Architecture-Aware Programming

DASMOD
Summer School 2006
Kaiserslautern, Germany

Ulrich Rüde, Jan Treibig
Universität Erlangen

Overview

- Introduction and Motivation
- Part I: Cache-Based Architectures
  Modern Microprocessor and Cache Design
- Part II: Programming Techniques
  Prefetching, Cache blocking, SIMD
- Part III: Case Studies
  Data Layout/Access, Iterative Solvers

Introduction – Contact Us

Ulrich Rüde (ruede@cs.fau.de)
Jan Treibig (jan.treibig@cs.fau.de)

Motivation

Best possible usage
of a given processor

- Meet the assumptions of the hardware designer
- The CPU itself is a compromise
- Special techniques necessary to optimize an algorithm for a specific architecture

A little quiz ...

1 Kflops = 10^3, 1 Mflops = 10^6, 1 Gflops = 10^9, 1 Tflops = 10^{12},
1 Pflops = 10^{15} *floating point operations per second*

1. What is the speed of your PC?
2. What is the speed of the fastest computer currently available (and where is it located)
3. What was the speed of the fastest computer in
   a) 1995?
   b) 2000?
   c) 2005?

A little quiz ...

1 Kflops = 10^3, 1 Mflops = 10^6, 1 Gflops = 10^9, 1 Tflops = 10^{12},
1 Pflops = 10^{15} *floating point operations per second*

1. What is the speed of your PC?
   probably between 1 and 6.5 GFlops

A little quiz ...

1 Kflops = 10^3, 1 Mflops = 10^6, 1 Gflops = 10^9, 1 Tflops = 10^{12},
1 Pflops = 10^{15} *floating point operations per second*

1. What is the speed of your PC?
2. What is the speed of the fastest computer currently available (and where is it located)
3. What was the speed of the fastest computer in
   a) 1995?
   b) 2000?
   c) 2005?

A little quiz ...

1 Kflops = 10^3, 1 Mflops = 10^6, 1 Gflops = 10^9, 1 Tflops = 10^{12},
1 Pflops = 10^{15} *floating point operations per second*

1. What is the speed of your PC?
2. What is the speed of the fastest computer currently available (and where is it located)

367 Tflop, it is a Blue Gene/L in Livermore/ California with >130 000 processors

A little quiz ...

1 Kflops = 10^3, 1 Mflops = 10^6, 1 Gflops = 10^9, 1 Tflops = 10^{12},
1 Pflops = 10^{15} *floating point operations per second*

1. What is the speed of your PC?
2. What is the speed of the fastest computer currently available (and where is it located)
3. What was the speed of the fastest computer in
   a) 1995?
   b) 2000?
   c) 2005?

A little quiz ...

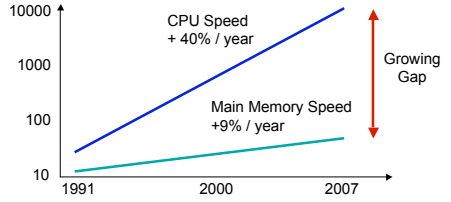
1 Kflops = 10^3, 1 Mflops = 10^6, 1 Gflops = 10^9, 1 Tflops = 10^{12},
1 Pflops = 10^{15} *floating point operations per second*

1. What is the speed of your PC?
2. What is the speed of the fastest computer currently available (and where is it located)
3. What was the speed of the fastest computer in
   a) 1995? 0.236 TFlops
   b) 2000? 12.3 TFlops
   c) 2005? 367 TFlops

... and how much has the speed of cars/airplanes/... improved in the same time?

Introduction – Why?

- Bottleneck "Memory Wall" getting worse



Fast Caches Strongly Needed !
Is this still valid (see later)?

Introduction - Focus

- Architectural Background
- How to find and isolate bottlenecks
- Strategies to speed up "**sequential**" numerical code
  Parallelization should complement cache-/sequential optimizations, but is not a substitute for bad node performance)

Our motivation …

How fast should a solver be?

- Poisson problem can be solved by a multigrid method in < 30 operations per unknown (known since late 70ies, see talks by Achi Brandt)
- More general elliptic equations may need O(100) operations per unknown
- A modern CPU could do 5-10+x GFLOPS
- So we should be solving 50-100 million unknowns per second
- Should need O(2.4) Gbytes of memory

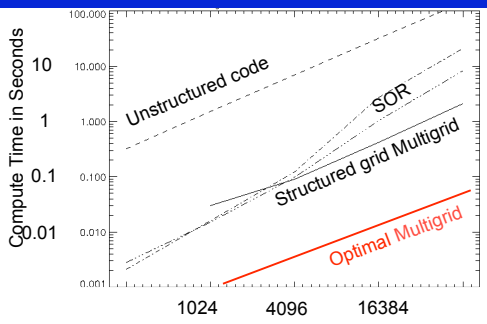
How fast are solvers today?

- Often no more than 100,000 to 1,000,000 unknowns possible before the code breaks
- In a time of minutes to hours (or days)
- Needing horrendous amounts of memory

Even state of the art codes are often very inefficient

- But: see e.g. paper on vectorizing multigrid by A. Brandt, 1983
- Our own (world?) record: $1.7*10^{10}$ unknowns in 42 seconds on 1024 processors - see Supercomputing 2005 and ISC award 2006 -> T. Gradl

Comparison of solvers



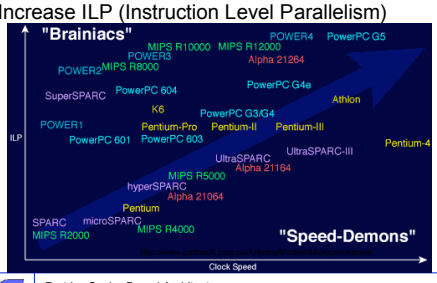
Part I: Cache-Based Architectures

CPU Architectures - Overview

- Goal: Understand performance issues
- Techniques used
  - in CPUs: Pipelining, Superscalar, OOO, …
  - for main memory
  - for acceleration of memory access (caches)

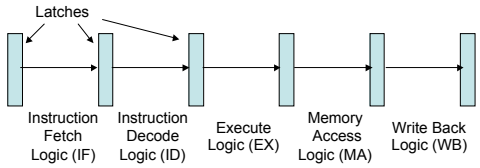
How to make a fast CPU ?

- Increase Clock Rate
- Increase ILP (Instruction Level Parallelism)

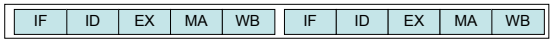


Pipelining

- Pipelined Instruction Execution (RISC)

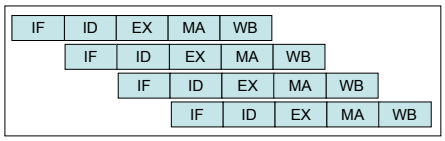


- Sequential Execution (5 CPI)



Pipelining (Cont'd)

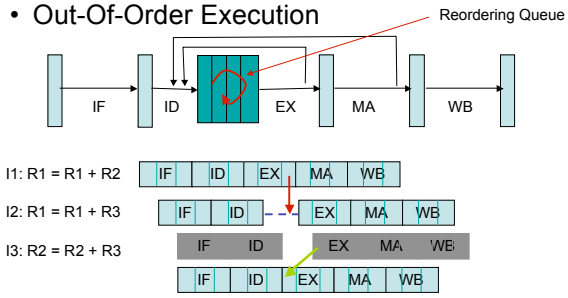
- Pipelined execution (1 CPI)



  - Slowest phase/stage determines cycle time
  - All stages executed for all instructions
  - Data / external dependences causes stalls

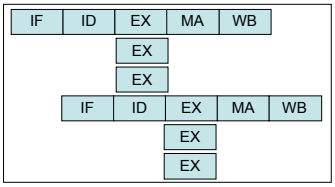
Minimize Data Deps. (Cont'd)

- Out-Of-Order Execution



Pipelining 2 (Cont'd)

- Pipeline with VLIW/EPIC Instructions



  - Multiple Parallel Actions per Instruction
  - Data Dep. Analysis moved to Compiler

Pipelining - Examples

Processor	Pipeline Stages	Superscalar Issue	Reordering	GHz
PowerPC G3/G4	4-7	3	OOO	0.3/1.3
MIPS R10000	5-7	4	OOO	0.195
Alpha 21164	7-9	4	In-Order	0.4
PowerPC G4e	7-12	4	OOO	1.7
Itanium-II	8-10	6 (EPIC)	In-Order	1.3-1.7
UltraSPARC	9	4	In-Order	0.3
Athlon	10-15	3	OOO	1.3 - 2.8
Pentium-II/III	12+	3	OOO	0,3 - 1.0
UltraSPARC-III	14	4	In-Order	1.2
PowerPC G5	16-25	5	OOO	2.0
Pentium-4	20-31	3	OOO	- 3.8
Opteron	12	3	OOO	-2.8
Core2	14	4	OOO	-3

Further CPU Techniques

- Data parallelism (SIMD instructions)
  - Large registers (e.g. 128 Bit), independent operations (e.g. Bit0-63, 64-127)
  - Examples
    - SPARC: VIS
    - Alpha: MVI
    - X86: MMX/SSE/SSE2, 3DNow!
    - PowerPC: AltiVec
  - Useful for HPC, Multimedia

Summary

- Pipelining technique good to increase both clock rate & ILP
- Problematic (leading to pipeline stalls)
  - Data dependencies
    - Use bypasses, OOO, speculation/prediction
  - External dependencies (memory accesses)
    - Use speculation (?), caches (!)
    - Very long stall times: – 300 cycles (3 GHz P-4)

Main Memory

Main Memory – The Problem

- Memory speed has 2 sides
  - Bandwidth
  - Latency
- Main memory is Off-Chip
  - Fixed limit wire length / speed of light
  - High Frequencies are difficult to handle for board manufactures
  - Less engineering progress…

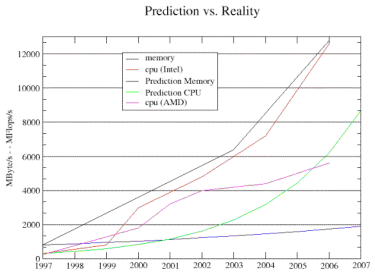
Main Memory - Bandwidth

- Possibilities for improvement (for both CPU-Chipset, Chipset-Memory)
  - Higher data width (more bits in parallel)
  - Higher bus clock rate, double/quad-pumped (e.g. DDR)
  - Memory bank interleaving
  - Long bursts
  - High-speed serial links (HyperLink, Rambus) easier for off-chip

Main Memory – Bandwidth (Cont'd)

- Example on Improvement:
  - 100 MHz Mem. Bus, 64 Bit, SDRAM (PC100): max. 800 MB/s (ca. 1997)
  - 200 MHz Mem. Bus, 128 Bit (2 Banks), DDR 400: max. 6,4 GB/s (ca. 2003)
  - 400 MHz Mem. Bus, 128 Bit (2 Banks), DDR2 800: max. 12,8 GB/s (present)

Real Bandwidth Measurements



Main Memory Speed - Latency

- Latency for SDRAM typically 9 Cycles (at Memory Bus Clock Rate!)
  - 1 from CPU to Chipset
  - 1 from Chipset to RAM Chip
  - 2 for RAS-to-CAS Delay (for Page Miss)
  - 2 for CAS Latency
  - 1 to get data to Output Buffer of RAM Chip
  - 1 from RAM Chip to Chipset
  - 1 from Chipset to CPU

The Memory Wall

- Bandwidth needs of CPUs
  - (2 DP load + 1 store)/cycle an 3 GHz: 72 GB/s (Pentium 4)
  - At best without latency at all
- Memory can deliver
  - Max. 6,4 GB/s (Itanium-II)
  - Best latency ~ 66 ns (= 121 Cyc. at 1,8GHz)
- Gap will be growing

The Memory Wall - Solution

- Caches
  - Small but fast on-board buffers holding copies of main memory content
  - Requirements
    - High bandwidth: Data paths with large width ("No Problem" on-chip)
    - Small latency: Tightly coupled to registers (Better use cache hierarchies!)
  - Only works when cache copies are reused!

Cache Design

Caches Work when…

- Memory access stream of an application shows some locality ("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

Basic Cache Properties

- Cache work on copies of memory blocks
  - Called "*cache lines*"
  - Size typically 32 - 128 bytes, based on Optimum for data bus and memory design
  - Transfers to main memory always at block granularity
  - Blocks exploit spatial locality
  - Each block knows its memory address (tag)

Basic Cache Properties (Cont'd)

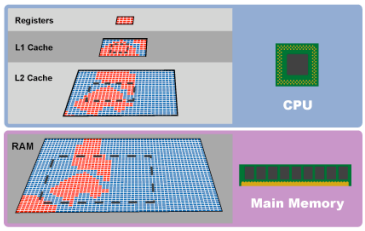
- Where gets a copy of a given memory block placed ?
  - In every cache line: **Full Associativity**
    - Best, but very costly (access time!)
    - Needs a compare for every line and access
  - In *N* Cache Lines: **N-Way Associativity**
    - Cache partitioned in sets with *N* lines each
  - Special case *N*=1: **Direct Mapped**
  - For N-Way/DM: eviction of lines possible even if cache is not full (conflict case)

Basic Cache Properties (Cont'd)

- Requests are satisfied from a cache
  - if it holds the appropriate copy: **Cache Hit**
  - if data is not in cache: **Cache Miss**
- When cache is full on a miss
  - Which cache line to replace? **Replacement Policy**
    - Theor. best: longest unused Copy (MIN)
    - Often used: LRU (least recently used)
    - Others: LFU (least frequently used), RANDOM
    - Promising are adaptive policies based on locality behavior
  - Not applicable to Direct-Mapped

Basic Cache Properties (Cont'd)

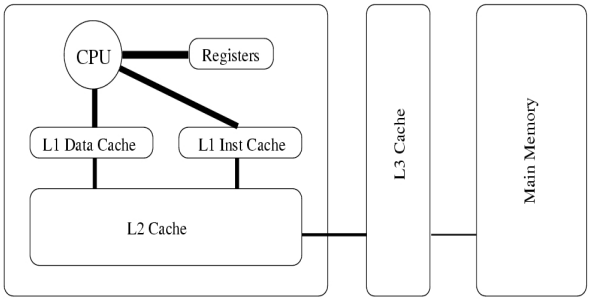
- Multiple Levels: L1, L2, sometimes L3



Basic Cache Properties (Cont'd)

- On a write access:
  - Create copy in cache (allocation policy)?
  - If Yes (write policy):
    - Also update slower levels/main memory: **Write Through**
      - Constant high bus traffic
    - Keep modified memory line in cache, write at eviction time: **Write Back**
      - Needs a "dirty" bit per cache line
      - Less bus traffic, risk of "write storms"

Cache Configurations (Cont'd)



Cache Issues

- Transparency for user/programmer
- Data consistency with main memory and other Caches on SMP machines
  - Cache consistency protocol
  - On SMP: MESI (via bus snooping)
  - On NUMA: directory based (node lists for every cache line)

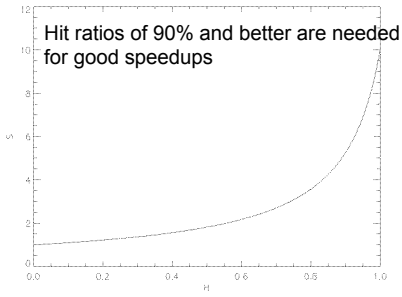
Effect of Cache Hit Ratio

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

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

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

Cache effectiveness depends on the hit ratio



Summary: Cache Organization

- Number of cache levels
- Set associativity
- Physical/virtual/hybrid addressing
- Write-through/write-back policy
- Replacement policy
- Cache line size

Cache Organization (Cont'd)

Processor	Size [kB] L1D/L2	Ass. L1D/L2	Line Size	L1 Hit Latency
Pentium-II/III	16	4	64	3
UltraSPARC	16	1	32	2
Itanium I	16/96	4/6	64	2
Itanium II	16/256/3MB	4	64	**1**
Alpha 21264	64/4MB	2/1	32/64	2
Pentium-4	16/2MB	8/8	64	**1**(3)
Athlon	64/256	2	64	3
PowerPC 3	64/4MB	8/1	128	2
PowerPC 4	32/1.5MB	8/2	128	3
Opteron	64/1MB	2/16	64	3
Core2	32/4MB	8/16	64	3

Cache Prefetching

- Memory latency can (sometimes) be hidden by computation (overlapping)
- Idea: Use memory bus when it would be idle (CPU is working/using data from L1/2)
  - Prefetch data into L2 which is used somewhat in the future
  - Software prefetching: programmer hints
    - Which address, How used (only once/multiple times)
    - Never raises exceptions
  - Hardware prefetch: Predict future from recent access stream of application

Cache Prefetching (Cont'd)

- HW-Prefetching
  - Stop prefetching at VM page boundaries
  - Algorithms
    - Often used: Stream Up/Down Detection Pentium-4 (8) ,P-M (16)
    - Stride detection (indexed by instruction address)
    - Address correlation predictors e.g. for linked lists (global detection of address diff. patterns)
  - Can improve performance significantly

References Part I

- Jason Patterson: Modern Microprocessors 90 Minute Guide (http://www.pattosoft.com.au/Articles/ModernMicroprocessors)
- Microprocessor Report (http://www.mdronline.com/mpr)
- Ars Technica (http://arstechnica.com)
- Intel, AMD, IBM: Processor Manuals
- Sandpile x86 processor specs (http://www.sandpile.org)

Part II - Programming Techniques

Basic efficiency guidelines

- **Choose the best algorithm !**
- Use efficient libraries
  - e.g. use DiMe librarary for 2D multigrid (J. Treibig/ M. Stürmer) or HHG for 3D Finite element multigrid (T. Gradl)
- Find good compiler and options
- Use suitable data layout and other architecture aware program optimization techniques

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
- E.g. SOR is (relatively) simple to optimize so that a well-tuned SOR program may outperform a poorly implemented multigrid method on moderately sized problems.

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

Optimization Techniques

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

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

Example: DAXPY operation

```
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
```

```
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
```

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= y(i)
  t2= y(i+1)
  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: 6 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)
  sum = sum + b(i)
enddo
```

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)`

Why should aliasing be forbidden?

```
tb = b(1)
tc = c(1)
do i=1,n-1
   ta = tb+2.d0*tc
   tc = c(i+1)

   sum = sum + tb
   tb = b(i+1)

   a(i) = ta
enddo
i = n
ta = tb + 2.0 *tc
sum = sum +tb
a(i) = ta
```

Assume that each block of ops can be executed in 1 cycle
Latency 1 cycle for load
Latency 2 cycles for a flop

Programm produces wrong results when used with second call a=b

Why should aliasing be forbidden? (cont'd)

```
do i=1,n
   LOAD tc = c(i)
   LOAD tb = b(i)
   ta = tb * 2d0 * tc
   NOOP
   STORE a(i) = ta
   LOAD sum
   LOAD tb = b(i)
   sum = sum + tb
   STORE sum
enddo
```

Consider version correct with aliasing

NOOP indicates CPU waiting because of dependencies

Explicit LOAD/STORE

Couldn't we save LOAD/STORE of sum?

Aliasing (cont'd)

- Aliasing forbidden in FORTRAN (result undefined when used)
- 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*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

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

- 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`?

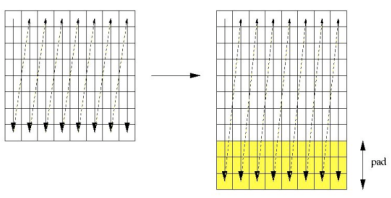
Data layout optimizations: Array padding

- 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 (cont'd)

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)

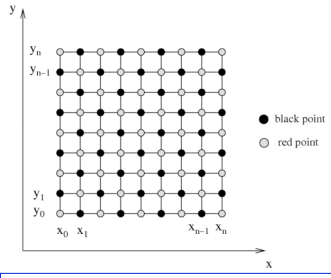
After:

```
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 (cont'd)

Example: red/black Gauss-Seidel iteration in 2D



Data access optimizations: Loop fusion (cont'd)

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
```

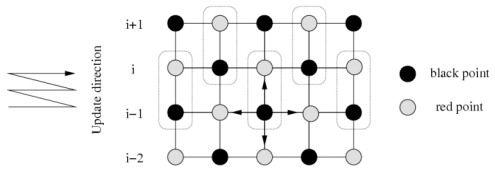

Data access optimizations: Loop fusion (cont'd)

```
   // 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 (cont'd)

How the fusion technique works:



Data access optimizations: Loop fusion (cont'd)

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 (cont'd)

```
// 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 (cont'd)

```
// 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

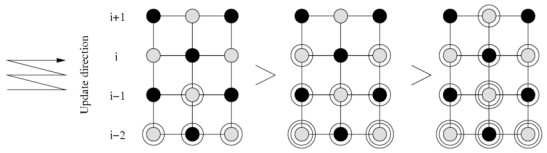
- The *inverse* transformation of loop fusion
- Divide work of one loop into two to make body less complicated
  - Leverage compiler optimizations
  - Enhance instruction cache utlization
  - Prevent resource bottlenecks (Write-Combine Buffers)

Data access optimizations: Loop blocking

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

Data access optimizations: Loop blocking

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



Data access optimizations: Loop blocking

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

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


Data access optimizations: Loop blocking

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

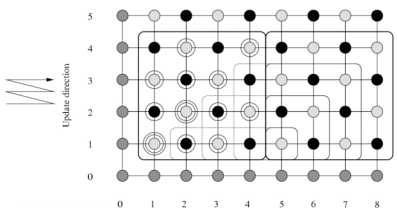

Data access optimizations: Loop blocking

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

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

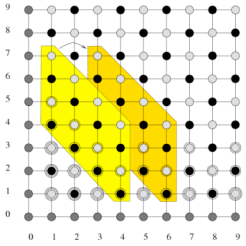
Data access optimizations Loop blocking

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



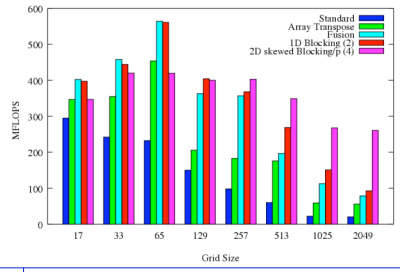
Data access optimizations: Loop blocking

- Illustration: 2D skewed blocking



Performance results

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 (cont'd)

- **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 (cont'd)

- 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

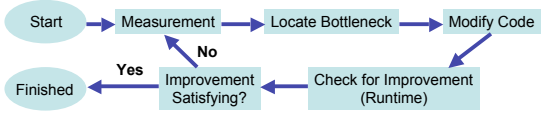
Memory access behavior (cont'd)

- 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 Analysis (Cont'd)

- How to do Performance Analysis
  - At End of (fully tested) Implementation
  - On Compiler-Optimized Release Version
  - With typical/representative Input Data
  - Steps of Optimization Cycle

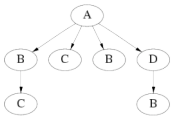


Performance Measurement

- Wanted:
  - Time partitioning with
    - Reason for performance loss (stall because of…)
    - Detailed relation to source (code, data structure)
    - Suitable metrics to get hints for improvements
  - Runtime numbers
    - Call relationships, call numbers
    - Loop iterations, jump counts
  - No perturbation of results b/o measurement

Measurement - Terms

- Trace
  - Record of time-stamped event stream
    - Enter/Leave of code region, actions, …
      Example: dynamic call tree



    - Huge amount of data (linear to runtime)
    - Unneeded for sequential analysis (?)

Methods - Instrumentation

- Manual
- Source instrumentation
- Library version with instrumentation
- Compiler
- Binary editing
- Runtime instrumentation / compiler
- Runtime injection

Methods (cont'd)

- Statistical measurement ("sampling")
  - TBS (Time based), EBS (Event based)
  - Assumption:
    Event distribution over code approximated by checking every N-th event only
  - Similar way for iterative code:
    Measure only every N-th iteration
- Data reduction tunable
  - Compromise between quality/overhead

Methods (cont'd)

- Simulation
  - Events for (non-existant ?) HW models
  - Results not influenced by measurement
  - Compromise quality / slowdown
    - Rough model = high discrepancy to reality
    - Detailed model = best match to reality
      But: reality (CPU) often unknown…
  - Comparison of models with changes of architecture parameters possible

Hardware Support

- Monitor Hardware
  - Event Sensors (in CPU, on Board)
  - Event Processing / Collection / Storing
    - Best: Separate HW
    - Comprimise: Use Same Resources after Data Reduction
  - Most CPUs nowadays include
    Performance Counters

Tools – Example 2

- PCL Hardware performance monitor: hpm
- Run:

```
hpm --events PCL_CYCLES, PCL_MFLOPS prog

hpm: elapsed time: 5.172 s
hpm: counter 0    : 2564941490 PCL_CYCLES
hpm: counter 1    : 19.635955 PCL_MFLOPS
```

(Similar: Perfex, Pfmon, hpccount)

Tools – Example 2 (Cont'd)

```
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;
    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;
}
```

Event Types to Measure

Start

Stop

Similar: PAPI, Pflib (Itanium)

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

Example: Optimization of 3D Geometric Multigrid on the Itanium2 Processor

The Itanium 2 Processor

- simple logic...
  - in-order execution
    - but speculative when branching
  - simple issue logic
    - explicitly parallel instruction computing (EPIC)
    - code generator must prove that parallel execution is possible
  - no hardware prefetcher
    - code generator must identify data streams and generate prefetch instructions

The Itanium 2 Processor

- ... but vast resources
  - large register set (128x 64 Bit GP, 128 FP ...)
  - multiple fully pipelined execution units
    - up to 6 operations per cycle
    - 4 load or 2 load + 2 store
    - 2 fully pipelined FP units (fused-multiply-add)
    - 6 simple (2 complex) integer operations
  - huge caches with high bandwidth
    - 1.5 to 12 MB of L3 Cache at core speed

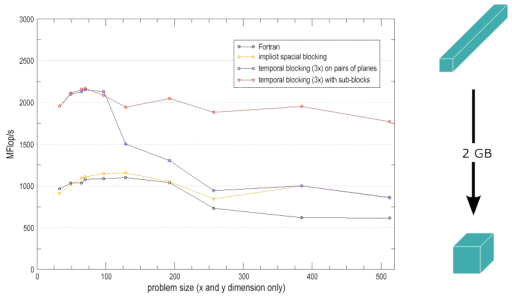
Challenges on the Itanium 2

- execution speed highly dependent on code quality
  - scheduling
    - considering memory and instruction latencies
    - create groups of parallel executable instructions
  - use of special capabilities of the IA-64 architecture
    - predicated execution can minimize branches
    - rotating registers enable high-throughput loops
    - software prefetching of data streams

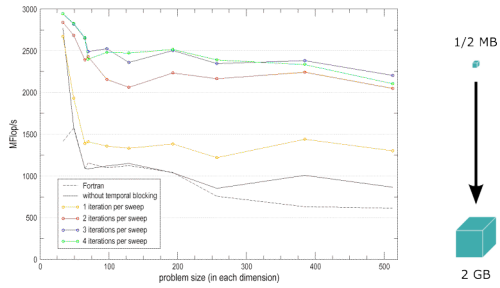
Overview of Code-Optimizations

- compute intensive parts written in assembly
- optimized memory layout
- temporal and spatial blocking
- enhanced prefetching techniques
  - follow jumps in data streams
  - smooth the data bursts generated by blocking techniques
- thread level parallelization (memory bus saturization)
- smoother (RBGS) is the most time consuming part

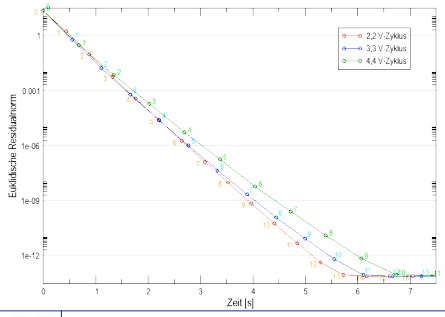
Temporal Blocking 3D RBGS



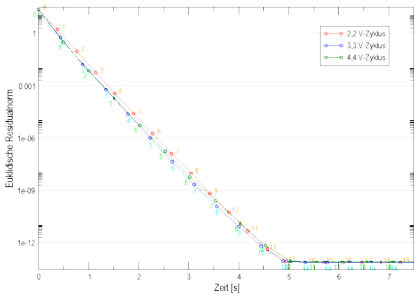
Results: Blocking



Convergence: 257^3



Convergence: 257^3 (parallel)



Example: Cache Optimizations for the Lattice Boltzmann Method

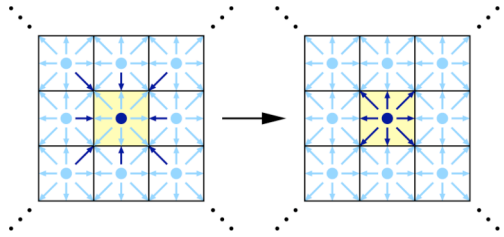
Lattice Boltzmann method

- 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*

Demo

LBM

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

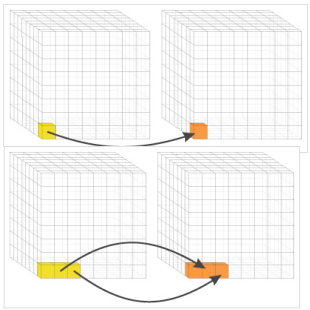


Data layout optimization

Layout 1:

Two separate grids

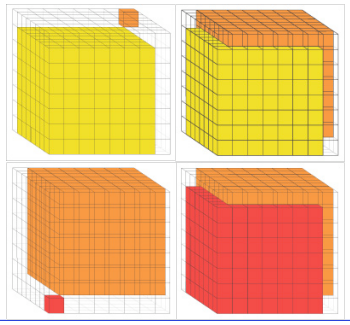
(standard approach)



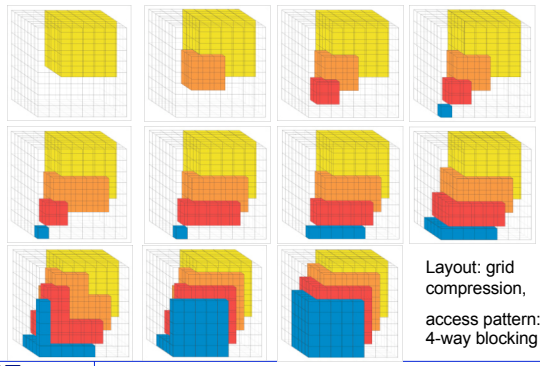
Data layout optimization (cont'd)

Layout 2:

Grid Compression:
save memory,
enhance locality



Data access optimizations (cont'd)

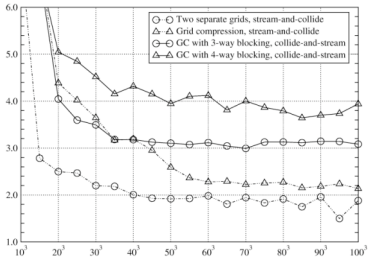


Layout: grid compression,

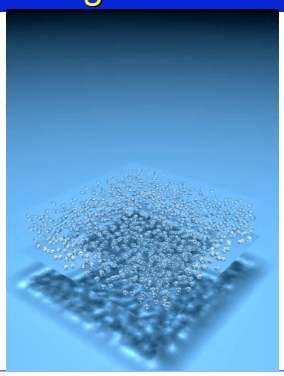
access pattern: 4-way blocking

Performance results (cont'd)

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

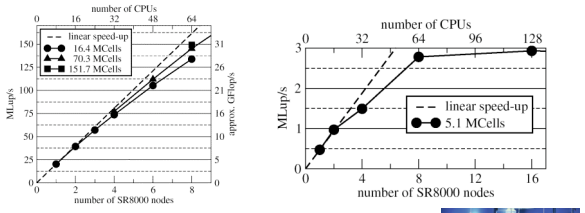


Foaming Simulation 1



Performance on SR 8000

Standard LBM-Code ⟷ Free surface LBM-Code



Performance lousy on a single node!
Conditionals: 2,9 SLBM → 51 free surface LBM
Pentium 4: almost no degradation ~ 10%
SR 8000: enormous degradation (pseudo-vector, predictable jumps)

Fun Simulations

Thanks for your interest

Any Questions?

Example Architectures: Intel P4 and AMD Opteron

Intel Pentium 4

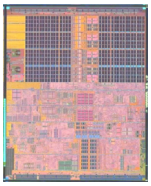
- Present x86 CPU generation from Intel
- Netburst Architecture
- Versions: Prescott (90nm), Cedar Mill (65nm), Presler (65nm, MC)
- Released 2000 with 1,5 GHz
- RISC OOO and Speculative Execution
- 31 stage pipeline (reaching 3,8 GHz)
- Superscalar: 7 Integer ALUs and 1 FPU
- Execution rate: 3 microOps/cycle

Intel Pentium 4

- Can perform one FMA each cycle
- Implements SSE instruction set extensions
- Implements x86-64 ISA
- Hardware support for multithreading (called Hyperthreading)
- Uses Trace Cache as L1 instruction cache
- Hardware prefetch unit: 1 stream per 4k page, up to 8 streams, 512 bytes ahead
- Data TLB 128 Entries

Intel Pentium 4

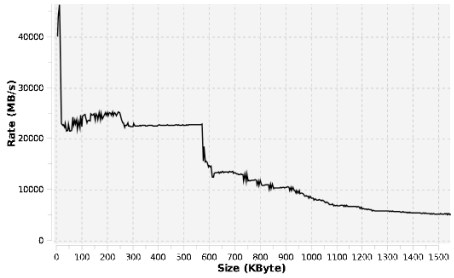
- 16 kB L1 Data Cache:
  - 8 way associative
  - 64 byte cacheline length
  - Write Through policy
  - Pseudo LRU
- 2048 kB L2 Data Cache:
  - 8 way associative
  - 64 byte cacheline length (2 lines sectored)
  - Write Back policy
  - 256 Bit Bus



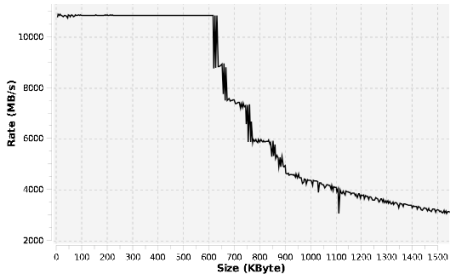
Memory Subsystem: Performance

L1 Cache Latency	1-4
L2 Cache Latency	Min 21 (average 56)
Memory read	5810 MByte/s
Memory write	4018 MByte/s
Memory copy	4651 MByte/s

Cache Characteristics Read



Cache Characteristics Write



AMD K8 (Athlon 64, Opteron)

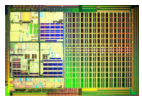
- AMDs evolution for the x86 architecture
- First implementation of the x86-64 ISA
- On chip memory controller
- Released 2002 with 1.6 GHz
- Recent Versions: Venice, San Diego, Toledo (MC), all 90 nm
- RISC OOO and Speculative Execution
- FP pipeline 17 staged (reaching 2,8 GHz)
- Superscalar:

AMD K8 (Athlon 64, Opteron)

- Can perform one FMA each cycle
- Implements x86-64 ISA
- Implements SSE instruction set extensions
- Data TLB 512 entries
- Hardware Prefetch Unit

AMD K8 (Athlon 64, Opteron)

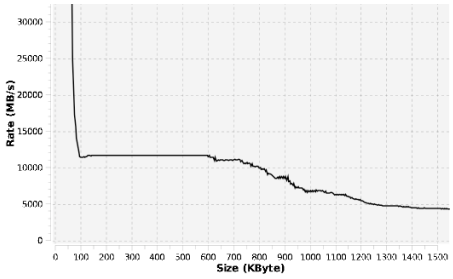
- 64 kB L1 Data Cache:
  - 2 way associative
  - 64 byte cacheline length
  - Write Back, Write Allocate Policy
  - LRU
- 1024 kB L2 Data Cache:
  - 16 way associative
  - 64 byte cacheline length
  - Pseudo LRU



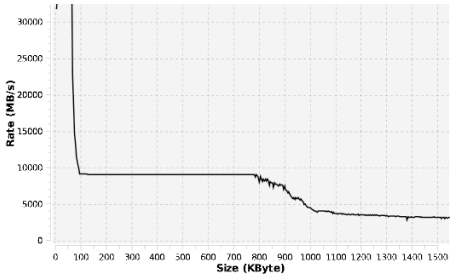
Memory Subsystem: Performance

L1 Cache Latency	2-3
L2 Cache Latency	Min 11 (average 13)
Memory read	6096 MByte/s
Memory write	6058 MByte/s
Memory copy	6035 MByte/s

Cache Characteristics Read



Cache Characteristics Write



SSE instruction set extensions

- 1997 Intel introduced MMX (57 instr.)
- Pentium 3: SSE adds FP instr. (70 instr.)
- Pentium 4: SSE2 adds double (144 instr.)
- Pentium 4 Prescott : SSE3 (13 instr.)

All these extensions among other things provide instructions for SIMD and a more explicit cache control.

SSE instruction set extensions

- Adds 8/16 128 bit wide registers which can hold 2 doubles or 4 floats
- Data has to be 16 byte aligned
- New instructions include:
  - 16 byte loads and stores
  - Non temporal stores (SSE)
  - SIMD Packed instructions
  - Software Prefetch instructions (SSE)
  - Cacheline invalidate instruction (SSE2)

X86-64

- 64 bit integers and addresses
- Doubles register sets (16 GPR, 16 xmm)
- SSE extensions are adopted as core instructions
- No execute bit, not relevant for scientific computing
- Cleanup of deprecated x86 features
- ABI has changed (e.g. subroutine parameters are passed in register)

In Cache Peak Flop Rate

	Intel Pentium 4 Prescott 3,2 GHz Peak 6400/12800 MFlops		AMD Athlon64 4000+ 2.4 GHz Peak 4800/9600 MFlops	
Synthetic Flops FMA Double RO	5653 MFlops (88%) 22615 MByte/s		4499 MFlops (94%) 17998 MByte/s	
Synthetic Flops FMA Double RW	2687 MFlops (41%) 21500 MByte/s		3149 MFlops (65%) 25198 MByte/s	
Vector Triade: A=A+alpha*B	3128 MFlops 25025 MByte/s (49%)	2668 Mflops 32025 MByte/s (41%)	3193 MFlops 25549 Mbyte/s (66%)	2605 Mflops 31260 Mbyte/s (54%)
Synthetic Flops FMA Single RO	10904 Mflops (85%) 21808 MByte/s		8990 Mflops (94%) 17981 MByte/s	

Peak Memory Bandwidth

•Vector Triade A=B+scalar*C
•Assembler version using block prefetching and vectorization
•Theoretical memory bandwidth is 6,4 GByte/s

	Intel Pentium 4 Prescott 3,2 GHz		AMD Athlon64 4000+ 2.4 GHz	
Triade Compiler icc 9.1 / gcc 4.0	4057 Mbyte/s	3060 Mbyte/s	4322 Mbyte/s	3071 Mbyte/s
Triade Assembly	4855 Mbyte/s		5170 Mbyte/s	

Conclusion: P4 vs. Opteron

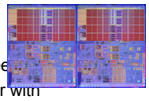
- The Opteron has a better memory subsystem
- P4s target cache is the L2 while the target cache on the Opteron is the L1
- P4 is better suited for streaming structured applications
- Opterons low latency design is well suited for OO unstructured applications
- The P4s performance strongly relies on vectorization (SSE)

Future Perspectives

- The run for higher clock rates has come to an end
- The question arises what to do with all the transistors
- There is a trend to reduce the hardware logic in favor of specialized execution units
- In mainstream processors multicore chips are already available
- The memory subsytem gets more in the focus of system architects

Future Perspectives cont.

- There seem to be three major trends:
  - Multicore processors
  - Off Chip specialized Coprocessors (GPU, Clearspee
  - Specialized Execution Units on chip: Cell Processor with strong SIMD capabilities
- But: To get performance on these new platforms you have to program low level and fit your program to the platform to benefit.



Introduction to assembly language (x86-64)

Function Skeleton

```
.text
.globl triad_asm
.type triad_asm, @function

triad_asm:

push rbp
mov rbp, rsp
/* Do something */
mov rsp, rbp      /*equivalent to leave */
pop rbp

ret
.size triad_asm, .-triad_asm
```


Declaring Data

```
.data
.align 16
ALPHA:
.double 1.0,1.0
```


loops in assembly

```
xor rax, rax   /* zero register rax */

.align 16
.loop:

movapd  xmm1, [rdx + rax*8]

add rax, 8
cmp rax, 1000
jb .loop  /* conditional branch*/
```

- Many other possibilities to control the loop.
- Often a second register for the stopping criterion is needed

Triade: Basic Version

Prototype in C: void triad_asm(double* A, double* B, double* C, int size)

```
movapd  xmm0, [ALPHA]

xor rax, rax   /* zero register rax */
.align 16
.loop:

movapd  xmm1, [rdx + rax*8]
movapd  xmm2, [rsi + rax*8]
mulpd   xmm2, xmm0
addpd   xmm1, xmm2
movapd [rdi + rax*8], xmm1

add rax, 1
cmp rax, rcx
jb .loop  /* conditional branch*/
```


First Optimization: loop unrolling

```
movapd  xmm0, [ALPHA]

xor rax, rax   /* zero register rax */
.align 16
.loop:

movapd  xmm1, [rdx + rax*8]
movapd  xmm2, [rsi + rax*8]
mulpd   xmm2, xmm0
addpd   xmm1, xmm2
movapd [rdi + rax*8], xmm1

add rax, 1
cmp rax, rcx
jb .loop  /* conditional branch*/
```

Enabling SIMD on modern processors

SIMD: Introduction

- SIMD means Single Instruction Multiple Data
- It is suited for the enhanced execution of similiar arithmetic operations on multiple input streams
- Typical SIMD implementations are classical vector machines as the NEC SX series
- Many modern microprocessors have instruction set extension to support SIMD (SSE and 3DNow on x86, Altivec on PowerPC)
- SIMD enables a better performance for streaming algorithms on all platforms

SSE: SIMD on the Desktop

- SSE is an instruction set extension, different from e.g. improvements of the hardware prefetcher or cache design it is necessary to recompile your code to use these new instructions
- Intel has put a lot of effort into enabling a good autovectorization by the compiler
- Still applying SIMD can be tricky and complicated for the compiler

Enable SSE usage Intel Compiler

- Allocate memory at least 16 byte aligned and inform the compiler with `#pragma vector aligned`
- Compile for a specific processor to enable vectorization (-xW, -xP)
- Turn on vectorization report with -vec-report3
- Declare your pointers restrict to indicate the absence of aliasing
- Keep the innermost loop body simple
- Prevent complicated nested loop structures

Enable SSE: Using pragmas

- Place the pragma `#pragma vector always` in front of an innermost loop control statement
- Use the pragma `#pragma ivdep` to override assumed vector dependencies
- Enable streaming stores for large vector lengths using pragma `#pragma vector nontemporal`
- Enable software prefetching with `#pragma prefetch`
- For more details on compiler pragmas have a look on the Intel Compiler Handbook

Enable SSE: Using pragmas

- Example Code:

```
void foo (double *a)
{

       #pragma vector aligned
       #pragma vector ivdep
       #pragma vector always
       for (i=0;i<m; i++){
              a[i] = a[i] * c;
       }
}
```

Intrinsics

- The usage of compiler options or pragmas is the preferred way to enable SSE because they enable compiler optimization while retaining the code portable
- Still in some cases using intrinsics may bring better results
- Intrinsics can be seen as high level assembly like instructions embedded into C or Fortran

Case Study: Vector Triad

- C Code:

```
void triad (double * restrict A, restrict double* B,
              restrict double *C)
{
       int i;
       double alpha=3.6;

       #pragma vector nontemporal
       #pragma vector aligned
       #pragma prefetch B C
       for (i=0;i<SIZE; i++){
              A[i] = B[i] + alpha * C[i];
       }
}
```


Case Study: Triad cont.

- Compiled with:

```
icc -S -fast -ansi -c99 -DSIZE=16777216 -o triad.s triad.c
```

```
..B1.2:                         # Preds ..B1.2 ..B1.1
       movapd    (%edx,%eax,8), %xmm1                          #12.24
       mulpd     %xmm0, %xmm1                                  #12.24
       addpd     (%ecx,%eax,8), %xmm1                          #12.24
       movapd    16(%edx,%eax,8), %xmm2                        #12.24
       movapd    32(%edx,%eax,8), %xmm3                        #12.24
       movapd    48(%edx,%eax,8), %xmm4                        #12.24
       movntpd   %xmm1, (%esi,%eax,8)                          #12.2
       mulpd     %xmm0, %xmm2                                  #12.24
       mulpd     %xmm0, %xmm3                                  #12.24
       mulpd     %xmm0, %xmm4                                  #12.24
       addpd     16(%ecx,%eax,8), %xmm2                        #12.24
       addpd     32(%ecx,%eax,8), %xmm3                        #12.24
       addpd     48(%ecx,%eax,8), %xmm4                        #12.24
       movntpd   %xmm2, 16(%esi,%eax,8)                        #12.2
       movntpd   %xmm3, 32(%esi,%eax,8)                        #12.2
       movntpd   %xmm4, 48(%esi,%eax,8)                        #12.2
       addl      $8, %eax                                      #11.5
       cmpl      $16777216, %eax                               #11.5
       jb        ..B1.2
```


Case Study: Triad cont.

- Handwritten assembly:

```
.addloop:
movapd  xmm2, [edx + eax*8]
movapd  xmm1, [esi + eax*8]
mulpd   xmm2, xmm0
addpd   xmm1, xmm2
movapd  xmm4, [edx + eax*8 + 16]
movapd  xmm3, [esi + eax*8 + 16]
mulpd   xmm4, xmm0
addpd   xmm3, xmm4
movapd  xmm6, [edx + eax*8 + 32]
movapd  xmm5, [esi + eax*8 + 32]
mulpd   xmm6, xmm0
addpd   xmm5, xmm6
movapd  xmm2, [edx + eax*8 + 48]
movapd  xmm7, [esi + eax*8 + 48]
mulpd   xmm2, xmm0
addpd   xmm7, xmm2
movntpd [edi + eax*8]      , xmm1
movntpd [edi + eax*8 + 16]  , xmm3
movntpd [edi + eax*8 + 32]  , xmm5
movntpd [edi + eax*8 + 48]  , xmm7
add eax, 8
cmp eax, 16777216
jb .addloop
```


Case Study: Triad cont.

- In this example it is trivial for the compiler to optimize the code, even without pragmas
- Still the code without pragmas needs 150 lines of assembler code while that with pragmas gets away with 50

compiler w/o pragmas	4059 MByte/s
compiler w pragmas	4095 MByte/s
assembly standard	4115 MByte/s
assembly optimized	4866 MByte/s

Case Study: Triad cont.

- Why is the optimized assembly code still so much better ?
- First of all the compiler does not prefetch anything, despite our pragma
- Lets try an intrinsic:

```
void triad (double * restrict A, restrict double* B,
              restrict double *C)
{
       int i;
       double alpha=3.6;
       #pragma vector nontemporal
       #pragma vector aligned
       #pragma prefetch B C
       for (i=0;i<SIZE; i++){
              _mm_prefetch((const char*) B+i*8+256, _MM_HINT_T1);
              _mm_prefetch((const char*) C+i*8+256, _MM_HINT_T1);
              A[i] = B[i] + alpha * C[i];
       }
}
```

Case Study: Triad cont.

```
..B1.2:                         # Preds ..B1.2 ..B1.1
        prefetcht1 256(%ecx,%eax,8)
        prefetcht1 256(%edx,%eax,8)
        movapd    (%edx,%eax,8), %xmm1                          #12.24
        mulpd     %xmm0, %xmm1                                  #12.24
        addpd     (%ecx,%eax,8), %xmm1                          #12.24
        movapd    16(%edx,%eax,8), %xmm2                        #12.24
        movapd    32(%edx,%eax,8), %xmm3                        #12.24
        movapd    48(%edx,%eax,8), %xmm4                        #12.24
        movntpd   %xmm1, (%esi,%eax,8)                          #12.2
        mulpd     %xmm0, %xmm2                                  #12.24
        mulpd     %xmm0, %xmm3                                  #12.24
        mulpd     %xmm0, %xmm4                                  #12.24
        addpd     16(%ecx,%eax,8), %xmm2                        #12.24
        addpd     32(%ecx,%eax,8), %xmm3                        #12.24
        addpd     48(%ecx,%eax,8), %xmm4                        #12.24
        movntpd   %xmm2, 16(%esi,%eax,8)                        #12.2
        movntpd   %xmm3, 32(%esi,%eax,8)                        #12.2
        movntpd   %xmm4, 48(%esi,%eax,8)                        #12.2
        addl      $8, %eax                                      #11.5
        cmpl      $16777216, %eax                               #11.5
        jb        ..B1.2
```


Case Study: Triad cont.

- Not that convincing: 4129 MByte/s
- While the Intel Optimization Guide suggests to insert prefetch instructions in the main loop experiments have shown that block prefetching works much better on the Pentium 4
- The implementation of the prefetch instruction moreover is not very efficient as it is only a hint to the CPU
- On many architectures (this may change on future processors) a cacheline wise preloading into register is the most effective way of prefetching

Case Study: Triad cont.

The basic algorithm for block prefetching looks like that:

```
void triad (double * restrict A, restrict double* B,
                    restrict double *C)
{
          int k;

          for (k=0;k<SIZE; k+=BLOCKSIZE){

                    prefetch_block(B,C);
                    compute_block(A,B,C);

                    A+=BLOCKSIZE;
                    B+=BLOCKSIZE;
                    C+=BLOCKSIZE;
          }

}
```


Case Study: Triad cont.

- Problem: We have to convince the compiler to generate desired code for us
- This is very difficult, even with intrinsics. One has always to check if the generated code on assembly level is the right one
- The actual arithmetic code of the compiler is optimal, we only have to assure that it stays optimal
- Still the preloading does not make any sense to the compiler

Case Study: Triad cont.

- The first try is to implement the cacheline wise prefetching in C (using bytewise moves)

```
char x, *arr;

arr=(char*) B;
for (i=0;i<BLOCKSIZE; i+=CLLENGTH){
          x= *(arr+i);
}

arr=(char*) C;
for (i=0;i<BLOCKSIZE; i+=CLLENGTH){
          x= *(arr+i);
}
```

If optimization is turned on the compiler just throws away above loops. With optimization turned off the loop code is so inefficient that our prefetching effort is for nothing. Even using intrinsics does not change that.

Case Study: Triad cont.

The following assembler snippet does the trick:

```
.prefetchloop1:
mov ebx, [esi + ebp *8  + 0]
mov ebx, [esi + ebp *8  + 128]
mov ebx, [esi + ebp *8  + 256]
mov ebx, [esi + ebp *8  + 384]
add ebp, 64
cmp ebp, 16384
jb  .prefetchloop1

xor ebp, ebp
align 16
.prefetchloop2:
mov ebx, [edx + ebp *8  + 0]
mov ebx, [edx + ebp *8  + 128]
mov ebx, [edx + ebp *8  + 256]
mov ebx, [edx + ebp *8  + 384]
add ebp, 64
cmp ebp, 16384
jb  .prefetchloop2
```

Placed into a function and called as external prefetching routine brings 4939 MByte/s.

Case Study: Bottomline

- Efficiency is generated on instruction level
- If the compiler does not generate efficient code, probably with compiler directives, in the first place it is hard to abuse him for generating a desired intruction code
- This process is tedious and often it is faster to write the necessary part of the code directly in assembly
- If you care about performance you need to check yourself if the compiler generated code is efficient or not.

The End

Pipelining - Examples

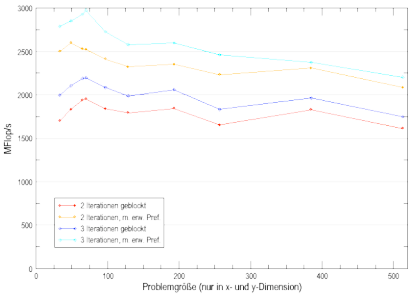
Processor	Pipeline Stages	Superscalar Issue	Reordering	GHz
PowerPC G3/G4	4-7	3	OOO	0.3/1.3
MIPS R10000	5-7	4	OOO	0.195
Alpha 21164	7-9	4	In-Order	0.4
PowerPC G4e	7-12	4	OOO	1.7
Itanium-II	8-10	6 (EPIC)	In-Order	1.3-1.7
UltraSPARC	9	4	In-Order	0.3
Athlon	10-15	3	OOO	1.3 - 2.8
Pentium-II/III	12+	3	OOO	0,3 - 1.0
UltraSPARC-III	14	4	In-Order	1.2
PowerPC G5	16-25	5	OOO	2.0
Pentium-4	20-31	3	OOO	- 3.5

Example: Optimization of 3D Geometric Multigrid on the Itanium2 Processor

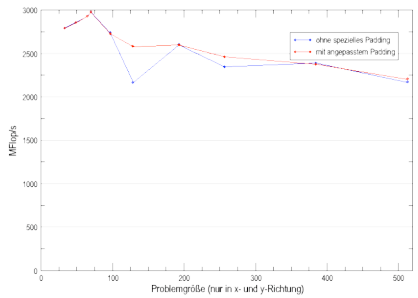
Overview

- The Itanium processor offers a lot of opportunities for low level optimization
- Optimizations applied include:
  - Temporal and spatial Blocking
  - all compute intensive parts are written in assembly
  - Enhanced prefetching mechanism to smooth memory bus usage
  - store red and black points in seperate arrays
  - Thread level parallelisation to saturize memory bus

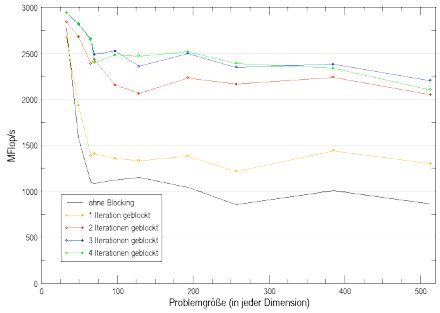
Enhanced prefetching in 3D, fixed problem size 2 GB



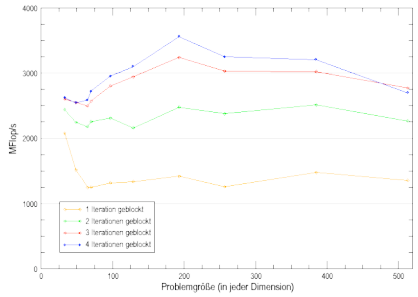
Padding



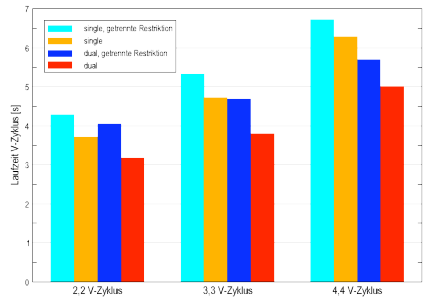
Results: Blocking



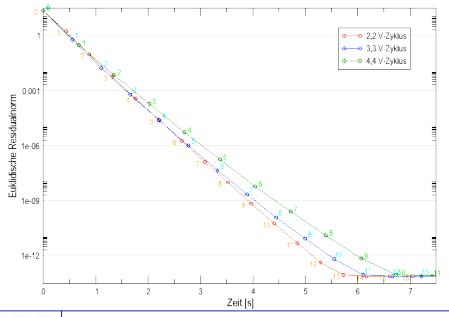
Results: 2 Threads



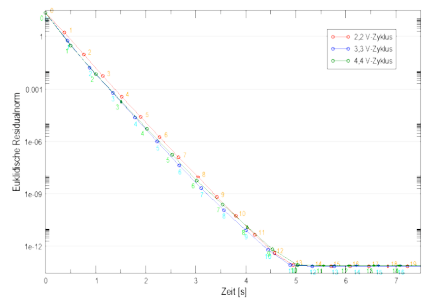
Results: 513^3



Convergence: 257^3



Convergence: 257^3 (parallel)



Multigrid 257^3 2,2-VCycle Profiling

	Gittergröße								
Komponente	**3**	**5**	**9**	**17**	**33**	**65**	**129**	**257**	
Vorglätter mit Residuum		<<0,1	<<0,1	<<0,1	<0,1	0,4	4,0	34,8	*39,4*
Restriktion		<<0,1	<<0,1	<<0,1	<<0,1	0,2	0,9	7,4	*8,5*
Initialisieren von V	<<0,1	<<0,1	<<0,1	<<0,1	<0,1	0,4	3,8		*4,2*
Interpolation		<<0,1	<<0,1	<<0,1	<<0,1	0,1	1,0	8,1	*9,3*
Nachglätter	<<0,1	<<0,1	<<0,1	<<0,1	<0,1	0,4	3,2		*3,7*
Nachglätter m. Res.-Norm								35,0	*35,0*
	<<0,1	*<<0,1*	*<<0,1*	*<0,1*	*0,2*	*1,5*	*12,9*	*85,3*	*100,0*