Hardware–oriented locality optimizations for iterative methods

Markus Kowarschik
markus.kowarschik@informatik.uni-erlangen.de

Lehrstuhl für Systemsimulation (Prof. Dr. U. Rüde)
Institut für Informatik
Friedrich–Alexander–Universität Erlangen–Nürnberg
Germany

This project is being supported by the DFG:
DiME — data–local iterative methods
(together with LRR–TUM, Prof. Dr. A. Bode)

Outline

- Motivating example
- Cache design aspects
- Code profiling — hardware support and tools
- Techniques to enhance cache utilization (preserving numerical properties)
  - Data layout optimizations
  - Data access optimizations
- Performance results
- Cache–oriented "non–standard" algorithms
- Conclusions

Motivating (frustrating?) example

Theoretically ...
modern workstations based on superscalar CPUs can do up to
4000 MFLOPS per CPU (i.e., $4 \cdot 10^9$ floating–point operations per second)

In practice ...
we often obtain disappointing results

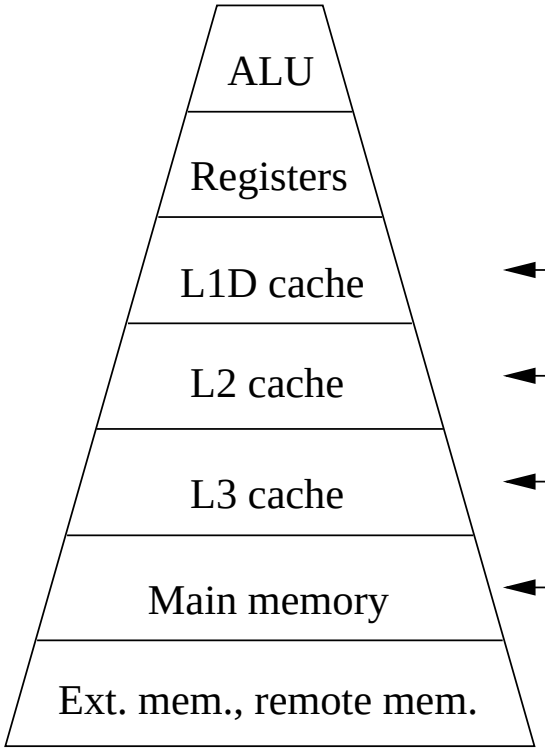
Example: 3D Gauss–Seidel on a Digital PWS 500au (1000 MFLOPS peak):

grid size	# unknowns	MFLOPS
16	4096	415
32	32768	194
64	262144	76
128	$\approx 2.1 \cdot 10^6$	73

$\Rightarrow$ Need to understand cache effects!

Cache design — a memory hierarchy example

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

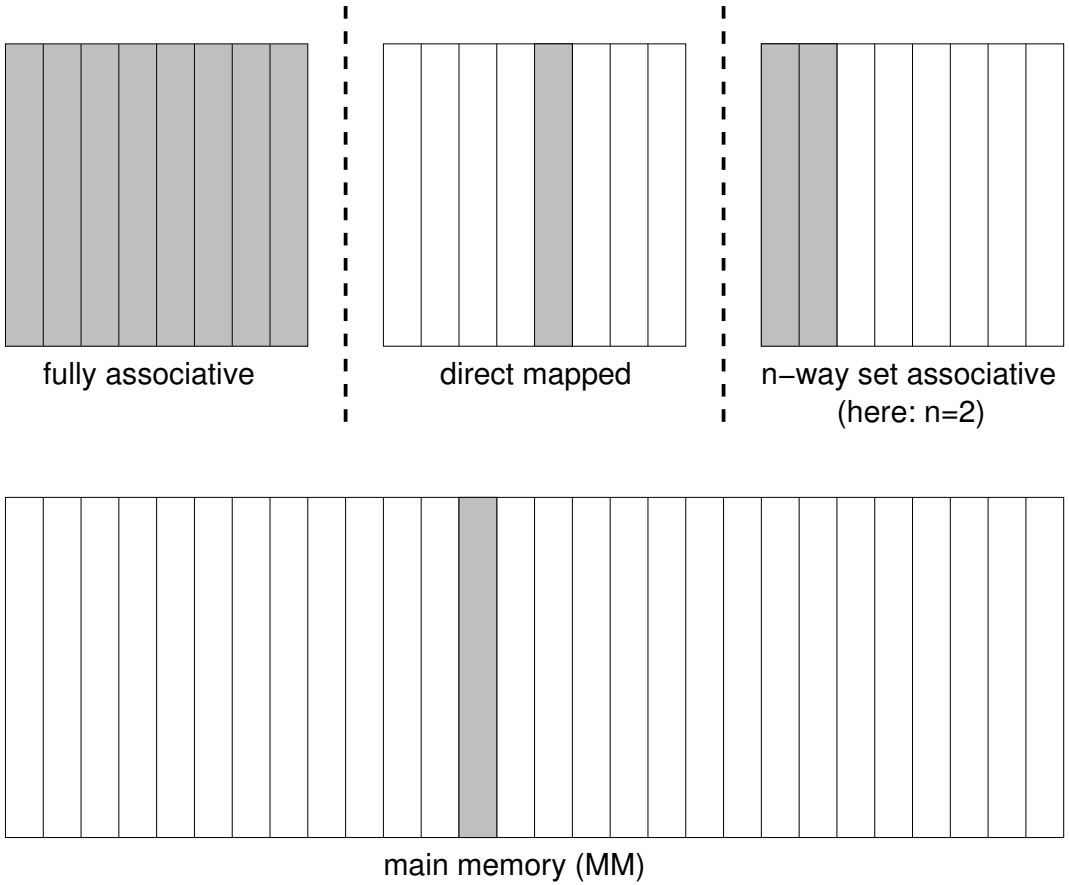


Capacity	Bandwidth	Latency
16 KB	16 GB/s	1 cycle
256 KB	32 GB/s	5+ cycles
1.5/3 MB	32 GB/s	12+ cycles
2 GB	6.4 GB/s	>100 cycles

Inevitable: exploit the cache architecture as efficiently as possible!

Cache design — associativity

Denotes the number of cache lines where a main memory block may be copied to



Low associativity (n small) ⇒ High potential for *cache conflict misses, cache thrashing*

Code profiling — hardware support and tools

Hardware performance counters (= special CPU registers) are used to count various events:

- Data cache misses (for different cache levels)
- TLB (translation lookaside buffer) misses
- Branch mispredictions
- Floating–point and/or integer operations
- etc.

Profiling tools (freely available):

- *PAPI: Performance Application Programming Interface*, Univ. of Tennessee
- *PCL: Performance Counter Library*, FZ Jülich
- *DCPI: Digital Continuous Profiling Infrastructure*, Digital/Compaq only
- etc.

Code profiling — PCL

We use a *Digital PWS 500au* machine for demonstration purposes: Alpha A21164 CPU, 500 MHz, 1000 MFLOPS peak, 3 on–chip performance counters

Hardware performance monitor based on PCL: `hpm`

Example: 2D "vanilla" multigrid code, structured grids

```
% hpm --events PCL_CYCLES,PCL_MFLOPS ./mg
hpm: elapsed time: 5.172 s
hpm: counter 0   : 2564941490 PCL_CYCLES
hpm: counter 1   : 19.635955 PCL_MFLOPS
```

This is $< 2\%$ peak!

Code profiling — DCPI (1)

```
% dcpiprof ./mg
Column              Total  Period (for events)
------              -----  ------
dmiss               45745    4096

The numbers given below are the number of samples for each
listed event type or, for the ratio of two event types, the
ratio of the number of samples for the two event types.
====================================================
dmiss       %    cum% procedure                  image
33320  72.84%  72.84% mgSmooth                   ./mg
10008  21.88%  94.72% mgRestriction              ./mg
 2411   5.27%  99.99% mgProlongation             ./mg
    3   0.01%  99.99% mgInitGrid                 ./mg
    2   0.00% 100.00% mgDirectSolve              ./mg
    1   0.00% 100.00% mgVcycle                   ./mg
```

Code profiling — DCPI (2)

```
% dcpiwhatcg ./mg

 I-cache (not ITB)   0.1% to  7.4%
  ITB/I-cache miss   0.0% to  0.0%
      D-cache miss  24.2% to 27.6%
          DTB miss  53.3% to 57.7%
      Write buffer   0.0% to  0.3%
   Synchronization   0.0% to  0.0%

 Branch mispredict   0.0% to  0.0%
         IMUL busy   0.0% to  0.0%
         FDIV busy   0.0% to  0.5%
             Other   0.0% to  0.0%

 Unexplained stall   0.4% to  0.4%
  Unexplained gain  -0.7% to -0.7%
-----------------------------------------
  Subtotal dynamic                 85.1%
```

```
              Slotting       0.5%
        Ra dependency        3.0%
        Rb dependency        1.6%
        Rc dependency        0.0%
        FU dependency        0.5%
-----------------------------------------
   Subtotal static                    5.6%
-----------------------------------------
       Total stall                   90.7%

             Useful        7.9%
               Nops        1.3%
-----------------------------------------
   Total execution                    9.3%
```

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

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:

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

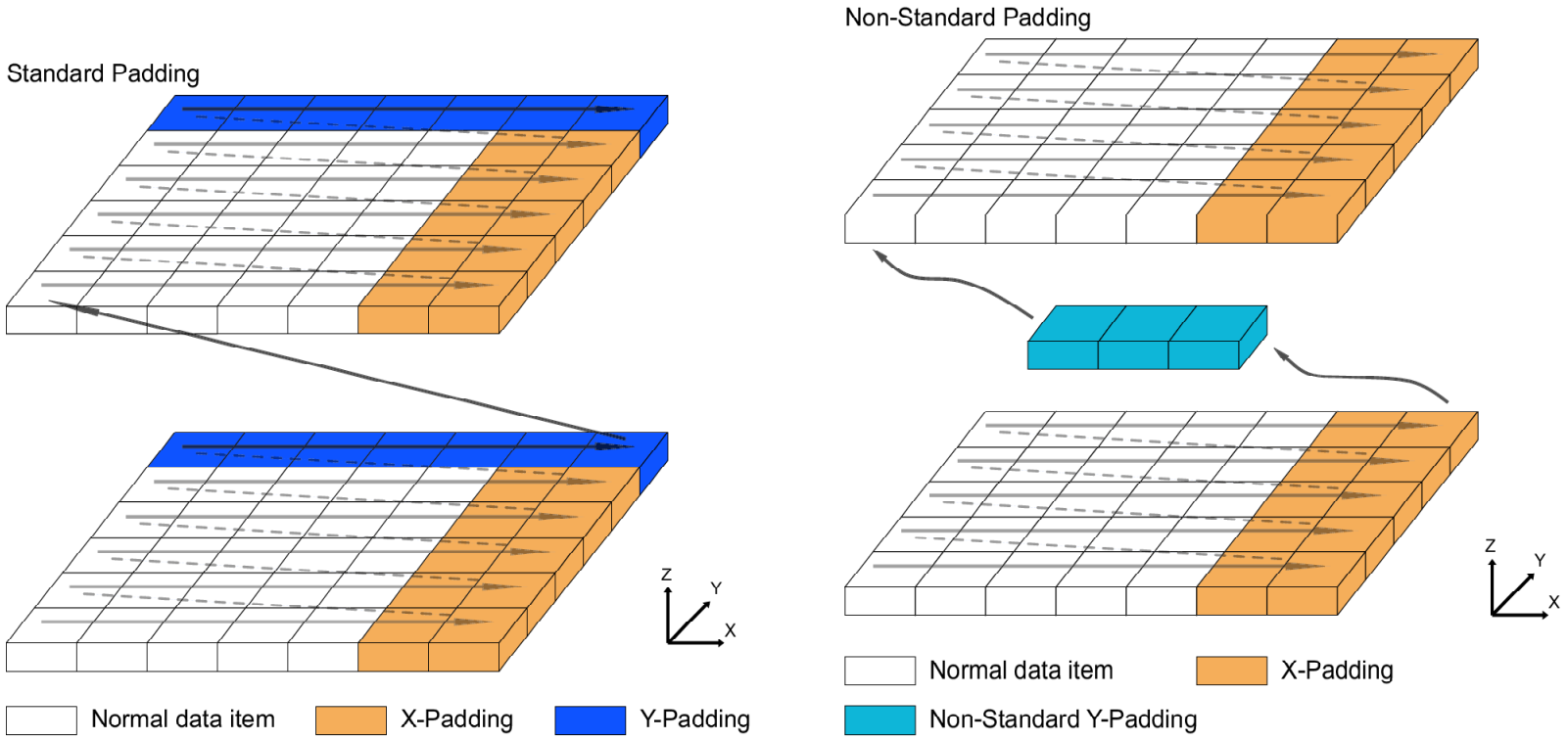
```
typedef struct {
  double f;
  double cCenter, cNorth, cEast, cSouth, cWest;
} eqnData;

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

Data layout optimizations — array padding (1)

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 (2)

Padding approaches:

- Analytic/Algebraic techniques (Rivera/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) (1)

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**

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:

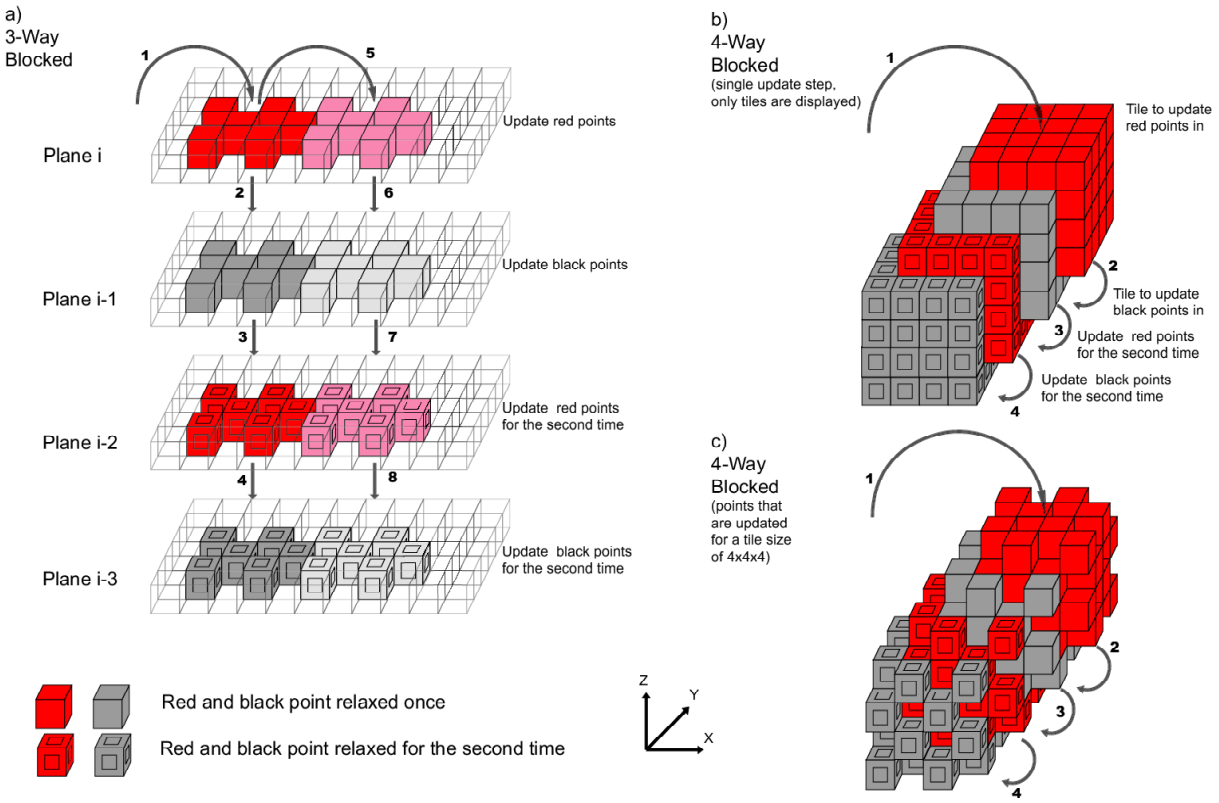
```
do KK= 1,N,W  // W = tile width
 do II= 1,N,H // H = tile height
  do J= 1,N
   do K= KK,min(KK+W-1,N)
    do I= II,min(II+H-1,N)
     C(I,J)= C(I,J)+A(I,K)*B(K,J)
    enddo
   enddo
  enddo
 enddo
enddo
```

Data access optimizations — loop blocking (2)

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)



Data access optimizations — other techniques

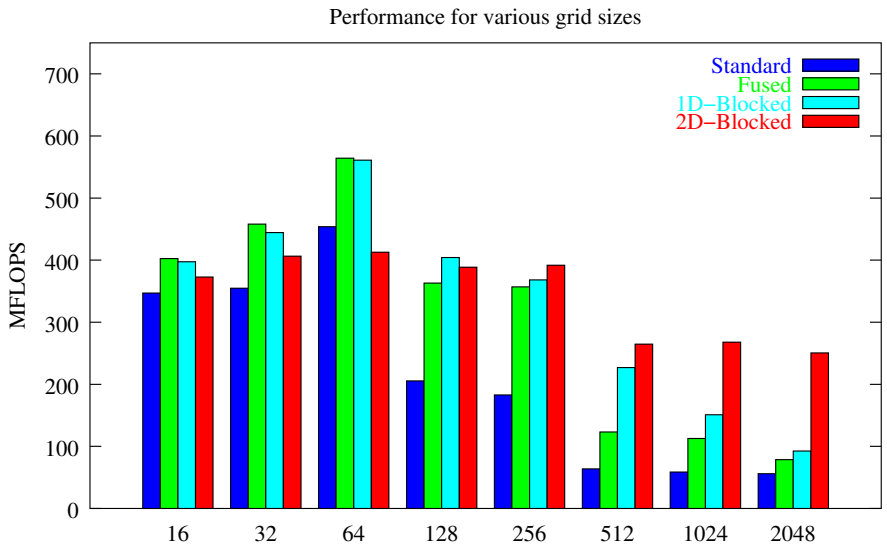
There is a variety of other data access optimizations

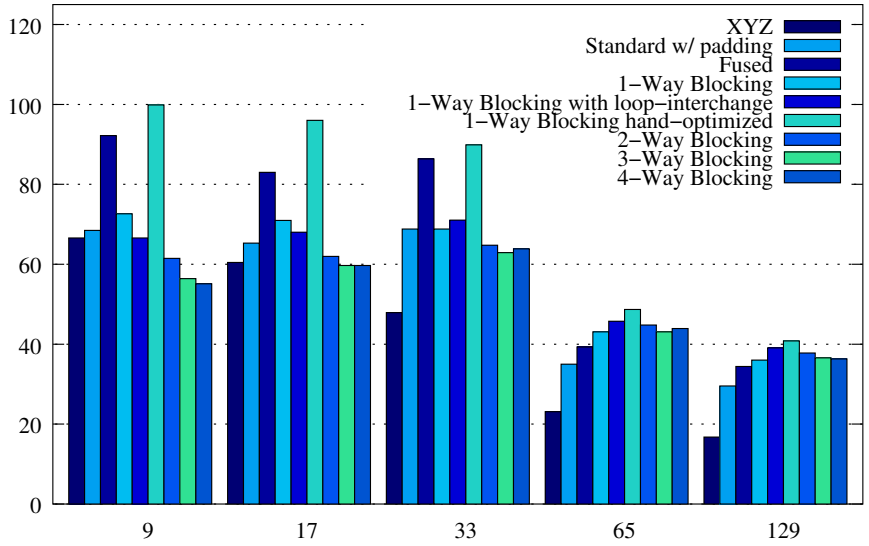
- *Loop interchange:* lessen the impact of non–unit stride accesses
- *Loop fusion:* reduce the number of sweeps through the data set $\Rightarrow$ Increase temporal locality
- *Data copying:* copy non–contiguous data to contiguous memory locations $\Rightarrow$ Reduce cache conflicts and/or drops in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

Performance results (1)

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

Speedups for Alpha A21164 machine, 500 MHz, 1000 MFLOPS peak:

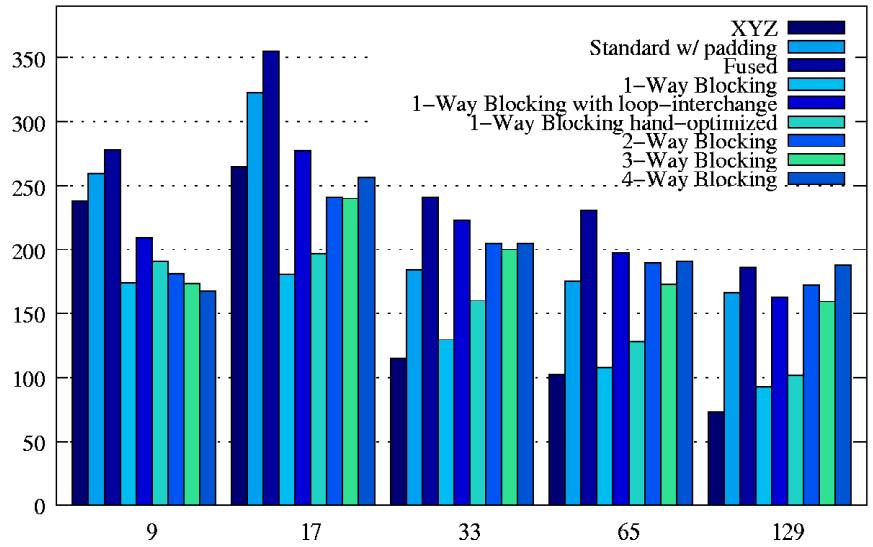


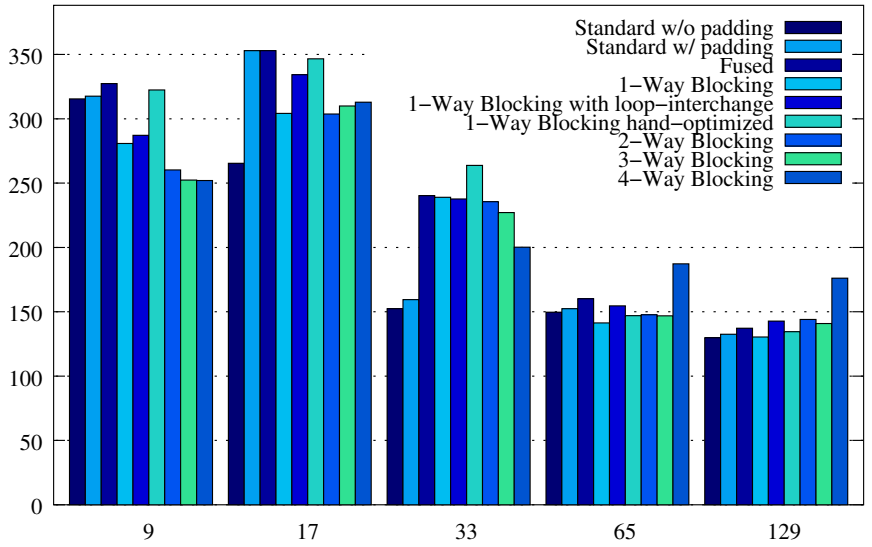


left: Gauss–Seidel, 2D, constant 5p stencil (C. Weiß, formerly LRR–TUM)
right: V(2,2) MG cycles, 3D, variable 7p stencils

Performance results (2)

Same codes on different architectures: V(2,2) MG cycles, 3D, variable 7p stencils:





left: Intel Itanium2 based HP zx6000, 900 MHz, 1.5 MB L3, 3.6 GFLOPS peak
Preliminary results!!!
right: Pentium 4–based Linux PC, 2.4 GHz, 4.8 GFLOPS peak

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

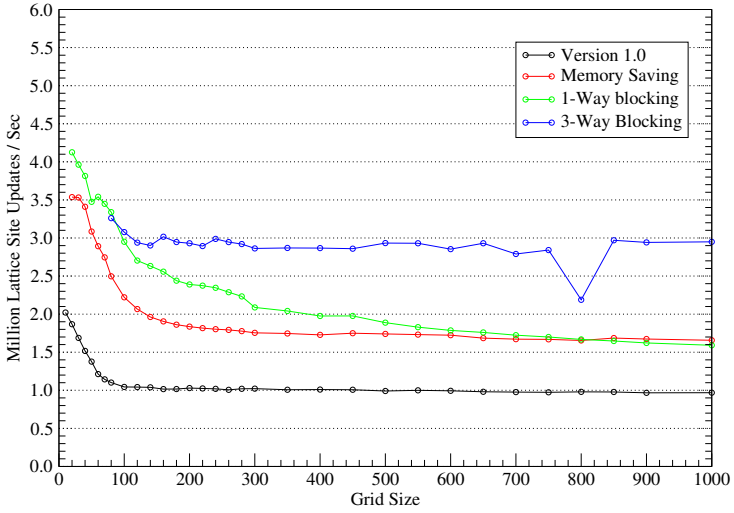
Impacts: TLB capacity, dynamic branch prediction, etc.

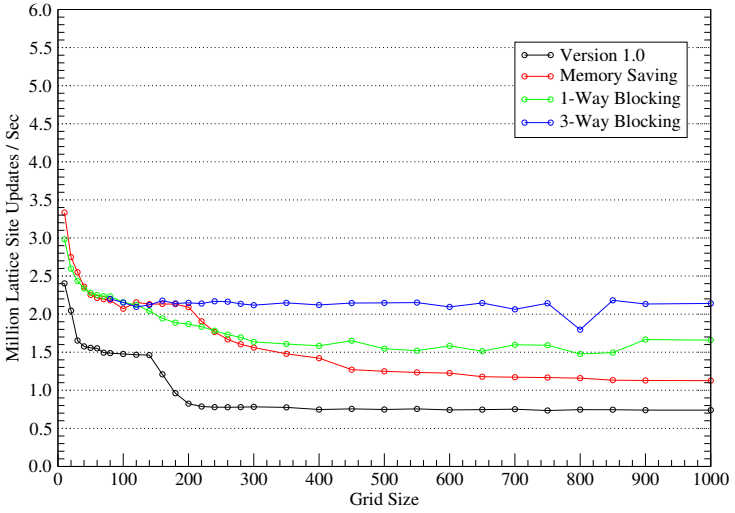
Performance results (3)

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

Successive passes through the highly structured data set, Jacobi–like update operation ("stream and collide") of grid cells *(lattice sites)* in each time step

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





left: 2D LB code, AMD Athlon machine, 700 MHz
right: 2D LB code, Alpha A21164 machine, 500 MHz

Cache–oriented "non–standard" algorithms (1)

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* (U. Rüde)

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:

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

Cache–oriented "non–standard" algorithms (2)

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 $\Rightarrow$ 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))