Cache–Aware Numerical Computations

What scientific computing people do in order to cache–tune their codes

Markus Kowarschik
mk@cs.fau.de

System Simulation Group (Informatik 10)
Department of Computer Science
Friedrich–Alexander–University Erlangen–Nürnberg
Germany

Outline

- Motivating example
- Cache design issues
- Code profiling — hardware support and tools
- Cache optimizations for numerically intensive codes
  - Data layout optimizations
  - Data access optimizations
- Algorithmic locality, cache–oriented algorithms, etc.

Motivating (frustrating?) example

Theoretically ...
modern workstations based on superscalar RISC processors can do by far more than 1000 MFLOPS

In practice ...
we often obtain disappointing results

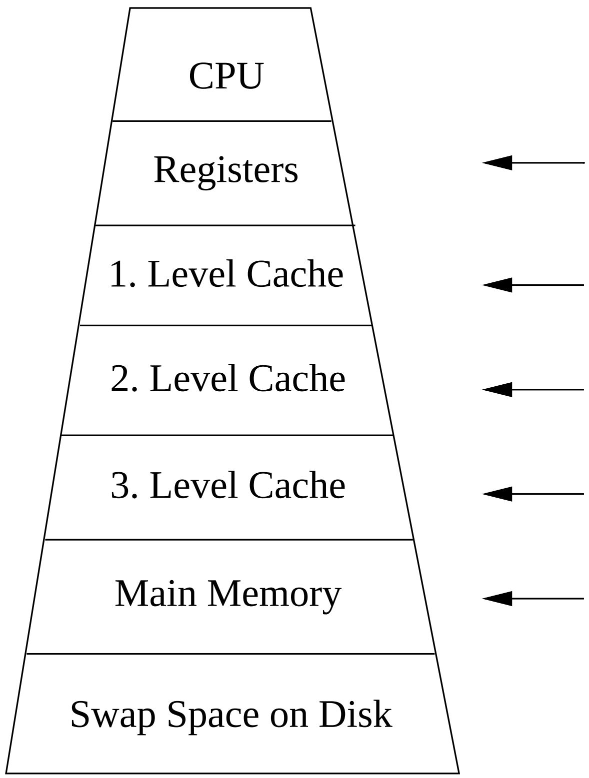
Example: 3D Gauss–Seidel iteration on a Digital PWS 500au:

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 issues — a memory hierarchy example

Digital PWS 500au memory architecture:

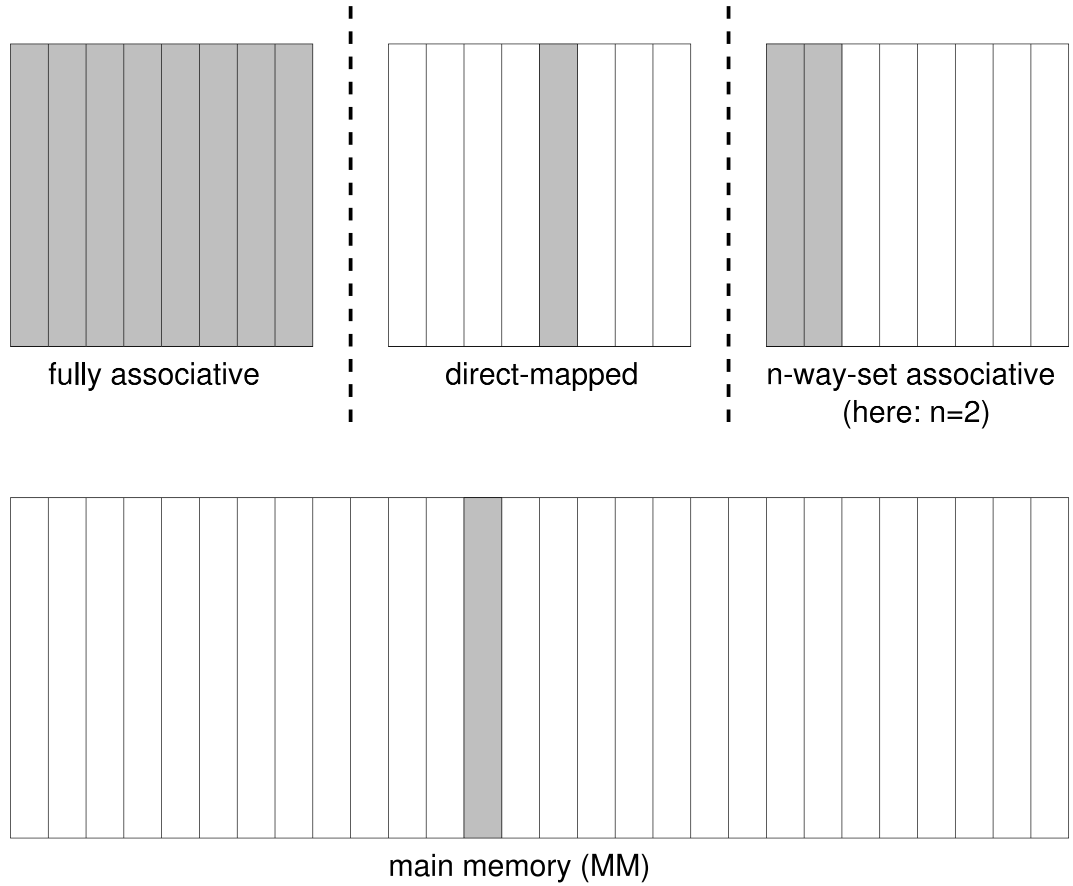


Capacity	Bandwidth	Latency
256 Bytes	24000 MB/s	2 ns
8 KBytes	16000 MB/s	2 ns
96 KBytes	8000 MB/s	6 ns
2 MBytes	888 MB/s	24 ns
1536 MBytes	1000 MB/s	112 ns

Exploit the cache architecture more efficiently!

Cache design issues — associativity

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



Low associativity (n small) $\Rightarrow$ High potential for *cache conflict misses, cache thrashing*

This is different from external memory management!

Code profiling — hardware support

Hardware performance counters (= dedicated CPU registers) can be used to count various events:

- Data cache misses (for different cache levels)
- Instruction cache misses
- TLB (translation lookaside buffer) misses
  (Especially dramatic for 3D computational domains!)
- Branch mispredictions
- Floating–point and/or integer operations
- Load/store instructions
- Stall cycles of integer/floating–point units
- etc.

Code profiling — tools

We consider the following two profiling tools:

1. *PCL: Performance Counter Library:*
   - R. Berrendorf et al., FZ Juelich, Germany
   - Available for many platforms (Portability!)
   - Usable from outside and from inside the code (library calls, C, C++, Fortran, and Java interfaces)
   - Similar to *PAPI (Performance Application Programming Interface)*
   - `www.fz-juelich.de/zam/PCL`

2. *DCPI: Compaq (Digital) Continuous Profiling Infrastructure:*
   - Only for Alpha machines running under Compaq Tru64 UNIX
   - Code execution is watched by a profiling daemon
   - Can only be used from outside the code
   - `www.tru64unix.compaq.com/dcpi`

Code profiling — target implementation

We consider a 2D multigrid code on structured grids:

- Written in C
- Double precision floating–point arithmetic
- 5–point stencils
- Dirichlet boundary conditions
- Red/black Gauss–Seidel smoother
- Residual restriction by full weighting
- Prolongation of the correction by bilinear interpolation
- Direct solver for the problems on the coarsest grid
  (LU factorization, LAPACK)

How well does this code perform?

Code profiling — PCL from outside the code

We use a *Digital PWS 500au* machine for demonstration purposes: Alpha 21164 CPU, 500 MHz, peak performance: 1000 MFLOPS, 3 on–chip performance counters

Hardware performance monitor based on PCL: `hpm`

Example:

```
% 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
```

Note that this is $< 2\%$ peak!

Code profiling — PCL from inside the code

```
#include <pcl.h>

int main(int argc, char **argv) {
  PCL_CNT_TYPE i_result[2];
  PCL_FP_CNT_TYPE fp_result[2];
  int counter_list[] = {PCL_FP_INSTR, PCL_MFLOPS}, res;
  unsigned int flags = PCL_MODE_USER;
  PCL_DESCR_TYPE descr;

  PCLinit(&descr);

  if (PCLquery(descr,counter_list,2,flags) != PCL_SUCCESS)
    printf("These two events are not available!\n");
  else {
    PCLstart(descr, counter_list, 2, flags);

    // *** DO WORK ***

    PCLstop(descr, i_result, fp_result, 2);
    printf("%i FP-instructions, MFLOPS: %f\n",
           i_result[0], fp_result[1]);
  }

  PCLexit(descr);
  return 0;
}
```

Code profiling — DCPI (Compaq TRU64 UNIX only)

How to proceed:

1. Start the DCPI daemon
2. Run your code
3. Stop the daemon
4. Use DCPI tools to analyze the profiling data:
   - `dcpiwhatcg`: Where have all the cycles gone?
   - `dcpiprof`: Breakdown of CPU time by procedures
   - `dcpilist`: Code listing annotated with profiling data
   - `dcpitopstalls`: Ranking of instructions causing stall cycles
   - etc.

Code profiling — DCPI example (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 example (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:

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

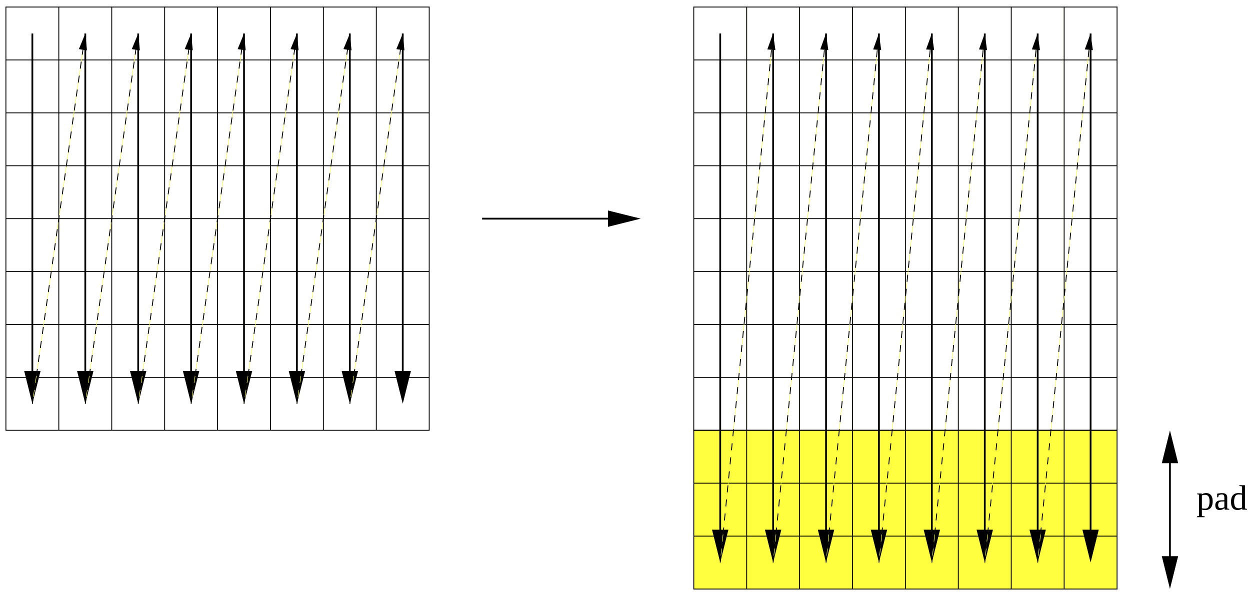
```
typedef struct {
  double f;
  double cCenter, cNorth, cEast, cSouth, cWest;
} eqnData;

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: 2D array, FORTRAN77 (column major ordering)

`double precision u(1024, 1024)` becomes
`double precision u(1024+pad, 1024)`, *problem:* `pad` = ?



Padding 3D arrays is more involved

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*
  - Examples:
    - *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

Popular 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:

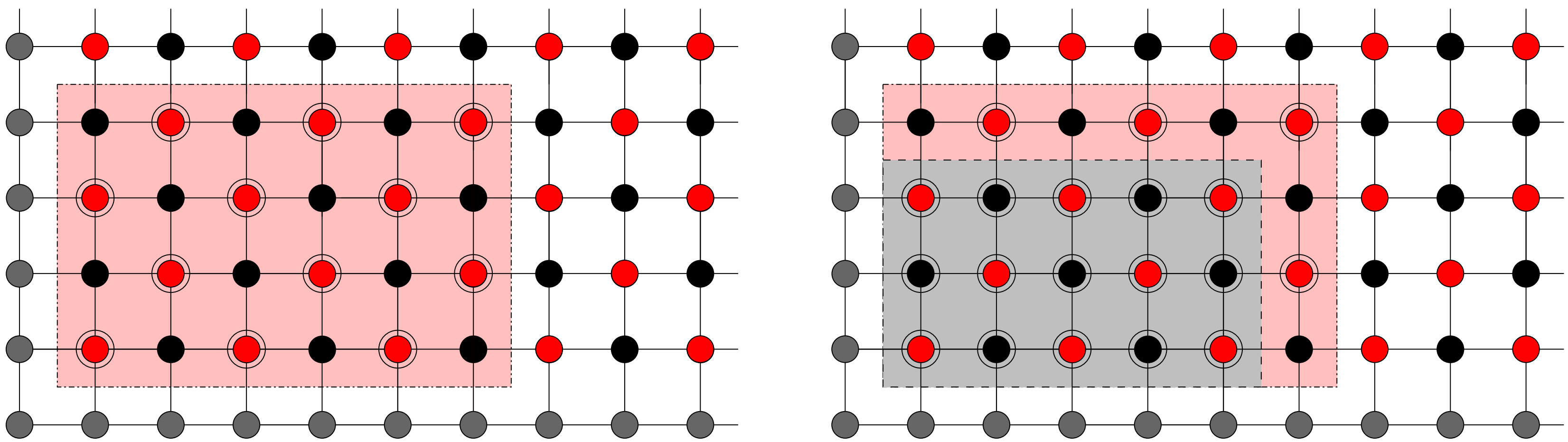
```
do KK= 1,N,W  // W = tile width
 do II= 1,N,H // H = tile height
  do J= 1,N
   do K= KK,min(KK+W-1,N)
    do I= II,min(II+H-1,N)
     C(I,J)= C(I,J)+A(I,K)*B(K,J)
    enddo
   enddo
  enddo
 enddo
enddo
```

Data access optimizations — loop blocking (2)

Blocking is also possible for iterative methods for linear systems

Blocking the iteration loop means merging successive iterations into a single pass through the data set $\Rightarrow$ Enhance cache reuse

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



Data dependencies need to be respected

Similar techniques for Jacobi's method

Data access optimizations — other techniques

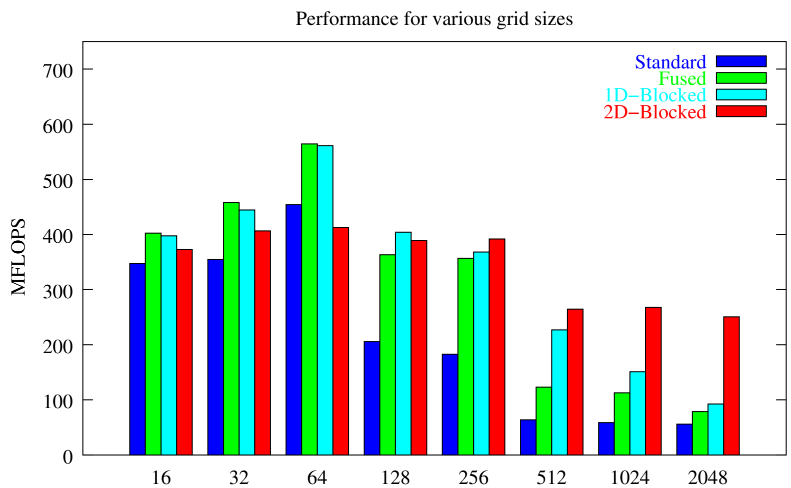
There is a variety of other data access optimizations

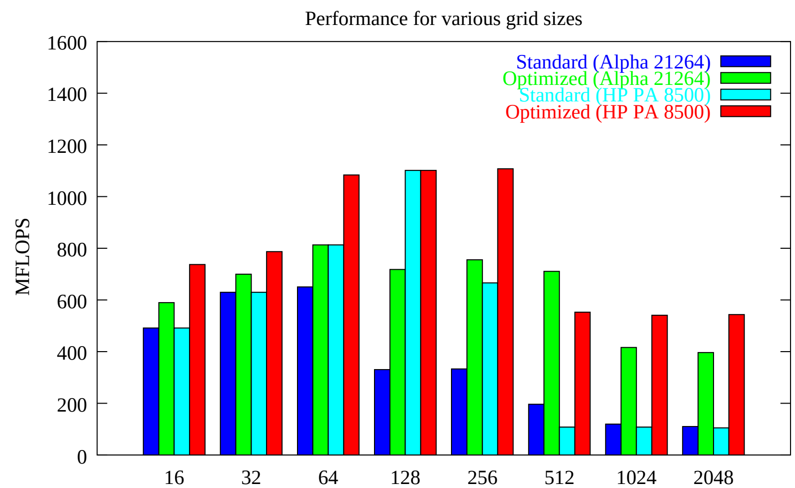
- *Loop interchange:* lessen the impact of non–unit stride accesses
- *Loop fusion:* reduce the number of sweeps through the data set $\Rightarrow$ Increase temporal locality
- *Data copying:* copy non–contiguous data to contiguous memory locations $\Rightarrow$ Reduce cache conflicts and/or drops in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

A few performance results

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

Speedups for 2D Gauss–Seidel smoother, constant coefficients
(left side: Alpha 21164, right side: Alpha 21264, HP PA 8500)





Variable–coefficient problems: speedup factors of 2–3 can be obtained for large grids, the MFLOPS rates are usually smaller since much more data have to be loaded

Current research efforts focus on the 3D case

Algorithmic locality, cache–oriented algorithms, etc.

Can we *re–design* numerical methods such that they *inherently respect* hierarchical memory (caches)? What are suitable *complexity models* for cache–based algorithms?

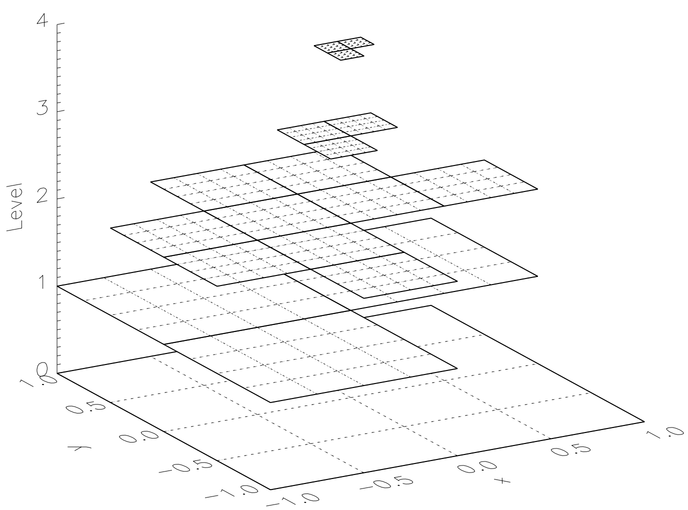
We are particularly interested in efficient algorithms for large linear systems; e.g.,

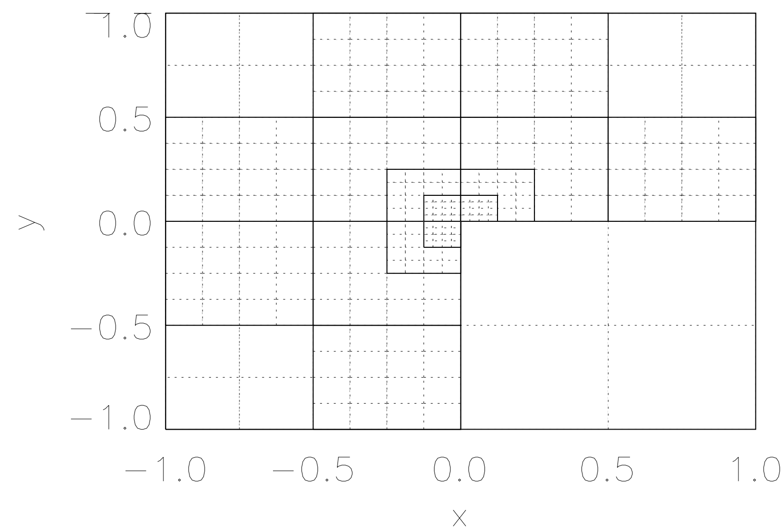
- Geometric MG
- AMG
- Hierarchical preconditioners
- Krylov subspace methods
- etc.

Algorithmic locality, cache–oriented algorithms, etc.

Possible approaches:

- Patch–adaptive grid structures *(structural adaptivity)*





- Adaptive relaxation, *Fully Adaptive Multigrid (FAMe)* (U. Rüde)
  Idea: Do computational work only where necessary *(runtime adaptivity)*
- Chaotic iterative schemes

Discussion and final remarks

We are currently combining

- Theoretical results from numerical analysis with
- Practical aspects of computer architecture

What can we learn from theoretical computer science?

Is there a cache-oblivious algorithm for computing a sparse matrix–vector product?

BTW: constant factors matter in numerically intensive codes! ;–)