Programming and Performance Tuning in OpenMP

Daniel Langr, Ivan Šimeček, Pavel Tvrdík

Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague

Parallel and Distributed Programming (MIE-PDP)
Summer Semester 2020/21, Lecture 04
(Version Timestamp: 5. 3. 2021 13:48)

https://courses.fit.cvut.cz/MIE-PDP

Lecture Outline

Memory access latencies and synchronization overheads

- For current commodity multi- and manycore processors, the following holds:

$$T_{comp} \ll T_{mem} \ll T_{synchro}, \text{ where}$$

  - T_{comp} is the latency of a computational operation (data in registers),
  - T_{mem} is the memory access latency (Read-Write), see Slide 4,
  - $T_{synchro}$ is the latency of a synchronization operation (e.g., T_{barr}).
- The latency of an atomic memory operation with a shared memory cell is

$$T_{atom} \geq T_{mem}$$

.
- In GPU, completely different relations apply:
  - within warp: $T_{synchro} = 0$ (SIMD model),
  - within the block of threads: $T_{comp} \ll T_{synchro} \ll T_{mem}$.

Parameters of the Intel memory hierarchy

architecture type	Skylake i7-6700	SandyBridge i3-2120
frequency f	3.4 GHz	3.3 GHz
# cores	4	4
L1 size L2 size L3 size	32 KB 256 KB 8 MB	32 KB 256 KB 3 MB
T_{mem}(L1) T_{mem}(L2) T_{mem}(L3)	4-5 C 12 C 42 C	4-5 C 12 C 28 C
T_{mem}(RAM) RAM type	42 C + 51 ns DDR4-2400 CAS15	28 C +56 ns DDR3-1600 CAS10

- Cache latency is in CPU clock cycles: 1 C $\doteq$ 0,3ns.
- RAM latency is therefore approx. 40 times greater then the L1 latency.

Memory hierarchy bandwidth

- Besides the memory latency T_{mem}, the **memory hierarchy (controllers) bandwidth** is a key performance parameter.
- Bandwidth for 1 cache memory module is given by:

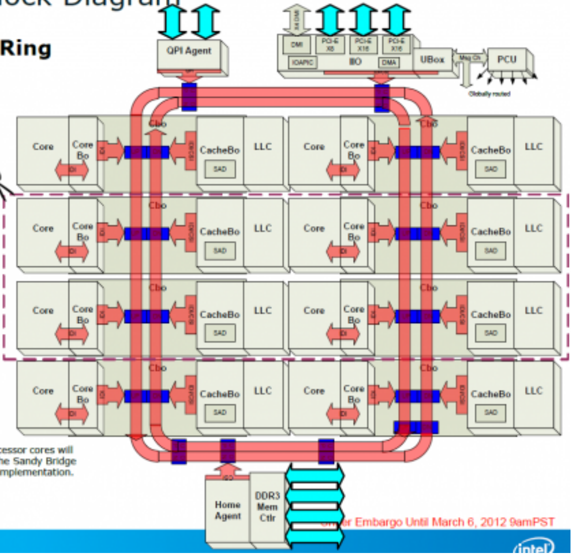
$$= f \times S, \quad \text{where}$$

  the **latency** S is # of Bytes transferred within 1 clock cycle C.
- For the current Intel processors x86-64, we get:
  - 1 L1 cache module per core and $S = 96B$,
  - 1 L2 cache module per core and $S = 32B$,
  - 1 L3 cache shared by CPU and $S = 16B$.
- Hence, the max. bandwidth, e.g., for i7-6700 is:
  - L1: approx. 320 GB/s (for each core),
  - L2: approx. 108 GB/s (for each core),
  - L3: approx. 54 GB/s (shared by all cores),
  - RAM: approx. 34 GB/s (shared by all cores).
- For RAM, this is in fact the real bandwidth of the memory **controllers**. The real memory bandwidth is smaller and it depends on the number and types of memory modules.

Xeon Microarchitecture

Xeon E5-2600 Block Diagram

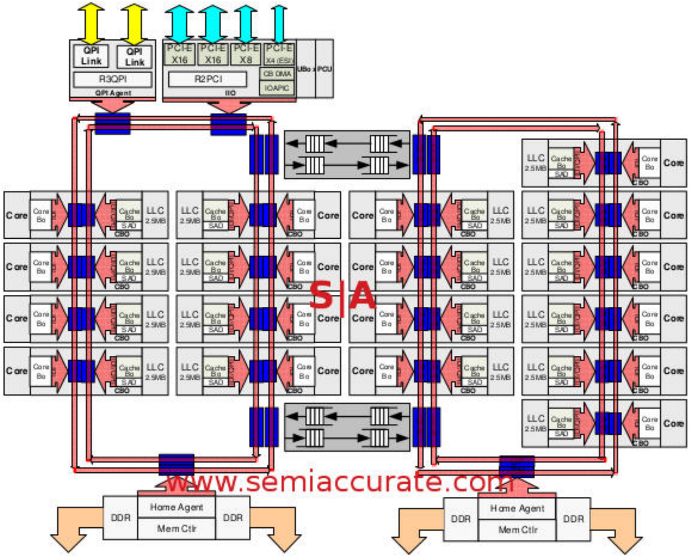
- **Bi-Directional Full Ring**
  - 32B/clk/agent
  - 8 Core/LLC slices
- **Last Level Cache**
  - 32B/clk/slice
- **Dual QPI Agent**
- **Integrated I/O**
- **Home Agent**
- **Integrated Memory Controller**
  - Connected to HA
- **PCU**
- **"Ubox"**



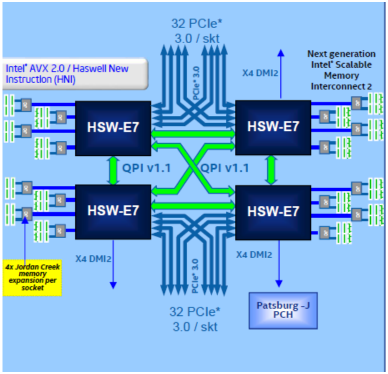
Block Diagram Illustrative only. Number of processor cores will vary with different processor models based on the Sandy Bridge Microarchitecture. Represents server processor implementation.

Under Embargo Until March 6, 2012 9amPST

Haswell Microarchitecture



Haswell server


32 PCIe* 3.0 / skt
Intel® AVX 2.0 / Haswell New Instruction (HNI)
X4 DMI2
Next generation Intel® Scalable Memory Interconnect 2
HSW-E7
HSW-E7
HSW-E7
HSW-E7
QPI v1.1
QPI v1.1
PCIe* 3.0
PCIe* 3.0
4x Jordan Creek memory expansion per socket
X4 DMI2
X4 DMI2
Patsburg -J PCH
32 PCIe* 3.0 / skt

- It is a system with HW distributed memory = NUMA (*NonUniform Memory Access Arch*).
- QPI (*Quick Path Interconnect*) = packet network (1 packet = 1 cache line = 1 cache block).
- QPI supports MESIF coherence protocol = virtual shared memory = CC-NUMA (*cache-coherent*).

CPU-memory bottleneck

- The memory bus (controller) becomes for **multicore** CPUs more and more the **narrower bottleneck**, esp. if the algorithm **cannot use** the date transferred to CPU **many times**.
- An example of the **scalar product** for 4-core CPU i7-6700:
  - Using the vector instruction AVX, each core is capable to execute 16 FP multiply and add operations in one CPU clock cycle.
  - 1 instruction AVX therefore needs 32 float numbers ($= 32 \times 4 = 128$B) each clock cycle.
  - 1 core therefore needs the memory bandwidth 3.4GHz $\times$ 128B $=$ 435 GB/s!!!
  - **Even the L1 cache modules** $(320$ GB/s$)$ **are not fast enough**!!!

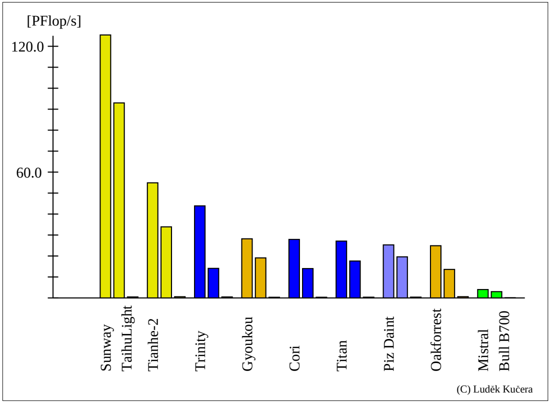
The limiting factor for achieving the peak CPU computational performance in similar computations is the data bandwidth of the CPU-memory interface.

Requirements for effectively parallelizable algorithms

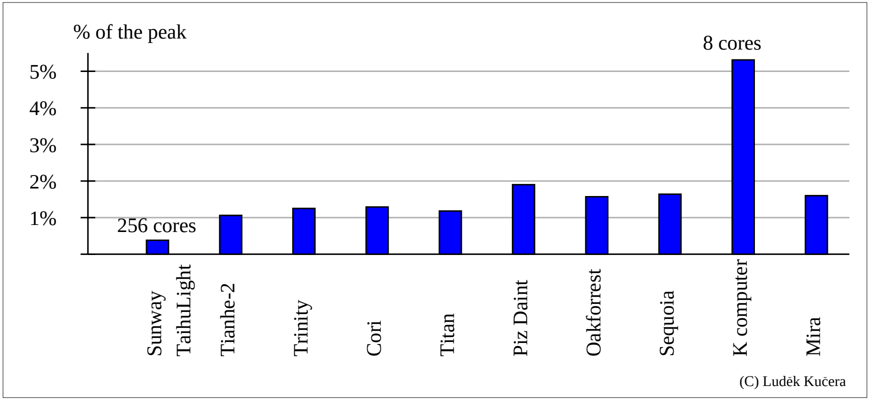
- From the viewpoint of effective parallelizability on multi-/manycore CPUs with shared memory, parallel algorithms belong to two categories:
  - **CPU/compute bound algorithms**: The performance is limited by the performance of CPU, i.e. the compute time with given data is larger than the time to move the data from memory to CPU and back.
    - Examples: C3S for NP-hard problems (e.g., TSP), matrix factorization, matrix-matrix multiplication ($O(n^3)$ for $(n \times n)$-matrices), Floyd-Warshall ($O(|V|^3)$).
  - **Memory bound algorithms**: The performance is limited by the CPU-memory bandwidth, i.e., the compute time on given data is less than the time needed for moving the data from memory to CPU and back. Typically, these are sequentially linear algorithms where each data element is used only k-times where $k \geq 1$ is a small constant.
    - Examples: scalar product, prefix sum, dynam. programming, FFT.
- **Criteria for good scalability**: The **type** of the compute task is important for decision whether parallelization makes sense or not.
  - Theoretical speedup (e.g., in the PRAM model) is a must.
  - Seq. time complexity should be greater than linear or if linear, than with big hidden constants.

High Performance LINPACK vs HPCG

- **Conjugate Gradients** method is memory intensive: consists of vector scalar product, `SpMVM`, solution of triangular sparse set of linear equations.
- It is a new benchmark for HPC: http://www.hpcg-benchmark.org/.
- It represents **most** of HPC apps.



High Performance LINPACK vs HPCG



- Efficiency of the HPCG benchmark is at most 5% (for CPUs with few cores).
- For manycore CPUs it drops to 0.3%!!!
- This is an important example of a memory-bound algorithm.

Several conditions to optimize sequential codes

- Maximize the number of operations per Byte read from memory.
- Maximize the utilization of cache memories.
- Explicitly address the fact that data are brought from memory to caches with granularity of 1 cacheline (e.g., 64B).
    - The best access pattern to data is therefore with stride 1.
- Try to avoid:
    - Alternating accesses to large data structures $\Rightarrow$ frequent cache misses.
    - Indirect addressing $\Rightarrow$ bad data locality and therefore frequent cache misses. Then the data prefetching will not work and therefore large memory access latency comes into effect.

Sources of OpenMP code inefficiencies

- Imbalanced compute load for individual threads: parallel region terminates with a barrier $\Rightarrow$ waiting shorter threads = wasted cores.
- Too tight synchronization $\Rightarrow$ great number of barriers or critical sections.
- Limited parallelism, e.g., # of iteration of a `for` loop < # of threads.
- High overhead of thread management, e.g.,
  - frequent forks and joins of threads (e.g., in nested loops),
  - `schedule(dynamic)`.
- Significant inherently sequential part: due to the Amdahl's law, as big as possible part of a code should be parallelized. At the same time, due to the thread management overheads, we should strive to use moderate number of parallel threads (ideally equal to the # of cores).
- There are two important cases of inefficient utilization of cache memories:
  1. frequent writes into cached shared variables,
  2. **false sharing**.

False sharing

- **False sharing, cache line ping-ponging**: various threads write to different memory addresses, but that close each to other that they are cached into the same **cacheline** and the cache coherence protocol invalidates all other copies in the other cache memories.
- Typical situation: cacheline = 64B, an `int` number = 4B $\Rightarrow$ 16 numbers are cached into the same cacheline.
- False sharing appears typically in data parallelism.
- Elimination of false sharing is contradictory to the requirement of unit stride for single-thread algorithms.
- The following code assigned the iterations to the threads very inappropriately: the MESI protocol will force the threads to invalidate cached copies of shared data.

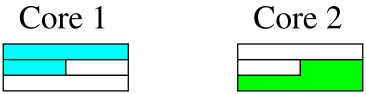
```
int A[n]; //we assume n >> p
#pragma omp parallel for shared(A) schedule(static,1)
  for (int i = 0; i < n; i++)
    A[i] = foo(i);
```

Reducing the effect of false sharing I

- A better distribution of loop iterations over a **sufficiently large array** among threads is chunk-uniform `schedule(static)`. The probability of writing of two or more threads into the same cacheline is lower.

```
int A[n];
#pragma omp parallel for shared(A) schedule(static)
  for (int i=0;i<n;i++)
    A[i] = foo(i);
```

- If, e.g., for the values from the previous slide we have $n \geq 16p$, then each thread is assigned a chunk of elements of array `A` of size $n/p \geq 16$ that fills more than one cacheline.
- False sharing becomes significant for $n = 24p$ (ping-pong of two consecutive pairs of threads).



Reducing the effect of false sharing II

- The effect of false sharing can be reduced by **user control** of static scheduling of iteration chunks (and therefore segments of elements of array `A`) to threads.
- False sharing, e.g., **will not happen** if

$$n/p = \texttt{cache\_line\_size/sizeof(int)}$$

  and array `A` is aligned equally as the cachelines. With this scheduling exactly one thread writes into exactly one cacheline.

```
int A[n]; int X=cache_line_size/sizeof(int);
#pragma omp parallel for shared(n,A) schedule(static,X)
  for (int i=0;i<n;i++)
    A[i] = foo(i);
```

Reducing the effect of false sharing III

- FS can be prevented with an artificial enlargement of the array elements using **dummy data**.
- In the following code, every array element is enlarged to the cacheline size. Therefore, no two threads can write into the same cacheline.
- This trick leads to multiplying the memory requirements and therefore flooding the memory bus bandwidth with dummy traffic.

```
int A[n][cache_line_size/sizeof(int)];
#pragma omp parallel for shared(A) schedule(static,1)
  for (int i=0;i<n;i++)
    A[i][0] = foo(i);
```

Core 1
Core 2

Reducing the effect of false sharing IV

- However, padding the cache memory with dummy contents is a good solution for small shared arrays of size p where each thread has a dedicated location for writing the result of its local computation.
- For example, parallel search of a given key `key` in a large unsorted array `A` where every thread writes into the shared array `results` its local result.
- Instead of array `Bool results[p]` we allocate `Bool results[p][r]` where `r = cache_line_size/sizeof(Bool)`.

```
void ParallelSearch(int A[n], int key) {
Bool results[p][r];
#pragma omp parallel for schedule(static)
{
  int my_index = omp_get_thread_num ();
  results[my_index][0] = NotFound;
  for (int i = 0; i < n; i++)
    if (A[i] == key) results[my_index][0] = Found;
} }
```

Histogram computation

- A **histogram** is a graphical representation of the distribution of numerical data.
- Very common algorithm. For example, in computer graphics, it is used for processing the tonal distribution in pixel images or to compress images.
- The input array values = histogram indexes. We limit ourselves to integers from $\langle 0, range - 1 \rangle$. For example, $range = 256$.
- Standard sequential algorithm has complexity $O(n) + O(range)$.

Code 1: Sequential histogram computation

```
int *buffer, n, //input array and its size
histogram[range]; //resulting histogram
  for (int i = 0; i < range; i++) //histogram initialization
    histogram[i] = 0;
  for (int j = 0; j < n; j++) //histogram computation
    histogram[buffer[j]]++;
```

Parallel computation of a shared histogram

Code 2: Parallel computation of a shared histogram

```
int *buffer,n; //input array and its size
histogram[range]; //resulting histogram
  for (int i = 0; i < range; i++) //histogram initialization
    histogram[i] = 0; //Master thread
  #pragma omp parallel for schedule(static) shared(histogram)
  { for (int j = 0; j < n; j++) { //histogram computation
      #pragma omp atomic update
          histogram[buffer[j]]++;
} }
```

- Neither additional memory nor additional computation is needed.
- Concurrent Writes into the same elements of the histogram happen
  $\Rightarrow$ increment operations must be atomic.
- Due to the indirect indexing, coherence cache misses will appear.
- $T(n, range, p) = O(range) + O(n/p)$.

Parallel reduction of local partial histograms

- Each thread has its own histogram copy in shared memory.
- Memory consumption increases p-times.
- Threads perform parallel Writes into disjoint histogram copies.
- At the end, they perform a parallel reduction of p-tuples of local

Code 3: Reduction of locally computed histograms

```
int *buffer,n, //input array and its size
histogram[p][range]; //p = # of threads
#pragma omp parallel
 {int my_id = omp_get_thread_num ();
  for (int i = 0; i < range; i++) //each thread initializes
    histogram[my_id][i] = 0; //sequentially its histogram
  #pragma omp for schedule(static)
    for (int j = 0; j < n; j++) //histogram computation
      histogram[my_id][buffer[j]]++;
  #pragma omp for schedule(static)
    for (int i = 0; i < range; i++) //each thread reduces
      for (int j = 1; j < p, j++) //range/p p-tuples of local
        histogram[0][i]+=histogram[j][i]; } //counters
```

Discussion of variants of parallel histogram algorithms

- Preferable scheduling is again `static`.
- The optimal variant depends on the performance of the atomic operations, the number of threads, the $range$ value.
- The Code 21 variant depends strongly on data. With high pp, indirect addressing will lead to non-locality of `W`rites into the histogram, and consequently to coherence cache misses.
- The Code 22 variant has:
  - p fold memory requirements. Hence:
    it cannot be used if $range$ is large or the number of threads p is large.
  - Parallel time = initialization + parallel computing + parallel reduction:
    $$T(n, range, p) = O(range) + O(n/p) + O(range).$$
- Memory bound computation: The total performance will be low, since it provides too little computing operations per 1 loaded Byte (see Lecture 4, Slide 8).

Multiplication of polynomials

- **Input**: polynomials $A = \sum_{i=0}^{m} a_i x^i$ and $B = \sum_{i=0}^{n} b_i x^i$, $a_m \neq 0$ a $b_n \neq 0$.
- **Output**: polynomial $C = A \times B = \sum_{i=0}^{m+n} c_i x^i$, where

$$c_k = \sum_{l=\max(0,k-n)}^{\min(k,m)} a_l b_{k-l}$$

- Classical sequential algorithm of $O(nm)$ complexity.
- Arrays `A,B,C` contain polynomial coefficients.
- To make the pseudocodes shorter, we will skip the initialization of `C`.

Code 4: Sequential multiplication of polynomials

```
int A[m+1],B[n+1],C[n+m+1];
for (int k = 0; k <= m + n, k++)
  C[k] = 0;
for (int i = 0; i <= m; i++)
  for (int j = 0; j <= n; j++)
    C[i+j] += A[i]*B[j];
```

Parallel multiplication of polynomials

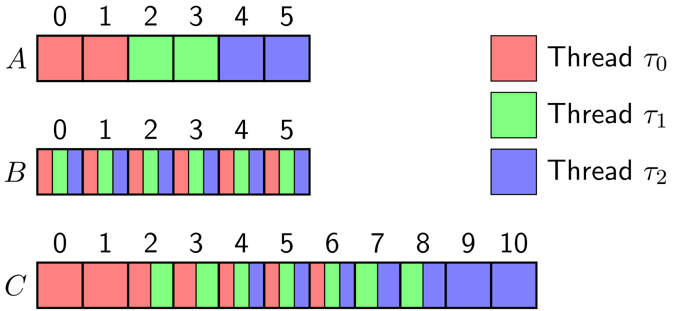
- $\exists$ 3 possible parallelizations.
  - Parallelization of the **outer** i-loop of the sequential algorithm.
  - Parallelization of the **inner** j-loop of the sequential algorithm.
  - Parallelization by decomposition of the output array into disjoint parts for the writes.
- Let us demonstrate graphically the differences among these 3 approaches of parallelization and distribution of the resulting access patterns for $p = 3$ threads τ_0, τ_1, τ_2 into the shared arrays of coefficients of input polynomials A, B of degree $m = n = 5$ (parallel `R`eads) and output polynomial C of degree 10 (parallel `W`rites).

Parallelization of the **outer** i-loop

Code 5: Polynomial multiplication using polynomial A parallelization

```
int A[m+1], B[n+1], C[n+m+1];
#pragma omp parallel for schedule(static)
  for (int i = 0; i <= m; i++) //this loop is parallelized
   for (int j = 0; j <=n; j++) {//each thread reads all B[j]
     #pragma omp atomic update //threads update
       C[i+j] += A[i]*B[j]; } //overlapping parts of C
```

$m = n = 5$, $p = 3$:



Parallelization of the **outer** i-loop

Code 5: Polynomial multiplication using polynomial A parallelization

```
int A[m+1], B[n+1], C[n+m+1];
#pragma omp parallel for schedule(static)
  for (int i = 0; i <= m; i++) //this loop is parallelized
   for (int j = 0; j <= n; j++) {//each thread reads all B[j]
     #pragma omp atomic update //threads update
       C[i+j] += A[i]*B[j]; } //overlapping parts of C
```

- Mapping of iteration chunks of the i-loop among threads is `schedule(static)`.
- Concurrently, all threads read sequentially all coefficients of the B polynomial.
- The segments of C where the threads write concurrently are not disjoint: different threads may try to **update simultaneously** elements of C

  $\Rightarrow$ **atomic update** operations are needed.

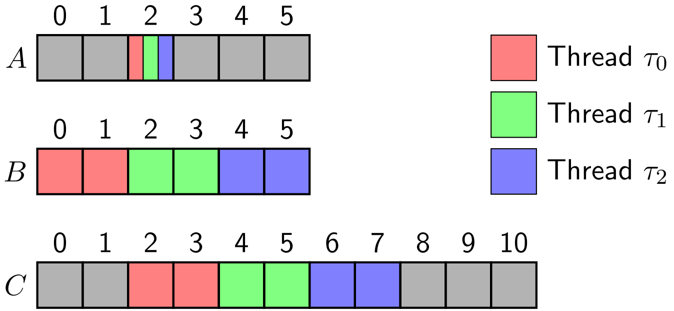
Parallelization of the **inner** j-loop

Code 6: Polynomial multiplication using polynomial B parallelization

```
int A[m+1], B[n+1], C[m+n+1]; //parallel inner loop
for (int i = 0; i <= m; i++) //in each iter. p new threads are forked
 #pragma omp parallel for schedule(static)
  for (int j = 0; j <= n; j++) //parallel Reads of the same A[i]
   C[i+j] += A[i]*B[j]; //disjoint Reads of B and Writes of C
```

$m = n = 5$, $p = 3$:

Visualization of accesses during the 3rd iteration of the i-loop.



Parallelization of the **inner** j-loop

Code 6: Polynomial multiplication using polynomial B parallelization

```
int A[m+1], B[n+1], C[n+m+1]; //parallel inner loop
for (int i = 0; i <= m; i++) //in each iter. p new threads are forked
 #pragma omp parallel for schedule(static)
  for (int j = 0; j <= n; j++) //parallel Reads of the same A[i]
   C[i+j] += A[i]*B[j]; //disjoint Reads of B and Writes of C
```

- All threads concurrently read subsequently all coefficients of A: they all read the same element $A[i]$ in the same step.
- The segments of C where they write are disjoint, but the repeated scheduling of $n+1$ iterations among p threads and the implicit barrier synchronization at the end of each iteration of the inner j-loop increases the overhead, i.e., $m \times T_{barr}$ (unless the compiler implements an optimization using a *thread pool*).
- During each outer iteration, threads invalidate each other cached updates of the resulting polynomial: **false sharing** is unavoidable.

Parallel Writes into disjoint areas of polynomial C

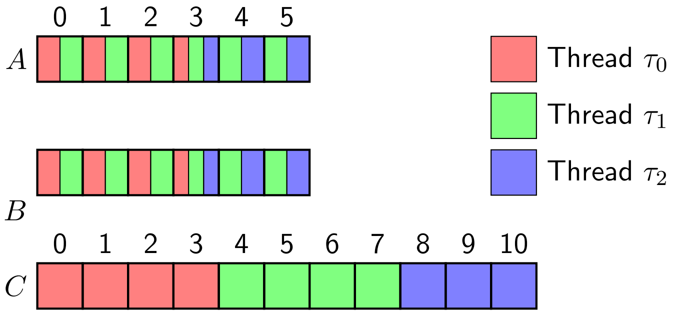
y	$[i,j]$
0	$[0,0]$
1	$[0,1]+[1,0]$
2	$[0,2]+[1,1]+[2,0]$
3	$[0,3]+[1,2]+[2,1]+[3,0]$
4	$[0,4]+[1,3]+[2,2]+[3,1]+[4,0]$
5	$[0,5]+[1,4]+[2,3]+[3,2]+[4,1]+[5,0]$
6	$[1,5]+[2,4]+[3,3]+[4,2]+[5,1]$
7	$[2,5]+[3,4]+[4,3]+[5,2]$
8	$[3,5]+[4,4]+[5,3]$
9	$[4,5]+[5,4]$
10	$[5,5]$

Parallel Writes into disjoint areas of polynomial C

Code 7: Polynomial multiplication using polynomial C parallelization

```
int A[m+1], B[n+1], C[m+n+1]; //output decomposition
int p = omp_get_num_procs();
#pragma omp parallel for schedule(static,(m+n)/c*p)
 for (int k = 0; k <= (m + n); y++)
   for (int l = max(0, k - n); l <= min(k,m); l++)
     C[k] += A[l]*B[k-l];
```

- In this case, thread τ_1 has twice as big load as the others.



Parallel Writes into disjoint areas of polynomial C

Code 7: Polynomial multiplication using polynomial C parallelization

```
int A[m+1], B[n+1], C[m+n+1]; //output decomposition
int p = omp_get_num_procs();
#pragma omp parallel for schedule(static,(m+n)/c*p)
 for (int k = 0; k <= (m + n); k++)
   for (int l = max(0, k - n); l <= min(k,m); l++)
     C[k] += A[l]*B[k-l];
```

- Threads statically or dynamically partition the output array C into p disjoint parts, each thread having its own part to write in.
- However, the number of Writes into elements of $C[k]$ depends on k.
- For example, if $m = n$, then for $k = 1, \ldots, m$, the number of Writes grows linearly from 1 to m and for $k = m, \ldots, 2m$, it linearly decreases from m to 1.
- So, we must balance the load, e.g., `schedule(dynamic)` or `schedule(static,chunk_size)` (however, beware false sharing).
- A possible solution: If $m, n \gg 16p$, then the iteration chunk size `chunk_size` could be $(m + n)/(c \times p)$ for a small constant c.

Sequential Matrix-Matrix Multiplication (MMM)

- Consider 2 square matrices A, B of dimension $n \times n$.
- The goal is to compute the product $C = A \times B$.
- Classical secondary school algorithm: n^2 of scalar products of rows of A and columns of B. Hence n^3 of the multiplication operations.

Code 8: Sequential matrix-matrix multiplication (MMM)

```
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) { //n^2 iterations
    s=0.0;
    for (int k = 0; k < n; k++) //scalar product
      s += A[i][k]*B[k][j];
    C[i][j] = s; }
```

- $\exists$ 3 possible levels of parallelizations since there are no data dependencies here.
- Any scheduling can be used: again, the `static` one is reasonable.

Parallel MMM I: Parallelization of the i-loop

- The written areas are disjoints.
- Minimal synchronization: only 1 implicit barrier.

Code 9: Parallelization of the i-loop with block iteration chunks

```
float A[n][n], B[n][n], C[n][n];
#pragma omp parallel for schedule(static)
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) { //blocks of n/p j-loops are executed
    float s = 0.0;            //sequentially by all threads concurrently
    for (int k = 0; k < n; k++)
      s += A[i][k]*B[k][j];
    C[i][j] = s; }
```

Parallel MMM II: Parallelization of the j-loop with block iteration chunks

- Higher synchronization overhead $n \times T_{barr}$.
- No Write collisions in the shared memory.
- For sufficiently large n/p, only few false sharings should appear, since each thread writes into a continuous segment of C of size n/p.

Code 10: Parallelization of the j-loop with block iteration chunks

```
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; i++) //master thread only
  #pragma omp parallel for schedule(static)
   for (int j = 0; j < n; j++) { //each thread obtains n/p iterations
    float s = 0.0; //local private variable
    for (int k = 0; k < n; k++) //scalar product
      s += A[i][k]*B[k][j];
    C[i][j] = s;} //all threads write into the same row of C
```

Parallel MMM III: Parallelization of the j-loop with cyclic iteration chunks

- The same as Code 10: $n \times T_{barr}$ synchronization overhead and no Write collisions into shared memory.
- However, false sharing can appear since different threads write simultaneously into adjacent elements of the same rows of C and therefore into the same cachelines.

Code 11: Parallelization of the j-loop with cyclic iteration chunks

```
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; i++) //master thread only
 #pragma omp parallel for schedule(static,1)
  for (int j = 0; j < n; j++) { //cyclic map of iterations to threads
   float s = 0.0; //local private variable
   for (int k = 0; k < n; k++) //scalar product
    s += A[i][k]*B[k][j];
   C[i][j] = s;} //2 threads can write into the same cacheline
```

Parallel MMM IV: scheduling of j-loop iterations within a parallel region

- The same as Code 10, except for the synchronization: Instead of n fold execution of the Fork-Join with complexity $n \times T_{barr}$.
- p threads are created only once, so that for larger p, this should be faster than Code 10.

Code 12: Scheduling of the j-loop iterations within a parallel region

```
float A[n][n], B[n][n], C[n][n];
#pragma omp parallel
for (int i = 0; i < n; i++) //all threads execute simultaneously
  #pragma omp for schedule(static)
   for (int j = 0; j < n; j++) { //each thread obtains n/p iterations
    float s = 0.0; //local private variable
    for (int k = 0; k < n; k++) //scalar product
      s += A[i][k]*B[k][j];
    C[i][j] = s;}
```

Parallel MMM V: Parallelization of the k-loop using reduction

- The master τ_0 runs sequentially n^2 iterations of the i- and j-loops.
- In each iteration, p threads perform a **parallel reduction** — scalar product of 1 row of A and 1 column of B.
- The biggest synchronization overhead so far: $n^2 \times (T_{barr} + T_{\mathrm{PR}}(n,p))$, see Lecture 2.

Code 13: Parallelization of the k-loop using reduction

```
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) { //seq. execution of n^2 iterations
    float s = 0.0;
    #pragma omp parallel for schedule(static) reduction(+ : s)
     for (int k = 0; k < n; k++) //scalar product by par.red.
      s += A[i][k]*B[k][j];
     C[i][j] = s; //Master thread writes 1 elem of C
 }
```

Parallel MMM VI: Parallelization of the k-loop with 1 Fork-Join

- All threads redundantly identically iterate the i- a j-loops.
- In contrast to Code 13, p threads are created only once and Code 14 should be faster than Code 13.

Code 14: Parallelization of the k-loop with 1 Fork-Join

```
float A[n][n], B[n][n], C[n][n];
#pragma omp parallel
for (int i = 0; i < n; i++)
 for (int j = 0; j < n; j++) { //n^2 iterations cloned for p threads
  float s = 0.0;
   #pragma omp for schedule(static) reduction(+: s)
    for (int k = 0; k < n; k++) //scalar product by par.red.
     s += A[i][k]*B[k][j];
   #pragma omp master
     C[i][j] = s;}
```

Performance analysis of parallel MMM variants

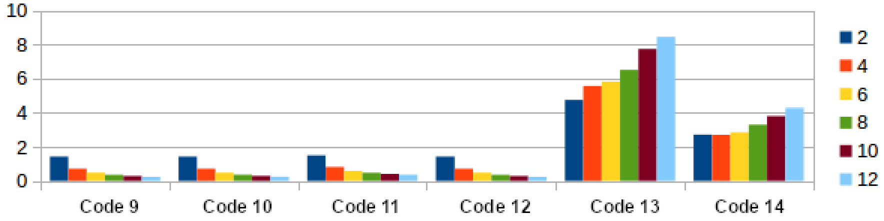
- We measured all 6 variants on a 12-core CPU of the `star` cluster .
- The table gives $T(n,p)$ pro $n = 1200$ (n^3 multiplications).
- The sequential time is $T(n,1) = 2.84$ [s].

p	Code 9	$S(n,p)$	Code 10	Code 11	Code 12	Code 13	Code 14
2	1,43	2	1,43	1,5	1,43	4,73	2,72
4	0,715	4	0,718	0,819	0,718	5,55	2,71
6	0,478	5,9	0,482	0,59	0,48	5,81	2,85
8	0,357	8	0,364	0,477	0,36	6,5	3,3
10	0,288	10	0,292	0,408	0,289	7,73	3,81
12	0,24	11,8	0,245	0,363	0,241	8,44	4,27

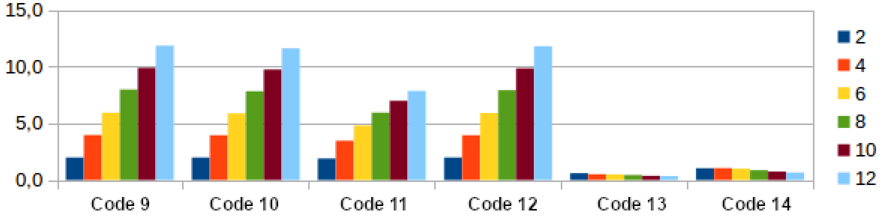
- Code 9 and Code 10 scale very well (linear speedup).
- Code 10 is just slightly slower than Code 9 (due to $1200 \times T_{barr}$).
- Code 11 is slower than Code 10 due to false sharing.
- Code 12 is faster than Code 10 due to 1 single Fork-Join.
- Code 13 is significantly slower than Code 9 (due to $1200^2 \times T_{barr}$).
- Code 14 is better than Code 13 due to 1 Fork-Join.
- Time of Code 13+14 grows with p (barriers dominate)!

Performance analysis of parallel MMM variants

$T(n, p)$:



$S(n, p)$:

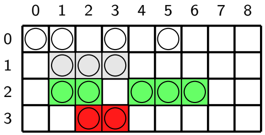


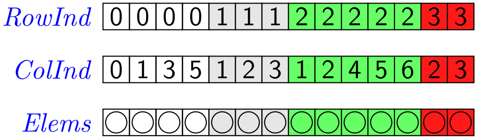
Sparse matrix vector multiplication

- Consider a computation $y = \mathcal{A}x$ where
  - $\mathcal{A}$ is an input **sparse** matrix of order $n \times n$, $\mathcal{A} = (a_{i,j})$.
  - x is an input vector of order n, and
  - y is the output vector of order n.
- The number of non-zero elements in $\mathcal{A}$ is denoted by N.
- Sparsity of matrix $\mathcal{A}$: we assume that $1 \ll n \leq N \ll n^2$.
- **Sparse matrix vector multiplication** (shortly `SpMVM`) is the most common operation in numeric linear algebra needed in iterative solvers of systems of linear equations, in the conjugate-gradient methods, and many others.
- The time complexity $O(N) \quad \Rightarrow \quad$ it is a classical memory-bound computation: little computing per 1 Byte transferred from memory to CPU and therefore the CPU/memory bottleneck applies.

The Coordinate format

- The **Coordinate** (COO) format is the simplest representation of sparse matrices.
- Sparse matrix A is represented by 3 arrays:
  - $RowInd[0, \dots, N-1]$ contains indexes of **rows** of non-zero elements of $\mathcal{A}$.
  - $ColInd[0, \dots, N-1]$ contains indexes of **columns** of non-zero elements of $\mathcal{A}$,
  - $Elems[0, \dots, N-1]$ contains **values** of non-zero elements of $\mathcal{A}$,
- The ordering of non-zero elements is not defines, but it is row-wise typically.




RowInd 0 0 0 0 1 1 1 2 2 2 2 2 3 3
ColInd 0 1 3 5 1 2 3 1 2 4 5 6 2 3
Elems

Sequential SpMVM in the COO format

- Since the i-th element of the resulting vector y, $y[i]$, is computed using a scalar product of the i-th row of matrix A and vector x

$$y[i] = \sum_{j=0}^{n-1} A[i,j] * x[j].$$

- The j-th element of vector x therefore meets the element from column j in matrix A.
- This leads to the following code:

Code 15: Sequential SpMVM in the COO format

```
struct A; //matrix in the COO format
float x[n]; //array representing the input vector x
float y[n]; //array representing the output vector y=Ax
for (i = 0; i < n; i++}
  y[i] = 0.0;
for (i = 0; i < N; i++)
  y[A.RowInd[i]] += A.Elems[i] * x[A.ColInd[i]];
```

Parallel SpMVM in the COO format

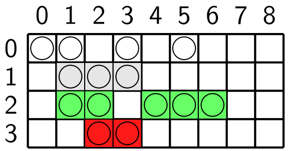
- The previous code can be parallelized easily, scheduling can be any.
- Drawbacks: the locality principle is not ensured (which may lead to cache misses) and **atomic update** operations must be used, therefore the parallelization will not be efficient.

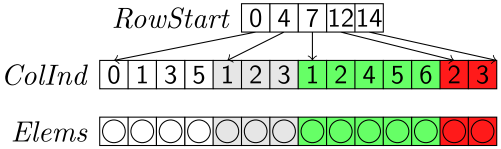
Code 16: Parallel SpMVM in the COO format

```
struct A; //matrix in the COO format
float x[n]; //array representing the input vector x
float y[n]; //array representing the output vector y=Ax
#pragma omp parallel for
  for (int j = 0; j < n; j++)
    y[j] = 0.0; //vector y initialization
#pragma omp parallel for
  for (int i = 0; i < N; i++) {
    float temp = A.Elems[i] * x[A.ColInd[i]];
    #pragma omp atomic update
      y[A.RowInd[i]] += temp;
}
```

The CSR format

- The format of **compressed rows** (Compressed Sparse Row, CSR).
- Sometimes called also Compressed Row Storage (CRS).
- Sparse matrix $\mathcal{A}$ is represented by 3 arrays $RowStart$, $ColInd$, and $Elems$.
    - Lexicographic elements ordering (by row and column indexes).
    - Array $RowStart[0,\ldots,n]$ contains indexes of the array $ColInd$ from where the elements of column indexes of non-zero elements of individual rows of A are stored.
    - Array $ColInd[0,\ldots,N-1]$ contains indexes of columns of non-zero elements $\mathcal{A}$.
    - Array $Elems[0,\ldots,N-1]$ contains the values of nonzero elements of $\mathcal{A}$.





Sequential SpMVM in the CSR format

Code 17: Sequential SpMVM in the CSR format

```
struct A; //matrix in the CSR format
float x[n]; //array representing the input vector x
float y[n]; //array representing the output vector y=Ax
for (j = 0; j < n; j++) { //loop on rows
   sum = 0.0;
   for (i = A.RowStart[j]; i < A.RowStart[j+1]; i++) {
      //loop on elements of 1 row
      sum  += A.Elems[i] * x[A.ColInd[i]];
   }
   y[j] = sum;
}
```

Observation: In case of the outer loop parallelization, the threads will write into disjoint regions of vector y and therefore atomic update operations are not needed.

Parallel SpMVM in the CSR format

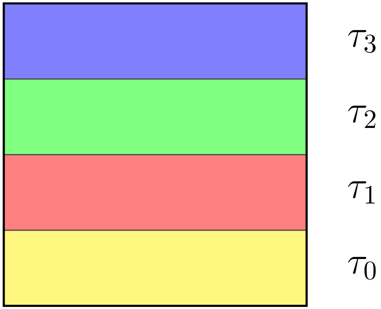
Code 18: Parallel SpMVM in the CSR format

```
struct A; //matrix in the CSR format
float x[n]; //array representing the input vector x
float y[n]; //array representing the output vector y=Ax
#pragma omp parallel for schedule([static[,X]]|[dynamic[,X]])
  for (j = 0; j < n; j++) { //loop on rows
    sum = 0.0;
    for (i = A.RowStart[j]; i < A.RowStart[j+1]; i++) {
      //loop on elements of 1 row
      sum  += A.Elems[i] * x[A.ColInd[i]];
    }
    y[j] = sum;
}
```

- The choice of iteration scheduling has significant impact on the speedup and performance.
- Assume $p = 4$ threads $\tau_0, \ldots, \tau_3$.

Parallel SpMVM in the CSR format II

- `#pragma omp parallel for schedule(static)`.
- Rows of $\mathcal{A}$ are distributed among threads **block-wise**.

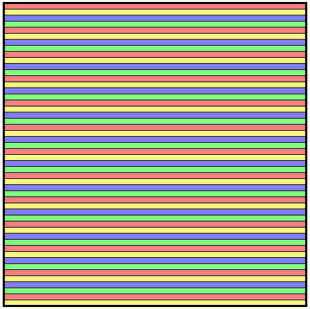


Drawbacks:

- If matrix $\mathcal{A}$ has irregular structure of non-zeros, then the loads of threads are not balanced.
- False sharing should not be an issue ($n \gg p$).

Parallel SpMVM in the CSR format III

- `#pragma omp parallel for schedule(static,1)`
- Rows of $\mathcal{A}$ are distributed among threads **cyclic-wise**.

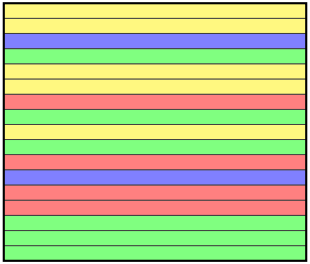


Drawbacks:

- Two threads executing `y[j] = sum` can suffer from false sharing.
- If matrix $\mathcal{A}$ has irregular structure of non-zeros, then the loads of threads are not balanced.

Parallel SpMVM in the CSR format IV

- `#pragma omp parallel for schedule(dynamic,X)`
- Blocks of X rows of matrix $\mathcal{A}$ are scheduled dynamically to threads (see the figure).
- False sharing is eliminated if $X \ll n$ is **the multiple of the number of elements in the cacheline** (we assume that every row of matrix $\mathcal{A}$ contains at least 1 non-zero element).

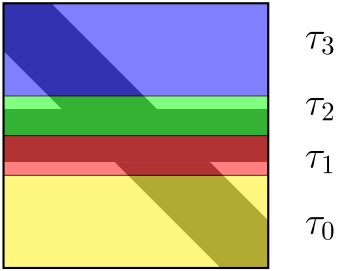


Drawbacks:

- Overhead of the `dynamic` scheduling.

Parallel `SpMVM` in the CSR format V

- Consider matrix $\mathcal{A}$ with non-uniform scattering of non-zeros.
- For example, the matrix on the figure has a dense diagonal band and a dense middle horizontal band.



- The **balanced** method: The matrix is split explicitly into p disjoint row bands so that they all contain **roughly the same number of non-zeros**.
- The width of the bands therefore varies. See the example on the above figure for $p = 4$.
- Every thread processes its own band.

Parallel SpMVM in the CSR format VI

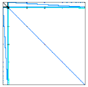
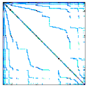
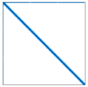
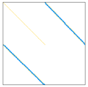
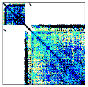
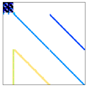
Code 19: Balanced parallel SpMVM for CSR

```
int band[p+1]; //pointers to 1st rows in thread bands
band[p] = n; //
#pragma omp parallel
{ int my_id = omp_get_thread_num();
  int my_number = my_id*N/p;
  //position of the 1st nonzero elem for me
  int my_index = binary_search(A.RowStart, my_number);
  //pointer to the start of the 1st row of my band
  band[my_id] = my_index;
  #pragma omp barrier
  for (int j = band[my_id]; j < band[my_id+1]; j++) {
   sum = 0.0; //parallel loop on all rows of my band
   for (int i = A.RowStart[j]; i < A.RowStart[j+1]; i++)
      //row No. i from my band is multiplied
      sum  += A.Elems[i] * x[A.ColInd[i]];
   y[j] = sum; }
}
```

Experimental evaluation of the SpMVM implementations

- We have conducted measurements of the SpMVM implementations on a 12core CPU in the star cluster for 6 matrices from the Suite Sparse Matrix Collection http://www.cise.ufl.edu/research/sparse/matrices:

Matrix name	Abbrev	n	N	N/n
circuit5M	ci	$5,5 \times 10^6$	$59,5 \times 10^6$	$10,7$
FullChip	fu	$3,0 \times 10^6$	$26,6 \times 10^6$	$8,9$
Rajat31	ra	$4,7 \times 10^6$	$20,3 \times 10^6$	$4,3$
Nlpkkt120	nl	$3,5 \times 10^6$	$95,1 \times 10^6$	$26,8$
ldoor	ld	$9,5 \times 10^5$	$42,5 \times 10^6$	$44,6$
TSOPF_RS_b2383	ts	$3,8 \times 10^4$	$16,2 \times 10^6$	424

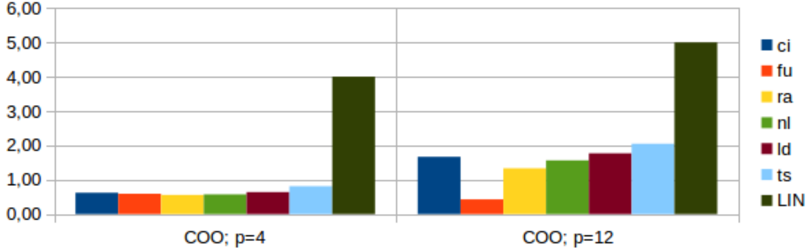







Sequential complexity of SpMVM in COO a CSR

	ci	fu	ra	nl	ld	ts
n [M]	5,5	3,0	4,7	3,5	0,95	0,038
N [M]	59,5	26,6	20,3	95,1	42,5	16,2
N/n	10,7	8,9	4,3	26,8	44,6	424
$T_{\text{COO}}(n)$ [ms]	194	82	56	283	143	69
$T_{\text{COO}}(n)/N$ [ms/M]	3,3	3,1	2,8	3,0	3,4	4,3
$T_{\text{CSR}}(n)$ [ms]	98	53	37	144	71	23
$T_{\text{CSR}}(n)/N$ [ms/M]	1,6	2,0	1,8	1,5	1,7	1,4

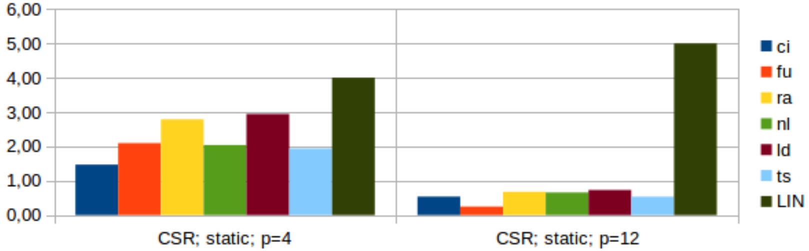
- CSR is sequentially approx. 2x faster than COO (CSR reads less data and performs less writes to shared memory).
- The relative performance of the SpMVM operation per 1 matrix element **depends** on the structure of non-zeros (due to cache misses).

Speedup of a parallel SpMVM: COO



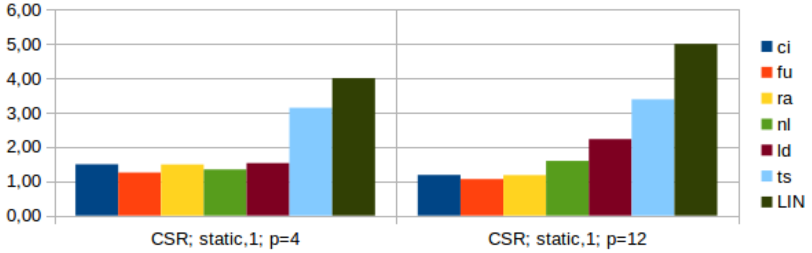
- Big overhead of atomic operations results in a **slowdown** for $p = 4$!!!!!
- $S(n, 12) \doteq 3 \times S(n, 4)$ (this is expected), except for matrix `fu` and $p = 12$, where we get a **slowdown**!!! (`fu` contains very dense rows and it leads to serialization of atomic accesses to the same address).
- But as a result, the parallelization for $p = 12$ is very inefficient, since we get only a slight speedup (at most 2) wrt the sequential times.
- The LIN bar should be 12 for $p = 12$ here and on the following slides. Just to enable smooth drawings it is cut to 5.

Speedup of a parallel SpMVM: CSR, schedule(static)



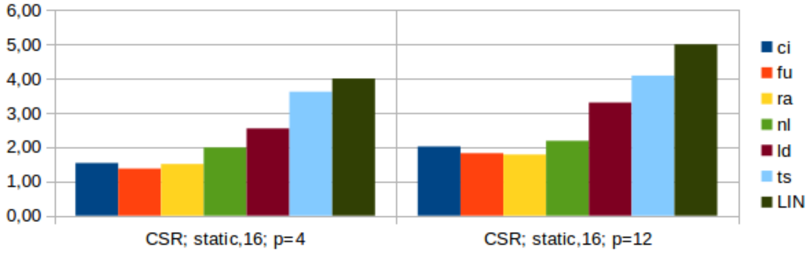
- $2 \leq S(n, 4) \leq 3$, which is quite a good result. The inefficiency is caused by unbalanced load and by saturation of the memory bus.
- $S(n, 12) \doteq S(n, 4)/3$!!! That is, 12 threads are cca 3× slower than 4 threads and finally even **slower** than the sequential algorithm.
- Two main factors why this solution with $p = 12$ fails:
  - Unbalanced load for $p = 12$ is **worse** than for $p = 4$ (esp. matrix fu).
  - The memory bus gets more saturated: 12 threads compete for 4 memory drivers.

Speedup of a parallel `SpMVM`: CSR, `schedule(static,1)`



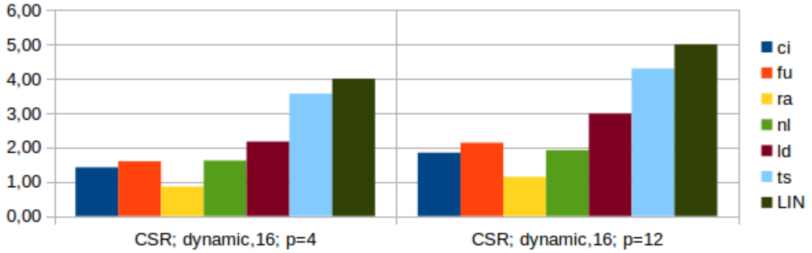
- **Row interleaving** leads to better load balancing in case of block-irregular matrices, but at the same time the **false sharing** arises. It becomes significant and the time gets worse than with `schedule(static)`, except for the densest matrix `ts`.
- 4 threads are a bit faster than sequential alg., across all matrices.
- Scaling to $p = 12$ is counterproductive for sparser matrices: 12 threads too often compete for 4 drivers. For denser matrices, this effect is better amortized with useful computation.

Speedup of a parallel `SpMVM`: CSR, `schedule(static,16)`



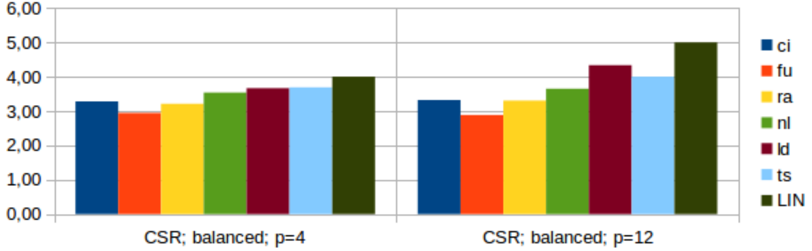
- Interleaving of row 16-tuples: improvement wrt the previous `(static,1)` since:
  - **there is no false sharing** and it slightly outweighs the effect of
  - **greater load imbalance** due to the coarser granularity of row 16-tuples.
- With $p = 12$, a small improvement appears. For denser matrices, it gets close to the saturation bound of 4 memory drivers.

Speedup of parallel SpMVM: CSR, `schedule(dynamic,16)`



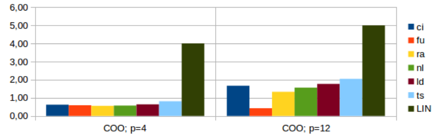
- Load balancing gets more data dependent: wrt `(static,16)` it can be better or worse.
- In every case, the runtime overhead of dynamic scheduling applies and in general, the time is worse wrt `schedule(static,16)`.
- For very sparse matrices (e.g., `ra`), the dynamic scheduling overhead dominates the time, it leads even to slowdown.
- Again, for denser matrices, the effect of dynamic scheduling overhead is amortized with useful computation.

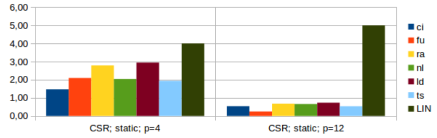
Speedup of a parallel SpMVM: CSR, balanced

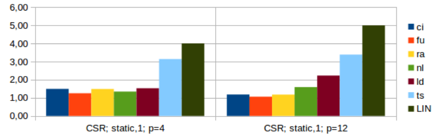


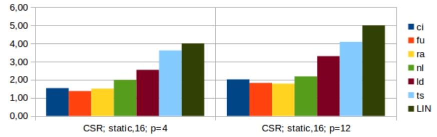
- Scheduling based on explicit algorithmic load balancing erases to some extent the structural differences among matrices.
- For $p = 4$, the speedup is nearly linear (4 memory drivers are saturated).
- For this reason, no speedup can appear with $p = 12$.
- It is the **best** variant in most cases, but the decomposition of the rows into equally filled bands cannot always achieve the best load balance (This appears, e.g., in case of matrix `fu`).

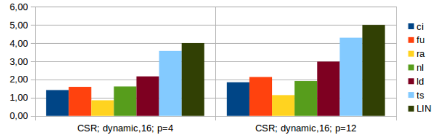
Comparison of speedups of parallel SpMVMs

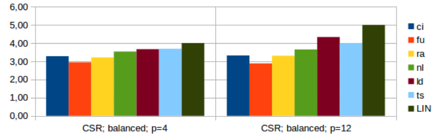












- Note that the SpMVM in the COO format performs N writes into the shared memory, whereas a SpMVM in the CSR format only n!!!!