Cache Performance Optimizations for Iterative Linear Solvers and the Lattice Boltzmann Method

Markus Kowarschik

Lehrstuhl für Systemsimulation

Institut für Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg

Outline

1. Introduction: hierarchical memory architectures

2. Target algorithms

   - Iterative linear solvers
   - Cellular automata (e.g., the lattice Boltzmann method)

3. Locality optimizations (which may be automized)

   - Data layout optimizations
   - Data access optimizations
   - Performance results

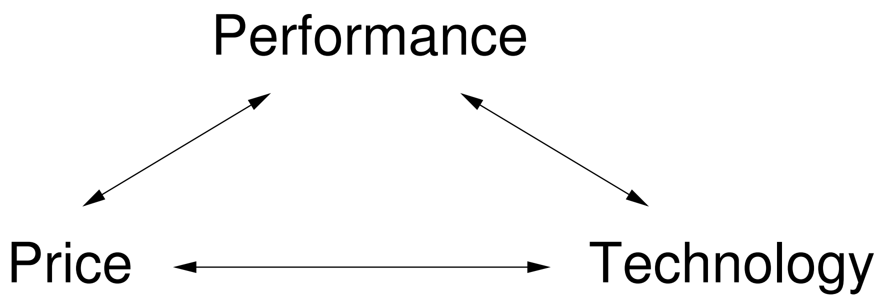
4. (Inherently cache-aware multigrid methods)

5. Related topics and conclusions

Introduction: hierarchical memory architectures

Goal: Mitigate the effects of the constantly widening gap:
CPU speed $\Longleftrightarrow$ DRAM main memory performance

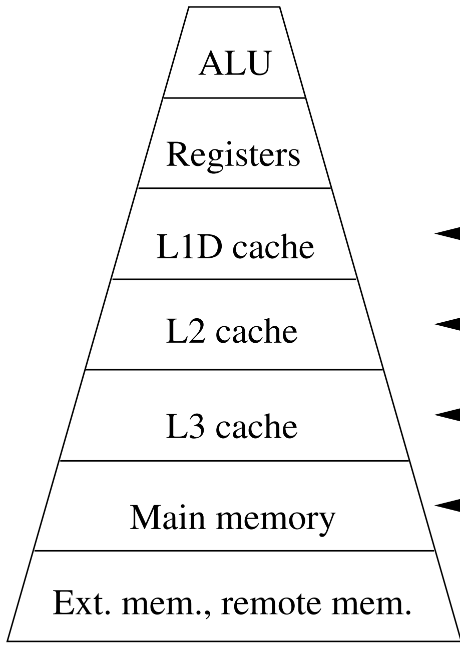
Trade-off:



Approach: Hierarchical memory designs involving several layers of cache memories

Introduction: hierarchical memory architectures

Example: Intel Itanium 2 machine (1 GHz):



Capacity	Bandwidth	Latency
16 KB	32 GB/s	1 cycle
256 KB	32 GB/s	5+ cycles
1.5/3/6 MB	32 GB/s	12+ cycles
2 GB	6.4 GB/s	>100 cycles

Estimate: 1 main memory access may take longer than 400 FP operations!

Introduction: hierarchical memory architectures

Goal: Exploit the memory hierarchy as efficiently as possible!

Spectrum of *data locality optimizations* to enhance cache utilization:

- Hardware techniques; e.g., data prefetching
- Compiler-based techniques; e.g., data prefetching, loop transformations

$\Updownarrow$

- Programming techniques; e.g.,
  - Data layout optimizations; e.g., array merging, array padding
  - Data access optimizations; e.g., loop transformations
- Inherently cache-aware (numerical) algorithms

Instruction cache performance is (typically) a minor issue in scientific computing

Introduction: hierarchical memory architectures

High performance computing requires the combination of

1. Efficient parallelization; e.g.,
   - Load balancing
   - Reduction of communication overhead

2. Optimal utilization of the individual resources,
   particularly by respecting the memory hierarchy

Target algorithms I — iterative linear solvers

We want to solve systems $Ax = b$ of linear equations

$A \in \mathbf{R}^{n \times n}$ is:

- highly structured; e.g., based on the discretization of PDEs using regular grids
- large, $n > 10^6$
- sparse; i.e., $\mathcal{O}(n)$ nonzeros

We consider *iterative numerical algorithms*, particularly:

- Elementary schemes: Gauss-Seidel, Jacobi, SOR, weighted Jacobi
- Multigrid methods
  - Asymptotically optimal complexity, $\mathcal{O}(n)$ floating-point operations
  - Goal: generation of "optimal implementations of optimal algorithms"

Target algorithms II — CA/LBM

Prominent example of a cellular automaton (CA): Conway's *game of life*

Lattice Boltzmann method (LBM): CA models for simulating fluid flows:

- Simulations in 2D/3D
- It is straightforward to handle complex time-dependent geometries
- Applications in
  - CFD; e.g, simulation of turbulent flows
  - Material science; e.g., simulation of metal foams, *FreeWiHR* project
  - Chemical engineering; e.g,, particle technology
- High computational requirements for realistic simulations
- Inherently parallel

The LBM

Starting point: Boltzmann equation (B.E.): $\frac{\partial f}{\partial t} + \langle u, \nabla f \rangle = Q$

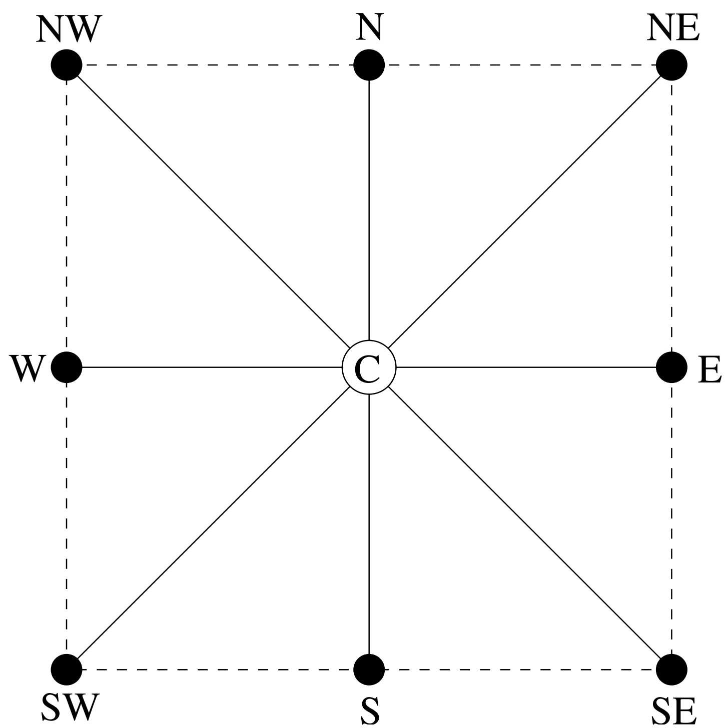
Continuous model of moving fluid particles:

- $f = f(x, u, t)$: particle distribution function
- x: position in physical space
- u: velocity
- t: time
- Q: collision operator

LBM principle: Discretization of the B.E. w.r.t. space, time, and velocity

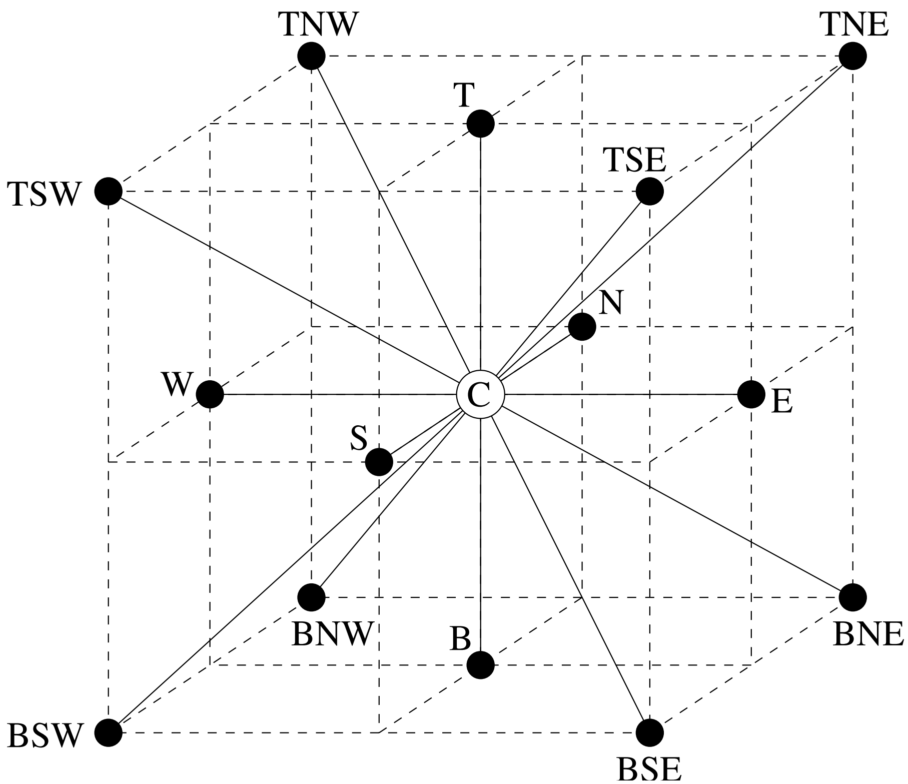
The LBM in 2D

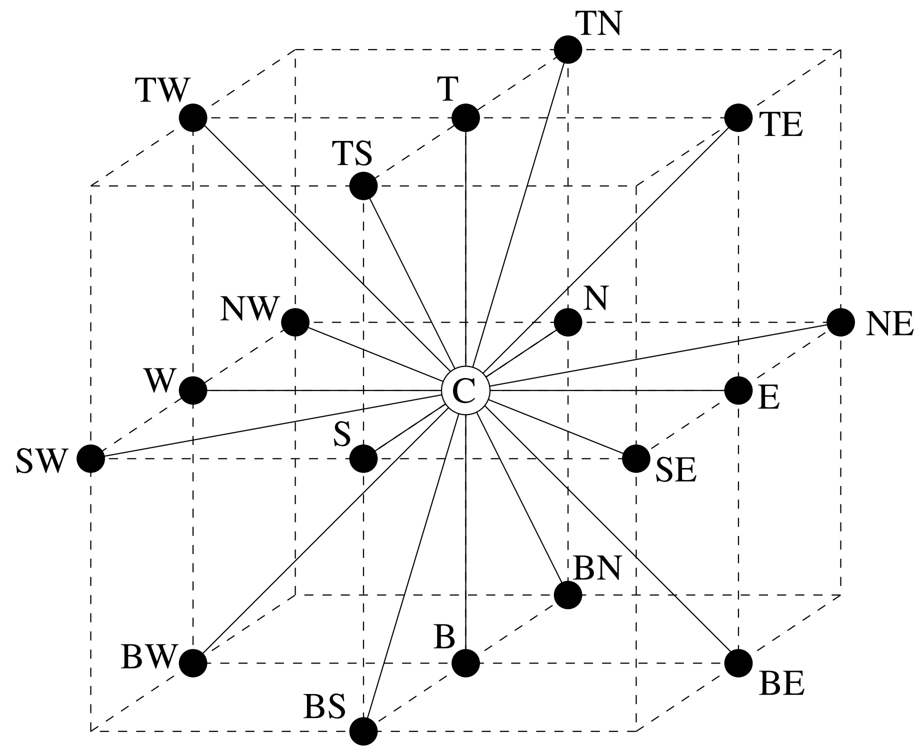
Example: A single cell of the D2Q9 LBM model:



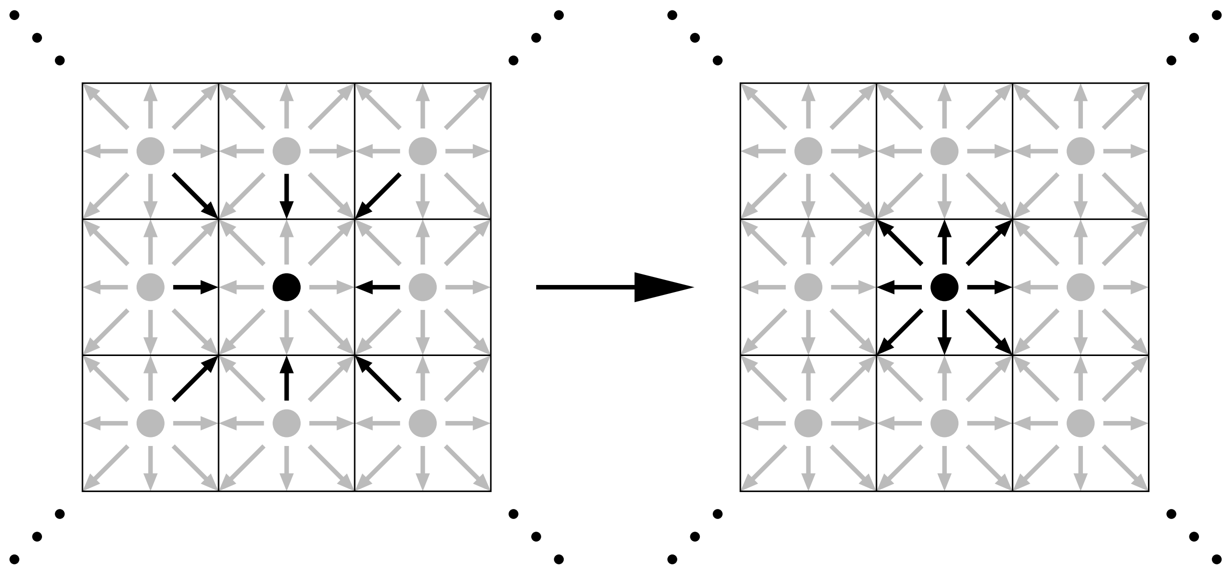
The LBM in 3D

Example: A single cell of the D3Q15 and the D3Q19 LBM model, respectively:





The LBM — how the algorithm works



- Successive passes through the data set, Jacobi-like update operation of grid cells in each time step

- "Stream and collide" or "collide and stream" update order

- Code efficiency measured in "million lattice site updates / second" (MLUPS)

The LBM — Example

Example: Simulation of free surfaces

- Nils Thürey's Diplomarbeit: `http://www.ntoken.com/fluid`
- Contribution to the FreeWiHR project: simulation of metal foams
- Show movies . . .

Locality optimizations

- *Data layout optimizations:*
  Address data storage schemes in memory;
  i.e., the way how the data are arranged in address space

- *Data access optimizations:*
  Address the order in which the data are accessed

Data layout optimizations — cache-aware data structures

Idea: Merge data which are needed together to increase *spatial locality:* cache lines contain several data items

Example: Gauss-Seidel on $Ax = b$, 2D, 5-point stencils:

$$x_i^{(k+1)} = a_{i,i}^{-1} \left(b_i - \sum_{j<i} a_{i,j} x_j^{(k+1)} - \sum_{j>i} a_{i,j} x_j^{(k)} \right), \quad i = 1, \ldots, N$$

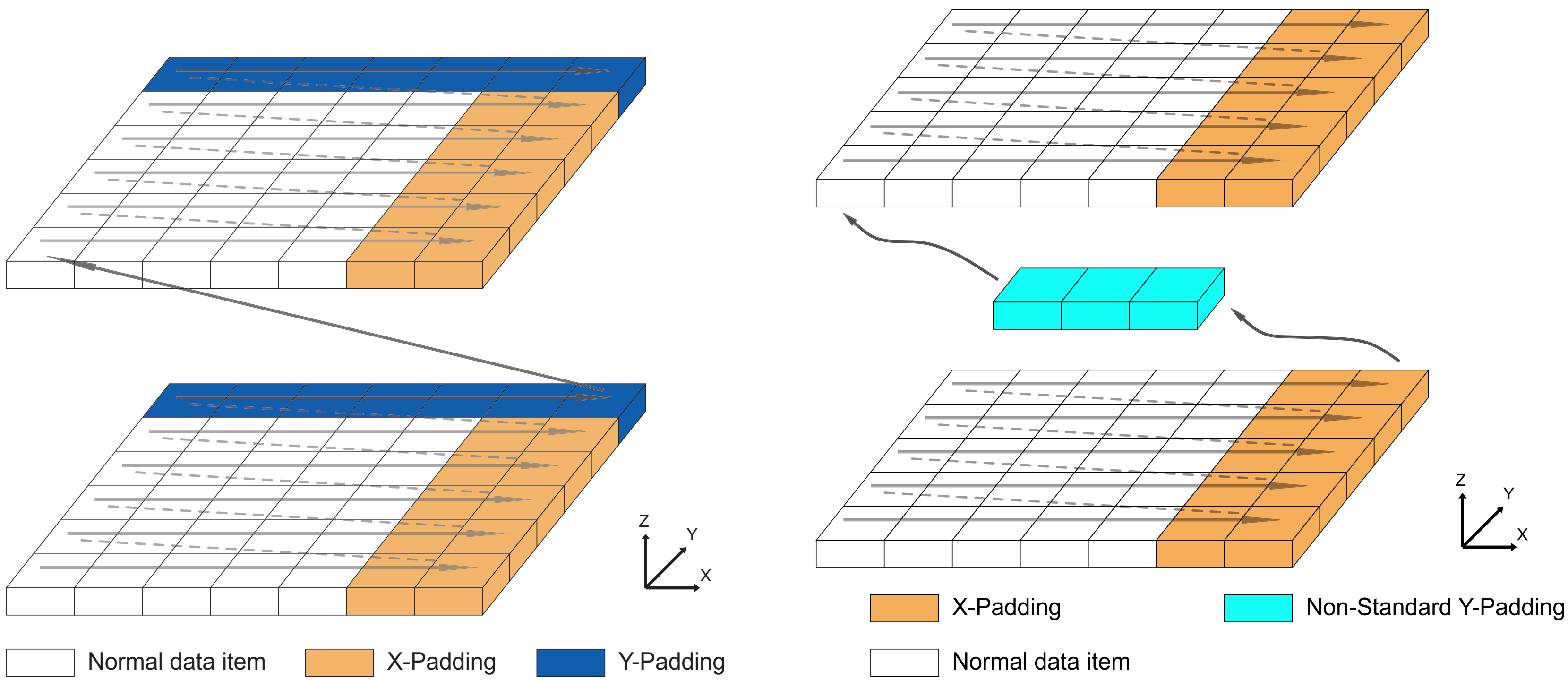
```
typedef struct {
  double b;
  double cCenter, cNorth, cEast, cSouth, cWest;
} eqnData;

double   x[N][N];             // Solution vector
eqnData rhsAndCoeff[N][N]; // Right-hand side and coefficients
```

Data layout optimizations — array padding

Idea: Increase array dimensions to change relative distances between elements
$\Longrightarrow$ Eliminate cache conflict misses; e.g., in stencil computations

Example: 3D arrays



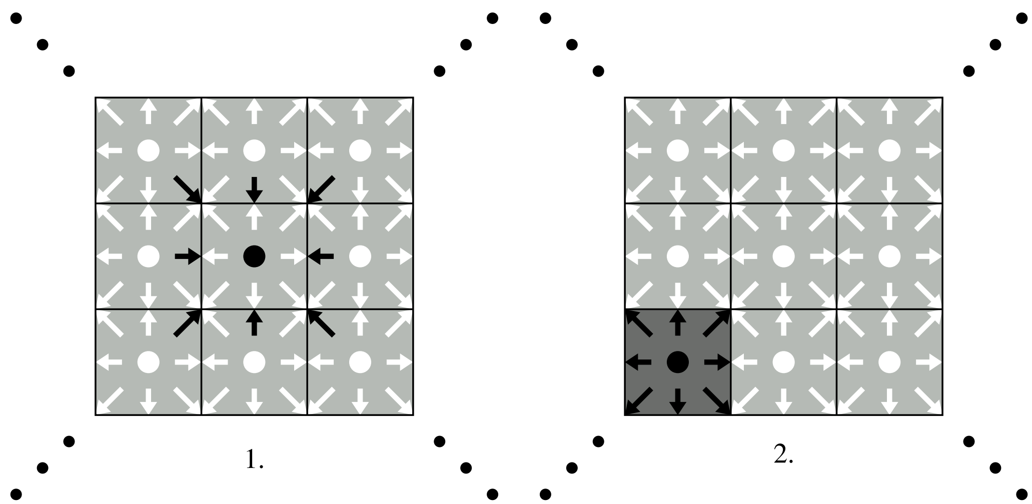
Standard padding in FORTRAN77:

```
double precision u(xdim + xpad, ydim + ypad, zdim)
```

Data layout optimizations — grid compression

Observation: It is not necessary to store two full grids (source + destination)

Idea: Overlay the source and the destination grid and introduce diagonal shifts:

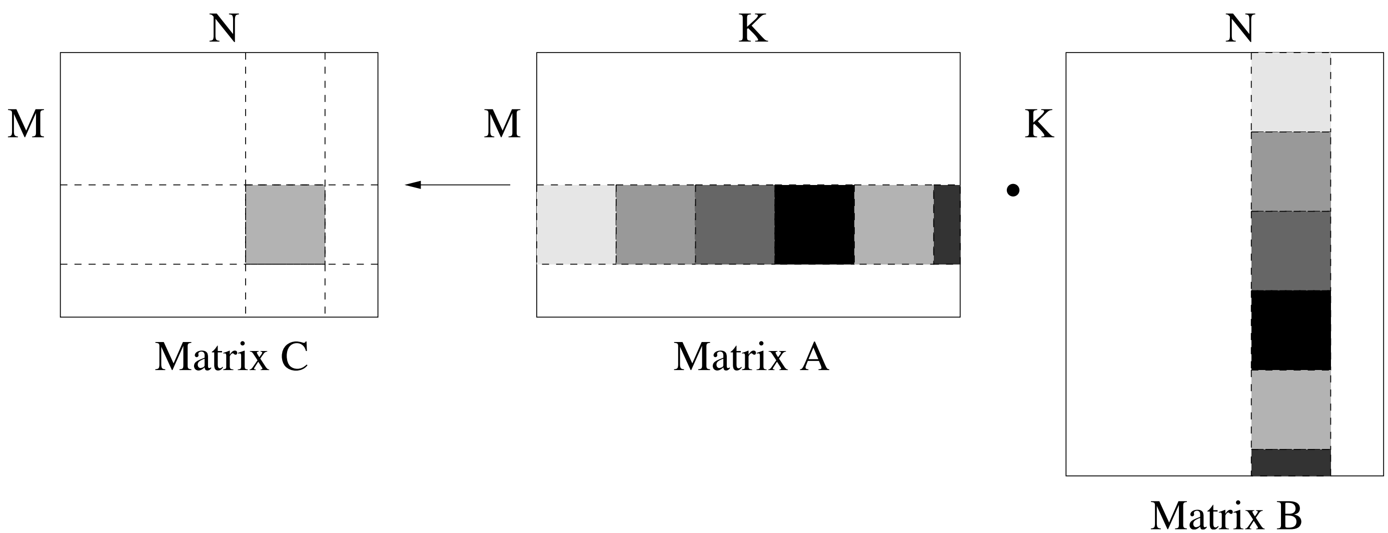


- GC reduces the memory requirements of the LBM by almost a factor of $\frac{1}{2}$
- GC implicitly enhances spatial locality

Data access optimizations — loop blocking

Principle: Divide the iteration space into blocks and perform as much work as possible on the data in cache (i.e., on the current *block*) before switching to the next block $\Longrightarrow$ Enhance spatial/temporal locality **while respecting data dependencies**

Popular textbook example: Matrix multiplication: $C = A \cdot B$:



$A, B, C \in \mathbf{R}^{n \times n}$: $\mathcal{O}(n^2)$ data accesses + $\mathcal{O}(n^3)$ floating-point operations
$\Longrightarrow$ High potential for data reuse

Data access optimizations — loop blocking

Blocked implementations of iterative linear solvers:

- Iteration loop: merge consecutive iterations into a single pass through the data set:

$$x^{(k+1)} = Mx^{(k)} + d, \quad x^{(k+2)} = M(Mx^{(k)} + d) + d, \quad \dots$$

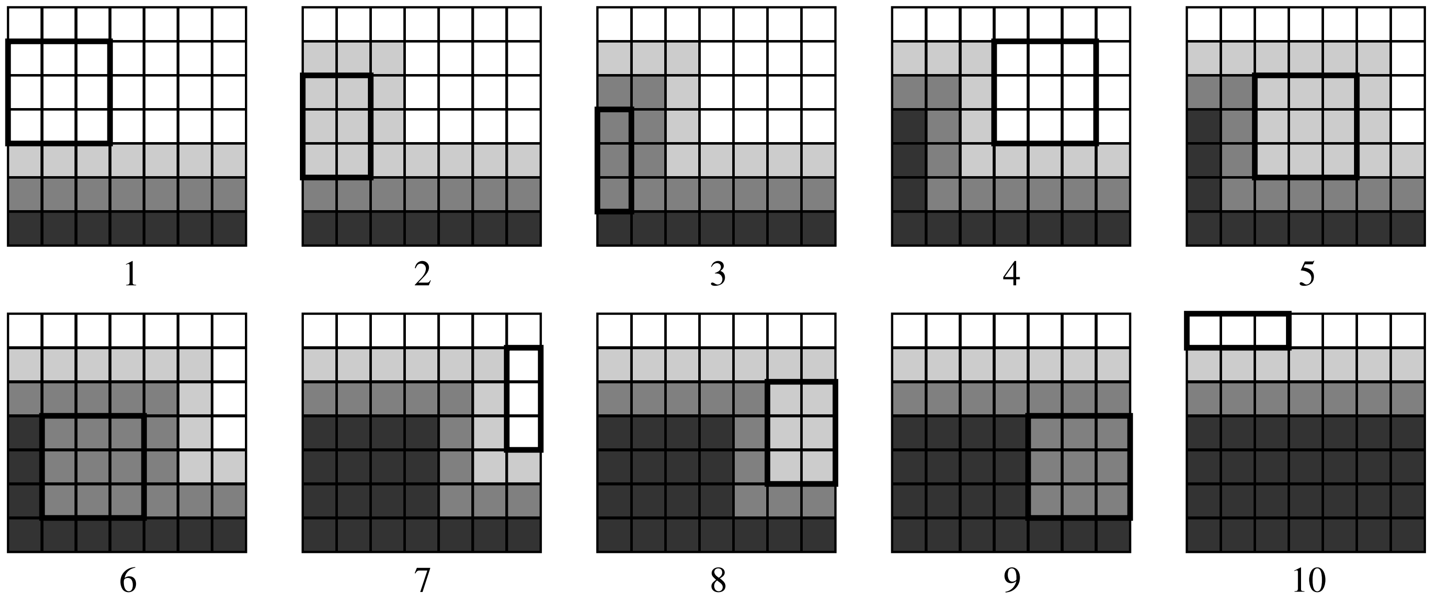
- Loops along spatial dimensions can additionally be blocked

Blocked LBM implementations:

- Time loop: merge consecutive time steps into a single pass through the data set
- Loops along spatial dimensions can additionally be blocked

Data access optimizations — loop blocking

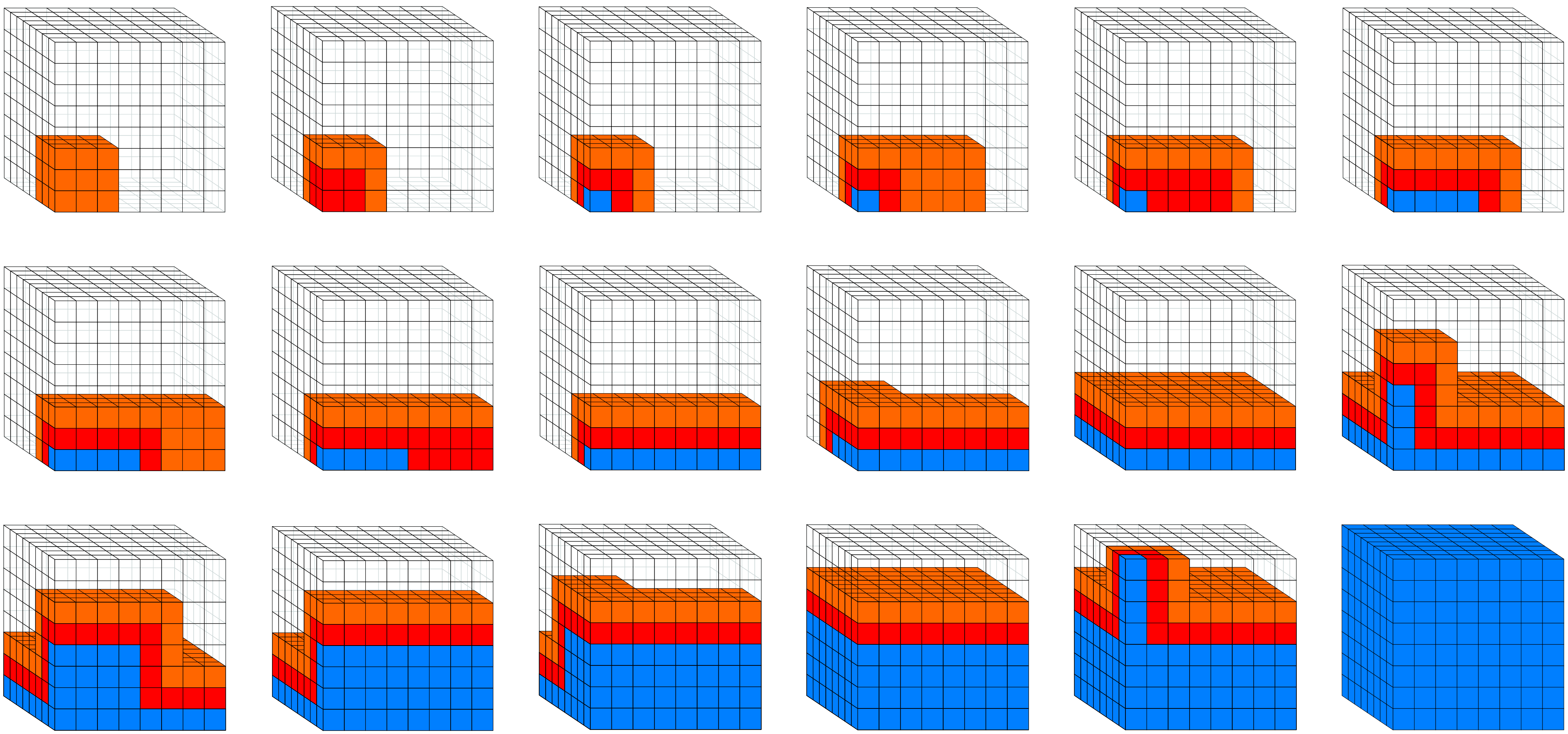
Example: LBM in 2D, 3-way blocking:



- Here: three consecutive time steps are executed during a single pass through the grid ($t_B = 3$)
- Ideally: MLUPS rate independent of the grid size if the size of the "moving 2D block" is chosen appropriately

Data access optimizations — loop blocking

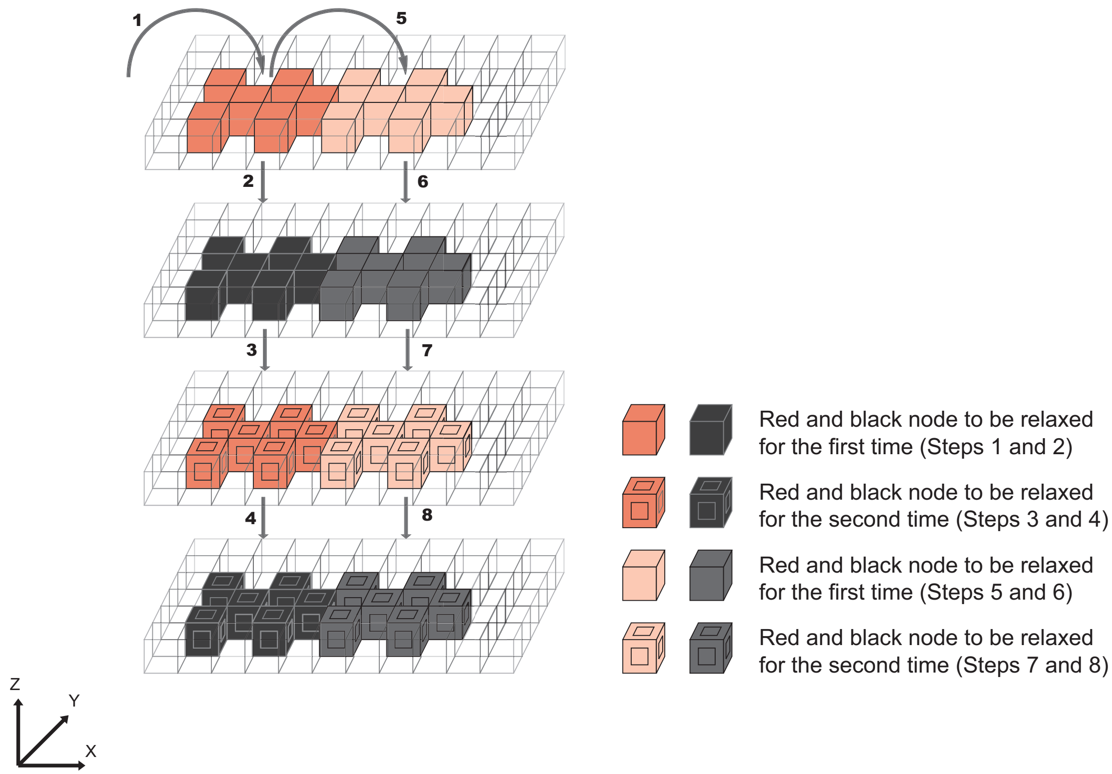
Example: LBM in 3D, 4-way blocking, $t_B = 3$:



Ideally: MLUPS rate independent of the grid size if the size of the "moving 3D block" is chosen appropriately

Data access optimizations — loop blocking

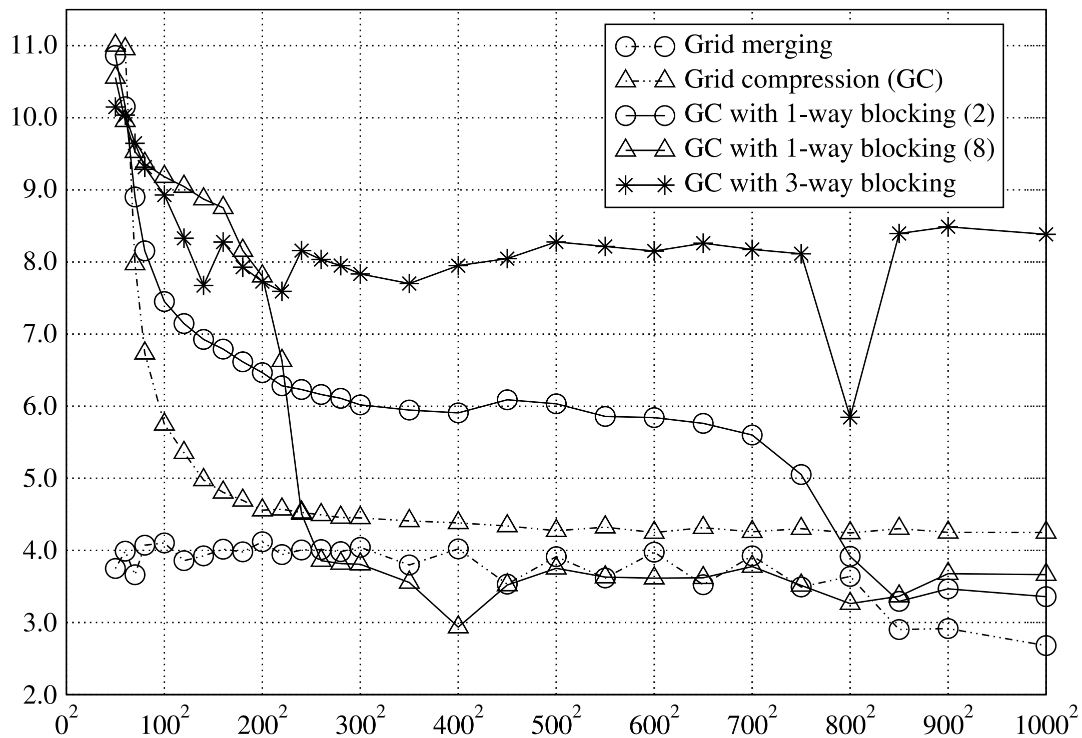
Example: 3D Red/black Gauss-Seidel (e.g., as a multigrid smoother), 3-way blocking:



Iteration loop and the loops along directions x and the y have been blocked

Performance results — LBM in 2D

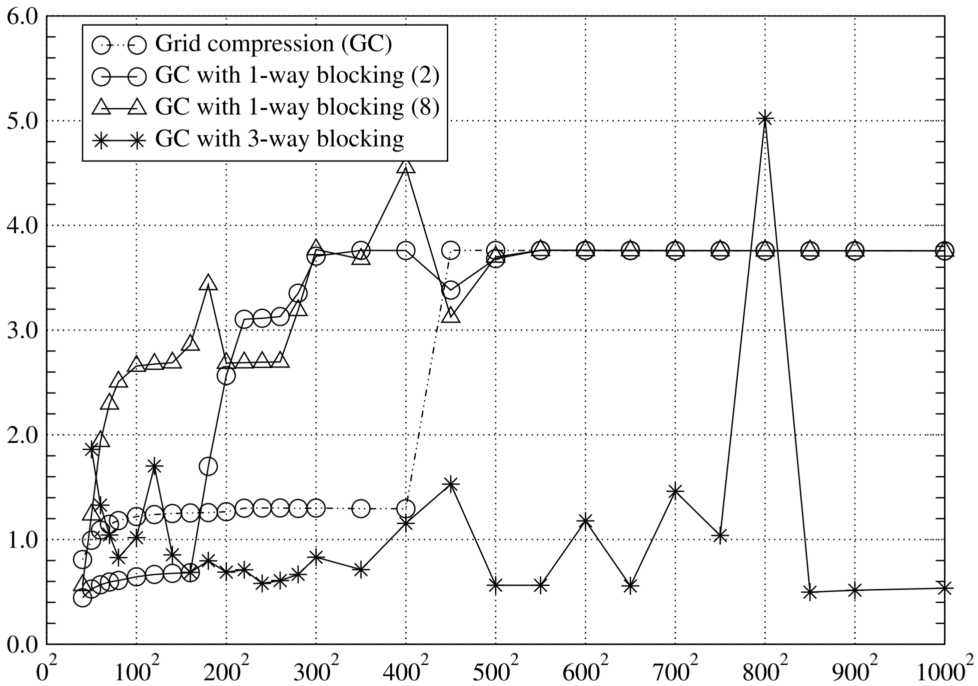
AMD Athlon XP 2400+ CPU, Linux, GNU gcc 3.2.1, flags: `-O3 -ffast-math`:

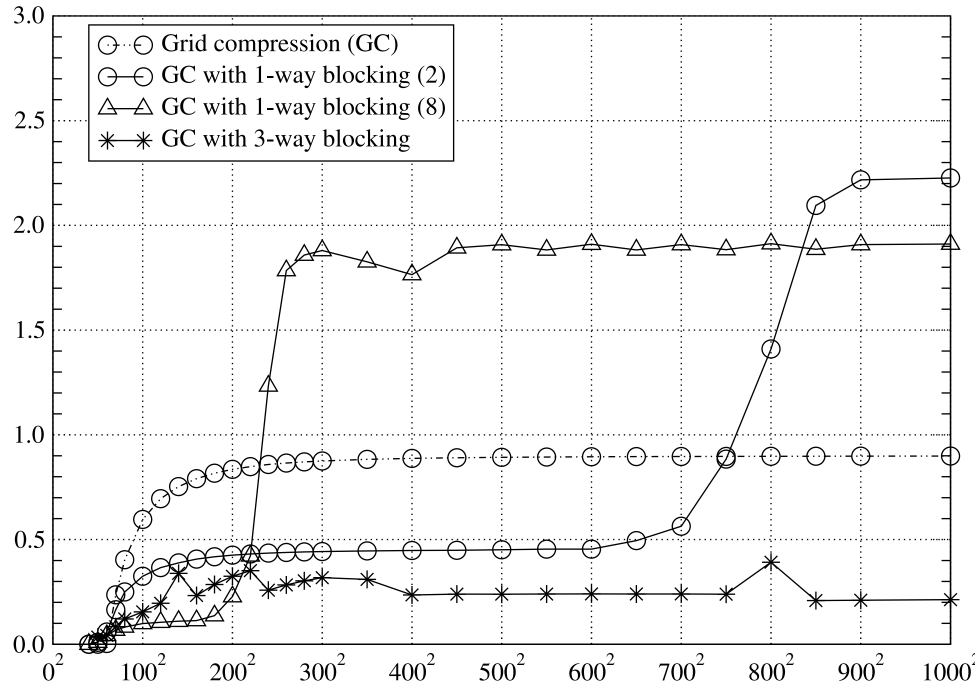


- 1-way blocked codes perform well as long as working set fits into cache
- MLUPS rate of the 3-way blocked code (almost) independent of the grid size
- 8.4 MLUPS ≈ 974 MFLOPS

Performance results — LBM in 2D

Profiling results using PAPI:

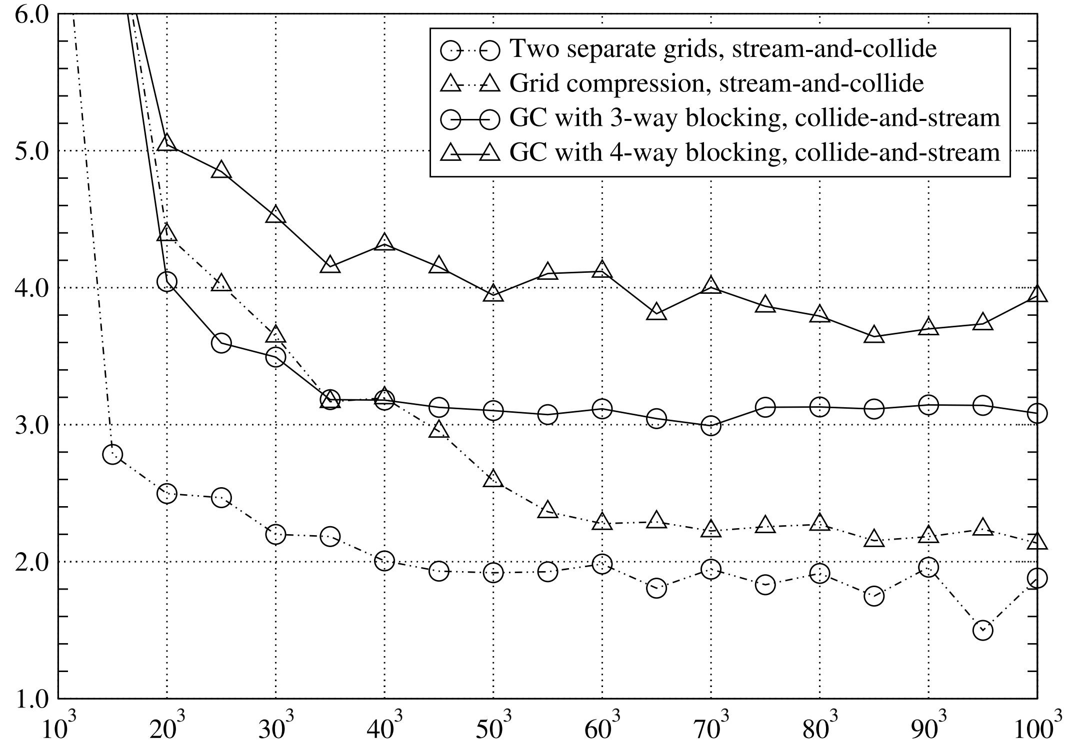




- Left: L1 (64 kB) misses per cell update, right: L2 (256 kB) misses per cell update
- L1 cache thrashing effect for grid size 800^2, single grid row ≈ 64 kB

Performance results — LBM in 3D

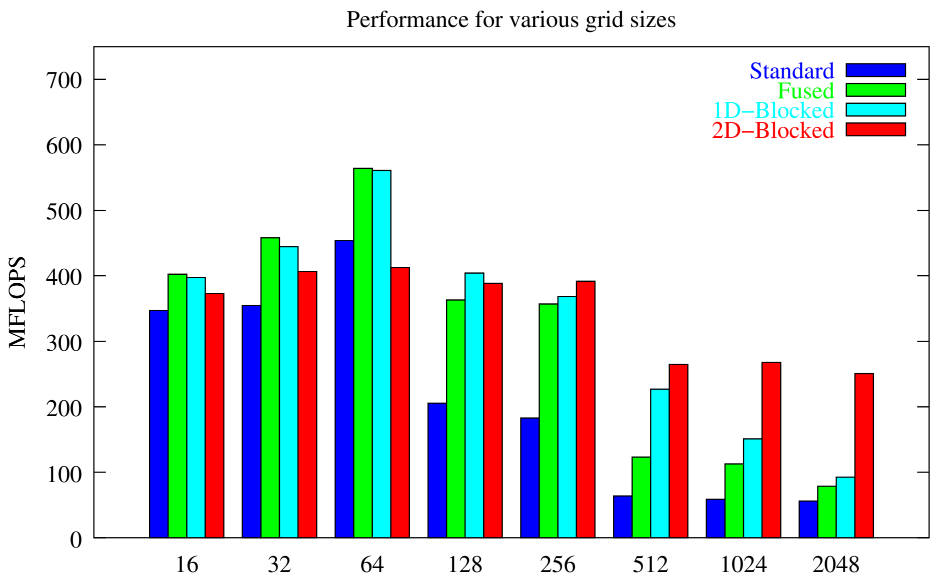
AMD Opteron CPU, 1.6 GHz, Linux, GNU gcc 3.2.2, flags: -O3:

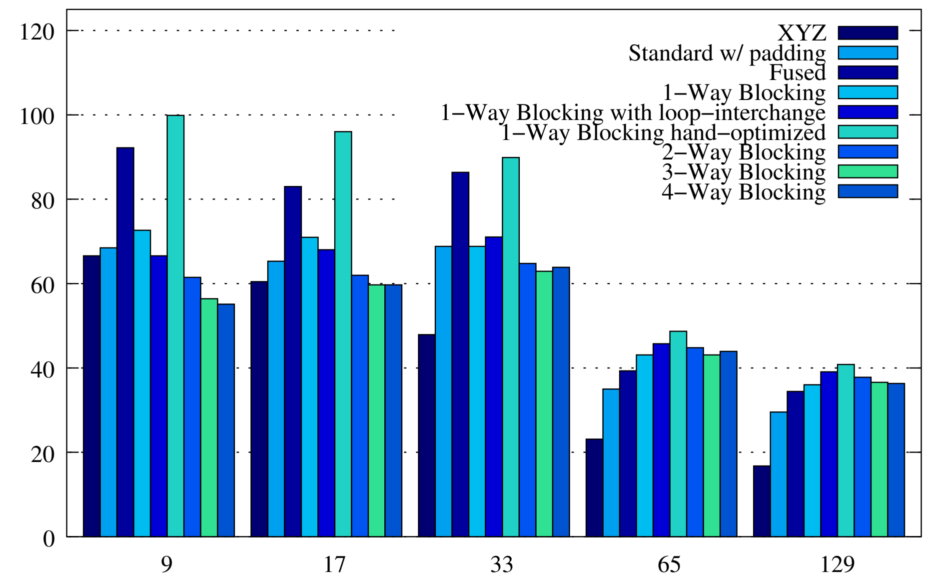


- 3D case exhibits lower speedups than 2D case
- 4 MLUPS ≈ 1020 MFLOPS
- AMD Athlon XP: fastest 3D LBM code runs at 2.2 MLUPS ≈ 560 MFLOPS

Performance results — multigrid

Alpha A21164 machine, 500 MHz, Tru64 UNIX, F77, aggressive opt. enabled:





- Left: Gauss–Seidel, 2D, constant 5-point stencils (C. Weiß)
  right: V(2,2) MG cycles, 3D, variable 7-point stencils
- Generally: MFLOPS rates for 3D MG codes must be considered disappointing (when compared with the theoretically available peak performance)

Performance results — LBM vs. multigrid

Comparison: LBM vs. multigrid (MG):

- Typically: LBM is computationally more intensive (less memory-bound) than MG, example:
  - 3D Gauss-Seidel, variable 7-point stencils: 0.9 FP ops./data access
  - D3Q19 LBM model: 6.7 FP ops./data access
- Consequences:
  - MFLOPS rates for MG codes much lower than for LBM codes
  - Comparable speedups (?)

(Inherently cache-aware multigrid methods)

So far: optimization techniques have maintained numerical properties

One step further: novel multigrid algorithms

Starting point: *fully adaptive multigrid method* (U. Rüde)

Essential component: *adaptive relaxation* on $Ax = b$, $A = (a_{i,j})$ s.p.d.

Definition *(scaled residual)*: $\theta_i(x) := a_{i,i}^{-1} e_i^T (b - Ax)$

Motivation:
Error reduction for one elementary relaxation step $x \leftarrow x + \theta_i(x) e_i$ is given by

$$||x^{\text{old}} - x^*||_E^2 - ||x^{\text{new}} - x^*||_E^2 = a_{i,i} \theta_i (x^{\text{old}})^2$$

(Inherently cache-aware multigrid methods)

ActiveSet: set of indices of nodes with “large” scaled residuals

Algorithm: adaptive relaxation

```
while ActiveSet ≠ 0 do
    pick i ∈ ActiveSet
    ActiveSet ← ActiveSet \ {i}
    if |θ_i(x)| > θ then
        x ← x + θ_i(x) e_i
        ActiveSet ← ActiveSet ∪ Conn(i)
    end if
end while
```

Fully adaptive multigrid:
Adaptive relaxation on the (well-conditioned) extended system which represents the entire grid hierarchy (U. Rüde, M. Griebel)

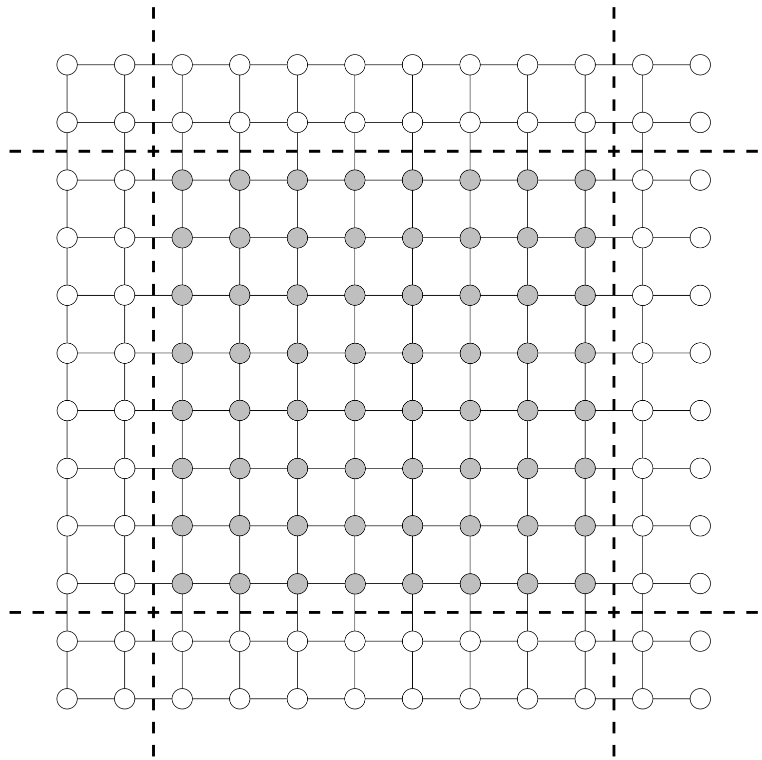
$\Longrightarrow$ Multigrid efficiency + flexible update strategy + enhanced robustness

Flexible update strategy can be exploited to enhance cache utilization

(Inherently cache-aware multigrid methods)

Introducing patch adaptivity:

Overhead to maintain the active set motivates *patch-based* instead of *node-based* processing strategies



Performance tuning and numerical experiments: work in progress ...

Things we have not addressed in this talk ...

- Extended complexity models covering
  1. Arithmetic work and
  2. Data locality/memory access behavior
- Code performance analysis, profiling techniques
- Alternative inherently data-local PDE solvers;
  e.g., based on domain decomposition approaches (D. Keyes, S. Turek, ...)
- Aspects of software engineering; e.g.,
  - Combining efficiency and portability
  - Self-tuning numerical codes (*ATLAS*, *SPIRAL*, ...)
- Automized introduction of cache-based source code transformations;
  e.g., the *ROSE* project (D. Quinlan)
- Cache optimizations for unstructured meshes (C.C. Douglas, ...)

Conclusions and final remarks

- Cache-aware programming can yield *remarkable speedups*
- We have targeted:
  - Iterative solvers for large sparse linear systems (particularly multigrid)
  - Cellular automata (particularly the LBM)
- Code optimizations may have conflicting goals; e.g.,
  - Blocking for cache may cause an increase in the TLB miss rate
  - Blocking for cache $\Longleftrightarrow$ dynamic branch prediction
  - Blocking for cache $\Longleftrightarrow$ pipelining, data prefetching

Thank you!

Any questions?

DiME project: data-local iterative methods

http://www10.informatik.uni-erlangen.de/dime