Iterative Algorithms on High Performance Architectures

Ulrich Rüde

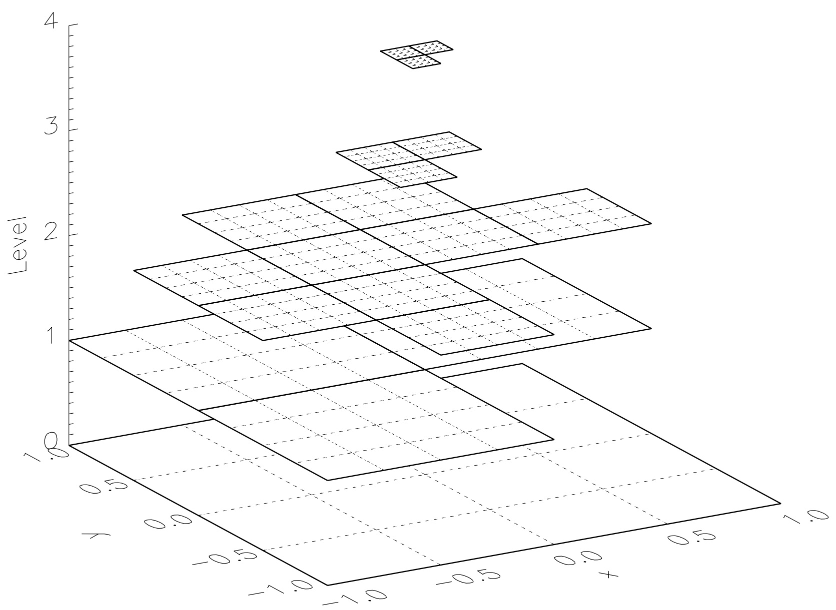
(joint work with H. Lötzbeyer and L. Stals)

Institut für Mathematik
Universität Augsburg

http://scicomp.math.uni-augsburg.de/ruede/me.html

DFG project: *Datenlokale Iterationsverfahren*

August 97



Outline

- the efficiency paradox
- PDE algorithms and data structures
- adaptive multilevel PDE methods
- what structures cost efficiency
- cache oriented iterative methods
- PAM: patch adaptive multigrid
- conclusions

Ideally ...

we can use

- multigrid with nested iteration to solve scalar elliptic PDE with approximately
  - 100 operations
  - storing 8 reals

  per unknown
- on a workstation
  - that can do 1×10^9 operations (= 1000 MFlop) per second
  - in 64×10^6 words (=512 MByte) of storage

and so, we can solve for

$$\boxed{10^7 \text{ unknowns in about } \boxed{\text{one}} \text{ second}}$$

.

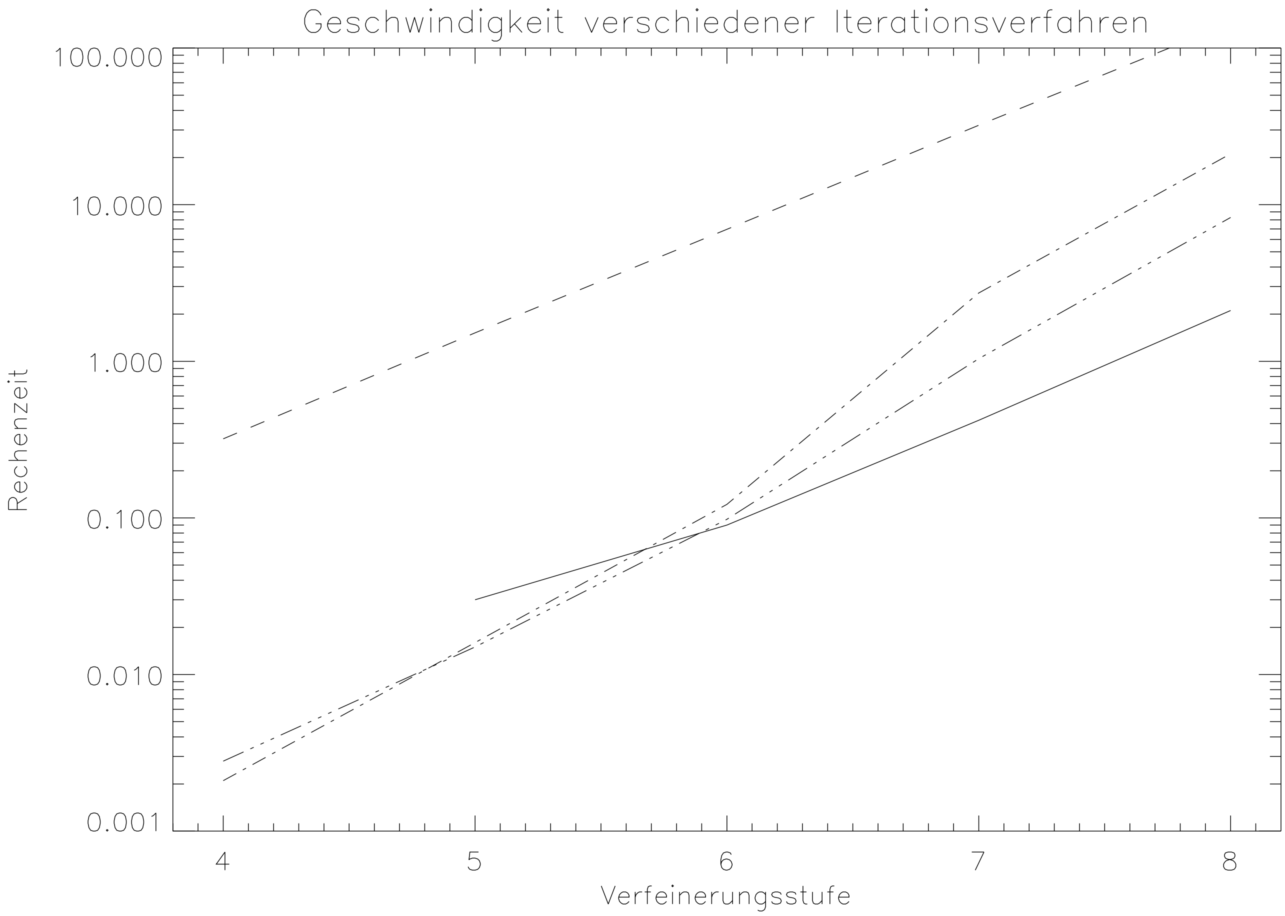
In practice ...

we are happy when our algorithm

- can do 10^5 unknowns
- on a (massively parallel) supercomputer
- where it doesn't run for hours

Run time comparison of iterative algorithms on uniform grids

- Standard Multigrid
- Adaptive Multigrid
- SOR
- SOR with cache optimization



Elliptic PDE

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) + \text{BC}$$

Given: An (infinitesimal) description of $\boxed{\text{local}}$ relations in $d = 2, 3$ dimensions.

Problem: Compute $\boxed{\text{global}}$ solution.

Step 1: $\boxed{\text{Discretization}}$ leads to a mesh (parameter h) and an algebraic system

$$A_h u_h = f_h$$

with $N = O(h^{-d})$ unknowns, but (locality!) only few entries per row.

Quality of discretization measured by truncation error: $O(h^{\{2,4\}})$ (if mesh is suitably designed) and problem not too bad.

Step 2: Solve algebraic system with $\boxed{\text{iterative methods}}$.

Quality of iterative solver: # Ops to reach approximation to *algebraic* solution better than truncation error.

Iterative Solvers

Form $x_{i+1} = \tilde{A}_h x_i$ for $i = 0, 1, 2, 3, \ldots, k(N)$ (e.g. relaxation). Execute this $k(N)$ times.

For $d = 2$ (fixed accuracy):

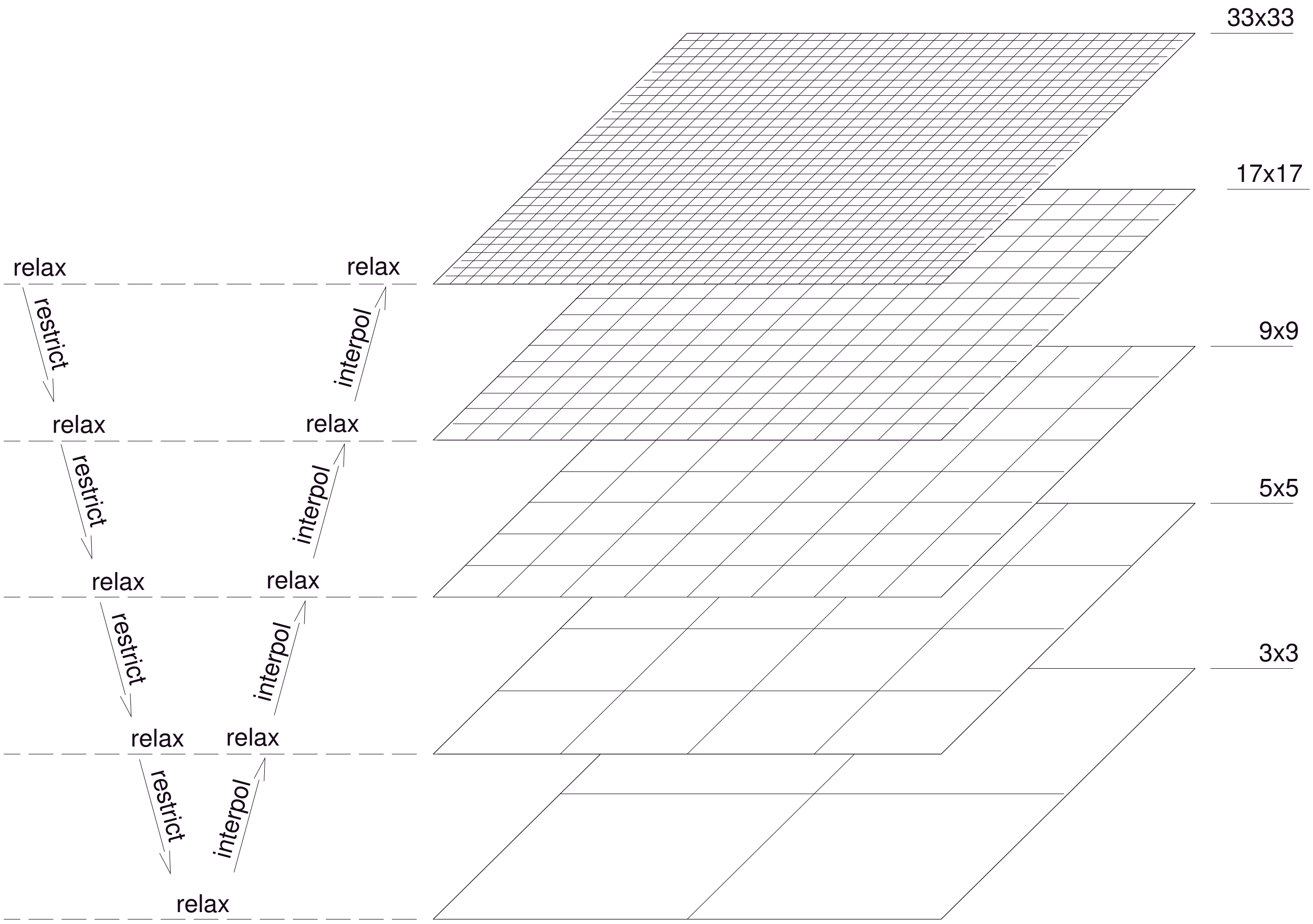
Name	$k(N)$	#Ops
Relaxation	$O(N)$	$O(N^2)$
SOR	$O(\sqrt{N})$	$O(N^{1.5})$
CG-type		$O(N^{1.5})$
SPAI	...	...
Chebychev	...	...
Multigrid	1	$O(N)$
...		

Multigrid

Multigrid and multigrid-like methods, like hierarchical or wavelet preconditioners use a stack of meshes to provide a more efficient data exchange between mesh subdomains.

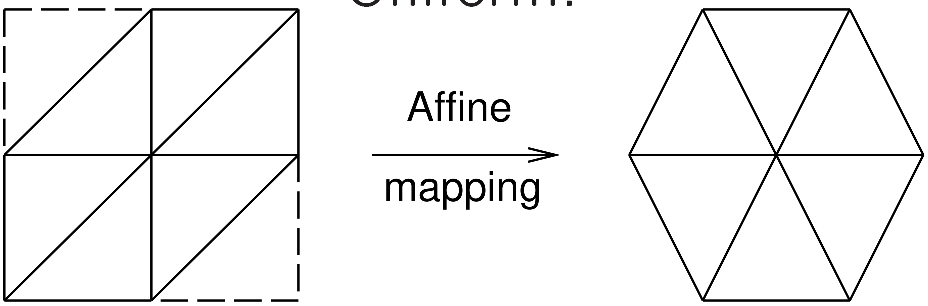
Multigrid pseudocode

Vcycle(k, ν_1, ν_2){
    Relax ν_1 times
    Compute residual
    Restrict residual
    Vcycle($k-1$, ν_1, ν_2)
    Interpolate correction
    Relax ν_2 times }

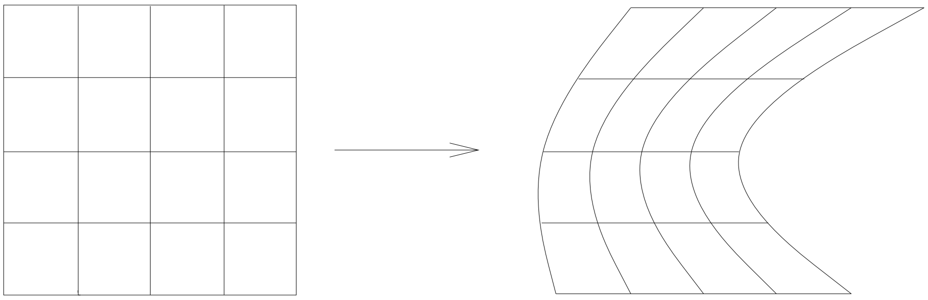


Mesh structures

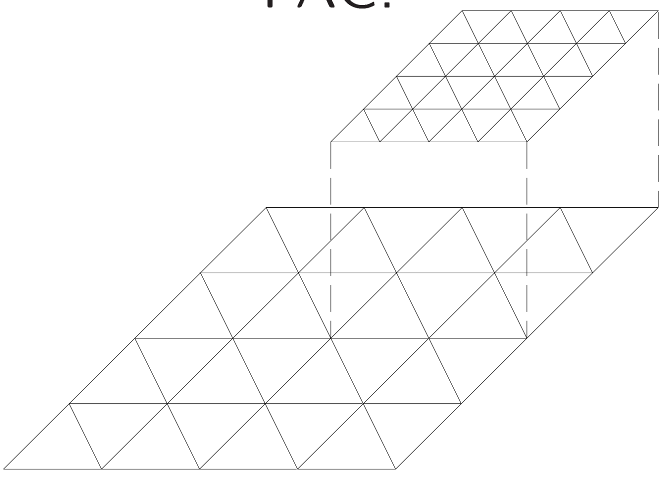
Uniform:



Quasi-uniform:

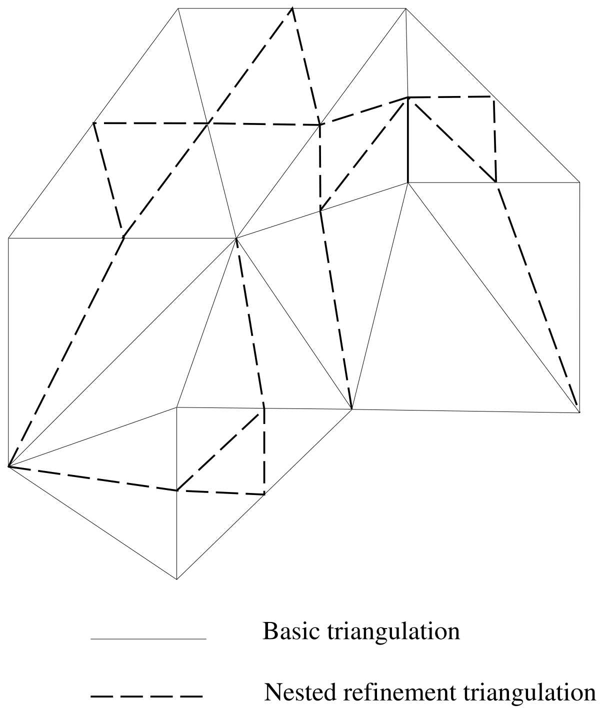


FAC:

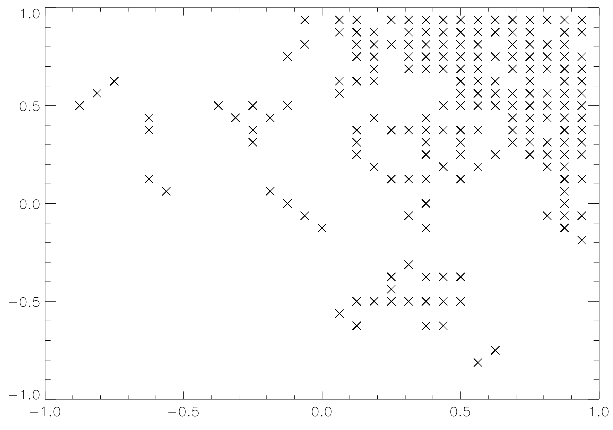


Mesh structures cont'd

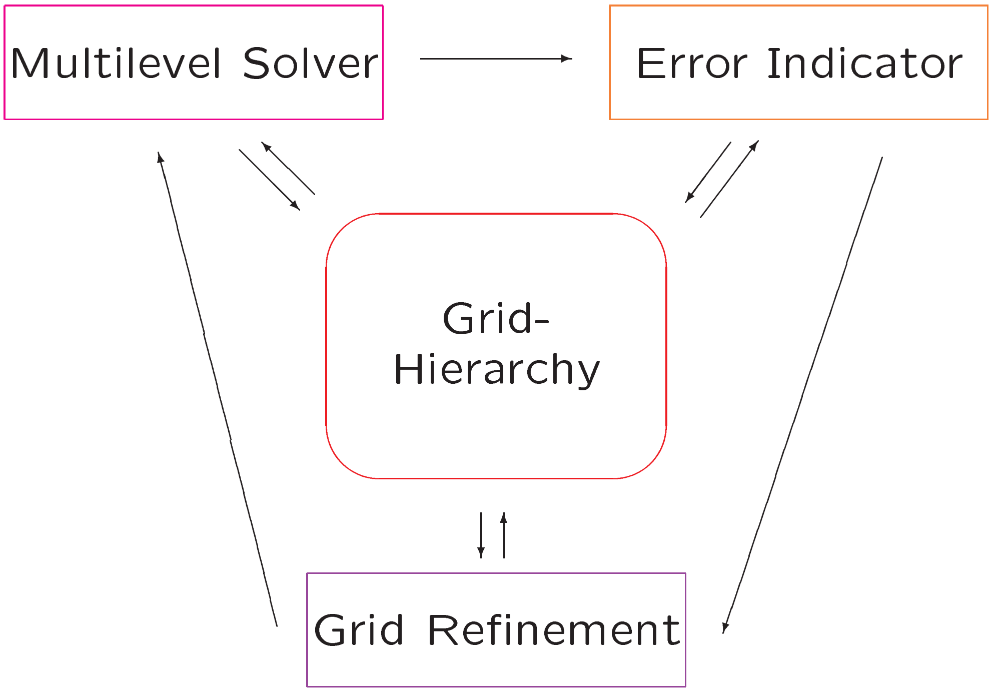
Unstructured:

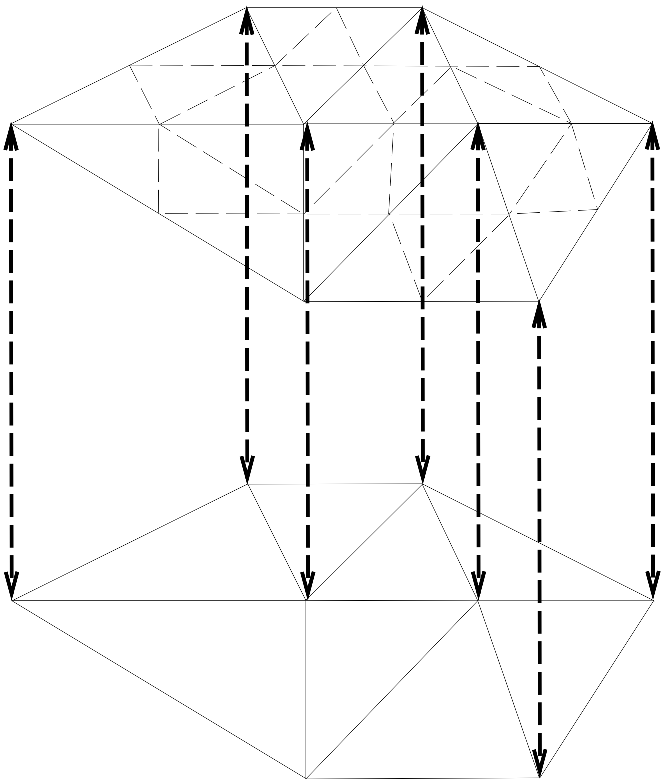


Virtual Global Grids:



Multilevel Adaptive Elliptic Solvers





Parallel Multigrid

- domain oriented mesh partitioning
- partitioning of coarse levels
- various strategies to to reduce number of communication steps
- additive vs. multiplicative algorithms
- various variants to avoid/reduce the idle processor problem on coarse grids (processor ABM).
- load balancing is usually more expensive than solving
- only hierarchical load balancing?

Instruction-level vs. multiprocessor parallelism

Szenario 1: Program A is parallelized on 10 processors giving supposedly reasonable 80 % efficiency: Of 10 min run time, 2 min are spent waiting for communication.

Szenario 2: Program A is tuned to exploit instruction level parallelism independently of coarse grain parallelism. This speeds up the program by say a factor of 8. Thus the efficiency of the parallel code drops to 33%: 1 min for computing, 2 min waiting for communication.

Good programming and a good program design is counter-productive to successful parallel processing.

What algorithms and data structures spoil performance?

Simple computational kernel: scalar product (BLAS `ddot`)

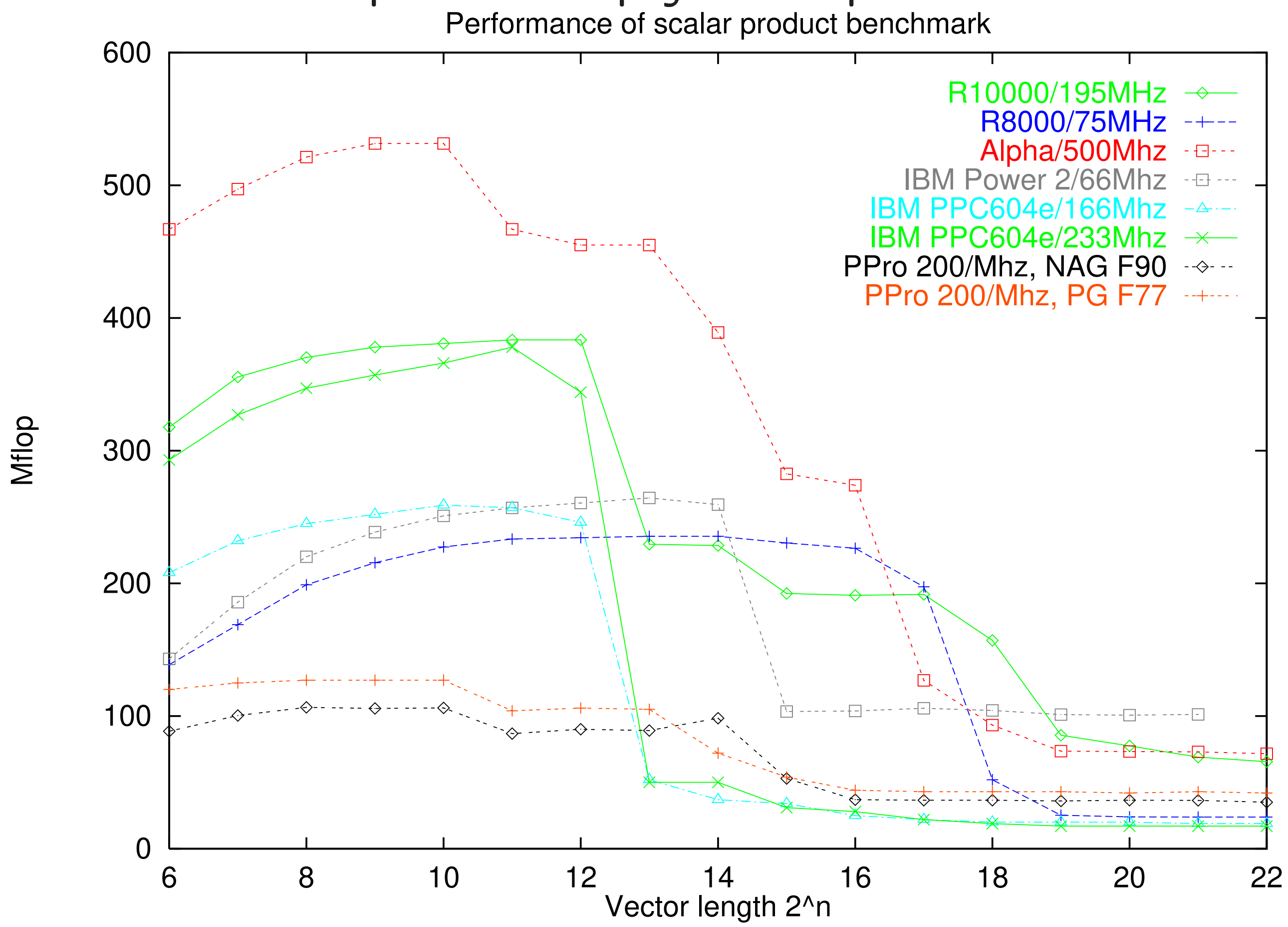
Unrolled dot product

```
sum1= 0.0d0
sum2= 0.0d0
sum3= 0.0d0
sum4= 0.0d0
do i=1,n,4
   sum1= sum1+ a(i)*b(i)
   sum2= sum2+ a(i+1)*b(i+1)
   sum3= sum3+ a(i+2)*b(i+2)
   sum4= sum4+ a(i+3)*b(i+3)
enddo
sum= sum1+sum2+sum3+sum4
```

This exploits superscalar and pipelined execution significantly better than the straightforward code.

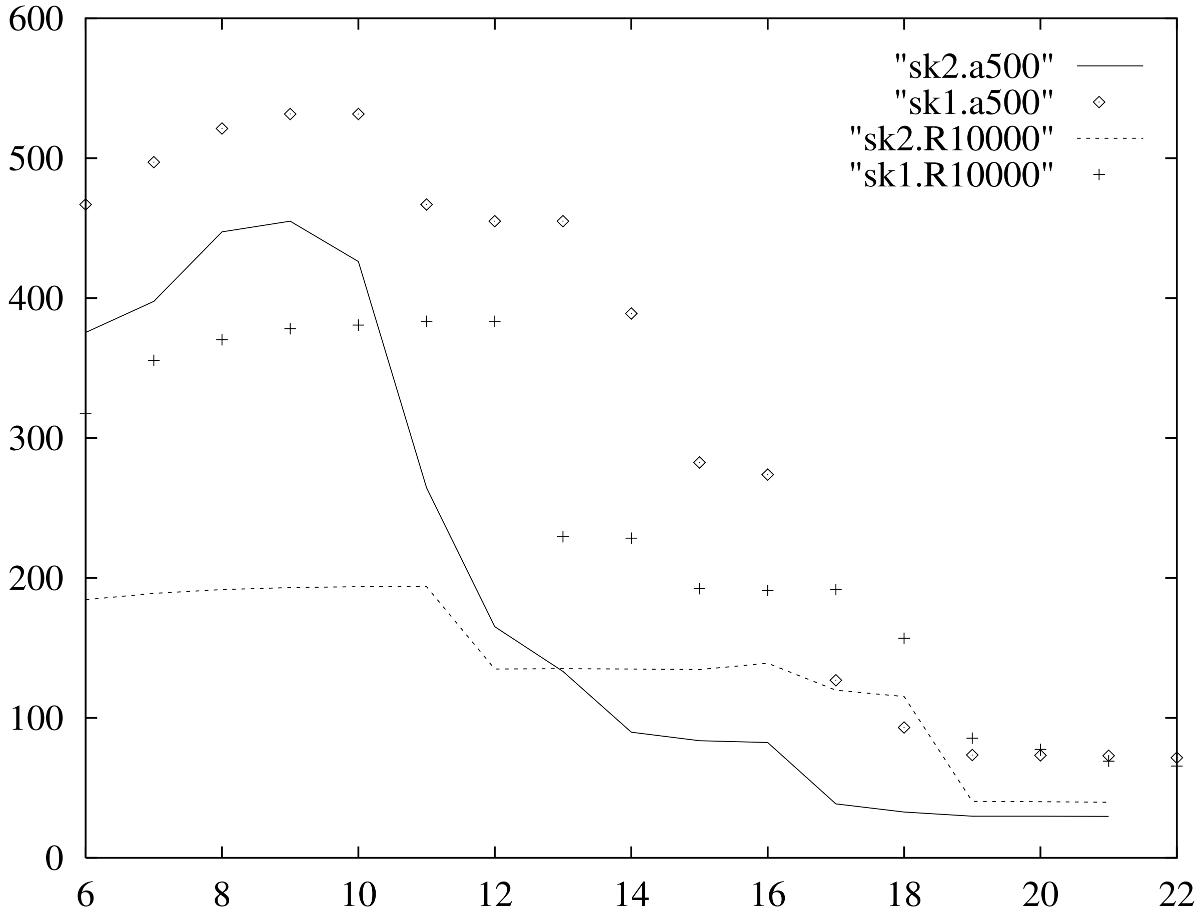
Performance depending on vector length

- $\sum_{1}^{n} a_i^2$ for $n = 64, 128, \ldots, 2^22$
- a $\equiv$ b
- Stride 1
- One read per Multiply-Add pair



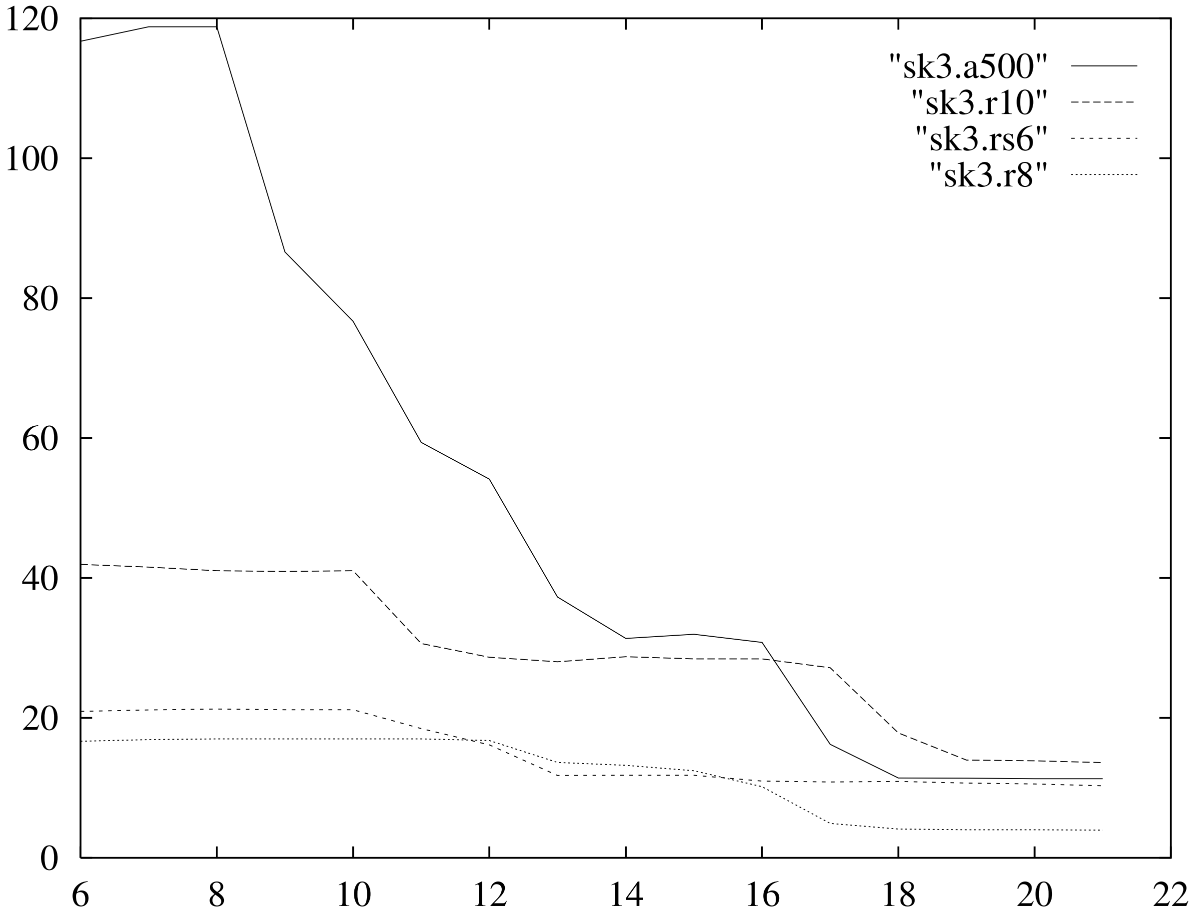
Performance depending on vector length

- $\sum_{1}^{n} a_i b_i$ for $n = 64, 128, \ldots, 2^2 2$
- a $\not\equiv$ b
- Stride 1
- two data sets, possible cache associativity conflicts
- two reads per Multiply-Add pair



Performance depending on vector length

- $\sum_{1}^{n} a_{j_i} b_{k_i}$ for $n = 64, 128, \ldots, 2^21$
- `a` $\not\equiv$ `b`, access simulating a linked list
- Indirect addressing for `a` and `b`
- two reads per Multiply-Add pair (plus index array)
- effective data layout uniform with stride 2



What does that mean?

What determines the run time of an algorithm?

− Processor (# operations)

+ Memory (data layout)
  + *temporal* locality of memory accesses
  + *spatial* locality of memory accesses

+ use of CPU-internal parallelism

We need performance models which help to *evaluate* the realistic performance of algorithms and *design* the algorithms in the correct way in the future.

Problem: Real behavior of modern computers is difficult to *understand* and describe analytically.

CPU (R)Evolution

- Workstations and
- parallel computers

both increasingly employ standard microprocessors. They have

- pipelined execution with 5-10 stages.
- superscalar execution with 2 or 4 floating point pipelines
- clock rates of up to 600 MHz
- peak speed of 1200 MFlop

The performance increase is approx. 80 % per year.

Thus any modern CPU is a $\boxed{\text{Parallel Processor}}$ in itself, employing typically something like 20-40 parallel units (and more in the future).

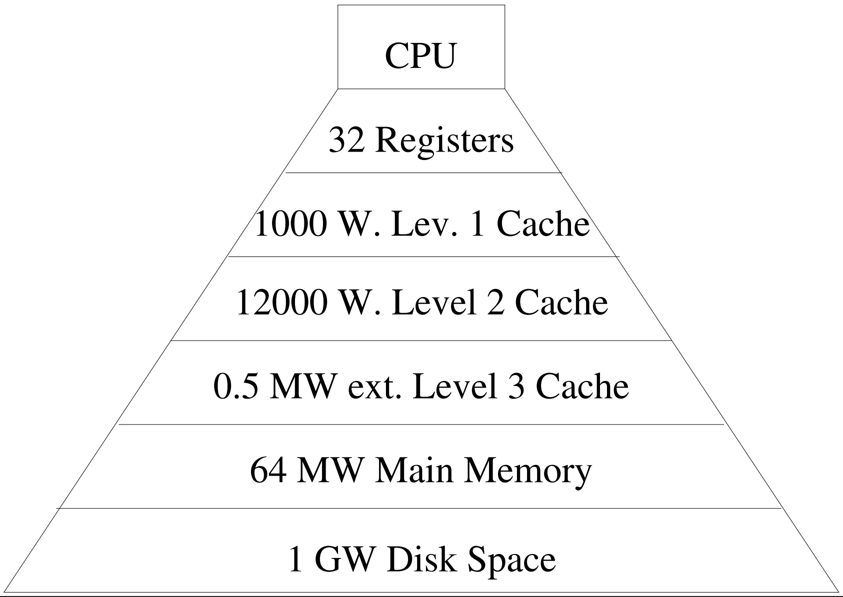
Memory

- DRAM still has access times of only 50 ns.
- access times decrease, but only at about 5% per annum.
- additional overhead for memory access on the order of 100ns.
- access to virtual shared memory is an additional order of magnitude slower.

Thus all systems have

- memory speed $\ll$ processor speed
- a hierarchical memory system with up to 3 levels of caches with successively longer latencies and lower bandwith

Memory Hierarchy (DEC PWS 600)



Level	Capac.	(MB/s)	Latency
FP Register	256 B	28,800	1.7 ns
Cache 1	8 KB	19,200	1.7 ns
Cache 2	96 KB	9,600	5.0 ns
Cache 3	2 MB	873	23.3 ns
Main Mem	1,536 MB	1,070	105.0 ns

SGI Origin 2000

Level	Capac.	(MB/s)	Latency
FP Register	512 B	?	5 ns
Cache 1	32 KB	3,120	10 ns
Cache 2	4 MB	2,080	41 ns
Main Mem	4,000 MB	780	313 ns

Backus (1977):

I propose to call this tube the

Von-Neumann bottleneck.

What are the consequences?

To avoid inefficiency we must:

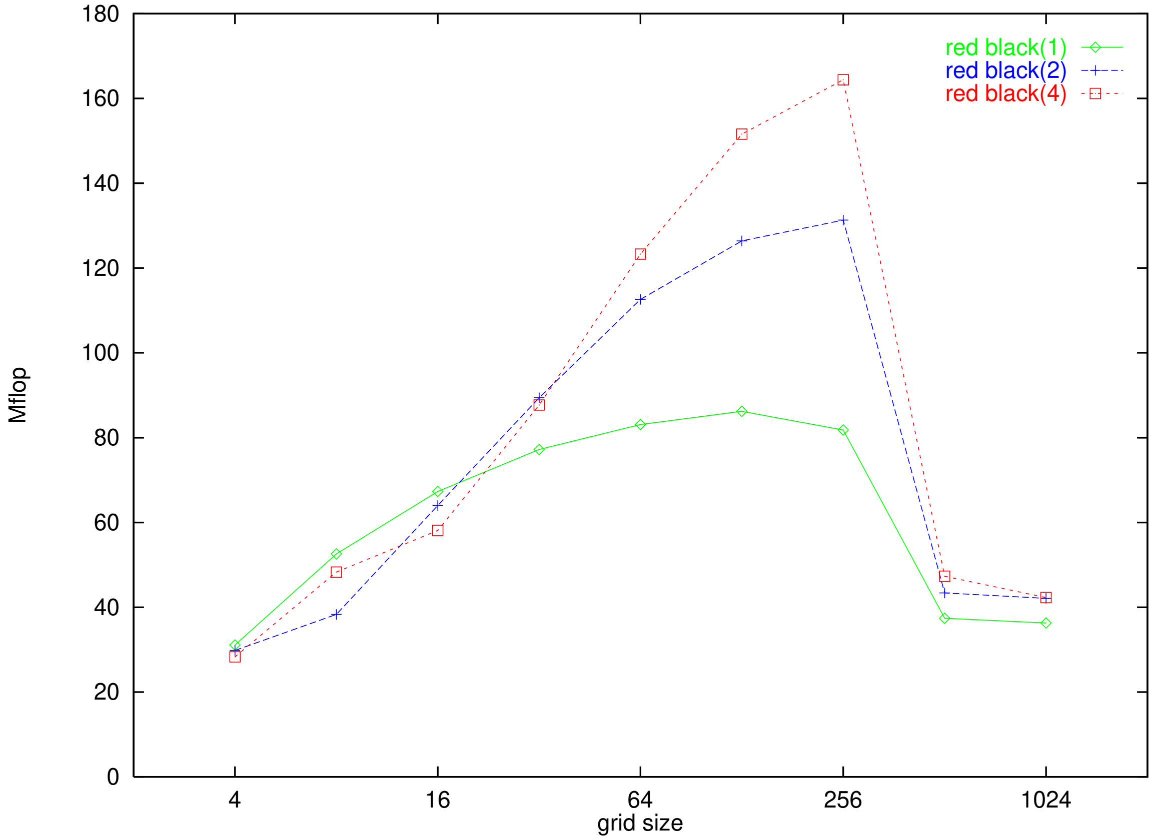
Avoid dynamic structures. No linked lists, binary trees, etc. on too low granularity. How to implement sparse matrices then? We don't.

Exploit instruction-level parallelism. Prepare the codes such that automatic restructuring tools and compilers (optimizers) can extract the parallelism. F90 and HPF array syntax are counter-productive, since we also need to

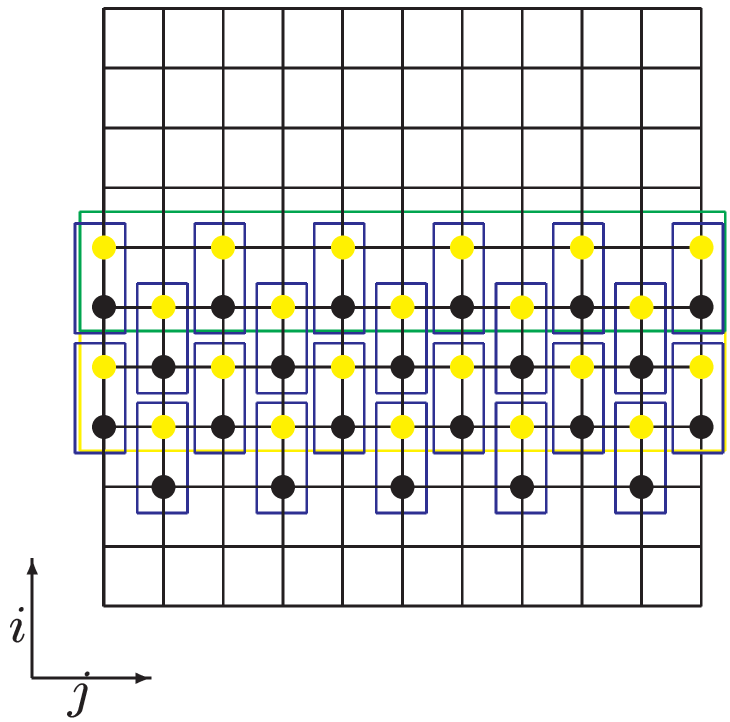
Exploit data locality. Do not program in global sweeps! We cannot save in forming Ax, but we can save when $Ax, A^2x, A^3x, \ldots$ are needed. This is awkward programming and in the future we will need tools for this job!

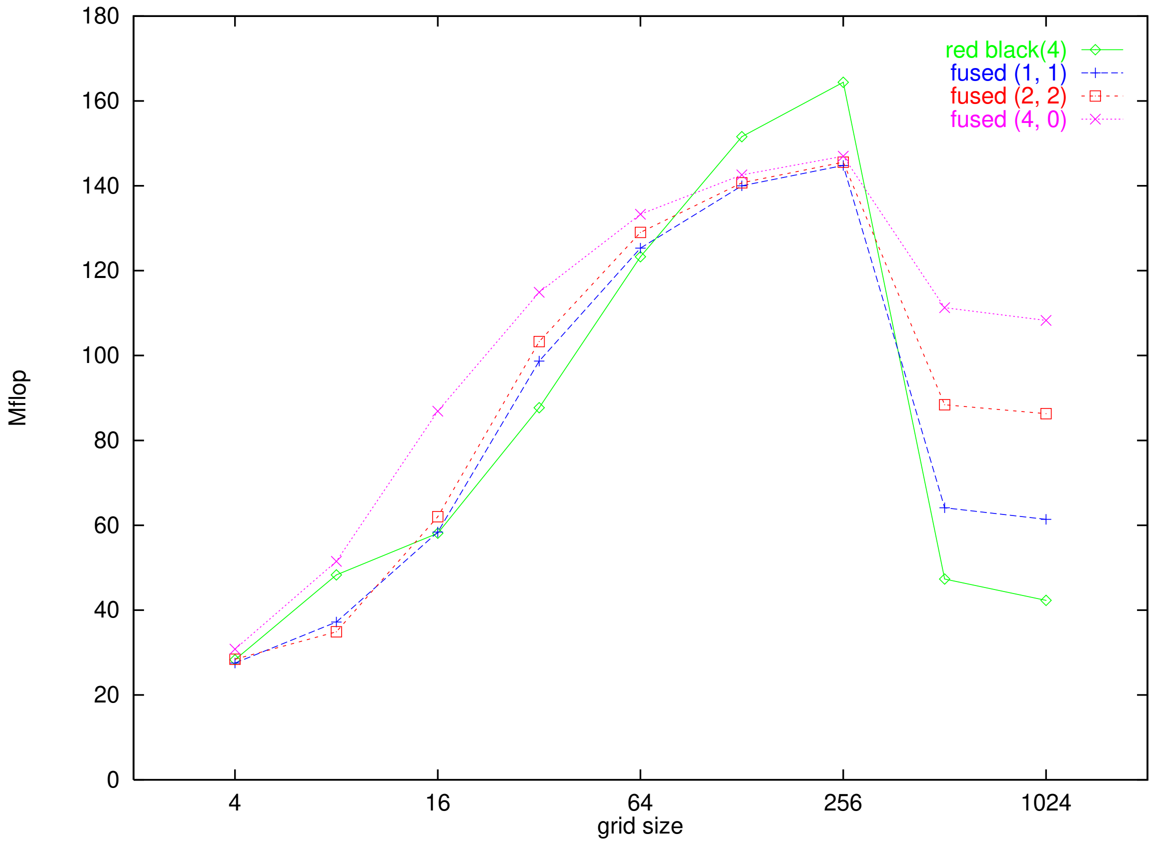
Red-Black Relaxation (square grid)

```
do j=1,n-1
do i=2-j%2,n-1,2 (Red nodes)
    Relax( u(i,j) )
enddo
enddo
do j=1,n-1
do i=1+j%2,n-1,2 (Black nodes)
    Relax( u(i,j) )
enddo
enddo
```



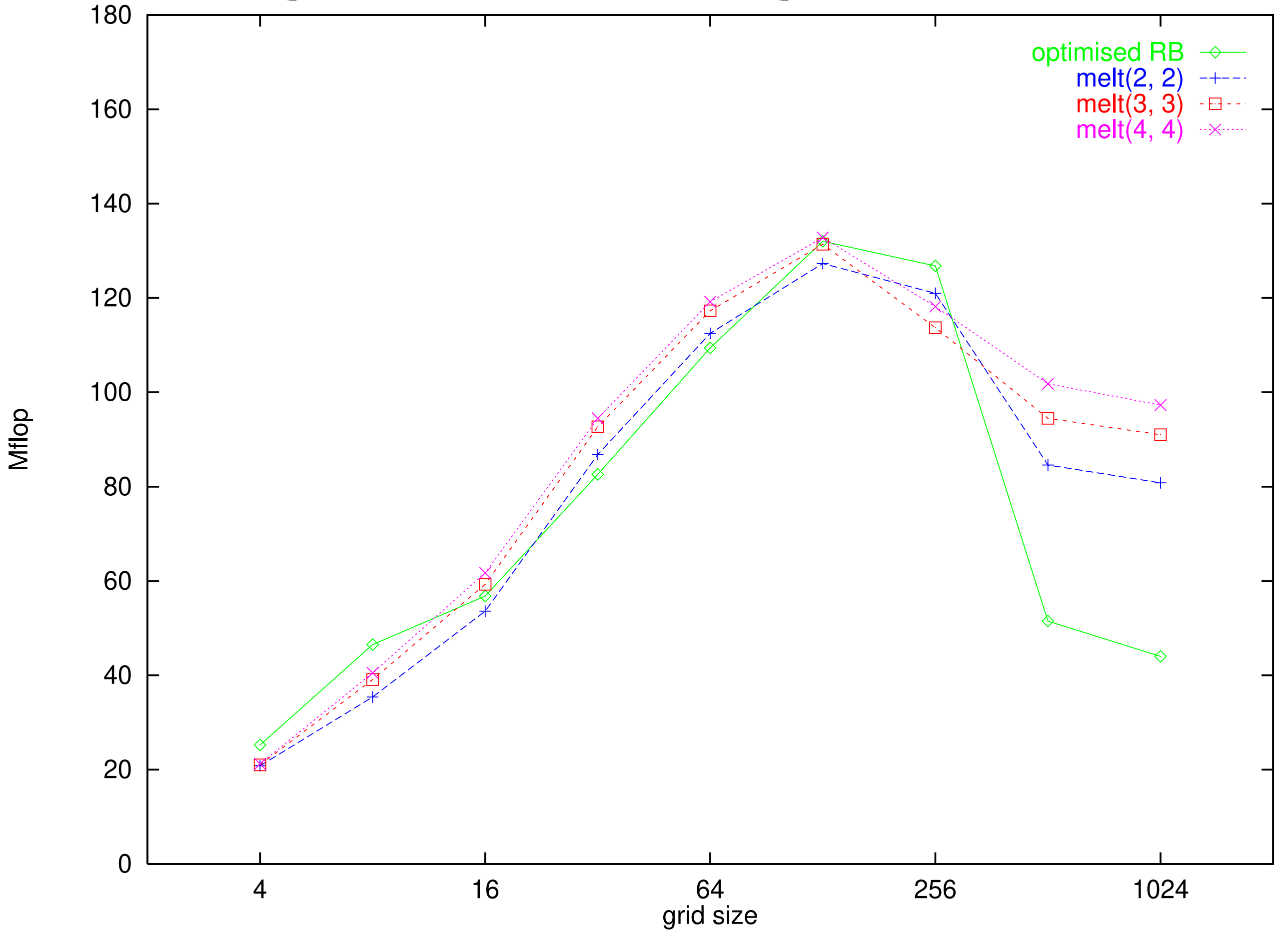
Cache aware Red-Black Relaxation





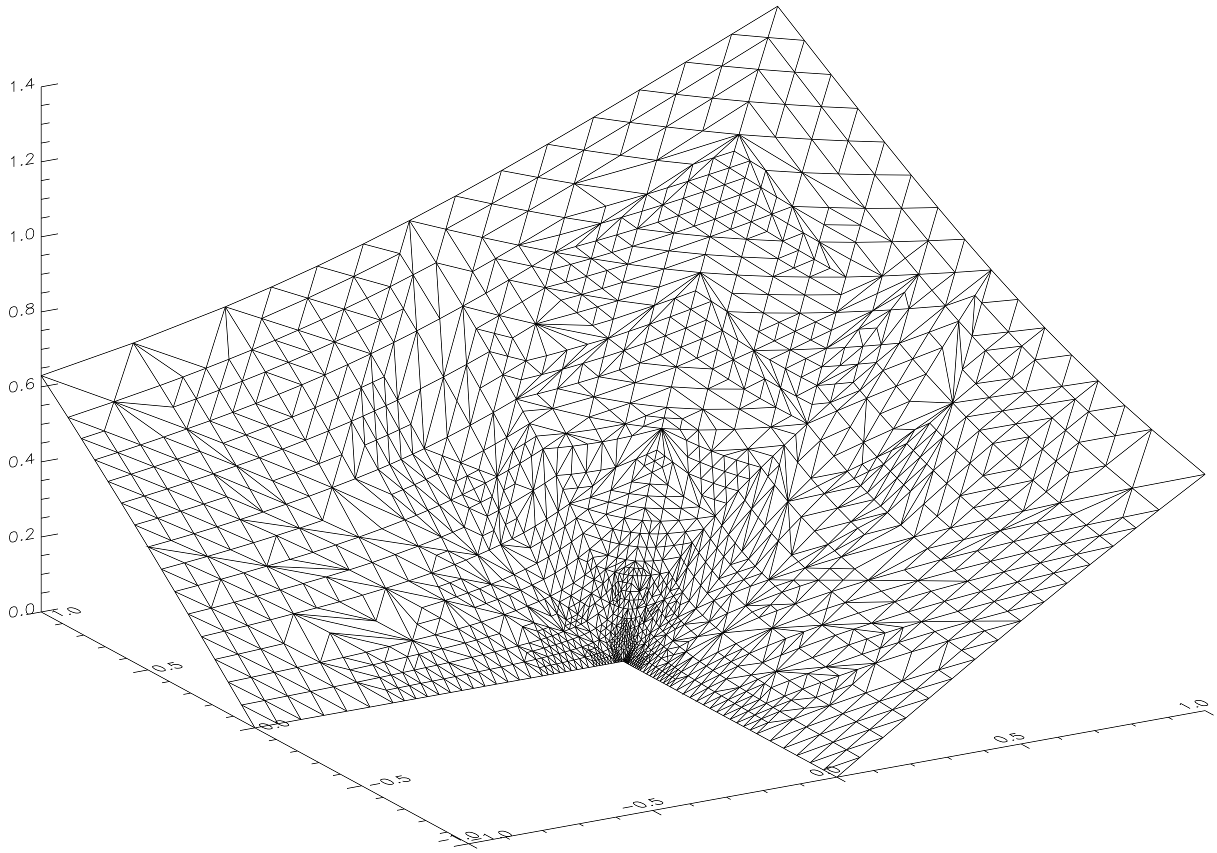
Cache aware multigrid

Fusing pre- and coarse grid operations



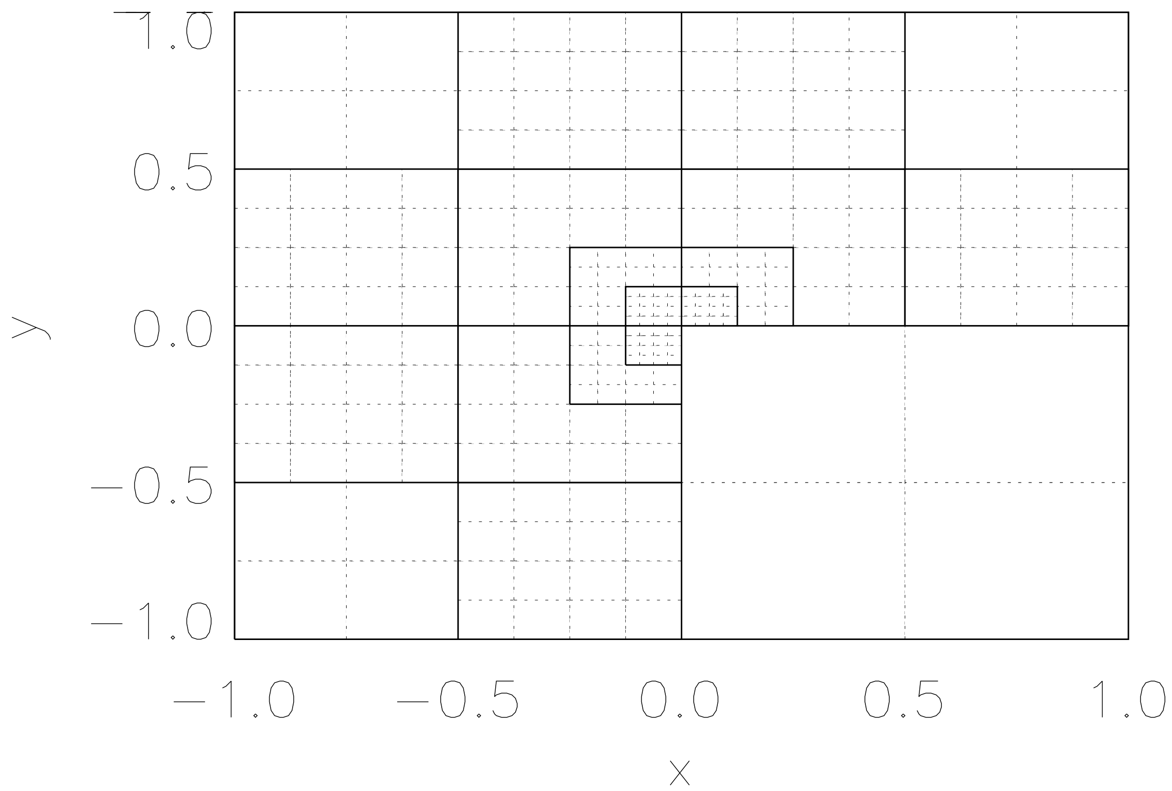
Alternatives to using unstructured grids

$$\begin{aligned} \Delta u &= 0 \text{ in } \Omega \\ u &= g(x,y) \text{ on } \partial\Omega \end{aligned}$$

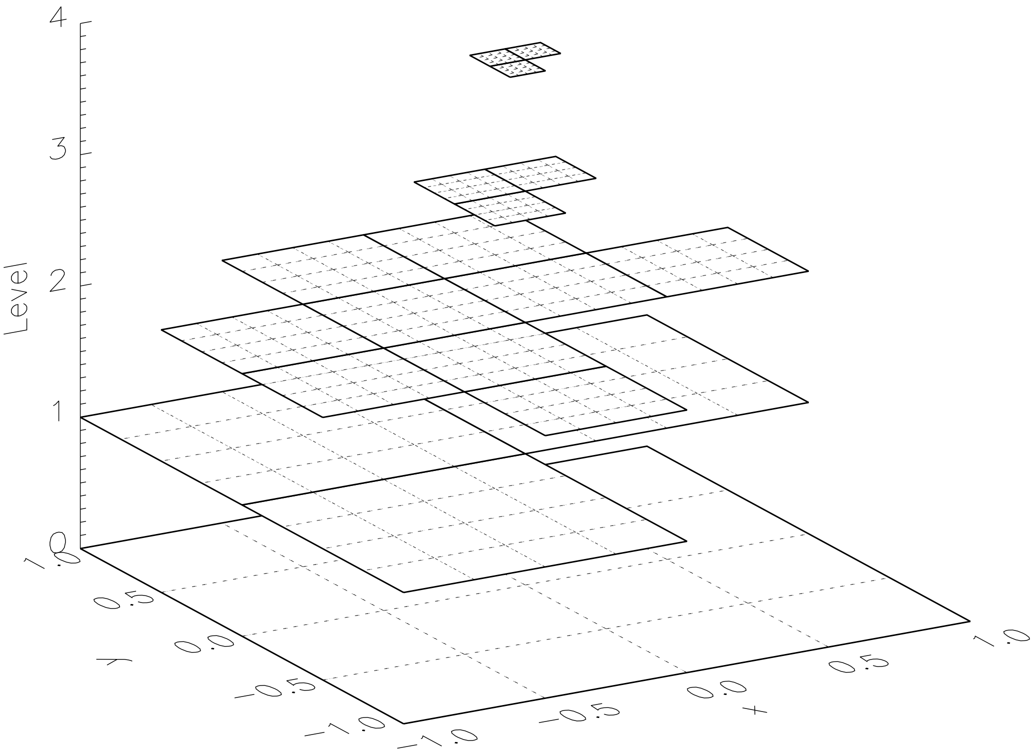


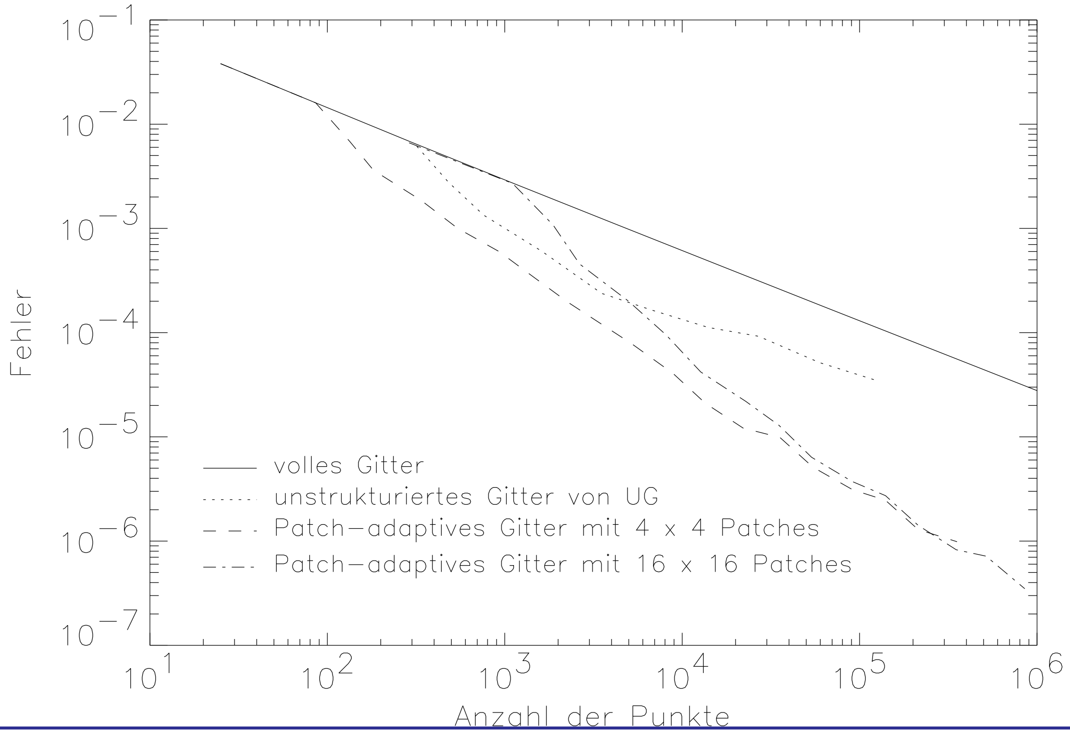
PAM - Patch Adaptive Multigrid

- nodes are grouped in (non-overlapping) patches of fixed size
- each level consists of a collection of patches
- patches may be present (live) or
- patches may be virtual (ghosts)



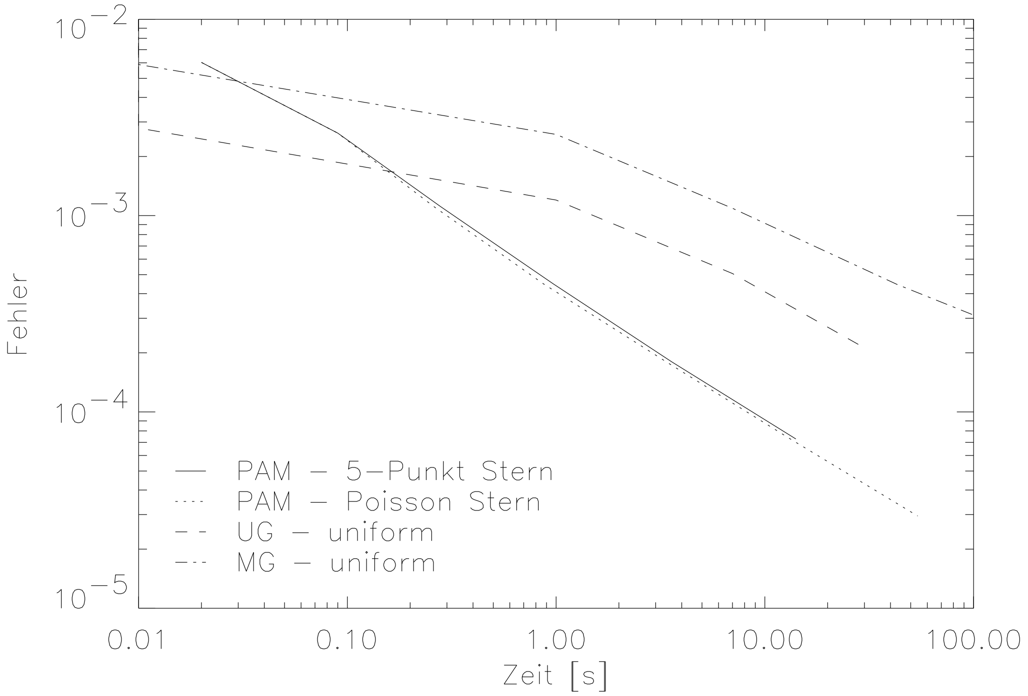
PAM (cont'd)

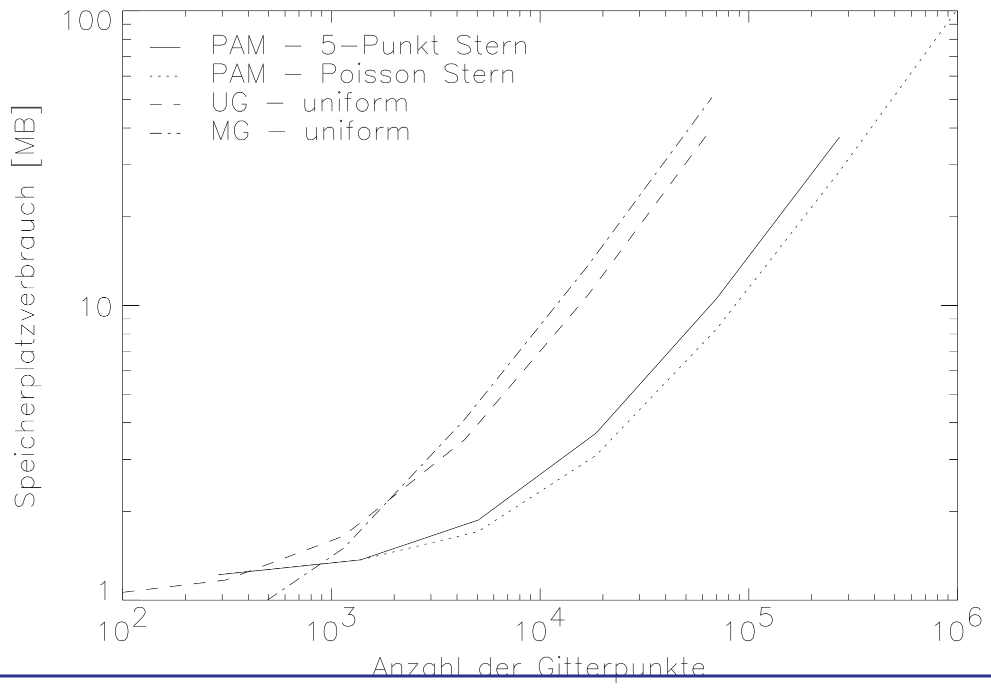

Level
x
y


Fehler
Anzahl der Punkte
volles Gitter
unstrukturiertes Gitter von UG
Patch–adaptives Gitter mit 4 x 4 Patches
Patch–adaptives Gitter mit 16 x 16 Patches

Comparison in Speed and Storage

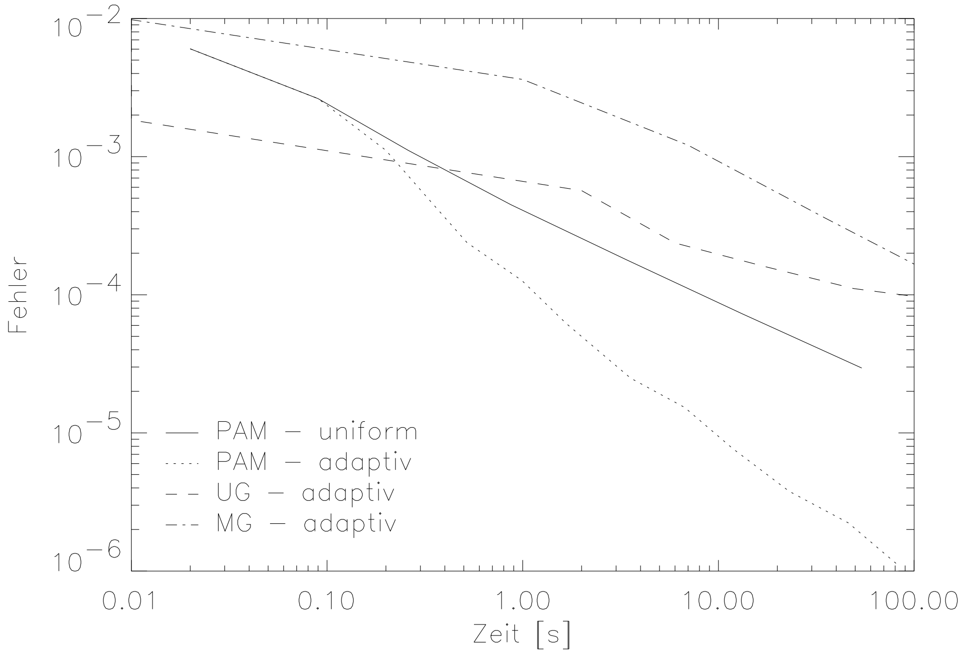
Uniform grids

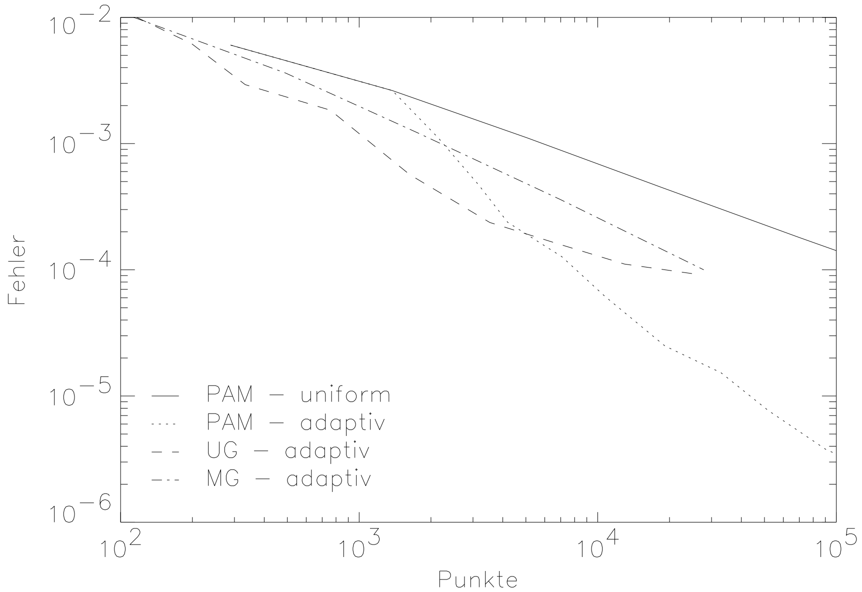




Comparison in Speed and Storage

Adaptive grids





Exploiting Caches

Efficiency of Patch-Smoother

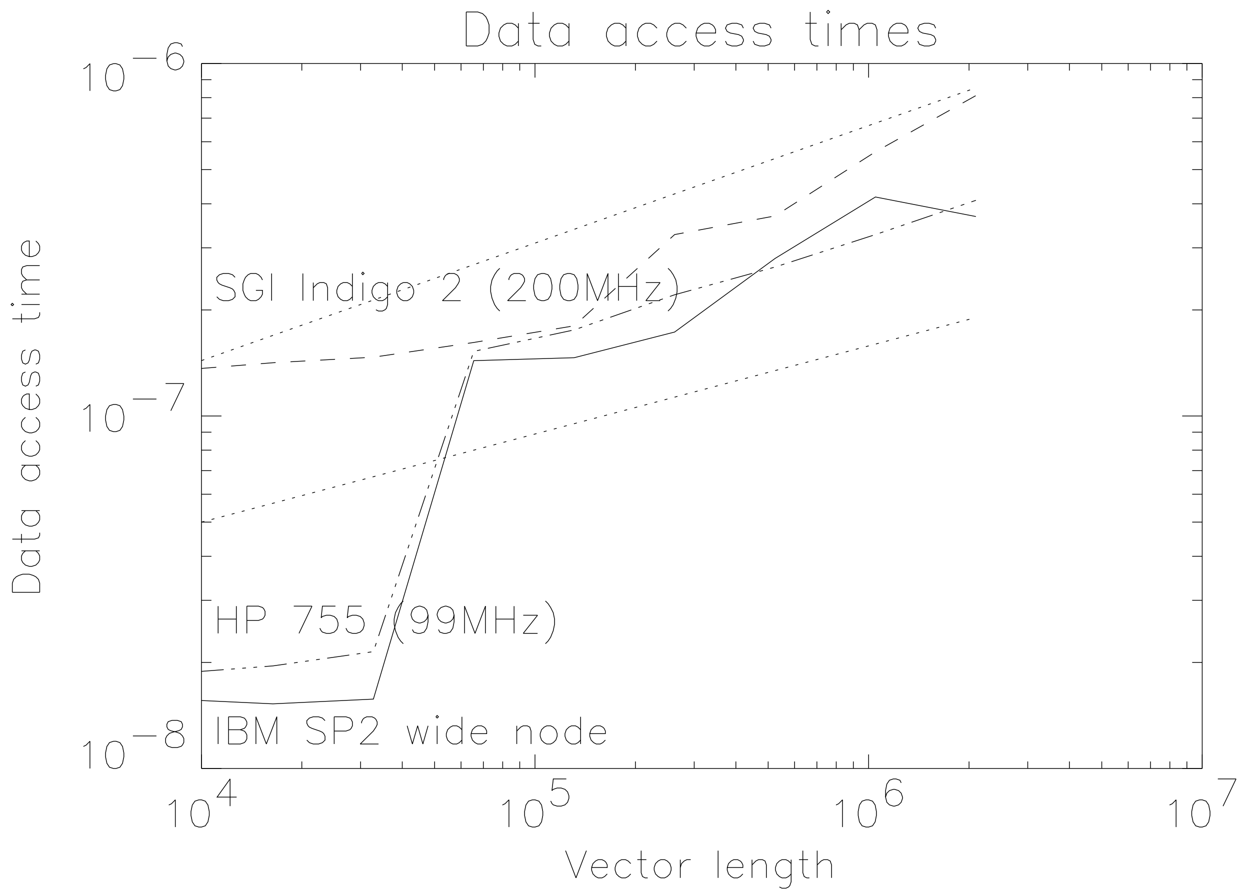
steps	time [sec]	speed (nodes/sec)
1	0.40	657 922
2	0.48	1 096 537
3	0.59	1 338 147
4	0.68	1 548 052
5	0.78	1 686 980
6	0.90	1 754 460

Multiple smoothing steps

	global		patch-wise	
Steps	[sec]	L_2-err	[sec]	L_2-err
1	0.75	$1.47\ 10^{-3}$	0.75	$1.47\ 10^{-3}$
2	1.61	$1.02\ 10^{-3}$	1.04	$1.02\ 10^{-3}$
3	2.35	$8.27\ 10^{-4}$	1.26	$8.28\ 10^{-4}$
4	3.28	$7.10\ 10^{-4}$	1.48	$7.22\ 10^{-4}$
5	4.16	$6.31\ 10^{-4}$	1.64	$6.57\ 10^{-4}$

Application oriented modeling of memory performance

Data access cost



The measurements suggest that the access cost is a function of data set size,

$$C(n) \approx \begin{cases} O(n^{(1/3)}) \\ O(n^{(1/4)}) \end{cases}$$

Performance Model

Only *temporal* locality

Simplified assumption: The cost to access address a is a function of the time, when a has been accessed last. So, when a program accesses a sequence of memory references

$$a_1, a_2, a_3, \ldots$$

at times $i = 1, 2, \ldots,$ the cost of the access at time k is

$$\tilde{C}(a_k) = C(L(a_k)),$$

where the *last access* $L(a_k)$ is defined as

$$L(a_k) = \begin{cases} \min\limits_{a_j = a_k, j<k} k - j & \text{if defined} \\ L_{\max} & \text{otherwise} \end{cases}$$

and where $C(n) \approx n^{\alpha}$, $\alpha = 1/3, 1/4$.

Performance Analysis for Relaxation Methods

```
do it=1,N
do i=1,N-1
do j=1,N-1
  y= 0.25 * (u(j,i-1)+u(j-1,i)+
                 u(j,i+1)+u(j+1,i))
  x= y - u(j,i)
  u(j,i)= u (j,i) + w*x
enddo
enddo
enddo
```

Cost in each pass (except the first)

Datum x	$L(x)$	$C(L(x))$
`u(j,i+1)`	$O(N^2)$	$O(N^{2\alpha})$
`u(j+1,i)`	$O(N)$	$O(N^\alpha)$
`u(j,i)`	$O(1)$	$O(1)$
`u(j,i-1)`	$O(1)$	$O(1)$
`u(j-1,i)`	$O(N)$	$O(N^\alpha)$

Performance Analysis for Relaxation Methods

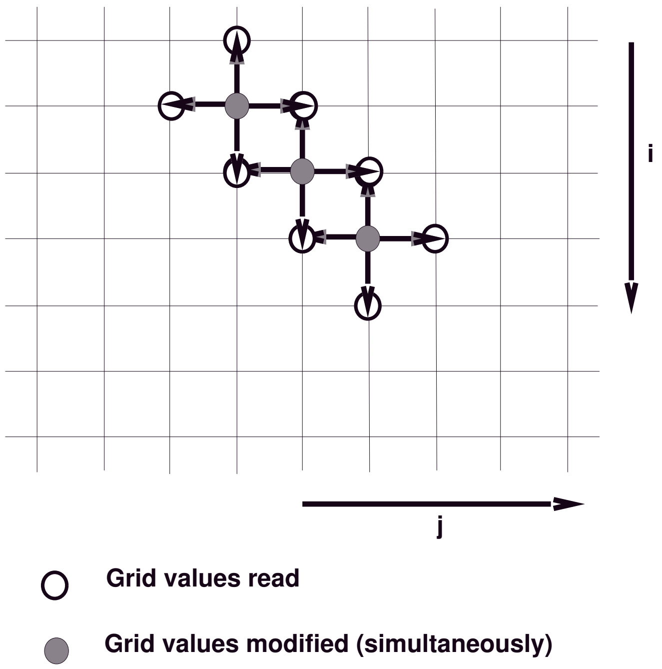
Cost of one full sweep:

$$O(N^2 * (C(N^2) + 2C(N) + 2C(1)) = O(N^{2+2\alpha})$$

With Overrelaxation $O(N)$ sweeps are necessary, so the cost of the solution is

$$O(N) * O(N^{2+2\alpha}) = O(N^{3+2\alpha})$$

Pipelined Iterations (Fusing iterations)



Cost of k iterations in one pipeline

$$N^2(C(N^2) + (k+1)C(N) + 2kC(1))$$

Cost of N/k such sweeps performing k iterations:

$$O\left(\frac{N^3}{k}N^{2\alpha} + N^3\frac{k+1}{k}N^{\alpha} + 2N^3\right)$$

With $k = N^\alpha$, the total cost is $O(N^{3+\alpha})$.

Summary and Conclusions

- structured vs. unstructured
- adaptivity with uniform patches
- current architectures require new style of programming
- and new algorithms
- tools
- basic performance models