The problem: 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);
}
```

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);
```

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);
}
```

- **Serial baselines (N=10^9)**

RAND = myrand(&lseed)
Time = 3.6 s
abserr = $3 * 10^{-6}$

Standard thread-safe random number generator

RAND = rand_r(&lseed)
Time = 6.7 s
abserr = $4 * 10^{-6}$

© 2004-14 LRZ and RRZE

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.

© 2004-14 LRZ and RRZE

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];
}
```

© 2004-14 LRZ and RRZE

Get rid of the `critical` statement (2)

Result Quality

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

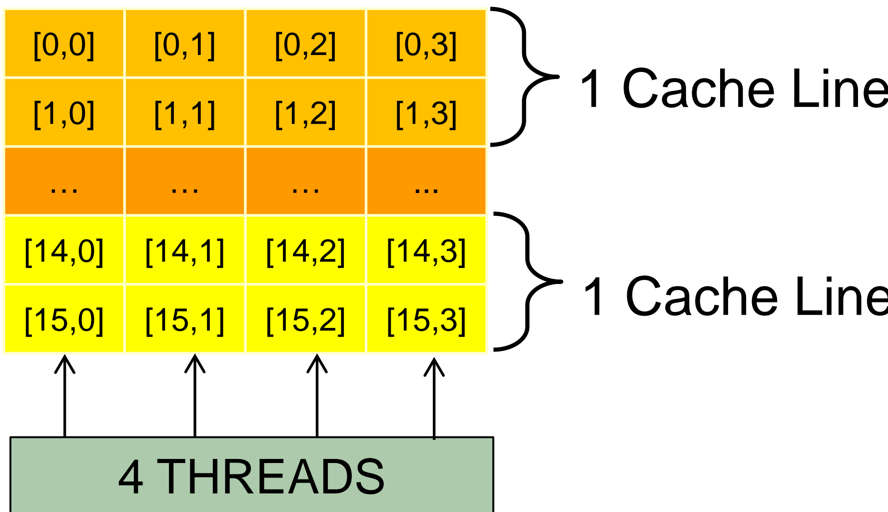
Baseline: $3*10^{-6}$

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

© 2004-14 LRZ and RRZE

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

© 2004-14 LRZ and RRZE

- 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 ←

© 2004-14 LRZ and RRZE

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

© 2004-14 LRZ and RRZE

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 10, Task 2:

PPP for Gauss-Seidel smoother

Principle in 3D ("pipeline parallel processing"):

T_0 T_1 T_2 . . . T_t

k j_{max}/t i j

W_1 W_2 W_3 W_7

2D:

k ↑

T0				
	T1			
		T2		
			T3	
				T4

→ i

Assignment 10, Task 2:
PPP for Gauss-Seidel smoother

Pipelined parallel processing via OpenMP

```
!$OMP parallel private(nthreads,istart,iend,tid,kk,it,k,i)
  nthreads = omp_get_num_threads()
  tid = omp_get_thread_num()
  istart= (imax-1)/nthreads * tid +1
  iend  = istart+(imax-1)/nthreads-1
  do it=1,itmax
     do k=1,kmax-1+nthreads-1
        kk = k - tid
        if(kk .ge. 1 .and. kk .le. kmax-1) then
           do i=istart, iend
              ! may also use pipeline-optimized version
              phi(i,kk) = 0.25d0 * ( phi(i,kk-1)+
                phi(i+1,kk) +  phi(i,kk+1) + phi(i-1,kk))
           enddo
        endif
!$OMP barrier
     enddo ! k
  enddo !it
!$OMP end parallel
```

PPP 2D Gauss-Seidel Performance model

- Bandwidth-bound performance limit on one Emmy socket:

  Layer condition: 3 rows must fit in 12 MB of L3 → imax <= 521000

  B_c = 16 B/LUP (read & write each lattice node)
  b_S = 41 GB/s

  → b_S / B_c = 2560 MLUP/s

- Impact of 2000 cy OpenMP barrier @ 8000x8000

  Duration of one barrier-free update sweep (8000 – 2 LUPs):

  T_{bf} = (7998 LUPs / 2560 MLUP/s) * 2.2 Gcy/s = 6870 cy

  → performance reduction by factor of
  6880/(2000+6880) = 0.77 → 1980 MLUP/s

PPP 2D Gauss-Seidel measurements

- Emmy node @ 2.2 GHz, grid size 8000x8000

- Performance saturates just barely w/ physical cores only
- SMT helps a lot below saturation (filling pipeline bubbles) but cannot circumvent BW saturation or barrier pain
- No scaling across ccNUMA domains
  - Barrier is **much** more costly
  - First-touch placement? NUMA balancing?

Assignment 10, Task 2:
Using the LIKWID Marker API

```
 call likwid_MarkerInit
 call likwid_MarkerRegisterRegion("ITER")
!...
 call likwid_MarkerStartRegion("ITER")
!$OMP parallel private(nthreads,istart,iend,tid,kk,it,k,i)
  nthreads = omp_get_num_threads()
  tid = omp_get_thread_num()
  istart= (imax-1)/nthreads * tid +1
  iend   = istart+(imax-1)/nthreads-1
  do it=1,itmax
! ... Sweep here
  enddo
!$OMP end parallel
call likwid_MarkerStopRegion("ITER")
!...
call likwid_MarkerClose
```

Optional – Reduces overhead for 1st marker call

Region for counting events

```
$ ifort -qopenmp ... $LIKWID_INC gs_opt_par.f90 timing.o \
   $LIKWID_LIB -llikwid
$ likwid-perfctr -g MEM -C S0:0-9 -m ./a.out
```

Assignment 10, Task 2:
Using the LIKWID Marker API

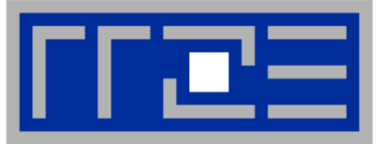
- **Output:**

```
[...]
Performance:     1839.7 MLUPs
Region ITER, Group 1: MEM
+-------------------+----------+
|    Region Info    |  Core 0  |
+-------------------+----------+
| RDTSC Runtime [s] | 5.213142 |
|     call count    |        4 |
+-------------------+----------+
[...]
+------------------------------------+------------+
|               Metric               |   Core 0   |
+------------------------------------+------------+
|         Runtime (RDTSC) [s]        |     5.2131 |
|        Runtime unhalted [s]        |     5.1748 |
|             Clock [MHz]            |  2200.0600 |
|                 CPI                |     1.1604 |
|  Memory read bandwidth [MBytes/s]  | 14808.2476 |
|  Memory read data volume [GBytes]  |    77.1975 |
| Memory write bandwidth [MBytes/s]  | 14762.9001 |
| Memory write data volume [GBytes]  |    76.9611 |
|     Memory bandwidth [MBytes/s]    | 29571.1477 |
|     Memory data volume [GBytes]    |   154.1586 |
+------------------------------------+------------+
```

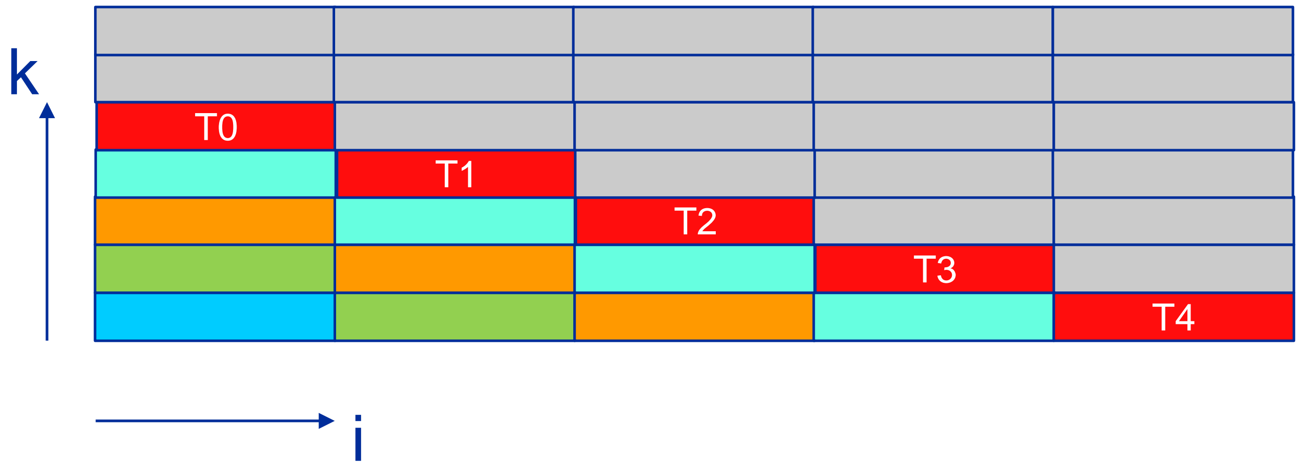
$$B_C = \frac{29.57\ \mathrm{GB/s}}{1.84\ \mathrm{GLUP/s}} \approx 16.1\ \mathrm{B/LUP}$$

Measured code balance

PPP 2D Gauss-Seidel ccNUMA issues



- Which loop to parallelize upon initialization for proper ccNUMA placement?



  - Steady state (after wind-up) has fixed i-block-to-thread mapping
  - → parallelize i loop upon init!
- But does it work for imax=8000 on Emmy?
  - No! ← 2 MiB page size on Emmy nodes
    (it would work with 4 KiB, sort of)
  - Last resort: static,1 parallelization of outer loop

PPP 2D Gauss-Seidel ccNUMA issues

- Round-robin first touch initialization → „random“ placement

```
!$OMP PARALLEL DO schedule(static,1)
  do k=1,kmax-1
     do i=1,imax-1
        phi(i,k)=1.d0
     enddo
  enddo
!$OMP END PARALLEL DO
```