Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

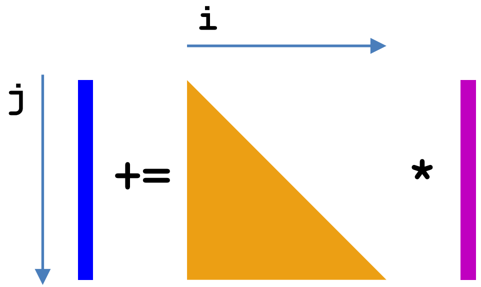
Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Assignment 5 discussion

High Performance Computing

Assignment 5, Task 1: dense triangular parallel MVM

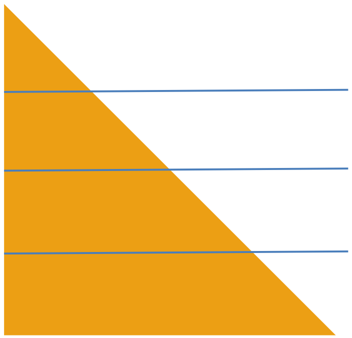
```
#pragma omp parallel private(i,k)
{
  for(k=0; k<niter; k++){
#pragma omp for schedule(runtime)
    for(j=0; j<n; ++j)
      for(i=0; i<=j; ++i)
        c[j] += a[i+n*j] * b[i];
    if(c[size >> 1]<0.0) whatever();
  }
}
```



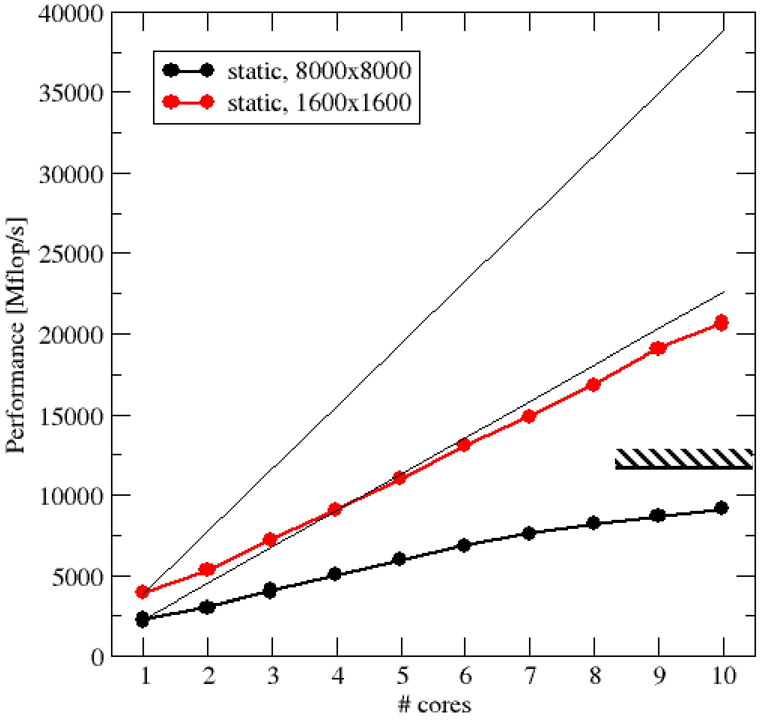
- Parallelize outer loop (least overhead)
- Multiple overlapping effects:
  - Load imbalance
  - Bandwidth saturation (this is where Roofline applies!)
  - Prefetcher madness
  - OpenMP overhead (at small sizes)

Assignment 5, Task 1: static scheduling

- Expected saturation pattern @ large size not really visible
- Expect good scaling @ smaller size, but weird scaling pattern because of load imbalance

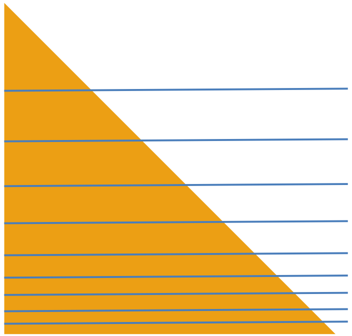


static

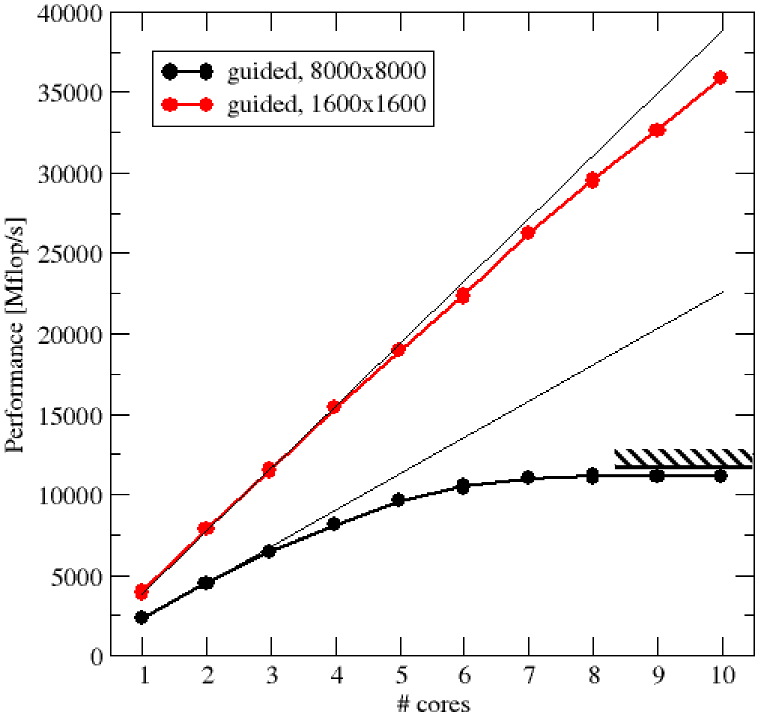


Assignment 5, Task 1: guided scheduling

- Large size now shows saturation in accordance with Roofline model
- Smaller size shows very good scaling (L3 cache is scalable resource)
- → proper scheduling lets us reach the hardware limits

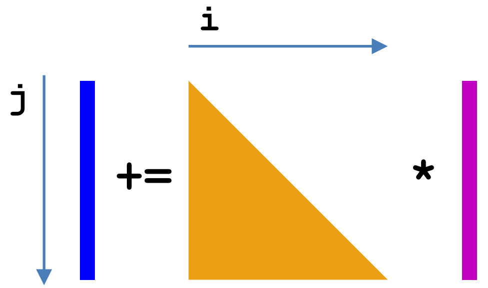


guided,1



Assignment 5, Task 1: performance model

```
for(j=0; j<n; ++j)
  for(i=0; i<=j; ++i)
    c[j] += a[i+n*j] * b[i];
```



Traffic analysis

- `a[]`: read each element once
  traffic: $\frac{n(n+1)}{2} + \varepsilon$
- `c[]`: update each element once
  traffic: $2n$
- `b[]`: read from memory once, reuse from cache $(n-1)$ times if n is small enough
  traffic: n
- $\rightarrow B_C = \frac{(3n+n(n+1)/2)\times 8}{2n(n+1)/2} \frac{\text{byte}}{\text{flop}} \approx 4 \frac{\text{byte}}{\text{flop}}$
- $b_S = 47 \frac{GB}{s}$ (read-only)

$$P_{BW} = \frac{b_S}{B_c} = \frac{47 \frac{GB}{s}}{4 \frac{B}{F}} = 11.8 \frac{GF}{s}$$

Assignment 5, Task 2: OpenMP histogram

- Compute simplified histogram of a (integer) random number generator:
  `hist[rand() % 16]`

- 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);
}
```

`RAND = myrand(&lseed)`

Time = 3.6 s

abserr = $3 * 10^{-6}$

Standard libc RNG

`RAND = rand_r(&lseed)`

Time = 6.7 s

abserr = $4 * 10^{-6}$

Straightforward parallelization

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

Getting it correct

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 more overhead (synchronization)

Avoid serialization (partially)

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

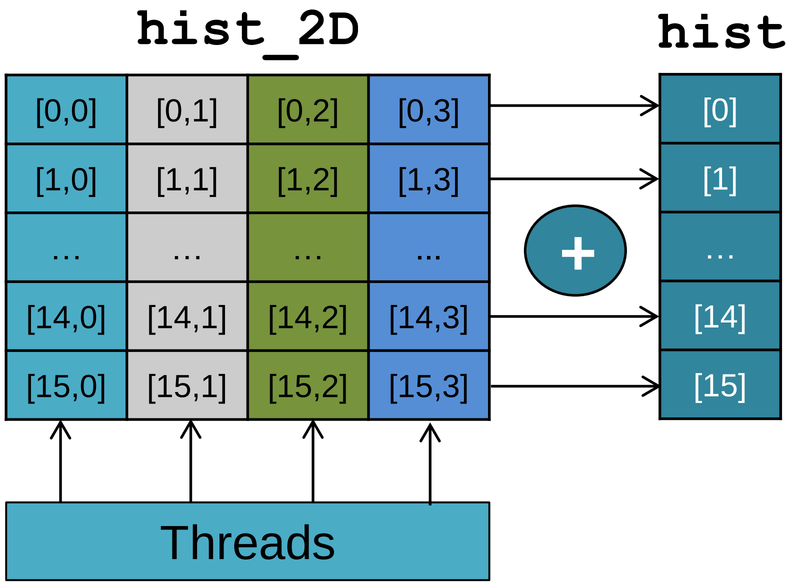
```
#pragma omp parallel for \
 firstprivate(lseed) private(value)
for(i = 0; i < n_loop; ++i) {
  value = myrand(&lseed) & 0xf;

  #pragma omp critical
    hist[value]++;

}
```

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

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

Getting rid of the critical section

Idea: give each thread its own histogram and accumulate at the end




```
// 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];
}
```

Getting rid of the critical section

Result Quality

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

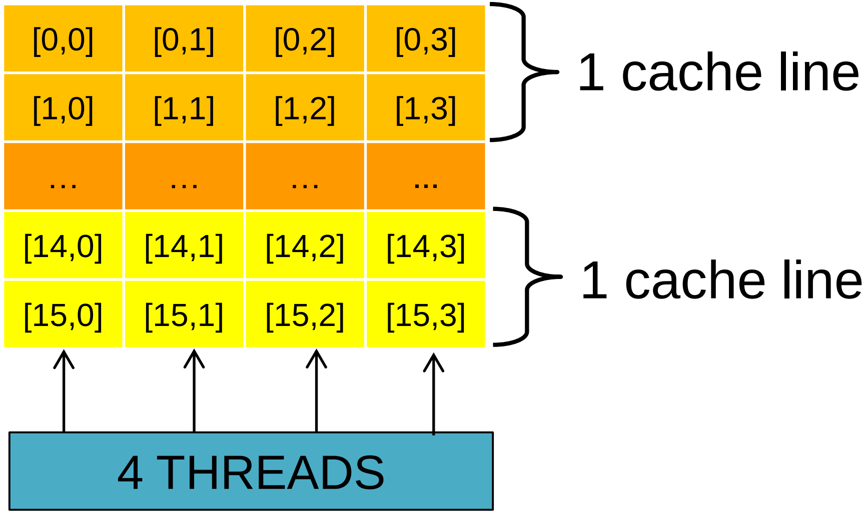
Baseline: $3*10^{-6}$

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 ?!

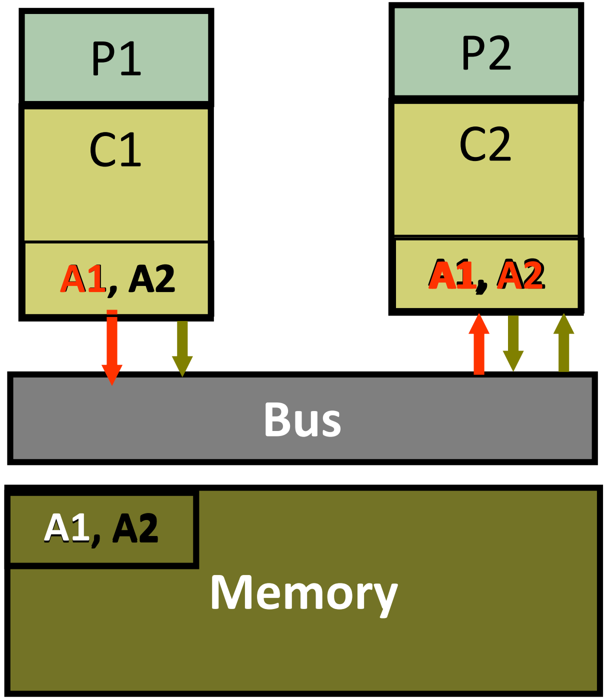


Each thread writes frequently to every cache line of `hist_2D`

→ False Sharing

Interlude: cache coherence and false sharing

Cache coherence protocol must keep track of cache line status



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

C2 is exclusive owner of CL ←

time ↓

Avoiding false sharing

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)
  for(i = 0; i < n_loop; ++i) {
    hist_local[myrand(&lseed) & 0xf]++;
  }
  #pragma omp critical
    for (i = 0; i < 16; ++i)
      hist[i] += hist_local[i];
}
```

Performance: OK now – nice scaling, too

Problem: Quality still gets worse as number of threads increase?! → same seed per thread!

Improve result quality (statistics)

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

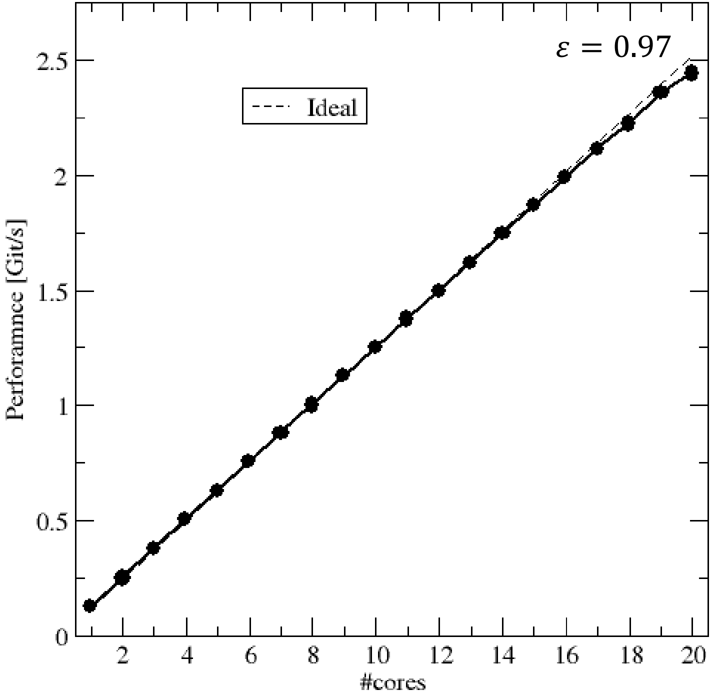
Baseline: 3.6s

```
#pragma omp parallel
{
  int hist_local[16] = { 0 };
  int myseed;
  #pragma omp critical
    myseed = myrand(&seed);
  #pragma omp for
  for(i = 0; i < n_loop; ++i) {
    hist_local[myrand(&myseed) & 0xf]++;
  }
  #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……..

Final version with array reduction

```
#pragma omp parallel
{
  int myseed;
  #pragma omp critical
    myseed = myrand(&seed);
  #pragma omp for reduction(+:hist[0:16])
  for(i = 0; i < n_loop; ++i) {
    hist[myrand(&myseed) & 0xf]++;
  }
}
```



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 nontrivial