Cache Optimization Techniques for Numerical Computations*

Markus Kowarschik, Christian Weiß,
Jonathan Hu, Craig C. Douglas,
Ulrich Rüde, Wolfgang Karl

Universität Erlangen-Nürnberg
Technische Universität München
University of Kentucky

SPEC Benchmarking Workshop,
Wuppertal, Paderborn, September 1999

*Partially supported by DFG (project Ru 422/7-1), NSF (grants DMS-9707040 and ACR-9721388), and NATO (grant CRG 971574)

Outline

1. Introduction

2. Optimization techniques for iterative methods in 2D and 3D

3. Closing remarks

4. Future work

Introduction: memory hierarchies

Theoretically ...

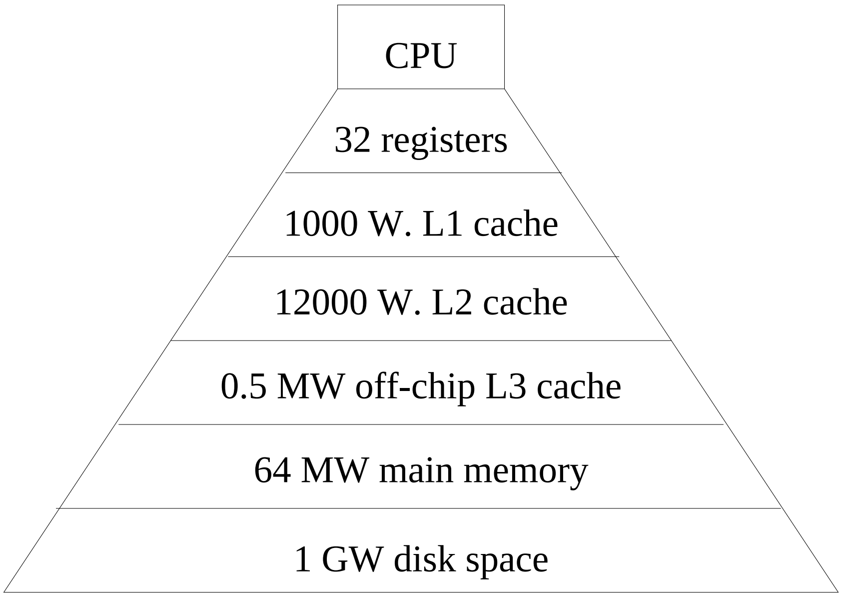
Modern workstations based on superscalar RISC processors can do > 1000 MFlops/sec

But in practice ...

Main memory performance often deteriorates the performance of the code dramatically

Remedy: memory hierarchies

Example: DEC Alpha PWS 500au (A21164):



Level	Capacity	Throughput	Latency
Register	512 Byte	24 GByte/s	2 ns
L1 Cache	8 kByte	16 GByte/s	2 ns
L2 Cache	96 kByte	8 GByte/s	6 ns
L3 Cache	4 MByte	888 MByte/s	24 ns
Main Mem	512 MByte	1 GByte/s	112 ns

Introduction: cache optimizations

$\boxed{\text{Basic idea:}}$
Use the memory hierarchy as efficiently as possible by applying $\boxed{\text{data locality optimizations}}$

- Compilers do not (yet?) optimize well in most of the practically relevant cases
- Therefore, codes have to be tuned manually

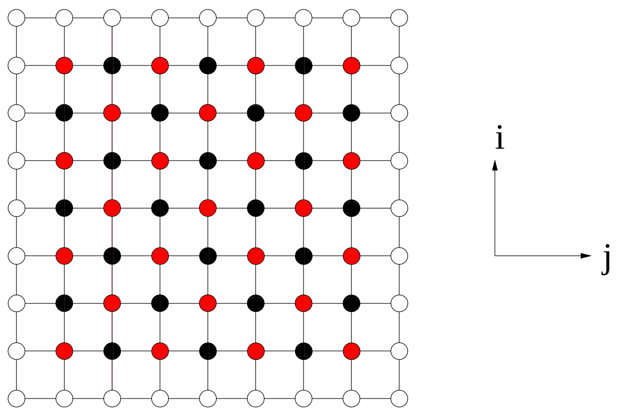
$\boxed{\text{The interests of our group:}}$
Cache optimizations for iterative methods (especially multigrid (MG) algorithms) for solving partial differential equations (PDEs)

$\boxed{\text{Related work:}}$

- D. Quinlan (Lawrence Livermore), F. Bassetti (Los Alamos), et al.: cache–optimized OO framework for numerical computations
- D.E. Keyes, et al. (Old Dominion, ICASE): cache–friendly PDE solvers (particularly for parallel CFD applications)
- C.–W. Tseng, et al. (Maryland): compiler–based data locality optimizations
- M.E. Wolf, M.S. Lam, et al. (Stanford): compiler–based data locality optimizations (*SUIF* project), modeling of memory accesses
- ...

Introduction: red/black Gauss–Seidel

Example: red/black Gauss–Seidel (or SOR–) iteration in 2D on a DEC Alpha PWS 500au:




```
double u(0:n,0:n), f(0:n,0:n)

do it = 1, iter
   // red grid points:
   do i = 1, n-1
     do j = 1+(i+1)%2, n-1, 2
       u(i,j)=f(i,j)+ce*u(i,j)+ea*u(i+1,j)+
                we(i-1,j)+no*u(i,j+1)+so*u(i,j-1)
     enddo
   enddo
   // black grid points:
   do i = 1, n-1
     do j = 1+i%2, n-1, 2
       u(i,j)=f(i,j)+ce*u(i,j)+ea*u(i+1,j)+
                we(i-1,j)+no*u(i,j+1)+so*u(i,j-1)
     enddo
   enddo
enddo
```

Introduction: code profiling

Analysis of this Gauss–Seidel code using the *Digital Continuous Profiling Infrastructure (DCPI)* on a DEC PWS 500au:

- Processor cycles:

Grid Size	MFLOPS	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
16	347.0	60.7	0.3	2.6	6.7	21.1	4.5
32	354.8	59.1	10.9	7.0	4.6	11.0	5.4
64	453.9	78.8	1.4	15.7	0.1	0.0	4.2
128	205.5	43.8	6.3	47.5	0.0	0.0	2.4
256	182.9	31.9	60.6	4.2	0.0	0.0	3.3
512	63.7	11.3	85.2	2.2	0.0	0.0	1.2
1024	58.8	10.5	85.9	2.4	0.0	0.0	1.1
2048	55.9	10.1	86.5	2.4	0.0	0.0	1.1

- Memory accesses:

Grid Size	Data Set Size	% of all accesses which are satisfied by				
		$\pm$	L1 Cache	L2 Cache	L3 Cache	Memory
32	17 kByte	4.5	63.6	32.0	0.0	0.0
64	66 kByte	0.5	75.7	23.6	0.2	0.0
128	260 kByte	-0.2	76.1	9.3	14.8	0.0
256	1 MByte	5.3	55.1	25.0	14.5	0.0
512	4 MByte	4.9	29.9	50.7	7.3	7.2
1024	16 MByte	5.1	27.8	50.0	9.9	7.2
2048	64 MByte	4.5	30.3	45.0	13.0	7.2

Introduction: model problems

$$\Delta u = f \text{ in } \Omega = (0,1)^2 \text{ or } (0,1)^3,$$
$$u = g \text{ on } \partial\Omega$$

Profile of a standard 3D MG V(2,2)–cycle:

- uniform grids, 7–point stencil
- size of the finest grid: 129^3 points
- restriction: half injection
- (tri–)linear interpolation
- standard r/b Gauss–Seidel smoother

subroutine	time consumption
relaxation	86.0%
interpolation	8.0%
restriction	5.1%

In 2D: similar profiles

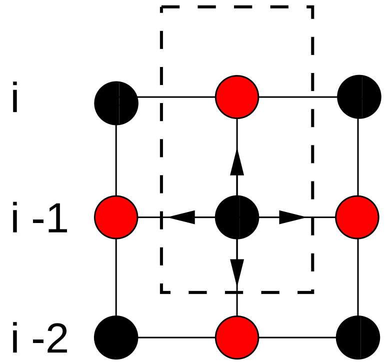
Tune the relaxation algorithm first to

optimize data locality behavior

Optimization techniques for 2D

In the beginning: 60 MFlops/sec
(DEC Alpha PWS 500au, grid size 1024×1024)

$\boxed{\text{1. Fusing relaxation of red and black points}}$



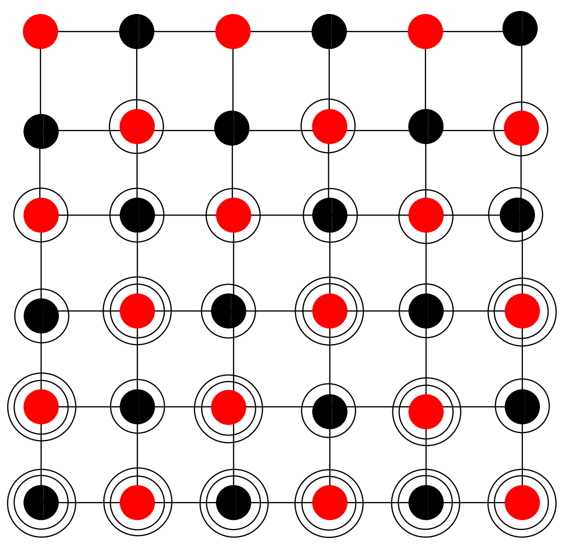
Result: 60 MFlops/sec $\longrightarrow$ 113 MFlops/sec

Optimization techniques for 2D

$\boxed{\text{2. Blocking of several Gauss–Seidel sweeps}}$

By one pass through the whole grid several smoothing steps are performed

<u>Illustration:</u>

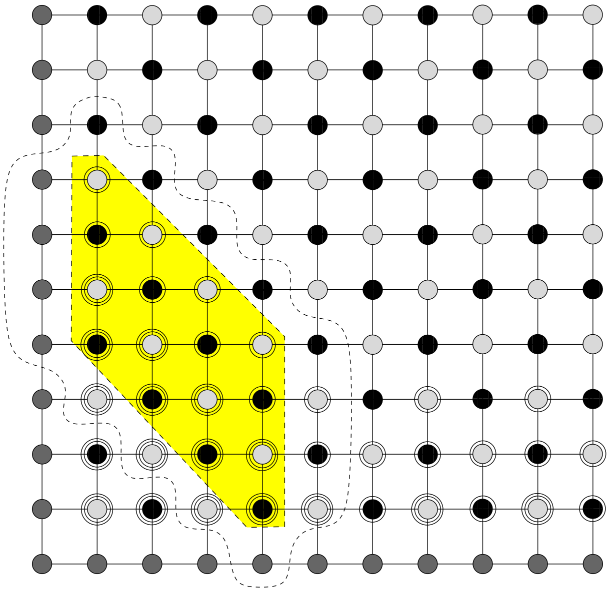


<u>Result:</u> 113 MFlops/sec $\longrightarrow$ 150 MFlops/sec

Optimization techniques for 2D

Finally: Sophisticated data access patterns + ...

(still respecting all data dependencies)



... + array padding

```
double precision u(0:N+pad1,0:N)
double precision f(0:N+pad2,0:N)
```

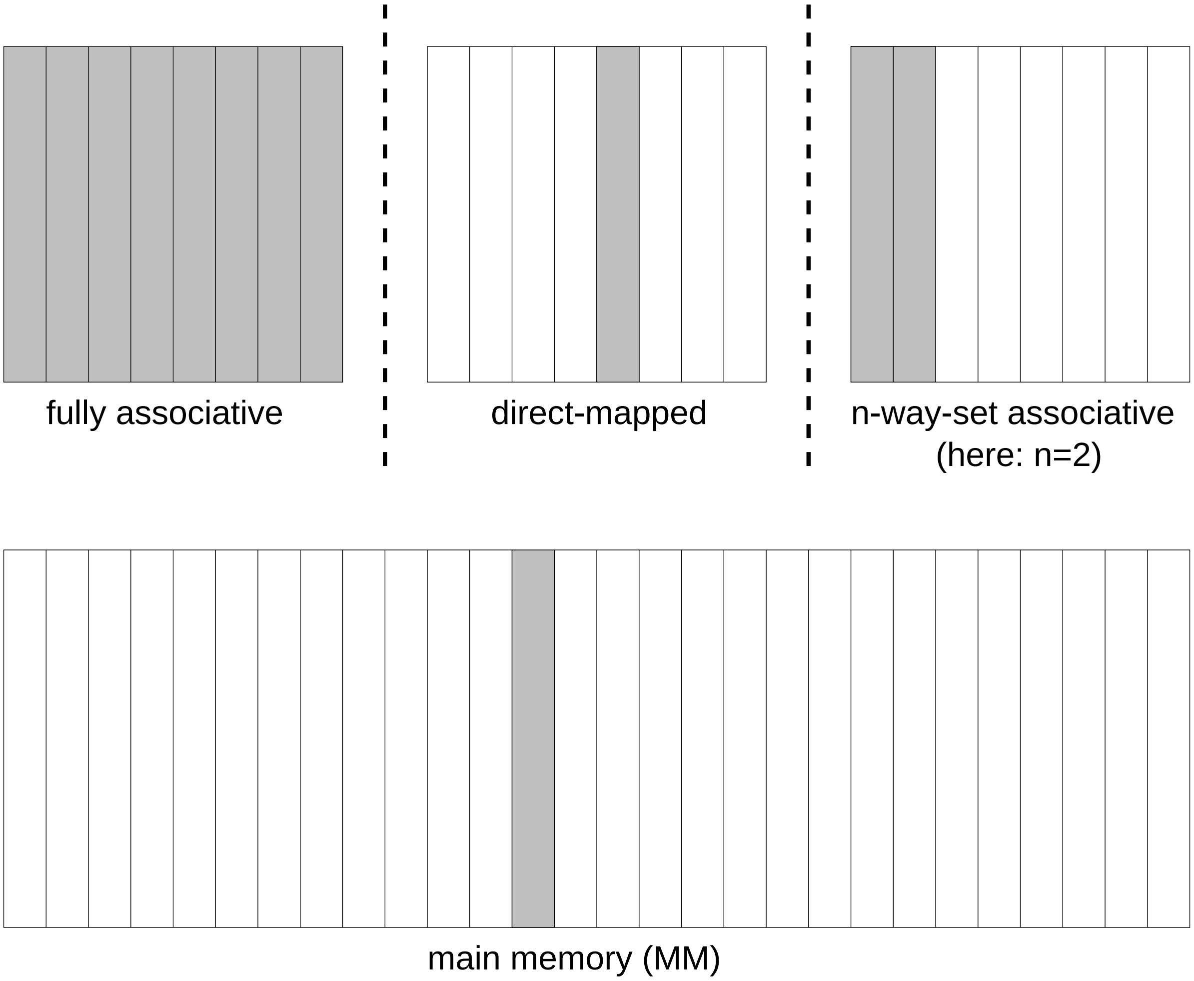
Result: 150 MFlops/sec ⟶ 270 MFlops/sec

Why does array padding work?

Where are MM blocks placed in the cache?

Classification of cache types by their

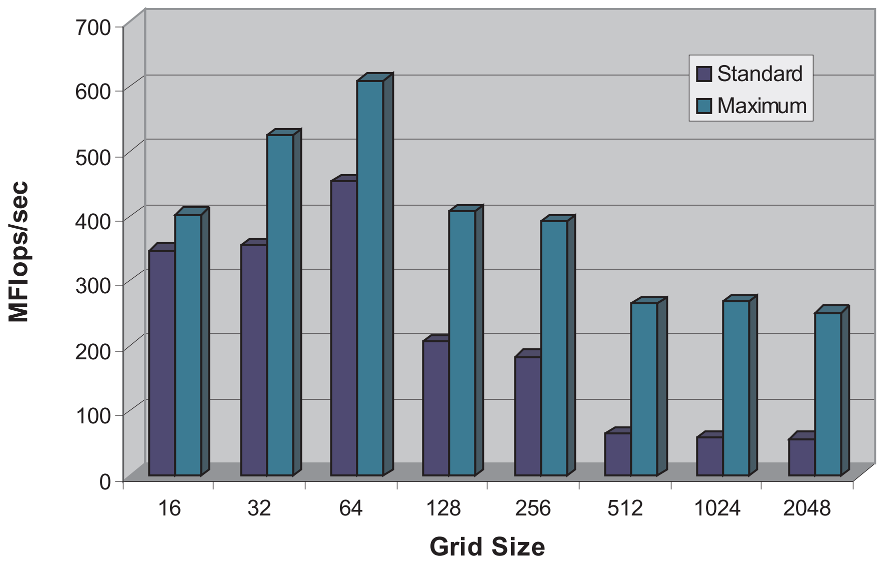
degree of associativity



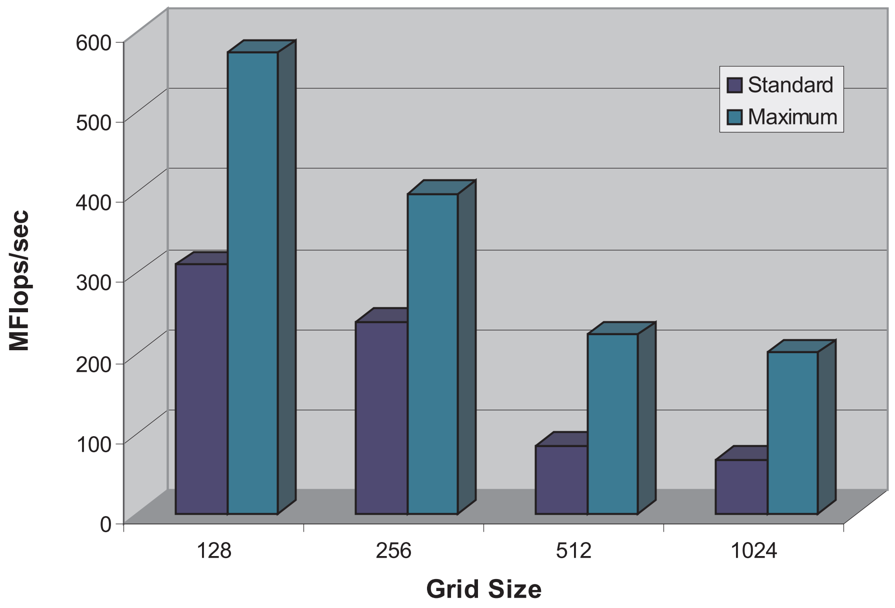
Padding changes the distance of blocks in MM and helps to avoid cache conflict misses

Speedups for 2D

Red/black Gauss–Seidel:



V(4,0)–cycle using cache–optimized relaxation:



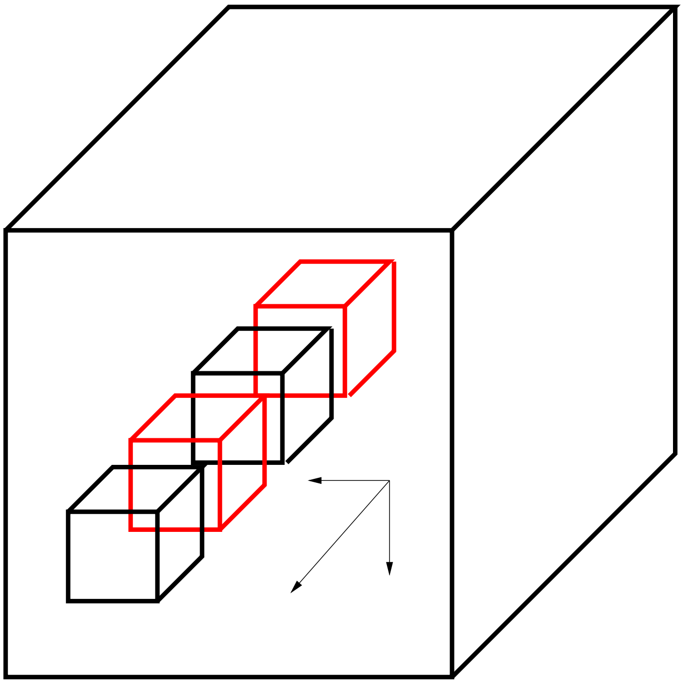
Optimization techniques for 3D

Problem:

Usually caches are not big enough to hold several adjacent planes of large 3D grids

Idea:

Use a blocking technique based on a small subcube moving through the grid, respecting the data dependencies



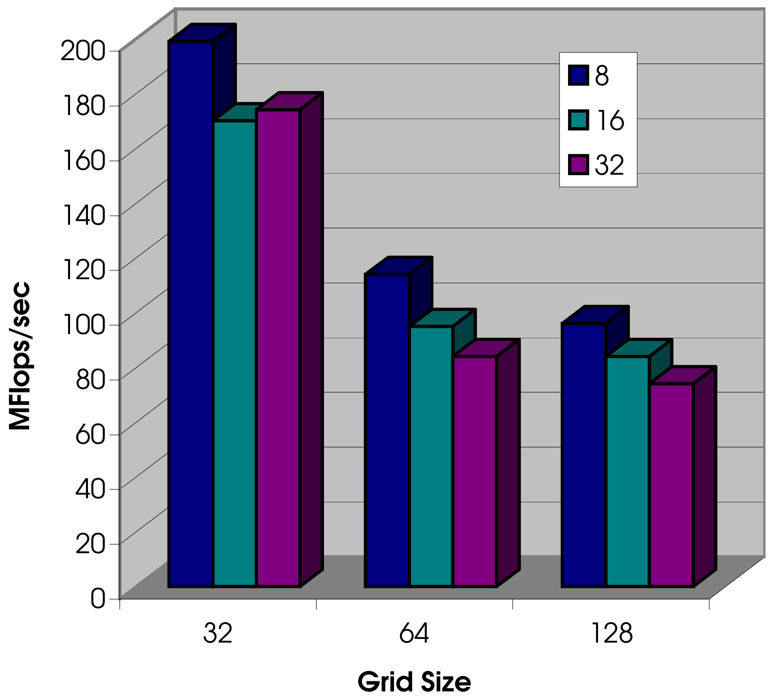
Algorithm:

```
for each subcube position within the grid
      update red points in subcube
      pos = pos + (-1,-1,-1)
      update black points in subcube
```

Optimization techniques for 3D

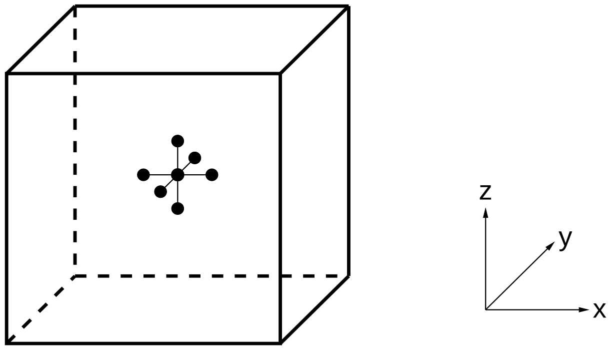
Poor performance results:

We vary the size of the moving subcube



This is due to a high rate of cache conflict misses and can easily be illustrated ...

Optimization techniques for 3D



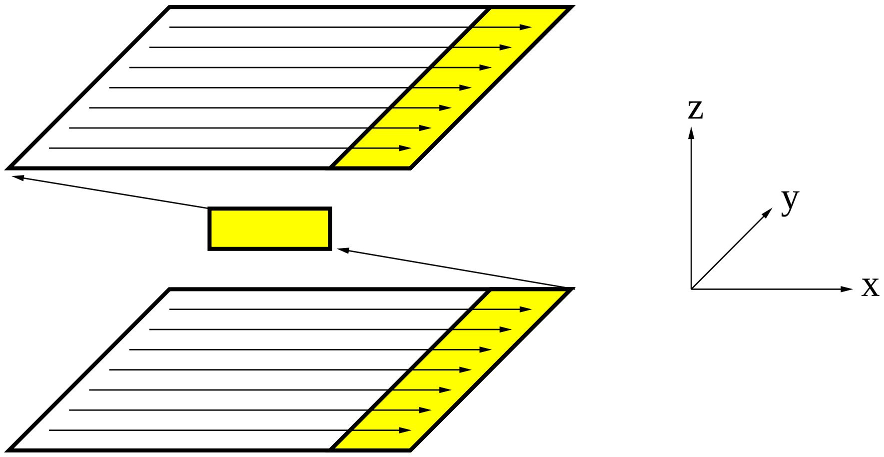
- 3D grid (array) with 64^3 elements
  `double precision u(0:63,0:63,0:63)`
- 8 kByte direct–mapped L1 cache
  (e.g. Alpha 21164)
- x is leading, z is trailing array dimension

Consequences:

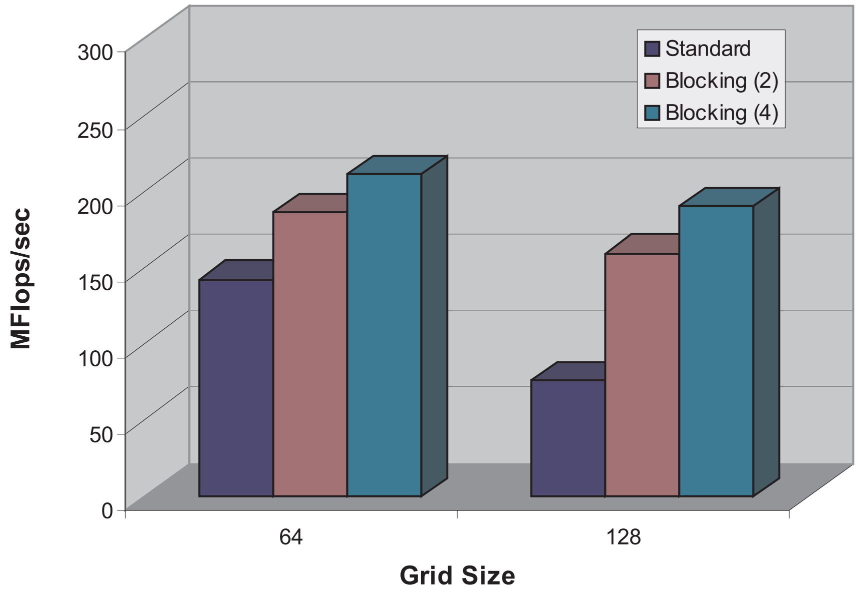
- x-adjacent grid points are adjacent in MM
- y-adjacent grid points are 512 Byte away from each other in MM
- z-adjacent grid points are $4 \cdot$ 8 kByte away from each other in MM and therefore map to the same cache location: conflict misses

Optimization techniques for 3D

Remedy:
Multi–dimensional array padding:



Further speedups:
Blocking of several red/black Gauss–Seidel sweeps



- efficient particularly on the large grids of the hierarchy
- the relaxation of these large grids consumes most of the V–cycle execution time

Optimization techniques for 3D

Performance of MG V–cycles (in MFlops/sec):

127^3 unknowns on the finest level

- V(2,2): only two GS sweeps can be blocked

standard	cache–optimized
72	126

- V(4,0): four GS sweeps can be blocked

standard	cache–optimized
73	138

Closing remarks

Compaq XP1000 (Alpha 21264):

Speedup results in MFlops/sec:

	standard	cache–optimized
red/black GS	140	315
V(2,2)	184	224
V(4,0)	184	243

- Larger L1 cache (64 instead of 8 kByte)
- Higher memory bandwidth

Nevertheless:
Remarkable speedups can be obtained

More details, e.g. concerning different machines:
Supercomputing '99 Conference,
November, 13–19, Portland, Oregon

Closing remarks

"Mathematical tuning":

- Higher–oder discretizations
  (e.g. Mehrstellen operator in 2D)
  - More computational work
  - More accurate results
- Use of relaxation parameters
  - Possibility of applying local smoothers to moderately anisotropic problems (I. Yavneh)
    $\longrightarrow$ (almost) no loss of performance
  - Use of SOR smoothers can improve MG convergence speed

Future work

- Up to now:
  Our efforts have focused on understanding (simple) special cases and the basic effects

- Extensions:

  - More general PDEs
    (e.g. involving variable coefficients or more complicated smoothing methods)
  - Use of adaptively refined meshes
  - Application of inherently data–local algorithms (e.g. adaptive GS, U. Rüde)

- Goal: High–Performance Multigrid Library

  - Framework of cache–optimized MG components
  - User–friendly interface

- Contact:
  Markus.Kowarschik@cs.fau.de