Supercomputing 2001 Tutorial M13: Cache Based Iterative Algorithms

Craig C. Douglas[1] and *Ulrich Ruede*[2]

and

J.Hu, *M.Kowarschik*,

G. Haase, W. Karl, H. Pfaender, L.Stals,

D.T. Thorne, and C.Weiss

November 12, 2001

[1] University of Kentucky and Yale University

[2] Universitaet Erlangen-Nuernberg

Contacting Us by Email

- douglas@ccs.uky.edu or douglas-craig@cs.yale.edu
- ulrich.ruede@informatik.uni-erlangen.de
- jhu@ca.sandia.gov
- markus.kowarschik@cs.fau.de

Overview

- Part I: Architectures and fundamentals
- Part II: Techniques for structured grids
- Part III: Unstructured grids

Part I
Architectures and Fundamentals

- Why worry about performance (an illustrative example)
- Fundamentals of computer architecture
  - CPUs, pipelines, superscalar operation
  - Memory hierarchy
- Cache memory architecture
- Optimization techniques for cache based computers

How Fast Should a Solver Be

(just a simple check with theory)

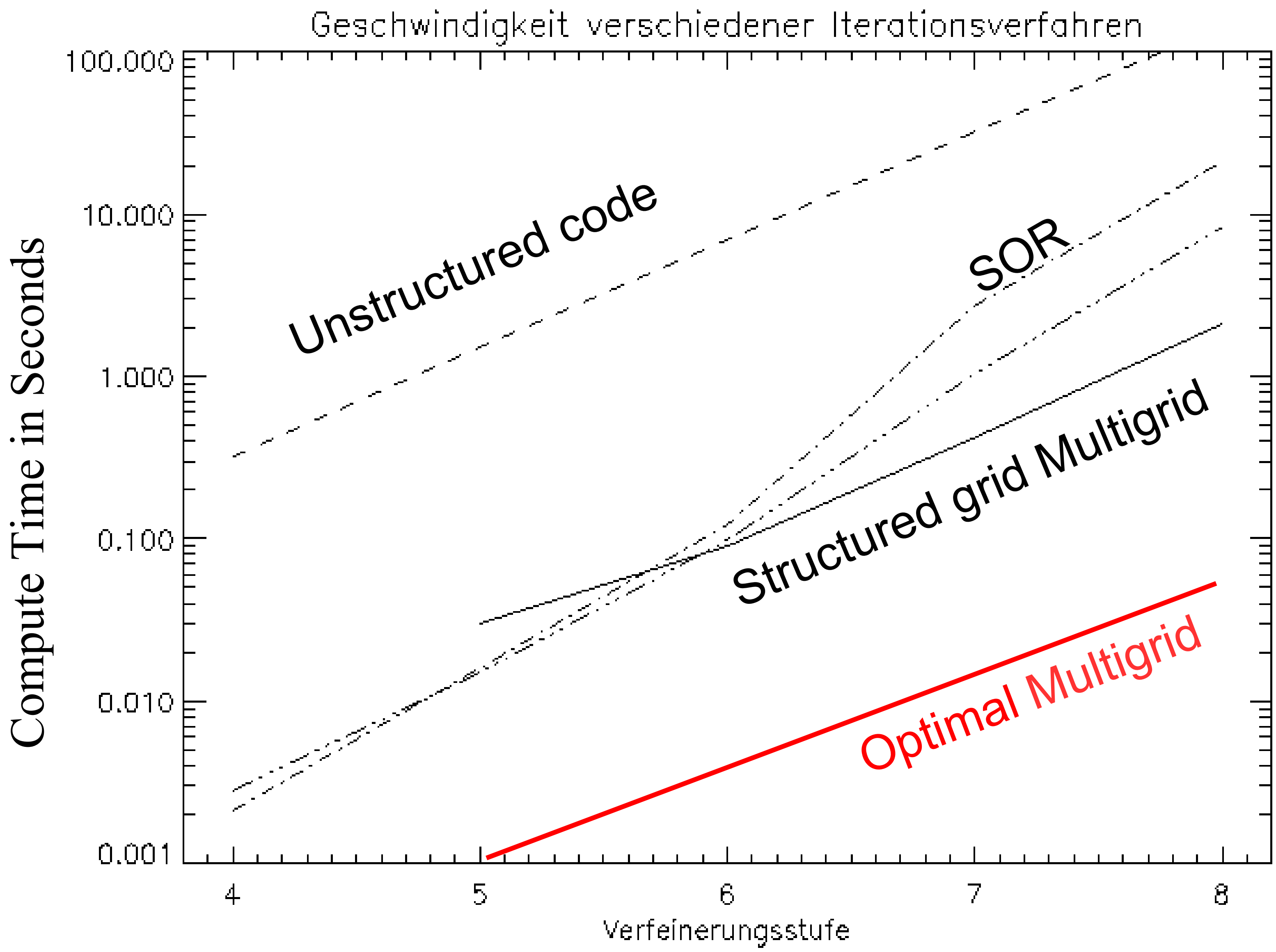
- Poisson problem can be solved by a multigrid method in <30 operations per unknown (known since late 70ies)
- More general elliptic equations may need O(100) operations per unknown
- A modern CPU can do >1 Gflop per second
- So we should be solving 10 Million unknowns per second
- Should need O(100) Mbyte memory

How Fast Are Solvers Today

- Often no more than 10,000 to 100,000 unknowns possible before the code breaks
- In a time of minutes to hours
- Needing horrendous amounts of memory

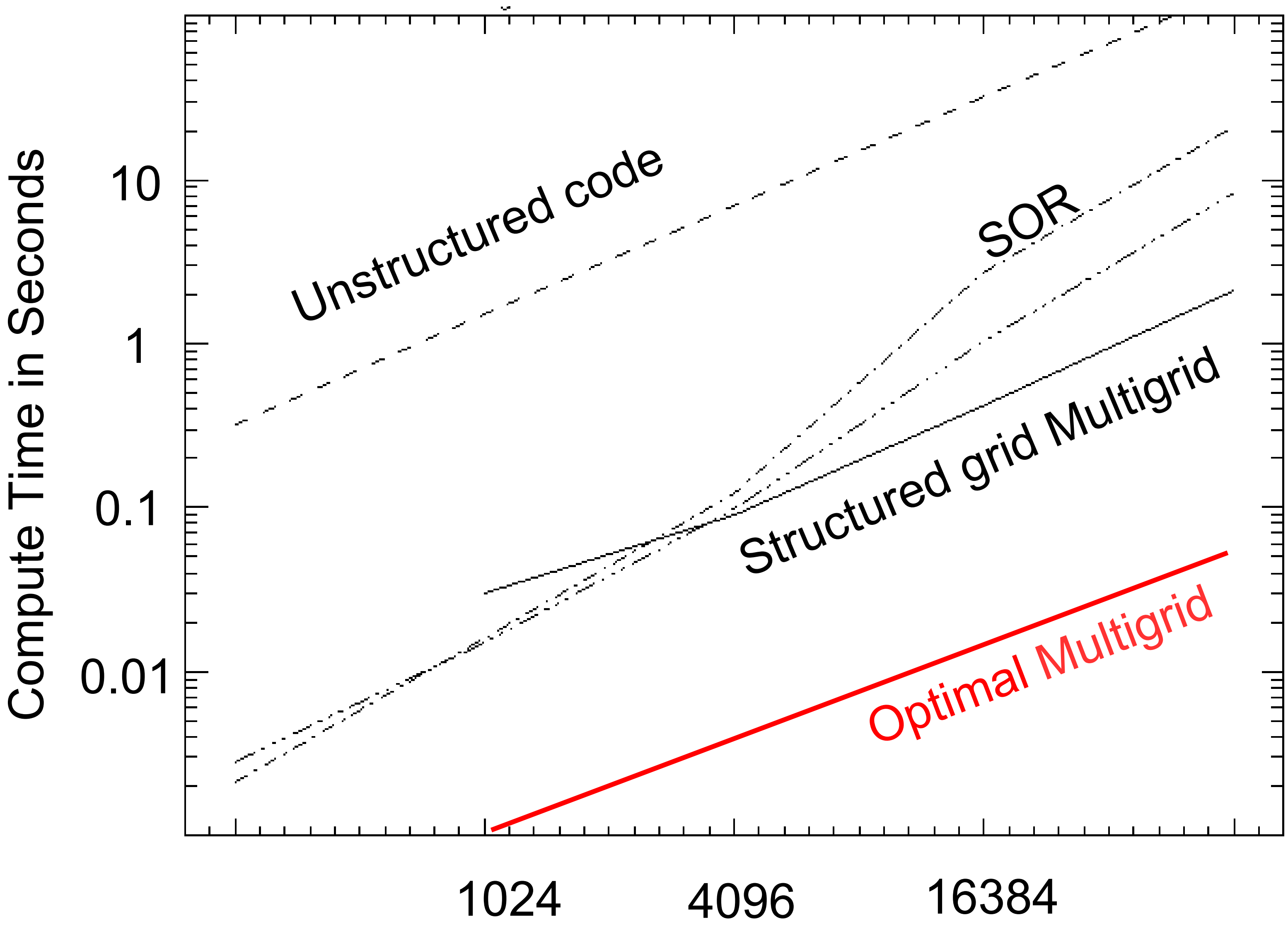
Even state of the art codes
are often very inefficient

Comparison of Solvers



Comparison of Solvers

(what got me started in this business ~ '95)

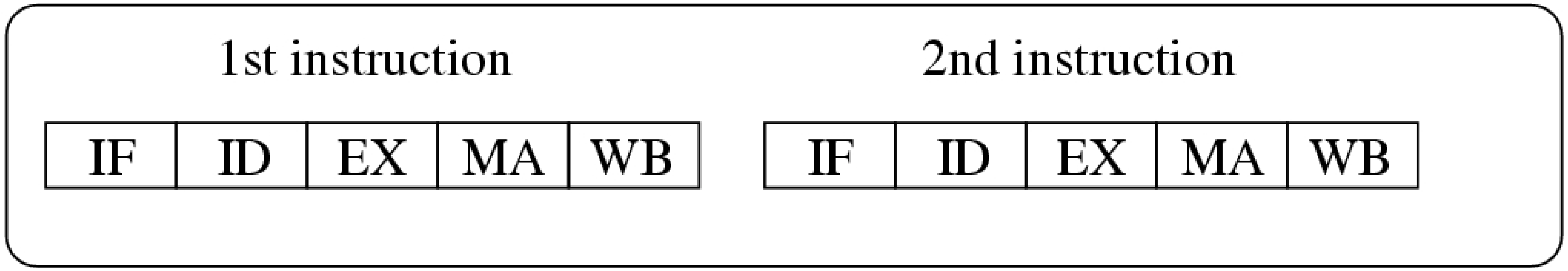


Elements of CPU Architecture

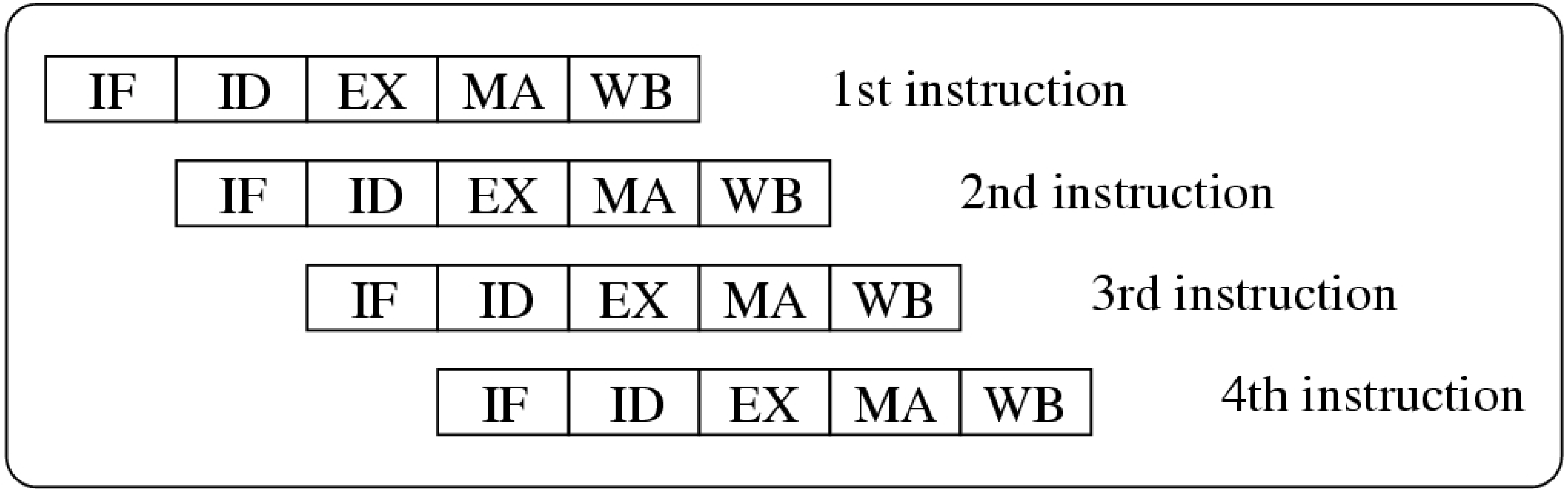
- Modern CPUs are
  - Superscalar: they can execute more than one operation per clock cycle, typically:
    - 4 integer operations per clock cycle plus
    - 2 or 4 floating-point operations (multiply-add)
  - Pipelined:
    - Floating-point ops take O(10) clock cycles to complete
    - A set of ops can be started in each cycle
  - Load-store: all operations are done on data in registers, all operands must be copied to/from memory via load and store operations
- Code performance heavily dependent on compiler (and manual) optimization

Pipeling (I)

sequential execution:

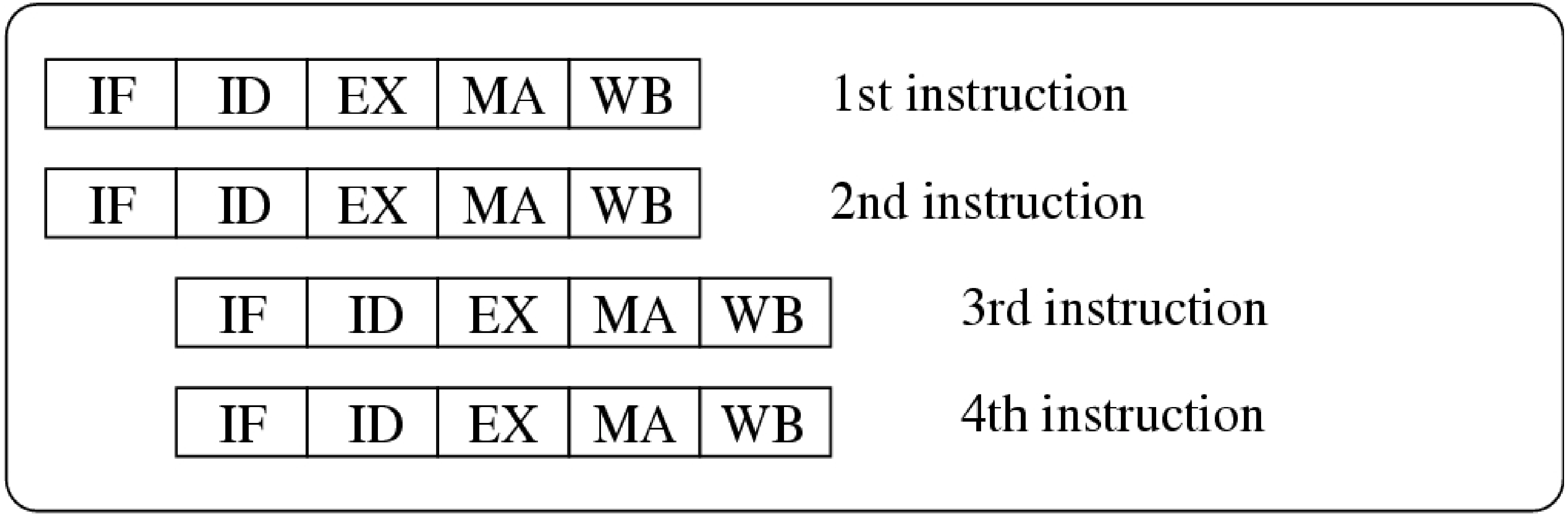


pipelined execution:



Pipelining (II)

pipelined and superscalar execution:



CPU Trends

- EPIC (alias VLIW)
- Multi-threaded architectures
- Multiple CPUs on a single chip
- Within the next decade
  - Billion transistor CPUs (today 100 million transistors)
  - Potential to build TFLOPS on a chip
  - But no way to move the data in and out sufficiently quickly

The Memory Wall

- Latency: time for memory to respond to a read (or write) request is too long
  - CPU ~ 0.5 ns (light travels 15cm in vacuum)
  - Memory ~ 50 ns
- Bandwidth: number of bytes which can be read (written) per second
  - CPUs with 1 GFLOPS peak performance standard: needs 24 Gbyte/sec bandwidth
  - Present CPUs have peak bandwidth <5 Gbyte/sec and much less in practice

Memory Acceleration Techniques

- Interleaving (independent memory banks store consecutive cells of the address space cyclically)
  - Improves bandwidth
  - But *not* latency
- Caches (small but fast memory) holding frequently used copies of the main memory
  - Improves latency and bandwidth
  - Usually comes with 2 or 3 levels nowadays
  - But only works when access to memory is *local*

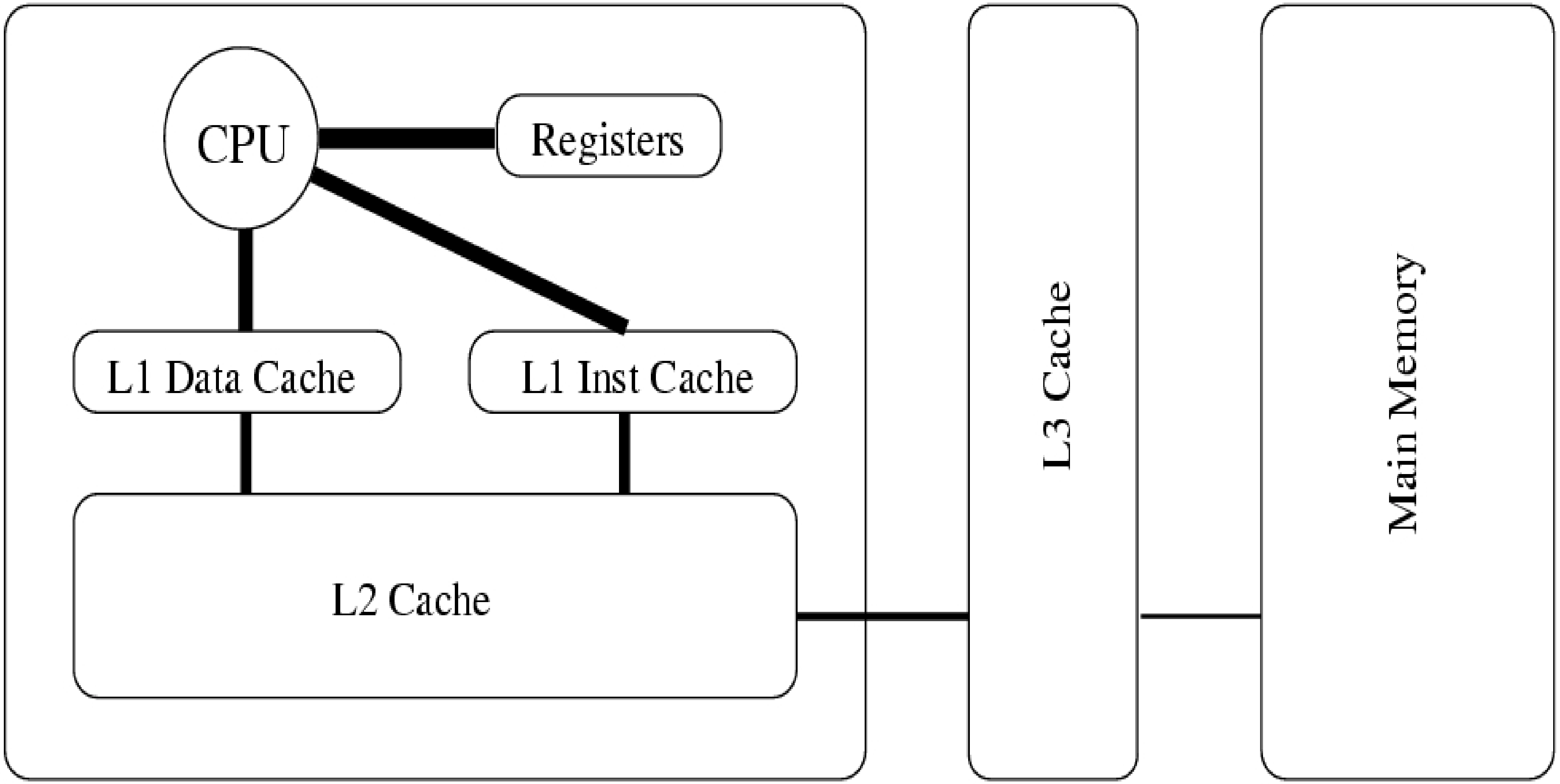
Principles of Locality

- Temporal: an item referenced now will be again soon.
- Spatial: an item referenced now causes neighbors to be referenced soon.
- *Cache lines* are typically 32-128 bytes with 1024 being the longest currently. Lines, not words, are moved between memory levels. Both principles are satisfied. There is an optimal line size based on the properties of the data bus and the memory subsystem designs.

Caches

- Fast but small extra memory
- Holding identical copies of main memory
- Lower latency
- Higher bandwidth
- Usually several levels (2 or 3)
- Same principle as virtual memory
- Memory requests are satisfied from
  - Fast cache (if it holds the appropriate copy): **Cache Hit**
  - Slow main memory (if data is not in cache): **Cache Miss**

Alpha Cache Configuration



Cache Issues

- Uniqueness and transparency of the cache
- Finding the *working set* (what data is kept in cache)
- Data consistency with main memory
- Latency: time for memory to respond to a read (or write) request
- Bandwidth: number of bytes which can be read (written) per second

Cache Issues

- Cache line size:
  - Prefetching effect
  - False sharing (cf. associativity issues)
- Replacement strategy
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)
  - Random (would you buy a used car from someone who advocated this method?)
- Translation Look-aside Buffer (TLB)
  - Stores virtual memory page translation entries
  - Has effect similar to another level of cache

Effect of Cache Hit Ratio

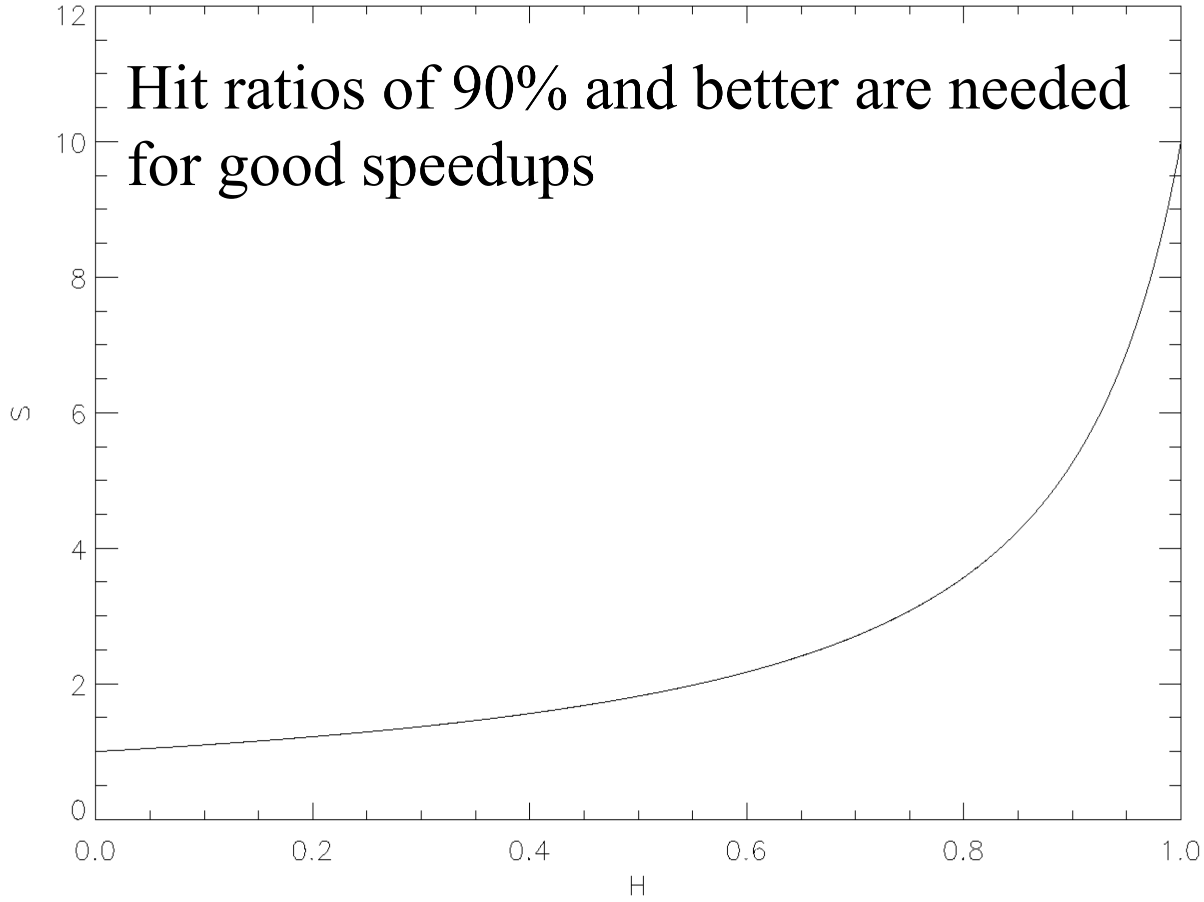
The cache efficiency is characterized by the cache hit ratio, the *effective* time for a data access is

$$T_{\text{eff}} = H \cdot T_c + (1 - H) \cdot T_m.$$

The *speedup* is then given by

$$S = \frac{T_m}{T_{\text{eff}}} = \frac{1}{1 - H(1 - T_c/T_m)}$$

Cache Effectiveness Depends on the Hit Ratio



Cache Organization

- Number of levels
- Associativity
- Physical or virtual addressing
- Write-through/write-back policy
- Replacement strategy (e.g., Random/LRU)
- Cache line size

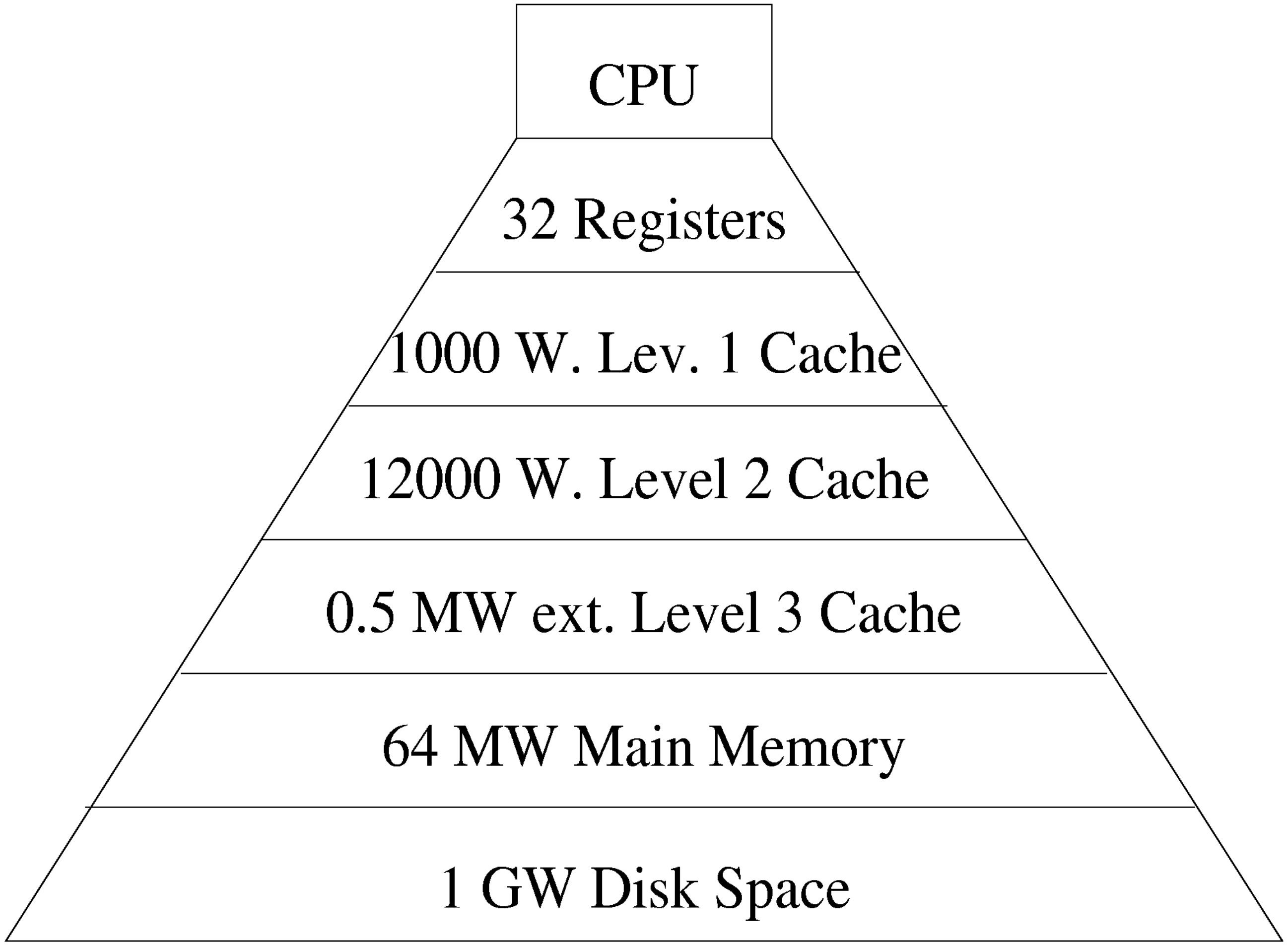
Cache Associativity

- Direct mapped (associativity = 1)
  - Each word of main memory can be stored in *exactly one word* of the cache memory
- Fully associative
  - A main memory word can be stored in any location in the cache
- Set associative (associativity = k)
  - Each main memory word can be stored in one of k places in the cache

Direct mapped and set-associative caches give rise to *conflict misses*.
Direct mapped caches are faster, fully associative caches are too expensive and slow (if reasonably large).
Set-associative caches are a compromise.

Memory Hierarchy

Example: Digital PWS 600 au, Alpha 21164 CPU, 600 MHz



Example: Memory Hierarchy of the Alpha 21164 CPU

Level	Capacity	Throughput	Latency
Register	512 W	24 GB/Sec	2 ns
L1 Cache	8 KB	16 GB/Sec	2 ns
L2 Cache	96 KB	8 GB/sec	6 ns
L3 Cache	4 MB	888 MB/sec	24 ns
Main Mem	512 MB	1 GB/sec	112ns

Typical Architectures

- IBM Power 3:
  - L1 = 64 KB, 128-way set associative
  - L2 = 4 MB, direct mapped, line size = 128, write back
- Compaq EV6 (Alpha 21264):
  - L1 = 64 KB, 2-way associative, line size= 32
  - L2 = 4 MB (or larger), direct mapped, line size = 64
- HP PA: no L2
  - PA8500, PA8600: L1 = 1.5 MB, PA8700: L1 = 2.25 MB
- AMD Athlon: L1 = 64 KB, L2 = 256 KB
- Intel Pentium 4: L1 = 8 KB, L2 = 256 KB
- Intel Itanium:
  - L1 = 16 KB, 4-way associative
  - L2 = 96 KB
    6-way associative, L3 = off chip, size varies

How to Make Codes Fast

1. Use a fast algorithm (multigrid)
   I. It does not make sense to optimize a bad algorithm
   II. However, sometimes a fairly simple algorithm that is well implemented well will beat a very sophisticated, super method that is poorly programmed
2. Use good coding practices
3. Use good data structures
4. Use special optimization techniques
   I. Loop unrolling
   II. Operation reordering
   III. Cache blocking
   IV. Array padding
   V. Etc.

Special Optimization Techniques

- Loop unrolling
- Loop interchange/fusion/tiling (= blocking):
  - Operation reordering
- Cache blocking
  - For *spatial* locality: Use prefetching effect of cache lines (cf. prefetch operations by compiler or programmer)
  - For *temporal* locality: re-use data in the cache often
- Array padding: mitigates associativity conflicts

Some Cache Optimization Strategies

- Prefetching

```
for i = 1,n do
    A(i) = A(i) + β ∗ B(i)
    (intervening code)
```

⇒

```
Prefetch A(1), B(1), and β
(intervening code)
for i = 1,n do
    A(i) = A(i) + β ∗ B(i)
    (intervening code)
    Prefetch A(i+1), B(i+1)
    (intervening code)
```

- Software pipelining

```
for i = 1,n do
    A(i) = A(i) + β ∗ B(i) + γ ∗ C(i)
```

⇒

```
for i = 1,n do
    so =  β ∗ B(I+1)
    to  = γ  ∗ C(I+1)
    A(i) = A(i) + se + te
```

Some Cache Optimization Strategies

- Loop unrolling

```
for i = 1,100  do
    A(i) = A(i) + β ∗ B(i)
```

⇒

```
for i = 1,50  do
    A(2 ∗ i – 1) = A (2 ∗ i – 1)  +
                        β ∗B(2 ∗ i – 1)
    A(2 ∗ i) = A (2 ∗ i) + β ∗B(2 ∗ i)
```

- Loop blocking (multiplication of two M by M matrices)

```
for i = 1,M  do
    for j = 1,M  do
        for k = 1,M  do
            A(i,j) = B(i,k) ∗ C(k,j)
```

⇒

```
for i = 1,M, step by s, do
    for j = 1,M, step by s, do
        for l = i, i + s – 1 do
            for m = j, j + s – 1 do
                for k = 1,M  do
                    A(l,m) = B(l,k) ∗ C(k,m)
```

Some Cache Optimization Strategies

- Array Padding

(cache size = 1 MB, size of real = 8 bytes, line length = 32 bytes)

```
integer CS = 1024 ∗ 1024
real A(CS), B(CS)
for i = 1,CS  do
    A(i) = A(i) + β ∗ B(i)
```

⇨

```
integer CS = 1024 ∗ 1024
real A(CS), pad(4), B(CS)
for i = 1,CS  do
    A(i) = A(i) + β ∗ B(i)
```

Cache Optimization Techniques

- Data layout transformations
- Data access transformations

Here we only consider equivalent transformations of a given algorithm, except possibly different roundoff properties.

- Optimize for *spatial* locality: Use automatic prefetching of data in same cache line
- Optimize for *temporal* locality: Re-use data which has been brought to cache as often as possible

Types of Cache Misses

- Compulsory misses (= cold/startup misses)
- Capacity misses (working set too large)
- Conflict misses (degree of associativity insufficient)

Summary of Part I

- Iterative algorithms frequently underperform on deep memory hierarchy computers
- Must understand and use properties of the architectures to exploit potential performance
- Using a good algorithm comes first
- Optimization involves
  - Designing suitable data structures
  - Standard code optimization techniques
  - Memory layout optimizations
  - Data access optimizations

More about this in the next two parts!

Part II
Techniques for Structured Grids

- Code profiling
- Optimization techniques for computations on structured grids
  - Data layout optimizations
  - Data access optimizations

Profiling: Hardware Performance Counters

Dedicated CPU registers are used to count various events at runtime, e.g.:

- Data cache misses (for different levels)
- Instruction cache misses
- TLB misses
- Branch mispredictions
- Floating-point and/or integer operations
- Load/store instructions

Profiling Tools: PCL

- 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)
- http://www.kfa-juelich.de/zam/PCL

Profiling Tools: PAPI

- PAPI = Performance API
- Available for many platforms (Portability!)
- Two interfaces:
  - *High-level* interface for simple measurements
  - Fully programmable *low-level* interface, based on thread-safe groups of hardware events *(EventSets)*
- **http://icl.cs.utk.edu/projects/papi**

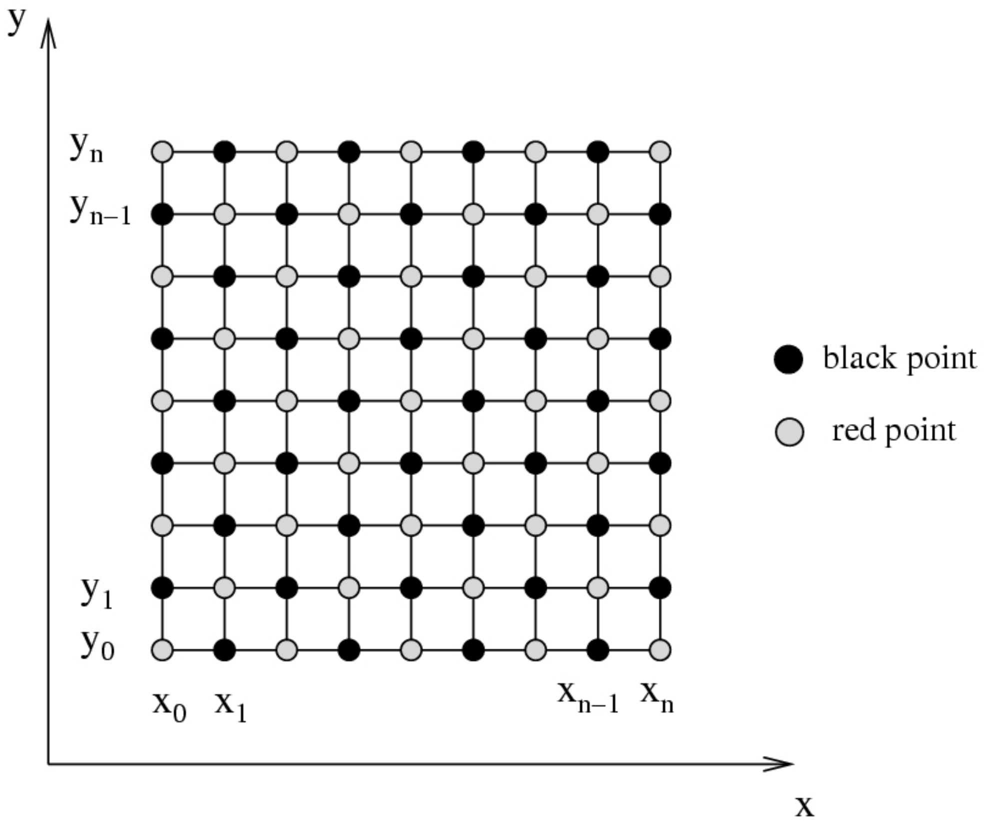
Profiling Tools: DCPI

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

Our Reference Code

- 2D structured multigrid code written in C
- Double precision floating-point arithmetic
- 5-point stencils
- Red/black Gauss-Seidel smoother
- Full weighting, linear interpolation
- Direct solver on coarsest grid (LU, LAPACK)

Structured Grid



Two Common Multigrid Algorithms

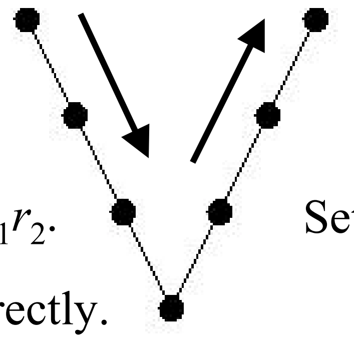
V Cycle to solve $A_4 u_4 = f_4$

Smooth $A_4 u_4 = f_4$. Set $f_3 = R_3 r_4$.

Smooth $A_3 u_3 = f_3$. Set $f_2 = R_2 r_3$.

Smooth $A_2 u_2 = f_2$. Set $f_1 = R_1 r_2$.

Solve $A_1 u_1 = f_1$ directly.

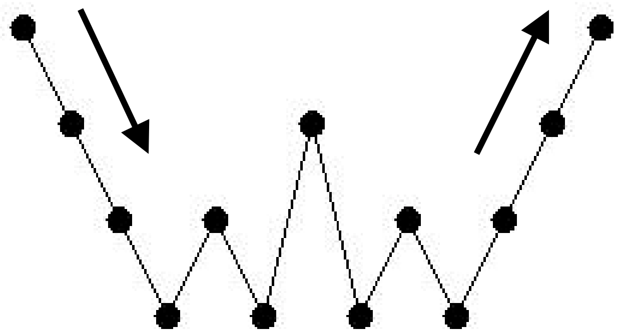


Set $u_4 = u_4 + I_3 u_3$. Smooth $A_4 u_4 = f_4$.

Set $u_3 = u_3 + I_2 u_2$. Smooth $A_3 u_3 = f_3$.

Set $u_2 = u_2 + I_1 \mathrm{u}_1$. Smooth $A_2 u_2 = f_2$.

W Cycle



Using PCL – Example 1

- Digital PWS 500au
  - Alpha 21164, 500 MHz, 1000 MFLOPS peak
  - 3 on-chip performance counters
- PCL Hardware performance monitor: hpm

```
% 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
```

Using PCL – Example 2

```
#include <pcl.h>

int main(int argc, char **argv) {
  // Initialization
  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;
```

Using PCL – Example 2

```
PCLinit(&descr);
if(PCLquery(descr,counter_list,2,flags)!=
   PCL_SUCCESS)
   // Issue error message …
else {
   PCL_start(descr, counter_list, 2, flags);
   // Do computational work here …
   PCLstop(descr,i_result,fp_result,2);
   printf("%i fp instructions, MFLOPS: %f\n",
         i_result[0], fp_result[1]);}
PCLexit(descr);
return 0;}
```

Using DCPI

- Alpha based machines running Compaq Tru64 UNIX
- How to proceed when using DCPI
  1. Start the DCPI daemon (**`dcpid`**)
  2. Run your code
  3. Stop the DCPI daemon
  4. Use DCPI tools to analyze the profiling data

Examples of DCPI Tools

- **`dcpiwhatcg`**: Where have all the cycles gone?
- **`dcpiprof`**: Breakdown of CPU time by procedures
- **`dcpilist`**: Code listing (source/assembler) annotated with profiling data
- **`dcpitopstalls`**: Ranking of instructions causing stall cycles

Using DCPI – Example 1

```
%  dcpiprof ./mg
Column      Total    Period (for events)
------      -----    ------
dmiss       45745      4096
====================================================
dmiss          %      cum%  procedure            image
33320  72.84%   72.84%  mgSmooth             ./mg
10008   21.88%  94.72%  mgRestriction        ./mg
 2411    5.27%  99.99%  mgProlongCorr        ./mg
 [...]
```

Using DCPI – Example 2

Call the DCPI analysis tool:

```
% dcpiwhatcg ./mg
```

Dynamic stalls are listed first:

```
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%
```

Using DCPI – Example 2

```
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%
```

Using DCPI – Example 2

Static stalls are listed next:

```
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%
```

Using DCPI – Example 2

Useful cycles are listed in the end:

```
Useful             7.9%
Nops               1.3%
-----------------------------------------------
Total execution                         9.3%
```

Compare to the total percentage of stall cycles: 90.7% (cf. previous slide)

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 iteration, determine data items needed simultaneously

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

Data Layout Optimizations: Cache Aware Data Structures

- Right-hand side, coefficients are accessed simultaneously, reuse cache line contents (enhance spatial locality) by *array merging*

```
typedef struct {
   double f;
   double aNorth, aEast, aSouth, aWest, aCenter;
} equationData; // Data merged in memory

double u[N][N];                     // Solution vector
equationData rhsAndCoeff[N][N];     // Right-hand side
                                    // andcoefficients
```

Data Layout Optimizations: Array Padding

- Idea: Allocate arrays larger than necessary
  - ⇒Change relative memory distances
  - ⇒Avoid severe *cache thrashing* effects
- Example (Fortran: row major ordering):
  Replace `double precision u(1024, 1024)`
  by `double precision u(1024+pad, 1024)`
- Question: How to choose **`pad`**?

Data Layout Optimizations: Array Padding

- C.-W. Tseng et al. (UMD):
  Research on cache modeling and compiler based array padding:
  - *Intra-variable padding:* pad within the arrays

  ⇒Avoid *self-interference* misses
  - *Inter-variable padding:* pad between different arrays

  ⇒Avoid *cross-interference* misses

Data Access Optimizations: Loop Fusion

- Idea: Transform successive loops into a single loop to enhance temporal locality
- Reduces cache misses and enhances cache reuse (exploit temporal locality)
- Often applicable when data sets are processed repeatedly (e.g., in the case of iterative methods)

Data Access Optimizations: Loop Fusion

- Before:

```
do i= 1,N
   a(i)= a(i)+b(i)
enddo
do i= 1,N
   a(i)= a(i)*c(i)
enddo
```

- **a** is loaded into the cache twice (if sufficiently large)

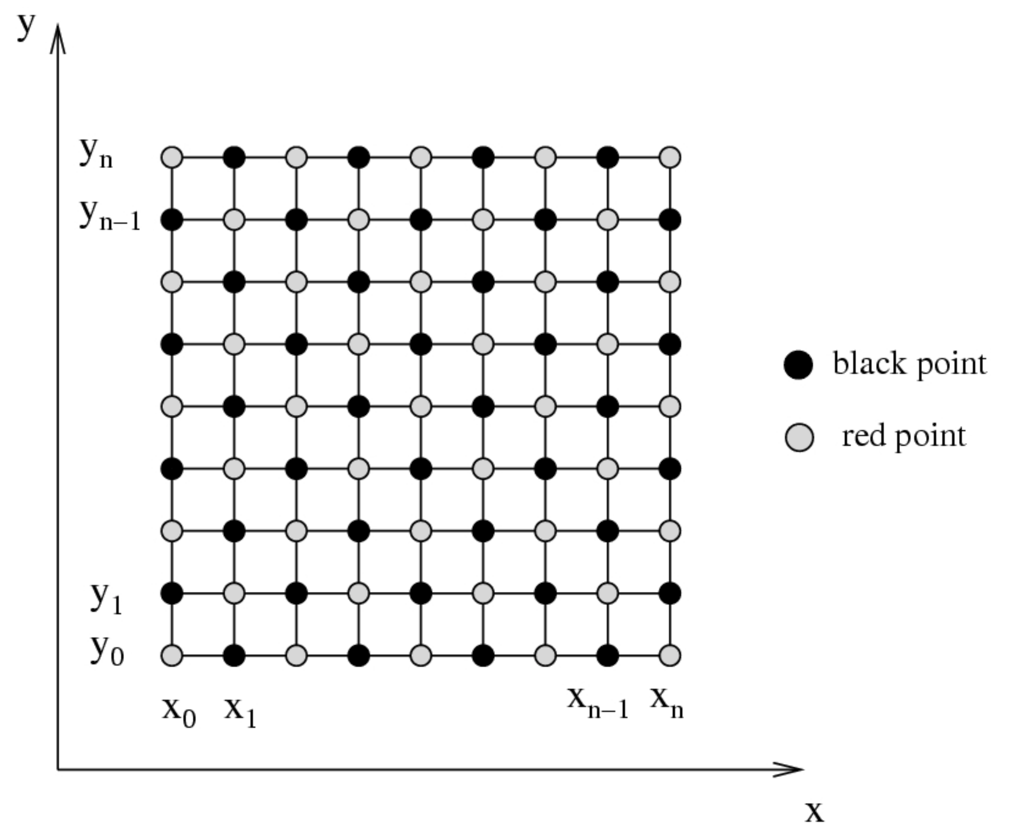
- After:

```
do i= 1,N
   a(i)= (a(i)+b(i))*c(i)
enddo
```

- **a** is loaded into the cache only once

Data Access Optimizations: Loop Fusion

Example: red/black Gauss-Seidel iteration



Data Access Optimizations: Loop Fusion

Code **before** applying loop fusion technique (standard implementation w/ efficient loop ordering, Fortran semantics: row major order)

```
for it= 1 to numIter do
   // Red nodes
   for i= 1 to n-1 do
      for j= 1+(i+1)%2 to n-1 by 2 do
         relax(u(j,i))
      end for
   end for
```

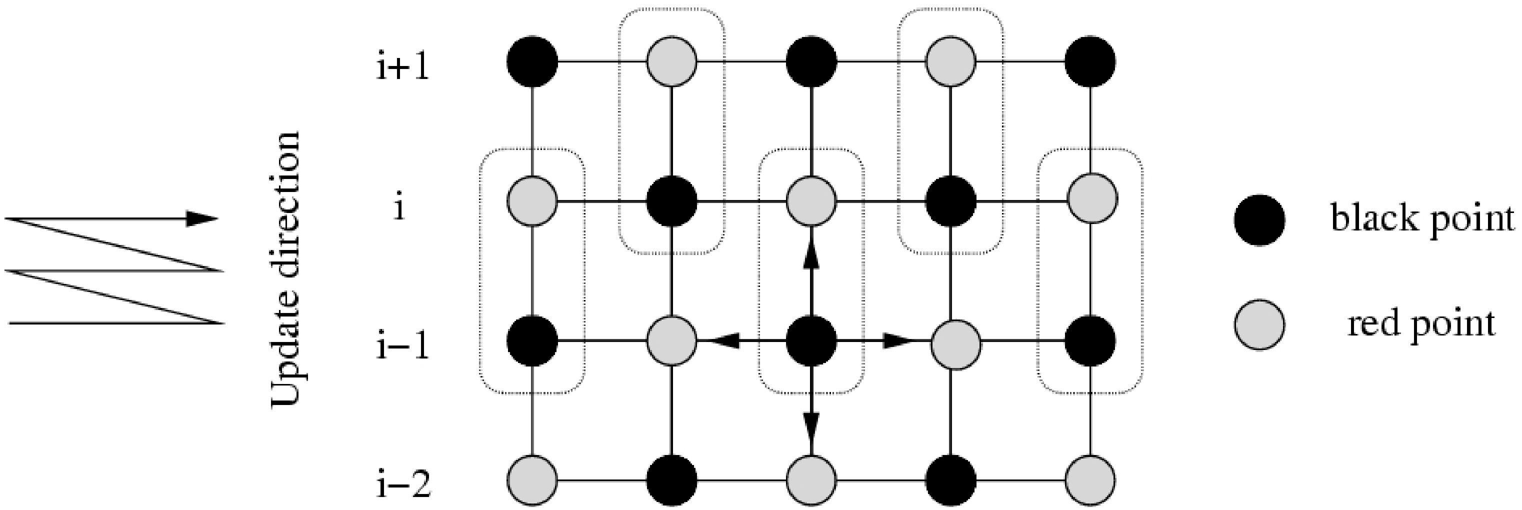
Data Access Optimizations: Loop Fusion

```
  // Black nodes
  for i= 1 to n-1 do
    for j= 1+i%2 to n-1 by 2 do
      relax(u(j,i))
    end for
  end for
end for
```

This requires two sweeps through the whole data set per single GS iteration!

Data Access Optimizations: Loop Fusion

How the fusion technique works:



Data Access Optimizations: Loop Fusion

Code **after** applying loop fusion technique:

```
for it= 1 to numIter do
  // Update red nodes in first grid row
  for j= 1 to n-1 by 2 do
     relax(u(j,1))
  end for
```

Data Access Optimizations: Loop Fusion

```
// Update red and black nodes in pairs
for i= 1 to n-1 do
    for j= 1+(i+1)%2 to n-1 by 2 do
        relax(u(j,i))
        relax(u(j,i-1))
    end for
end for
```

Data Access Optimizations: Loop Fusion

```
// Update black nodes in last grid row
   for j= 2 to n-1 by 2 do
     relax(u(j,n-1))
   end for
```

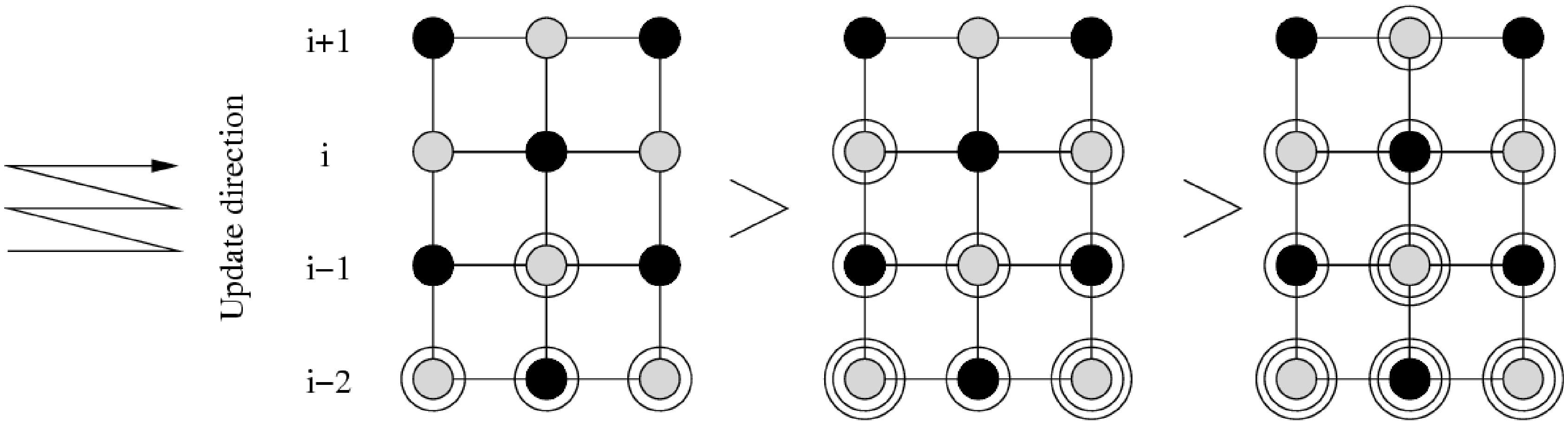
Solution vector u passes through the cache only once instead of twice per GS iteration!

Data Access Optimizations: Loop Blocking

- *Loop blocking = loop tiling*
- Divide the data set into subsets (blocks) which are small enough to fit in cache
- Perform as much work as possible on the data in cache before moving to the next block
- This is not always easy to accomplish because of data dependencies

Data Access Optimizations: Loop Blocking

Example: 1D blocking for red/black GS, respect the data dependencies!



Data Access Optimizations: Loop Blocking

- Code **after** applying 1D blocking technique
- B = number of GS iterations to be blocked/combined

```
for it= 1 to numIter/B do
  // Special handling: rows 1, …, 2B-1
  // Not shown here …
```

Data Access Optimizations: Loop Blocking

```
// Inner part of the 2D grid
for k= 2*B to n-1 do
   for i= k to k-2*B+1 by -2 do
      for j= 1+(k+1)%2 to n-1 by 2 do
         relax(u(j,i))
         relax(u(j,i-1))
      end for
   end for
end for
```

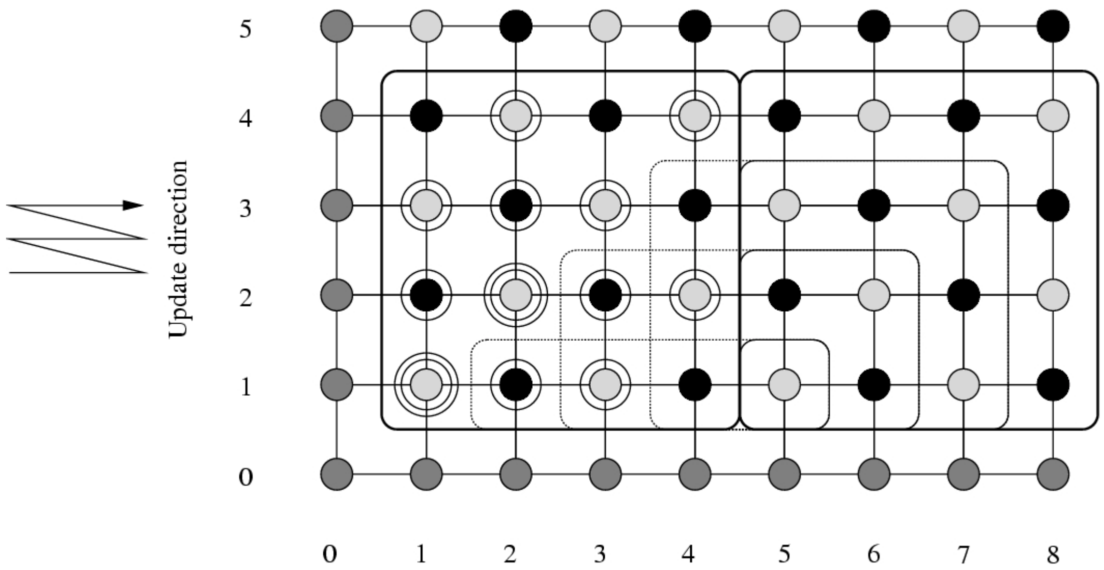
Data Access Optimizations: Loop Blocking

```
  // Special handling: rows n-2B+1, …, n-1
  // Not shown here …
end for
```

- Result: Data is loaded once into the cache per B Gauss-Seidel iterations, if 2*B+2 grid rows fit in the cache simultaneously
- If grid rows are too large, 2D blocking can be applied

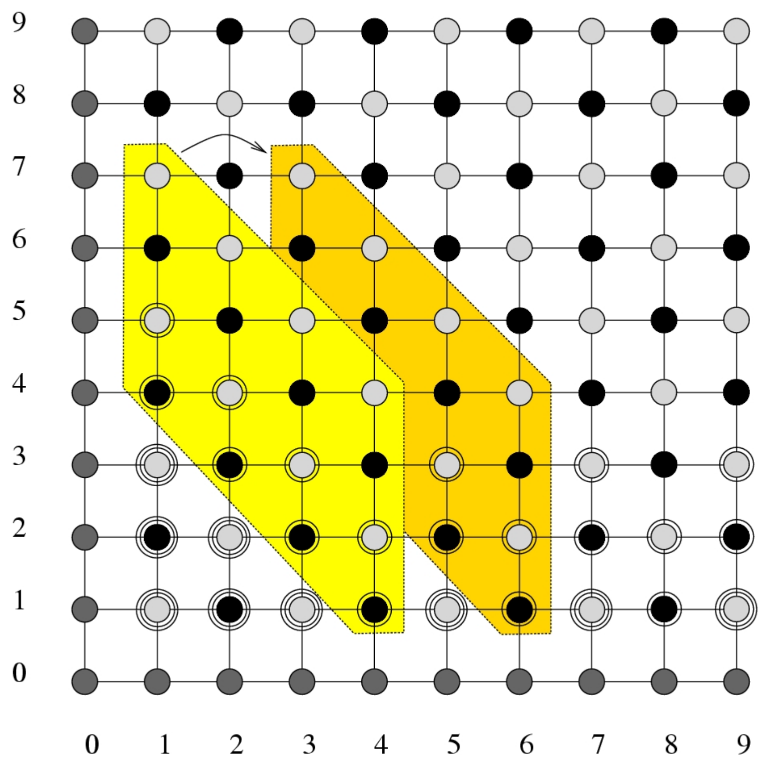
Data Access Optimizations: Loop Blocking

- More complicated blocking schemes exist
- Illustration: 2D square blocking



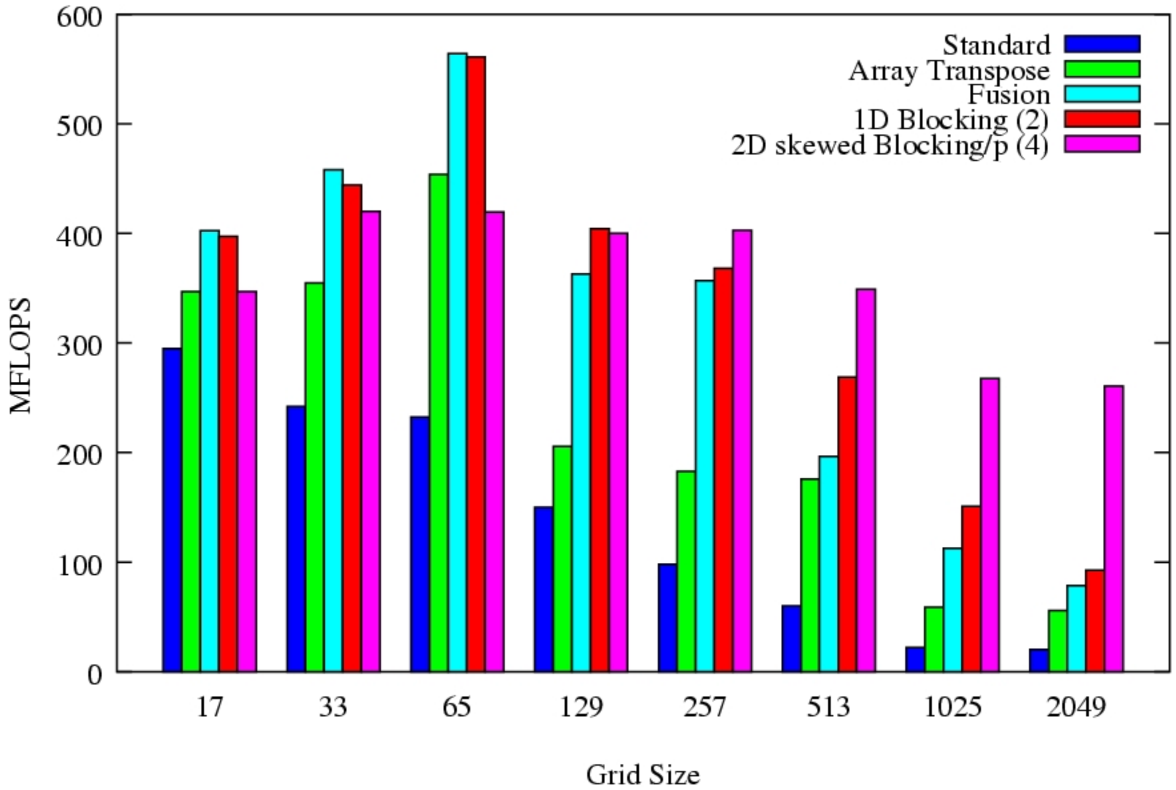
Data Access Optimizations: Loop Blocking

- Illustration: 2D skewed blocking



Performance Results:| Overview of MFLOPS Rates

Digital PWS 500au, Alpha 21164, 500 MHz



Performance Results: Memory Access Behavior

- Digital PWS 500au, Alpha 21164 CPU
- L1 = 8 KB, L2 = 96 KB, L3 = 4 MB
- We use DCPI to obtain the performance data
- We measure the percentage of accesses which are satisfied by each individual level of the memory hierarchy
- Comparison: standard implementation of red/black GS (efficient loop ordering) vs. 2D skewed blocking (with and without padding)

Memory Access Behavior

- Standard implementation of red/black GS, without array padding

Size	+/-	L1	L2	L3	Mem.
33	4.5	63.6	32.0	0.0	0.0
65	0.5	75.7	23.6	0.2	0.0
129	-0.2	76.1	9.3	14.8	0.0
257	5.3	55.1	25.0	14.5	0.0
513	3.9	37.7	45.2	12.4	0.8
1025	5.1	27.8	50.0	9.9	7.2
2049	4.5	30.3	45.0	13.0	7.2

Memory Access Behavior

- 2D skewed blocking without array padding, 4 iterations blocked (B = 4)

Size	+/-	**L1**	**L2**	**L3**	**Mem.**
33	27.4	43.4	29.1	0.1	0.0
65	33.4	46.3	19.5	0.9	0.0
129	36.9	42.3	19.1	1.7	0.0
257	38.1	34.1	25.1	2.7	0.0
513	38.0	28.3	27.0	6.7	0.1
1025	36.9	24.9	19.7	17.6	0.9
2049	36.2	25.5	0.4	36.9	0.9

Memory Access Behavior

- 2D skewed blocking with appropriate array padding, 4 iterations blocked (B = 4)

Size	+/-	L1	L2	L3	Mem.
33	28.2	66.4	5.3	0.0	0.0
65	34.3	55.7	9.1	0.9	0.0
129	37.5	51.7	9.0	1.9	0.0
257	37.8	52.8	7.0	2.3	0.0
513	38.4	52.7	6.2	2.4	0.3
1025	36.7	54.3	6.1	2.0	0.9
2049	35.9	55.2	6.0	1.9	0.9

Cache Optimized Multigrid on Simple Grids: DiMEPACK Library

- DiME: Data-local iterative methods
- Fast algorithm + fast implementation
- Correction scheme: V-cycles, FMG
- Rectangular domains
- Constant 5-/9-point stencils
- Dirichlet/Neumann boundary conditions

DiMEPACK Library

- C++ interface, fast Fortran77 subroutines
- Direct solution of the problems on the coarsest grid (LAPACK: LU, Cholesky)
- Single/double precision floating-point arithmetic
- Various array padding heuristics (Tseng)
- http://wwwbode.in.tum.de/Par/arch/cache

How to Use DiMEPACK

```
void DiMEPACK_example(void) {
    // Initialization of various multigrid/DiMEPACK parameters:
    const int nLevels=7, maxIt= 5, size= 1025, nu1= 1, nu2= 2;
    const tNorm nType= L2;
    const DIME_REAL epsilon= 1e-12, h= 1.0/(size-1);
    const tRestrict rType= FW;
    const DIME_REAL omega= 1.0;
    const bool fixSol= TRUE;
    // Grid objects:
    dpGrid2d u(size, size, h, h), f(size, size, h, h);
    // Initialize u, f here …
```

How to Use DiMEPACK

```
const int nCoeff= 5;
const DIME_REAL hSq= h*h;
DIME_REAL *matCoeff= new DIME_REAL[nCoeff];

// Matrix entries: -Laplace operator:
matCoeff[0]= -1.0/hSq;
matCoeff[1]= -1.0/hSq;
matCoeff[2]= 4.0/hSq;
matCoeff[3]= -1.0/hSq;
matCoeff[4]= -1.0/hSq;
```

How to Use DiMEPACK

```
// Specify boundary types:
tBoundary bTypes[4];
for(i= 0; i<4; i++) {
        bTypes[i]= DIRICHLET;
}

// Specify boundary values:
DIME_REAL **bVals= new DIME_REAL*[4];
bVals[dpNORTH]=     new DIME_REAL[size];
bVals[dpSOUTH]=     new DIME_REAL[size];
bVals[dpEAST]=      new DIME_REAL[size];
bVals[dpWEST]=      new DIME_REAL[size];
```

How to Use DiMEPACK

```
for(i= 0; i<size; i++) {
bVals[dpNORTH][i]= 0.0;        bVals[dpSOUTH][i]= 0.0;
bVals[dpEAST][i]= 0.0;         bVals[dpWEST][i]= 0.0;
}

// Call the DiMEPACK library function, here FMG:
dpFMGVcycleConst(nLevels, nType, epsilon, 0, (&u), (&f), nu1, nu2,
1, nCoeff, matCoeff, bTypes, bVals, rType, omega, fixSol);

// Now, grid object u contains the solution
// "delete" the arrays allocated using "new" here …
}
```

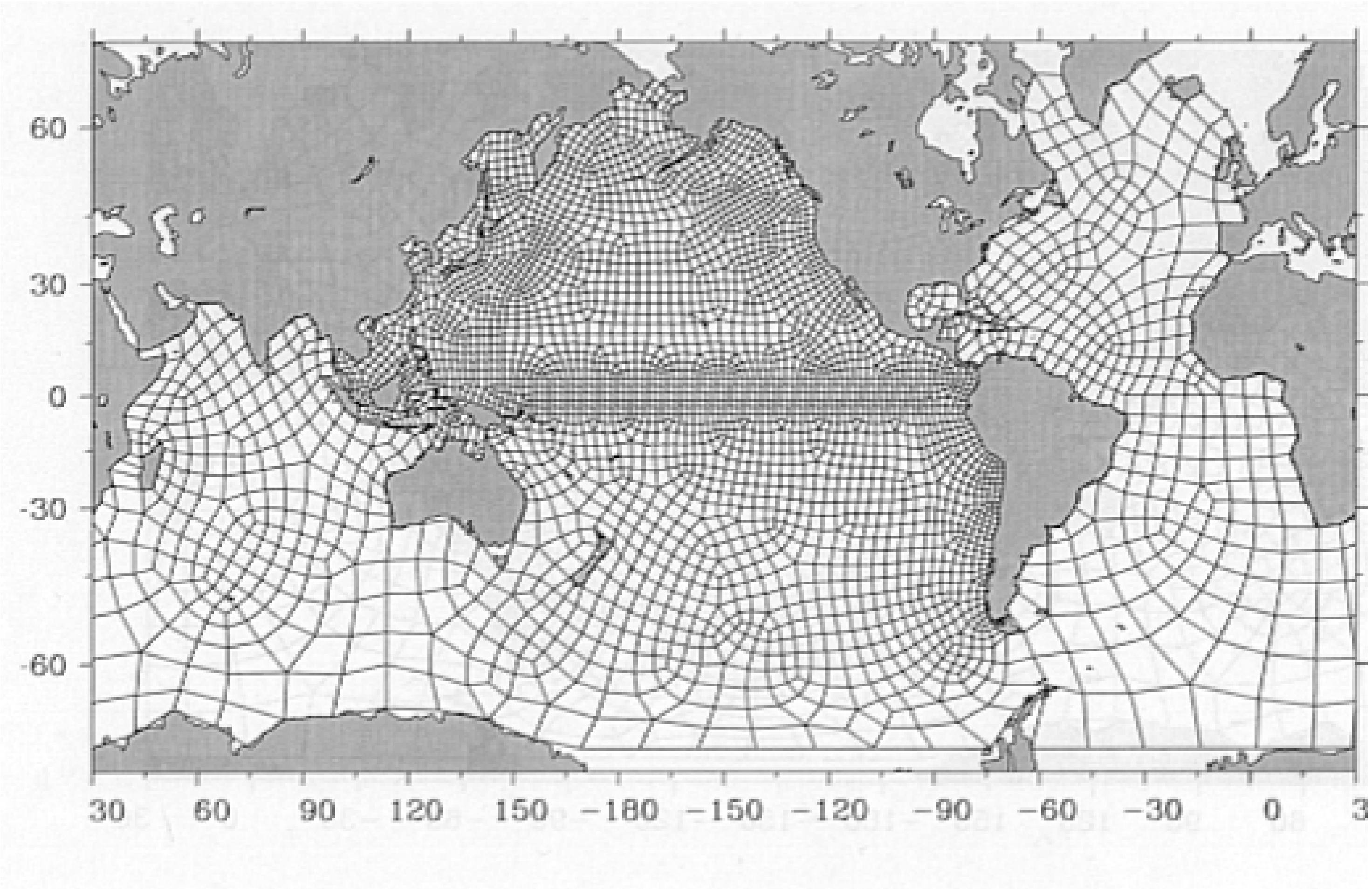
Vcyle(2,2) Bottom Line

Mflops	For what
13	Standard 5-pt. Operator
56	Cache optimized (loop orderings, data merging, simple blocking)
150	Constant coeff. + skewed blocking + padding
220	Eliminating rhs if 0 everywhere but boundary

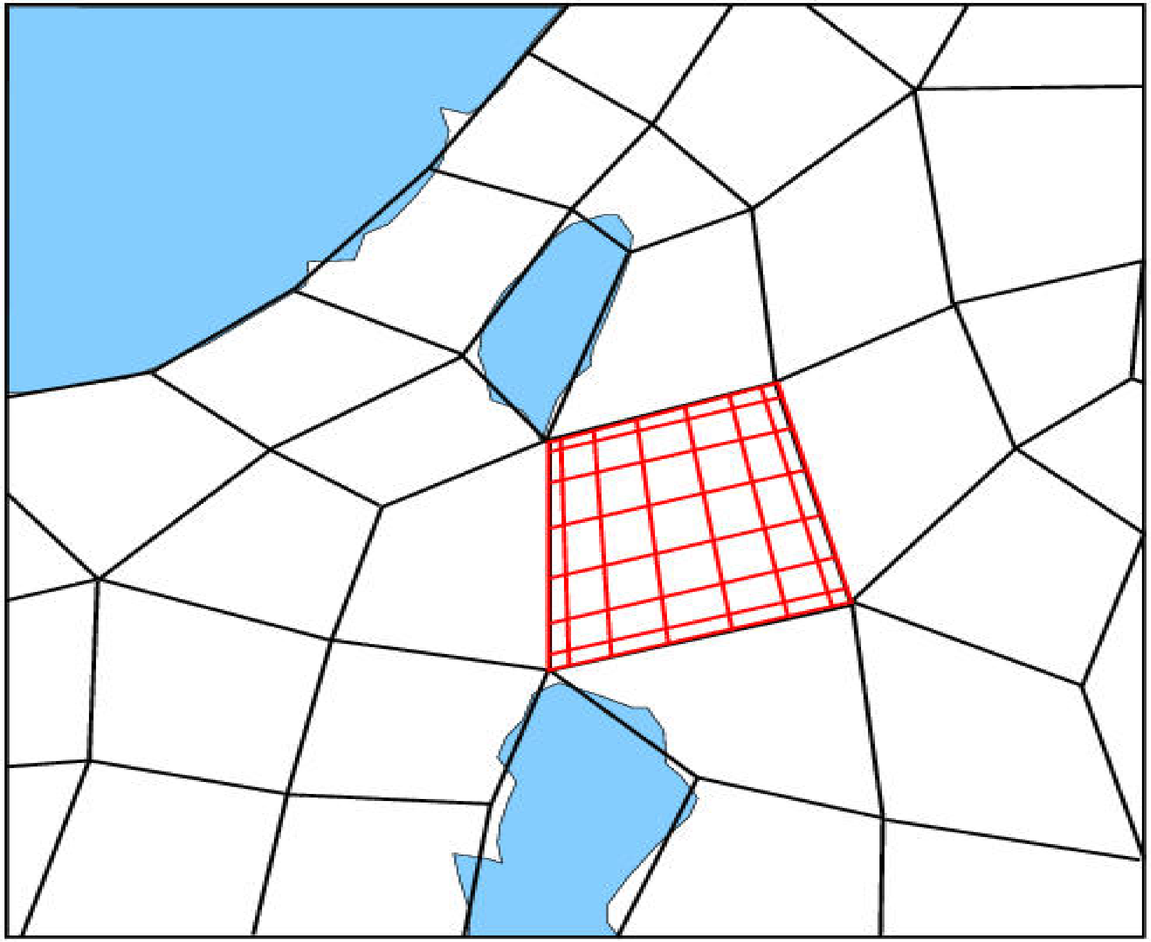
Part III
Unstructured Grid Computations

- How unstructured is the grid?
- Sparse matrices and data flow analysis
- Grid processing
- Algorithm processing
- Examples

Is It Really Unstructured?



Subgrids and Patches



Sparse Matrix Connection

- How the data is stored is crucial since *Ax=b* gets solved eventually assuming a matrix is stored.
- In row storage format, we have 2-3 vectors that represent the matrix (row indices, column indices, and coefficients). The column and coefficients are picked up one at a time (1x1). Picking up multiple ones at a time is far more efficient: 1x2, 2x2, …, *n*x*m*. The best configuration depends on the problem. Ideally this is a parameter to a sparse matrix code. Unfortunately, compilers do better with explicit values for *n* and *m* a priori. The right choice leads to 50% improvement for many PDE problems (Sivan Toledo, Alex Pothen).

Sparse Matrix Connection

- Store the right hand side in the matrix. (Even in Java you see a speedup.)
- Combine vectors into matrices if they can be accessed often in the same statements.
  - r1(i) = r2(i) + r3(i)
  - r(1,i) = r(2,i) + r(3,i)

  The first form has 3 cache misses at a time. The second form has only 1 cache miss at a time and is usually 3 times faster than the first form.

Small Block BLAS

- Most BLAS are optimized for really big objects. Typical Level 3 BLAS are for 40x40 or 50x50 and up.
- Hand coded BLAS for small data objects provide a 3-4X speedup.
- G. Karniadakis had students produce one for the IBM Power2 platform a few years ago. He was given a modest SP2 as a thank you by the CEO of IBM.

Data Flow Analysis

- Look at an algorithm on paper. See if statements can be combined at the vector element level.
- In conjugate gradients,

  $x(\mathrm{i}) = x(\mathrm{i}) + alpha*(\text{row of } A)*w$

  $r(\mathrm{i}) = r(\mathrm{i}) - alpha*w(\mathrm{i})$

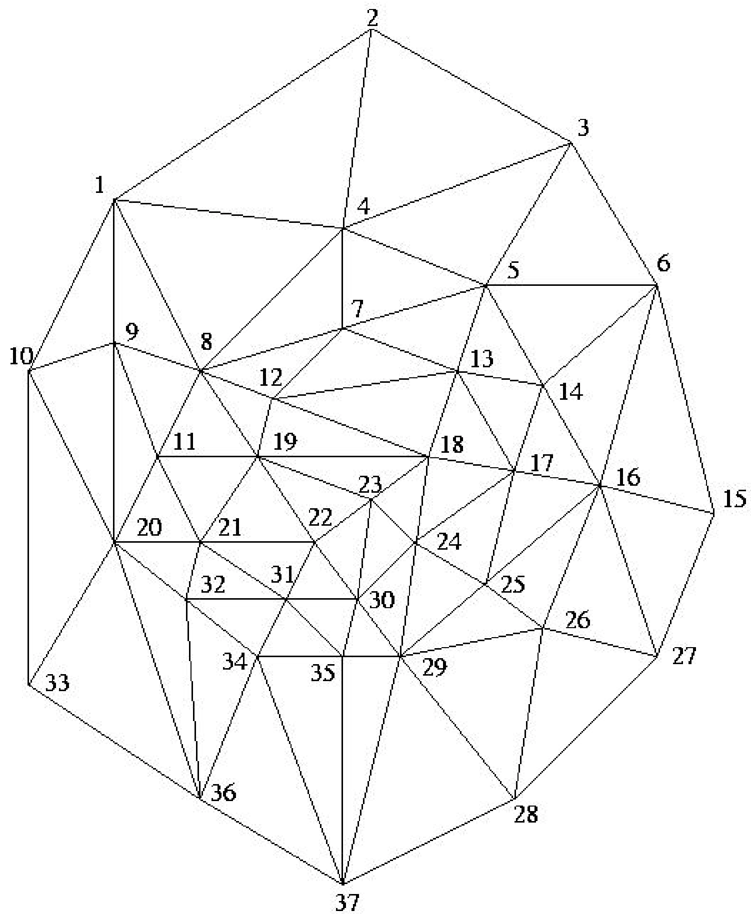
  $beta = beta + r(\mathrm{i})*r(\mathrm{i})$

  Combine vectors into a single matrix and write a loop that updates all 3 variables at once, particularly since *A*'s main diagonal is nonzero so *w*(i) will be accessed during *x*(i) update. Use a *n*x*m* *A* sparse matrix storage scheme, too.

Grid Preprocessing

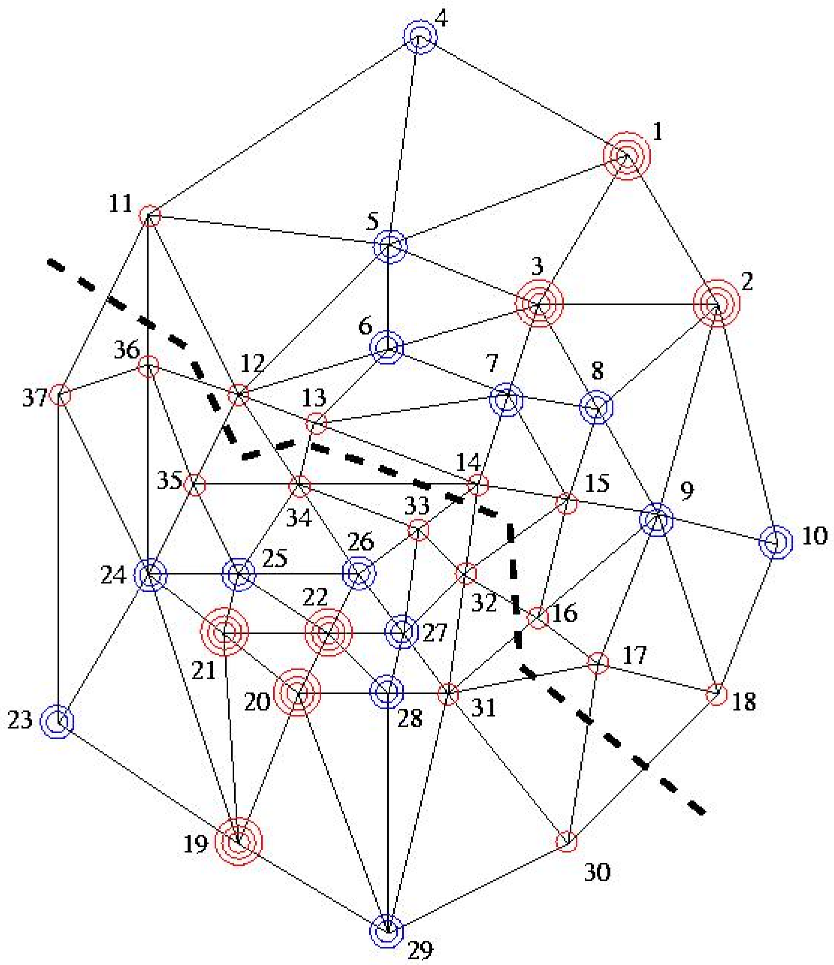
- Look for quasi-unstructured situation. In the ocean grid, almost all of the grid vertices have a constant number of grid line connections. This leads to large sections of a matrix with a constant and predictable set of graph connections. This should be used to our advantage.
- Specific code that uses these subgrids of structuredness leads to much, much faster code.
- Do not trust your compiler to do this style of optimization: it simply will not happen since it is a runtime effect.
- You must do this coding as spaghetti style code yourself.

Motivating Example for Multigrid



- Suppose problem information for only half of nodes fits in cache.
- Gauss-Seidel updates nodes in order
- Leads to poor use of cache
  - By the time node 37 is updated, information for node 1 has probably been evicted from cache.
  - Each unknown must be brought into cache at each iteration.

Motivating Example for Multigrid



- Alternative
  - Divide into two connected subsets.
  - Renumber
  - Update as much as possible within a subset before visiting the other.
- Leads to better data reuse within cache.
- Some unknowns can be completely updated.
- Some partial residuals can be calculated.

Cache Aware Gauss-Seidel

- Preprocessing phase
  - Decompose each mesh into disjoint cache blocks.
  - Renumber the nodes in each block.
  - Produce the system matrices and intergrid transfer operators with the new ordering.
- Gauss-Seidel phase
  - Update as much as possible within a block without referencing data from another block.
  - Calculate a (partial) residual on the last update.
  - Backtrack to finish updating cache block boundaries.

Preprocessing: Mesh Decomposition

- Goals:
  - maximize interior of cache block.
  - minimize connections between cache blocks.
- Constraint:
  - Cache should be large enough to hold the part of matrix, right hand side, residual, and unknown associated with a cache block.
- Critical parameter: cache size
- Such decomposition problems have been studied in load balancing for parallel computation.

Preprocessing on a Triangular Mesh: Renumbering within a Cache Block

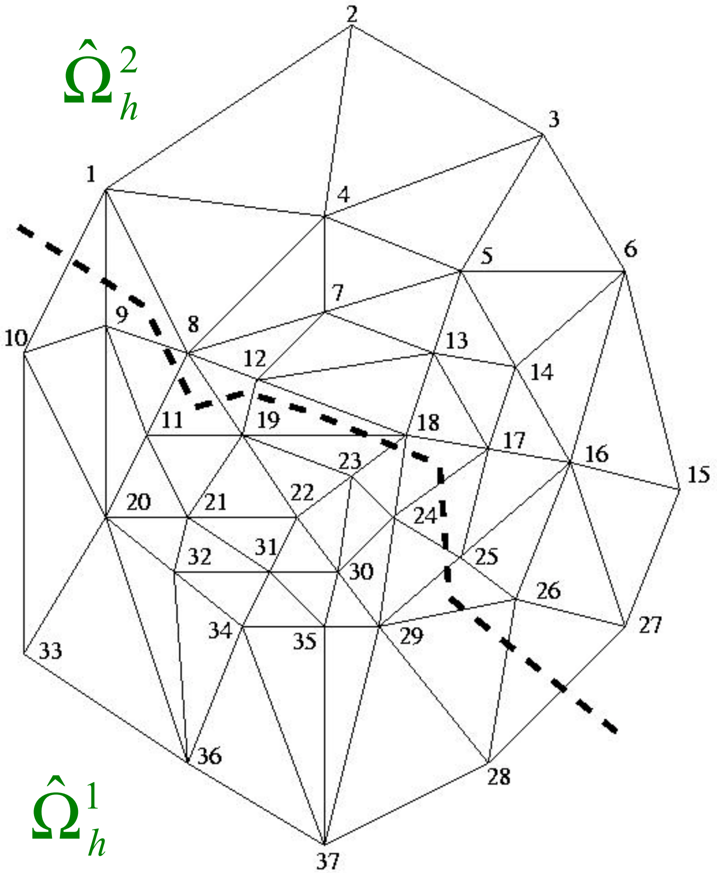
- Let m be the number of smoothing sweeps.
- Find distance (shortest path) between each node and cache block boundary.
  - Subblock L_j^s, $j < m$, is the set of all nodes that are distance j from cache block boundary.
  - Subblock L_m^s is the remainder of the cache block nodes.
    - Should contain majority of the cache block nodes.
- Renumber the nodes in $L_m^s, \ldots, L_1^s$ in that order.
  - Contiguous within cache blocks.
  - Contiguous within subblocks.

Preprocessing on a Triangular Mesh: Renumbering within a Cache Block

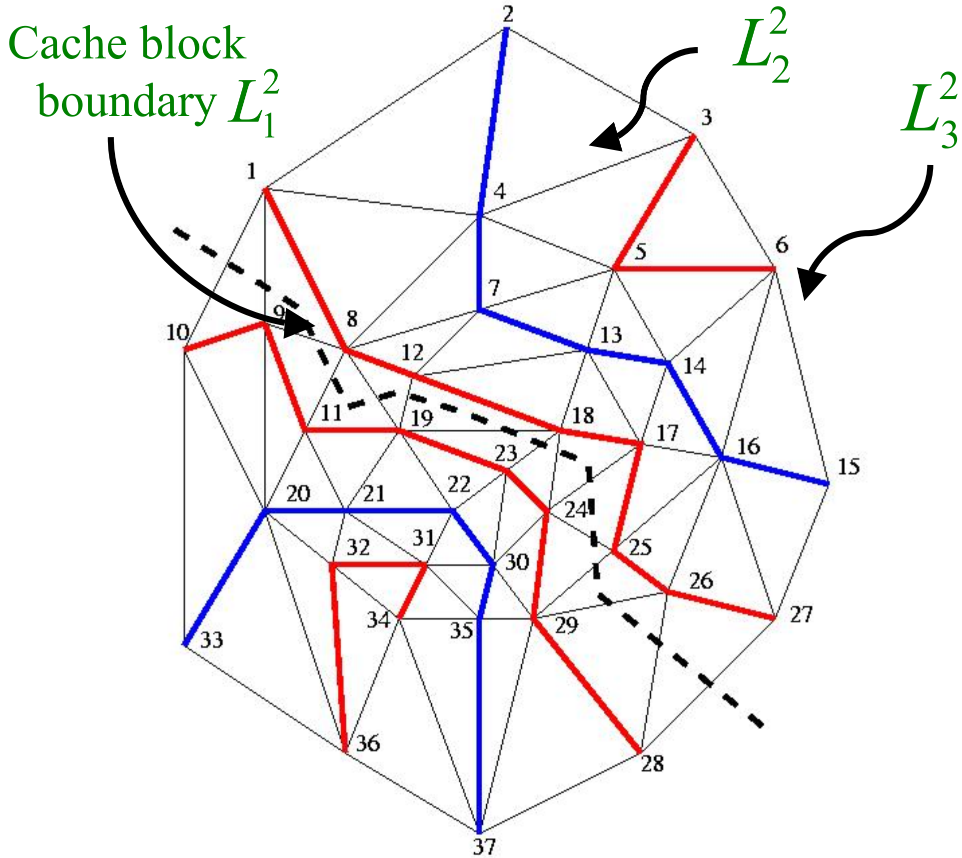
- Denote cache block by $\hat{\Omega}_h^s$.
- Find distance (shortest path) between each node and cache block boundary $\partial\hat{\Omega}_h^s$.
  - Define subblock L_j^s , $j < m$, to be the set of all nodes that are distance j from $\partial\hat{\Omega}_h^s$.
  - Let $L_m^s = \hat{\Omega}_h^s \setminus \bigcup_{j<m} L_j^s$, where m is number of smoothing sweeps.
  - Note that $L_1^s = \partial\hat{\Omega}_h^s$.
- Renumber the nodes in $L_m^s, \ldots, L_1^s$ in that order.
  - Contiguous within cache blocks and within subblocks.
- L_m^s should contain majority of the cache block nodes.

Example of Subblock Membership

Cache blocks identified



Subblocks identified



Algorithm 1 (unknown physical boundary nodes) Mark cache boundary nodes.

Mark-Boundary-Nodes

Initialize stacks S_1 and S_2.
Set distance $D_i = 0$ for all i in Ω_s.
for each node i in Ω_s such that $D_i == 0$ **do**
  if i is on $\partial\Omega_s$ **then**
    Push i onto S_1.
    while S_1 is not empty **do**
      Pop node i off S_1.
      if i is in $\partial\Omega_s$ **then**
        Set $D_i = 1$.
        for each node j connected to i **do**
          if $(D_j == 0)$ AND (j is in Ω_s) **then**
            Set $D_j = -1$.
            Push j onto S_1.
          end if
        end for
      else
        Set $D_i = 2$.
        Push i onto S_2.
      end if
    end while
  end if
end for

Algorithm 1 Mark cache interior nodes.

Mark-Interior-Nodes

```
Set current distance c = 2.
Let m be the number of Gauss-Seidel updates desired.
while S_2 is not empty do
    if c < m then
        Set current distance c = c + 1.
    end if
    Move contents of S_2 to S_1.
    while S_1 is not empty do
        Pop node i off S_1.
        for each node j adjacent to i do
            if distance D_j == 0 then
                Set distance D_j = c.
                Push j onto S_2.
            end if
        end for
    end while
end while
```

Two Common Multigrid Algorithms

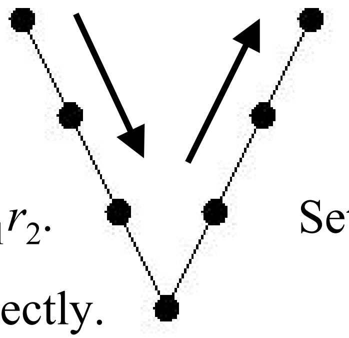
V Cycle to solve $A_4 u_4 = f_4$

Smooth $A_4 u_4 = f_4$. Set $f_3 = R_3 r_4$.

Smooth $A_3 u_3 = f_3$. Set $f_2 = R_2 r_3$.

Smooth $A_2 u_2 = f_2$. Set $f_1 = R_1 r_2$.

Solve $A_1 u_1 = f_1$ directly.

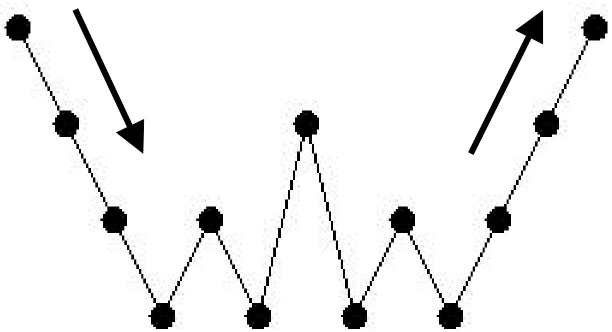


Set $u_4 = u_4 + I_3 u_3$. Smooth $A_4 u_4 = f_4$.

Set $u_3 = u_3 + I_2 u_2$. Smooth $A_3 u_3 = f_3$.

Set $u_2 = u_2 + I_1 \mathrm{u}_1$. Smooth $A_2 u_2 = f_2$.

W Cycle



Preprocessing Costs of Distance Algorithms

- Goal: Determine upper bound on work to find distances of nodes from cache block boundary.
- Assumptions
  - Two dimensional triangular mesh.
  - No hints as to where the cache block boundaries are.

Symbol	Definition
N	Number of nodes
K	Average number of connections per node
d	Degrees of freedom per node
m	Number of smoothing sweeps

Preprocessing Costs: Work Estimates

Cache aware Gauss-Seidel:

$$C_{cags} \le \begin{cases} 10NK + 6N & \text{if } m = 2 \\ \dfrac{(7K+2)N}{m-1} + 3NK + 4N & \text{if } m > 2 \end{cases}$$

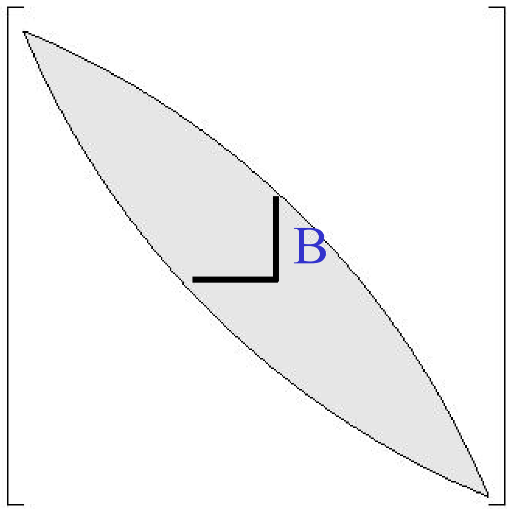
Standard Gauss-Seidel: $C_{gs} \le 2d^2NK$

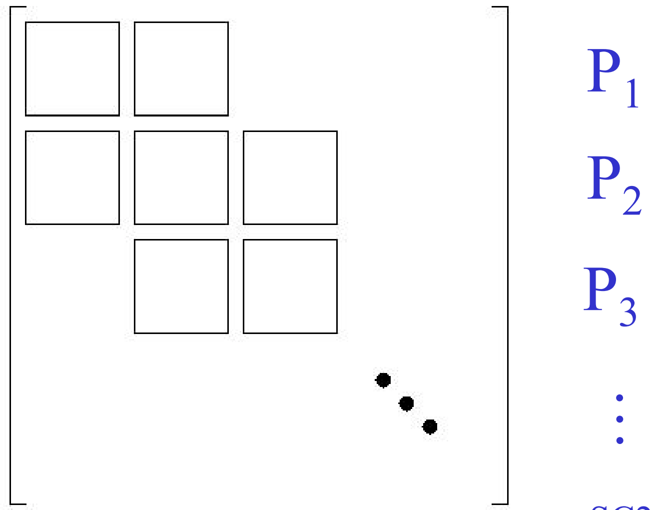
How do these constants compare?

Dof	Equivalent number of fine grid sweeps
1	5
2	Less than one

Multigrid with Active Sets

- Another cache aware method with Gauss-Seidel smoothing.
  - Reduce matrix bandwidth to size B.
  - Partition rows into sets P_1, P_2, …, each containing B rows.





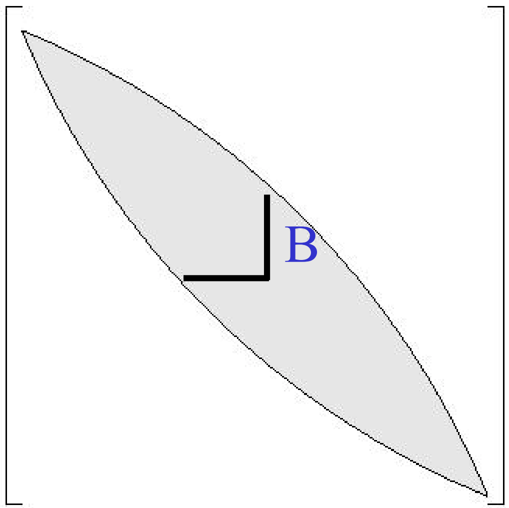
Multigrid with Active Sets

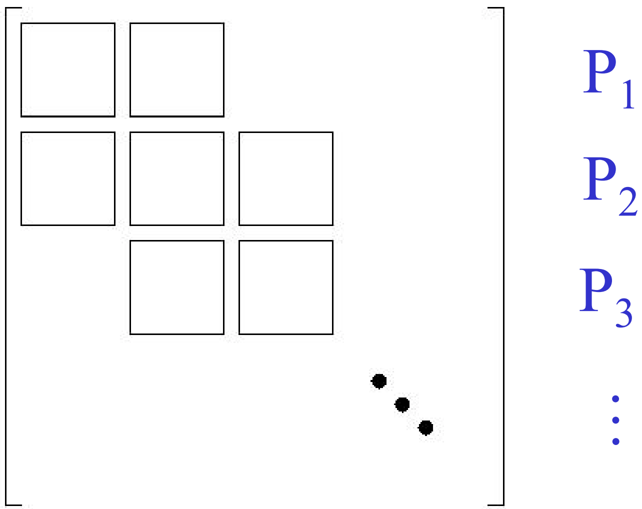
- Let $u(P_i)$ = number of updates performed on unknowns in P_i.
- Key point: As soon as $u(P_{i-1}) = k$, $u(P_i) = k - 1$, and $u(P_{i+1}) = k-1$, unknowns in P_i can be updated again.
- For m updates: if mB rows fit in cache, then all unknowns can be updated with one pass through cache.

Multigrid with Active Sets

– Example: For $m = 3$, the following schedule updates all unknowns:

$P_1, P_2, P_1, P_3, P_2, P_1, P_4, P_3, P_2, \ldots, P_n, P_{n-1}, P_{n-2}, P_n, P_{n-1}, P_n$





Numerical Results: Hardware

	SGI O2	HP PA 8200
CPU	300 MHz MIPS ip32 R12000	200 MHz HP PA 8200
Memory	128 MB	2150 MB
L1 cache	32 KB split, 2-way associative, 8 byte lines	1 MB split, direct mapped, 32 byte lines
L2 cache	1 MB unified, 2-way associative	Nada! Zip! None!!!

Numerical Results

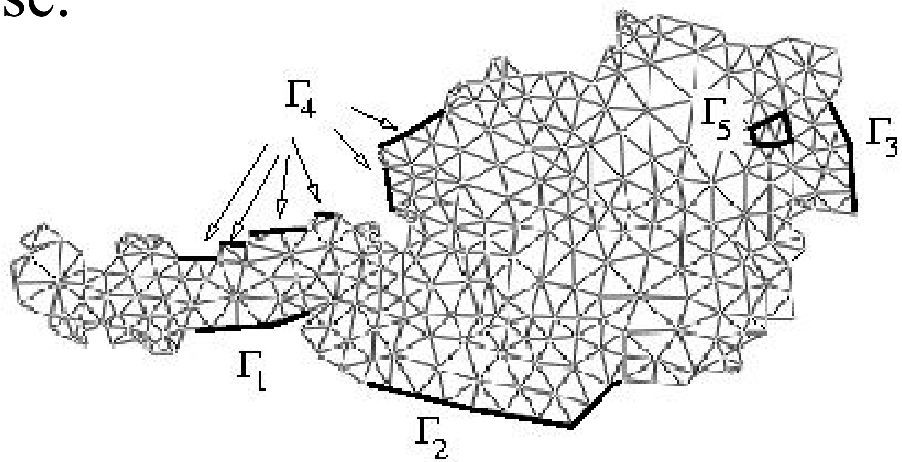
Experiment: Austria

Two dimensional elasticity

$\boldsymbol{T}$ = Cauchy stress tensor

$\boldsymbol{w}$ = displacement

$$\boldsymbol{f} = \begin{cases} (1,-1)^{\mathrm{T}} & \text{on } \Gamma_4, \\ (9.5-x, 4-y) & \text{if } (x,y) \text{ is in the region surrounded by } \Gamma_5, \\ 0 & \text{otherwise.} \end{cases}$$



$-\nabla \boldsymbol{T} = \boldsymbol{f}$ in Ω,

$\partial \boldsymbol{w}/\partial n = 100w$ on Γ_1,

$\partial \boldsymbol{w}/\partial y = 100w$ on Γ_2,

$\partial \boldsymbol{w}/\partial x = 100w$ on Γ_3,

$\partial \boldsymbol{w}/\partial n = 0$ everywhere else.

5 Level V Cycle Times: SGI O2

	V(2,2)	V(3,3)	V(4,4)
Standard	3.28	4.19	5.22
Cache blocking	1.99	2.21	2.71
Active set	1.83	2.11	2.71
Cache blocking Speedup	1.64	1.90	2.04
Active set Speedup	1.78	1.90	1.93

Austria

5 Level V Cycle Times: HP PA 8200

	V(2,2)	V(3,3)	V(4,4)
Standard	1.88	2.34	2.84
Cache blocking	0.79	0.92	1.27
Active set	0.79	0.94	1.06
Cache blocking Speedup	2.38	2.55	2.78
Active set Speedup	2.38	2.50	2.68

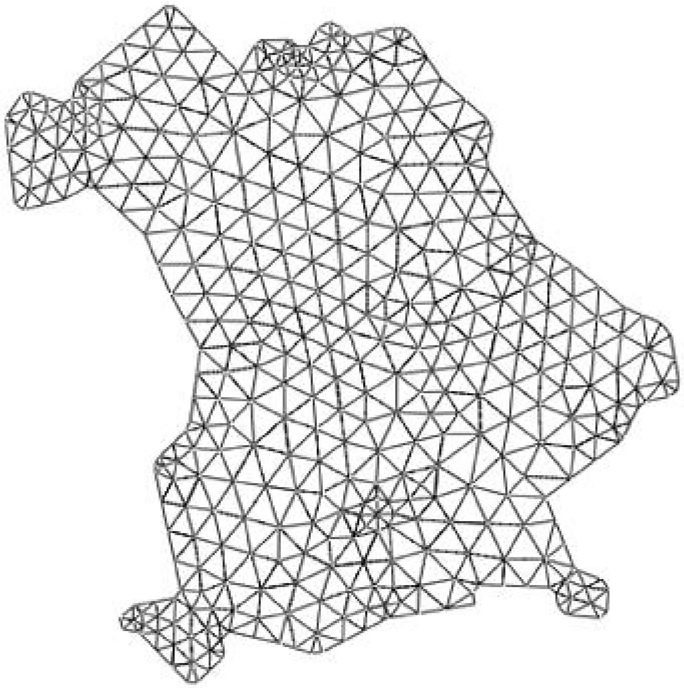
Austria

Numerical Experiments

Experiment: Bavaria
Stationary heat equation
with 7 sources and one sink
(Munich Oktoberfest).

Homogeneous Dirichlet boundary conditions on the Czech border (northeast), homogeneous Neumann b.c.'s everywhere else.

Coarse grid mesh



5 Level V Cycle Times: SGI O2

	V(2,2)	V(3,3)	V(4,4)
Standard	1.46	1.92	2.32
Cache blocking	0.87	1.08	1.16
Cache blocking Speedup	1.67	1.77	2.00

Bavaria

5 Level V Cycle Times: HP PA 8200

	V(2,2)	V(3,3)	V(4,4)
Standard	0.743	0.921	1.09
Cache blocking	0.447	0.4845	0.529
Cache blocking Speedup	1.66	1.90	2.05

Bavaria

Summary for Part III

- In problems where unstructured grids are reused many times, cache aware multigrid provides very good speedups.
- Speedups of 20–175% are significant for problems requiring significant CPU time.
- If you run a problem only once in 0.2 seconds, do not bother with this whole exercise.
- Our cache aware multigrid implementation is not tuned for a particular architecture. In particular, the available cache size is the *only* tuning parameter.
- Google search: sparse matrix AND cache.

References

- W. L. Briggs, V. E. Henson, and S. F. McCormick, A Multigrid Tutorial, SIAM Books, Philadelphia, 2000.
- C. C. Douglas, Caching in with multigrid algorithms: problems in two dimensions, Parallel Algorithms and Applications, 9 (1996), pp. 195-204.
- C. C. Douglas, G. Haase, J. Hu, W. Karl, M. Kowarschik, U. Rüde, C. Weiss, Portable memory hierarchy techniques for PDE solvers, Part I, SIAM News 33/5 (2000), pp. 1, 8-9. Part II, SIAM News 33/6 (2000), pp. 1, 10-11, 16.
- C. C. Douglas, G. Haase, and U. Langer, A Tutorial on Parallel Solution Methods for Elliptic Partial Differential Equations, 2001, see Douglas' preprint web page.

References

- C. C. Douglas, J. Hu, M. Iskandarani, Preprocessing costs of cache based multigrid, in Proceedings of the Third European Conference on Numerical Mathematics and Advanced Applications, P. Neittaanmäki, T. Tiihonen, and P. Tarvainen (eds.), World Scientific, Singapore, 2000, pp. 362-370.
- C. C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rüde, C. Weiss, Maximizing cache memory usage for multigrid algorithms, in Multiphase Flows and Transport in Porous Media: State of the Art, Z. Chen, R. E. Ewing, and Z.-C. Shi (eds.), Lecture Notes in Physics, Vol. 552, Springer-Verlag, Berlin, 2000, pp. 234-247.

References

- C. C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Ruede, C. Weiss, Cache optimization for structured and unstructured grid multigrid, Electronic Transactions on Numerical Analysis, 10 (2000), pp. 25-40.
- C. C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Ruede, C. Weiss, Fixed and adaptive cache aware algorithms for multigrid methods, in Multigrid Methods VI, E. Dick, K. Riemslagh, and J. Vierendeels (eds.), Lecture Notes in Computational Sciences, Springer-Verlag, Berlin, 2000, pp. 87-93.

References

- W. Hackbusch, Iterative Solution of Large Sparse Systems of Equations, Applied Mathematical Sciences series, vol. 95, Springer-Verlag, Berlin, 1994.
- J. Handy, The Cache Memory Book, 2^{nd} ed., Academic Press, 1998.
- J. L. Hennesey and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2^{nd} ed., Morgan Kauffmann Publishers, 1996.

References

- M. Kowarschik, C. Weiss, DiMEPACK – A Cache-Optimized Multigrid Library, Proc. of the Intl. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), vol. I, June 2001, pp. 425-430.
- J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, Thread scheduling for cache locality, in Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, October 1996, pp. 60-73.

References

- U. Ruede, Iterative Algorithms on High Performance Architectures, Proc. of the EuroPar97 Conference, Lecture Notes in Computer Science series, Springer-Verlag, Berlin, 1997, pp. 57-71.
- U. Ruede, Technological Trends and their Impact on the Future of Supercomputing, H.-J. Bungartz, F. Durst, C. Zenger (eds.), High Performance Scientific and Engineering Computing, Lecture Notes in Computer Science series, Vol. 8, Springer, 1998, pp. 459-471.

References

- U. Trottenberg, A. Schuller, C. Oosterlee, Multigrid, Academic Press, 2000.
- C. Weiss, W. Karl, M. Kowarschik, U. Ruede, Memory Characteristics of Iterative Methods. In Proceedings of the Supercomputing Conference, Portland, Oregon, November 1999, pp. ?-?.
- C. Weiss, M. Kowarschik, U. Ruede, and W. Karl, Cache-aware Multigrid Methods for Solving Poisson's Equation in Two Dimension. Computing, 64(2000), pp. 381-399.

Related Web Sites

- http://www.mgnet.org
- http://www.ccs.uky.edu/~douglas/ccd-kfcs.html
- http://wwwbode.in.tum.de/Par/arch/cache
- http://www.kfa-juelich.de/zam/PCL
- http://icl.cs.utk.edu/projects/papi
- http://www.tru64unix.compaq.com/dcpi