Memory Characteristics of Iterative Methods

Christian Weiß, Markus Kowarschik

Ulrich Rüde, Wolfgang Karl

University of Erlangen–Nuremberg
Technical University of Munich

in collaboration with:
Craig. C. Douglas, Jonathan Hu (University of Kentucky)
Dan Quinlan (Lawrence Livermore National Laboratory)
Federico Bassetti (Los Alamos National Laboratory)

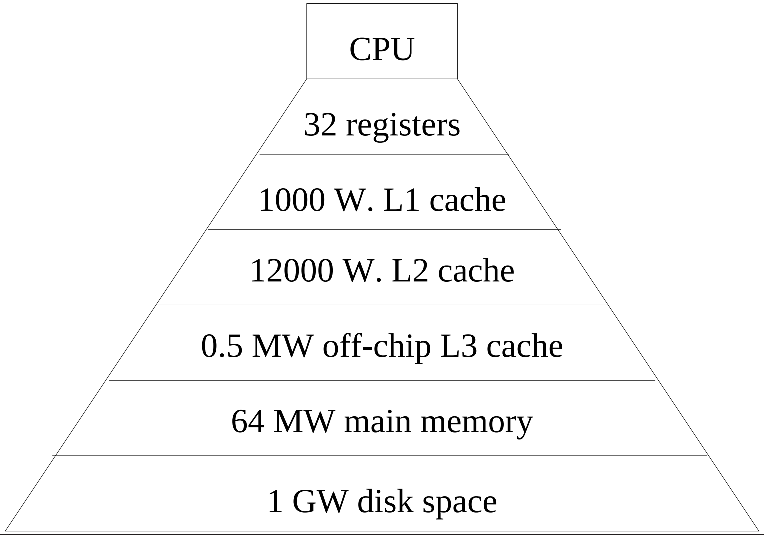
Outline

1. The fundamental problem:
   deep memory hierarchies — iterative numerical algorithms
2. Cache optimization techniques for numerical algorithms
3. Conclusions and future work

Memory hierarchies

Intended to hide performance gap between CPU and main memory

Example: DEC Alpha PWS 500au (A21164–based):



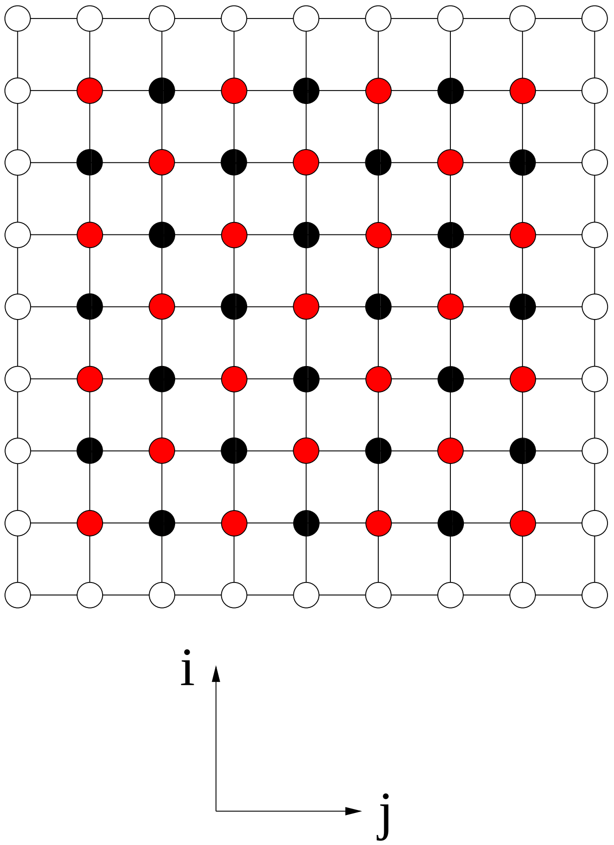
Level	Capacity	Throughput	Latency
Register	512 Byte	24 GByte/s	2 ns
L1 Cache	8 kByte	16 GByte/s	2 ns
L2 Cache	96 kByte	8 GByte/s	6 ns
L3 Cache	4 MByte	888 MByte/s	24 ns
Main Mem	512 MByte	1 GByte/s	112 ns

Iterative methods

- Numerical solution of large linear systems of equations (e.g. arising from PDE discretization)
- Repeated passes through the whole data set (grid), even in the case of more involved iteration schemes like conjugate gradient and multigrid methods
- The problems may be very large, particularly in 3D:
  e.g. $128^3 > 2 \cdot 10^6$

Iterative methods

Example: red–black Gauss–Seidel (or SOR–) iteration in 2D:




```
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
```

First profiling results

Analysis of this Gauss–Seidel code using the *Digital Continuous Profiling Infrastructure (DCPI)* on a DEC PWS 500au (peak performance: 1 GFLOPS):

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

First profiling results

Memory accesses:

Grid Size	Data Set Size	% of all accesses which are satisfied by				
		±	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

How can we improve cache performance?

$\Rightarrow$ This is the subject of our DiME project
(Data–local iterative methods)

Summing up the problem

Fact:

Data kept in "higher layers" of the memory hierarchy can be accessed much faster

But:

Iterative algorithms (e.g. MG methods) on grids usually perform entire sweeps through the whole data set

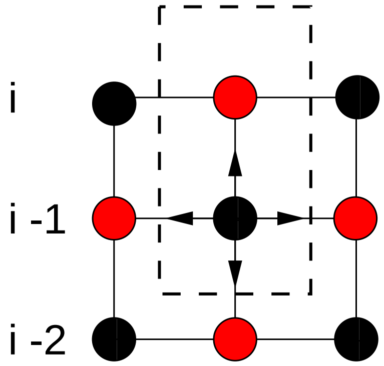
Conclusion:

Iterative algorithms should take care of the underlying memory architecture in order to be efficient, but this is difficult!

Data locality optimizations in 2D

In the beginning: 60 MFLOPS
(refers to a DEC Alpha PWS 500au, grid size 1024×1024)

1. Fusing relaxation of red and black points

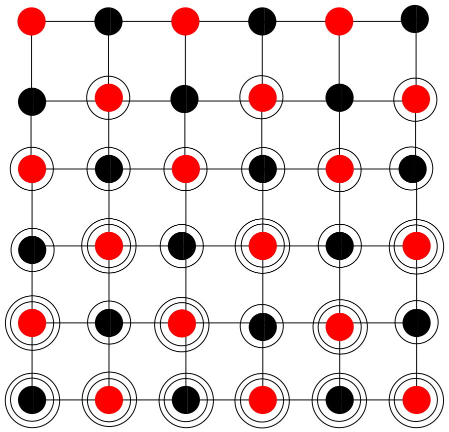


Result: 60 MFLOPS $\longrightarrow$ 113 MFLOPS

Data locality optimizations in 2D

2. Blocking of several Gauss–Seidel sweeps

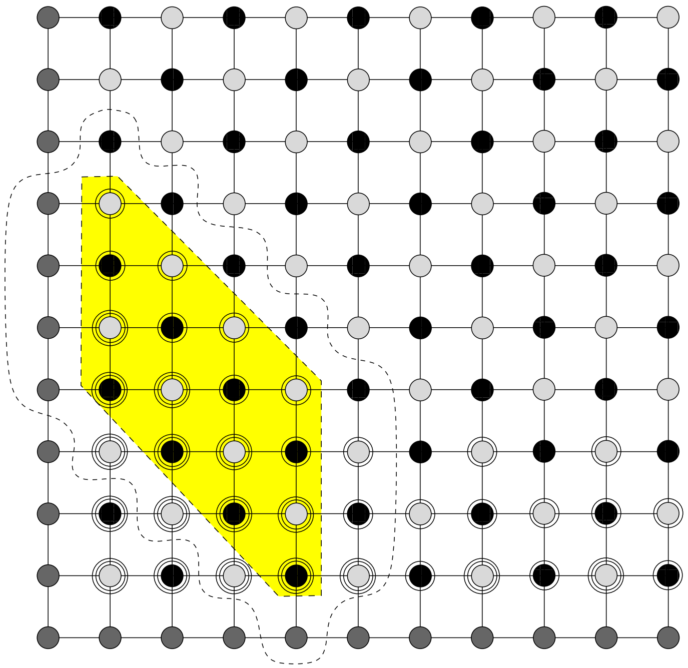
By one pass through the whole grid several smoothing steps can be performed together



Result: 113 MFLOPS $\longrightarrow$ 150 MFLOPS (for two iterations fused)

Data locality optimizations in 2D

Finally: Sophisticated data access patterns + ...



Result: 90 MFLOPS (Why?)

Data locality optimizations in 2D

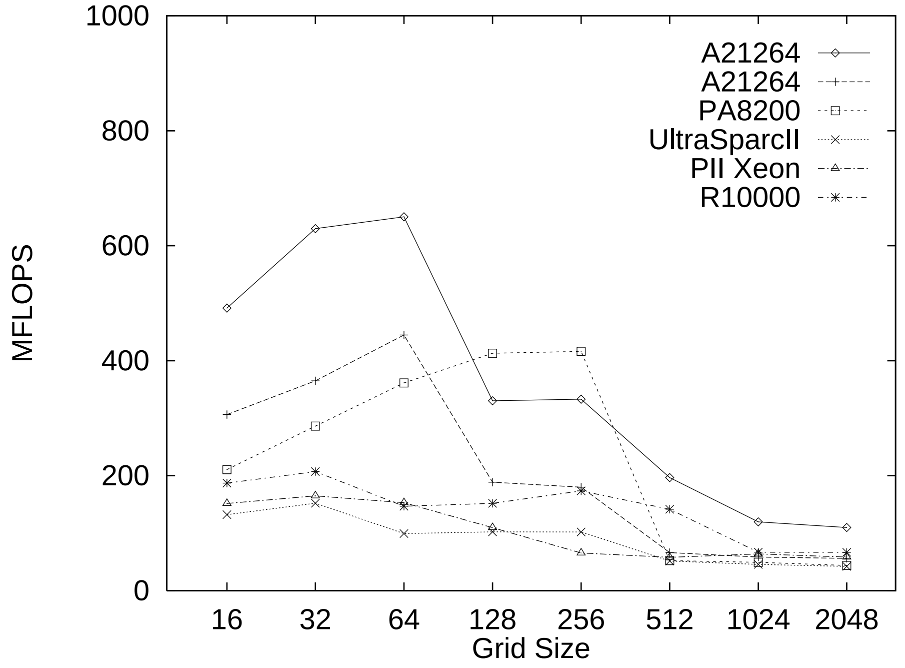
... + array padding to reduce cache conflict misses

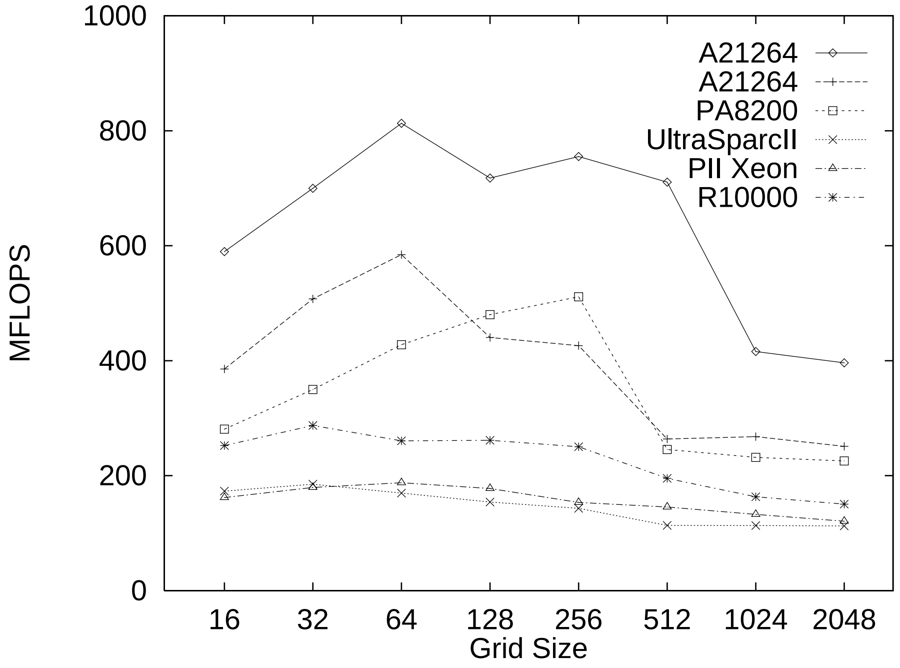
```
double precision u(0:N+pad1,0:N)
double precision f(0:N+pad2,0:N)
```

Result: 90 MFLOPS $\longrightarrow$ 270 MFLOPS (!)

Data locality optimizations in 2D

Resulting speedups for red–black Gauss–Seidel:

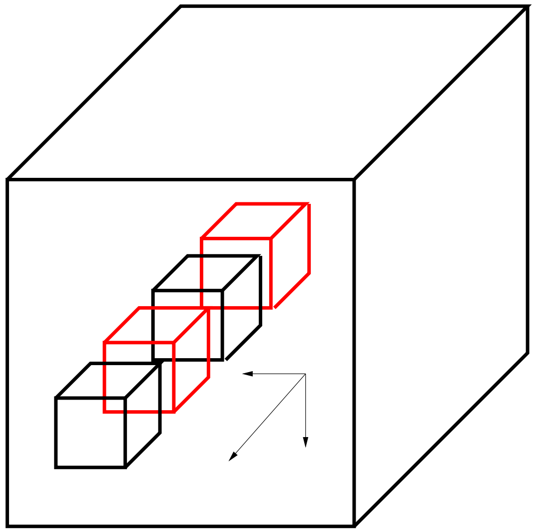




- Our techniques are generally applicable
- Speedups depend on the machine architecture

Data locality optimizations in 3D

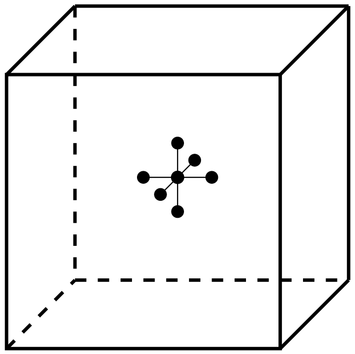
Idea: Use a 3D blocking technique

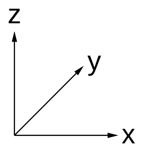


- Based on a small cuboid marching through the large grid
- Data dependencies of red–black Gauss–Seidel are respected
- Like in 2D: fusion and blocking technique

Data locality optimizations in 3D

We obtain poor performance results:





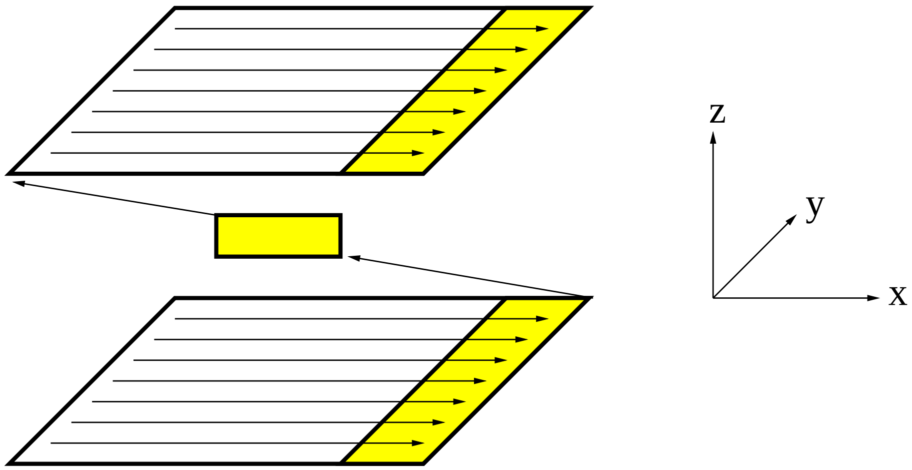
- 3D grid (array) with 64^3 elements
  `double precision u(0:63,0:63,0:63)`
- 8 kByte direct–mapped L1 cache
  (e.g. Alpha 21164)
- x is leading, z is trailing array dimension

Consequences:

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

Data locality optimizations in 3D

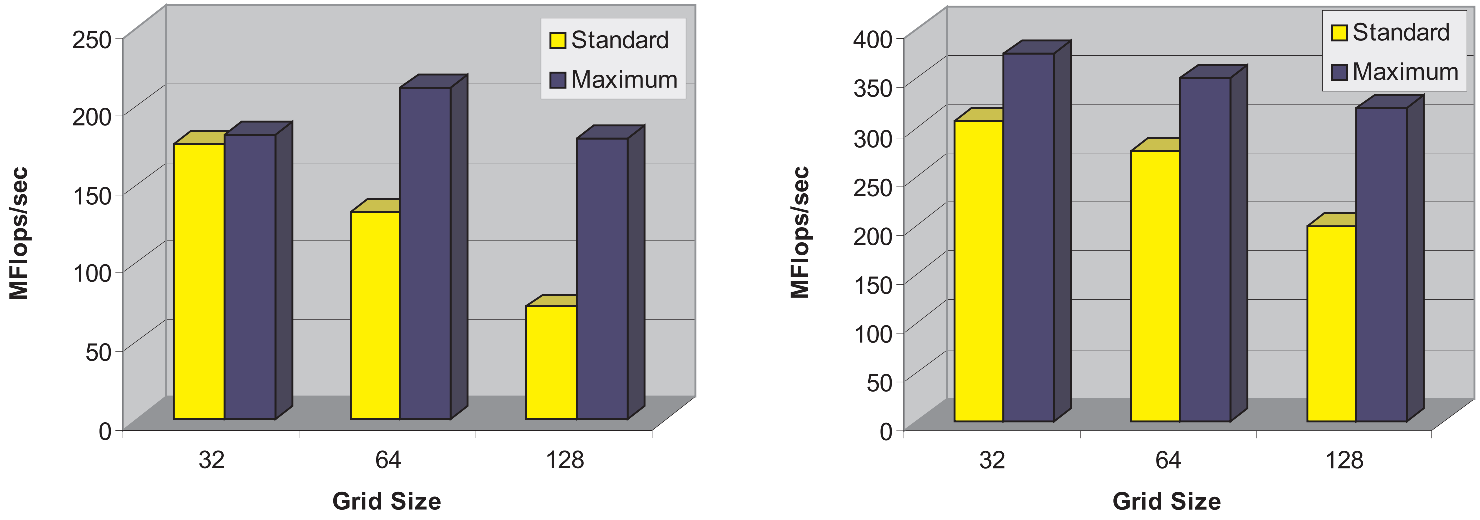
Remedy: Use a multi–dimensional padding technique



- Intra–plane padding reduces conflicts caused by y–neighbors
- Inter–plane padding reduces conflicts caused by z–neighbors

Data locality optimizations in 3D

Speedups for 3D red–black Gauss–Seidel on a DEC PWS 500au (A21164–based) and a Compaq XP1000 (A21264–based)



- Speedups are not as impressive as in the 2D case
- This is mainly due to a higher potential of conflict misses

Conclusions and future work

- We are presently developing a cache–optimized multigrid library, involving more general problem classes, e.g.
  - variable coefficients
  - general boundary conditions
  - anisotropic behavior
- Our colleagues Craig C. Douglas and Jonathan Hu focus on unstructured grids, whereas we study the case of structured grids
- By now, we have only focused on algorithmic modifications which maintain bitwise compatibility (in order to enable automatic code transformations)
- We will investigate non–standard algorithms which are inherently data–local, invloving e.g. patch–adaptive mesh refinement and adaptive smoothing techniques

Further information available on the web

http://wwwbode.in.tum.de/Par/arch/cache