Very High Performance Cache Based Techniques for Iterative Methods

Craig C. Douglas
University of Kentucky and Yale University

Jonathan J. Hu
Sandia National Laboratories

Ulrich Rüde and Markus Kowarschik
Lehrstuhl für Systemsimulation (Informatik 10)
Universität Erlangen-Nürnberg
E-mail contacts

- craig.douglas@uky.edu
- craig.douglas@yale.edu
- jhu@sandia.gov
- ulrich.ruede@cs.fau.de
- markus.kowarschik@cs.fau.de
Overview

- Part I: Architectures and Fundamentals
- Part II: Optimization Techniques for Structured Grids
- Part III: Optimization Techniques for Unstructured Grids
Part I

Architectures and Fundamentals
Architectures and fundamentals

- Why worry about performance
  - an illustrative example
- Fundamentals of computer architecture
  - CPUs, pipelines, superscalarity
  - Memory hierarchy
- Basic efficiency guidelines
- Profiling
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 70’s)
- More general elliptic equations may need $O(100)$ operations per unknown
- A modern CPU can do 1-6 GFLOPS
- So we should be solving 10-60 million unknowns per second
- Should need $O(100)$ Mbytes of 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
(what got me started in this business ~ '95)

Unstructured code
Structured grid Multigrid
Optimal Multigrid
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
Pipelining

**sequential execution:**

<table>
<thead>
<tr>
<th>1st instruction</th>
<th>2nd instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

**pipelined execution:**

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
<th>1st instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
<td>2nd instruction</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
<td>3rd instruction</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
<td>4th instruction</td>
</tr>
</tbody>
</table>
Pipelining (cont’d)

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1st instruction

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2nd instruction

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3rd instruction

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4th instruction

**pipelined and superscalar execution:**
CPU trends

- **EPIC** (similar to VLIW) (IA64)
- **Multi-threaded** architectures (Alpha, Pentium4HT)
- Multiple CPUs on a single chip (IBM Power 4)
- Within the next decade
  - Billion transistor CPUs (today 200 million transistors)
  - Potential to build TFLOPS on a chip (e.g., SUN graphics processors)
  - But no way to move the data in and out sufficiently quickly!
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 <10 Gbyte/sec (6.4 Itanium II) 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 locality:** an item referenced now will be referenced again soon
- **Spatial locality:** an item referenced now indicates that neighbors will be referenced soon
- **Cache lines** are typically 32-128 bytes with 1024 being the longest recently. 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, 3, or 4)
- 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**
Typical cache configuration

CPU

Registers

L1 Data Cache

L1 Inst Cache

L2 Cache

L3 Cache

Main Memory
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 that can be read (written) per second
Cache issues (cont’d)

- Cache line size
  - Prefetching effect
  - False sharing (cf. associativity issues)
- Replacement strategy
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)
  - Random
- Translation lookaside buffer (TLB)
  - Stores virtual memory page translation entries
  - Has effect similar to another level of cache
  - TLB misses are very expensive
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

Hit ratios of 90% and better are needed for good speedups
Cache organization

- Number of cache levels
- Set 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 cache block can be stored in exactly one cache line of the cache memory

- **Fully associative**
  - A cache block can be stored in any cache line

- **Set-associative** (associativity = k)
  - Each cache block 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.
Typical architectures

- **IBM Power 3:**
  - L1 = 64 KB, 128-way set associative (funny definition, however)
  - L2 = 4 MB, direct mapped, line size = 128, write back

- **IBM Power 4 (2 CPU/chip):**
  - L1 = 32 KB, 2-way, line size = 128
  - L2 = 1.5 MB, 8-way, line size = 128
  - L3 = 32 MB, 8-way, line size = 512

- **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-RISC:**
  - PA8500, PA8600: L1 = 1.5 MB, PA8700: L1 = 2.25 MB
  - no L2 cache!
Typical architectures (cont’d)

- AMD Athlon (from “Thunderbird” on):
  - L1 = 64 KB, L2 = 256 KB

- Intel Pentium 4:
  - L1 = 8 KB, 4-way, line size = 64
  - L2 = 256 KB up to 2MB, 8-way, line size = 128

- Intel Itanium:
  - L1 = 16 KB, 4-way
  - L2 = 96 KB, 6-way
  - L3: off-chip, size varies

- Intel Itanium² (McKinley / Madison):
  - L1 = 16 / 32 KB
  - L2 = 256 / 256 KB
  - L3: 1.5 or 3 / 6 MB
Basic efficiency guidelines

- Choose the best algorithm
- Use efficient libraries
- Find good compiler options
- Use suitable data layouts
Choose the best algorithm

Example: Solution of linear systems arising from the discretization of a special PDE
- Gaussian elimination (standard): $n^{3/3}$ ops
- Banded Gaussian elimination: $2n^2$ ops
- SOR method: $10n^{1.5}$ ops
- Multigrid method: $30n$ ops
Choose the best algorithm (cont’d)

- For n large, the multigrid method will always outperform the others, even if it is badly implemented
- Frequently, however, two methods have approximately the same complexity, and then the better implemented one will win
Use efficient libraries

Good libraries often outperform own software

- Clever, sophisticated algorithms
- Optimized for target machine
- Machine-specific implementation
Sources for libraries

- Vendor-independent
  - Commercial: NAG, IMSL, etc.; only available as binary, often optimized for specific platform
  - Free codes: e.g., NETLIB (LAPACK, ODEPACK, ...), usually as source code, not specifically optimized
- Vendor-specific; e.g., cxml for HP Alpha with highly tuned LAPACK routines
Sources for libraries (cont’d)

- Many libraries are quasi-standards
  - BLAS
  - LAPACK
  - etc.
- Parallel libraries for supercomputers
- Specialists can sometimes outperform vendor-specific libraries
Find good compiler options

- Modern compilers have numerous flags to select individual optimization options
  - `-On`: successively more aggressive optimizations, n=1,...,8
  - `-fast`: may change round-off behavior
  - `-unroll`
  - `-arch`
  - Etc.

- Learning about your compiler is usually worth it: RTFM (which may be hundreds of pages long).
Find good compiler options (cont‘d)

Hints:

- Read `man cc` (or `man f77`) or `cc --help` (or whatever causes the possible options to print)
- Look up compiler options documented in [www.specbench.org](http://www.specbench.org) for specific platforms
- Experiment and compare performance on your own codes
Use suitable data layout

Access memory in order! In C/C++, for a 2D matrix

```c
double a[n][m];
the loops should be such that
for (i...)
    for (j...)
        a[i][j]...
```

In FORTRAN, it must be the other way round

Apply *loop interchange* if necessary (see below)
Use suitable data layout (cont’d)

Other example: *array merging*

Three vectors accessed together (in C/C++):

```c
double a[n], b[n], c[n];
```

can often be handled more efficiently by using

```c
double abc[n][3];
```

In FORTRAN again indices permuted
Profiling

- Subroutine-level profiling
  - Compiler inserts timing calls at the beginning and end of each subroutine
  - Only suitable for coarse code analysis
  - Profiling overhead can be significant
  - E.g., prof, gprof
Profiling (cont’d)

- Tick-based profiling
  - OS interrupts code execution regularly
  - Profiling tool monitors code locations
  - More detailed code analysis is possible
  - Profiling overhead can still be significant

- Profiling using hardware performance monitors
  - Most popular approach
  - Will therefore be discussed next in more detail
Profiling: hardware performance counters

Dedicated CPU registers are used to count various events at runtime:

- Data cache misses (for different levels)
- Instruction cache misses
- TLB misses
- Branch mispredictions
- Floating-point and/or integer operations
- Load/store instructions
- Etc.
Profiling tools: DCPI

- DCPI = Digital Continuous Profiling Infrastructure (still supported by current owner, HP; source is even available)
- Only for Alpha-based machines running Tru64 UNIX
- Code execution is watched by a profiling daemon
- Can only be used from outside the code
  - [http://www.tru64unix.compaq.com/dcpi](http://www.tru64unix.compaq.com/dcpi)
Profiling tools: valgrind

- Memory/thread debugger and cache profiler (4 tools). Part of KDE project: free.
- Run using *cachegrind program*
  Not an intrusive library, uses hardware capabilities of CPUs.
- Simple to use (even for automatic testing).
- Julian Seward et al
- http://valgrind.kde.org
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.fz-juelich.de/zam/PCL](http://www.fz-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: HPCToolkit

- High level portable tools for performance measurements and comparisons
  - Uses browser interface
  - PAPI should have looked like this
  - Make a change, check what happens on several architectures at once
- http://www.hipersoft.rice.edu/hpctoolkit
- Rob Fowler et al, Rice University, USA
hpcview Screen Shot

The pane

File pane

source pane

flatten/unflatten

parent scope

current scope

child scopes
HPCToolkit Philosophy 1

- Intuitive, top down user interface for performance analysis
  - Machine independent tools and GUI
    - Statistics to XML converters
  - Language independence
    - Need a good symbol locator at run time
  - Eliminate invasive instrumentation
  - Cross platform comparisons
HPCToolkit Philosophy 2

- Provide information needed for analysis and tuning
  - Multilanguage applications
  - Multiple metrics
    - Must compare metrics which are causes versus effects (examples: misses, flops, loads, mispredicts, cycles, stall cycles, etc.)
  - Hide getting details from user as much as possible
HPCToolkit Philosophy 3

- Eliminate manual labor from analyze, tune, run cycle
  - Collect multiple data automatically
  - Eliminate 90-10 rule
    - 90% of cycles in 10% of code… for a 500K line code, the hotspot is only 5,000 lines of code. How do you deal with a 5K hotspot???
  - Drive the process with simple scripts
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

\[
\begin{array}{c}
\text{black point} \\
\text{red point}
\end{array}
\]
Using PCL – Example 1

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

```plaintext
% 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

```c
#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 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

%   dcpioprof ./mg

<table>
<thead>
<tr>
<th>Column</th>
<th>Total</th>
<th>Period (for events)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmiss</td>
<td>45745</td>
<td>4096</td>
</tr>
</tbody>
</table>

===================================================================

<table>
<thead>
<tr>
<th>procedure</th>
<th>image</th>
</tr>
</thead>
<tbody>
<tr>
<td>mgSmooth</td>
<td>./mg</td>
</tr>
<tr>
<td>mgRestriction</td>
<td>./mg</td>
</tr>
<tr>
<td>mgProlongCorr</td>
<td>./mg</td>
</tr>
</tbody>
</table>

[...]
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

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch mispredict</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>IMUL busy</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>FDIV busy</td>
<td>0.0% to 0.5%</td>
</tr>
<tr>
<td>Other</td>
<td>0.0% to 0.0%</td>
</tr>
<tr>
<td>Unexplained stall</td>
<td>0.4% to 0.4%</td>
</tr>
<tr>
<td>Unexplained gain</td>
<td>-0.7% to -0.7%</td>
</tr>
<tr>
<td><strong>Subtotal dynamic</strong></td>
<td><strong>85.1%</strong></td>
</tr>
</tbody>
</table>
Using DCPI – Example 2

Static stalls are listed next:

<table>
<thead>
<tr>
<th>Source</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slotting</td>
<td>0.5%</td>
</tr>
<tr>
<td>Ra dependency</td>
<td>3.0%</td>
</tr>
<tr>
<td>Rb dependency</td>
<td>1.6%</td>
</tr>
<tr>
<td>Rc dependency</td>
<td>0.0%</td>
</tr>
<tr>
<td>FU dependency</td>
<td>0.5%</td>
</tr>
<tr>
<td><strong>Subtotal static</strong></td>
<td><strong>5.6%</strong></td>
</tr>
<tr>
<td><strong>Total stall</strong></td>
<td><strong>90.7%</strong></td>
</tr>
</tbody>
</table>
Using DCPI – Example 2

Useful cycles are listed in the end:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Useful</td>
<td>7.9%</td>
</tr>
<tr>
<td>Nops</td>
<td>1.3%</td>
</tr>
</tbody>
</table>

-----------------------------------------------
Total execution 9.3%

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

Optimization Techniques for Structured Grids
How to make codes fast

1. Use a fast algorithm (e.g., multigrid)
   I. It does not make sense to optimize a bad algorithm
   II. However, sometimes a fairly simple algorithm that is well implemented will beat a very sophisticated, super method that is poorly programmed

2. Use good coding practices

3. Use good data structures

4. Apply appropriate optimization techniques
Optimization of Floating-Point Operations
Optimization of FP operations

- Loop unrolling
- Fused Multiply-Add (FMA) instructions
- Exposing instruction-level parallelism (ILP)
- Software pipelining (again: exploit ILP)
- Aliasing
- Special functions
- Eliminating overheads
  - if statements
  - Loop overhead
  - Subroutine calling overhead
Loop unrolling

- Simplest effect of loop unrolling: fewer test/jump instructions (fatter loop body, less loop overhead)
- Fewer loads per flop
- May lead to threaded code that uses multiple FP units concurrently (instruction-level parallelism)

- How are loops handled that have a trip count that is not a multiple of the unrolling factor?
- Very long loops may not benefit from unrolling (instruction cache capacity!)
- Very short loops may suffer from unrolling or benefit strongly
Loop unrolling: Making fatter loop bodies

Example: DAXPY operation

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

ii = imod(N,4)
do i= 1,ii
  a(i) = a(i) + b(i) * c
enddo

doi= 1+ii,N,4
  a(i) = a(i) + b(i) * c
  a(i+1) = a(i+1) + b(i+1) * c
  a(i+2) = a(i+2) + b(i+2) * c
  a(i+3) = a(i+3) + b(i+3) * c
endo

Preconditioning loop handles cases when N is not a multiple of 4
Loop unrolling: Improving flop/load ratio

Analysis of the flop-to-load-ratio often unveils another benefit of unrolling:

```fortran
  do i = 1, N
    do j = 1, M
      y(i) = y(i) + a(j, i) * x(j)
    enddo
  enddo
```

Innermost loop: three loads and two flops performed; i.e., we have one load per flop
Loop unrolling: Improving flop/load ratio

```plaintext
do  i = 1, N, 2
   t1 = 0
   t2 = 0
   do  j = 1, M, 2
       t1 = t1 + a(j, i) * x(j) + a(j+1, i) * x(j+1)
       t2 = t2 + a(j, i+1) * x(j) + a(j+1, i+1) * x(j+1)
   enddo
   y(i) = t1
   y(i+1) = t2
enddo
```

Both loops unrolled twice

Innermost loop: 8 loads and 8 flops!
Exposes instruction-level parallelism
How about unrolling by 4?

Watch out for register spill!
Fused Multiply-Add (FMA)

On many CPUs (e.g., IBM Power3/Power4) there is an instruction which multiplies two operands and adds the result to a third.

Consider code

\[ a = b + c \times d + f \times g \]

versus

\[ a = c \times d + f \times g + b \]

Can reordering be done automatically?
Exposing ILP

program nrm1
real a(n)
tt = 0d0

do j = 1, n
   tt = tt + a(j) * a(j)
enddo

print *, tt
end

program nrm2
real a(n)
tt1 = 0d0

do j = 1, n
   tt1 = tt1 + a(j) * a(j)
enddo

tt2 = 0d0

print *, tt1 + tt2
Exposing ILP (cont’d)

- Superscalar CPUs have a high degree of on-chip parallelism that should be exploited
- The optimized code uses temporary variables to indicate independent instruction streams
- This is more than just loop unrolling!
- Can this be done automatically?
- Change in rounding errors?
Software pipelining

- Arranging instructions in groups that can be executed together in one cycle
- Again, the idea is to exploit instruction-level parallelism (on-chip parallelism)
- Often done by optimizing compilers, but not always successfully
- Closely related to loop unrolling
- Less important on out-of-order CPUs
Aliasing

- Arrays (or other data) that refer to the same memory locations
- Aliasing rules are different for various programming languages; e.g.,
  - FORTRAN forbids aliasing: unknown result
  - C/C++ permit aliasing
- This is one reason why FORTRAN compilers often produce faster code than C/C++ compilers do
Aliasing (cont’d)

Example:

```fortran
subroutine sub(n, a, b, c, sum)
double precision sum, a(n), b(n), c(n)

sum= 0d0
do i= 1,n
   a(i)= b(i) + 2.0d0*c(i)
enddo
return
end
```

FORTRAN rule: two variables cannot be aliased, when one or both of them are modified in the subroutine

Correct call: `call sub(n,a,b,c,sum)`
Incorrect call: `call sub(n,a,a,c,sum)`
Aliasing (cont’d)

- Aliasing is legal in C/C++: compiler must produce conservative code
- More complicated aliasing is possible; e.g., \( a(i) \) with \( a(i+2) \)
- C/C++ keyword restrict or compiler option -noalias
Special functions

- / (divide)
- sqrt
- exp, log
- sin, cos, …
- Etc.

are expensive (up to several dozen cycles)

- Use math Identities, e.g., \( \log(x) + \log(y) = \log(x \times y) \)
- Use special libraries that
  - vectorize when many of the same functions must be evaluated
  - trade accuracy for speed, when appropriate
Eliminating overheads: \textbf{if} statements

- \textbf{if} statements ...
  - Prohibit some optimizations (e.g., loop unrolling in some cases)
  - Evaluating the condition expression takes time
  - CPU pipeline may be interrupted
  - (dynamic jump prediction)

Goal: avoid \textbf{if} statements in the innermost loops
No generally applicable technique exists 😞
Eliminating \texttt{if} statements:
An example

\begin{verbatim}
subroutine thresh0(n,a,thresh,ic)
dimension a(n)
ic= 0
tt= 0.d0
do  j= 1,n
tt= tt + a(j) * a(j)
if (sqrt(tt).ge.thresh) then
  ic= j
  return
endif
ddoo
return
end
\end{verbatim}

- Avoid \texttt{sqrt} in condition! (square \texttt{thresh} instead)
- Add \texttt{tt} in blocks of 128 for example (without condition) and repeat last block when condition is violated
Eliminating loop overheads

- For starting a loop, the CPU must free certain registers: loop counter, address, etc.
- This may be significant for a short loop!
- Example: for n>m
  
  ```
  do i = 1, n
  do j = 1, m
  ...
  ```

  is less efficient than

  ```
  do j = 1, m
  do i = 1, n
  ...
  ```

- However, data access optimizations are even more important, see below
Eliminating subroutine calling overhead

- Subroutines (functions) are very important for structured, modular programming
- Subroutine calls are expensive (on the order of up to 100 cycles)
- Passing value arguments (copying data) can be extremely expensive, when used inappropriately
- Passing reference arguments (as in FORTRAN) may be dangerous from a point of view of correct software
- Reference arguments (as in C++) with `const` declaration
- Generally, in tight loops, no subroutine calls should be used
Eliminating subroutine calling overhead (cont’d)

- **Inlining**: `inline` declaration in C++ (see below), or done automatically by the compiler.

- **Macros** in C or any other language

```c
#define sqre(a) (a)*(a)
```

What can go wrong:

- `sqre(x+y)` → `x+y*x+y`
- `sqre(f(x))` → `f(x) * f(x)`

What if `f` has side effects?

What if `f` has no side effects, but the compiler cannot deduce that?
Memory Hierarchy Optimizations: Data Layout
Data layout optimizations

- Array transpose to get stride-1 access
- Building cache-aware data structures by array merging
- Array padding
- Etc.
Data layout optimizations

- Stride-1 access is usually fastest for several reasons; particularly the reuse of cache line contents
- Data layout for multidimensional arrays in FORTRAN: column-major order

Example:
4x3 array
\[ A(i,j) \]

<table>
<thead>
<tr>
<th>Memory address</th>
<th>A(1,1)</th>
<th>A(2,1)</th>
<th>A(3,1)</th>
<th>A(4,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(1,2)</td>
<td>A(2,2)</td>
<td>A(3,2)</td>
<td>A(4,2)</td>
</tr>
<tr>
<td>4</td>
<td>A(1,3)</td>
<td>A(2,3)</td>
<td>A(3,3)</td>
<td>A(4,3)</td>
</tr>
</tbody>
</table>

Data arrangement is “transpose” of usual matrix layout
Data layout optimizations

Stride-1 access: innermost loop iterates over first index

- Either by choosing the right data layout (array transpose) or
- By arranging nested loops in the right order (loop interchange):

\[
\begin{align*}
\text{do } i=1,N & \quad \text{do } j=1,M \\
\text{do } j=1,M & \quad \text{do } i=1,N \\
\quad a(i,j) &= a(i,j) + b(i,j) & \quad a(i,j) &= a(i,j) + b(i,j) \\
\text{ enddo } & \quad \text{ enddo } \\
\text{ enddo } & \quad \text{ enddo }
\end{align*}
\]

This will usually be done by the compiler!
Data layout optimizations:
Stride-1 access

```fortran
do i=1,N
    do j=1,M
        s(i)=s(i)+b(i,j)*c(j)
    enddo
enddo
```

Better transpose matrix b so that inner loop gets stride 1

How about loop interchange in this case?
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

- Example (cont’d): right-hand side and coefficients are accessed simultaneously, reuse cache line contents by array merging
  - Enhance spatial locality

```c
typedef struct {
    double f;
    double c_N, c_E, c_S, c_W, c_C;
} equationData; // Data merged in memory

double u[N][N]; // Solution vector
equationData rhsAndCoeff[N][N]; // Right-hand side and coefficients
```
Data layout optimizations: Array padding

- Idea: Allocate arrays larger than necessary
  - Change relative memory distances
  - Avoid severe cache thrashing effects
- Example (FORTRAN: column-major order):
  Replace
  \[
  \text{double precision } u(1024, 1024) \\
  \text{by} \\
  \text{double precision } u(1024+\text{pad}, 1024)
  \]
- How to choose \textit{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 arrays ⇒ Avoid *self-interference* misses
  - *Inter-variable padding*: pad between different arrays ⇒ Avoid *cross-interference* misses
Data layout optimizations: Array padding

- Padding in 2D; e.g., FORTRAN77:
  
  ```
  double precision u(0:1024+pad,0:1024)
  ```
Memory Hierarchy Optimizations:
Data Access
Loop optimizations

- Loop unrolling (see above)
- Loop interchange
- Loop fusion
- Loop split = loop fission = loop distribution
- Loop skewing
- Loop blocking
- Etc.
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:

\[
\begin{align*}
\text{do } & i= 1,N \\
& a(i) = a(i) + b(i) \\
\text{enddo} \\
\text{do } & i= 1,N \\
& a(i) = a(i) \times c(i) \\
\text{enddo}
\end{align*}
\]

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

After:

\[
\begin{align*}
\text{do } & i= 1,N \\
& a(i) = (a(i) + b(i)) \times c(i) \\
\text{enddo}
\end{align*}
\]

- \(a\) is loaded into the cache only once
Data access optimizations:
Loop fusion

Example: red/black Gauss-Seidel iteration in 2D
Data access optimizations: Loop fusion

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

```plaintext
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
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:

```plaintext
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
```

Prof. Craig C. Douglas
University of Kentucky and
Yale University
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 split

- The *inverse* transformation of loop fusion
- Divide work of one loop into two to make body less complicated
  - Leverage compiler optimizations
  - Enhance instruction cache utilization
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

```plaintext
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
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 $u_3 = u_3 + I_2 u_2$. Smooth $A_3 u_3 = f_3$.
Set $u_2 = u_2 + I_1 u_1$. Smooth $A_2 u_2 = f_2$.

W Cycle
Cache-optimized multigrid: DiMEPACK library

- DFG project 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://www10.informatik.uni-erlangen.de/dime
V(2,2) cycle - bottom line

<table>
<thead>
<tr>
<th>Mflops</th>
<th>For what</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Standard 5-pt. Operator</td>
</tr>
<tr>
<td>56</td>
<td>Cache optimized (loop orderings, data merging, simple blocking)</td>
</tr>
<tr>
<td>150</td>
<td>Constant coeff. + skewed blocking + padding</td>
</tr>
<tr>
<td>220</td>
<td>Eliminating rhs if 0 everywhere but boundary</td>
</tr>
</tbody>
</table>
Example: Cache-Optimized Multigrid on Regular Grids in 3D
Data layout optimizations for 3D multigrid

- Array padding
Data layout optimizations for 3D multigrid

- Standard padding in 3D; e.g., FORTRAN77:

  `double precision u(0:1024,0:1024,0:1024)`

  becomes:

  `double precision u(0:1024+pad1,0:1024+pad2,0:1024)`
Data layout optimizations for 3D multigrid

- Non-standard padding in 3D:

\[
\begin{align*}
\text{double precision } & \ u(0:1024+\text{pad1}, 0:1024, 0:1024) \\
& \ldots \\
& u(i+k*\text{pad2}, j, k) \\
\text{(or use hand-made index linearization – performance effect?)}
\end{align*}
\]
Data layout optimizations for 3D multigrid

- Array merging
Data access optimizations for 3D multigrid

- 1-way blocking with loop-interchange
Data access optimizations for 3D multigrid

- 2-way blocking and 3-way blocking
Data access optimizations for 3D multigrid

- 4-way blocking
Example: Cache Optimizations for the Lattice Boltzmann Method
Lattice Boltzmann method

- Mainly used in CFD applications
- Employs a regular grid structure (2D, 3D)
- Particle-oriented approach based on a microscopic model of the moving fluid particles
- Jacobi-like cell update pattern: a single time step of the LBM consits of
  - stream step and
  - collide step
LBM (cont’d)

- Stream: read *distribution functions* from neighbors
- Collide: re-compute own distribution functions
Data layout optimization

**Layout 1:**
Two separate grids
(standard approach)
Layout 2: *Grid Compression*: save memory, enhance locality
Data access optimizations

Access pattern 1: 3-way blocking:
3-way blocking (cont’d):
Access pattern 2: 4-way blocking:
4-way blocking (cont’d):
Illustration of the combination of layout + access optimizations

Layout: separate grids, access pattern: 3-way-blocking:

Layout: separate grids, access pattern: 4-way-blocking:
Layout: Grid compression, access pattern: 3-way blocking:
Layout: grid compression, access pattern: 4-way blocking:
Performance results

MFLOPS for 2D GS, const. coeff.s, 5-pt.,
DEC PWS 500au, Alpha 21164 CPU, 500 MHz
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

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

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

<table>
<thead>
<tr>
<th>Size</th>
<th>+/-</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>27.4</td>
<td>43.4</td>
<td>29.1</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>65</td>
<td>33.4</td>
<td>46.3</td>
<td>19.5</td>
<td>0.9</td>
<td>0.0</td>
</tr>
<tr>
<td>129</td>
<td>36.9</td>
<td>42.3</td>
<td>19.1</td>
<td>1.7</td>
<td>0.0</td>
</tr>
<tr>
<td>257</td>
<td>38.1</td>
<td>34.1</td>
<td>25.1</td>
<td>2.7</td>
<td>0.0</td>
</tr>
<tr>
<td>513</td>
<td>38.0</td>
<td>28.3</td>
<td>27.0</td>
<td>6.7</td>
<td>0.1</td>
</tr>
<tr>
<td>1025</td>
<td>36.9</td>
<td>24.9</td>
<td>19.7</td>
<td>17.6</td>
<td>0.9</td>
</tr>
<tr>
<td>2049</td>
<td>36.2</td>
<td>25.5</td>
<td>0.4</td>
<td>36.9</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Memory access behavior

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

<table>
<thead>
<tr>
<th>Size</th>
<th>+/-</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>28.2</td>
<td>66.4</td>
<td>5.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>65</td>
<td>34.3</td>
<td>55.7</td>
<td>9.1</td>
<td>0.9</td>
<td>0.0</td>
</tr>
<tr>
<td>129</td>
<td>37.5</td>
<td>51.7</td>
<td>9.0</td>
<td>1.9</td>
<td>0.0</td>
</tr>
<tr>
<td>257</td>
<td>37.8</td>
<td>52.8</td>
<td>7.0</td>
<td>2.3</td>
<td>0.0</td>
</tr>
<tr>
<td>513</td>
<td>38.4</td>
<td>52.7</td>
<td>6.2</td>
<td>2.4</td>
<td>0.3</td>
</tr>
<tr>
<td>1025</td>
<td>36.7</td>
<td>54.3</td>
<td>6.1</td>
<td>2.0</td>
<td>0.9</td>
</tr>
<tr>
<td>2049</td>
<td>35.9</td>
<td>55.2</td>
<td>6.0</td>
<td>1.9</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Performance results (cont’d)

3D MG, F77, var. coeff.s, 7-pt., Intel Pentium4, 2.4 GHz, Intel ifc V7.0 compiler

![Bar chart showing performance results for different blocking schemes and compiler optimizations.](chart.png)
Performance results (cont’d)
2D LBM (D2Q9), C(++), AMD Athlon XP 2400+, 2.0 GHz, Linux, gcc V3.2.1 compiler

![Graph showing performance results for different grid sizes and grid compression methods.]
Performance results (cont’d)

Cache behavior (left: L1, right: L2) for previous experiment, measured with PAPI

---

Prof. Craig C. Douglas  
University of Kentucky and  
Yale University

HiPC2003, 12/17/2003  
vHPC Cache Aware Methods
Performance results (cont’d)

3D LBM (D3Q19), C, AMD Opteron, 1.6 GHz, Linux, gcc V3.2.2 compiler
C++-Specific Considerations
C++-specific considerations

We will (briefly) address the following issues:

- Inlining
- Virtual functions
- Expression templates
Inlining

- Macro-like code expansion: replace function call by the body of the function to be inlined
- How to accomplish inlining:
  - Use C++ keyword `inline`, or
  - Define the method within the declaration
- In any case: the method to be inlined needs to be defined in the header file
- However: inlining is just a suggestion to the compiler!
Inlining (cont‘d)

- Advantages:
  - Reduce function call overhead (see above)
  - Leverage *cross-call optimizations*: optimize the code after expanding the loop body

- Disadvantage:
  Size of the machine code increases (instruction cache capacity!)
Virtual functions

- Member functions may be declared to be virtual (C++ keyword `virtual`)
- This mechanism becomes relevant when base class pointers are used to point to instances of derived classes
- Actual member function to be called can often be determined only at runtime (polymorphism)
- Requires virtual function table lookup (at runtime!)
- Can be very time-consuming!
Inlining virtual functions

- Virtual functions are often not compatible with inlining where function calls are replaced by function bodies at compile time.
- If the type of the object can be deduced at compile time, the compiler can even inline virtual functions (at least theoretically ...)

Prof. Craig C. Douglas
University of Kentucky and
Yale University
Expression templates

- C++ technique for passing expressions as function arguments
- Expression can be inlined into the function body using (nested) C++ templates
- Avoid the use of temporaries and therefore multiple passes of the data through the memory subsystem; particularly the cache hierarchy
Example

Define a simple vector class in the beginning:

class vector {
    private:
        int length;
        double a[];
    public:
        vector(int l);
        double component(int i) { return a[i]; }
    ...
};
Example (cont’d)

Want to efficiently compute vector sums like
\[ c = a + b + d; \]

Efficiently implies

- Avoiding the generation of temporary objects
- **Pumping** data through the memory hierarchy several times. This is actually the time-consuming part. Moving data is more expensive than processing data!
Example (cont’d)

Need a wrapper class for all expressions:

template<class A>
class DExpr { // double precision expression
    private:
        A wa;
    public:
        DExpr(const A& a) : wa(a) {} // Constructor
        double component(int i) { return wa.component(i); } // Component function
};
Example (cont’d)

Need an expression template class to represent sums of expressions

template<class A, class B>
class DExprSum {
    A va;
    B vb;
    public:
    DExprSum(const A& a, const B& b) : va(a), vb(b) {}
    double component(int i) {
        return va.component(i) + vb.component(i);
    }
};
Example (cont’d)

Need overloaded `operator+()` variants for all possible return types, for example:

```cpp
template<class A, class B>
DExpr<
DExprSum<DExpr<A>, DExpr<B>>,>
operator+(const DExpr<A>& a, const DExpr<B>& b) {
    typedef DExprSum<DExpr<A>, DExpr<B>> ExprT;
    return DExpr<ExprT>(ExprT(a,b));
}
```
Example (cont‘d)

- The **vector** class must contain a member function `operator=(const A& ea)`, where `A` is an expression template class.
- Only when this member function is called, the actual computation (vector sum) takes place.
Part III

Optimization Techniques for Unstructured Grids
Optimizations for unstructured grids

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

This is really a quasi-unstructured mesh: there is plenty of structure in most of the oceans. Coastal areas provide a real challenge.
Subgrids and patches
Motivating example

- 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

- 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.
- Find structures in quasi-unstructured case.
- 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:** *usable cache size*.
- Such decomposition problems have been studied in depth for load balancing parallel computation.
Example of subblock membership

Cache blocks identified

Subblocks identified

Cache block boundary $L_1^2$

$\Omega_h^2$

$\Omega_h^1$
Distance algorithms

- Several constants
  - $d$ The degrees of freedom per vertex.
  - $K$ Average number of connections per vertex.
  - $N_{\Omega}$ Number of vertices in cache block $N_{\Omega}$.

- Three cases for complexity bounds
  - Cache boundaries connected.
  - Physical boundaries unknown.
  - Physical boundaries known.
Standard Gauss-Seidel

- The complexity bound $C_{gs}$ in this notation is given by

$$C_{gs} \leq 2d^2 N_\Omega K + d.$$
Cache boundaries connected

Algorithm 1 Mark cache interior nodes.

Label-Internal-Nodes
1: Set current distance \( c = 3 \).
2: Let \( m \) be the number of subblocks desired.
3: while \( S_2 \) is not empty do
4:   Move contents of \( S_2 \) to \( S_1 \).
5:   while \( S_1 \) is not empty do
6:     Pop node \( i \) off \( S_1 \).
7:     for each node \( j \) adjacent to \( i \) do
8:       if distance \( D_j \) == 0 then
9:         Set distance \( D_j = c \).
10:        Push \( j \) onto \( S_2 \).
11:     end if
12:   end for
13: end while
14: if \( c < m \) then
15:   Set \( c = c + 1 \).
16: end if
17: end while
Algorithm 2 (cache boundaries connected) Mark cache boundary nodes.

Label-Boundary-Nodes

1: Initialize stacks $S_1$ and $S_2$.
2: Set $D_i = 0$ for all $i$ in cache block $\Omega_s$.
3: Find any node $i$ on cache boundary $\partial \Omega_s$.
4: Push $i$ onto $S_1$.
5: while $S_1$ is not empty do
6:   Pop node $i$ off $S_1$.
7:   if $i$ is in $\partial \Omega_s$ then
8:     Set $D_i = 1$.
9:     for each node $j$ connected to $i$ do
10:    if $D_j == 0$ and $j$ is in $\Omega_s$ then
11:       Set $D_j = -1$.
12:       Push $j$ onto $S_1$.
13:    end if
14:  end for
15: else
16:  Set $D_i = 2$.
17:  Push $i$ onto $S_2$.
18: end if
19: end while
Cache boundaries connected

- The complexities of Algorithms 1 and 2 are

\[ C_1 \leq 5N_\Omega K \]

and

\[ C_2 \leq (7K+1)N_\Omega. \]

- The cost of Algorithms 1 and 2 with respect to Gauss-Seidel is at most \(6d^{-2}\) sweeps on the finest grid.
Physical boundaries unknown

**Algorithm 3** (unknown physical boundary nodes) Mark cache boundary nodes.

```
Label-Boundary-Nodes
1: Initialize stacks $S_1$ and $S_2$.
2: Set distance $D_i = 0$ for all $i$ in $\Omega_s$.
3: for each node $i$ in $\Omega_s$ such that $D_i == 0$ do
4:     if $i$ is on $\partial\Omega_s$ then
5:         Push $i$ onto $S_1$.
6:     while $S_1$ is not empty do
7:         Pop node $i$ off $S_1$.
8:         if $i$ is in $\partial\Omega_s$ then
9:             Set $D_i = 1$.
10:        for each node $j$ connected to $i$ do
11:            if ($D_j == 0$) AND ($j$ is in $\Omega_s$) then
12:                Set $D_j = -1$.
13:            Push $j$ onto $S_1$.
14:        end if
15:    end for
16:    else
17:        Set $D_i = 2$.
18:        Push $i$ onto $S_2$.
19:    end if
20: end while
21: end if
22: end for
```
Physical boundaries unknown

- The complexity of Algorithm 3 is
  \[ C_1 \leq 9N_\Omega K. \]

- The cost of Algorithms 1 and 3 with respect to Gauss-Seidel is at most \( 7d^{-2} \) sweeps on the finest grid.
Algorithm 4 (known physical boundary nodes) Main algorithm for marking cache boundary nodes.

Label-Boundary-Nodes
1: Let PBN denote a physical boundary node.
2: Initialize stacks $S_1, S_2, S_3, S_4$.
3: Set $D_i = 0$ for each node $i$ in $\Omega_s$.
4: Find any node $k$ in $\partial \Omega_s$, and push $k$ onto stack $S_4$.
5: repeat
6:   repeat
7:     while ($S_1 \neq \emptyset$) AND ($S_4 \neq \emptyset$) do
8:         Pop $k$ off stack $S_4$.
9:         if $D_k = 0$ then
10:            Set $D_k = 1$.
11:            Push $k$ onto $S_1$.
12:            end if
13:     end while
14: while $S_1 \neq \emptyset$ do
15:     Pop $i$ off of $S_1$.
16:     if $i$ is in $\partial \Omega_s$ then
17:         Set $D_i = 1$.
18:         for each node $j$ connected to $i$ do
19:             if ($D_j = 0$) AND ($j$ is in $\Omega_s$) then
20:                 Set $D_j = -1$.
21:                 Push $j$ onto $S_1$.
22:             end if
23:         end for
24: else
25:     Set $D_i = 2$.
26:     Push $i$ onto $S_2$.
27:     if $i$ is a PBN then
28:         Push $i$ onto $S_3$.
29:     end if
30: end if
31: end while
32: until $S_4$ is empty.
33: Call Find-Next-Cache-Boundary-Component($S_3, S_4$).
34: until $S_4$ is empty
Physical boundaries known

Algorithm 5 (known physical boundary nodes) Finds next cache boundary component for Alg. 4.

Find-Next-Cache-Boundary-Component($S_3, S_4$)

1: while $S_3 \neq \emptyset$ do
2:   Pop $i$ off of $S_3$.
3:   if $i$ is in $\partial \Omega_s$ then
4:     Push $i$ onto $S_4$.
5:   end if
6:   for each node $j$ connected to $i$ do
7:     if ($j \in \Omega_s$) AND ($D_j == 0$) AND ($j$ is a PBN) then
8:       Set $D_j = -1$.
9:     Push $j$ onto $S_3$.
10:   end if
11: end for
12: end while
Physical boundaries known

- The complexity of Algorithms 1, 4, and 5 is
  \[ C_{1,4,5} \leq (11K+1)N_\Omega. \]

- The cost of Algorithms 1, 4, and 5 with respect to Gauss-Seidel is at most \(10d^{-2}\) sweeps on the finest grid. This assumes that the number of physical boundary nodes is \((N_\Omega)^{1/2}\), which is quite pessimistic.
Physical boundaries known

- If we assume that there are no more than $\frac{1}{2}N_\Omega$ physical boundary nodes per cache block, then we get a better bound, namely, the cost of Algorithms 1, 4, and 5 with respect to Gauss-Seidel is at most $5.5d^{-2}$ sweeps on the finest grid.

- Clearly with better estimates of realistic number of physical boundary nodes (i.e., much less than $\frac{1}{2}N_\Omega$) per cache block, we can reduce this bound.
Preprocessing costs

- Pessimistic (never seen to date in a real problem) bounds
  - $d = 1$: 5.5 Gauss-Seidel sweeps
  - $d = 2$: 1.375 Gauss-Seidel sweeps
  - $d = 3$: 0.611 Gauss-Seidel sweeps

- In practice bounds
  - $d = 1$: ~1 Gauss-Seidel sweeps
  - $d = 2$: ~½ Gauss-Seidel sweeps
Numerical results: Austria

Experiment: Austria

Two dimensional elasticity

\[ T = \text{Cauchy stress tensor}, \ w = \text{displacement} \]

\[ f = (1,-1)^T \text{ on } \Gamma_4, \]
\[ (9.5-x,4-y) \text{ if } (x,y) \text{ is in region surrounded by } \Gamma_5, \text{ and } \]
\[ 0 \text{ otherwise.} \]

\[ -\nabla T = f \text{ in } \Omega, \]
\[ \frac{\partial w}{\partial n} = 100w \text{ on } \Gamma_1, \]
\[ \frac{\partial w}{\partial y} = 100w \text{ on } \Gamma_2, \]
\[ \frac{\partial w}{\partial x} = 100w \text{ on } \Gamma_3, \]
\[ \frac{\partial w}{\partial n} = 0 \text{ everywhere else.} \]
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.
Code commentary

- Preprocessing steps separate.
- Computation
  - Standard algorithms implemented in as efficient a manner known without the cache aware or active set tricks. This is not Randy Bank’s version, but much, much faster.
  - Cache aware implementations uses quasi-unstructured information.
  - The codes work equally well in 2D and 3D (the latter needs a larger cache than 2D) and are really general sparse matrix codes with a PDE front end.
Implementation details

- **One parameter tuning**: usable cache size, which is normally ~60% of physical cache size.

- **Current code**
  - Fortran + C (strict spaghetti coding style and fast).
  - Requires help from authors to add a new domain and coarse grid.

- **New code**
  - C++
  - Should not require any author to be present.
  - Is supposed to be available by end of September and will probably be done earlier. Aiming for inclusion in ML (see software.sandia.gov)
Books

Journal articles


Conference proceedings

Conference proceedings

Related websites

- http://www.mgnet.org
- http://www.mgnet.org/~douglas/ccd-preprints.html
- http://www.mgnet.org/~douglas/ml-dddas.html
- http://www10.informatik.uni-erlangen.de/dime
- http://valgrind.kde.org
- http://www.fz-juelich.de/zam/PCL
- http://icl.cs.utk.edu/projects/papi
- http://www.hipersoft.rice.edu/hpctoolkit
- http://www.tru64unix.compaq.com/dcpi