RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

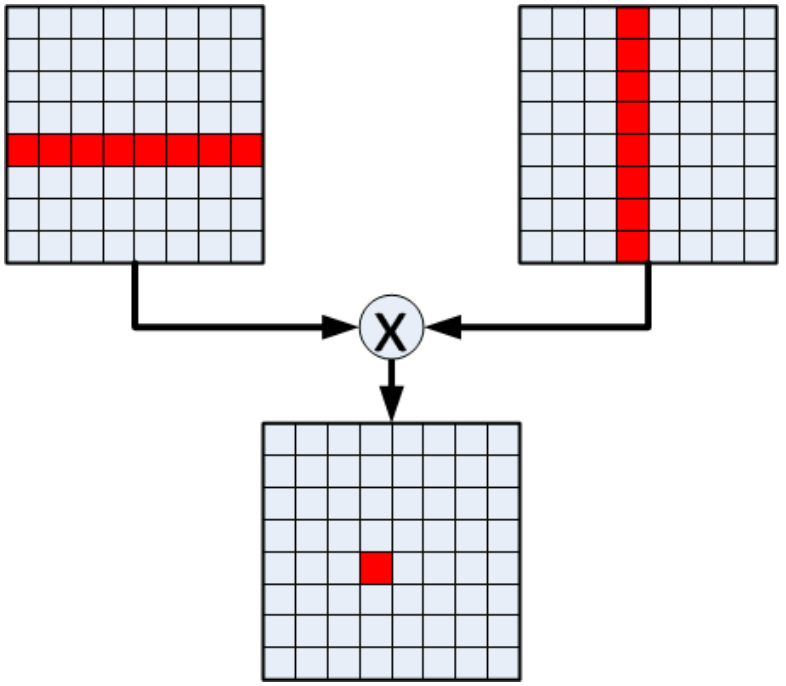
# Introduction to High Performance Computing

*Lecture 05 – Practical Parallel Programming Example*

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg

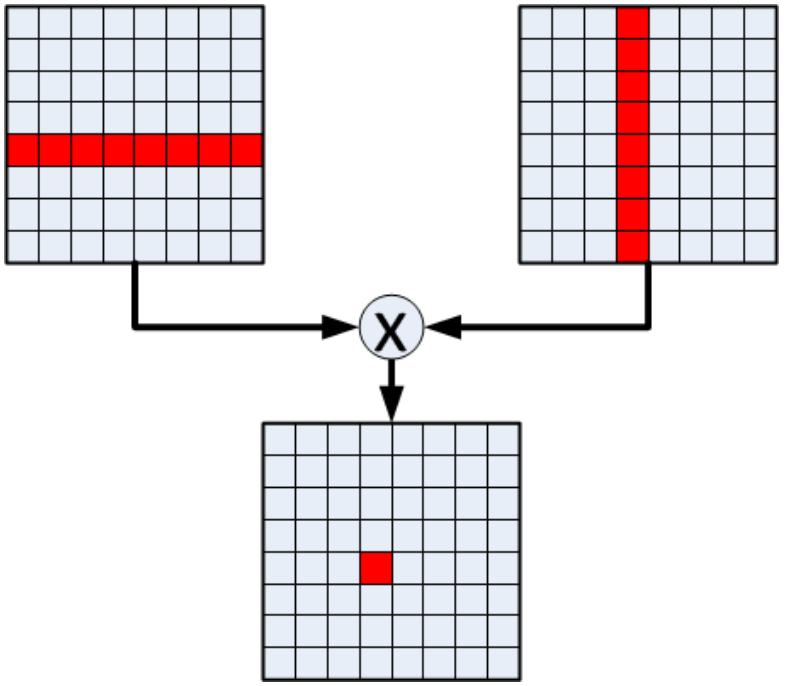
# Matrix Multiply

- Why always Matrix Multiply?
  - One of the most heavily optimized codes in HPC
  - Interesting access patterns
  - Good mixture of sufficient complexity but still simple enough for a comprehensive understanding
  - Finally, it‘s an important operation!
- Used in many applications as computational kernel
  - In particular for sparse matrix operations
- Here: for dense matrices
  - Experiments and learning
  - High sustained/peak ratio
  - Test system/compiler/OS
- Note on notation
  - M[row,column] = M[row][column]
  - Analogous to C



# Matrix Multiply – Sequential version

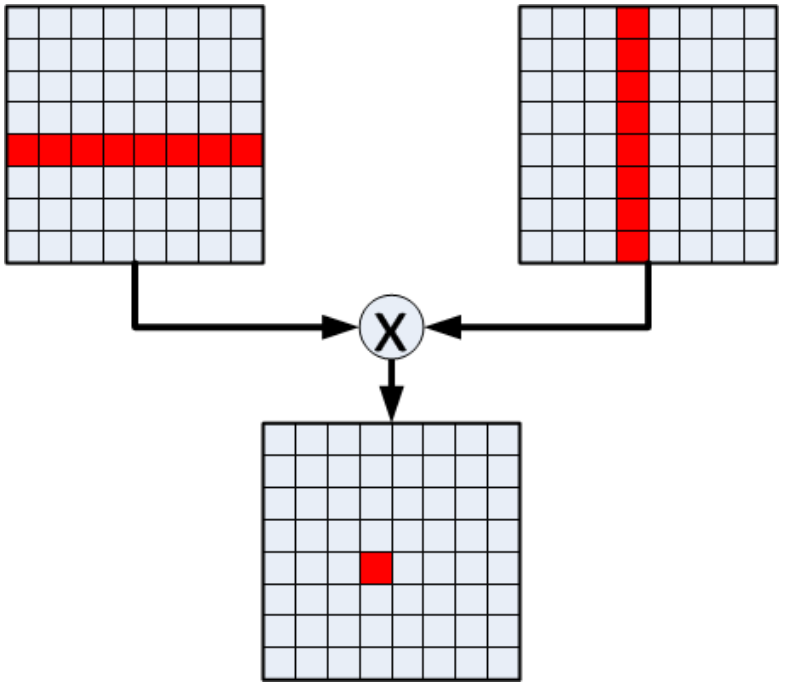
- C = A[][] * B[][]
  - A[i,j] == A[i][j]
- Locality
  - Row-major order storage for C programming language
- Caching effects
  - Nice for A, bad for B
  - True/False sharing
  - Potentially evicts other useful blocks (3C)
    - Compulsory
    - Conflict
    - Capacity




```
for i in n
  for j in n
    for k in n
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

# Matrix Multiply - Analysis

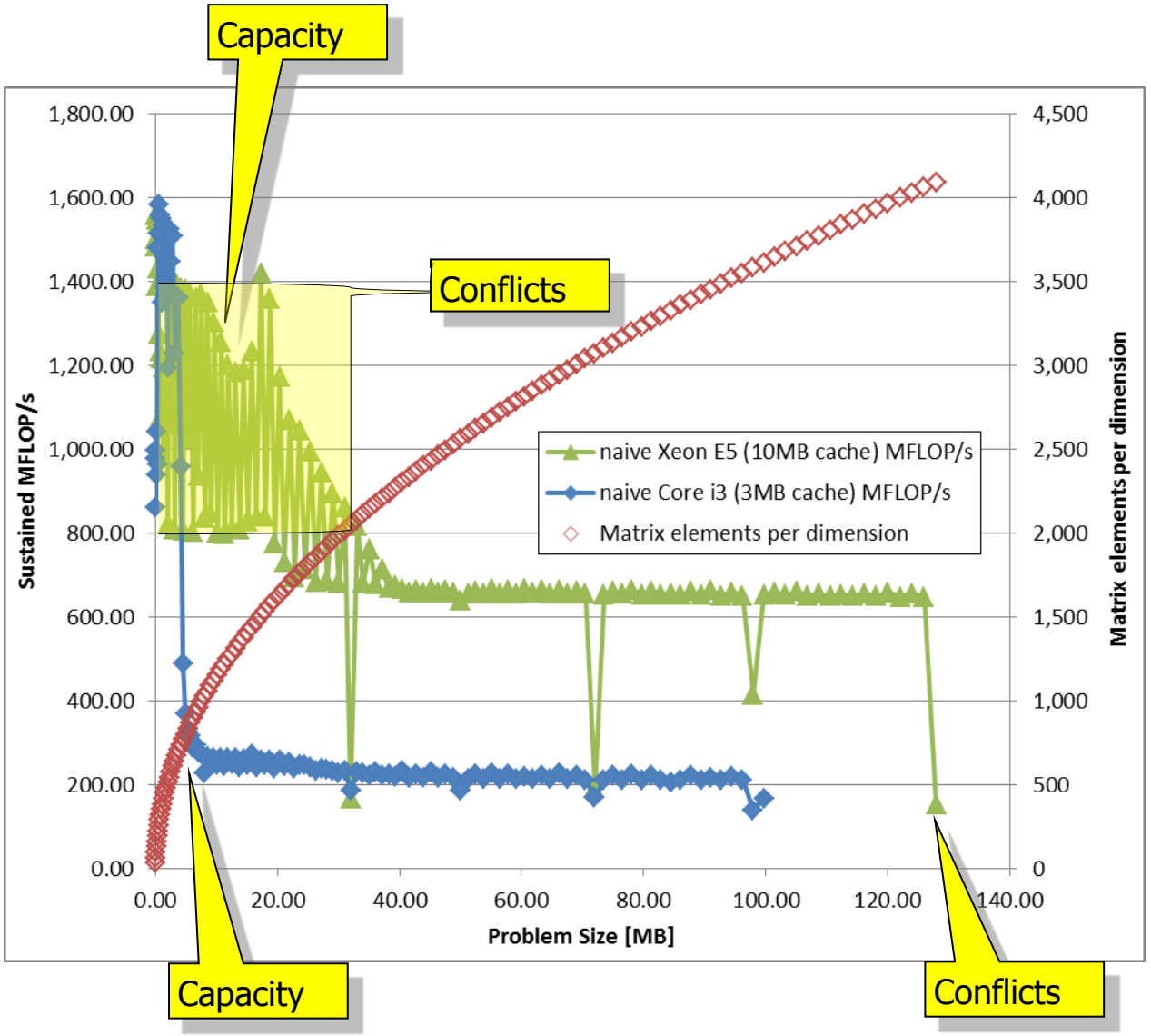
- Analysis
  - Assume square matrices
  - Assume perfect write-through cache (no conflict, no capacity)
- Number of flops: $f = 2*N^3$
  - $N^2$ elements in C, each N steps
  - Each step: multiply & add
- Number of cache misses: $m = 3*N^2$
  - Load from A,B,C, store to C
  - Counting all accesses: $4*N^3$
- Ratio mem/flop: $r = m/f = 2/N = O(1/N)$
  - Computationally intensive
  - Peak performance expected for cache-based processor




```
for i in n
  for j in n
    for k in n
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

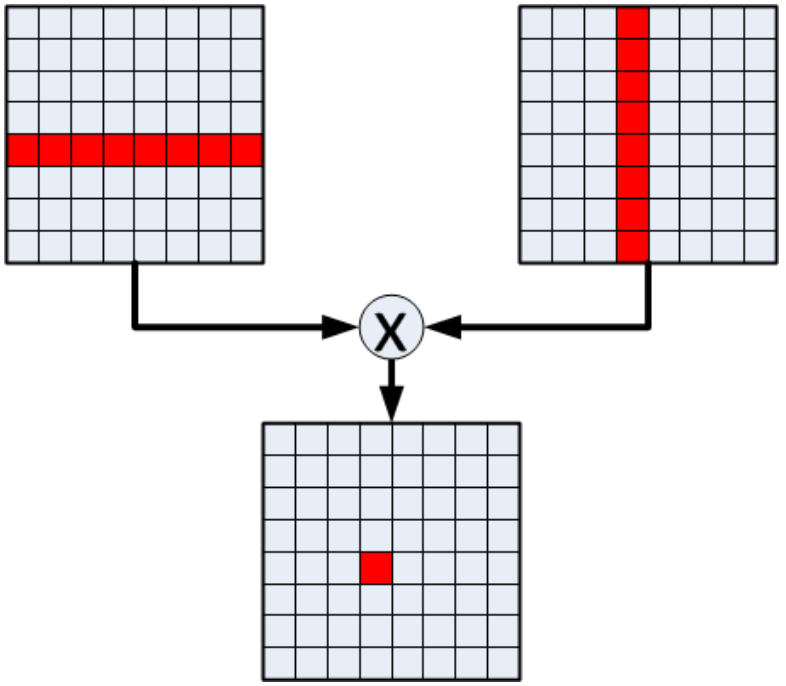
# Matrix Multiply - Baseline Performance

- Core i3, 3.1GHz dual core, AVX
  - 49.6 GFLOP/s peak
- Xeon E5, 2.4GHz quad core, AVX
  - 76.8 GFLOP/s peak
- Size = 3 matrices each $N^2$
- Flops = $2N^3$
- Nice caching effects visible
  - Core i3
  - Xeon E5



# Matrix Multiply – Analysis

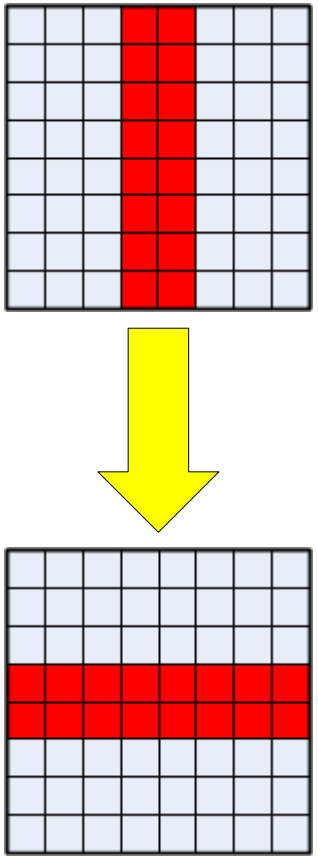
- Suffers from non-linear access to B
  - „Pseudo-Random“ from the cache point of view
  - Neither spatial nor temporal locality exploitable
- r = O(1/N) only holds for optimal caching
- Otherwise:
  - $m_{uncached} = 3*N^3$ for all accesses (worst caching)
  - $F = 2*N^3$
  - $r = m/f = 2$ or O(1)




```
for i in n
  for j in n
    for k in n
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

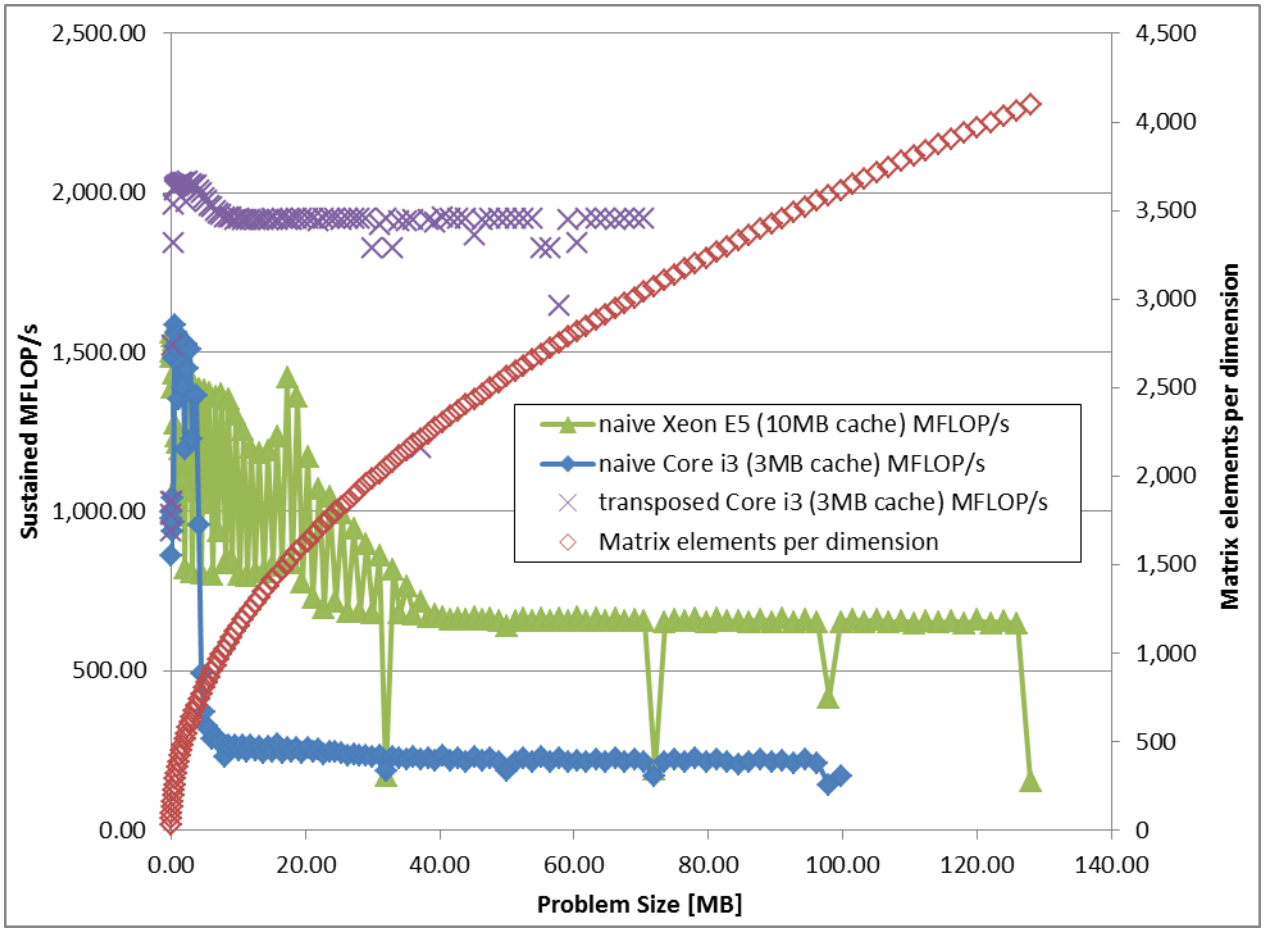
# Matrix Multiply – Seq. Version First Optimization

- Solution: transpose B
  - In a linear load fashion, written back with stride
  - Optimized for caches
- Costs:
  - $f_t = 0$
  - $m_t = 2N^2$
- New overall r:
  - $r = (m_{cached} + m_t) / (f + f_t)$
    $= (3N^2 + 2N^2) / (2*N^3 + 0)$
    $= 5/(2N)$
    $= O(1/N)$



# Matrix Multiply – Seq. Version First Optimization

- Theoretical peak in this case (single threaded, no AVX or SSE)
  - Core i3: 3.1 x 2 (MADD) = 6.2 GFLOP/s



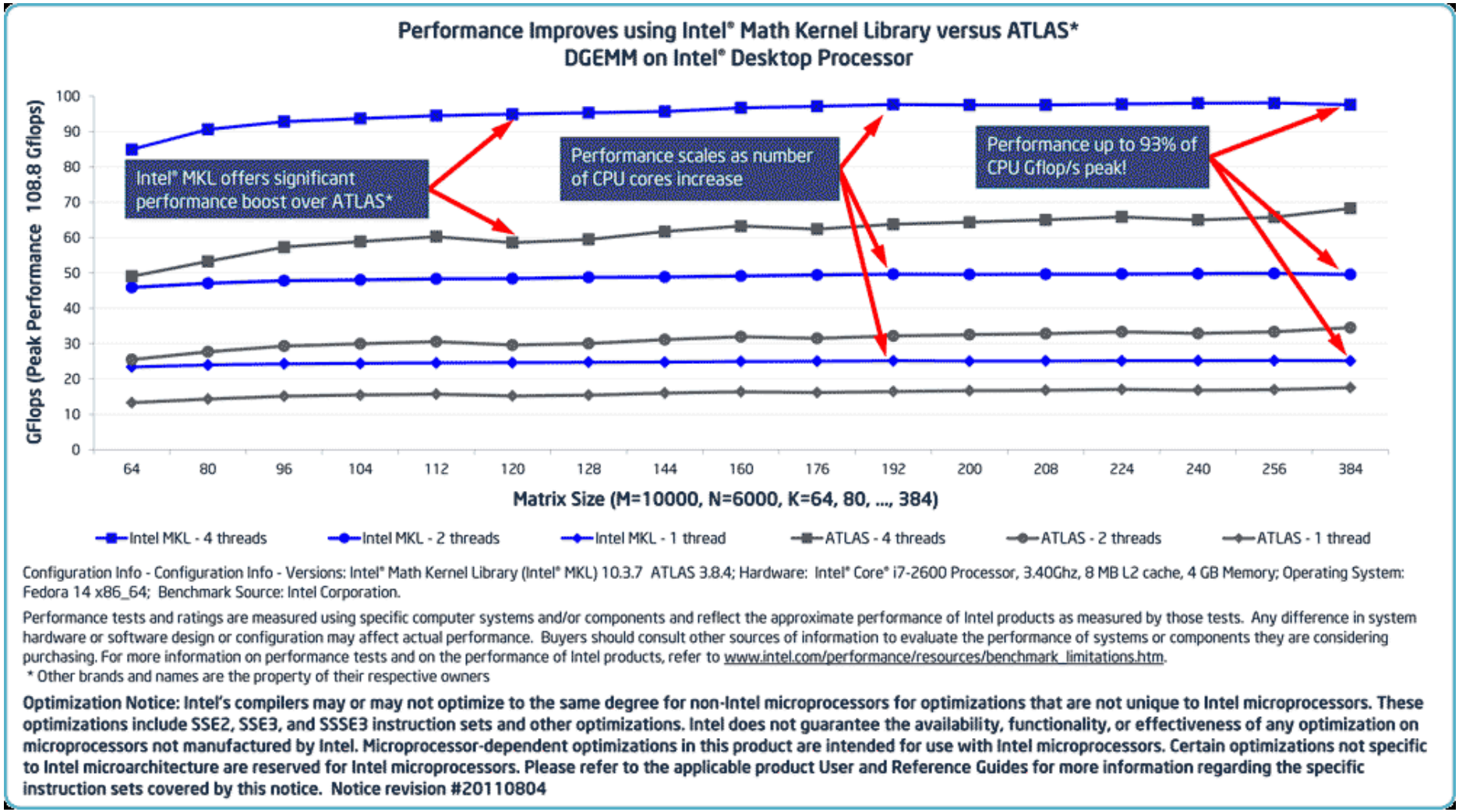
# Tiling/Blocking

- Increase locality by reordering memory accesses
  - Associativity
  - Tiling or blocking
  - Each TxT tile uses each element T times
- Calculate only parts of the elements of C, so that access pattern has high locality
  - Beneficial for both sequential and parallel algorithms

Time

| $C_{0,0}$ | $C_{1,0}$ | $C_{0,1}$ | $C_{1,1}$ |
|---|---|---|---|
| $B_{0,0} * A_{0,0}$ | $B_{0,0} * A_{1,0}$ | $B_{0,1} * A_{0,0}$ | $B_{0,1} * A_{1,0}$ |
| $B_{1,0} * A_{0,1}$ | $B_{1,0} * A_{1,1}$ | $B_{1,1} * A_{0,1}$ | $B_{1,1} * A_{1,1}$ |
| $B_{2,0} * A_{0,2}$ | $B_{2,0} * A_{1,2}$ | $B_{2,1} * A_{0,2}$ | $B_{2,1} * A_{1,2}$ |
| $B_{3,0} * A_{0,3}$ | $B_{3,0} * A_{1,3}$ | $B_{3,1} * A_{0,3}$ | $B_{3,1} * A_{1,3}$ |

# Commercial BLAS library




Configuration Info - Configuration Info - Versions: Intel® Math Kernel Library (Intel® MKL) 10.3.7 ATLAS 3.8.4; Hardware: Intel® Core® i7-2600 Processor, 3.40Ghz, 8 MB L2 cache, 4 GB Memory; Operating System: Fedora 14 x86_64; Benchmark Source: Intel Corporation.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/performance/resources/benchmark_limitations.htm.

* Other brands and names are the property of their respective owners

**Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804**

Source: http://software.intel.com/en-us/articles/intel-mkl

Per core: achieved about 25 GFLOP/s out of 27.2 GFLOP/s peak

# Parallelization

# Parallelization

- ***Now:*** local and remote accesses
  - Distinguish between local accesses $\mathbf{m_l}$ and remote accesses $\mathbf{m_r}$
  - Remote access much worse than local access
- Serial version:
  - Due to memory gap, ***f/m*** crucial for overall performance
- Parallel version:
  - Hierarchical approach, first optimize $\mathbf{m_r}$, then $\mathbf{m_l}$
  - Remote communication more important for overall performance

# Parallel MMULT - Parallelization

- PCAM: Partition, Communicate, Agglomerate, Map
- **Partition**
  - Domain decomposition of C (output matrix)
  - Each C[i,j] assigned to one task
- Resulting **Communication** pattern depends on distribution of input matrices A,B
  1. Copies: no communication except for initialization/completion
     - For large problem sizes not possible (capacity constraints)
     - Huge initialization overhead, $2N^2$ elements to copy
  2. Partitioning of A,B according to C: same as unique memory references ($m = 4N^2$), but here m = number of (potential) messages
     - Depends on tiling/blocking
- **Agglomeration**
  - Increase computational load per thread/process
  - Reduce communication overhead
- **Mapping**
  - Easy as communication pattern is rather simple (global communication dominates)

# Parallel MMULT - Overlap

- Overlap between computation and communication
  - For complete matrix copies, dependencies will prevent overlap
    - 3 clearly separated steps: distribute, compute, collect
  - For partitioning, fine grain communication allows for overlap:
    - While the first blocks are processed, further distribution of input matrices can already take place
- Goal is provide as much overlap as possible
  - Latency hiding resp. hiding of communication costs
  - Blocking/Tiling

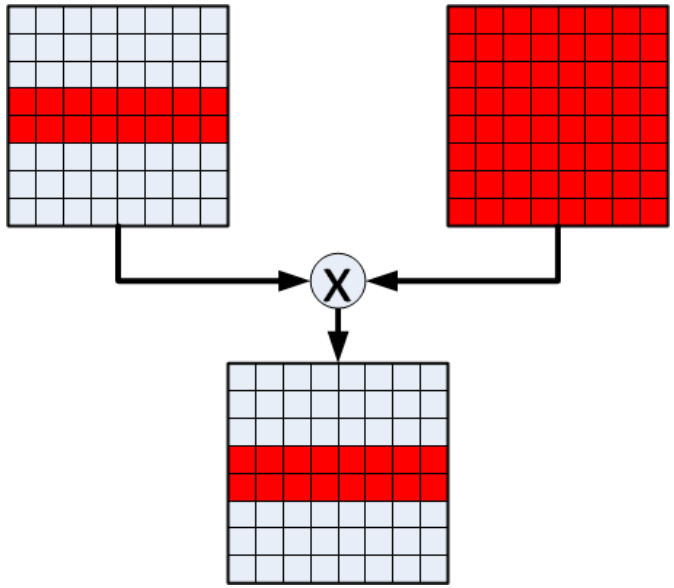
# Parallel MMULT – Data distribution

- Data distribution primary concern for agglomeration
  - 1D partitioning by blocks of rows or columns
  - 2D partitioning by both blocks of rows and columns
- Optimal choice depends on algorithm
  - Locality, communication pattern and dependencies

- Here: 1D (row and column) and 2D analyzed
- In general for MPI: owner computes and handles associated communication

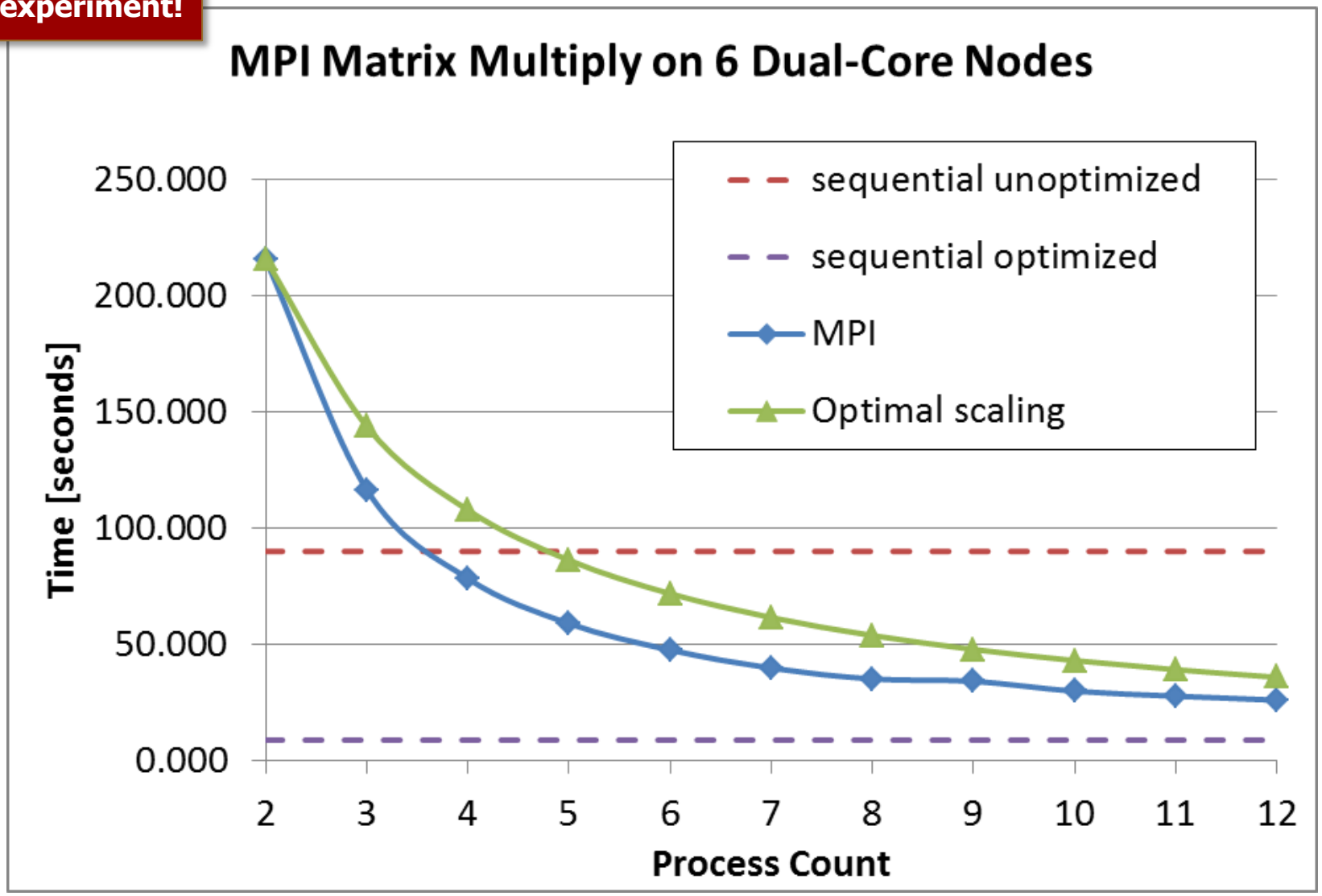
# Parallel MMULT – 1D row partitioning

- Data layout: process i owns row(s) C[i,*] (short: C[i,])
- Owner computes: process i calculates C[i,]
- C[i,] = A [i,]*B[,]
  - B has to be owned completely, but only parts of A and C



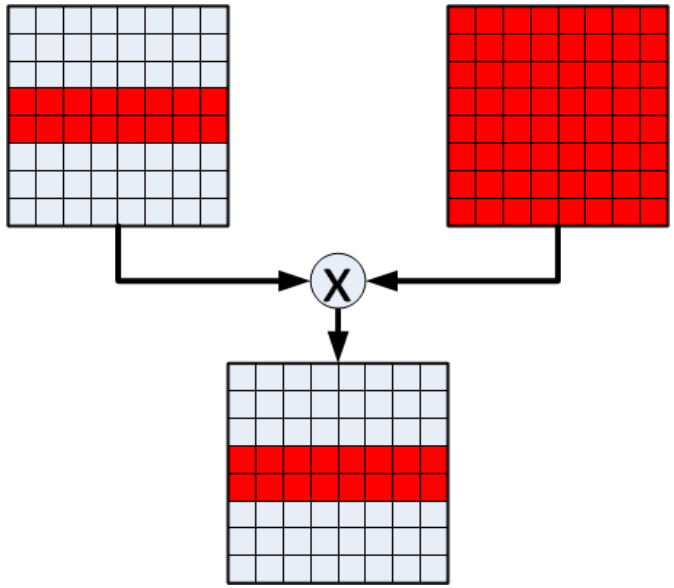
# Parallel MMULT – 1D row partitioning

**Older experiment!**

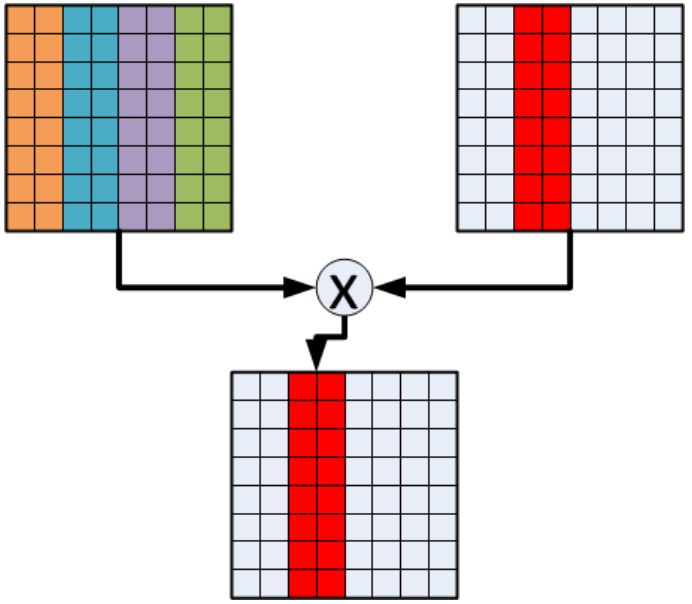


# Parallel MMULT – 1D row partitioning

- Analysis
  - + No communication among workers
  - + Nice scalability
  - - Large memory footprint will sooner or later limit scalability
  - - Large amounts of memory seldom used, but allocated for complete execution time
  - - Analogy to caches
- More sophisticated partitioning methods
  - More messaging overhead
  - Basic idea is based on the associativity of the addition
  - Operate on blocks (or tiles) of the matrices that you own, then communicate to get new blocks



# Parallel MMULT – 1D column partitioning




- Idea:
  1. First, let i compute all it owns:
     - C[,i] = C[,i] + A[,i] * B[i,]
       i = my_rank
  2. Then, process i sends A[,i] to process (i+1) % p
     - p is number of processes
  3. Repeat for p-1 times

```
C[,m] = C[,m] + A[,m]*B[m,]
S = A[,m]

For i = 1 to p-1
   send S to process (m+1)%p
   recv S from process (m-1)%p
   // S is now A[,(m-i)%p]

   C[,m] = C[,m] + S*B[(m-i)%p,]
```

# Parallel MMULT – 1D column partitioning

- Computation costs per step
  - p processes *elements * madd * (vector elements / blocks)*
  - Compution costs per step: $f = N^2 * 2 * (N / p) = 2N^3/p$
- Timing model for communication
  - Communication follows a linear model: $t(s) = t_{lat} + s * t_{BW}$
  - **s**: size (number of elements), $\mathbf{t_{lat}}$:constant overhead in time, $\mathbf{t_{BW}}$: time to transfer one element
- Communication costs per step
  - $s = N^2/p$ (size of A divided by number of processes)
- Total costs:
  - $c = f + ( p-1 ) * ( f+t(s) )$
    - Initial step compute, then (p-1) times repeating
  - $c = f * p + ( p-1 ) * t_{lat} + ( p-1 ) * N^2/p * t_{BW}$

# Parallel MMULT – 1D column partitioning

- Total costs $c_{parallel} = 2N^3/p + (p-1)*t_{lat} + (1-1/p)*N^2 * t_{BW}$
  - $t_{new} = c_{parallel}$; $t_{old} = f = 2N^3$
- Comments:
  - Term 1: Perfect scalability (O(1/p)) if no communication costs, resp. no communication
  - Term 2+3: Overhead due to parallelization
- Scaling the problem size
  - $c_{parallel} = O(N^3/p)$
  - N → oo: SU → ???

# Parallel MMULT – 1D column partitioning

- Total costs $c_{parallel} = 2N^3/p + (p-1)*t_{lat} + (1-1/p)*N^2 * t_{BW}$

  $2N^3/p$: Computation; $(p-1)*t_{lat} + (1-1/p)*N^2 * t_{BW}$: Communication

- Scalability
  - Computation $O(N^3)$
  - Communication $O(N^2)$

- As long as communication costs grow slower than computation costs, system is considered scalable
  - One of the strictest definitions
  - Rarely used because difficult to verify

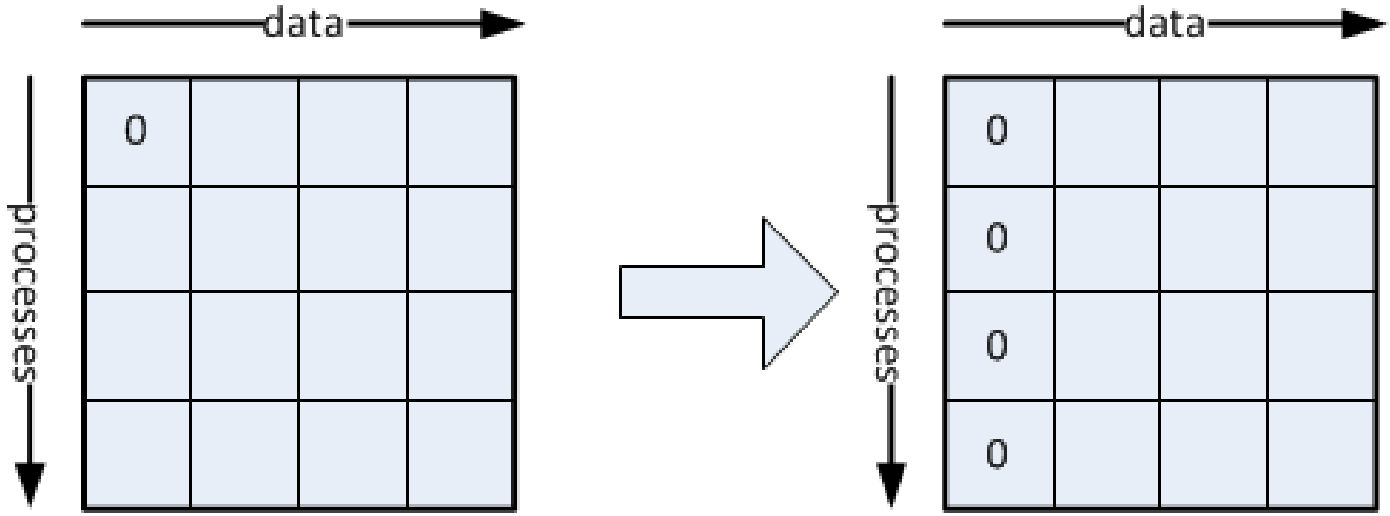
# Parallel MMULT – 1D column partitioning

- Parallel MMULT – 1D column partitioning
  - Initialization: distribute A[,i] to process i, i=1..P, broadcast B
    - MPI_SCATTER
    - MPI_BCAST
  - During each step: sending and receiving of blocks
    - MPI_SEND & MPI_RECV
    - MPI_SENDRECV
  - Final result: gather C[,i] from process i, i=1..P
    - MPI_GATHER
- Collective operations can help to simplify a program, like library calls
  - Every involved process has to make the function call

# Parallel MMULT – 1D column partitioning

## MPI_BCAST

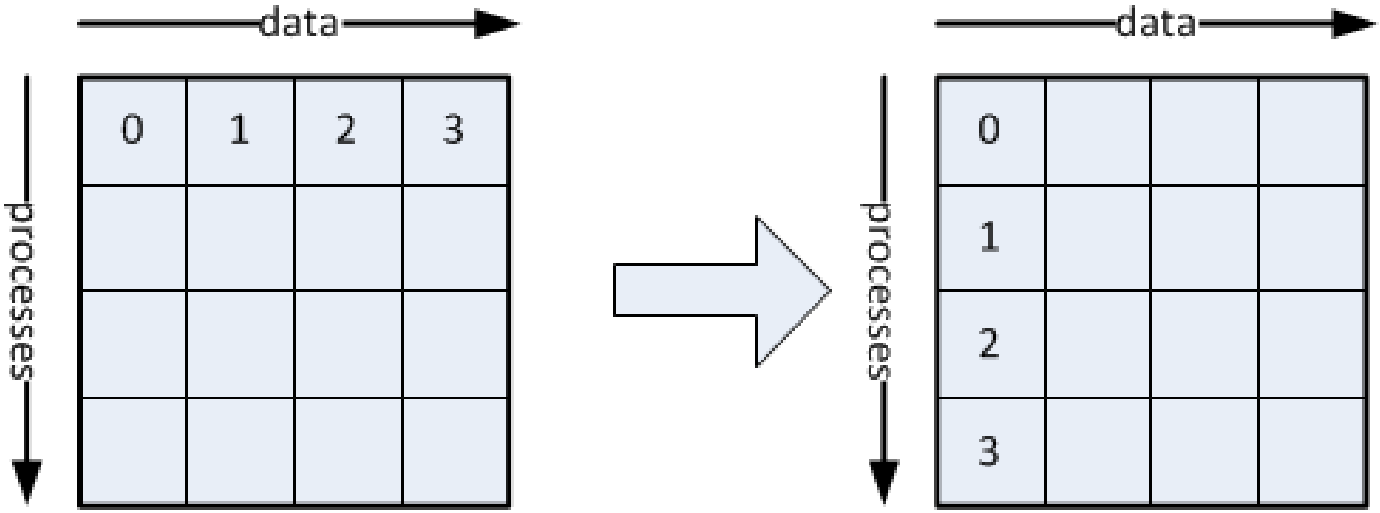
- MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
- As if process *root* executes n send operations to n destinations



# Parallel MMULT – 1D column partitioning

## MPI_SCATTER

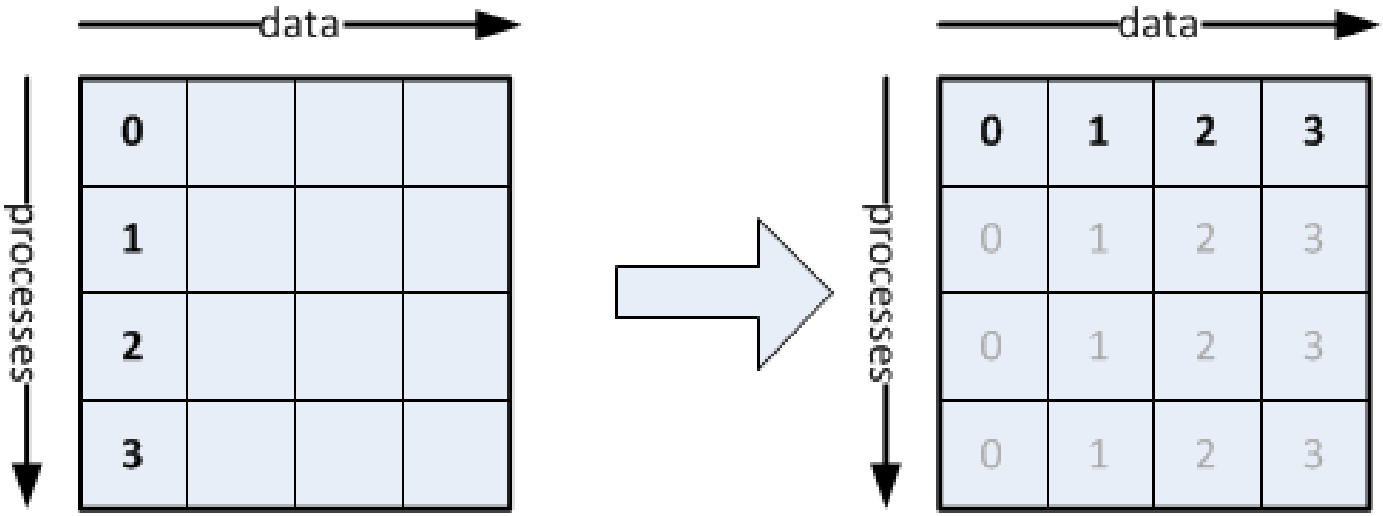
- MPI_Scatter ( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm )
- As if process *root* executes n send operations to n destinations
- See also MPI_Scatterv - various



# Parallel MMULT – 1D column partitioning

## MPI_GATHER

- MPI_Gather ( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- As if process *root* makes n receives from n sources
- See also MPI_Gatherv – various, MPI_Allgather (shown in gray)



# Notes on collective operations

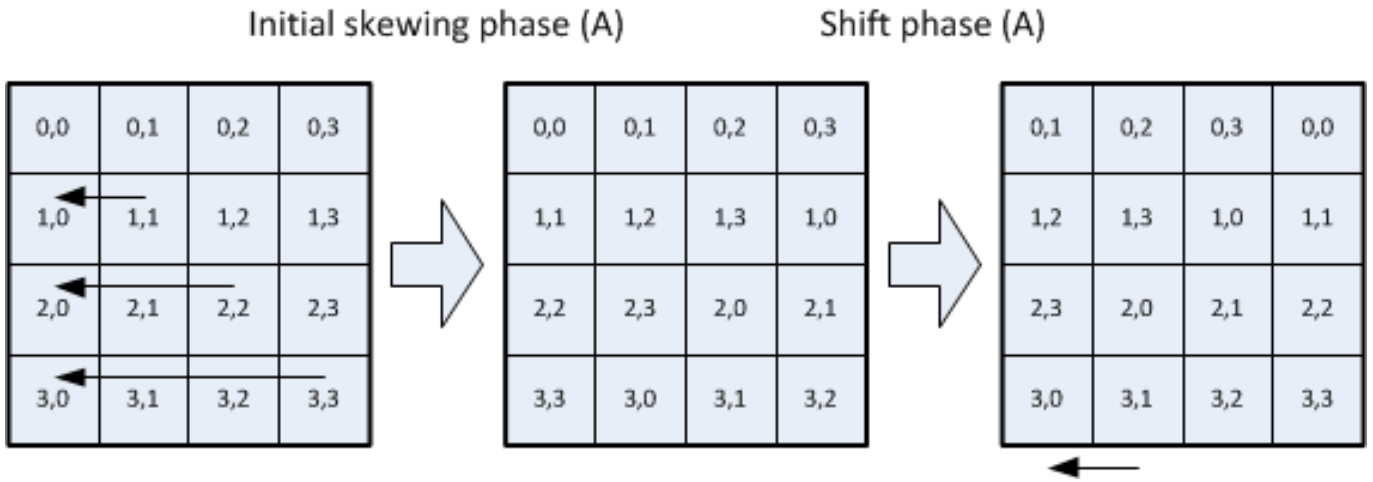
- Plenty of them, to cover all the various patterns
- Like libraries, others are responsible for optimization
  - E.g., for different topologies
- Communication cost: between O(n) and O(n²)
  - n=process count
- Dedicated hardware support for multicasts
  - Only found in advanced communication devices
    - BlueGene, Cray, EXTOLL
  - Typically not covered in specifications like Ethernet or Infiniband
- Multicast != Ethernet broadcast
  - Ethernet broadcasts are not reliable
- Currently collectives are always blocking (preventing overlap)
  - See MPI 3.0 for more advanced solutions

# Parallel MMULT – 2D partitioning

- 1D partitioning
  - Maps nicely to ring, also to mesh/tori/hypercube but does not utilize all network resources
  - Still huge memory requirements ($N^2/P$)
  - $P \sim N^2$ ❾ scaling P with N is difficult
  - P ~ N would be better
- Assume $P=s^2$, s whole number
- Then each process computes a block of C:
  - For all i,j of block(C); for k = 0..(s-1)
    - C[i,j] = C[i,j] + sum ( A [ i, k ] * B [ k, j ] )
  - Rotate blocks of A & B circularly
- Owner computes, optimize locality

# Parallel MMULT – 2D partitioning

- Three phases:
  1. Initial skewing phase: left shift circular row i of A by i times to the left, up shift circular column j of B by j times
     - For all i: A[i,j] = A[(i+1)%s,j], B accordingly
  2. Computation
  3. Shift: {left,up} shift circular of each {row,column} of {A,B} by one
- In-place algorithm
- Communication costs now by factor sqrt(p) smaller



# Parallel MMULT – 2D partitioning

- Broadcast-Multiply-Roll (BMR)
  - Using a broadcast instead of the initial skewing process
  - Three phases: broadcast, multiply, roll
  - Algorithm is not in-place
- SUMMA: Scalable Universal Matrix Multiplication Algorithm
  - Highly general algorithm
  - B small: less memory requirement, lower efficiency; B large: high memory requirement, higher efficiency
- Many more
  - Recursive layouts

# Summary

- PCAM used for the MMULT example
  - P: easy as domain decomposition is applied
  - C: trivial (copies), difficult (1D/2D partitioning)
  - A: trivial (copies), difficult (1D/2D partitioning)
  - M: easy
- Data handling (initial distribution & later movement) is most difficult when writing MPI programs
- Sequential versions can help to verify correctness
- Collective MPI calls can make the program easier, both for the initial coder and later readers
  - Topology agnostic