Parallel Processing

WS 2017/18

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

Stand: January 15, 2018

Parallel Processing

WS 2017/18

4 Optimization Techniques

4 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

4.1 Cache Optimization

Example: summation of a matrix in C++ (☞ 04/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

	column–wise traversal	row–wise traversal
N=8192:	Run time: 4,15 s	Run time: 0,14 s
N=8193:	Run time: 0,72 s	Run time: 0,14 s

(bsclk01, 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 8 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 - 8 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 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

Parallel Processing

WS 2017/18

15.01.2018

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

Stand: January 15, 2018

4.1 Cache Optimization ...

Example: Matrix multiply (☞ 04/matmult.c)

➥ Naive code:

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

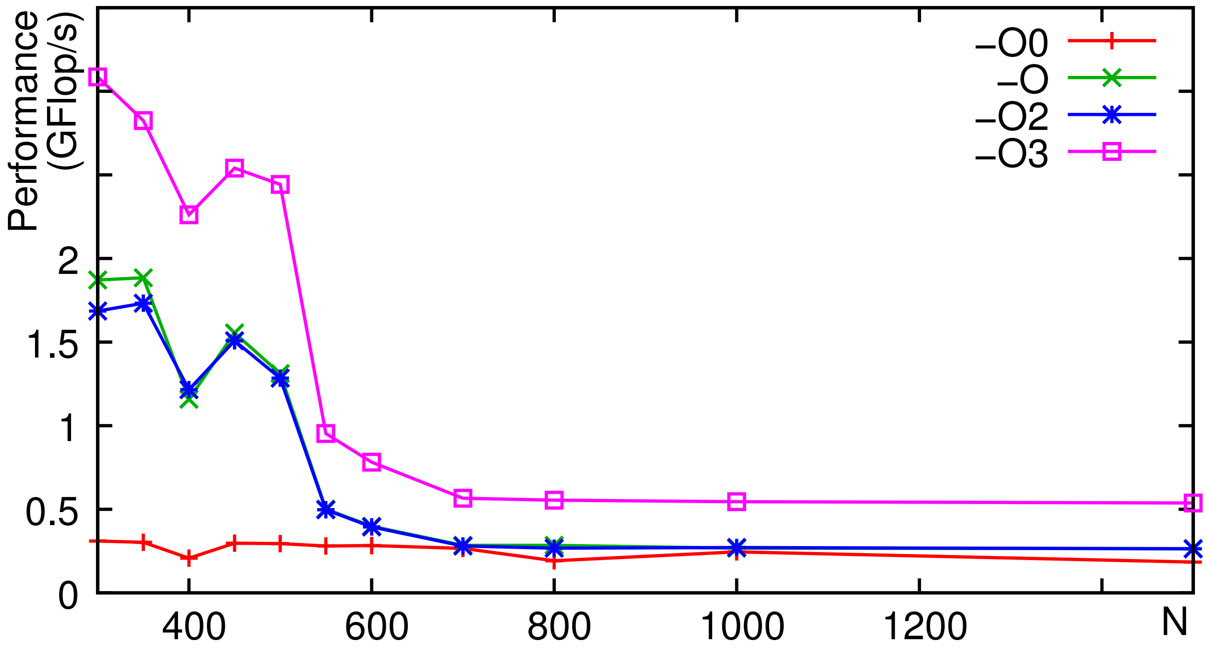
➥ Performance with different compiler optimization levels: (N=500, g++ 4.6.3, Intel Core i7 2.8 GHz (bspc02))

- ➥ -O0: 0.3 GFlop/s
- ➥ -O: 1.3 GFlop/s
- ➥ -O2: 1.3 GFlop/s
- ➥ -O3: 2.4 GFlop/s (SIMD vectorization!)

4.1 Cache Optimization ...

Example: Matrix multiply ...

- Scalability of the performance for different matrix sizes:



4.1 Cache Optimization ...

Example: Matrix multiply ...

- Optimized order of the loops:

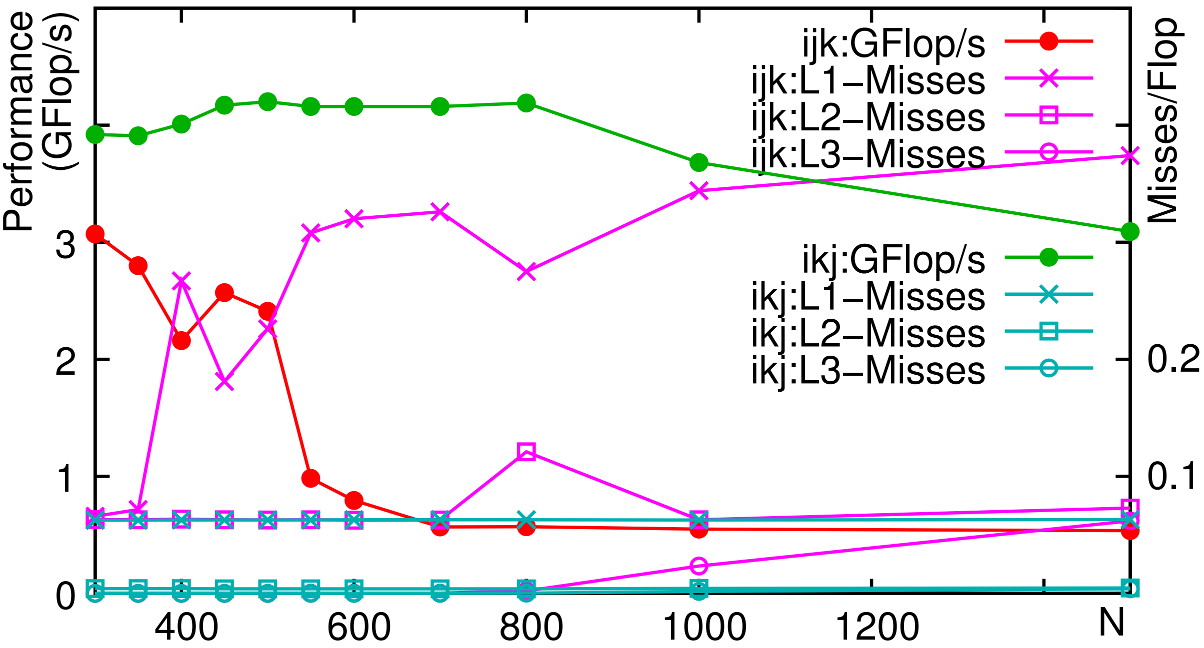
```
double a[N][N], b[N][N], ...
for (i=0; i<N; i++)
  for (k=0; k<N; k++)
    for (j=0; j<N; j++)
      c[i][j] += a[i][k] * b[k][j];
```

- Matrix b now is traversed row-wise
  - considerably less L1 cache misses
  - substantially higher performance:
    - N=500, -O3: 4.2 GFlop/s instead of 2.4 GFlop/s
  - considerably better scalability

(Animated slide)

Example: Matrix multiply ...

- Comparison of both loop orders:



Notes for slide 326:

The decrease in performance of the 'ikj' loop order for larger matrices is due to an increase in the L3 misses (which is not visible in the figure due to the scaling).

Parallel Processing

WS 2017/18

22.01.2018

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

Stand: January 15, 2018

4.1 Cache Optimization ...

Example: Matrix multiply ...

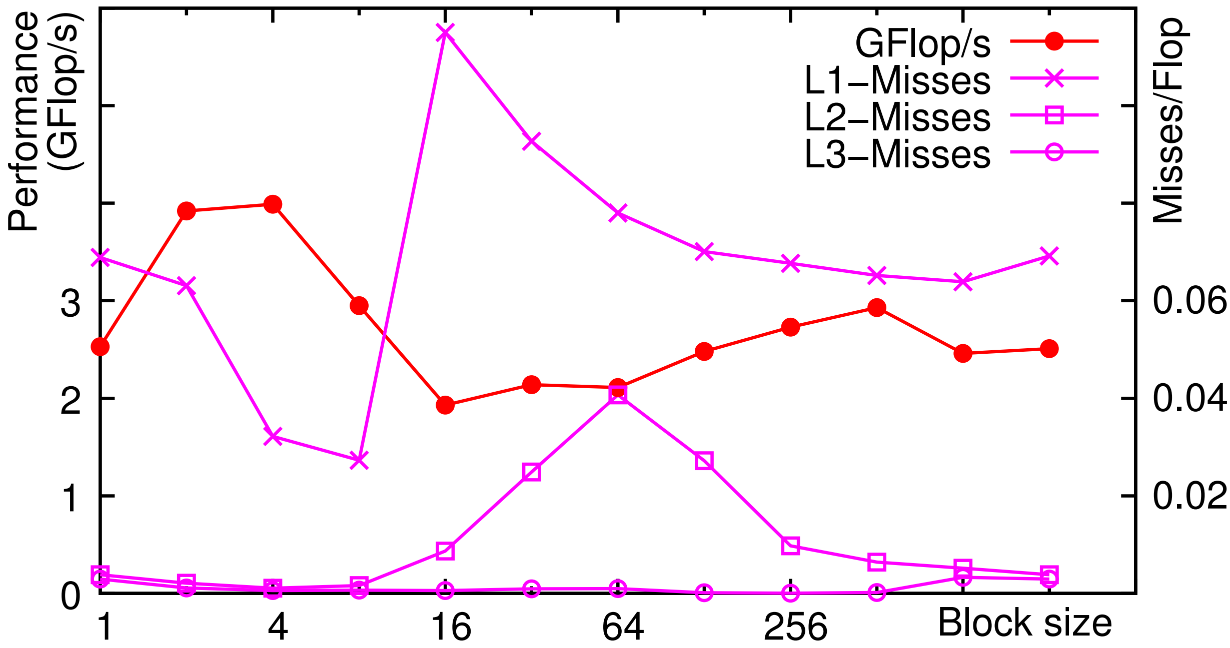
- Block algorithm (tiling) with array padding:

```
double a[N][N+1], b[N][N+1], ...
for (ii=0; ii<N; ii+=4)
 for (kk=0; kk<N; kk+=4)
  for (jj=0; jj<N; jj+=4)
   for (i=0; i<4; i++)
    for (k=0; k<4; k++)
     for (j=0; j<4; j++)
      c[i+ii][j+jj] += a[i+ii][k+kk] * b[k+kk][j+jj];
```

- Matrix is viewed as a matrix of 4x4 sub-matrices
  - multiplication of sub-matrices fits into the L1 cache
- Acheives a performance of 4 GFlop/s even with N=2048

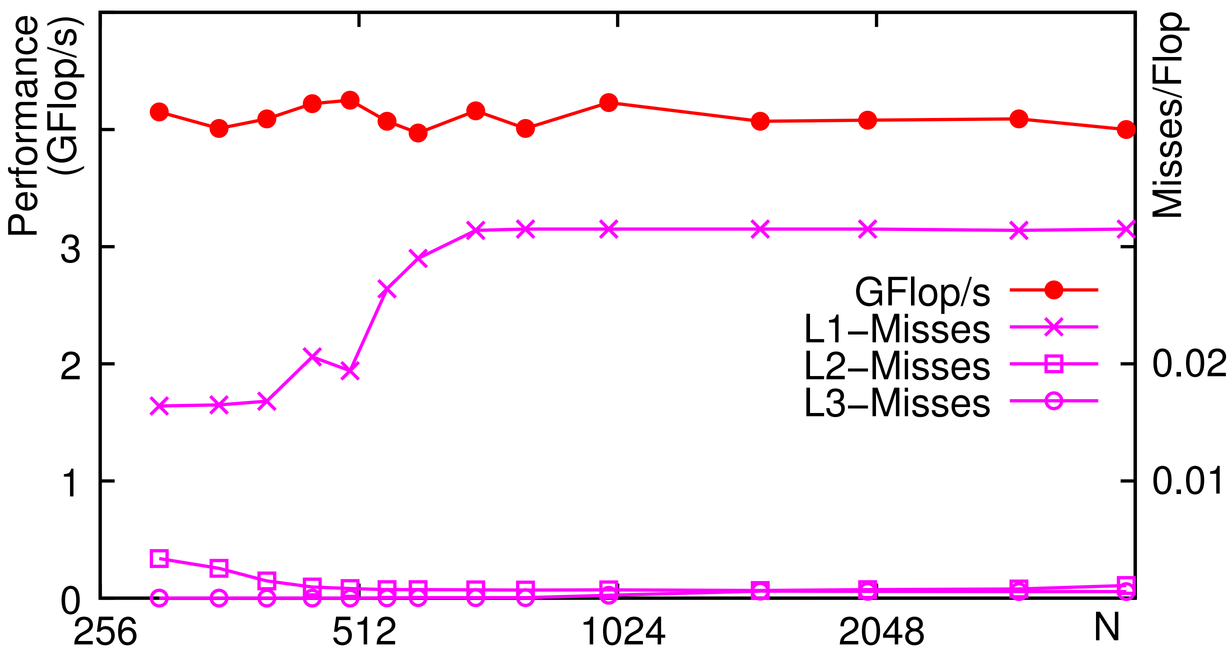
Example: Matrix multiply ...

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



Example: Matrix multiply ...

➥ Scalability of performance for different matrix sizes:



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

(☞ `04/false.cpp`)

- Global variable `double sum[P]` for the partial sums
- Version 1: thread `i` adds to `sum[i]`
  - run-time(*) with 4 threads: 0.4 s, sequentially: 0.24 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: 0.09 s
- **Rule**: variables that are used by different threads should be separated in main memory (e.g., use padding)!

(*) 8000 x 8000 matrix, Intel Xeon 2.66 GHz, without compiler optimization

Notes for slide 331:

When compiler optimization is enabled with `gcc`, the run-time of the parallel program in version 1 is reduced to 0.11 s (version 2: 0.09 s, sequentially: 0.22 s)

4.2 Optimization of Communication

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)

4.2 Optimization of Communication ...

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;
}                              }
```

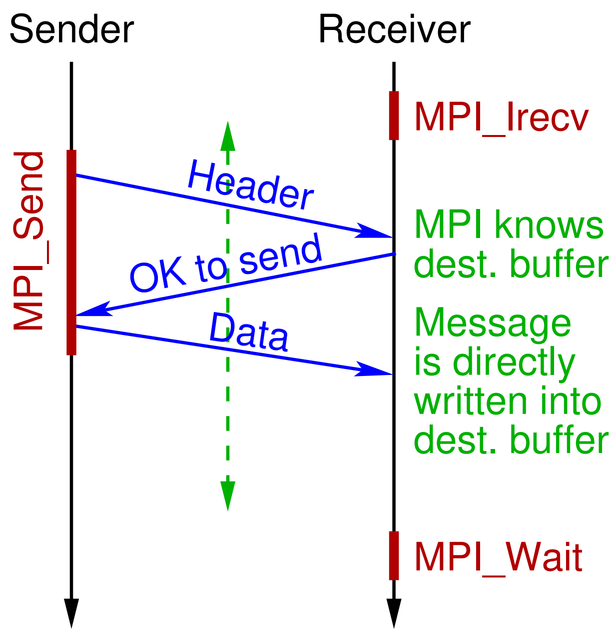
```
for (i=0; i<N; i++) {          recv(b, N, P1);
  b[i] = f(..., i);            for (i=0; i<N; i++) {
}                                a[i] = a[i] + b[i];
send(b, N, P2);                }
```

- 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)

4.2 Optimization of Communication ...

Latency hiding

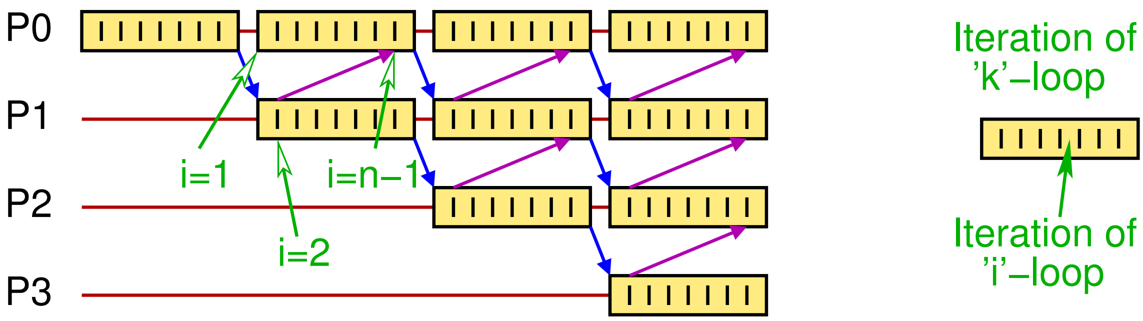
- Goal: hide the communication latency, i.e., overlap it with computations
- As early as possible:
  - post the receive operation (MPI_Irecv)
- Then:
  - send the data
- As late as possible:
  - finish the receive operation (MPI_Wait)



4.3 A Story from Practice

Gauss/Seidel with MPICH (version 1) on Intel Pentium 4

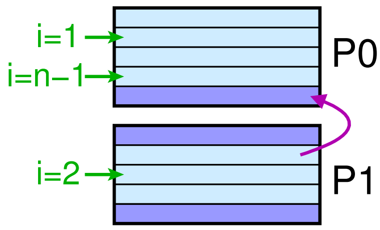
➥ Intended course of execution of the parallel program:



Flow of the communication → :

i=1: MPI_Irecv()
i=2: MPI_Bsend()
i=n−1: MPI_Wait()

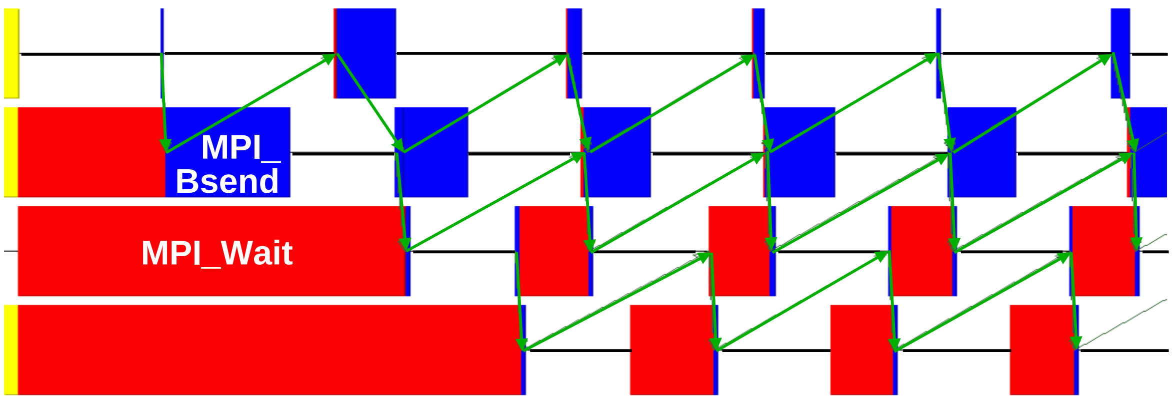
} at the beginning of the 'i'−loop



4.3 A Story from Practice ...

Gauss/Seidel with MPICH (version 1) on Intel Pentium 4 ...

➥ Actual temporal behavior (Jumpshot):



➥ Speedup only 2.7 (4 proc., 4000x4000 matrix, run-time: 12.3s)

➥ MPI_Bsend (buffered send) blocks! Why?

Communication in MPICH-p4

- The MPI implementation MPICH-p4 is based on TCP/IP
- MPICH-p4 retrieves the messages from the operating system's TCP buffer and copies it to the process's receive buffer
- However, the MPI library can do this, only if the process periodically calls (arbitrary) MPI routines
  - during the compute phase of Gauss/Seidel this is, however, not the case
- If the TCP buffer is not emptied:
  - the TCP buffer becomes full
  - TCP flow control blocks the sender process

Notes for slide 337:

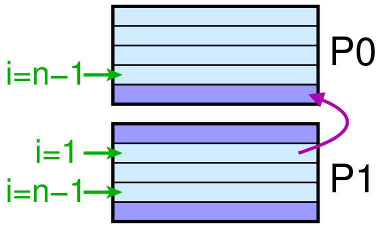
MPICH2 ensures the progress of communication even without periodic MPI calls.

4.3 A Story from Practice ...

Gauss/Seidel: improvements

- In order to ensure the progress of communication:
  - insert calls to MPI_Test into the computation
  - improves run-time to 11.8s, speedup is 2.85
  - problem: overhead due to the calls to MPI_Test
- Different approach: tightly synchronized communication

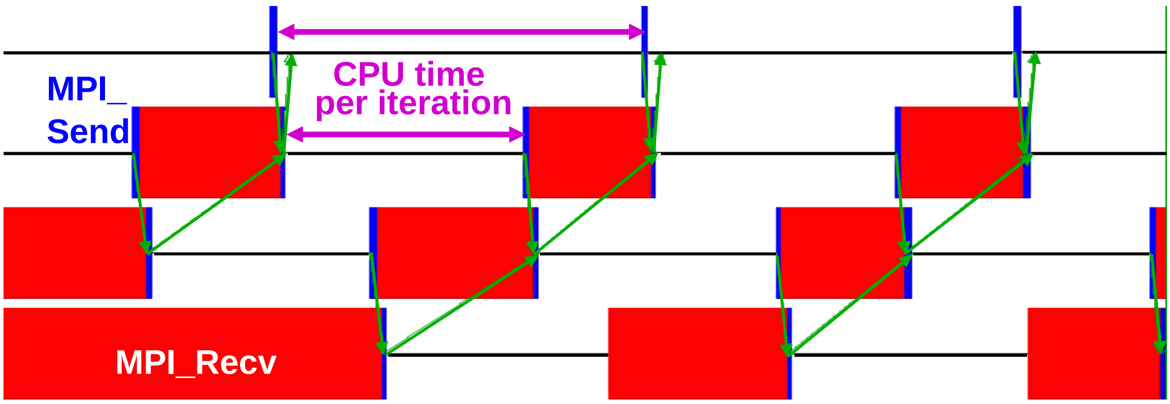
i=n−1: MPI_Send()
i=n−1: MPI_Recv()
} at the beginning of the 'i'–loop



  - run-time: 11.6s, speedup 2.9
  - drawback: performance is sensitive to delays, e.g., background load on the nodes, network load

4.3 A Story from Practice ...

Gauss/Seidel: result



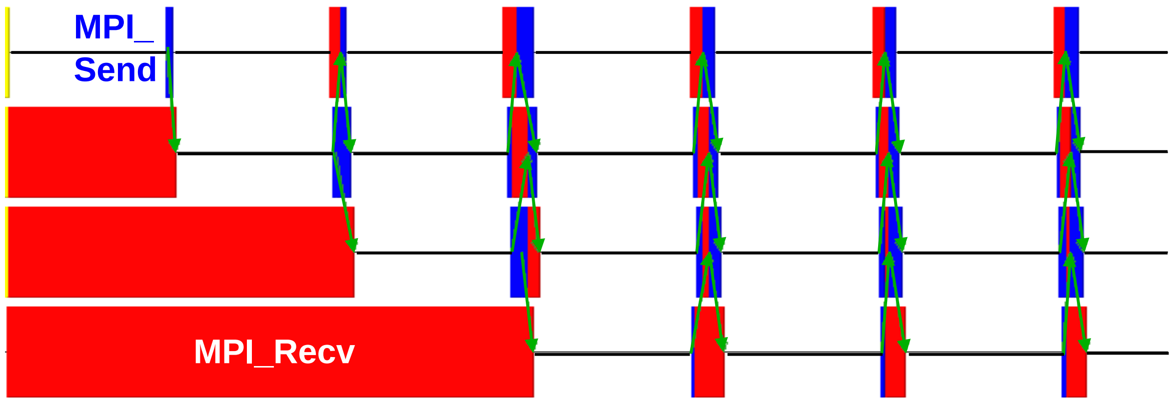
- Load imbalance in spite of uniform distribution of the matrix!
  - reason: the arithmetic of the Pentium 4 is extremely slow with denormalized numbers (range $10^{-308} - 10^{-323}$)
  - e.g., addition of 10^9 numbers: 220 s instead of 9 s!

Notes for slide 339:

Current Intel-CPUs (and AMD-CPUs) do no longer show this behavior. Today, the floating point unit is completely realized in hardware (the Pentium 4 implemented the arithmetic for denormalized numbers in micro-code).

4.3 A Story from Practice ...

Gauss-Seidel: success!



- When initializing the matrix with 10^{-300} instead of 0, the problem disappears
- Run-time: 7.0 s, speedup: 3.4
- Sequential run-time now only 23.8 s instead of 33.6 s

4.3 A Story from Practice ...

Lessons learned:

- Latency hiding only works reasonably, when the progress of the communication is ensured
  - e.g., with MPI over Myrinet: the network interface card writes the arriving data directly into the process's receive buffer
  - or with MPICH2 (separate thread)
- Tightly synchronized communication can be better, but is susceptible to delays
- Load imbalance can also occur when you don't expect it
  - the execution times of modern processors are unpredictable

4.4 Summary

- Take care of good locality (caches)!
  - traverse matrices in the oder 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