The problem

- 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); }
```

Serial implementation and baseline

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=10^9)**

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: $3*10^{-6}$

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: $3*10^{-6}$

Performance

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

Baseline: 3.6s

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

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

}
```

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]++;

}
```

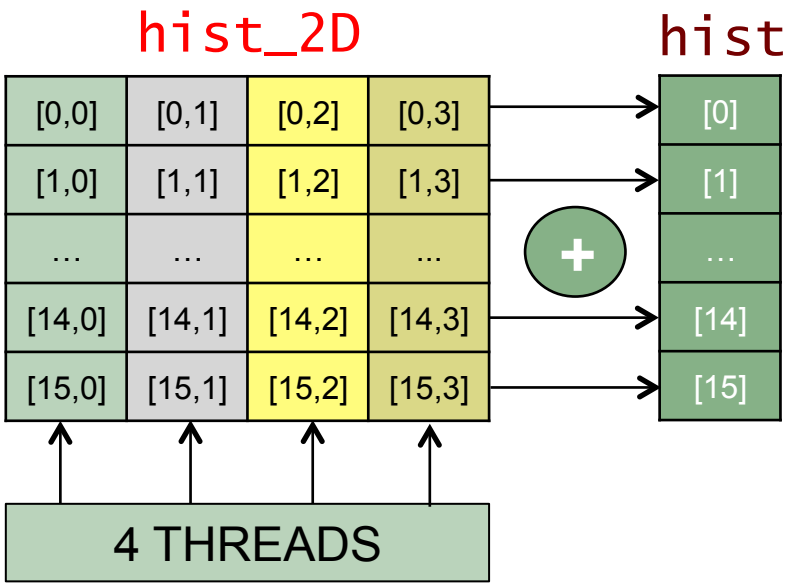
Problem: Performance improves only marginally → critical is still an issue!

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	$60 * 10^{-6}$

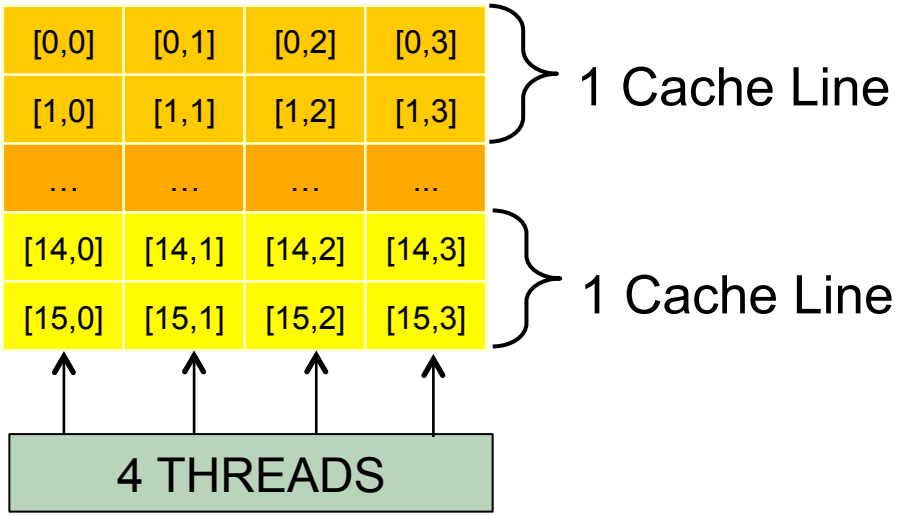
Baseline: $3*10^{-6}$

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 ?!

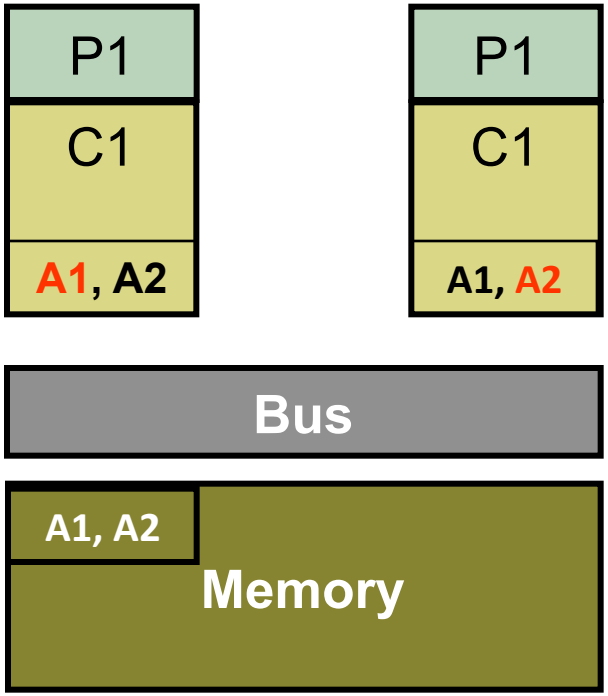


Each thread writes frequently to every cache line of `hist_2D`

→ False Sharing

Excursion:
Cache coherence protocol → 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	P2
Load A1	Load A2
Write A1=0	Write A2=0

Write-back to memory leads to incoherent data

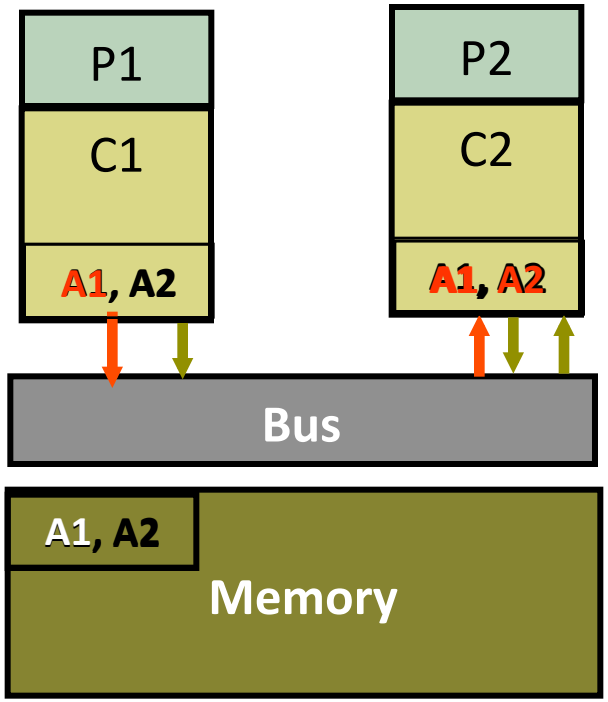
A1, A2	A1, A2	A1, A2

C1 & C2 entry can not be merged to:

A1, A2

Excursion:
Cache coherence protocol → False Sharing

- **Cache coherence protocol must keep track of cache line status**



P1	P2
`Load A1`	`Load A2`
`Write A1=0:`	
1. Request exclusive access to CL	
2. Invalidate CL in C2	
3. Modify A1 in C1	
	`Write A2=0:`
	1. Request exclusive CL access
	2. CL write back+ Invalidate
	3. Load CL to C2
	4. Modify A2 in C2

t

C2 is exclusive owner of CL

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

Improve Result Quality

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**