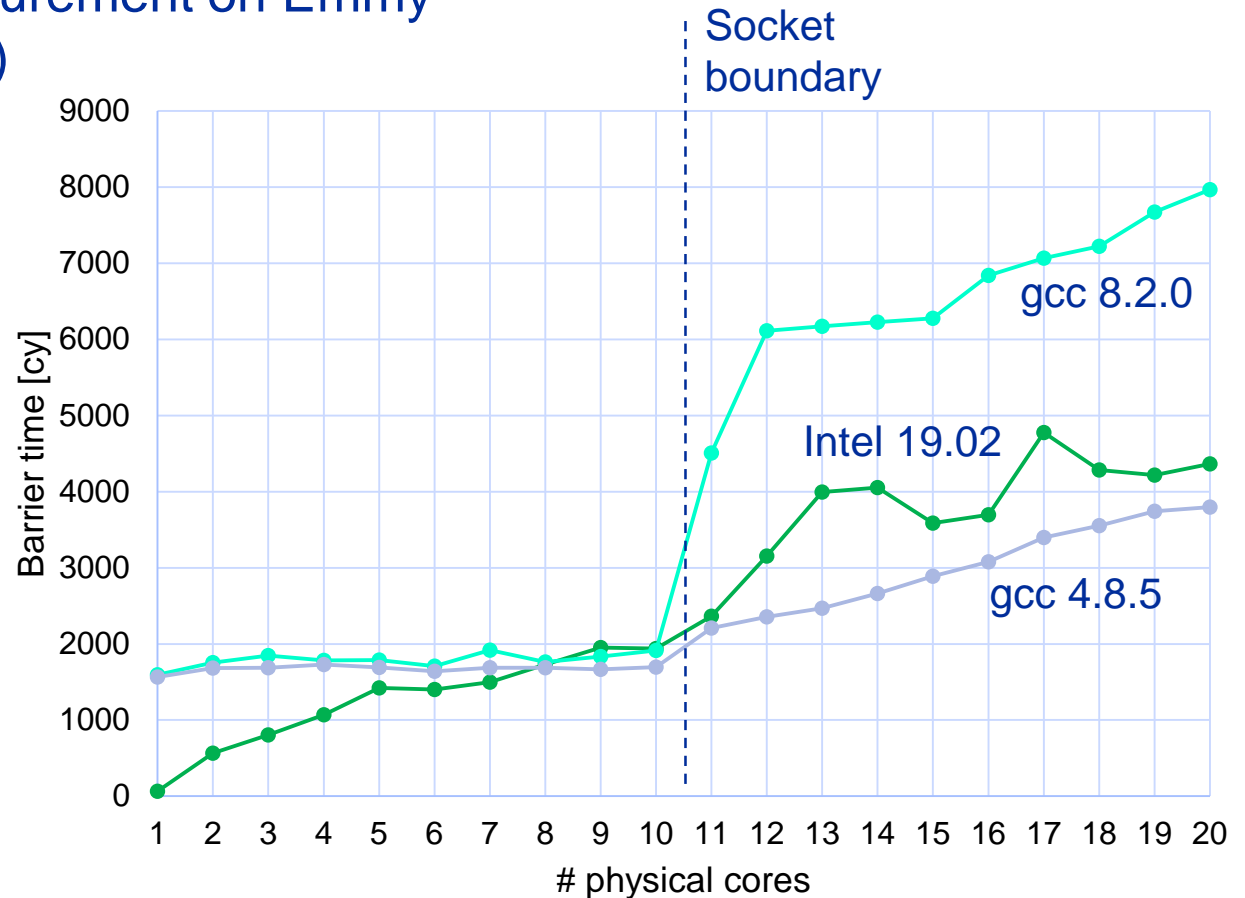
- Measured performance: ~20-40 MPixels/s
- Every pixel is one byte that has to be read from memory and written back
- The memory bandwidth caused by the code is thus about 40-80 Mbyte/s

- The available bandwidth on an Emmy socket is 40 Gbyte/s, so we are far away from saturation → this code is not limited by memory bandwidth.

Assignment 7 – Task 2



a) Barrier measurement on Emmy (Intel vs. gcc)



Intel19, 40 threads (incl. SMT): 5500-6500 cy

Assignment 7 – Task 2



b) Barrier overhead for

- IVB dual-socket (40 threads): 5000 cy
- Intel Xeon Phi (240 threads): 18000 cy

That does not look bad for Xeon Phi. Or does it???

3 x cores (20 vs 60) on Phi

2 x more operations per cycle per core on Phi (16 F/cy instead of 8 F/cy)

→ 6 x more work done on full Xeon Phi per cycle

3.6 x more barrier penalty (cycles) on Phi

→ One barrier causes $3.6 \times 2 \times 3 \sim 22$ x more “pain” 😊.



- c) Performance prediction: Schönauer triad on Emmy, N=500, 10 threads on 10 physical cores

$$A(:) = B(:) + C(:) * D(:)$$

Pure execution in L1 cache:

Assuming AVX, IVB core: 1 LD + ½ ST per cycle

→ 4 it / 3 cy (= 5.83 Gflop/s @ 2.2 GHz)

→ 500 it. on 10 cores: $500 \text{ it} / (10 * 4 \text{ it} / 3 \text{ cy}) = 37.5 \text{ cy}$

Including barrier overhead:

Barrier on 10 cores: 2000 cy

→ Overall time: $T = 2037.5 \text{ cy}$

→ Performance:

$$P = 1000 \text{ flops} / 2037.5 \text{ cy} * 2.2 \text{ Gcy/s} = 1.08 \text{ Gflop/s}$$

(instead of 58.3 Gflop/s)

Assignment 7 – Task 3:

OpenMP histogram



- Compute simplified histogram of a (integer) random number generator:

```
hist[rand() % 16]
```

- Check if `rand()` generates a homogeneous distribution:

```
hist[rand() % 16] = N/16
```

(N: random numbers generated)

- Architecture: Intel Xeon/Sandy Bridge 2.7 GHz (fixed clock speed)
- Compiler: Intel 13.1 (no inlining)
- Simple Random number generator (taken from `man rand`; there are much better ones...)

```
int myrand(unsigned long* next)
{
    *next = *next * 1103515245 + 12345;
    return((unsigned)(*next/65536) % 32768);
}
```




Computation

```
lseed = 123;  
for(i = 0; i < 16; ++i)  
    hist[i] = 0;
```

```
timing(&wcstart, &ct);
```

```
for(i = 0; i < n_loop; ++i)  
    hist[RAND & 0xf]++;
```

```
timing(&wcend, &ct);
```

- **Serial baselines (N=10⁹)**

RAND = myrand(&lseed)

Time = 3.6 s

abserr = 3 * 10⁻⁶

Quality evaluation

```
double av = n_loop / 16.0;  
double abserr = 0.0;  
double err;
```

```
for(i = 0; i < 16; ++i) {  
    err = (hist[i] - av) / av;  
    abserr = MAX(fabs(err), abserr);  
}
```

Standard thread-safe random
number generator



RAND = rand_r(&lseed)

Time = 6.7 s

abserr = 4 * 10⁻⁶

Straightforward parallelization?!



- Just add a single OpenMP directive.....

Result Quality

Threads	abserr
2	~0.38
4	~0.61
8	~0.80
16	~0.89

Baseline: $3 \cdot 10^{-6}$

Performance

Threads	Time
2	~20s
4	~23s
8	~28s
16	~105s

Baseline: 3.6s

```
lseed = 123;
for(i = 0; i < 16; ++i)
    hist[i] = 0;

timing(&wcstart, &ct);
```

```
#pragma omp parallel for
for(i =0; i < n_loop; ++i) {
    hist[myrand(&lseed) & 0xf]++;
}
```

```
timing(&wcend, &ct);
```

Problem: Uncoordinated concurrent updates of `hist[]` and `lseed`
→ Runtime and result changes between runs

Get it correct first!



- Protect update of `lseed` and `hist[]` by critical region

Result Quality

Threads	abserr
2	$3 * 10^{-6}$
4	$3 * 10^{-6}$
8	$3 * 10^{-6}$
16	$3 * 10^{-6}$

Baseline: $3 * 10^{-6}$

```
#pragma omp parallel for
for(i=0; i<n_loop; ++i) {

    #pragma omp critical
    hist[myrand(&lseed) & 0xf]++;

}
```

Performance

Threads	Time
2	201s
4	221s
8	217s
16	427s

Baseline: 3.6s

Result Quality: OK

Problem: Performance: ~50x-100x slower!
Serialization and some (?) more overhead,
e.g. “synchronization”

Avoid complete serialization



- Define a private `lseed` and `value`
- Only histogram update needs a `#pragma omp critical`

Result Quality

Threads	abserr
2	$6 * 10^{-6}$
4	$15 * 10^{-6}$
8	$24 * 10^{-6}$
16	$60 * 10^{-6}$

Baseline: $3 * 10^{-6}$

Performance

Threads	Time
2	191s
4	201s
8	194s
16	413s

Baseline: 3.6s

```
#pragma omp parallel for \  
  firstprivate(lseed) private(value)  
for(i = 0; i < n_loop; ++i) {  
  value = myrand(&lseed) & 0xf;  
  
  #pragma omp critical  
  hist[value]++;  
  
}
```

Problem: Performance improves only marginally → `critical` is still an issue!

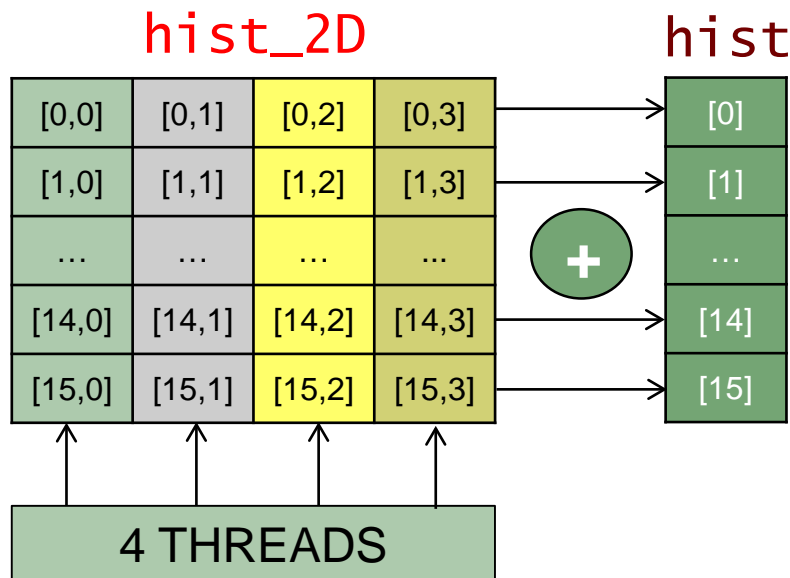
Problem (?): Result Quality is slightly worse than baseline.

Get rid of the `critical` statement (1)



Use a shared scoreboard (`hist_2D`):

- Each thread writes to a separate column of length 16
- Sum up the numbers across each row to get the final `hist[]`



```
// additional shared array
// assuming 4 threads
hist_2D[16][4] = { 0 };
```

```
#pragma omp parallel
{
    int tId = omp_get_num_threads();
```

```
#pragma omp for \
    firstprivate(lseed) private(value)
for(i = 0; i < n_loop; ++i) {
    value = myrand(&lseed) & 0xf;
    hist_2D[value][tID]++;
}
```

```
#pragma omp critical
for (i = 0; i < 16; ++i)
    hist[i] += hist_2D[i][tID];
}
```

Get rid of the `critical` statement (2)



Result Quality

Threads	abserr
2	$6 * 10^{-6}$
4	$15 * 10^{-6}$
8	$24 * 10^{-6}$
16	$60 * 10^{-6}$

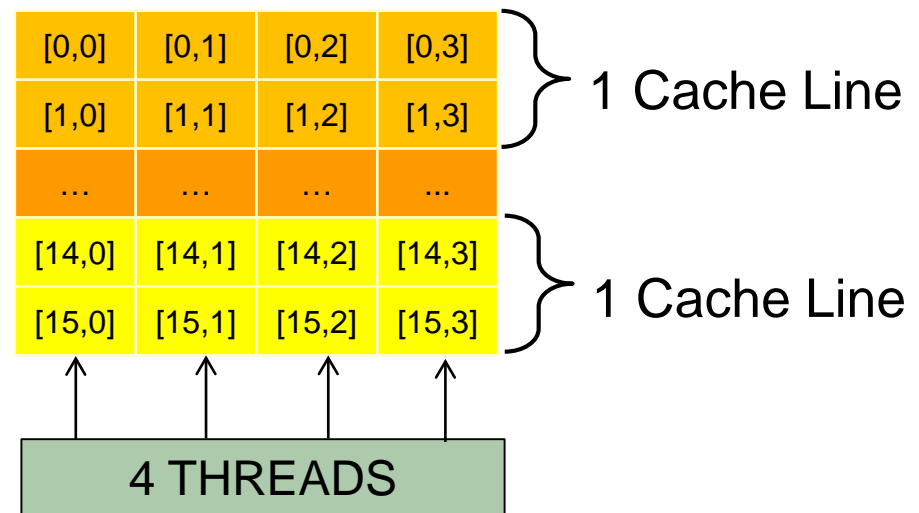
Baseline: $3 * 10^{-6}$

Performance

Threads	Time
2	11.7s
4	9.3s
8	6.6s
16	19.3s

Baseline: 3.6s

Performance improves 30x but still much slower than serial version ?!



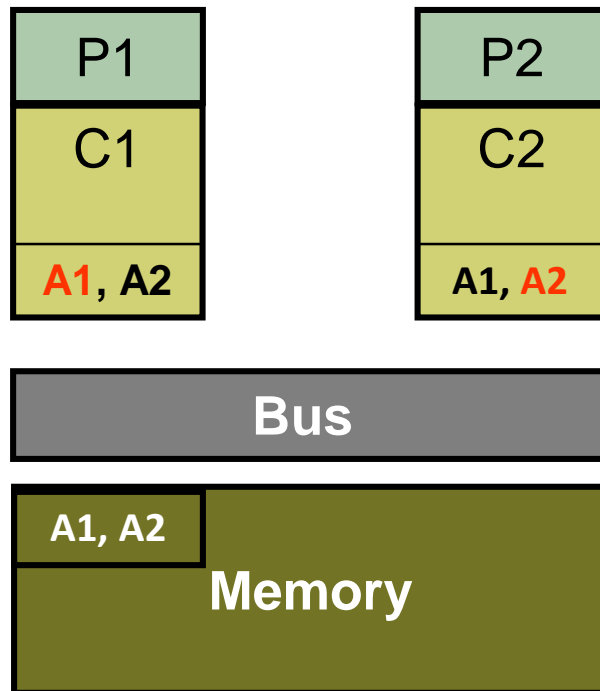
Each thread writes frequently to every cache line of `hist_2D`

→ False Sharing



- **Data in cache is only a copy of data in memory**

- Multiple copies of same data on multiprocessor systems
- Cache coherence protocol/hardware ensure consistent data view
- Without cache coherence, shared cache lines can become clobbered:
(Cache line size = 2 WORD; A1+A2 are in a single CL)



P1

Load A1
Write A1=0

P2

Load A2
Write A2=0

Write-back to memory leads to incoherent data

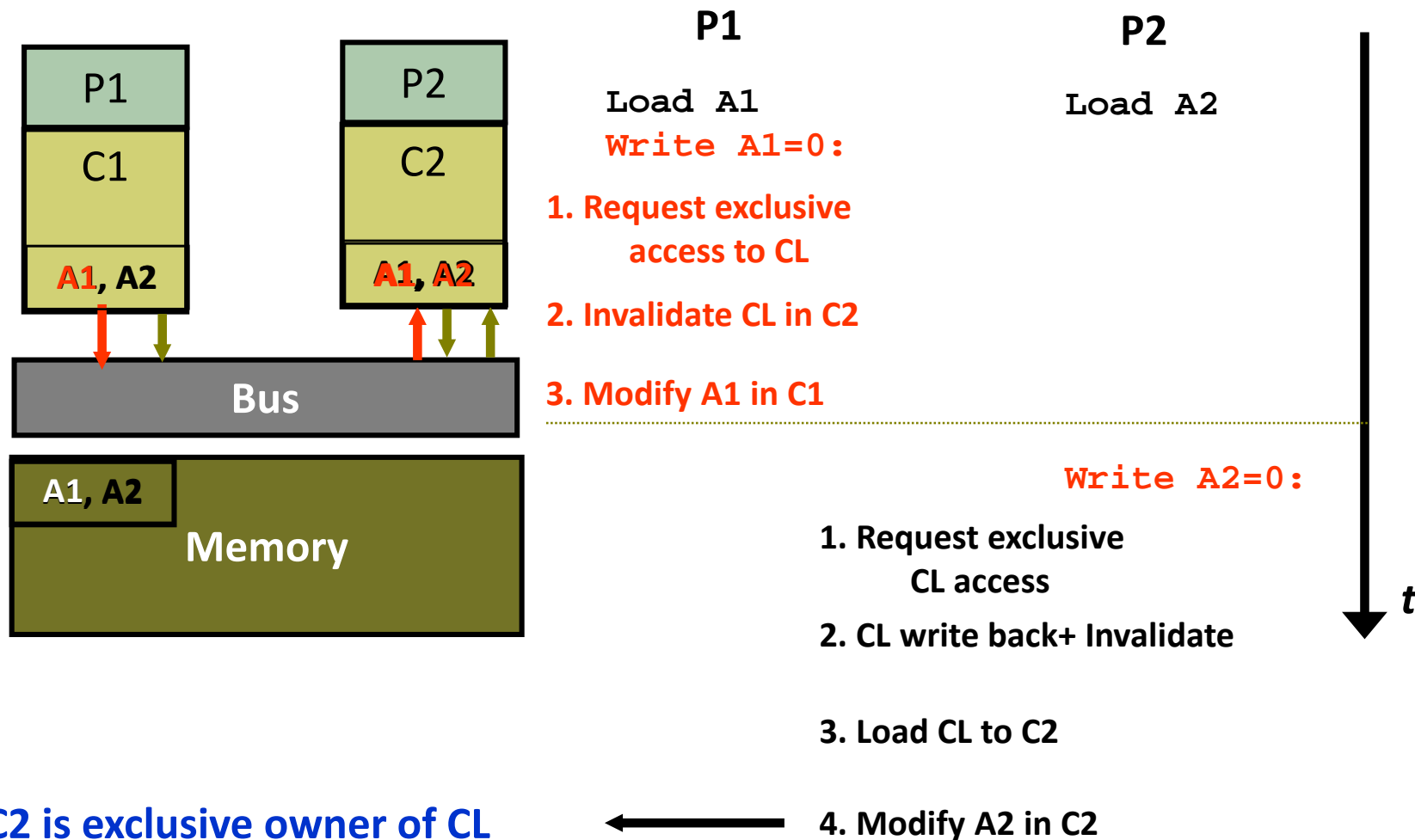


C1 & C2 entry can not be merged to:





- Cache coherence protocol must keep track of cache line status



Avoid False Sharing



Use thread private histogram (`hist_local[16]`) for thread local computation & sum up all results at the end

Result Quality

Threads	abserr
2	$6 * 10^{-6}$
4	$15 * 10^{-6}$
8	$24 * 10^{-6}$
16	$60 * 10^{-6}$

Baseline: $3 * 10^{-6}$

Performance

Threads	Time
2	1.78s
4	0.89s
8	0.44s
16	0.22s



Baseline: 3.6s

```
#pragma omp parallel
{
    int hist_local[16] = { 0 };

    #pragma omp for \
        firstprivate(lseed) private(value)
    for(i = 0; i < n_loop; ++i) {
        value = myrand(&lseed) & 0xf;
        hist_local[value]++;
    }

    #pragma omp critical
    for (i = 0; i < 16; ++i)
        hist[i] += hist_local[i];
}
```

Performance: OK now – nice scaling

PROBLEM: Quality still gets worse as number of threads increase?!

Every thread does the same (`lseed` is the same!)

○ → more threads less statistics



Use different seeds for each thread!

Result Quality

Threads	abserr
2	$4 * 10^{-6}$
4	$7 * 10^{-6}$
8	$10 * 10^{-6}$
16	$10 * 10^{-6}$

Baseline: $3 * 10^{-6}$

Performance

Threads	Time
2	1.78s
4	0.89s
8	0.44s
16	0.22s

Baseline: 3.6s

```
#pragma omp parallel
{
    int hist_local[16] = { 0 };
    int myseed;
    #pragma omp critical
    myseed = myrand(&seed);

    #pragma omp for private(value)
    for(i = 0; i < n_loop; ++i) {
        value = myrand(&myseed) & 0xf;
        hist_local[value]++;
    }

    #pragma omp critical
    for (i = 0; i < 16; ++i)
        hist[i] += hist_local[i];
}
```

Result quality is slightly worse - we are doing different things than in the serial version.....

Conclusions from the histogram example



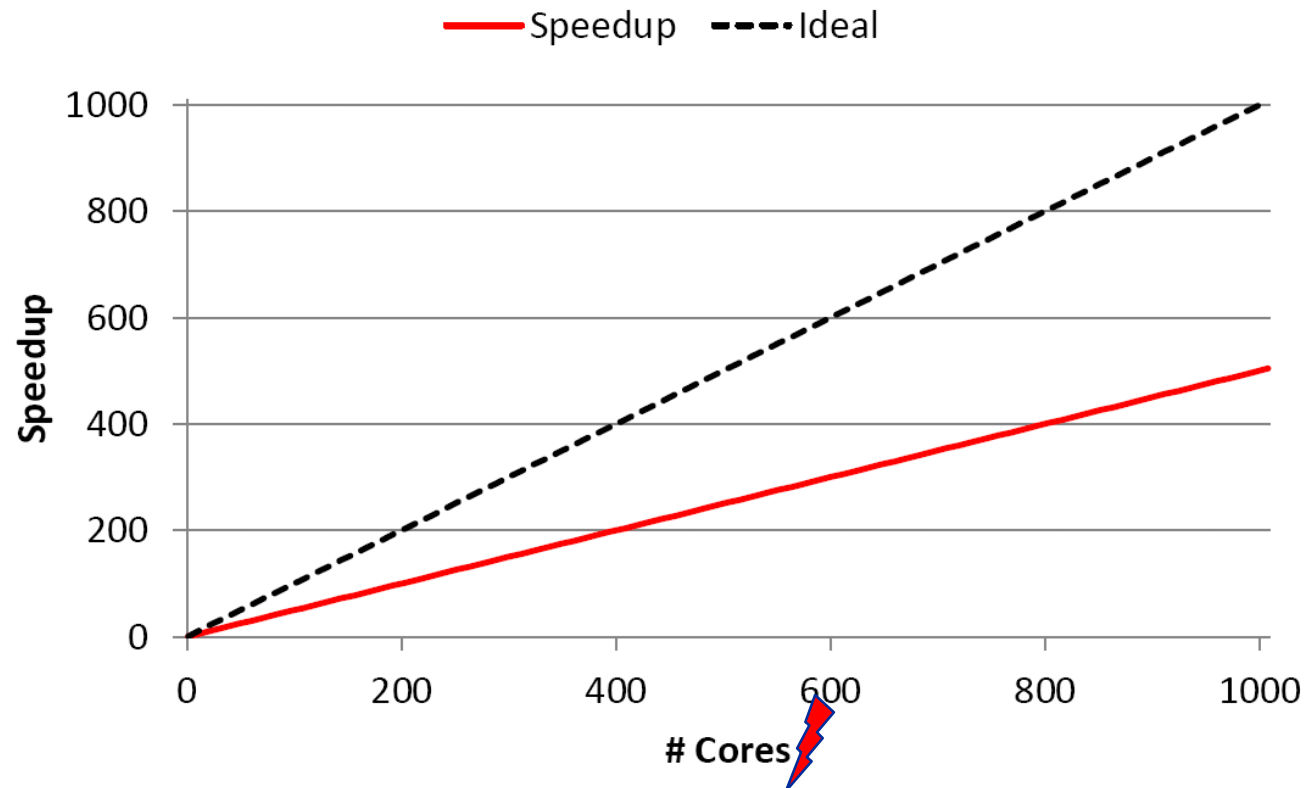
- **Get it correct first!**
 - Race conditions, deadlocks...
- **Avoid complete serialization**
 - Thread-local data
- **Avoid false sharing**
 - Proper shared array layout
 - Padding
- **Parallel random numbers may be non-trivial**

Assignment 7 – Task 4



- Natural question: **What is the scaling behavior inside one multicore socket?**
- Plotting speedup vs. cores is not useful in most cases since the intra-socket scaling may be limited by other factors than the inter-node scaling

- Think **“scaling baseline”!**



Scaling baselines



- Example: sparse matrix-vector multiply on a 4-socket AMD system (6 cores/socket)
- Typical pattern for mem-bound code and compact pinning

