



Dr. Gerhard Wellein, Dr. Georg Hager  
 HPC Services –  
 Regionales Rechenzentrum Erlangen  
 Universität Erlangen-Nürnberg  
 Vorlesung an der FHN im SS 2008

### Format of lecture



- **4 days course: 3.-6.3.**
- **16 units (90 minutes each) in total**
- **2 lectures in the morning**
  - 8:30-10:00
  - 10:30-12:00
- **2 tutorials in the afternoon (180 minutes)**
  - 13:30-16:30
  - Exercises will be performed at RRZE cluster
- **7.3.: Visit to RRZE (??:??-??:??)**
- **10.3.: Exam (8:30-10:00 ??)**

## Topics of lecture



- **Day 1: Introduction & Single processor**
- **Day 2: (Shared memory) Parallelism, OpenMP & Multi-Core**
- **Day 3: Distributed memory parallelism, MPI & Clusters**
- **Presentations are available on the web:**  
*<http://www.blogs.uni-erlangen.de/hager/topics/OHN/>*
- **Exam: 60 minutes, no supporting material allowed**

Parallelrechner – Vorlesung im SS2008

(3)



## Survey (3.3.)



- **Introduction**
- **Single Processor: Architecture & Programming**
- **Microprocessors & Pipelining**
- **Memory hierarchies of modern processors**
- **Literature**

Parallelrechner – Vorlesung im SS2008

(4)



## Introduction



- **Parallel Computer (personal opinion):**  
*Multiple processors or compute nodes TIGHTLY connected*
- **Parallel Computing (personal opinion):**  
*Multiple processors solve cooperatively a SINGLE numerical problem*
- **(Massively) Parallel computing is the basic paradigm of modern supercomputer architectures**
- **Distributed resources connected via GRID technologies are**
  - neither a parallel computer nor
  - parallel computing can be done on it

Parallelrechner – Vorlesung im SS2008

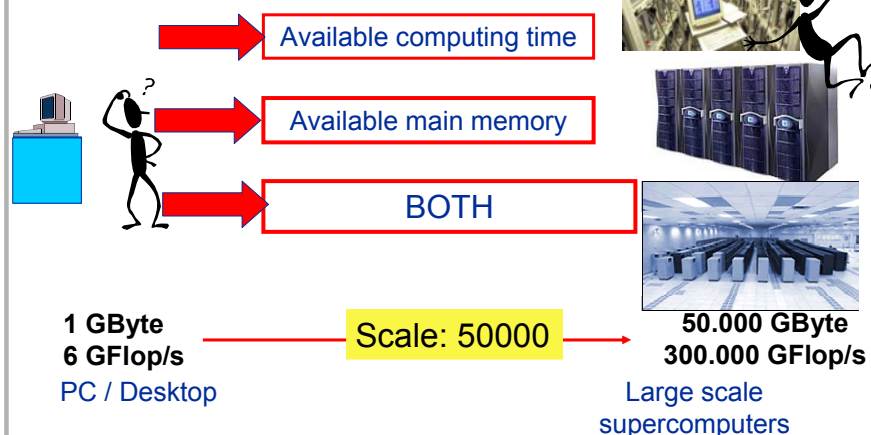
(5)

CX  
HPC

## Introduction



Traditionally we used parallel computers to overcome limitations of desktop computers..



.. but with Multi-Core parallel computing is entering the desktop...

Parallelrechner – Vorlesung im SS2008

(6)

CX  
HPC

### Last but not least TOP500 list



- **Comprehensive & state-of-the-art survey: TOP500 list**
- **Top 500: Survey of the 500 most powerful supercomputers**
  - <http://www.top500.org>
  - Solve a large system of linear equations:  $A \cdot x = b$  („LINPACK“)
  - Published twice a year (ISC Heidelberg/Dresden, SC in USA)
  - Established in 1993: 60 GFlop/s (Top1)
  - Nov. 2007: 472.800 GFlop/s (Top1)
  - Performance increase: 90 % p.a. !
- Performance measure: MFlop/s, GFlop/s, TFlop/s, PFlop/s
  - Number of FLOATING POINT operations per second
  - FLOATING POINT operations: double precision (64 bit) Add & Mult ops
  - $10^6$ : MFlop/s;  $10^9$ : GFlop/s;  $10^{12}$ : TFlop/s;  $10^{15}$ : PFlop/s

Parallelrechner – Vorlesung im SS2008

(7)



### Last but not least Top500 list as of November 2007



Rank	Site	Computer	Country	Procs.	RMax	RPeak
1	DOE/NNSA/ LLNL	IBM eServer Blue Gene /L Solution	U.S.	2007 212992	478200	596378
2	FZ Jülich	IBM eServer Blue Gene/P Solution	U.S.	2007 65536	167300	222822
3	New Mexico Appl. Center	SGI IB-Cluster+ Quad-Core Xeon	U.S.	2007 14336	126900	172032
4	TATA SONS	HP IB-Cluster+ Quad-Core Xeon	INDIA	2007 14240	117900	170880
5	Gov. Agency	HP IB-Cluster+ Quad-Core Xeon	USA	2007 13728	102800	146400
6	Sandia National Laboratories	Cray Red Storm Cray XT3, Opteron 2.4 GHz	U.S.	2006 53138	102200	127531
18	LRZ Munich	SGI Altix4700 Itanium2-1.6 GHz	Germany	2006 9728	56520	62259
329	RRZE Erlangen	HP Intel Xeon5160 3 Ghz Infiniband	Germany	2006 876	7315	10512

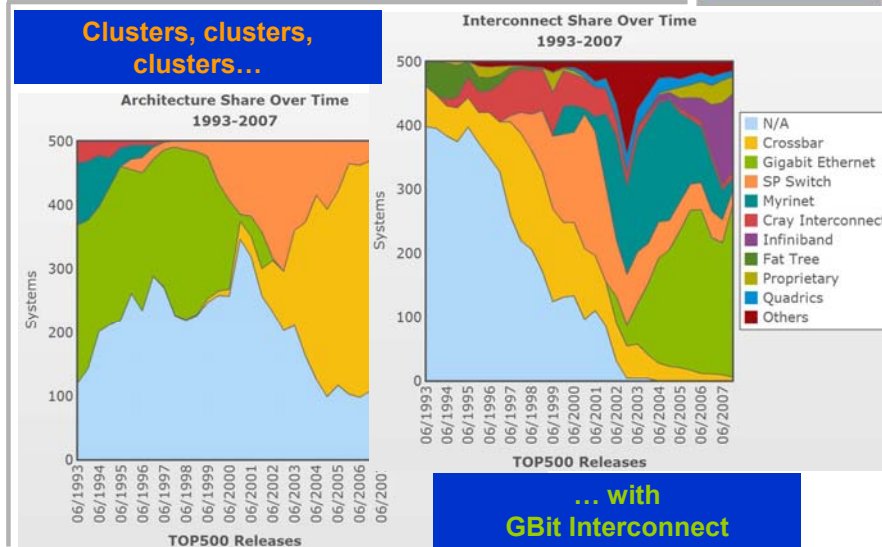
LINPACK [GFlop/s]

Parallelrechner – Vorlesung im SS2008

(8)



**Last but not least**  
*Top500 list as of November 2007*

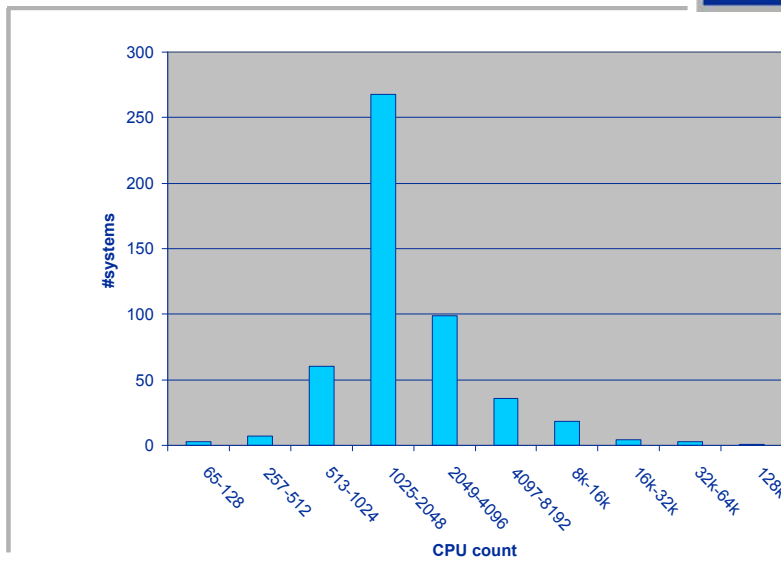


Parallelrechner – Vorlesung im SS2008

(9)



**Last but not least**  
*Top500 is going massive parallel (Nov. 2007)*



Parallelrechner – Vorlesung im SS2008

(10)



**Introduction**  
IBM BlueGene/L

Top1 (Nov. 2007)  
**212992 CPUs**  
596 TFlop/s Peak

**System**  
64 cabinets  
65,536 nodes  
(131,072 CPUs)  
(32x32x64)  
180/360 TF/s  
32 TiB\*  
1.2 MW  
2,500 sq.ft.  
MTBF 6.16 Days

**Node (2 CPUs):**  
2 x 2.8 GFlop/s (Peak)  
2 x 256 MByte main memory  
180 MByte/s in each dir. (Interconnect)

Parallelrechner – Vorlesung im SS2008 (11)

**Introduction**  
CRAY XT3

- 3rd generation of CRAY MPP systems (1 node = 1 CPU or DC chip)
- Successor of CRAY T3E
- System is designed to scale to 10.000s CPUs
- SANDIA: 127 TFlop/s
- NERSC: 100 TFlop/s in '07
- Original development: 40 TFlop/s Red Storm (Sandia)
- OS: Linux micro kernel

**Cray XT3 Scalable Architecture**  
By courtesy of W. Oed, CRAY

Parallelrechner – Vorlesung im SS2008 (12)

**Introduction**  
*HPC Centers in Germany: A view from Erlangen*

**Jülich Supercomputing Center**  
 8,9 TFlop/s IBM Power4+  
 46 TFlop/s Blue Gene

**HLR Stuttgart: 12 TFlop/s NEC SX8**

**LRZ-München**  
 2006: SGI Altix (26 TFlop/s)  
 2007: Upgrade to 62.3TFlop/s

Parallelrechner – Vorlesung im SS2008 (13)

**CX HPC**

**Introduction**  
*HLRB II@LRZ Munich SGI Altix 4700 / 9728 cores*

by courtesy of LRZ

11 m

22 m

Parallelrechner – Vorlesung im SS2008 (14)

**CX HPC**



**Introduction**  
HLRB-II: 2D-Torus between 19 „compute nodes“

**Inter-Partition NUMalink configuration** 51.2 GByte/s per direction

by courtesy of LRZ

Each grey box:

- 512 core partition (SSI)
- L login, B batch

Each line represents:

- 2 NUMalink4 planes with 16 cables (total)
- each cable: 2 \* 3.2 GB/s
- high density partitions share cables in each line

**Each compute node:**

- 512 processors**
- 2000 GByte main memory**

**Introduction**  
System configuration: HLRB-II

by courtesy of LRZ

**Phase 1**

- 4096 Intel Madison9M cores
  - single core = 4096 cores
  - 6 MByte L3
  - FSB 533: 1.33 Byte/Flop
- 17 TByte memory
- 26.2 TF Peak
- 24.5 TF Linpack
- 8.2 TF aggreg. weighted performance on LRZ BM
- 256 core single system images
- ccNUMA
- Power: ~ 1 MW

**Phase 2**

- 9728 Intel Montecito Cores
  - Dual Core
  - 9 Mbyte L3 per core
  - FSB 533
- 39 TByte Memory
- 62.2 TF Peak
- 53.3 TF Linpack
- 16 TF aggreg. weighted performance on LRZ BM
- 512 core single system image
- ccNUMA
- Power: ~1.1 MW

**NUMalink4 Interconnect (2D Torus based on 256/512 core fat trees)**  
globally addressable with low latency  
**High bandwidth/Low Latency (2x6.4 GByte/s link bandwidth, 1-6µs latency)**  
2 x 204 GByte/s between nodes

Parallelrechner – Vorlesung im SS2008 (16)



## Introduction

RRZE "Woody-Cluster"



- **860 Intel Xeon5160 processor cores**
    - Core2Duo architecture
    - 3.0 GHz → 12 GFlop/s per core
    - 4 cores per compute node
    - Installation: November 2006
  - **Peak performance: 10400 GFlop/s**
  - **Main memory:**
    - 2 GByte per core
    - 1720 GByte in total
  - **Infiniband network**
    - Voltaire DDRx 216 ports
    - 10 GBit/s+ per node & direction
  - **OS: SuSe Linux: SLES9**
- 
- **Parallel filesystem:** 15 TByte
  - **NFS filesystem:** 15 TByte



**Power consumption > 100 kW**

Parallelrechner – Vorlesung im SS2008

(17)



## Introduction

RRZE: Other Compute Resources (2003-2005)



### transtec compute cluster

- 216 2-way compute nodes:
  - 86 nodes: Intel Xeon 2.6 GHz; FSB533
  - 64 nodes: Intel Xeon 3.2 GHz; FSB800
  - **66 nodes: Intel Xeon 2.66 GHz; Dual-Core**
- 25 4-way compute nodes:
  - AMD Opteron270 (2.0 GHz); **Dual-Core**
- GBit Ethernet network
- Infiniband: 24 nodes (SDR) + 66 nodes (DDR)
- 5,5+13 TByte Disc Space
- Installation: 4/ 2003 ; Upgrades: 12 / 2004, Q4/2005, Q3/2007



### SGI Altix3700

- 32 Itanium2 1.3 GHz
- 128 GByte Memory
- 3 TByte Disc Space
- Inst.: 11 / 2003



### Compute Servers

### SGI Altix330

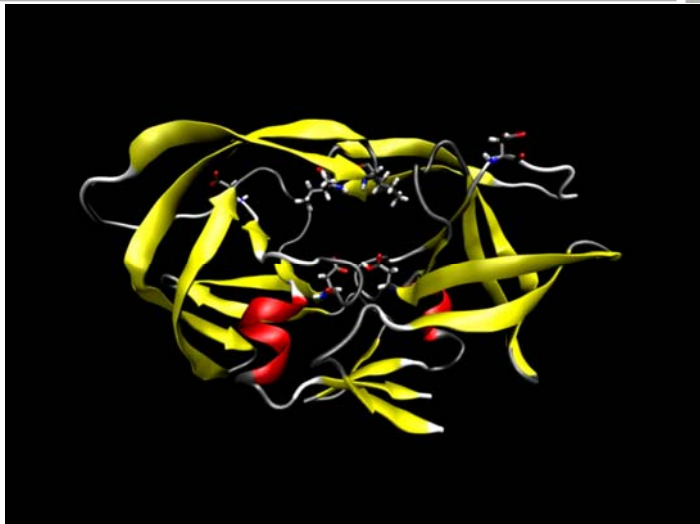
- 16 Itanium2 1.5 GHz
- 32 GByte Memory
- Inst.: 3 / 2006

Parallelrechner – Vorlesung im SS2008

(18)

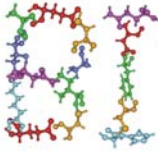


**Introduction**  
*MD Simulation of HIV protease dynamics*




**Real time:**  
10 ns

**Compute time:**  
18.000 CPU-hrs  
8 CPUs – 90 days

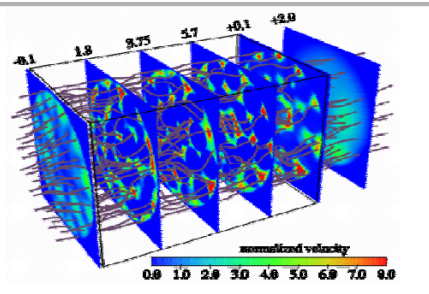


Courtesy: Prof. Sticht, Bio-Informatics, Emil-Fischer Center, FAU


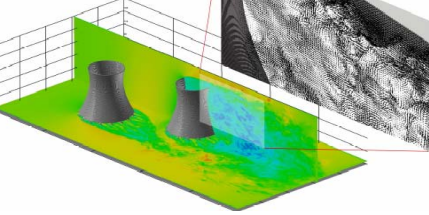
Parallelrechner – Vorlesung im SS2008 (19)



**Introduction**  
*Lattice Boltzmann flow solvers*




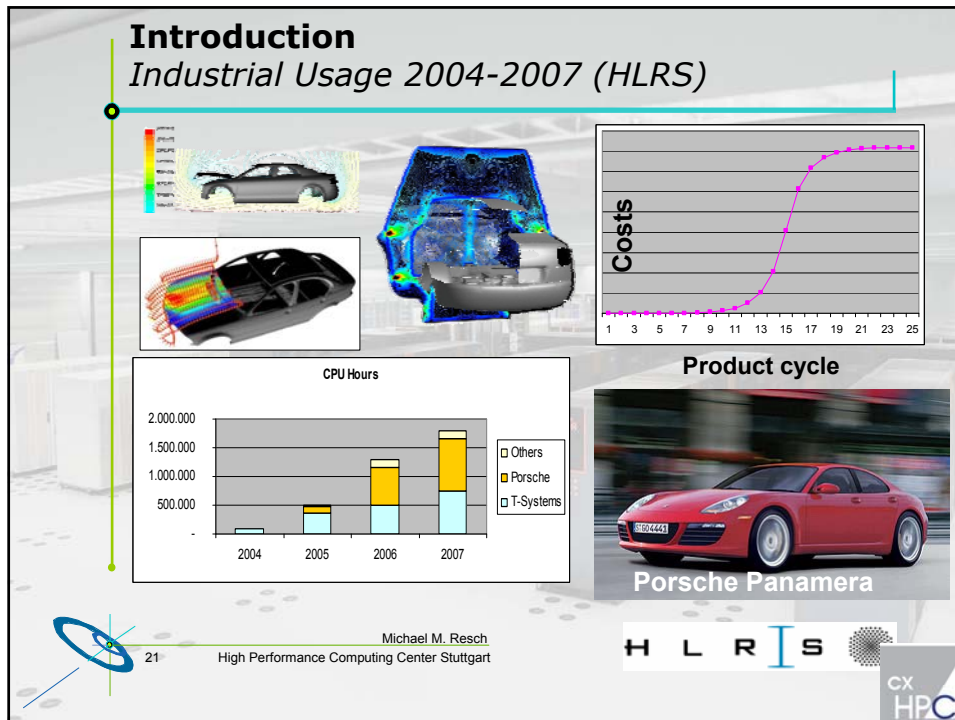
normalized velocity  
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0

Figures by courtesy of LS CAB-Braunschweig, Thomas Zeiser, N. Thürey

Parallelrechner – Vorlesung im SS2008 (20)





## Introduction

### How to build faster computers

- Increase performance / throughput of CPU**
  - Reduce cycle time, i.e. increase clock speed
  - Increase throughput, i.e. superscalar
- Improve data access time**
  - Increase cache size
  - Improve main memory access (bandwidth & latency)
- Introduce multi-(core) processing**
  - Requires shared-memory parallel programming
  - Shared/separate caches
  - Possible memory access bottlenecks
- Use external parallelism**  
**"Cluster" of computers tightly connected**
  - Almost unlimited scaling of memory and performance
  - Distributed-memory parallel programming

Parallelrechner – Vorlesung im SS2008 (22)

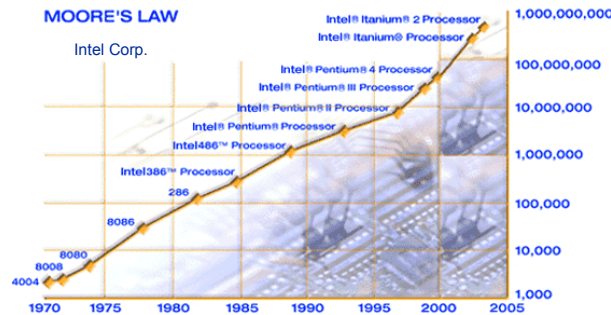
CX HPC

## Introduction

*Faster computers: Complexity and clock speed*



- 1965 G. Moore claimed  
#transistors on processor chip doubles every 24 months



This trend is currently changing:  
see multi-core

- Processor speed grew roughly at the same rate  
My computer: 350 MHz (1998) – 3,000 MHz (2004)  
Growth rate: 43 % p.a. → doubles every 24 months
- Problem: Power dissipation (see RRZE systems...)**

Parallelrechner – Vorlesung im SS2008

(23)



## Introduction

*Faster computers: Clock speed & Superscalarity*



- High clock speeds require pipelining of functional units:  
E.g. it may take 30 cycles to perform a single instruction on a Intel P4
  - Superscalarity – multiple functional units work in parallel:  
E.g. most processors can perform
    - 4-6 instructions per cycle
    - 2-4 floating point operations per cycle
- ↓
- High complexity of computer architectures
    - CISC → RISC
    - Out-of Order Execution
    - Introduction of new architectures:
      - EPIC/Itanium with fully in-order instruction issue
- 
- Manual optimization of code is mandatory
  - Memory bandwidth imposes restrictions for most applications

Parallelrechner – Vorlesung im SS2008

(24)



## Introduction

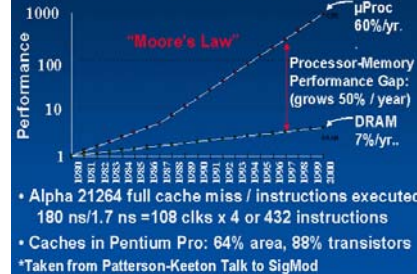
*Faster computers: Clock speed vs. DRAM gap*



### Memory (DRAM) Gap

- **Memory bandwidth grows only at a speed of 7% a year**
- Memory latency remains constant / increases in terms of processor speed
- Loading a single data item from main memory can cost 100s of cycles on a 3 GHz CPU
- Introducing memory hierarchies (caches) – Complex optimization of code

### Processor Limit: DRAM Gap



**Optimization of main memory access is mandatory for most applications**

Parallelrechner – Vorlesung im SS2008

(25)



**Microprocessors & Pipelining**

## Architecture of modern microprocessors

### History



- In the beginning: **Complex Instruction Set Computers (CISC)** :
  - Powerful & complex instructions, e.g:  $A=B*C$ : 1 instruction
  - Instruction set is close to high-level programming language
  - Variable length of instructions - Save storage!
- Mid 80's: **Reduced Instruction Set Computer (RISC)** evolved:
  - Fixed instruction length; enables pipelining and high clock frequencies
  - Uses simple instructions, e.g.:  $A=B*C$  is split into at least 4 operations (LD B, LD C, MULT  $A=B*C$ , ST A)
  - **Nowadays: Superscalar RISC processors**
  - IA32 (Core2, Athlon, Opteron): Compiler still generates CISC instructions; but processor core: RISC like
  - RISC is still implemented in most dual- / quad-core CPUs
- ~2001: **Explicitly Parallel Instruction Computing (EPIC)** introduced
  - Compiler builds large group of instruction to be executed in parallel
  - First processors: **Intel Itanium1/2 using the IA64** instruction set.

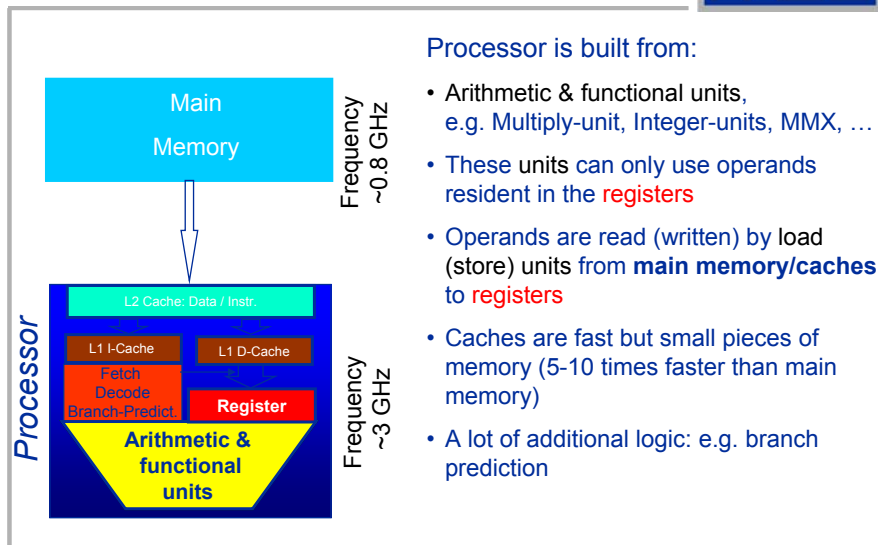
Parallelrechner – Vorlesung im SS2008

(27)



## Architecture of modern microprocessors

### Cache based microprocessors (e.g. Intel P4, AMD)



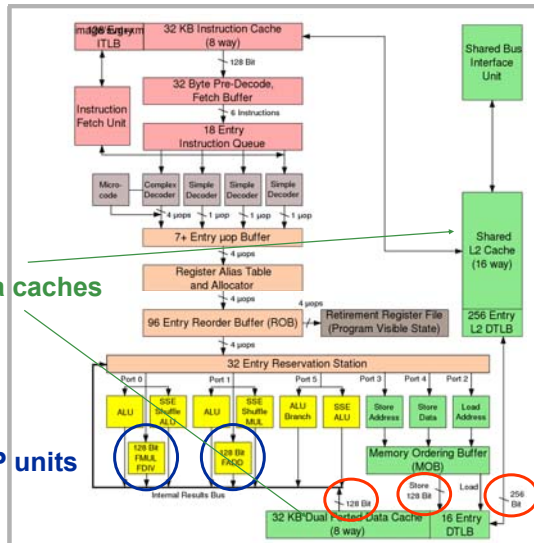
Parallelrechner – Vorlesung im SS2008

(28)





### Architecture of modern microprocessors Architectural block diagram – Intel Core 2



Data caches

FP units

#### Intel Core 2 architecture

- Successor of Netburst
- Reduced pipeline length
- Improved instruction issue
- Double FP performance
- Instruction level parallelism: 4 μops issue/cycle
- Desktop variants (65 nm)
  - Core2Duo → „Conroe“
  - Core2Quad → „Kentsfield“
  - Top bin: 3 GHz



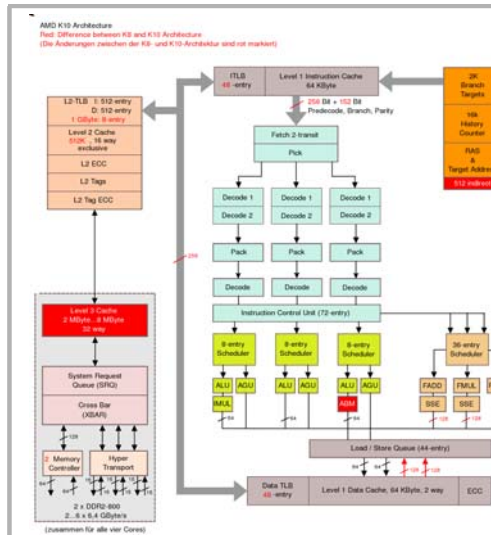
Intel Core 2 Architecture  
(cf. Wikipedia)

Parallelrechner – Vorlesung im SS2008

(29)



### Architecture of modern microprocessors Architectural block diagram – AMD K10



(cf. Wikipedia)

#### AMD K10 architecture

- To be used in first native Quad-Core („Barcelona“)
- L3 cache shared by all 4 cores
- There are only a few „Barcelona“ systems in field... (one at RRZE)
- Top bin: 2.0 GHz



Parallelrechner – Vorlesung im SS2008

(30)





## Architecture of modern microprocessors

### Pipelining of arithmetic/functional units



- **Split complex operations (e.g. multiplication) into several simple / fast sub-operations (stages)**
- **Makes short cycle time possible (simpler logic circuits), e.g.:**
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultan.
  - one result at each cycle after the pipeline is full
- **Drawback:**
  - Pipeline must be filled - startup times (#Operations >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order
- **Vector processors use large numbers of parallel pipelines!**

Parallelrechner – Vorlesung im SS2008

(31)



## Pipelining

### 5-stage Multiplication-Pipeline: $A(i)=B(i)*C(i)$ ; $i=1,\dots,N$



	Cycle: 1	2	3	4	5	6	...	N+4
<b>Operation</b>								
Separate Mant. / Exp.	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	B(6) C(6)	...	
Mult. Mantissa		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	
Add. Exponents			B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	
Normal. Result				A(1)	A(2)	B(3) C(3)	...	
Insert Sign					A(1)	A(2)	...	A(N)

First result is available after 5 cycles (=latency of pipeline)!

Parallelrechner – Vorlesung im SS2008

(32)



## Pipelining

### Speed-Up and Throughput



- In general (m-stage pipe /pipeline depth: m)

#### Speed-Up:

$$T_{\text{seq}} / T_{\text{pipe}} = (m \cdot N) / (N + m - 1) \sim m \text{ for large } N \ (\gg m)$$

#### Throughput (=Results per Cycle):

$$N / T_{\text{pipe}}(N) = N / (N + m - 1) = 1 / [1 + (m - 1) / N] \sim 1 \text{ for large } N$$

- Number of independent operations ( $N_C$ ) required to achieve  $T_p$  results per cycle:

$$T_p = 1 / [1 + (m - 1) / N_C] \quad \longrightarrow \quad N_C = T_p (m - 1) / (1 - T_p)$$

$$T_p = 0.5 \quad \longrightarrow \quad N_C = m - 1$$

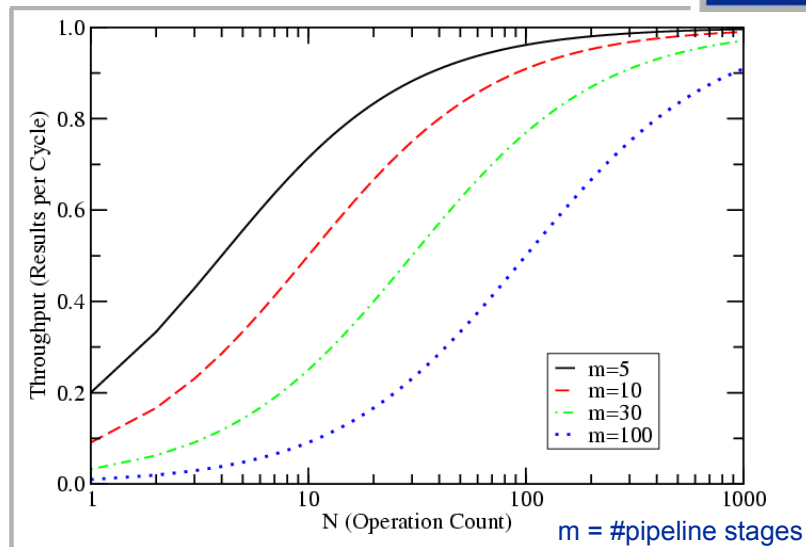
Parallelrechner – Vorlesung im SS2008

(33)



## Pipelining

### Throughput as function of pipeline stages



Parallelrechner – Vorlesung im SS2008

(34)



### Pipelining

#### Software pipelining


**Example:**

```

Fortran Code:
do i=1,N
  a(i) = a(i) * c
end do
        
```

**Assumption:**

Instructions block execution if operands are not available



load a[i]      Load operand to register (4 cycles) ← Latencies

mult a[i] = c, a[i]      Multiply a(i) with c (2 cycles); a[i], c in registers

store a[i]      Write back result from register to mem./cache (2 cycles)

branch.loop      Increase loopcounter as long i less equal N (0 cycles)

**Simple Pseudo Code:**

```

loop:  load a[i]
      mult a[i] = c, a[i]
      store a[i]
      branch.loop
        
```


**Optimized Pseudo Code:**

```

loop:  load a[i+6]
      mult a[i+2] = c, a[i+2]
      store a[i]
      branch.loop
        
```

Parallelrechner – Vorlesung im SS2008

(35)



### Pipelining

#### Software pipelining

$a[i]=a[i]*c; N=12$


Naive instruction issue	Optimized instruction issue	
Cycle 1	load a[1]	} Prolog
Cycle 2	load a[2]	
Cycle 3	load a[3]	
Cycle 4	load a[4]	
Cycle 5	load a[5]    mult a[1]=c, a[1]	} Kernel
Cycle 6	load a[6]    mult a[2]=c, a[2]	
Cycle 7	load a[7]    mult a[3]=c, a[3]	
Cycle 8	load a[8]    mult a[4]=c, a[4]	
Cycle 9	load a[9]    mult a[5]=c, a[5]	
Cycle 10	load a[10]    mult a[6]=c, a[6]	
Cycle 11	load a[11]    mult a[7]=c, a[7]	
Cycle 12	load a[12]    mult a[8]=c, a[8]	
Cycle 13	mult a[2]=c, a[2]	} Epilog
Cycle 14	mult a[9]=c, a[9]	
Cycle 15	mult a[10]=c, a[10]	
Cycle 16	mult a[11]=c, a[11]	
Cycle 17	mult a[12]=c, a[12]	
Cycle 18	store a[1]	
Cycle 19	store a[2]	
	store a[3]	
	store a[4]	
	store a[5]	
	store a[6]	
	store a[7]	
	store a[8]	
	store a[9]	
	store a[10]	
	store a[11]	
	store a[12]	

T= 96 cycles

T= 19 cycles

Parallelrechner – Vorlesung im SS2008

(36)



## Pipelining

### Efficient use



- Software pipelining can be done by the compiler, but efficient reordering of the instructions requires deep insight into application (data dependencies) and processor (latencies of functional units)
- (Potential) dependencies within loop body may prevent efficient software pipelining, e.g.:

#### No dependency:

```
do i=1,N
  a(i) = a(i) * c
end do
```

#### Dependency:

```
do i=2,N
  a(i) = a(i-1) * c
end do
```

#### Pseudo-Dependency:

```
do i=1,N-1
  a(i) = a(i+1) * c
end do
```

#### General version (offset as input parameter):

```
do i=max(1-offset,1),min(N-offset,N)
  a(i) = a(i-offset) * c
end do
```

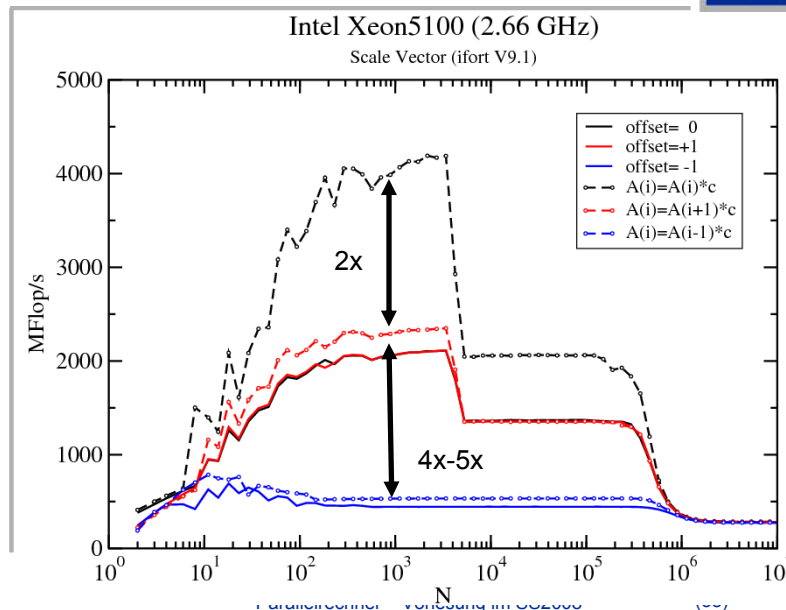
Parallelrechner – Vorlesung im SS2008

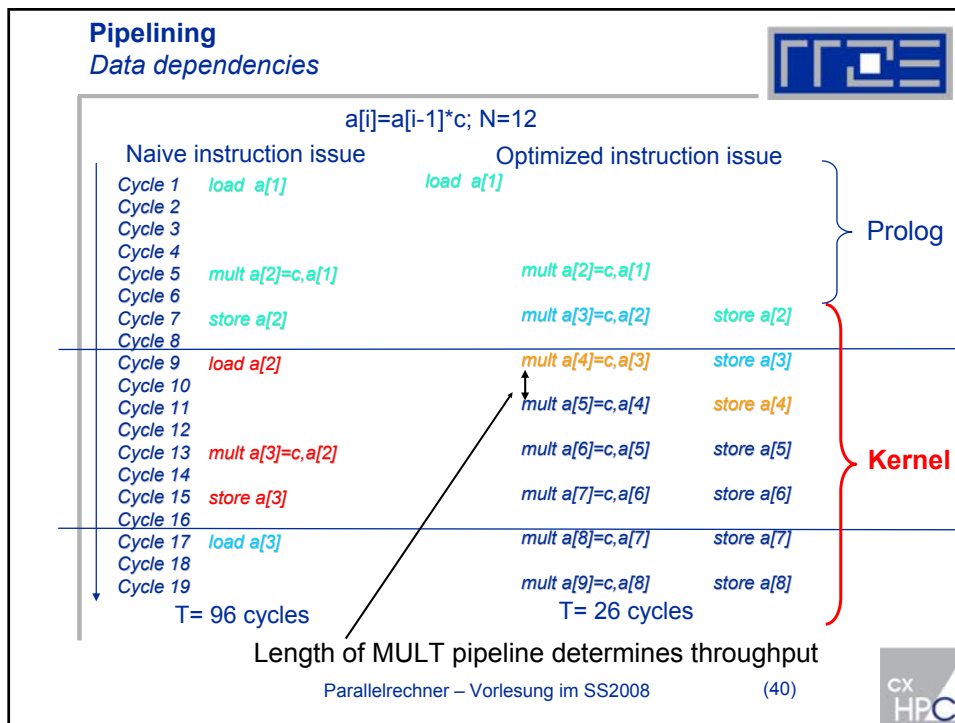
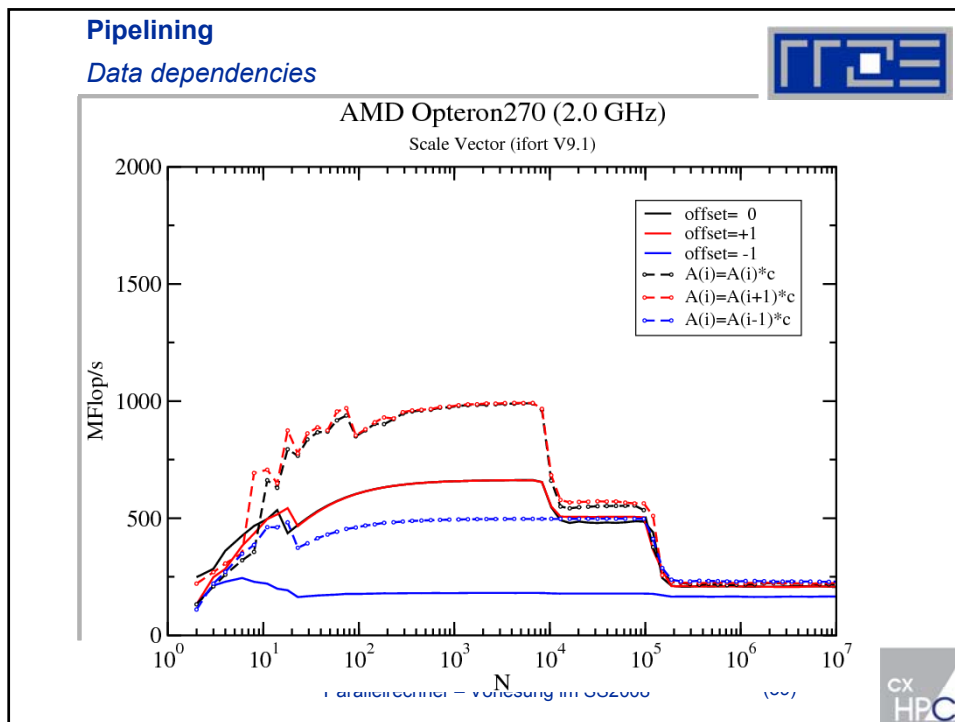
(37)



## Pipelining

### Data dependencies





## Pipelining

### Further potential problems



- Typical number of pipeline stages: 2-5 for the hardware pipelines on modern CPUs (e.g. Intel Core architecture: 5 cycles for FP MULT)
- Modern microprocessors do not provide pipelines for *div* / *sqrt* or *exp* / *sin* !  
Example: Cycles per Floating Operation (8-Byte) for Xeon/Netburst

Operation	y=a+y (y=a*y)	y=a/y	y=dsqrt(y)	y=sin(y)
Latency	4*	70*	70*	~160-180
Throughput	2*	70*	70*	130
Cycles/Operation	1*	35*	35*	130

- Reduce number of complex operations if necessary.
- Replace function call with a table lookup if the function is frequently computed for a few different arguments only.

\* Using SIMD instructions (SSE2)

Parallelrechner – Vorlesung im SS2008

(41)



## Pipelining

### Further potential problems



- Data dependencies: Compiler can not resolve aliasing conflicts!

```
void subscale( A , B )
...
for (i=0;...) A(i) = B(i-1)*c
```

In C/ C++ the pointers of A and B may point to the same memory location  
→ see above

- Tell compiler if your are never using aliasing ( -fno-alias for Intel Compiler)

- Subroutine/function calls within a loop

```
do i=1, N
  call elementprod(A(i), B(i), partsum)
  sum=sum+partsum
enddo
...
function elementprod( a, b, sum)
...
sum=a*b
```

Inline short subroutine/functions!

Parallelrechner – Vorlesung im SS2008

(42)



## Pipelining

### Instruction pipeline

- Besides the arithmetic and functional unit, the instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:

Fetch Instruction from L1I → Decode instruction → Execute Instruction

- Hardware Pipelining on processor (all units can run concurrently):

1	Fetch Instruction 1 from L1I		
2	Fetch Instruction 2 from L1I	Decode Instruction 1	
3	Fetch Instruction 3 from L1I	Decode Instruction 2	Execute Instruction 1
4	Fetch Instruction 4 from L1I	Decode Instruction 3	Execute Instruction 2

...

- Branches can stall this pipeline! (Speculative Execution, Predication)
- Each Unit is pipelined itself (cf. Execute=Multiply Pipeline)

Parallelrechner – Vorlesung im SS2008 (43)

## Pipelining

### Instruction pipeline

- Problem: Unpredictable branches to other instructions

Assume: Result determines next instruction!

1	Fetch Instruction 1 from L1I		
2		Decode Instruction 1	
3			Execute Instruction 1
4	Fetch Instruction 2 from L1I		
5		Decode Instruction 2	
6			Execute Instruction 2
7	Fetch Instruction 3 from L1I		
8		Decode Instruction 3	

Parallelrechner – Vorlesung im SS2008 (44)



## Pipelining

### Superscalar Processors



- Superscalar Processors can run multiple Instruction Pipelines at the same time!
- Parallel hardware components / pipelines are available to
  - fetch / decode / issues multiple instructions per cycle (typically 3 – 6 per cycle)
  - load (store) multiple operands (results) from (to) cache per cycle (typically 2-4 8-byte words per cycle)
  - perform multiple integer / address calculations per cycle (e.g. 6 integer units on Itanium2)
  - perform multiple floating point operations per cycle (typically 2 or 4 floating point operations per cycle)
- On superscalar RISC processors out-of order execution hardware is available to optimize the usage of the parallel hardware

Parallelrechner – Vorlesung im SS2008

(45)

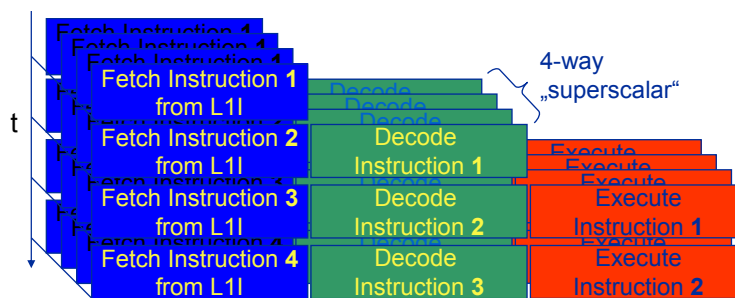


## Pipelining

### Superscalar Processors



- ❑ Multiple units enable use of **I**nstruction **L**evel **P**arallelism (ILP):



- ❑ Issuing  $m$  concurrent instructions per cycle:  $m$ -way superscalar
- ❑ Modern processors are 3- to 6-way superscalar & can perform 2 or 4 floating point operations per cycles

Parallelrechner – Vorlesung im SS2008

(46)



## Pipelining

### Superscalar Processor



- Example: Calculate norm of a vector on a CPU with 2 MultAdd (MADD) units

- Naive version:

t=0

do i=1, n

t=t+a(i)\*a(i)

end do

2 FP Mult/Add units cannot be busy at the same time because of dependency in summation variable t



- 2<sup>nd</sup> MADD has to wait for the first to be completed, although in principle two independent MADD could be done

Parallelrechner – Vorlesung im SS2008

(47)



## Pipelining

### Superscalar Processor



- Optimized version:

t1=0

t2=0

do I=1, N, 2

t1=t1+a(i)\*a(i)

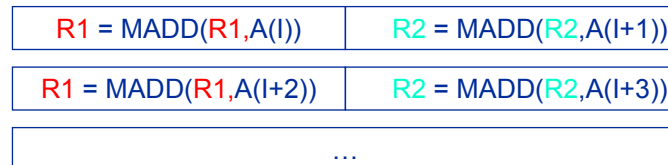
t2=t2+a(i+1)\*a(i+1)

end do

t=t1+t2

Most compilers can do those optimizations automatically (if you allow them to do so)!

Two independent „instruction streams“ can be processed by two separate FP Mult/Add units!



Parallelrechner – Vorlesung im SS2008

(48)



## Pipelining

### Superscalar PCs



	Intel P4/Netburst	Intel Core
FP units	1 MULT & 1 ADD pipeline	
Width of operands	128 Bit	
FP ops/unit	2 DP or 4 SP	
Throughput	2 cycles	1 cycle
Max. FP ops/cycle	2 DP or 4 SP	4 DP or 8 SP
Latency of FP units (FPMULTD)	7 cycles	5 cycles

DP: double precision, i.e. 64 bit operands (double)

SP: single precision, i.e. 32 bit operands (float)

Throughput: Repeat rate of instruction issue,  
e.g. 1 cycle -> in each cycle a new operation can be started

Parallelrechner – Vorlesung im SS2008

(49)

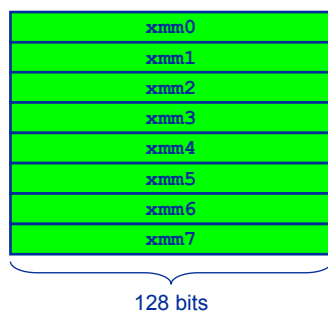


## Pipelining

### Superscalar PCs – SSE



- **Streaming SIMD Extensions (SSE) instructions must be used to operate on the 128 bit registers**
  - **Register Model:**



- Each register can be partitioned into several integer or FP data types
  - 8 to 128-bit integers
  - single (SSE) or double precision (SSE2) floating point
- SIMD instructions can operate on the lowest or all partitions („Packed SSE“) of a register **at once**

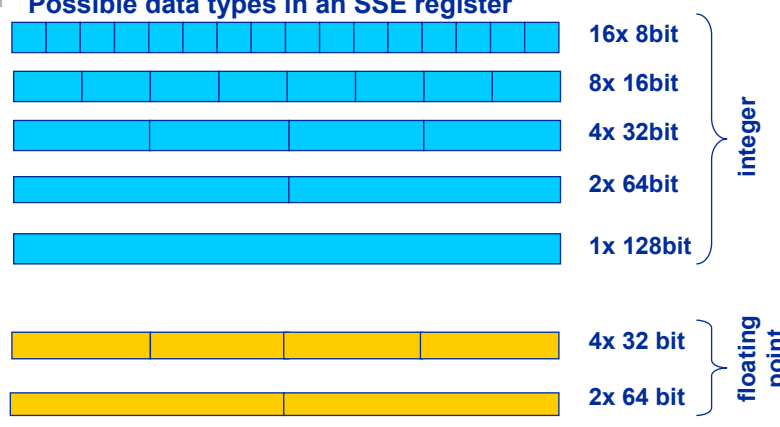
Parallelrechner – Vorlesung im SS2008

(50)



### Pipelining

#### Superscalar PCs – SSE



Possible data types in an SSE register

- 16x 8bit
- 8x 16bit
- 4x 32bit
- 2x 64bit
- 1x 128bit

integer

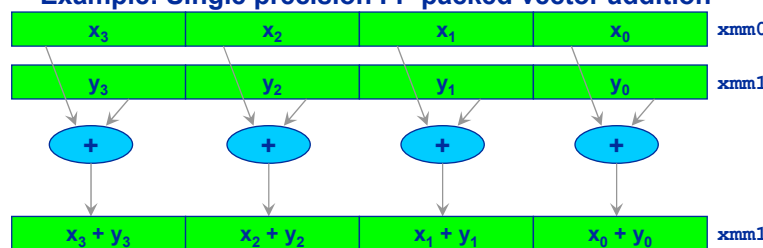
- 4x 32 bit
- 2x 64 bit

floating point

Parallelrechner – Vorlesung im SS2008 (51) CX HPC

### Pipelining

#### Superscalar PCs – SSE



Example: Single precision FP packed vector addition

Four single precision FP additions with one single instruction

- Packed SSE → Code vectorization is a must
- Vectorization only possible if data are independent
- Automatic vectorization by compiler (appropriate compiler flag needs to be set) or forced by programmer (via directive)
  - “LOOP WAS VECTORIZED” messages ✓

Parallelrechner – Vorlesung im SS2008 (52) CX HPC

## Pipelining

### Efficient Use of Pipelining



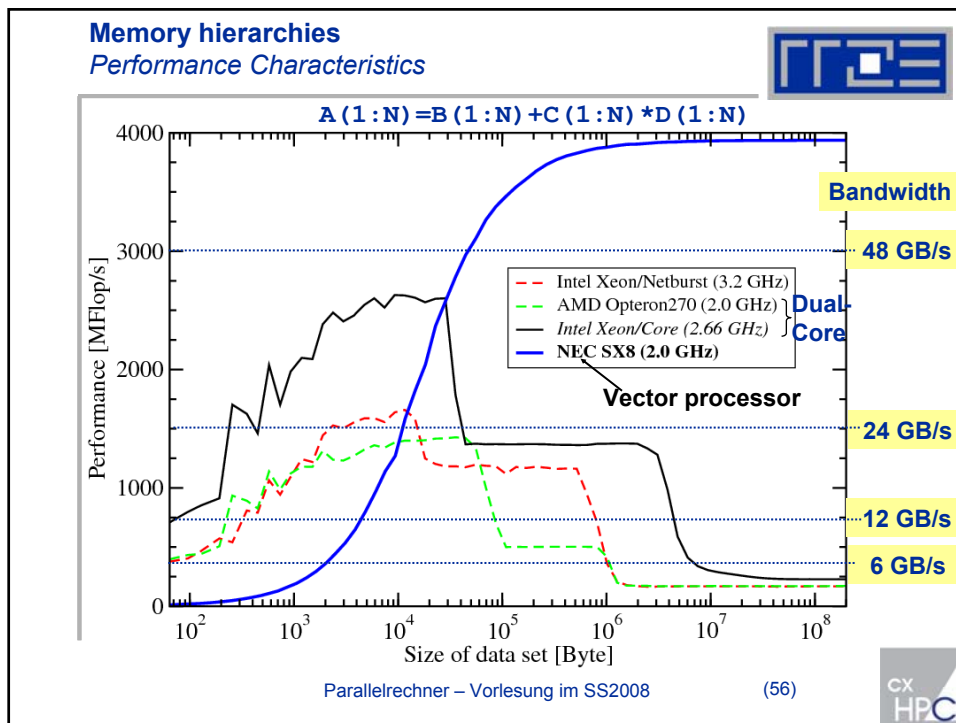
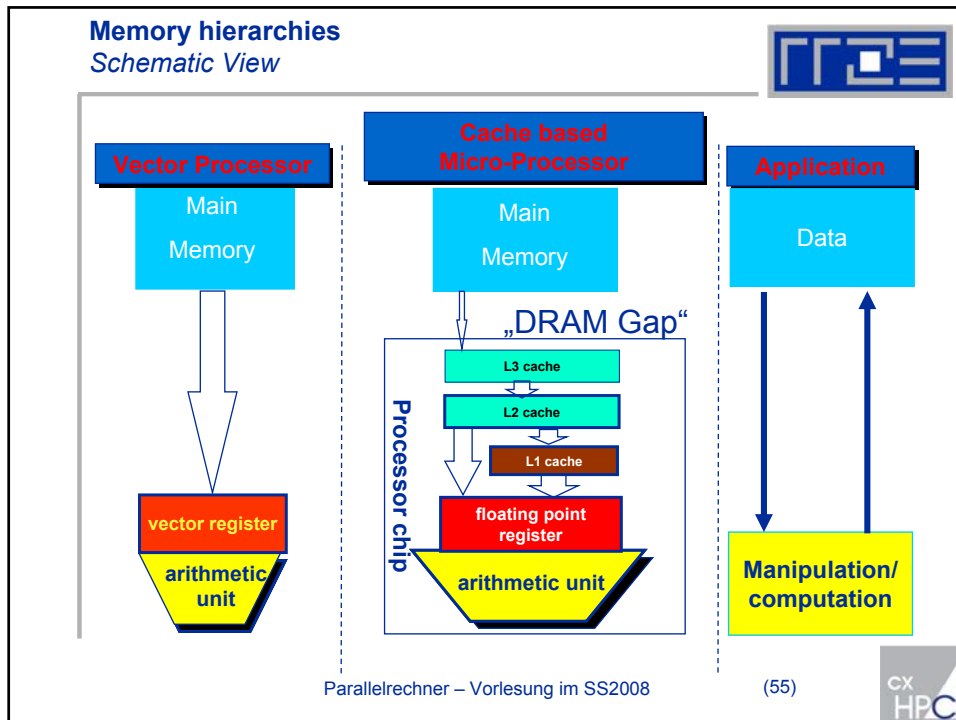
- **Efficient use of pipelining/ILP requires intelligent compilers**
  - Rearrangement of instructions to hide latencies
  - „Software pipelining“
  - Remove interdependencies that block parallel execution
- **Programmer should**
  - Avoid unpredictable branches (stop & restart of instruction pipeline!)
  - Avoid Data dependencies (if possible)
  - Tell compiler that instructions are independent (e.g. do not use pointer aliasing: `-fno-alias` with intel compiler)
- **Long pipelines are inefficient for very small loops**
  - Pipeline must be filled, i.e. long start-up times (latency!)
- **Summary:**
  - Large number of independent / parallel instruction is mandatory to efficiently use pipelined, superscalar processors.
  - Most of the work can be done by the compiler, however programmer must provide reasonable code

Parallelrechner – Vorlesung im SS2008

(53)



## Memory hierarchies of modern processors



## Memory hierarchies

Cache-based architectures – “private cluster”



*SSE2		Intel Xeon/ Netburst	Intel Xeon / Core	AMD Opteron
<b>Peak Performance</b> Core frequency		7.2 GFlop/s 3.6 GHz	<b>12.0 GFlop/s</b> 3.0 GHz	<b>5.6 GFlop/s</b> 2.8 GHz
<b>#Registers</b>		16 / 32*	<b>16 / 32*</b>	<b>16 / 32*</b>
L1	<b>Size</b>	16 kB	32 kB	64 kB
	<b>BW</b>	115 GB/s	96 GB/s	45 GB/s
	<b>Latency</b>	12 cycles	3 cycles	3 cycles
L2	<b>Size</b>	2 MB	<b>4 MB (2cores)</b>	<b>1 MB</b>
	<b>BW</b>	115 GB/s	<b>96 GB/s</b>	<b>45 GB/s</b>
	<b>Latency</b>	20 cycles	<b>13 cycles</b>	<b>12 cycles</b>
Memory	<b>BW</b>	6.4 GB/s	<b>10.6 GB/s</b>	<b>10.6 GB/s</b>
	<b>Latency</b>	~200 ns	<b>~200 ns</b>	<b>&lt; 100 ns</b>

**dual-core CPUs**

Parallelrechner – Vorlesung im SS2008

(57)



## Memory hierarchies

Processor architectures – “supercomputing centers”



*dual - core		NEC SX8	IBM Power5*	Intel Itanium2*
<b>Peak Performance</b> Core frequency		<b>16 GFlop/s</b> 2.0 GHz	<b>7.2 GFlop/s</b> 1.9 GHz	<b>6.4 GFlop/s</b> 1.6 GHz
<b>#Registers</b>		<b>8*256</b>	<b>32</b>	<b>128</b>
L1	<b>Size</b>		32 kB	16 kB
	<b>BW</b>		72 GB/s	51.2 GB/s
	<b>Latency</b>		3 cycles	1 cycle
L2	<b>Size</b>		<b>1.9 MB (2 cores)</b>	256 kB
	<b>BW</b>		<b>72 GB/s</b>	51.2 GB/s
	<b>Latency</b>		<b>~13 cycles</b>	5-6 cycles
L3	<b>Size</b>		36 MB	<b>6 / 12 MB</b>
	<b>BW</b>		~10 GB/s	<b>51.2 GB/s</b>
	<b>Latency</b>		~80 cycles	<b>12-13 cycles</b>
Memory	<b>BW</b>	<b>64 GB/s</b>	~10 GB/s	8.5 GB/s
	<b>Latency</b>	<b>vectorization</b>	<b>100 ns</b>	~200 ns

Parallelrechner – Vorlesung im SS2008

(58)





## Memory hierarchies

### Characterization



Two quantities characterize the quality of each memory hierarchy:

- **Latency ( $T_{lat}$ ):** Time to set up the memory transfer from source (main memory or caches) to destination (registers).
- **Bandwidth (BW):** Maximum amount of data which can be transferred per second between source (main memory or caches) and destination (registers).

$$\text{Transfer time: } T = T_{lat} + (\text{amount of data}) / \text{BW}$$

- For microprocessor holds  $T \approx T_{lat}$   
(e.g.:  $T_{lat}=100$  ns;  $\text{BW}=4$  GByte/s; amount of data=8 byte  $\rightarrow T=102$  ns)
- Caches are organized in cache lines that are fetched/stored as a whole (e.g. 128 byte = 16 double words)

Parallelrechner – Vorlesung im SS2008

(59)

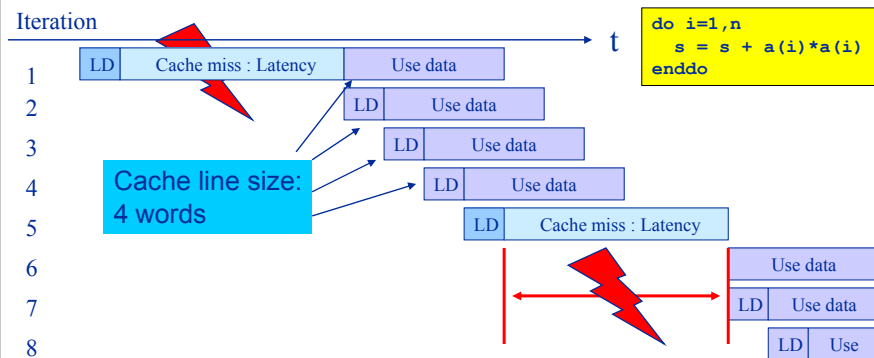


## Memory hierarchies

### Cache structure



- If one item is loaded from main memory (cache miss), the whole cache line it belongs is loaded to the caches
- Cache lines are contiguous in main memory, i.e. “neighboring” items can then be used from cache



$T_{lat}=100$  ns;  $\text{BW}=4$  GByte/s; amount of data=128 byte  $\rightarrow T=132$  ns

Parallelrechner – Vorlesung im SS2008

(60)



## Memory Hierarchies

### Cache Structure



- Cache line data is **always consecutive**
  - Cache use is optimal for contiguous access (stride 1)
  - Non-consecutive reduces performance
  - Access with wrong stride (e.g. with cache line size) can lead to disastrous performance breakdown
- Long cache lines reduces the latency problem for contiguous memory access. Otherwise: latency problem becomes worse.
- Calculations get cache bandwidth inside the cache line, but main memory latency still limits performance
- Cache lines must somehow be mapped to memory locations
  - **Cache multi-associativity enhances utilization**
  - **Try to avoid cache thrashing**

Parallelrechner – Vorlesung im SS2008

(61)



## Memory Hierarchies

### Cache Line Prefetch to hide latencies



- *Prefetch (PFT) instructions:*
  - Transfer of consecutive data (one cache line) from memory to cache
  - Followed by LD to registers
  - Useful for executing loops with **consecutive memory access**
  - Compiler has to ensure **correct placement** of PFT instructions
    - Knowledge about memory latencies required
    - Loop timing must be known to compiler
  - Due to large latencies, **outstanding pre-fetches** must be sustained

Parallelrechner – Vorlesung im SS2008

(62)



### Memory Hierarchies

Cache Line Prefetch to hide latencies

```
do i=1,n
  s = s + a(i)*a(i)
enddo
```

Prefetching allows to overlap of data transfer and calculation!

Hardware assisted prefetching for long contiguous data accesses

Iteration

1 PFT Cache miss : Latency LD Use data

2 LD Use data

3 LD Use data

4 LD Use data

5 PFT Cache miss : Latency LD Use data

6 Two outstanding prefetches LD Use data

7 LD Use data

8 LD Use data

9 PFT Cache miss : Latency LD Use data

Intel Itanium2/EPIC: Software pipelining using PFT operations

Parallelrechner – Vorlesung im SS2008 (63)

### Memory Hierarchies

Cache Line Prefetch to hide latencies

Hide memory latency on Itanium2 systems:

1. Latency approx. 140 ns
2. Time to transfer one cache-line: 128 Byte/6.4 GByte/s = 20 ns
3. Total time to transfer one cache line: 160 ns

Prefetching interferes with data dependencies, e.g. indirect addressing,...

Min. of 8 prefetches required to hide main memory latency!

Long loops: min. 8\*16

Parallelrechner – Vorlesung im SS2008 (64)

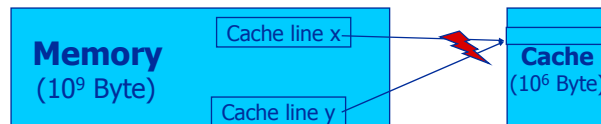
## Memory Hierarchies

### Cache Mapping



- **Cache Mapping**

- Pairing of memory locations with cache line
- e.g. mapping 1 GB of main memory to 1 MB of cache



- **Static Mapping**

- **Directly Mapped** caches vs. **m-way set associative** caches

- **Replacement strategies**

If all potential cache locations are full, one line has to be overwritten (“invalidated”) on next cache load using different strategies:

**Least Recently Used (LRU)** random vs. **Not Recently Used (NRU)**

May incur additional data transfer (→ cache thrashing) !

Parallelrechner – Vorlesung im SS2008

(65)



## Memory Hierarchies

### Cache Mapping – Directly mapped



- **Directly mapped cache:**

- Every memory location can only be mapped to exactly one cache location
- If cache size= $n$ ,  $i$ -th memory location can be stored at cache location  $\text{mod}(i,n)$
- Easy to implementation & fast lookup
- No penalty for stride-one access
- Memory access with stride=cache size will not allow caching of more than one line of data, i.e. **effective cache size** is one line!

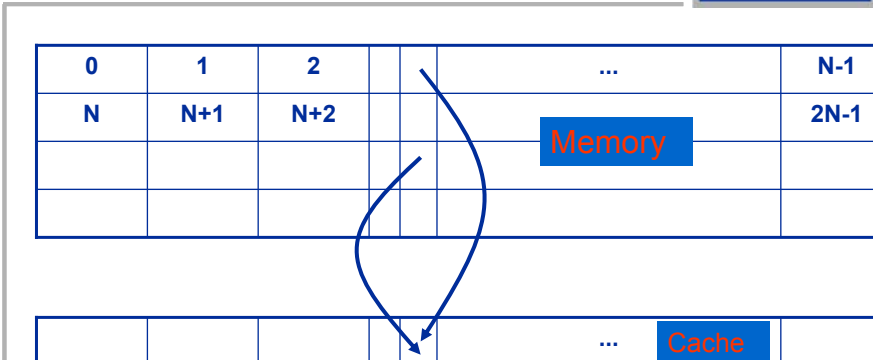
Parallelrechner – Vorlesung im SS2008

(66)




### Memory Hierarchies

#### Cache Mapping – Directly Mapped



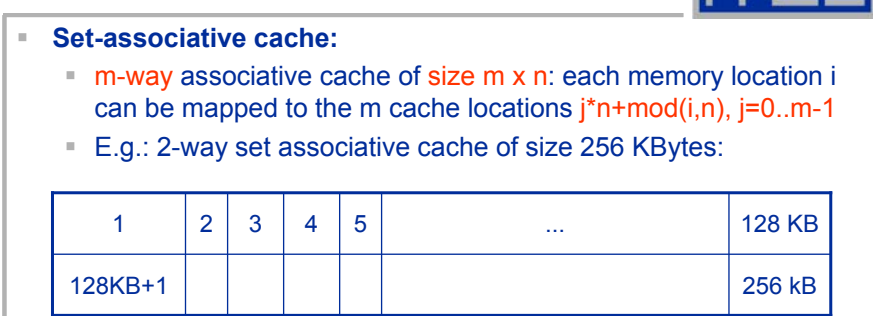
Example: Directly mapped cache. Each memory location can be mapped to one cache location only.

*E.g. Size of main memory= 1 GByte; Cache Size= 256 KB  
→ 4096 memory locations are mapped to the same cache location*

Parallelrechner – Vorlesung im SS2008 (67) 

### Memory Hierarchies


#### Cache Mapping – Associative Caches



- **Set-associative cache:**
  - **m-way** associative cache of **size m x n**: each memory location  $i$  can be mapped to the  $m$  cache locations  $j*n + \text{mod}(i,n)$ ,  $j=0..m-1$
  - E.g.: 2-way set associative cache of size 256 KBytes:

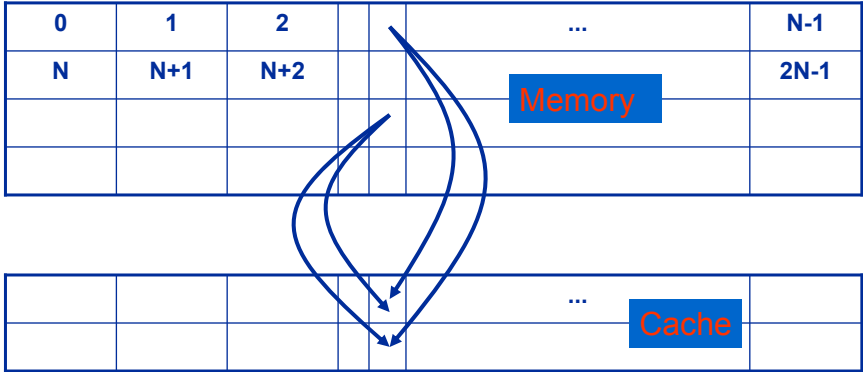
1	2	3	4	5	...	128 KB
128KB+1						256 KB

- **Ideal world:** Fully associative cache where every memory location is mapped to any cache line
  - Thrashing nearly impossible
  - The higher the associativity, the larger the overhead, e.g. latencies increase; cache complexity limits clock speed!


Parallelrechner – Vorlesung im SS2008 (68) 

### Memory hierarchies

#### Cache Mapping – Associative Caches




Example: 2-way associative cache. Each memory location can be mapped to two cache locations:  
*E.g. Size of main memory= 1 GByte; Cache Size= 256 KB → 8192 memory locations are mapped to **two** cache locations*

Parallelrechner – Vorlesung im SS2008 (69) 


### Memory hierarchies

#### Pitfalls & Problems


- If many memory locations are used that are mapped to the same  $m$  cache slots, cache reuse can be very limited even with  $m$ -way associative caches
 



**Warning: Using powers of 2 in the leading array dimensions of multi-dimensional arrays should be avoided! (Cache Thrashing)**
- If the cache /  $m$  associativity slots are full and new data comes in from main memory, data in cache (cache line) must be invalidated or written back to main memory
 

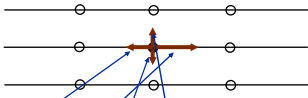


**Ensure spatial and temporal data locality for data access! (Blocking)**

Parallelrechner – Vorlesung im SS2008 (70) 

### Memory hierarchies

#### Cache thrashing - Example




Example: 2D – square lattice  
At each lattice point the 4 velocities for each of the 4 directions are stored

```

N=16
real*8 vel(1:N , 1:N, 4)
.....
s=0.d0
do j=1,N
  do i=1,N
    s=s+vel(i,j,1)-vel(i,j,2)+vel(i,j,3)-vel(i,j,4)
  enddo
enddo
    
```

Parallelrechner – Vorlesung im SS2008 (71)



### Memory hierarchies

#### Cache thrashing - Example

Memory to cache mapping for `vel(1:16, 1:16, 4)`  
Cache: 256 byte (=32 double) / 2-way associative / Cache line size=32 byte

1,1,1	2,1,1	3,1,1	4,1,1	....	1,1,2	2,1,2	3,1,2	4,1,2	....	1,1,3	2,1,3	3,1,3	4,1,3	....	1,1,4	2,1,4	3,1,4	4,1,4
-------	-------	-------	-------	------	-------	-------	-------	-------	------	-------	-------	-------	-------	------	-------	-------	-------	-------

$i=1, j=1$

1,1,1	2,1,1	3,1,1	4,1,1	....
1,1,2	2,1,2	3,1,2	4,1,2	....
1,1,3	2,1,3	3,1,3	4,1,3	....
1,1,4	2,1,4	3,1,4	4,1,4	....


`Vel(1:16,1:16,1)`  
`Vel(1:16,1:16,2)`  
`Vel(1:16,1:16,3)`  
`Vel(1:16,1:16,4)`

1,1,3	2,1,3	3,1,3	4,1,3
1,1,4	2,1,4	3,1,4	4,1,4

Cache:  
2 rows with 16 double each

Each cache line must be loaded 4 times from main memory to cache!

Parallelrechner – Vorlesung im SS2008 (72)





## Memory hierarchies

### Cache thrashing - Example



Memory to cache mapping for `vec1(1:18, 1:18, 4)`

Cache: 256 byte (=32 doubles) / 2-way associative / Cache line size=32 byte

1,1,1 2,1,1 3,1,1 4,1,1 .... 1,1,2 2,1,2 3,1,2 4,1,2 .... 1,1,3 2,1,3 3,1,3 4,1,3 .... 1,1,4 2,1,4 3,1,4 4,1,4

$i=1, j=1$

1,1,1	2,1,1	3,1,1	4,1,1	.....
17,1,1	18,1,1	1,1,2	2,1,2	3,1,2 4,1,2 .....
.....	.....	.....	1,1,3 2,1,3 3,1,3 4,1,3	.....
.....	.....	.....	.....	1,1,4 2,1,4 3,1,4 4,1,4

1,1,1	2,1,1	3,1,1	4,1,1	1,1,3	2,1,3	3,1,3	4,1,3	.....
.....	.....	.....	.....	1,1,2	2,1,2	3,1,2	4,1,2	1,1,4 2,1,4 3,1,4 4,1,4

Cache:

2 rows with 16 doubles each

Each cache line needs only be loaded **once** from memory to cache!

Parallelrechner – Vorlesung im SS2008

(73)



## Characterization of Memory Hierarchies:

Modeling,

measuring and

understanding

the performance limitations

## Characterization of Memory Hierarchies:

Kernel benchmark – Vector-Triad



- Kernel benchmarks:
  - Characterize computer architecture (effective performance of processor & memory hierarchies)
  - Results are easy to be interpreted and to be compared
  - Provide upper performance bounds for specific applications
  - E.g.: *stream*, *cachebench*

Vector-Triad

```
REAL*8 (SIZE) : A,B,C,D
DO ITER=1,NITER
  DO i=1,N
    A(i) = B(i) + C(i) * D(i)
  ENDDO
  <OBSCURE>
ENDDO
```

**Balance**

- Computation:  
**2 Flops / i-iteration**
- Bandwidth:  
**(3 LD & 1 ST) / i-iteration**
- *Balance = 2 Word / Flop*

Parallelrechner – Vorlesung im SS2008

(75)

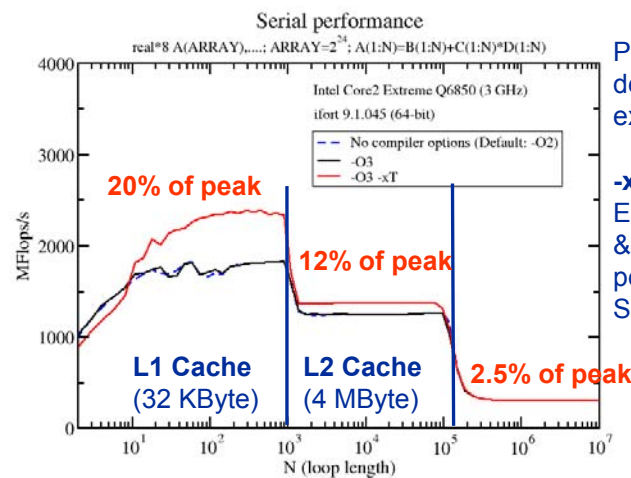


## Characterization of Memory Hierarchies:

Vector-Triad on Intel Core2 Q6850: 1 core results



Peak Performance for 1 core of Intel Core2 Q6850 (double arguments):  
 $3 \text{ GHz} * (2 \text{ Flops (DP-Add)} + 2 \text{ Flops (DP-Mult)}) = 12 \text{ GFlops/s}$



Performance decreases if data set exceeds cache size


**-xT** :  
 Enables vectorization & improves in-cache performance: Packed SSE instructions

Parallelrechner – Vorlesung im SS2008

(76)



### Characterization of Memory Hierarchies: Vector-Triad on Intel Core2 Q6850: 1 core results



$A(1:N) = B(1:N) + C(1:N) * D(1:N)$

- **Potential limiting factors:**
  - **FP units: 1 ADD + 1 MULTIPLY** 😊
  - **Data transfer: L1 cache – Registers**
    - Available per cycle (for consecutive data):  
128 Bit Load from L1 + 128 Bit Store to L1
    - Code requires per iteration (2 Flops):  
3 Loads (64 Bit) from L1 and 1 Store (64 Bit) to L1
    - Consider data transfer for four successive iterations:  
(assume no cost for FP computation)


```
LD C(i-2:i-1)
LD D(i-2:i-1)
LD B(i-2:i-1) ST A(i-4:i-3)

LD C(i :i+1)
LD D(i :i+1)
LD B(i :i+1) ST A(i-2:i-1)
```

3 cycles to transfer data for 2 iterations


Maximum performance for Vector-Triad:  
4 Flop/3 cycles

**4 GFlops/s at 3 GHz**




Parallelrechner – Vorlesung im SS2008

(77)



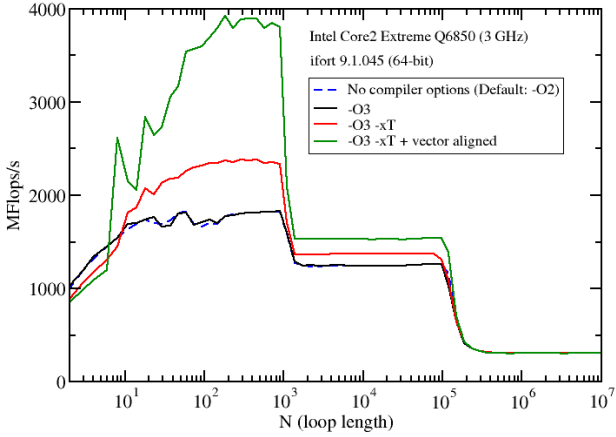
### Characterization of Memory Hierarchies: Vector-Triad on Intel Core2 Q6850: 1 core results



**Maximum performance of Vector-Triad in L1 cache: 4 GFlops/s**

Serial performance

real\*8 A(ARRAY),...; ARRAY=2<sup>24</sup>; A(1:N)=B(1:N)+C(1:N)\*D(1:N)



Intel Core2 Extreme Q6850 (3 GHz)  
ifort 9.1.045 (64-bit)

- No compiler options (Default: -O2)
- -O3
- -O3 -xT
- -O3 -xT + vector aligned

Put directive before the loop:

**!DEC\$ VECTOR ALIGNED**


↓

Compiler generates „perfect“ assembly code:  
Aligned SSE  
LD/ST instructions

However:  
Programmer has to guarantee that arrays are aligned to 16-Byte boundaries!!!

Parallelrechner – Vorlesung im SS2008

(78)



### Characterization of Memory Hierarchies:

Vector-Triad on Intel Core2 Q6850: 1 core results



$$A(1:N) = B(1:N) + C(1:N) * D(1:N)$$

#### Potential limiting factors:

##### Data transfer: L2 cache – L1 cache - Registers

- L2 – L1 bandwidth: 4 double words (256 Bits) per cycle
- Data transfer between L1 and L2 is on cache line basis (8 double words), i.e. 2 cycles are required to transfer 1 cache line
- L1 cache: write back cache (Core2 architectural design feature)  
If A(i) is not resident in L1 („L1 store miss“) it must first be loaded from L2 to L1 (complete cache line of A(i)): „Read for Ownership“ (RFO)
- Data transfers for a single iteration: L1 / L2
  - 1 L1→L2 transfer: A(i)
  - 4 L2→L1 transfers: A(i), B(i), C(i), D(i)

Parallelrechner – Vorlesung im SS2008

(79)



### Characterization of Memory Hierarchies:

Vector-Triad on Intel Core2 Q6850: 1 core results



$$A(1:N) = B(1:N) + C(1:N) * D(1:N)$$

#### Potential limiting factors:

##### Data transfer: L2 cache – L1 cache – Registers (continued)

- Assuming that L1 – Register and L2 – L1 transfer can not occur concurrently 10 more cycles are required for 8 iterations (1.25 cycles/iteration):
 

L2→L1: C(i:i+7)	[2 cycles]
L2→L1: D(i:i+7)	[2 cycles]
L2→L1: B(i:i+7)	[2 cycles]
L2→L1: A(i:i+7)	[2 cycles]
L1→L2: A(i:i+7)	[2 cycles]
- Total transfer time for 2 iterations:
 

L2→L1:	2.5 cycles
L1→Registers:	3.0 cycles (cf. L1-Register slide)

2 iterations/ 5.5 cycles → 4 Flops/5.5 cycles

→ Max. performance for data in L2 cache: 2.18 GFlops/s

Parallelrechner – Vorlesung im SS2008

(80)



### Characterization of Memory Hierarchies: Vector-Triad on Intel Core2 Q6850: 1 core results



$$A(1:N) = B(1:N) + C(1:N) * D(1:N)$$

- **Potential limiting factors:**
  - **Data transfer: Memory – L2 cache:**
  - **Memory access through FSB1333@64Bit, i.e. Memory-L2 bandwidth: 1.33 GHz \* 8 Byte = 10.67 GByte/s**
  - **L2 cache is write back cache -> RFO for A(i) from memory required**
  - **For a single iteration (4+1) double words = 40 Bytes have to be transferred → 20 Byte/Flop**

→ **Max. performance for data in main memory:**

$$(10.67 \text{ GByte/s}) / (20 \text{ Byte/Flop}) = 0.534 \text{ GFlops/s}$$

Parallelrechner – Vorlesung im SS2008

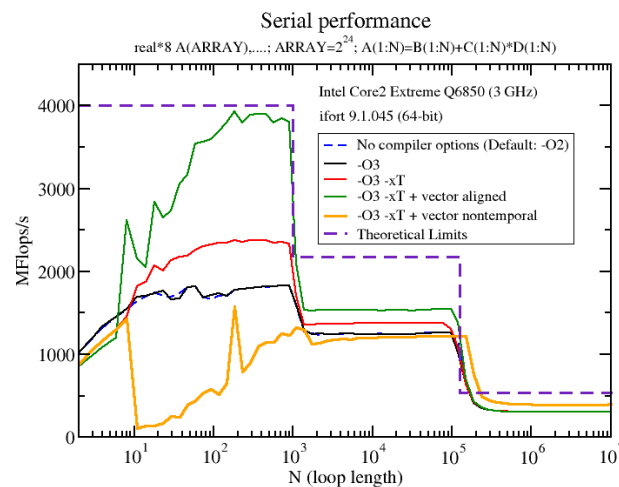
(81)



### Characterization of Memory Hierarchies: Vector-Triad on Intel Core2 Q6850: 1 core results



- **Insert !DEC\$ VECTOR NONTEMPORAL → A is directly written to main memory, BYPASSING the caches – No RFO load operation**



Parallelrechner – Vorlesung im SS2008

(82)





# Optimization of data access

Parallelrechner – Vorlesung im SS2008

(83)



## Data layout optimizations

### Basics



- Be aware of different memory mapping for FORTRAN & C:

`double a(4,4) // C version`

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

`real*8 a(0:3,0:3) ! FORTRAN Version`

0,0	1,0	2,0	3,0	0,1	1,1	2,1	3,1	0,2	1,2	2,2	3,2	0,3	1,3	2,3	3,3
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Ordering of nested loops!

- Stride one access in inner loops to exploit cache lines!

```
real*8 A(SIZE,SIZE)
real*8 B(SIZE,SIZE)
N=SIZE
do i=1,N
  do j=1,N
    A(j,i)=A(j,i)*B(j,i)
  enddo
enddo
```

```
real*8 A(SIZE,SIZE)
real*8 B(SIZE,SIZE)
N=SIZE
do j=1,N
  do i=1,N
    A(j,i)=A(j,i)*B(j,i)
  enddo
enddo
```

Parallelrechner – Vorlesung im SS2008

(84)



## Data layout optimizations

Example: Dense MVM



- Dense matrix vector multiplication (MVM) is a frequently used kernel operation

$$y = y + A * x$$

- DMVM involves data access (assuming square matrix):
  - Matrix: real\*8 A(N,N) → N\*N double words (8 Bytes)
  - Vector1: real\*8 x(SIZE) → N double words (8 Bytes)
  - Vector2: real\*8 y(SIZE) → N double words (8 Bytes)
- Amount of data involved:  $(N^2 + 2 * N)$  Words (=double words)
- #Floating point ops:  $2 * N * N$  Flop  
(1 ADD & 1 MULT for each matrix entry)
- Balance between data transfer and Flop=  $((N^2+2N) W) / (2 N^2 \text{ Flop})$   
=  $(0,5+ 1/N) W/\text{Flop} \sim 0.5 W/\text{Flop}$  (for large N)

Parallelrechner – Vorlesung im SS2008

(85)



## Data layout optimizations

Dense MVM



- If the matrices are big, data transfer should be minimized!
- Calculate number of memory references

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

$$N + N + N * N + N * N$$

$$= 2 * N + 2 * N^2$$

Benefit:

- Lower data transfer for large N
- Contiguous access to A

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

$$N * N + N * N + N * N + N$$

$$= N + 3 * N^2$$

Implementation still away from minimal data amount:  $2 * N + N^2!$

Parallelrechner – Vorlesung im SS2008

(86)



## Minimize Memory References

Dense MVM



- Start with stride 1 access

```
do i=1,N
  tmp=0.d0
  do j=1,N
    tmp = tmp + a(j,i) * x(j)
  end do
  y(i) = y(i) + tmp
end do
```

Innermost loop: two loads and two flops performed:

Balance=1 Word / Flop

Vector x is still loaded N times!

Use outer loop unrolling or blocking to reduce the number of references to vector x!

Parallelrechner – Vorlesung im SS2008

(87)

CX  
HPC

## Minimize Memory References

Dense MVM: Outer loop unrolling



### Outer loop unrolling

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

Outer loop unrolled twice

Innermost loop:  
three loads and four flops: **0.75 W/Flop**  
(1.5 N<sup>2</sup> data transfers)

How about unrolling by 4? **0.625 W/Flop**  
(1.25 N<sup>2</sup> data transfers)

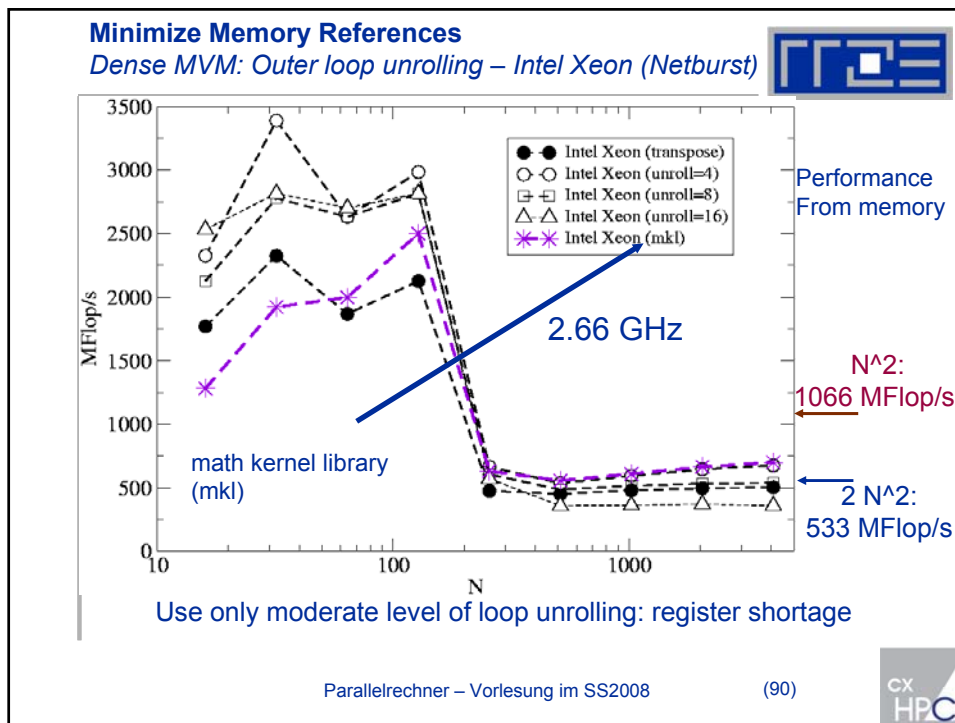
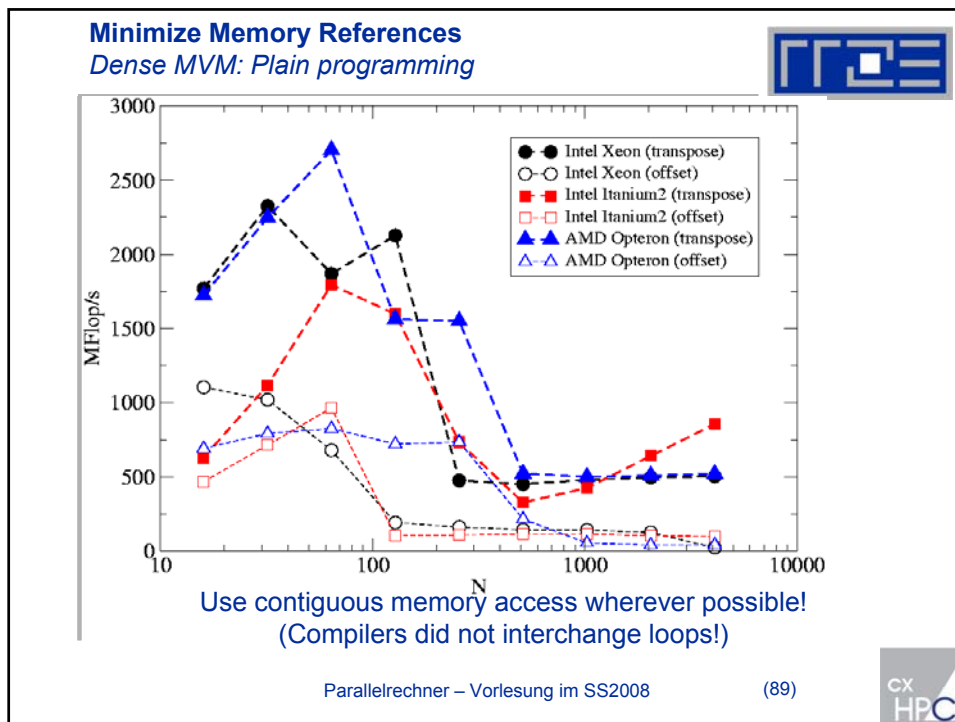
**Watch register spill!**

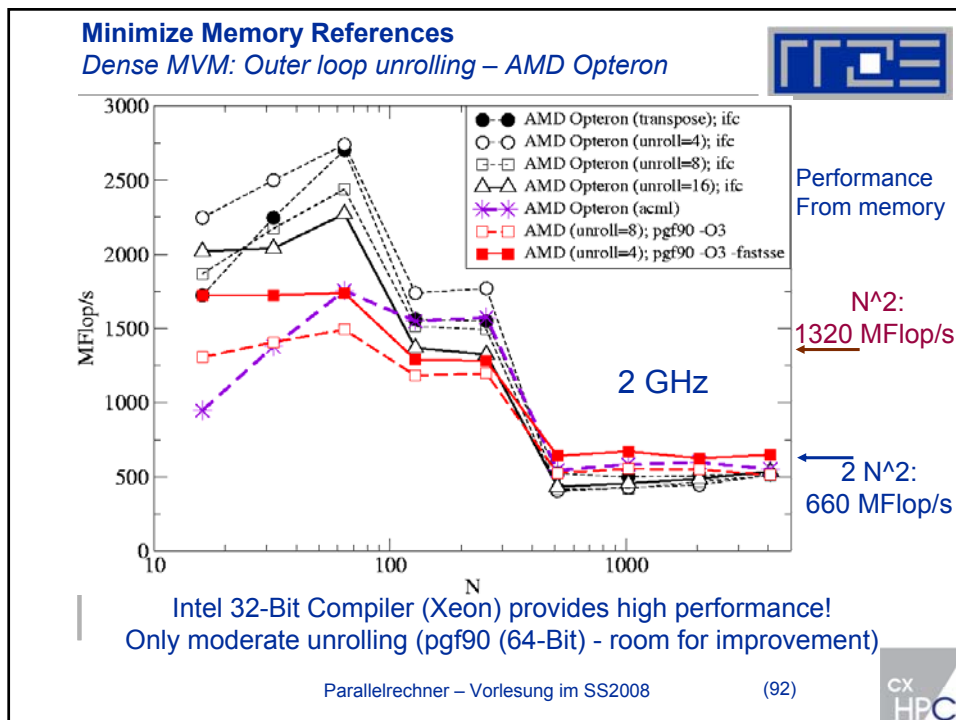
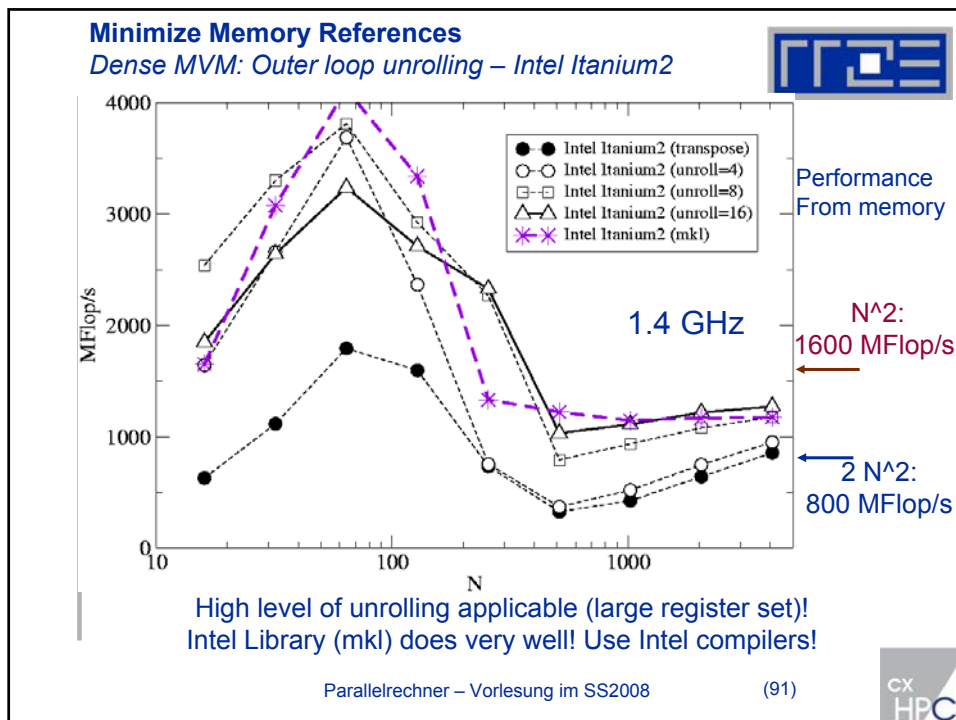
Parallelrechner – Vorlesung im SS2008

(88)

CX  
HPC







## Minimize Memory References

Dense MVM: Blocking



Blocking: Split up inner loop in small chunks (pref. Cache Line Size) and perform all computations.

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

Whole vector  $x$  is loaded from memory or cache to register\_  $N$  times!

```
bs=CLS, nb=N/bs
do k=1,nb
  do i=1,N
    do j=(k-1)*bs+1,k*bs
      y(i)=y(i)+a(j,i)*x(j)
    end do
  end do
end do
```

Vector  $x$  is loaded only once and (re-)used cacheline by cacheline.

CLS: Cache Line Size

Blocking size ( $bs$ ) should be CLS or multiple of it. Upper limit is imposed by cache size

What is the problem here?

Parallelrechner – Vorlesung im SS2008

(93)

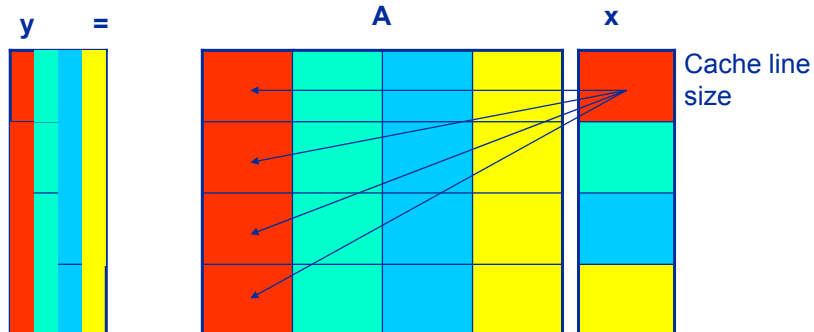


## Minimize Memory References

Dense MVM: Blocking



### Blocking Matrix-Vector Multiply



Save loads ( $x$ ) from memory at the cost of additional store operations ( $y$ ):

- Vector  $x$  is loaded only once instead of  $N$  times
- Vector  $y$  is loaded  $nb$  times instead of only once

Parallelrechner – Vorlesung im SS2008

(94)



## Minimize Memory References

Blocking – Exercise



# Optimize Matrix transpose !

$$A(i,j) = B(j,i)$$

Parallelrechner – Vorlesung im SS2008

(95)

CX  
HPC

## Minimize Memory References

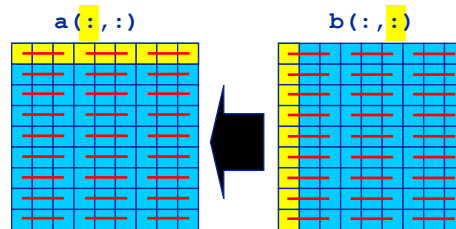
Exercise: Dense matrix transpose



- Simple example for data access problems in cache-based systems

- Naïve code:

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



- **Problem: Stride-1 access for a implies stride-N access for b**
  - Access to a is perpendicular to cache lines ( — )
  - Possibly bad cache efficiency (spatial locality)
- **Remedy: Outer loop unrolling and blocking**

Parallelrechner – Vorlesung im SS2008

(96)

CX  
HPC

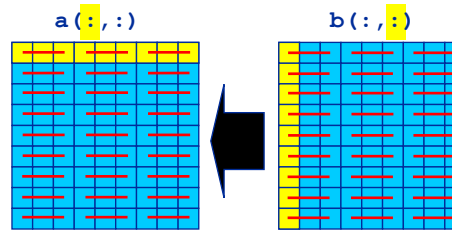
## Minimize Memory References

Exercise: Dense matrix transpose



### Data transfer analysis

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



- Assume a L2 cache of size L2SIZE and a CLS of 16 double (128 Byte)

- All data fits in L2 cache:

$$2 * N1^2 * 8 \text{ Byte} < L2SIZE$$

- All cache lines of B stay in L2 cache until they are fully used:

$$2 * N2 * 16 * 8 \text{ Byte} = 256 * N2 \text{ Byte} < L2SIZE$$

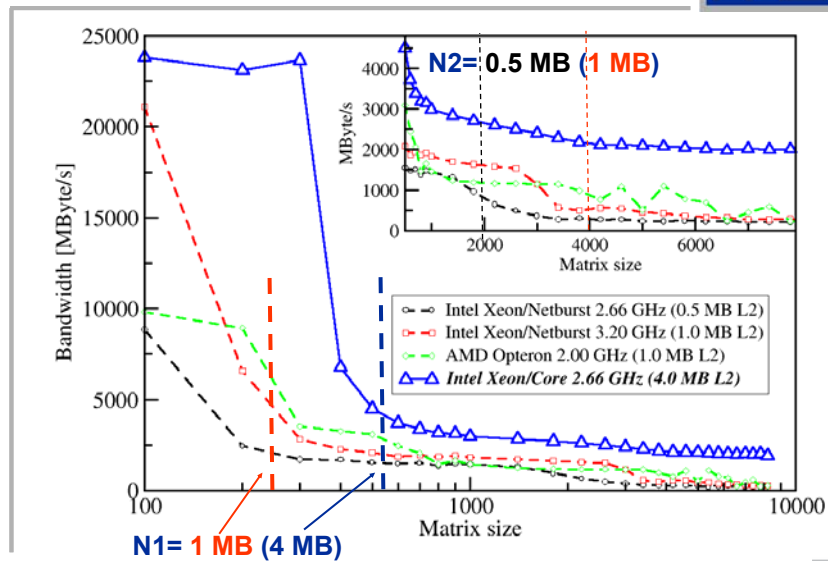
Parallelrechner – Vorlesung im SS2008

(97)



## Minimizing Memory References

Dense matrix transpose: Vanilla version



Parallelrechner – Vorlesung im SS2008

(98)



### Minimizing Memory References

Dense matrix transpose: Unrolling and blocking

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

→ unroll/jam

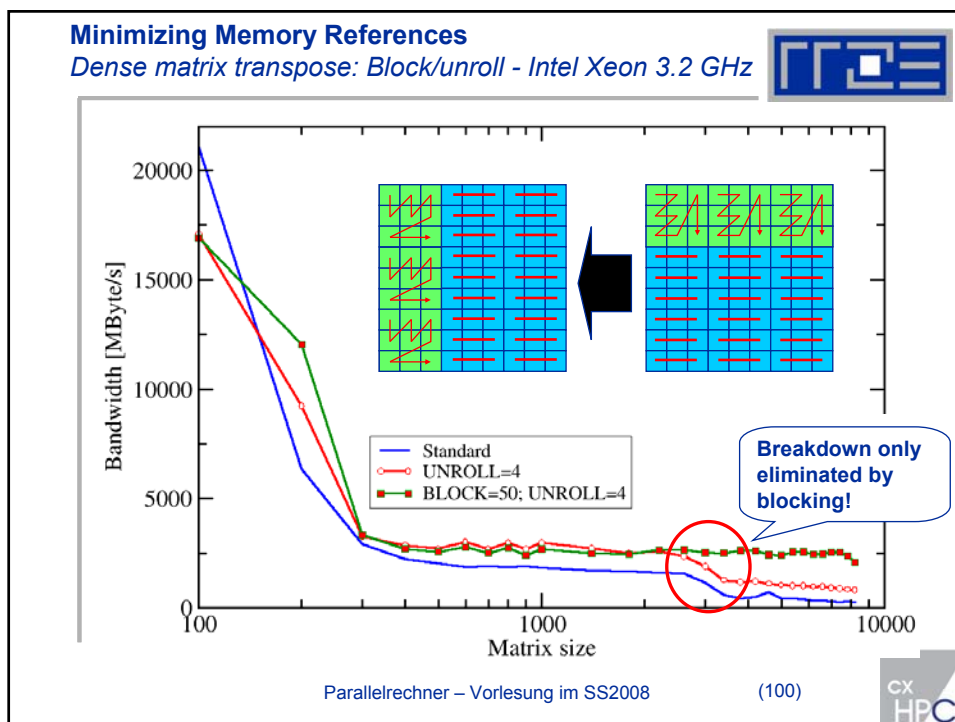
```
do i=1,N,U
  do j=1,N
    a(j,i)      = b(i,j)
    a(j,i+1)    = b(i+1,j)
    ...
    a(j,i+U-1) = b(i+U-1,j)
  enddo
enddo
```

```
do ii=1,N,B
  istart=ii; iend=ii+B-1
  do jj=1,N,B
    jstart=jj; jend=jj+B-1
    do i=istart,iend,U
      do j=jstart,jend
        a(j,i)      = b(i,j)
        a(j,i+1)    = b(i+1,j)
        ...
        a(j,i+U-1) = b(i+U-1,j)
      enddo;enddo;enddo;enddo
```

↙ block

Blocking and unrolling factors (B,U) can be determined experimentally; be guided by cache sizes and line lengths

Parallelrechner – Vorlesung im SS2008 (99)

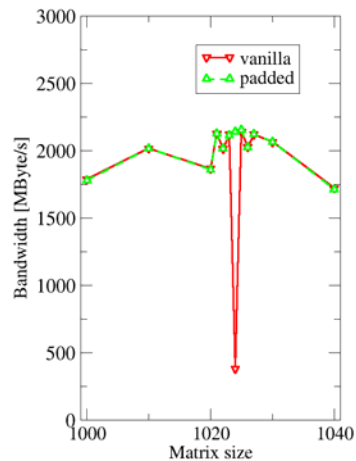


## Minimizing Memory References

Dense matrix transpose: Cache thrashing



- A closer look (e.g. on Xeon/Netburst) reveals interesting performance characteristics:



- Matrix sizes of powers of 2 seem to be extremely unfortunate
  - Reason: Cache thrashing!
- Remedy: Improve effective cache size by padding the array dimensions!
  - $a(1024, 1024) \rightarrow a(1025, 1025)$
  - $b(1024, 1024) \rightarrow b(1025, 1025)$
  - Eliminates the thrashing completely
- Rule of thumb: If there is a choice, use dimensions of the form  $16 \cdot (2k+1)$

Parallelrechner – Vorlesung im SS2008

(101)



## Literature



- G. Hager and G. Wellein  
*Concepts of High Performance Computing*  
<http://www.blogs.uni-erlangen.de/hager/topics/OHN/>
- K. Dowd, C. Severance  
*High Performance Computing*  
O' Reilly, 2nd Edition (ISBN 156592312X)
- S. Goedecker, A. Hoisie  
*Performance Optimization of Numerically Intensive Codes*  
Society for Industrial & Applied Mathematics, U.S. (ISBN 0898714842)
- J.L. Hennessy, D.A. Patterson  
*Computer Architecture – A Quantitative Approach*  
Morgan Kaufmann Publishers, Elsevier 4th Edition (ISBN 0123704901)

Parallelrechner – Vorlesung im SS2008

(102)

