
Parallel Processing

Winter Term 2025/26

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: December 15, 2025



Parallel Processing

Winter Term 2025/26

5 Optimization Techniques



- ➔ In the following: examples for important techniques to optimize parallel programs
- ➔ Shared memory:
 - ➔ cache optimization: improve the locality of memory accesses
 - ➔ loop interchange, tiling
 - ➔ array padding
 - ➔ false sharing
- ➔ Message passing:
 - ➔ combining messages
 - ➔ latency hiding

5.1 Cache Optimization



Example: summation of a matrix in C++ (👉 05/sum.cpp)

```
double a[N][N];
...
for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
        s += a[i][j];
    }
} column-wise traversal
```

```
double a[N][N];
...
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        s += a[i][j];
    }
} row-wise traversal
```

N=8192: Run time: 930ms

N=8193: Run time: 140 ms

Run time: 80ms

Run time: 80ms

(bspc02,
g++ -O3)

➡ Reason: caches

➡ higher hit rate when matrix is traversed row-wise

➡ although each element is used only once ...

➡ Remark: C/C++ stores a matrix row-major, Fortran column-major



Details on caches: cache lines

- ➔ Storage of data in the cache and transfer between main memory and cache are performed using larger blocks
 - ➔ reason: after a memory cell has been addressed, the subsequent cells can be read very fast
 - ➔ size of a cache line: 32-128 Byte
- ➔ In the example:
 - ➔ row-wise traversal: after the cache line for $a[i][j]$ has been loaded, the values of $a[i+1][j]$, $a[i+2][j]$, ... are already in the cache, too
 - ➔ column-wise traversal: the cache line for $a[i][j]$ has already been evicted, when $a[i+1][j]$, ... are used
- ➔ **Rule:** traverse memory in linearly increasing order, if possible!



Details on caches: set-associative caches

- ➔ A memory block (with given address) can be stored only at a few places in the cache
 - ➔ reason: easy retrieval of the data in hardware
 - ➔ usually, a set has 2 to 16 entries
 - ➔ the entry within a set is determined using the LRU strategy
- ➔ The lower k Bits of the address determine the set (k depends on cache size and degree of associativity)
 - ➔ for all memory locations, whose lower k address bits are the same, there are only 2 - 16 possible cache entries!



Details on caches: set-associative caches ...

- ➔ In the example: with $N = 8192$ and column-wise traversal
 - ➔ a cache entry is guaranteed to be evicted after a few iterations of the i -loop (address distance is a power of two)
 - ➔ cache hit rate is very close to zero
- ➔ **Rule:** when traversing memory, avoid address distances that are a power of two!
 - ➔ (avoid powers of two as matrix size for large multi-dimensional matrices)



Important cache optimizations

- ➔ **Loop interchange**: swapping of loops
 - ➔ such that memory is traversed in linearly increasing order
 - ➔ with C/C++: traverse matrices row-wise
 - ➔ with Fortran: traverse matrices column-wise
- ➔ **Array padding**
 - ➔ if necessary, allocate matrices larger than necessary, in order to avoid a power of two as the length of each row
- ➔ **Tiling**: blockwise partitioning of loop iterations
 - ➔ restructure algorithms in such a way that they work as long as possible with sub-matrices, which fit completely into the caches



Example: Matrix multiply

( 05/matmult.c)

➔ Naive code:

```
double a[N][N], b[N][N], c[N][N]
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        for (int k=0; k<N; k++)
            c[i][j] += a[i][k] * b[k][j];
```

➔ Performance with different compiler optimization levels:
(N=384, g++ 12.2.0, Intel Core i7 @ 3.4 GHz (bspc02))

➔ -O0: 0.4 GFlop/s

➔ -O: 1.6 GFlop/s

➔ -O2: 3.2 GFlop/s

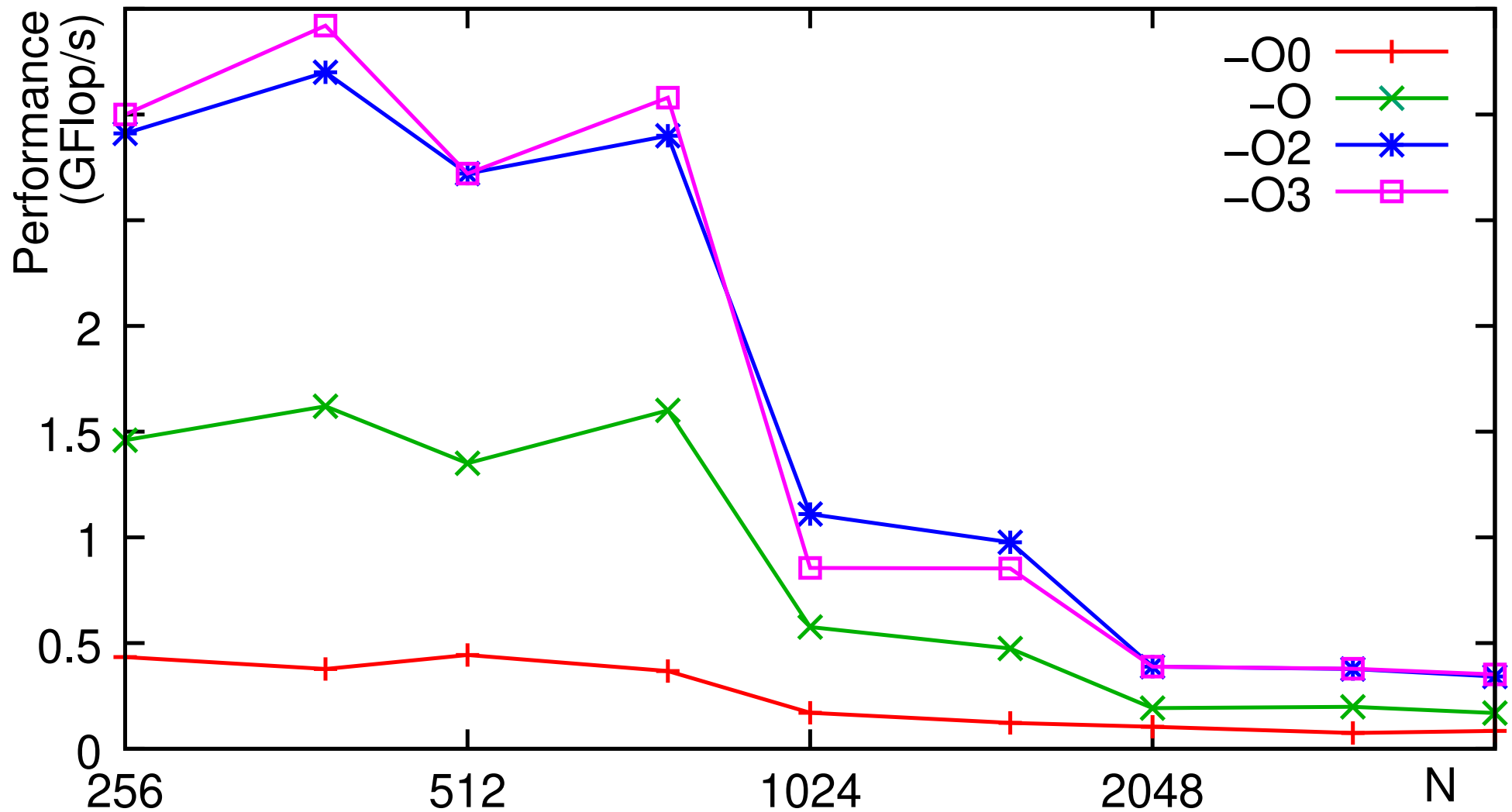
➔ -O3: 3.4 GFlop/s

5.1 Cache Optimization ...



Example: Matrix multiply ...

➔ Scalability of the performance for different matrix sizes:



Example: Matrix multiply ...

➔ Optimized order of the loops:

```
double a[N][N], b[N][N], c[N][N]
for (int i=0; i<N; i++)
    for (int k=0; k<N; k++)
        for (int j=0; j<N; j++)
            c[i][j] += a[i][k] * b[k][j];
```

➔ Matrix b now is traversed row-wise

➔ considerably less L1 cache misses

➔ substantially higher performance:

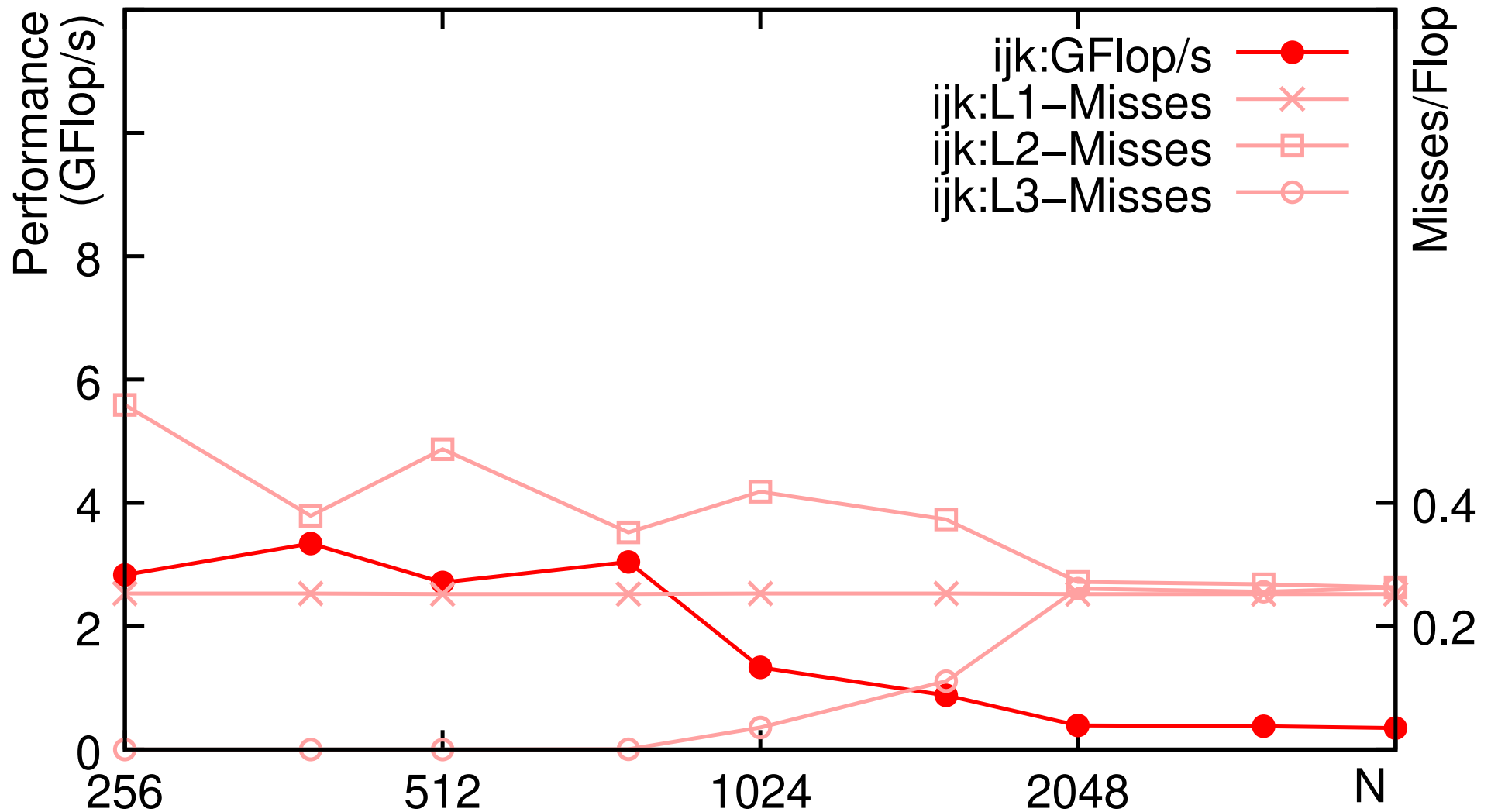
➔ N=384, -O3: 9.8 GFlop/s instead of 3.4 GFlop/s

➔ considerably better scalability



Example: Matrix multiply ...

➔ Comparison of both loop orders:

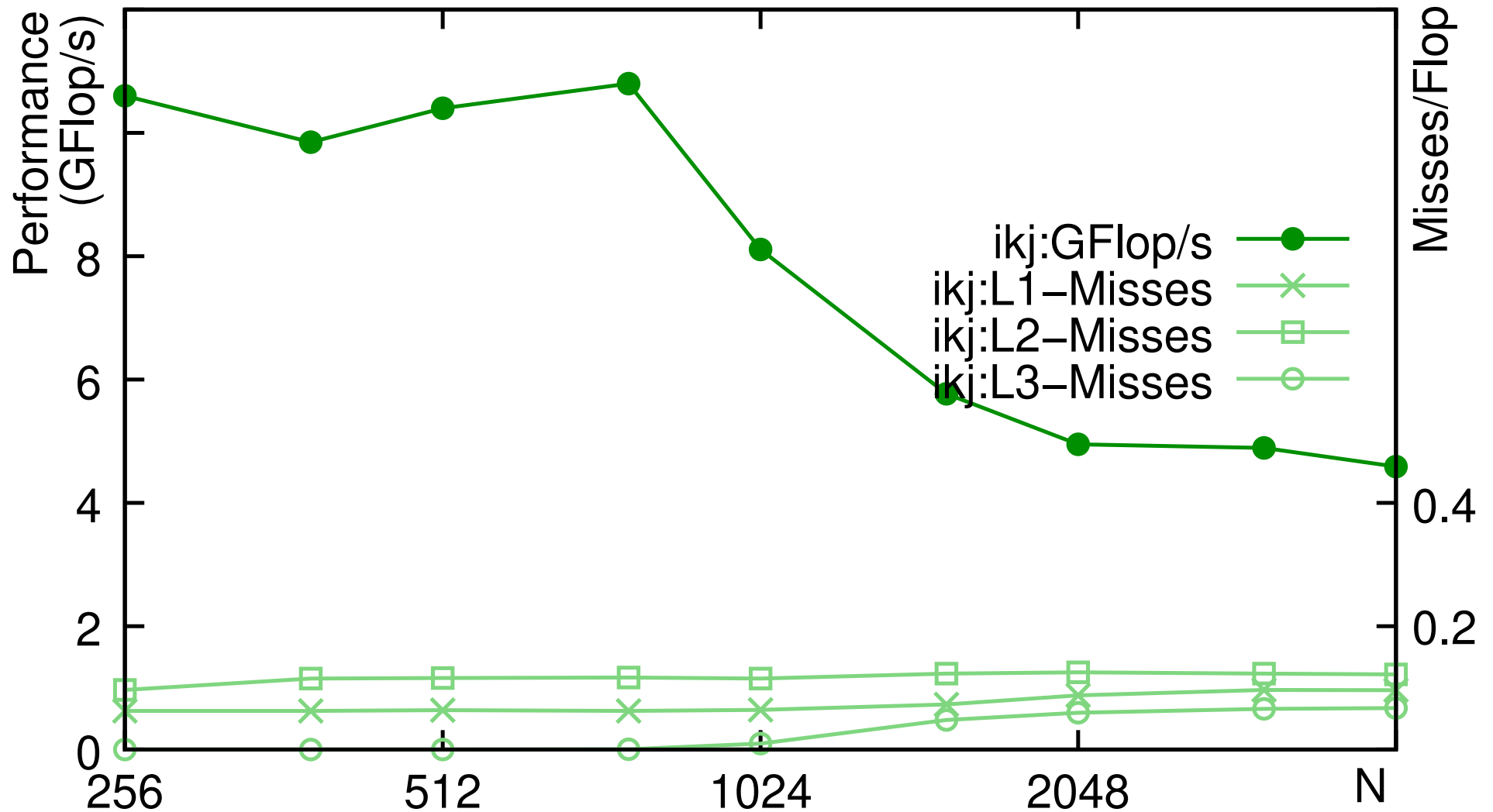


5.1 Cache Optimization ...



Example: Matrix multiply ...

➔ Comparison of both loop orders:





Example: Matrix multiply ...

➔ Block algorithm (tiling) with array padding:

```
double a[N][N+8], b[N][N+8], c[N][N+8]
for (int ii=0; ii<N; ii+=16)
  for (int kk=0; kk<N; kk+=16)
    for (int jj=0; jj<N; jj+=16)
      for (int i=0; i<16; i++)
        for (int k=0; k<16; k++)
          for (int j=0; j<16; j++)
            c[i+ii][j+jj] += a[i+ii][k+kk] * b[k+kk][j+jj];
```

➔ Matrix is viewed as a matrix of 16×16 sub-matrices

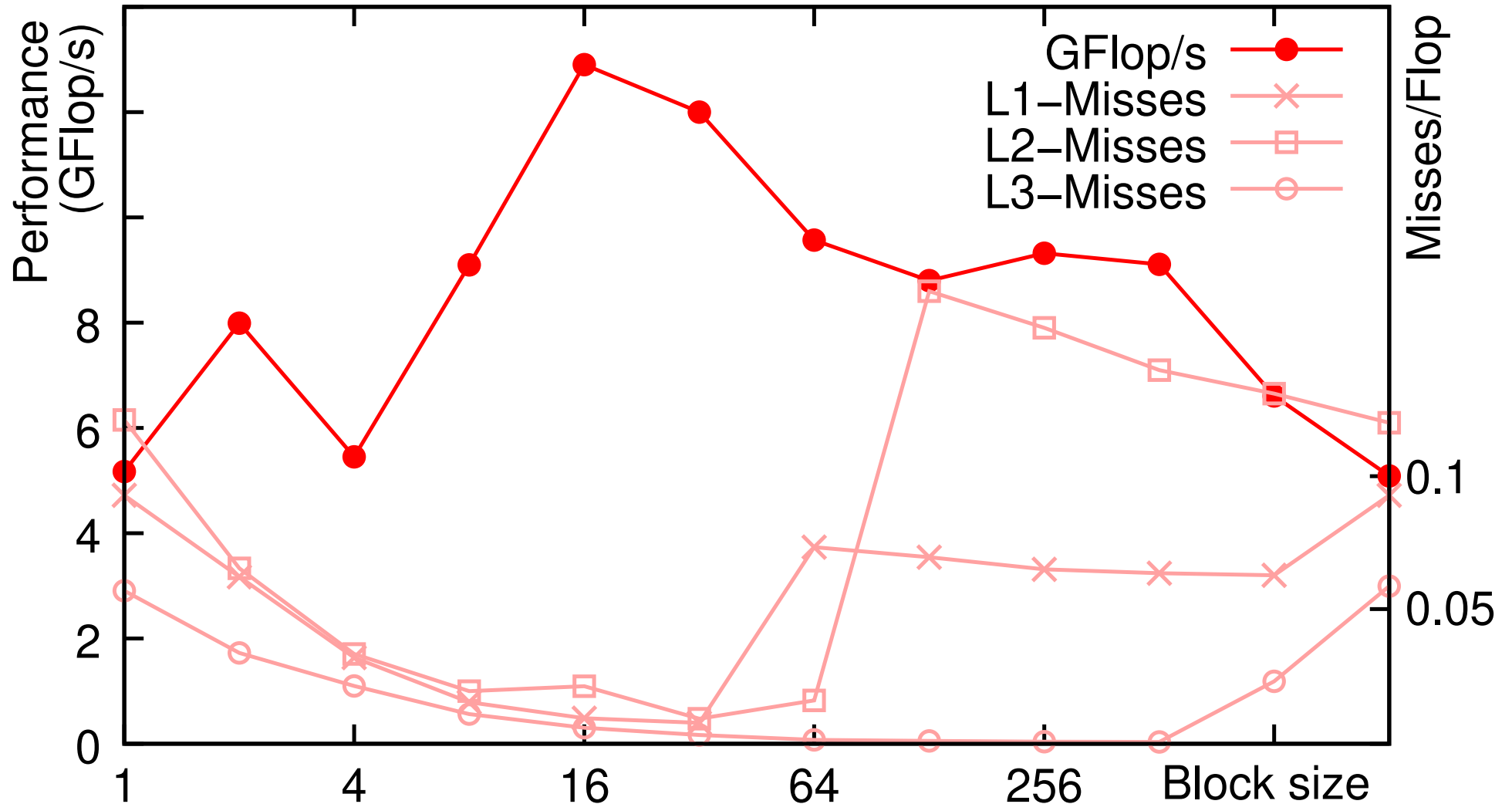
➔ multiplication of sub-matrices fits into the L1 cache

➔ Achieves a performance of 13 GFlop/s even with $N=2048$



Example: Matrix multiply ...

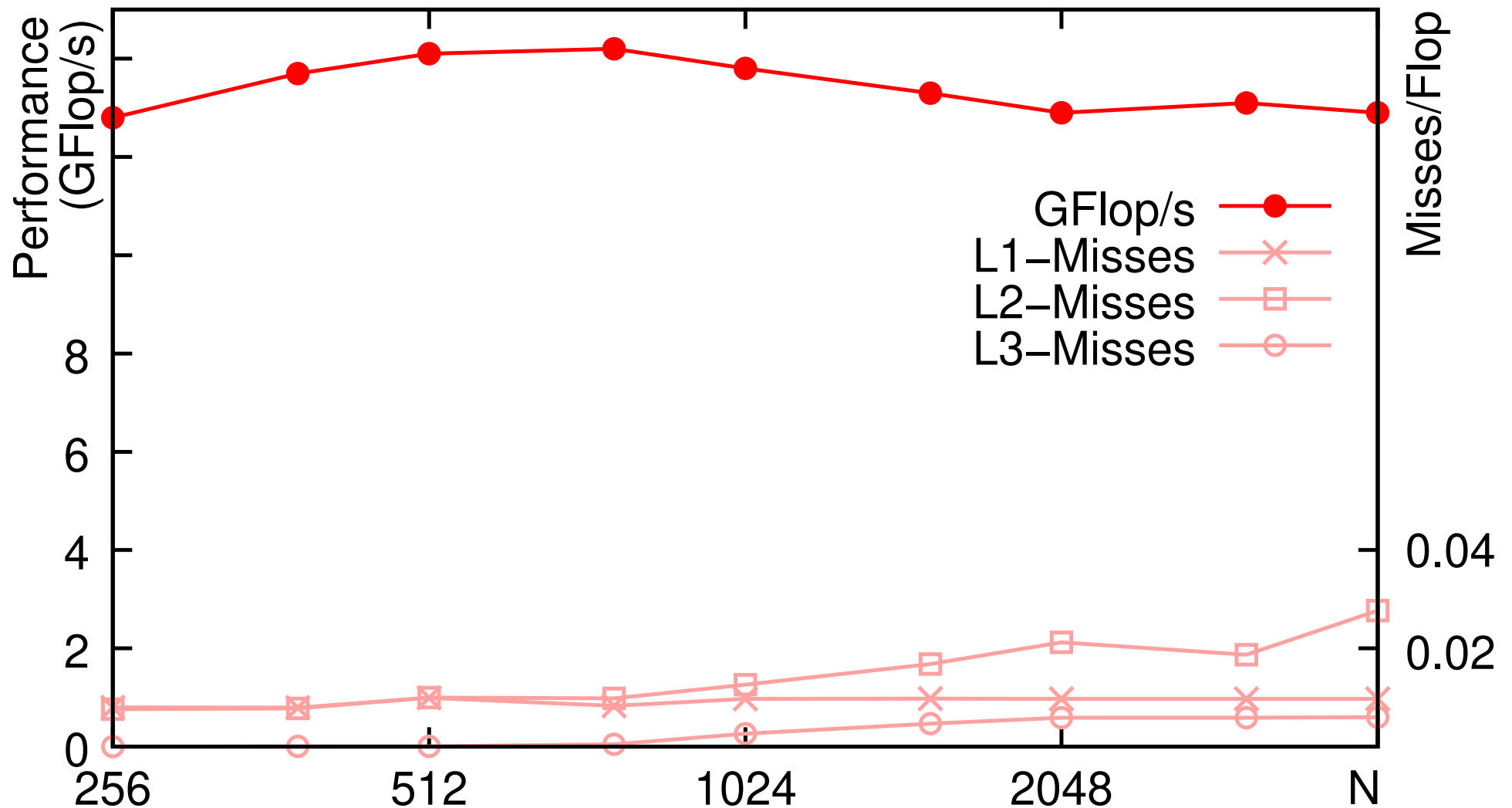
➔ Performance as a function of block size (N=2048):





Example: Matrix multiply ...

➔ Scalability of performance for different matrix sizes:

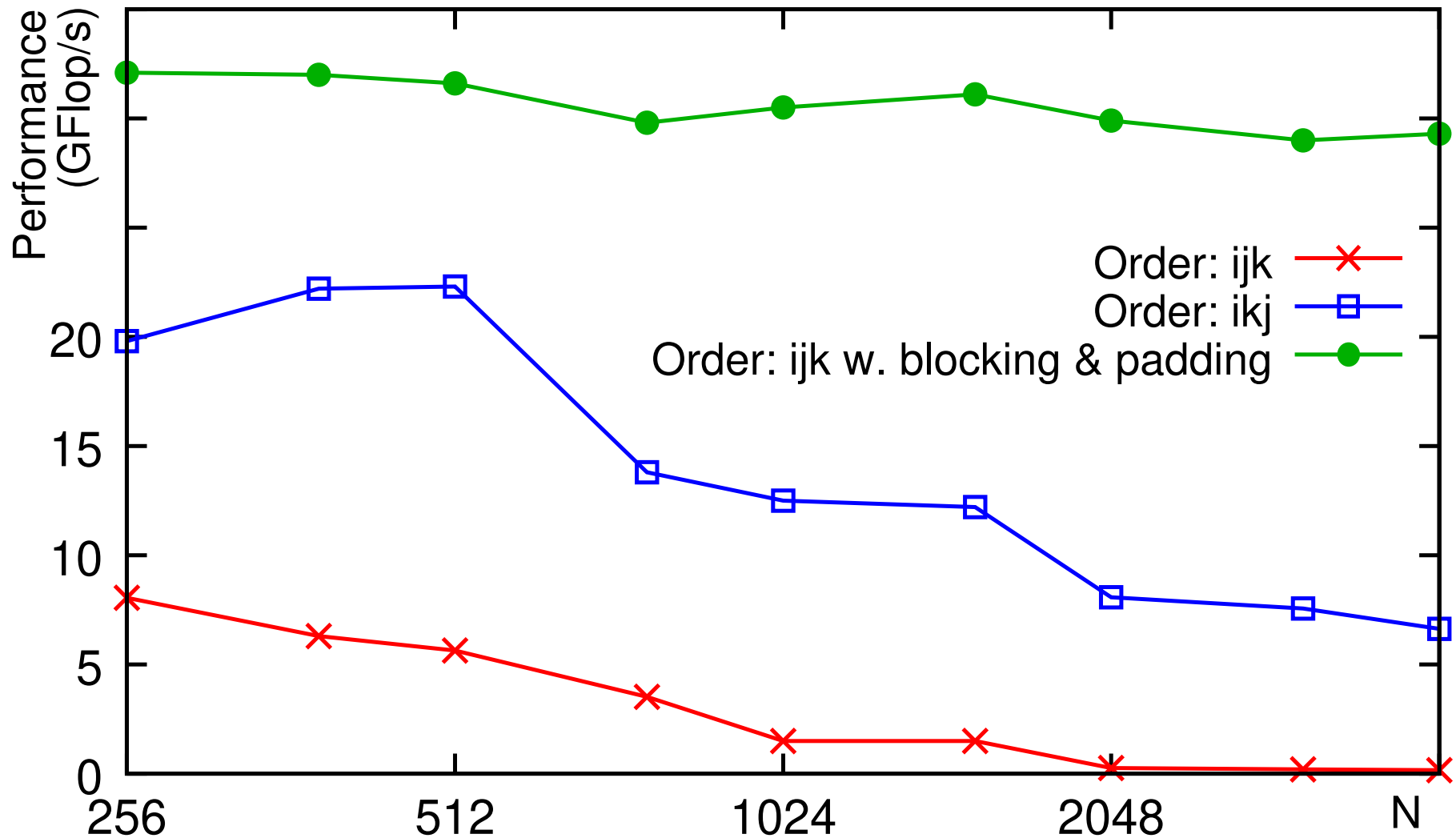


5.1 Cache Optimization ...



Example: Matrix multiply ...

➔ Results in the lab: (Intel Core Ultra 7 265, max. 5.2 GHz; -O3):





Cache optimization for parallel computers

- ➔ Cache optimization is especially important for parallel computers (UMA and NUMA)
 - ➔ larger difference between the access times of cache and main memory
 - ➔ concurrency conflicts when accessing main memory
- ➔ Additional problem with parallel computers: **false sharing**
 - ➔ several variables, which do not have a logical association, can (by chance) be stored in the same cache line
 - ➔ write accesses to these variables lead to frequent cache invalidations (due to the cache coherence protocol)
 - ➔ performance degrades drastically



Example for false sharing: parallel summation of an array

(👉 05/false.cpp)

- ➔ Global variable `double sum[NUM_THREADS]` for the partial sums
- ➔ Version 1: thread `i` adds to `sum[i]`
 - ➔ run-time(*) with 4 threads: 0.21 s, sequentially: 0.17 s !
 - ➔ performance loss due to false sharing: the variables `sum[i]` are located in the same cache line
- ➔ Version 2: thread `i` first adds to a local variable and stores the result to `sum[i]` at the end
 - ➔ run-time(*) with 4 threads: 0.043 s
- ➔ **Rule:** variables that are used by different threads should be separated in main memory (e.g., use padding)!

(*) 8000 x 8000 matrix, Intel Core i7, 2.8 GHz, without compiler optimization

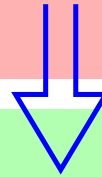


Combining messages

- ➔ The time for sending short messages is dominated by the (software) latency
 - ➔ i.e., a long message is “cheaper” than several short ones!
- ➔ Example: PC cluster in the lab H-A 4111 with MPICH2
 - ➔ 32 messages with 32 Byte each need $32 \cdot 145 = 4640\mu s$
 - ➔ one message with 1024 Byte needs only $159\mu s$
- ➔ Thus: combine the data to be sent into as few messages as possible
 - ➔ where applicable, this can also be done with communication in loops (hoisting)

Hoisting of communication calls

```
for (i=0; i<N; i++) {      for (i=0; i<N; i++) {
    b = f(..., i);          recv(&b, 1, P1);
    send(&b, 1, P2);         a[i] = a[i] + b;
}                            }
```

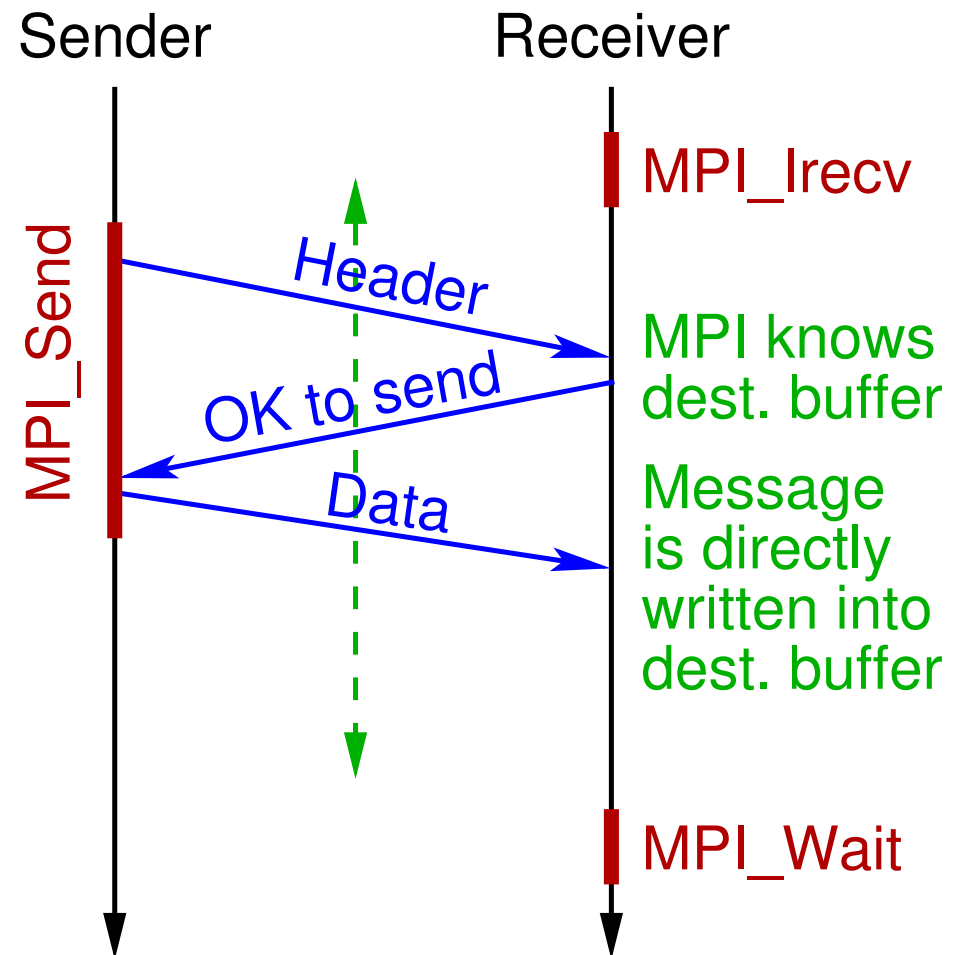


```
for (i=0; i<N; i++) {      recv(b, N, P1);
    b[i] = f(..., i);       for (i=0; i<N; i++) {
}                            a[i] = a[i] + b[i];
send(b, N, P2);            }
```

- ➔ Send operations are hoisted past the end of the loop, receive operations are hoisted before the beginning of the loop
- ➔ Prerequisite: variables are not modified in the loop (sending process) or not used in the loop (receiving process)

Latency hiding

- ➔ Goal: hide the communication latency, i.e., overlap it with computations
- ➔ As early as possible:
 - ➔ post the receive operation (MPI_Irecv)
- ➔ Then:
 - ➔ send the data
- ➔ As late as possible:
 - ➔ finish the receive operation (MPI_Wait)





- ➔ Take care of good locality (caches)!
 - ➔ traverse matrices in the order in which they are stored
 - ➔ avoid powers of two as address increment when sweeping through memory
 - ➔ use block algorithms
- ➔ Avoid false sharing!
- ➔ Combine messages, if possible!
- ➔ Use latency hiding when the communication library can execute the receipt of a message “in background”
- ➔ If send operations are blocking: execute send and receive operations as synchronously as possible