Efficient Programming in Fortran, C, and C++

Ulrich Rüde, Markus Kowarschik
Lehrstuhl für Systemsimulation (Informatik 10)
Universität Erlangen-Nürnberg

G. Hager, G. Wellein, F. Deserno, F. Brechtefeld
Regionales Rechenzentrum Erlangen (RRZE)

With help from
C.C. Douglas, C. Freundl, K. Iglberger, W. Karl, H. Pfänder, T. Pohl, N. Thürey, C. Weiß, and J. Wilke

Leibniz-Rechenzentrum München
July 21, 2003

Contacting us by eMail

- ulrich.ruede@cs.fau.de
- markus.kowarschik@cs.fau.de
- georg.hager@rrze.uni-erlangen.de
- gerhard.wellein@rrze.uni-erlangen.de
- frank.deserno@rrze.uni-erlangen.de
- frank.brechtefeld@rrze.uni-erlangen.de

Overview

- Part I: Architectures and Fundamentals
- Part II: Optimization Techniques
- Part III: Case Studies (G. Hager)

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 70ies)
- More general elliptic equations may need O(100) operations per unknown
- A modern CPU can do 1-5 GFLOPS
- So we should be solving 10-50 million unknowns per second
- Should need O(100) Mbytes of memory

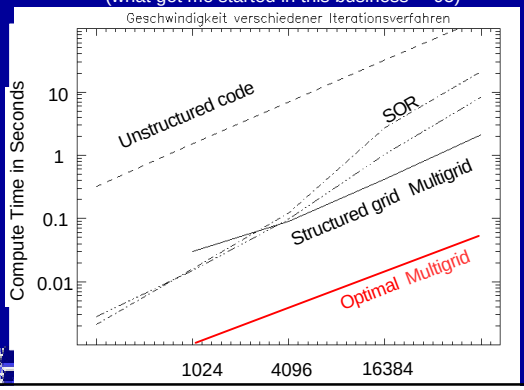
How fast are solvers today?

- 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

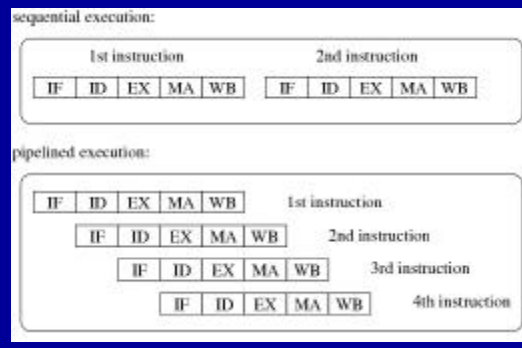
(what got me started in this business ~ '95)



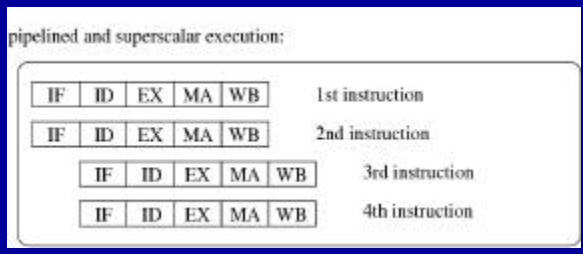
Elements of CPU architecture

- 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



Pipelining (cont'd)



CPU trends

- EPIC (similar to VLIW) (IA64)
- Multi-threaded architectures (Alpha?)
- 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
  - 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*

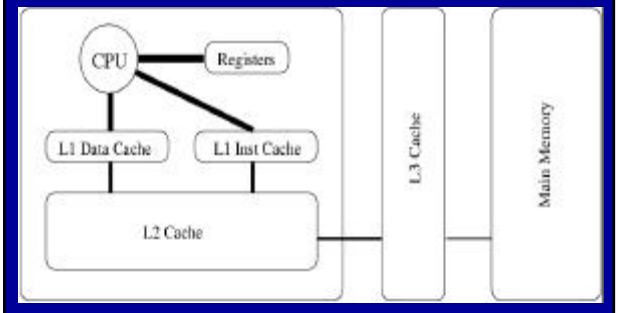
Principles of locality

- 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 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**

Typical 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 (cont'd)

- 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 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

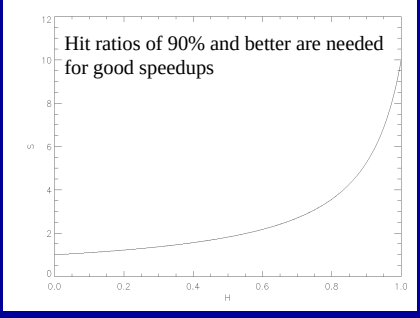
The cache efficiency is characterized by the cache hit ratio, the *effective* time for a data access is

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

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



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
  - L2 = 4 MB, direct mapped, line size = 128, write back
- IBM Power 4:
  - L1 = 32 KB, 2way, 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:
  - 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, 8-way, line size = 128
- Intel Itanium:
  - L1 = 16 KB, 4-way
  - L2 = 96 KB, 6-way
  - L3: off-chip, size varies
- Intel Itanium2:
  - L1 = 16 KB
  - L2 = 256 KB
  - L3: 1.5 MB or 3 MB

Basic efficiency guidelines

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

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, etc.), usually as source code, not specifically optimized
- Vendor-specific; e.g., cxml for Compaq/Alpha with highly tuned LAPACK routines, for example

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,...,5
  - **`-fast`**: may change round-off behavior
  - **`-unroll`**
  - **`-arch`**
  - Etc.
- Learning about your compiler is usually worth it: RTFM

Find good compiler options (cont'd)

Hints:

- Read **`man cc`** (**`man f77`**)
- Look up compiler options documented in www.specbench.org for specific platform
- Experiment and compare performance

Use suitable data layout

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

```
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++):

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

can often be handled more efficiently by using

```
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: 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`

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

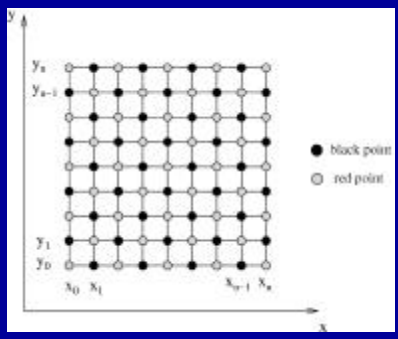
Profiling tools: DCPI

- DCPI = Compaq (Digital) Continuous Profiling Infrastructure (HP?)
- Only for Alpha-based machines running 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



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)

Part II

Optimization Techniques

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 which is not a multiple of the unrolling factor?
- Already fat loops do hardly 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

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

```
do i= 1,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
enddo
```

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

Preconditioning loop handles cases when N is no multiple of 4

Loop unrolling: Improving flop/load ratio

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

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

Innermost loop: two loads and two flops performed; i.e., we have one load per flop

Loop unrolling: Improving flop/load ratio

```
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 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*d + f*g
```

versus

```
a= c*d + f*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
tt2= 0d0
do j= 1,n,2
  tt1= tt1 + a(j)*a(j)
  tt2= tt2 + a(j+1)*a(j+1)
enddo

tt= tt1 + tt2
print *, tt
end
```


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, unspec. 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:

```
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 complicagted 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*y)
- Use special libraries that
  - vectorize when many of the same functions must be evaluated
  - trade accuracy for speed, when appropriate

Eliminating overheads: `if` statements

`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 `if` statements in the innermost loops

No generally applicable technique exists ☹

Eliminating `if` statements – Example

```
subroutine thresh0(n,a,threshh,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
enddo
return
end
```

- Avoid `sqrt` in condition!
- Add `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

```
#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)`

Memory address

8	A(1,1)	A(2,1)	A(3,1)	A(4,1)
4	A(1,2)	A(2,2)	A(3,2)	A(4,2)
0	A(1,3)	A(2,3)	A(3,3)	A(4,3)

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*):

```
do i=1,N                          do j=1,M
  do j=1,M                          do i=1,N
    a(i,j)=a(i,j)+b(i,j)    ⟶        a(i,j)=a(i,j)+b(i,j)
  enddo                             enddo
enddo                             enddo
```

Stride-N access ⟶ **Stride-1 access**

This will usually be done by the compiler!

Data layout optimizations: Stride-1 access

```
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

```
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

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

by

```
double precision u(1024+pad, 1024)
```

- How to choose **pad?**

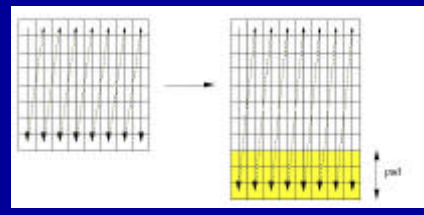
Data layout optimizations: Array padding

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

```
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)

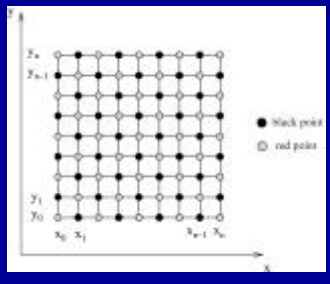
After:

```
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 in 2D



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
```

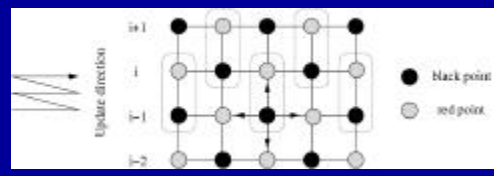
Data access optimizations: Loop fusion

```
// 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
```

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 split

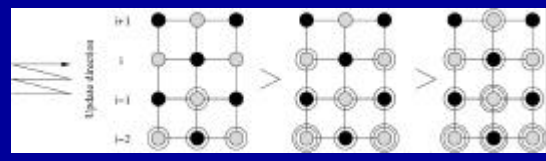
- The *inverse* transformation of loop fusion
- Divide work of one loop into two to make body less complicated
  - Leverage compiler optimizations
  - Enhance instruction cache utlization

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
```

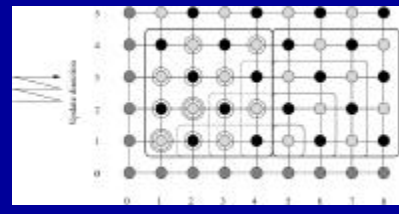

Data access optimizations: Loop blocking

```
  // 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

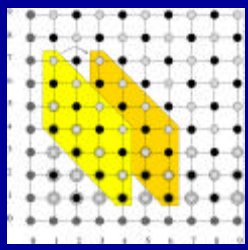
Data access optimizations Loop blocking

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



Data access optimizations: Loop blocking

- Illustration: 2D skewed blocking



Two common multigrid algorithms

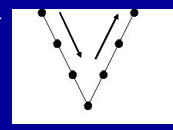
V Cycle to solve $A_4u_4=f_4$

Smooth $A_4u_4=f_4$. Set $f_3 = R_3r_4$.

Smooth $A_3u_3=f_3$. Set $f_2 = R_2r_3$.

Smooth $A_2u_2=f_2$. Set $f_1 = R_1r_2$.

Solve $A_1u_1=f_1$ directly.

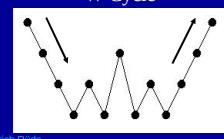


Set $u_4 = u_4 + I_3u_3$. Smooth $A_4u_4=f_4$.

Set $u_3 = u_3 + I_2u_2$. Smooth $A_3u_3=f_3$.

Set $u_2 = u_2 + I_1u_1$. Smooth $A_2u_2=f_2$.

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

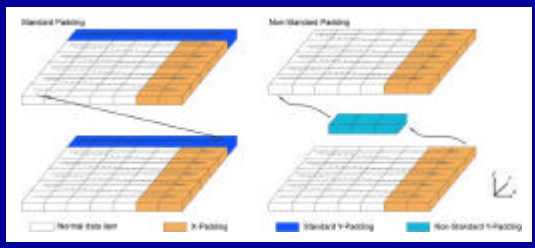
V(2,2) cycle - bottom line

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

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:

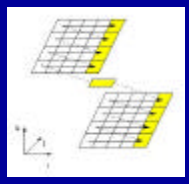
```
double precision u(0:1024,0:1024,0:1024)
```

becomes:

```
double precision u(0:1024+pad1,0:1024+pad2,0:1024)
```


Data layout optimizations for 3D multigrid

- Non-standard padding in 3D:

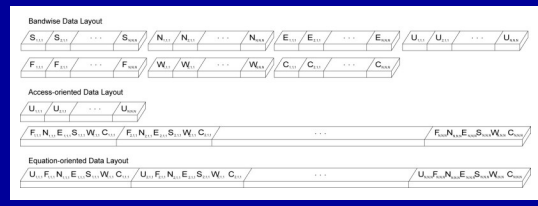


```
double precision u(0:1024+pad1,0:1024,0:1024)
      ...
u(i+k*pad2, j, k)
```

(or use hand-made index linearization – performance effect?)

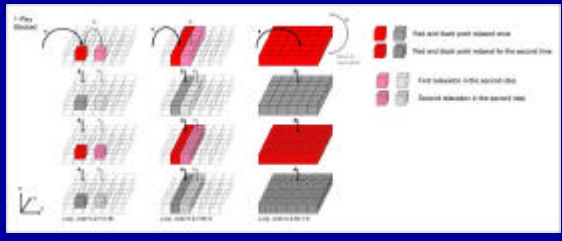
Data layout optimizations for 3D multigrid

- Array merging



Data access optimizations for 3D multigrid

- 1-way blocking with loop-interchange

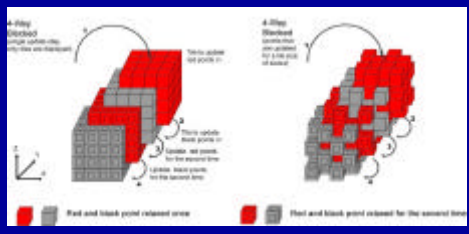


Data access optimizations for 3D multigrid

- 2-way blocking and 3-way blocking

Data access optimizations for 3D multigrid

- 4-way blocking



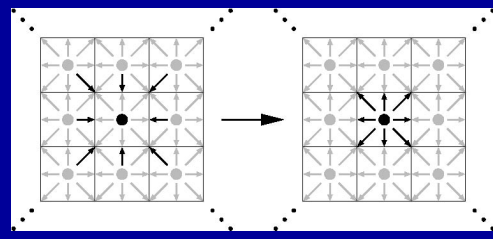
Example: Cache Optimizations for the Lattice Boltzmann Method

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

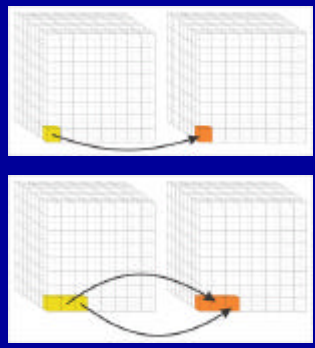


Data layout optimization

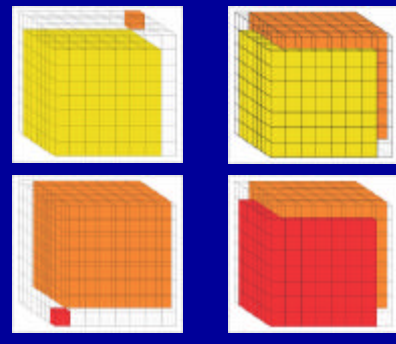
Layout 1:

Two separate grids

(standard approach)

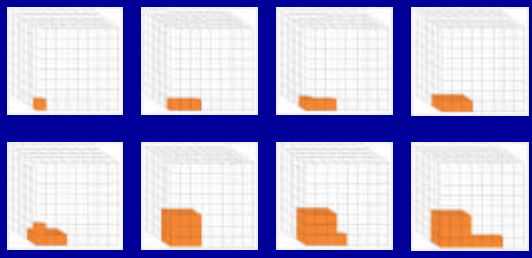


Layout 2: *Grid Compression:* save memory, enhance locality

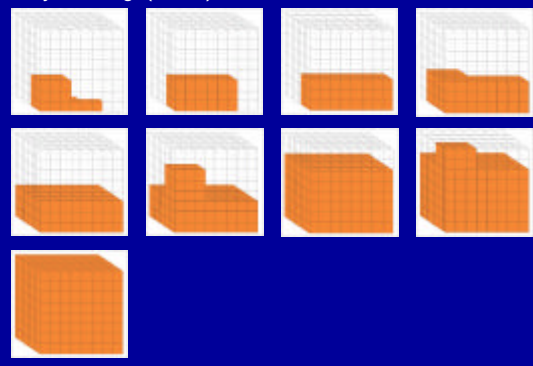


Data access optimizations

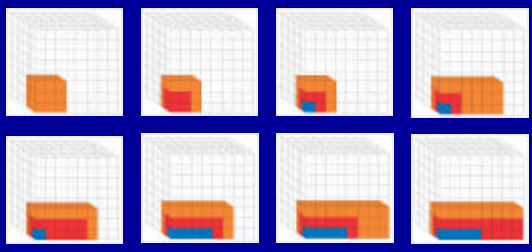
Access pattern 1: 3-way blocking:



3-way blocking (cont'd):



Access pattern 2: 4-way blocking:



4-way blocking (cont'd):

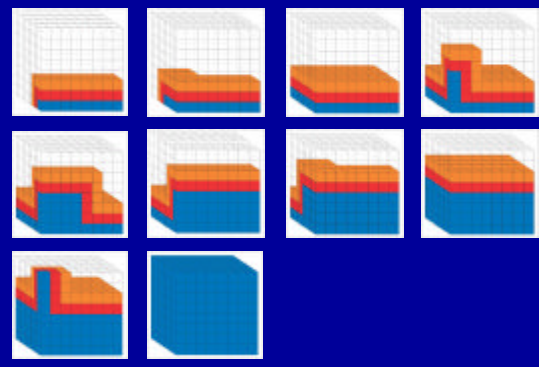
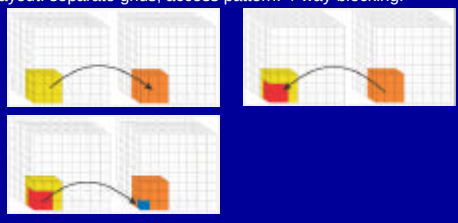


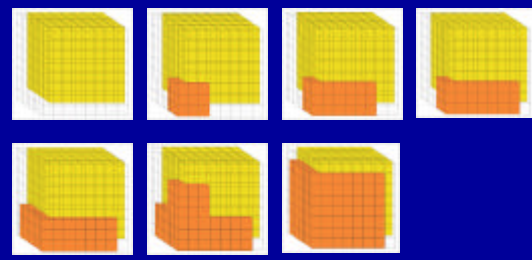
Illustration of the combination of layout + access optimizations

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

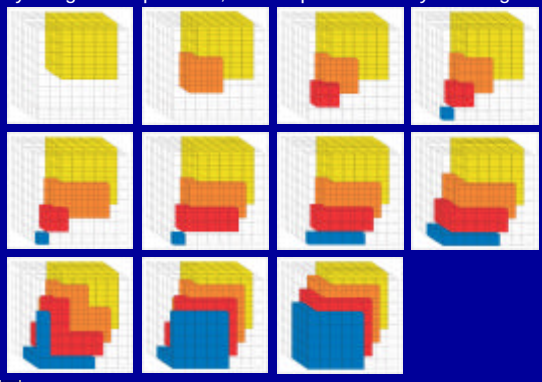
Layout: separate grids, access pattern: 4-way-blocking:



Layout: Grid compression, access pattern: 3-way blocking:

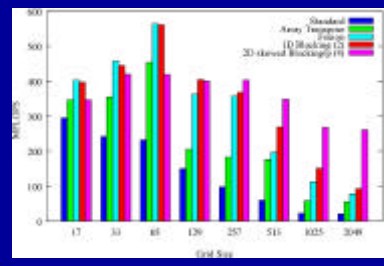


Layout: grid compression, access pattern: 4-way blocking:



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

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

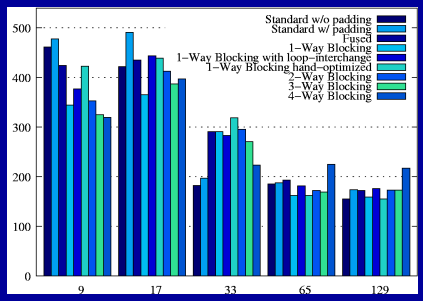
Memory access behavior

- 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

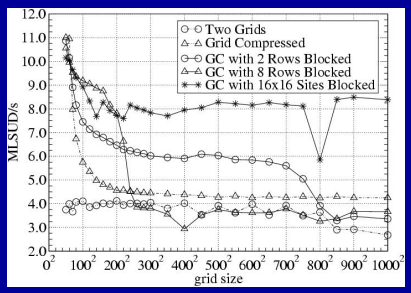
Performance results (cont'd)

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



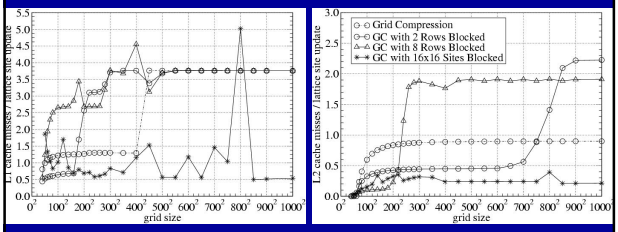
Performance results (cont'd)

2D LBM (D2Q9), C(++), AMD Athlon XP 2400+, 2.0 GHz, Linux, gcc V3.2.1 compiler



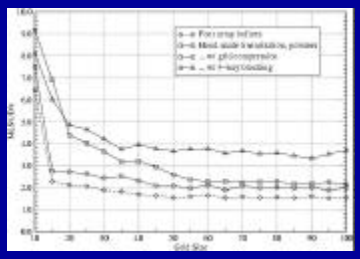
Performance results (cont'd)

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



Performance results (cont'd)

3D LBM (D3Q19), C, AMD Opteron, 1.6 GHz, Linux, gcc V3.2.2 compiler *(preliminary!)*



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 ...)

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) {}
    double component(int i) { return wa.component(i); }
};
```


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:

```
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.
- References: T. Veldhuizen, C. Pflaum

Optimizations for Computations on Unstructured Grids

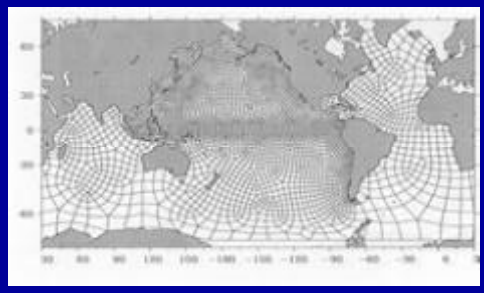
See C.C. Douglas et.al.

http://www.ccs.uky.edu/~douglas/ccd-kfcs.html

Optimizations for unstructured grids

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

Is it really unstructured?



Subgrids and patches

We are exploiting similar structures in the KONWIHR project *Gridlib*

References

- **S. Goedecker, A. Hoisie: Performance Optimization of Numerically Intensive Codes, SIAM, 2001.**
- 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, J. Hu, W. Karl, M. Kowarschik, U. Rüde, C. Weiss, Cache optimization for structured and unstructured grid multigrid, Electronic Transactions on Numerical Analysis, 10 (2000), pp. 25-40.
- J. Handy, The Cache Memory Book, 2nd ed., Academic Press, 1998
- J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd ed., Morgan Kauffmann Publishers, 1996.

References (cont'd)

- 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.
- U. Rüde, Iterative Algorithms on High Performance Architectures, Proc. of the EuroPar97 Conference, Lecture Notes in Computer Science, Springer, Berlin, 1997, pp. 57-71.
- U. Rüde, 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, Vol. 8, Springer, 1998, pp. 459-471.
- D. Bulka, D. Mayhew, Efficient C++, Addison-Wesley, 2000

References (cont'd)

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

Related websites

- http://www10.informatik.uni-erlangen.de/dime
- http://www.ccs.uky.edu/~douglas/ccd-kfcs.html
- http://www.mgnet.org
- http://www.fz-juelich.de/zam/PCL
- http://icl.cs.utk.edu/projects/papi
- http://www.tru64unix.compaq.com/dcpi

Part III

Case Studies (G. Hager)