Locality–Optimized Iterative Methods

Ulrich Rüde, Markus Kowarschik
{*ulrich.ruede,markus.kowarschik*}*@cs.fau.de*

Thanks to Thomas Pohl, Nils Thürey, and Jens Wilke

Lehrstuhl für Systemsimulation
Institut für Informatik
Friedrich–Alexander–Universität Erlangen–Nürnberg
Germany

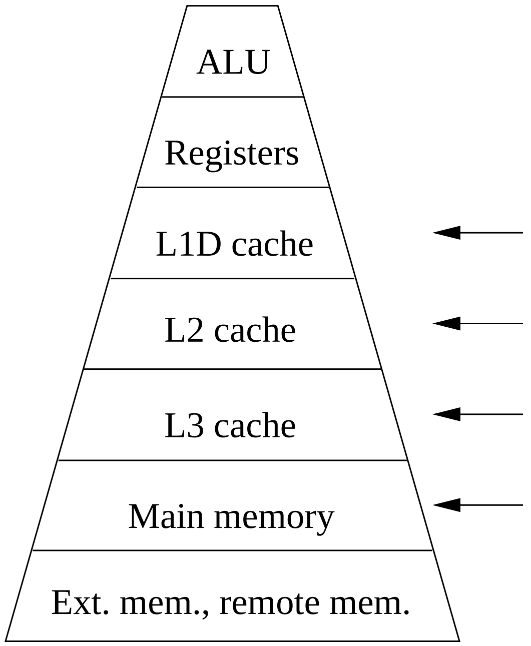
This project is being supported by the *Deutsche Forschungsgemeinschaft*:
DiME — data–local iterative methods

Outline

- A memory hierarchy example
- Techniques to enhance cache utilization (preserving numerical properties)
  - Data layout optimizations
  - Data access optimizations
- Performance results
  - Iterative methods for linear systems
  - Lattice Boltzmann methods
- Cache–aware "non–standard" algorithms
  - Adaptive relaxation
  - Fully adaptive multigrid
- Conclusions

Cache design — a memory hierarchy example

Memory architecture of an Intel Itanium2 (aka McKinley) machine (1 GHz):



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

CPU core can execute 2 fused multiply–add instructions per cycle
$\Rightarrow$ A single main memory access is more costly than the execution of 400 FP operations

Inevitable: exploit the cache architecture as efficiently as possible!

Techniques to enhance cache utilization

- *Data layout optimizations:*
  Address data storage schemes in memory
- *Data access optimizations:*
  Address the order in which the data are accessed

These are "compiler–oriented" techniques that do **not** change the properties of the numerical algorithms; e.g., convergence rate and robustness

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 $Au = f$, 2D, 5–point stencils:

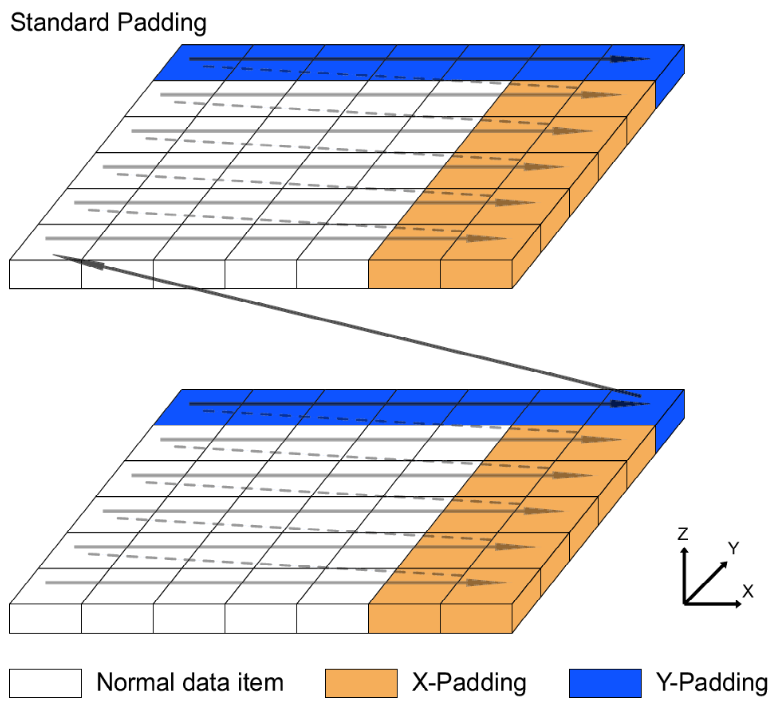
$$u_i^{(k+1)} = a_{i,i}^{-1} \left(f_i - \sum_{j<i} a_{i,j} u_j^{(k+1)} - \sum_{j>i} a_{i,j} u_j^{(k)} \right), \quad i = 1, \ldots, N$$

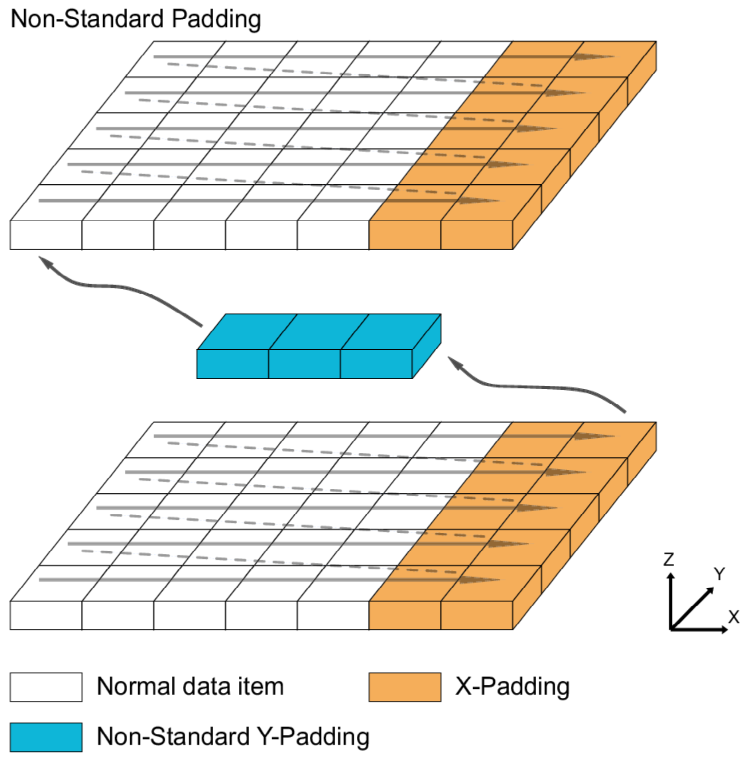
```
typedef struct {
  double f;
  double cCenter, cNorth, cEast, cSouth, cWest;
} eqnData;

double  u[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
⇒ Avoid severe 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 — array padding (cont'd)

Padding approaches:

- Analytic/Algebraic techniques (G. Rivera/C.–W. Tseng)
  - Block size (tile size) and paddings depend on array size and cache capacity
  - Often not general enough for realistic problems where several arrays are involved; e.g., CFD: pressure, velocity field, temperature, concentrations of chemical species, etc.
- Exhaustive parameter search
  - *AEOS* paradigm: *Automated Empirical Optimization of Software*; e.g.,
    - ∗ *ATLAS (Automatically Tuned Linear Algebra Software)*
    - ∗ *FFTW (The Fastest Fourier Transform in the West)*
  - Searching the parameter space is time–consuming, but currently the most promising cache tuning approach!

Data access optimizations — loop blocking (loop tiling)

Idea: 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
⇒ Enhance spatial and/or temporal locality **while respecting data dependencies**

Most popular textbook example: Matrix multiplication

Before loop blocking:

```
do J= 1,N
 do K= 1,N
  do I= 1,N
   C(I,J)= C(I,J)+A(I,K)*B(K,J)
  enddo
 enddo
enddo
```

After loop blocking:

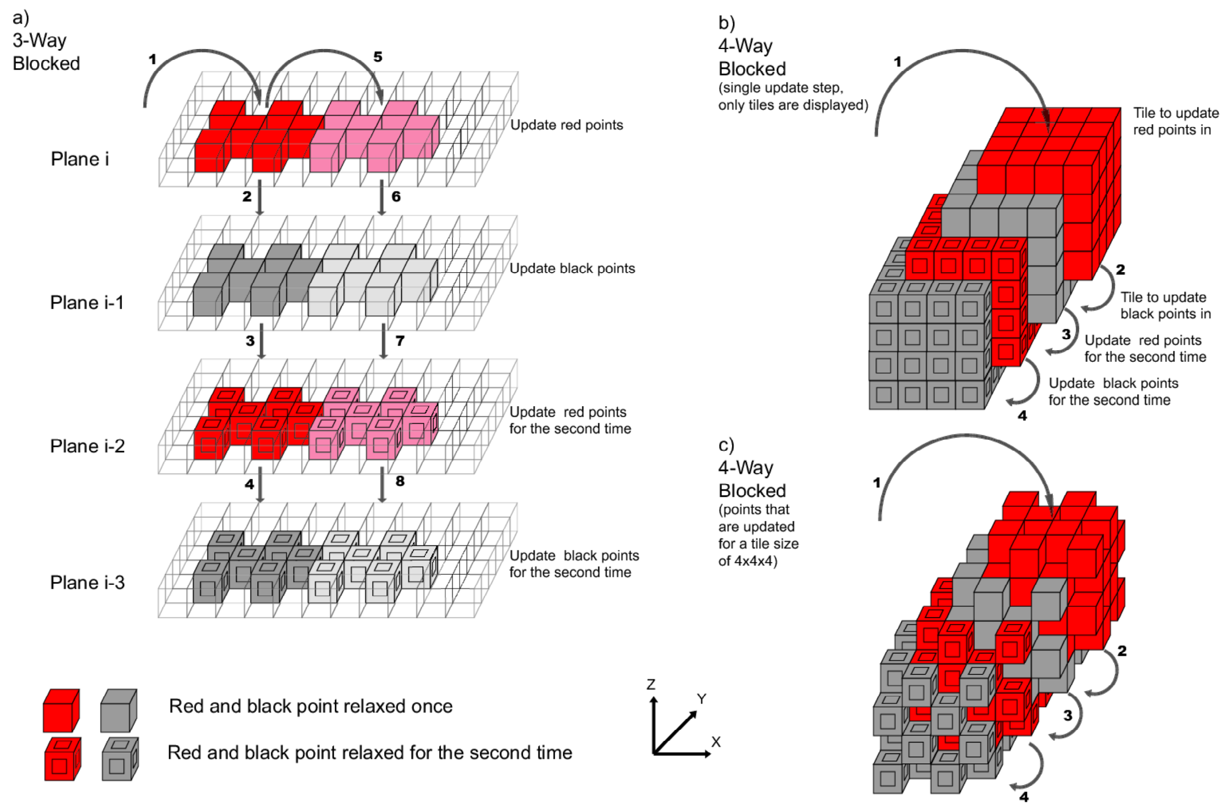
```
do KK= 1,N,W  // W = tile width
 do II= 1,N,H // H = tile height
  do J= 1,N
   do K= KK,min(KK+W-1,N)
    do I= II,min(II+H-1,N)
     C(I,J)= C(I,J)+A(I,K)*B(K,J)
    enddo
   enddo
  enddo
 enddo
enddo
```

Data access optimizations — loop blocking (cont'd)

Blocking the iteration loop of a linear stationary method means merging successive iterations into a single pass through the data set:
$x^{(k+1)} = Mx^{(k)} + d, \quad x^{(k+2)} = M(Mx^{(k)} + d) + d, \quad \ldots$

In addition, loops along spatial dimensions can be blocked

Example: Red/black Gauss–Seidel (e.g., as a multigrid smoother)


a)
3-Way
Blocked
Plane i
Plane i-1
Plane i-2
Plane i-3
1
2
3
4
5
6
7
8
Update red points
Update black points
Update red points for the second time
Update black points for the second time
Red and black point relaxed once
Red and black point relaxed for the second time
Z
Y
X
b)
4-Way
Blocked
(single update step, only tiles are displayed)
Tile to update red points in
Tile to update black points in
Update red points for the second time
Update black points for the second time
c)
4-Way
Blocked
(points that are updated for a tile size of 4x4x4)

Data access optimizations — other techniques

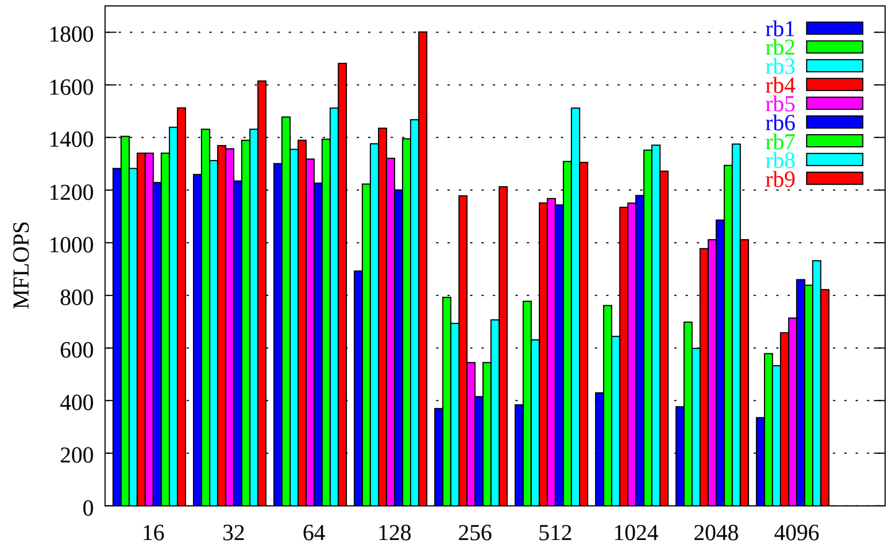
There is a variety of other data access optimizations

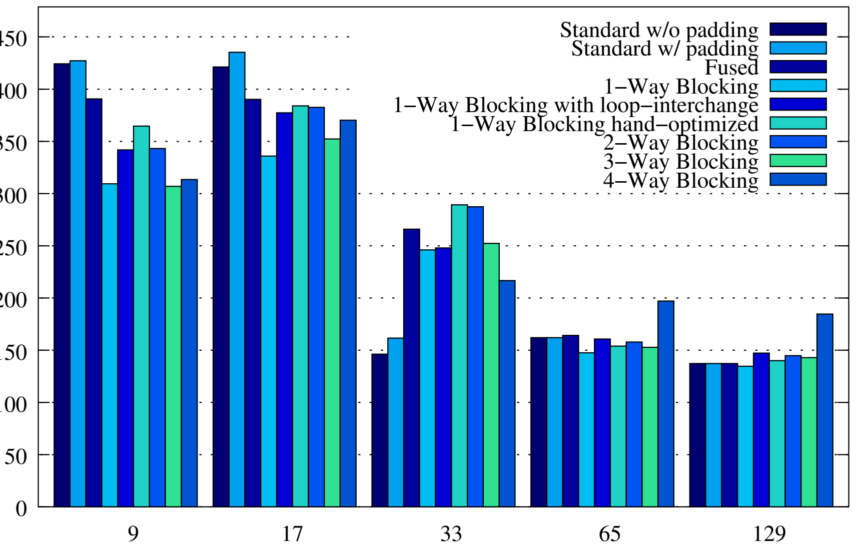
- *Loop interchange:* lessen the impact of non–unit stride accesses
- *Loop fusion:* reduce the number of sweeps through the data set $\Rightarrow$ Increase temporal locality
- *Data copying:* copy non–contiguous data to contiguous memory locations $\Rightarrow$ Reduce cache conflicts and/or drops in performance due to limited TLB capacity
- etc.

Performance results for iterative linear solvers

DiME project: data–local iterative methods for the efficient solution of PDEs: *http://www10.informatik.uni-erlangen.de/dime*

Speedups for Intel Pentium4 machine, 2.4 GHz, 4.8 GFLOPS peak, Intel ifc V7.0:

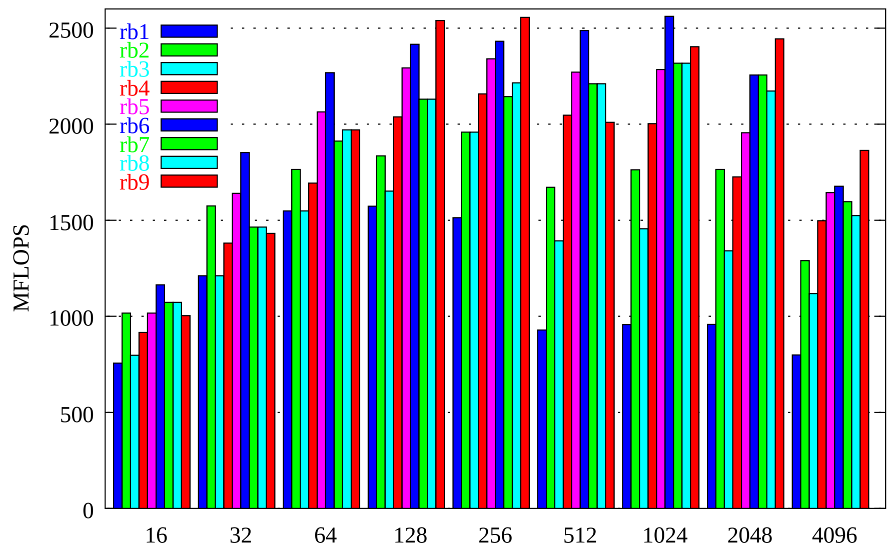


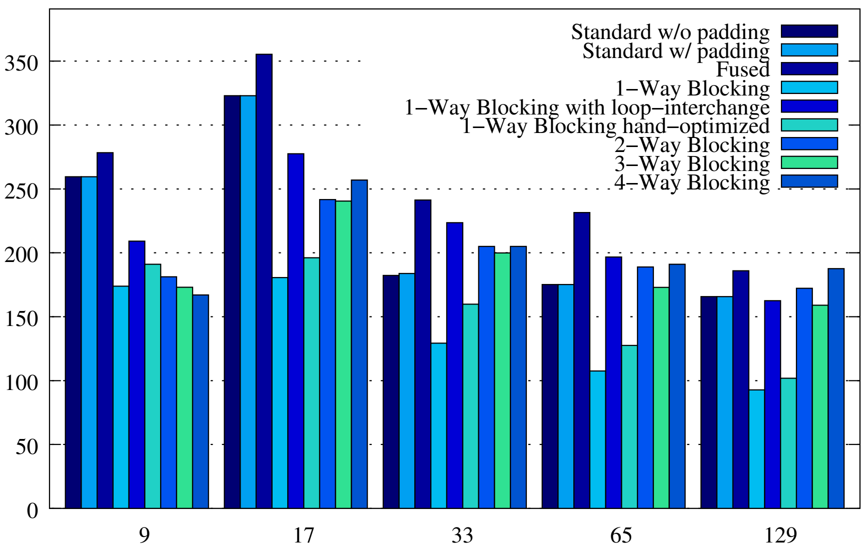


left: Gauss–Seidel, 2D, constant 5p stencil (C. Weiß)
right: V(2,2) MG cycles, 3D, variable 7p stencils

Performance results for iterative linear solvers (cont'd)

Preliminary!
Speedups for Intel Itanium2 machine, 900 MHz, 3.6 GFLOPS peak, Intel efc V7.0:
(No FP numbers cached in L1!)





left: Gauss–Seidel, 2D, constant 5p stencil (C. Weiß
right: V(2,2) MG cycles, 3D, variable 7p stencils

MFLOPS rates and speedups in 3D are often disappointing, more work to be done!

Impacts: TLB capacity, dynamic branch prediction, etc.

Lattice Boltzmann methods

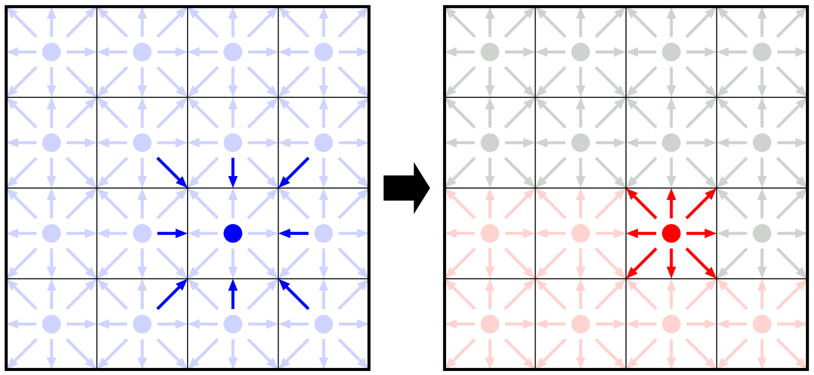
These data layout optimizations and data access optimizations can also be applied to *lattice Boltzmann methods*: particle–oriented CFD simulations

Particle distribution function $f(\mathbf{x}, \mathbf{u}, t)$ is discretized in space $(\mathbf{x})$, velocity $(\mathbf{u})$, and time (t), dynamic behavior is given by the *Boltzmann equation*:

$$\frac{\partial f}{\partial t} + \langle \mathbf{u}, \nabla f \rangle = \frac{1}{\lambda}\left(f - f^{(0)}\right) \quad ,$$

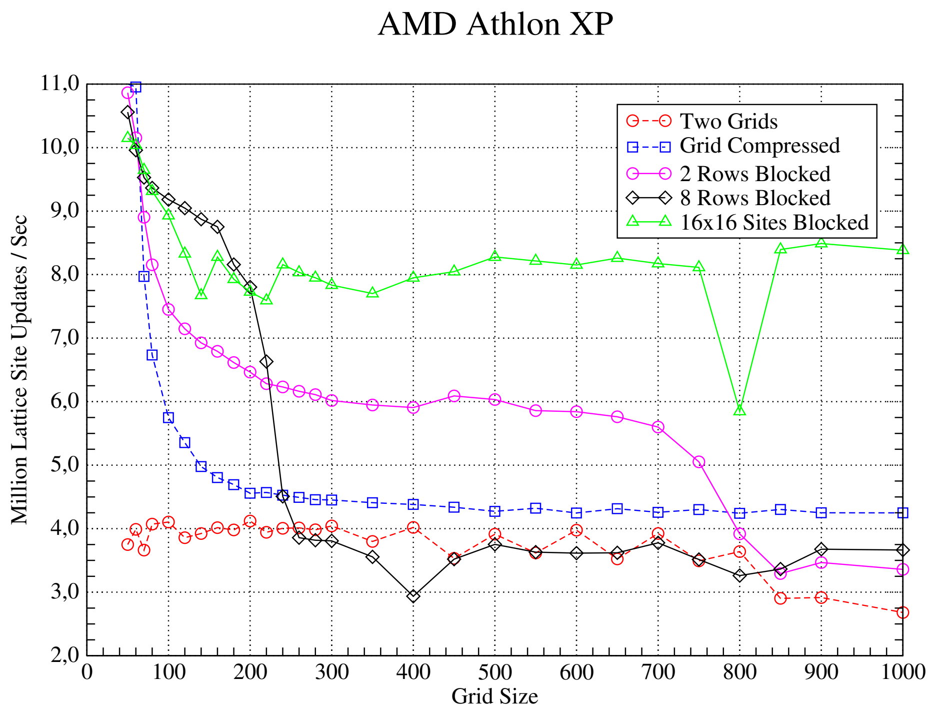
λ: relaxation time, f^0: equilibrium distribution function

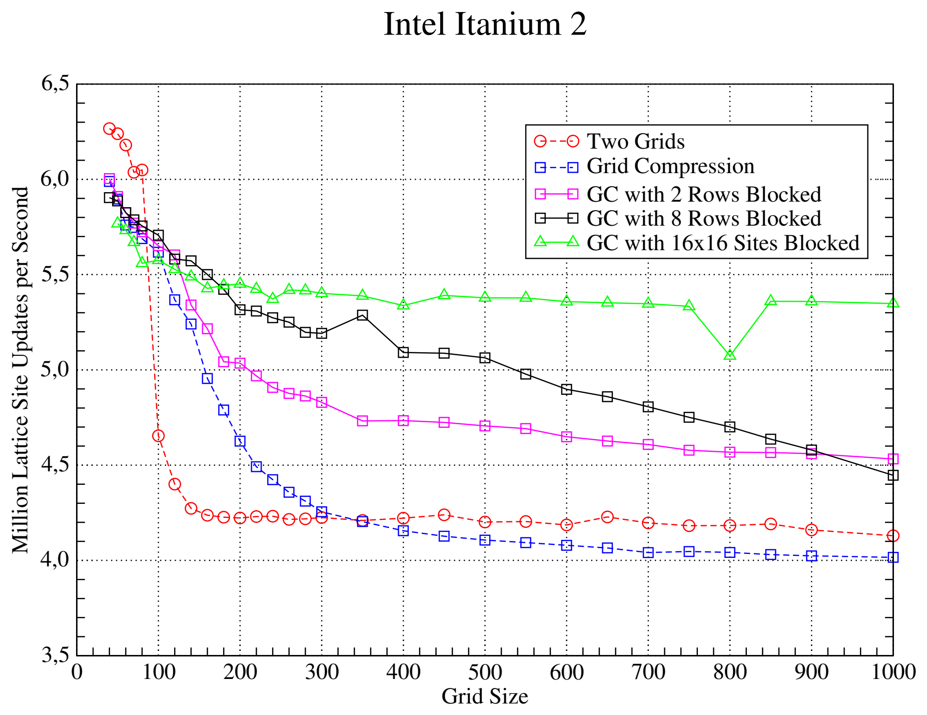
Algorithmic scheme: successive passes over the grid *(lattice)*, Jacobi–like update operation of grid cells *(lattice sites)* in each time step: *stream and collide*



Performance results for lattice Boltzmann methods

Code efficiency measured in "million lattice site updates / second"





left: 2D LB code, AMD Athlon machine, 1.4 GHz, gcc V3.2.1
right: 2D LB code, Intel Itanium2 machine, 900 MHz, Intel ecc V7.0

Cache–aware "non–standard" algorithms

So far: optimizations techniques that do not change numerical properties
(but: might trigger aggressive compiler optimizations, finite precision arithmetic)

One step further: (multilevel) algorithms with different numerical properties;
e.g., *Fully Adaptive Multigrid*

Essential component: *adaptive relaxation* on $Ax^* = b$,
$A = (a_{i,j})$: symmetric positive definite

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

Theorem: error reduction for one elementary relaxation step $x \leftarrow x + \theta_i(x) e_i$ can be written as follows:

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

Cache–aware "non–standard" algorithms (cont'd)

x: approximation of x^*,
ActiveSet: set of indices of nodes with "large" scaled residuals

Algorithm: adaptive relaxation

```
1: while ActiveSet $\neq 0$ do
2:     pick $i \in$ ActiveSet // Non–determinism: freedom of choice!
3:     if $|\theta_i(x)| > \theta$ then
4:         $x \leftarrow x + \theta_i(x) e_i$
5:         ActiveSet $\leftarrow$ ActiveSet $\cup$ Conn($i$)
6:     end if
7: end while
```

Fully Adaptive Multigrid: adaptive relaxation on the extended positive semidefinite system $\hat{A}\hat{x}^* = \hat{b}$ which represents the complete grid hierarchy (M. Griebel)

Combine multigrid efficiency with the freedom to choose any node to be updated from the active set (data locality $\Rightarrow$ higher cache utilization)

Overhead to maintain the active set ($\approx$ 25% runtime) motivates *patch–based* instead of *point–based* processing strategies (H. Lötzbeyer)

Conclusions

Gap between CPU speed and main memory performance will continue to increase

"Compiler–oriented" techniques to enhance cache performance:

- Data layout optimizations
- Data access optimizations

Introducing cache optimizations is tedious and error–prone ⇒ source–to–source compilers to automize the introduction of such transformations (e.g., *ROSE*, LLNL)

Fully Adaptive Multigrid as a "non–standard" multilevel algorithm that provides more flexibility w.r.t. the order of data accesses (i.e., the implementation of the active set)

"Turing machines are not enough", counting flops is no longer an adequate measure of efficiency! Need for *complexity models* and *locality metrics* for hierarchical memories (e.g., *Memtropy*: measure to quantify the regularity of memory references (D. Keyes))