Cache Performance Optimizations for Multigrid Computations

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

Thanks to Nils Thürey, Christian Weiß, and Jens Wilke

Lehrstuhl für Informatik 10 (Systemsimulation)
Institut für Informatik
Friedrich–Alexander–Universität Erlangen–Nürnberg
Germany

in collaboration with
Lehrstuhl für Rechnertechnik und Rechnerorganisation, TU München

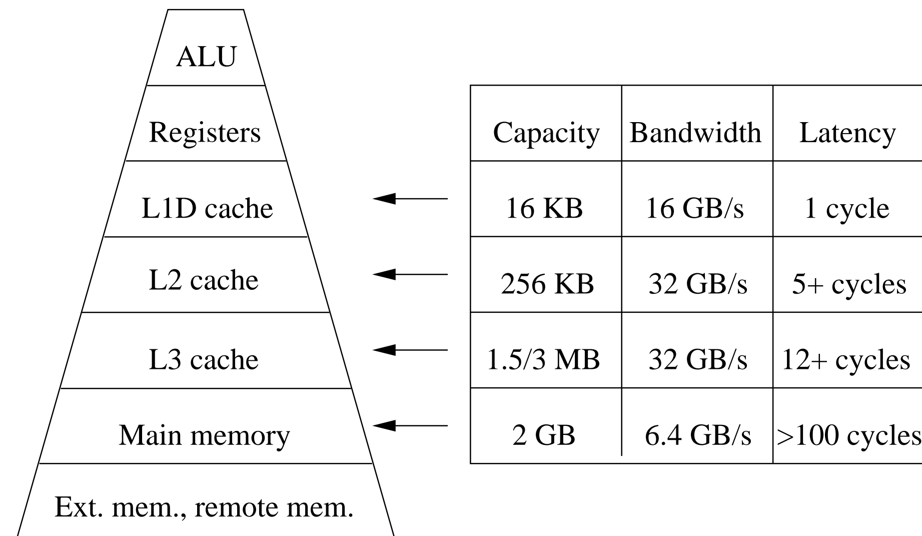
This project is being supported by the *Deutsche Forschungsgemeinschaft*:
DiME — data–local iterative methods

Outline

- A memory hierarchy example
- "Compiler–oriented" optimization techniques to enhance cache utilization; they do **not** change the numerical properties of the algorithms
  - Data layout optimizations
  - Data access optimizations
- Cache–oriented "non–standard" algorithms
- 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!

Future trend: increasing cache sizes; e.g.,
Intel Madison (2003/2004): 9MB L3 cache on–chip,
Intel Montecito (2005): “larger L3 cache”

Techniques to enhance cache utilization

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

- *Data layout optimizations:*
  Optimize how the data is arranged in memory

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

Data layout optimizations — cache–aware data structures

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

Example: Gauss–Seidel on $Au = f$:

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

In particular: 2D, 5–point stencils, C code:

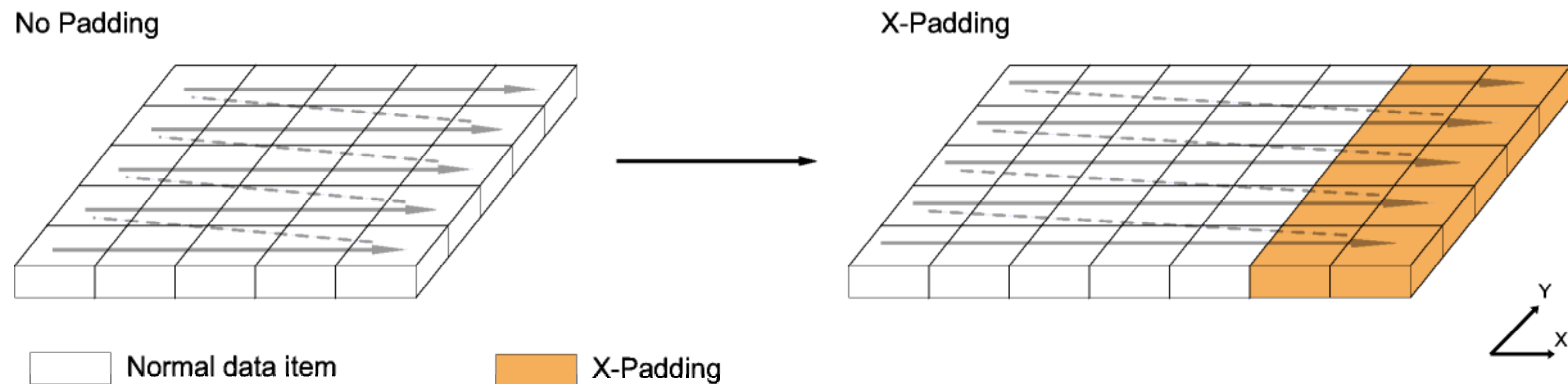
```
typedef struct {
  double f;
  double coeffCenter, coeffNorth, coeffEast, coeffSouth, coeffWest;
} equationData;

double       u[N][N];            // Solution vector
equationData 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: 2D arrays

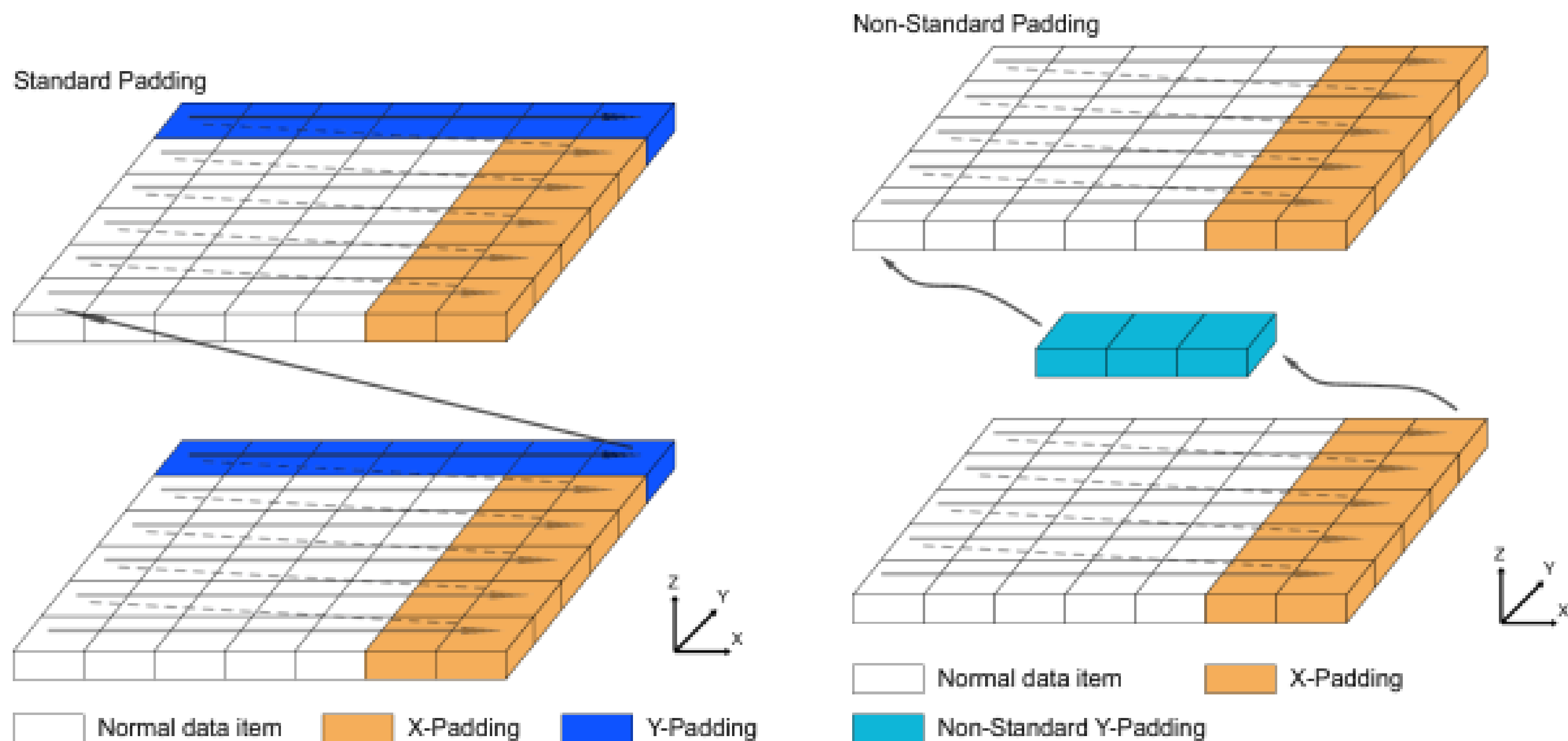


Standard 2D padding in FORTRAN77:

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

Data layout optimizations — array padding (cont'd)

Example: 3D arrays



Non–standard approach is harder to implement, but it saves memory

Current research topic: derivation of appropriate paddings — algebraic techniques vs. empirical parameter search

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**

Formally:

Before loop blocking:

```
do I= 1,N
  ...
enddo
```

After loop blocking:

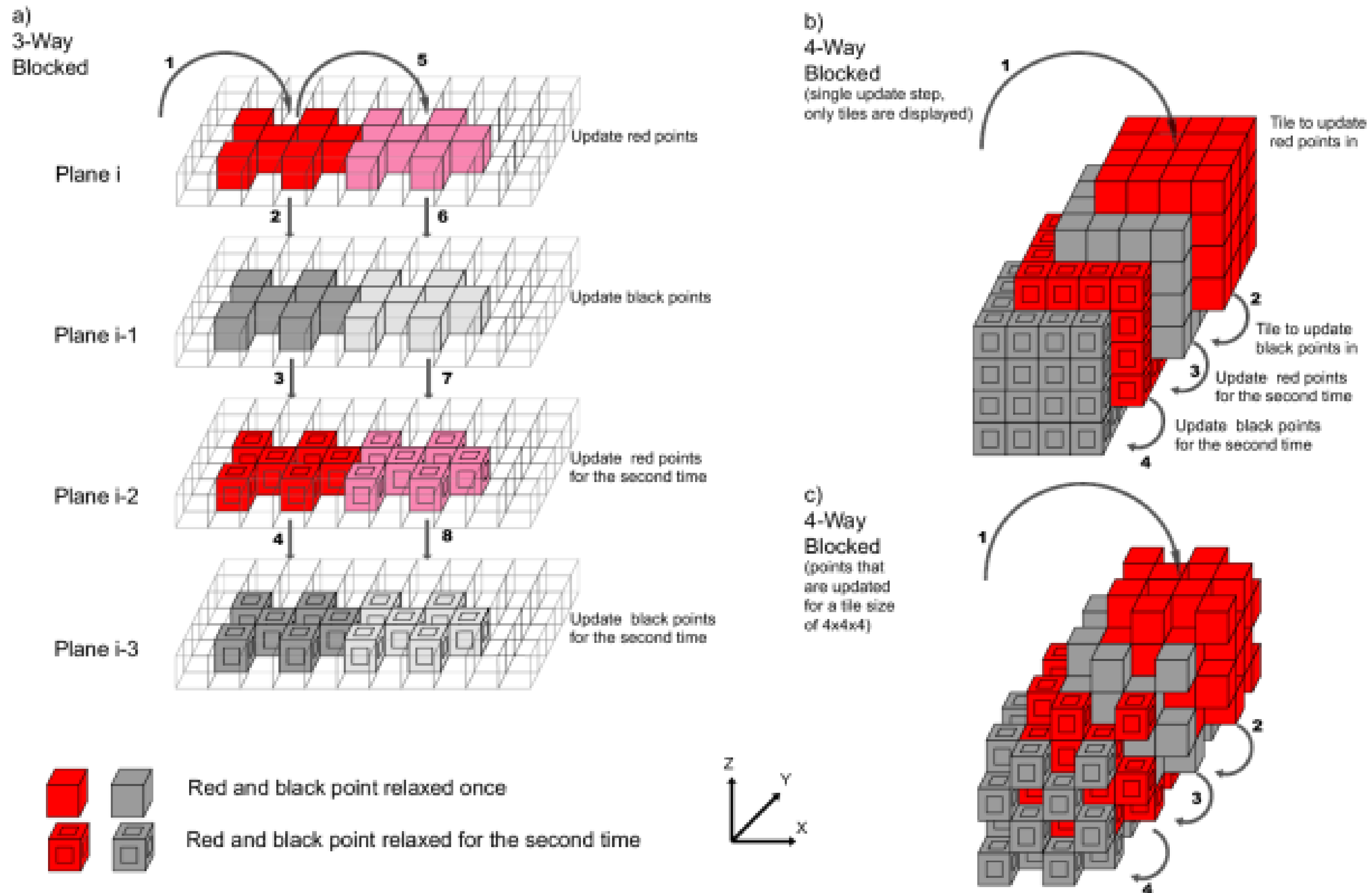
```
do II= 1,N,blocksize
  do I= II,min(II+blocksize-1,N)
    ...
  enddo
enddo
```

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

In addition, *loops along spatial dimensions* can be blocked ...

Data access optimizations — loop blocking (cont'd)

Example: 3D red/black Gauss–Seidel (e.g., as a multigrid smoother)



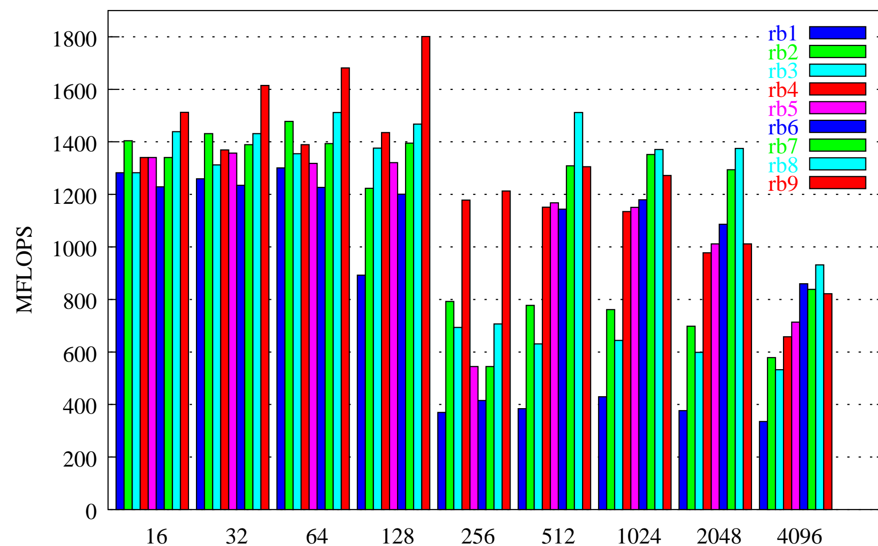
Data access optimizations — other techniques

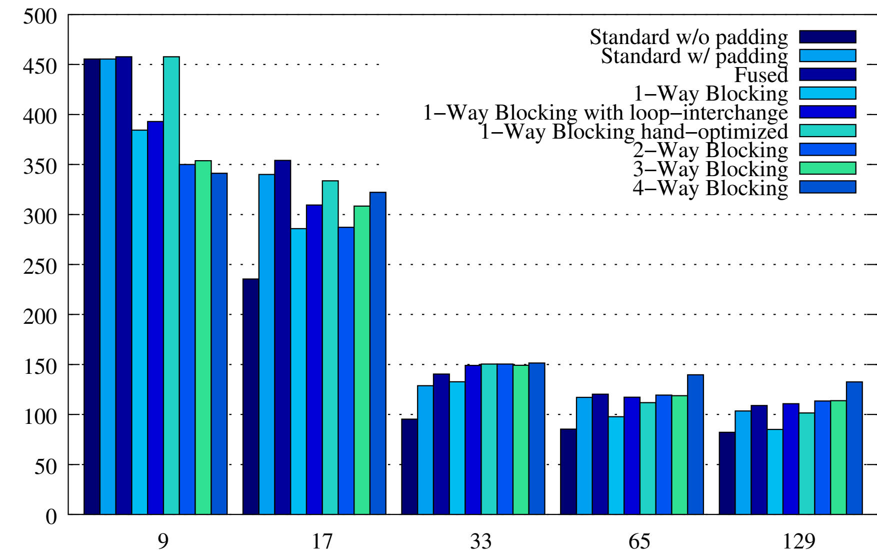
There is a variety of additional data access optimizations:

- *Loop interchange:* lessen the impact of non–unit stride accesses
- *Loop fusion:* reduce the number of sweeps through the data set
  ⇒ Increase temporal locality
- *Data copying:* copy non–contiguous data to contiguous memory locations
  ⇒ 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*





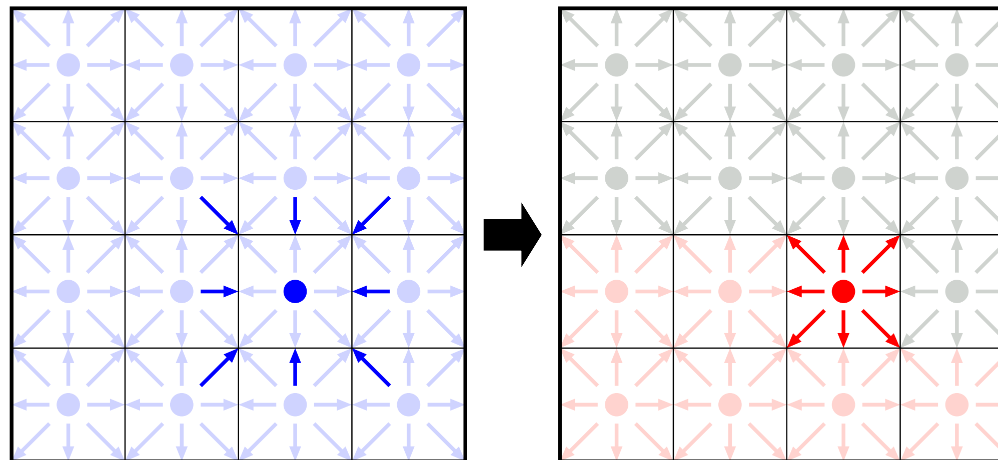
left: Gauss–Seidel, 2D, constant 5p stencils, F77,
Intel Pentium4, 2.4 GHz, 4.8 GFLOPS peak, Intel ifc V7.0
right: MG V(2,2) cycles, cache–optimized GS smoother, 3D, variable 7p stencils, F77,
AMD Athlon XP2400+, 2.0 GHz, 3.4 GFLOPS peak, gcc 3.2.1

MFLOPS rates and speedups in 3D are often disappointing, more work to be done ...
Impacts: TLB capacity, dynamic branch prediction, hardware–based prefetching, etc.

Lattice Boltzmann methods

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

Algorithmic structure parallels the structure of iterative linear solvers:
Successive passes over the grid, Jacobi–like update operation of grid cells in each time step: *stream and collide*



More details and performance results in Thomas Pohl's talk ...

Cache–oriented “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* (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)*

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

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

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

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

Algorithm: adaptive relaxation

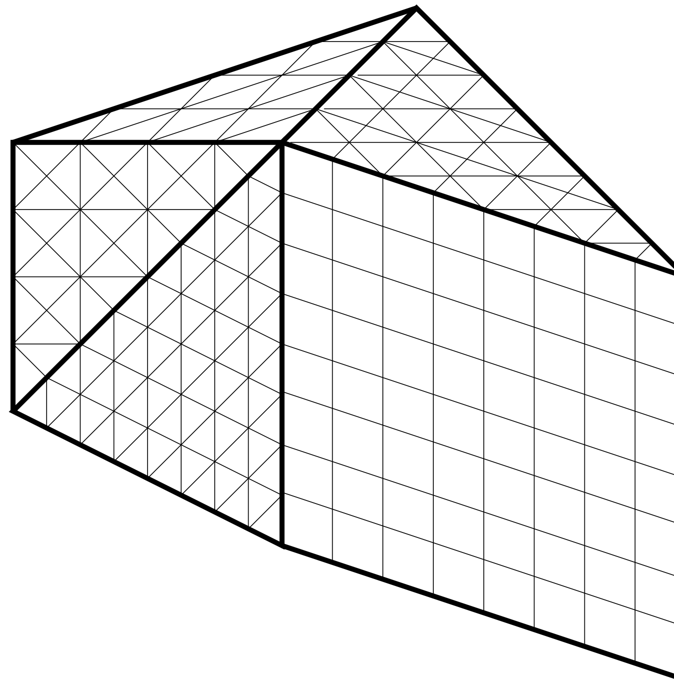
1: **while** ActiveSet $\neq 0$ **do**
2: pick $i \in$ ActiveSet // Non–determinism: freedom of choice!
3: **if** $|\theta_i(x)| > \theta$ **then**
4: $x \leftarrow x + \theta_i(x) e_i$
5: ActiveSet $\leftarrow$ ActiveSet $\cup$ Conn(i)
6: **end if**
7: **end while**

Fully adaptive multigrid:
Formally: adaptive relaxation on the (well–conditioned and positive semidefinite) system $\hat{A}\hat{x}^* = \hat{b}$ which represents the entire grid hierarchy (U. Rüde, M. Griebel)

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

Cache–oriented “non–standard” algorithms (cont’d)

Overhead to maintain the active set motivates *patch–based* instead of *node–based* processing strategies, need to “translate” the above algorithm



A patch–based fully adaptive multigrid algorithm is currently being implemented, more results to be presented soon ...

This adaptive processing strategy is closely related to the concept of *hierarchical hybrid grids*, more details in Frank Hülsemann’s talk ...

Conclusions

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

Counting flops is an insufficient 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)