Tutorial @ ICMMES 2009
Guangzhou, China
July 13, 2009

Efficient implementation of lattice Boltzmannflow solvers

Thomas Zeiser
Erlangen Regional Computing Center (RRZE)
University of Erlangen-Nuremberg, Germany

thomas.zeiser@rrze.uni-erlangen.de

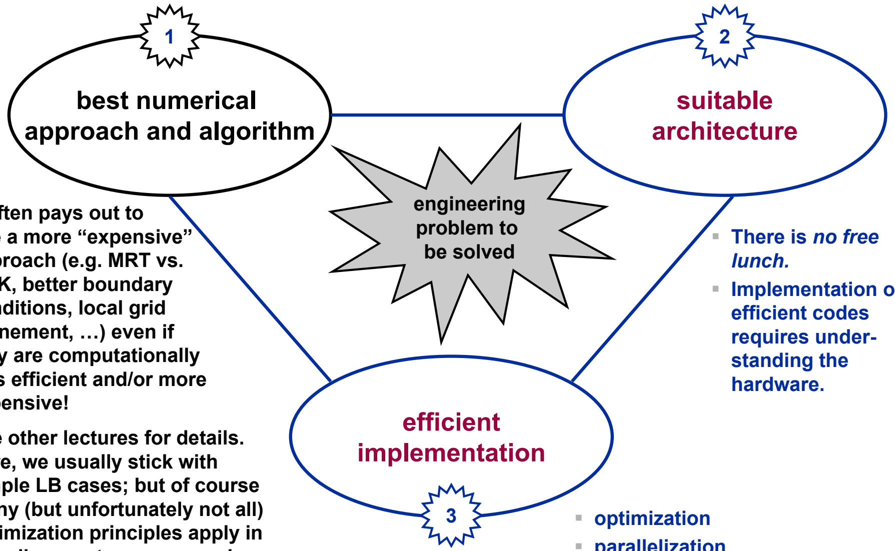
with contributions of G. Hager, G. Wellein, and many others

This work is financially supported by:
KONWIHR II
Federal Ministry of Education and Research
through grant SKALB (01IH08003)

Outline

- **Architectural developments**
  - multi-core everywhere
- **Caches and memory hierarchies**
  - cache and cache thrashing
- **Performance modeling**
  - expected performance vs. sustained performance
- **Optimization**
  - common sense optimizations, minimizing data access, effect of data layout
- **Parallelization**
  - OpenMP, MPI, parallel scalability, domain decomposition
- **Tools**
  - make, version control systems, MPI tracing

Preliminaries: what will / wont be covered here


1
best numerical approach and algorithm
2
suitable architecture
engineering problem to be solved
efficient implementation
3

- It often pays out to use a more "expensive" approach (e.g. MRT vs. BGK, better boundary conditions, local grid refinement, …) even if they are computationally less efficient and/or more expensive!
- See other lectures for details. Here, we usually stick with simple LB cases; but of course many (but unfortunately not all) optimization principles apply in a similar way to more complex scenarios, too.

- There is *no free lunch.*
- Implementation of efficient codes requires understanding the hardware.

- optimization
- parallelization
- sustainable development

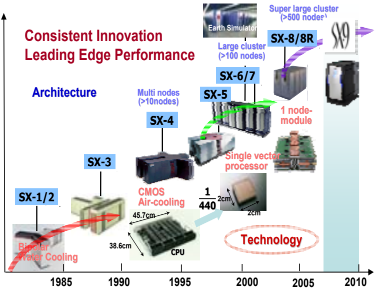
Architectural developments

Why we still have to worry about performance.

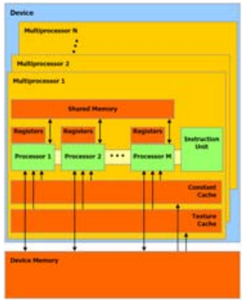
Currently available compute platforms

Commodity cluster

Vector computers



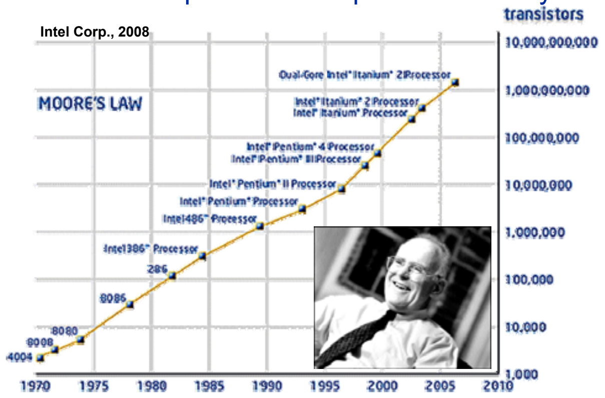
GPUs and other accelerators



- Vector computers are (unfortunately) a dieing species; but other architectures get more and more vector-like feature.
- GPUs and accelerators are not yet mature enough for general application (at least in my opinion).
- → focus on commodity CPUs

Ever growing processor speed & Moore´s Law

- 1965 G. Moore (co-founder of Intel) claimed
  #transistors on processor chip doubles every 12-24 months



Intel Corp., 2008

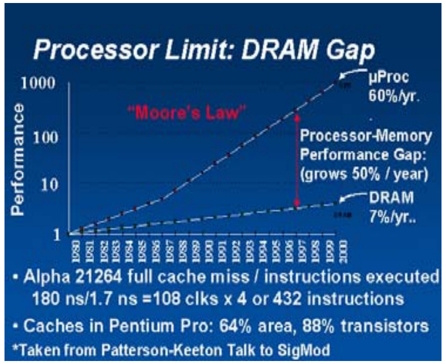
Attention:
Moore's law is about *number of transistors* and **not** about *speed or performance*, but …

- Processor speed *grew* roughly at the same rate
  My computer: 350 MHz (1998) – 3.000 MHz (2004)
  Growth rate: 43% p.y. → doubles every 24 months
- Why worry about performance?

Why worry about performance: memory 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.
- On-chip memory controllers recently gave boost (especially for multi-socket nodes); but still, memory bandwidth remains an issue.



- Alpha 21264 full cache miss / instructions executed 180 ns/1.7 ns =108 clks x 4 or 432 instructions
- Caches in Pentium Pro: 64% area, 88% transistors

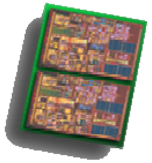
*Taken from Patterson-Keeton Talk to SigMod

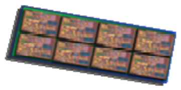
Optimization of main memory access is mandatory for most applications.

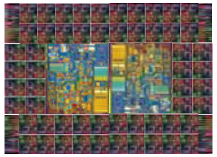
Why worry about performance: multi-core

Multi-Core Processors – The game is over…

- Problem: Moore's law is still valid but increasing clock speed hits a technical wall (power consumption / heat).
- Solution: Reduce clock speed of processor but put 2 (or more) processors (cores) on a single silicon die.
- We will have to use *many* but *less powerful* processors in the future.

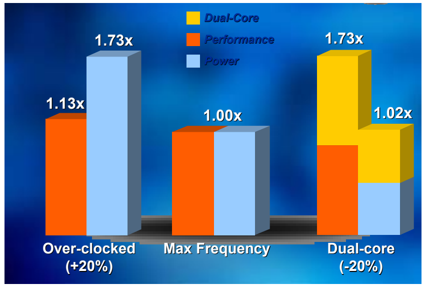






Parallelization is mandatory for most applications.

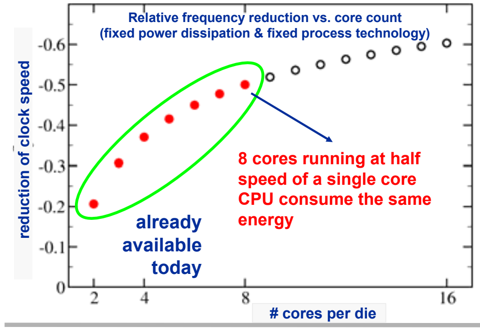
Making use of all the transistors available

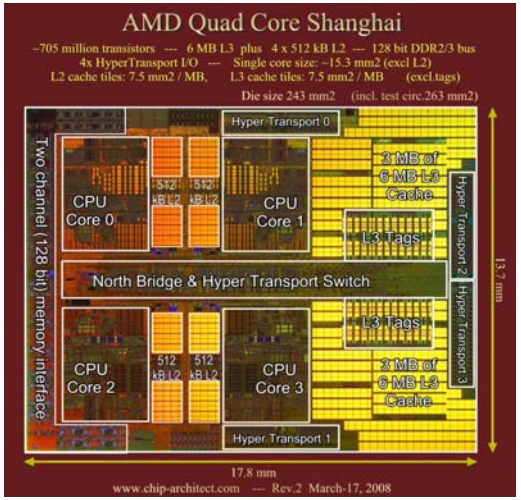


by courtesy of Intel, 2006

→ more cores

→ more/larger caches





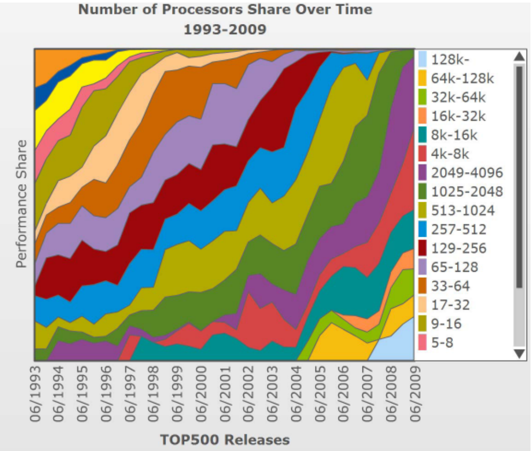
AMD Quad Core Shanghai
~705 million transistors --- 6 MB L3 plus 4 x 512 kB L2 --- 128 bit DDR2/3 bus
4x HyperTransport I/O --- Single core size: ~15.3 mm2 (excl L2)
L2 cache tiles: 7.5 mm2 / MB, L3 cache tiles: 7.5 mm2 / MB (excl.tags)
Die size 243 mm2 (incl. test circ.263 mm2)
Hyper Transport 0
Two channel (128 bit) memory interface
CPU Core 0
512 kB L2
512 kB L2
CPU Core 1
3 MB of 6 MB L3 Cache
L3 Tags
Hyper Transport 2
Hyper Transport 3
North Bridge & Hyper Transport Switch
CPU Core 2
512 kB L2
512 kB L2
CPU Core 3
L3 Tags
3 MB of 6 MB L3 Cache
Hyper Transport 1
13.7 mm
17.8 mm
www.chip-architect.com --- Rev.2 March-17, 2008

http://chip-architect.com/news/Shanghai_Nehalem.jpg

Top500 trends and cores on the desktop

- Number of processors is rapidly “exploding”
- due to more nodes but also due to multi-core chips
- more and more systems become “hybrid”

Rank	Site	System	Cores
1	LANL	IBM Roadrunner Opteron+Cell	129 600
2	Oak Ridge	Cray XT5	150 152
3	FZJ, DE	IBM BlueGene/P	294 912
4	NASA	SGI Altix ICE (Intel Harpertown)	51 200
8	TAC	Sun Ranger (Opteron)	62 976
15	*SSC, CN*	*QC AMD Opteron, Windows HPC*	*30 720*
22	JP	Earth Simulator 2, NEC SX-9/E	1 280



And on the desktop?

- single/dual/quad/… core CPUs
- or 960 cores with 4 GPUs

Caches and memory hierarchies of modern processors

Programming basics

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

```
real*8 a(0:3,0:3)   ! FORTRAN: column-major
```

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

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

```
double a(4,4)          // C/C++:  row-major
```

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

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

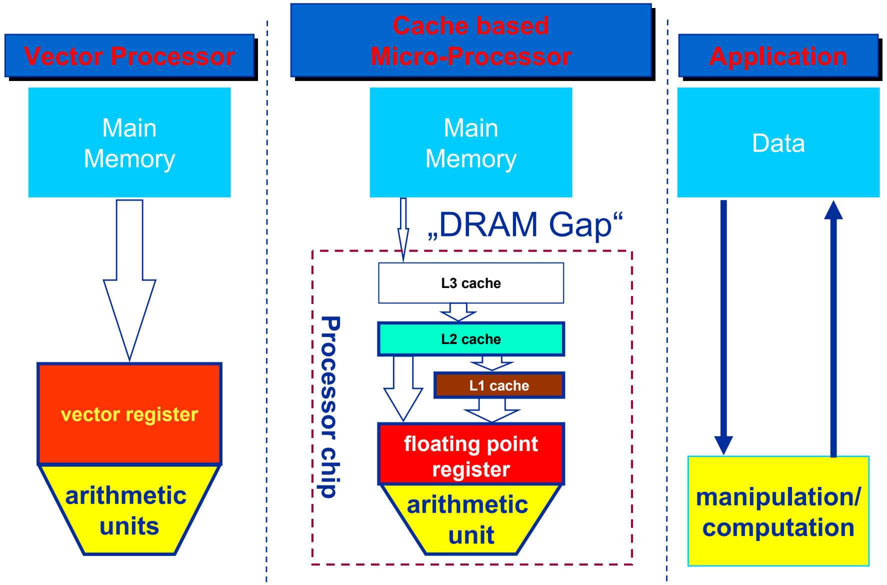
row *i* column *j*

- Different ordering of nested loops required for “stride 1 access” (i.e. consecutive memory access)!

```
do j=0,3
   do i=0,3
      a(i,j)= …
   enddo
enddo
```

```
for (i=0; i<4; i++) {
   for (j=0; j<4;j++) {
      a(i,j)= …;
   }
}
```

Memory hierarchies

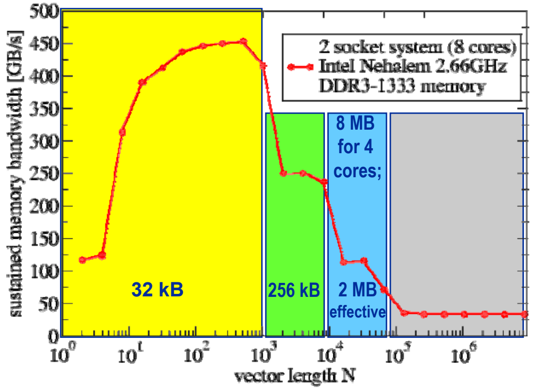


Memory hierarchies of recent (dual-core) CPUs

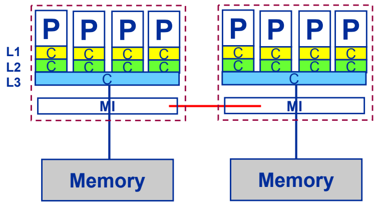
*SSE2		DC Intel Xeon5160	DC AMD Opteron2362	*DC Intel Itanium2*	QC Intel Nehalem
Peak Performance Core frequency		**12.0 GFlop/s** 3.0 GHz	**5.2 GFlop/s** 2.6 GHz	***6.4 GFlop/s*** *1.6 GHz*	**10.64 GFlop/s** 2.66 GHz
#Registers		**16 / 32***	**16 / 32***	***128***	**16 / 32***
L1(D)	**Size**	32 kB	64 kB	*16 KB*	32 kB
	Raw BW	96 GB/s	41.6 GB/s	*51.2 GB/s*	?
	Latency	2 cycles	3 cycles	*1 cycle*	4 cycles
L2	**Size**	**4 MB** (for 2 cores)	**1 MB**	*256 KB*	256 kB
	Raw BW	**96 GB/s**	**41.6 GB/s**	*51.2 GB/s*	?
	Latency	**7 cycles**	**~13 cycles**	*5-6 cycles*	~10 cycles
L3	**Size**			***6 / 12 MB***	**8 MB** (for 4 cores)
	Raw BW			***51.2 GB/s***	**?**
	Latency			***12-13 cycles***	**40-50 cycles**
Memory	**Raw BW**	**10.6 GB/s**	**6.4 GB/s**	***8.5 GB/s***	**32 GB/s**
	Latency	**~200 ns** ~600 cycles	**< 100 ns**	***~200 ns***	**~75 ns**

Vector TRIAD test: sustained memory bandwidth

```
double precision :: a(maxLen), b(maxLen), c(maxLen), d(maxLen)
do j=1, repeatCount
   !DEC$ VECTOR NONTEMPORAL
   do i=1, N
      a(i) = b(i) + c(i) * d(i)
   end do
   ! obscure statement to prevent loop optimization
end do
```

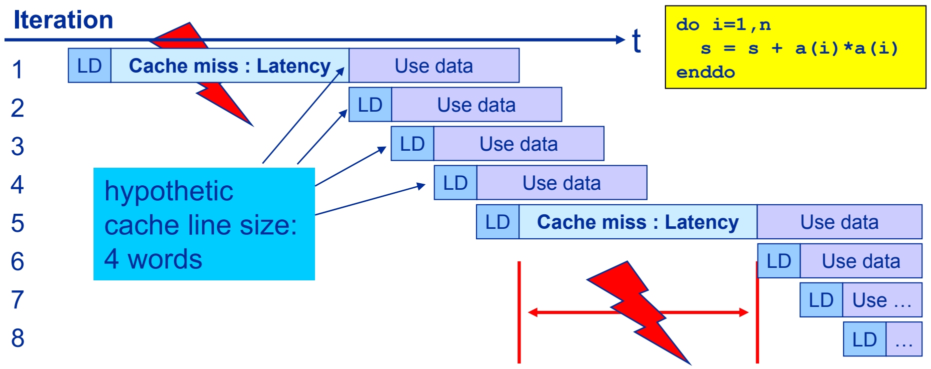


Nehalem
„Core i7“



Memory hierarchies: cache structure

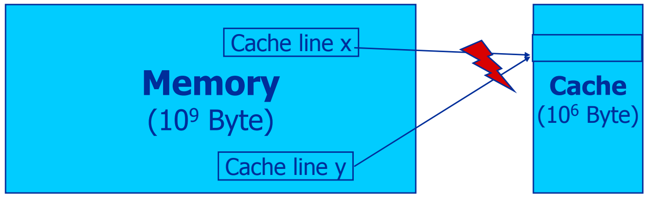
- **caches are organized in cache lines that are fetched/stored as a whole (e.g. 64 bytes = 8 double words)**
  - if one item is needed, the cache line it belongs to is fetched (miss)
  - cache line fetch/load has large latency penalty
  - “neighboring“ items can then be used from cache



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

Memory hierarchies: cache structure *(cont.)*

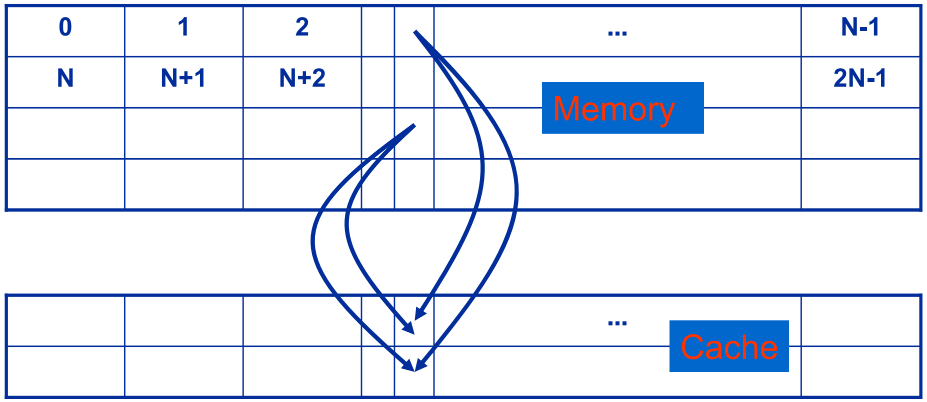
- **Cache line data is always consecutive**
  - cache use is optimal for **contiguous access (stride 1)**
  - non-consecutive access reduces performance (worst case: ≥ cache line size)
  - ensure **spatial locality** by blocking or **optimizing data layout** → *see lattice Boltzmann part later on*
- **Modifying/writing data (usually) requires loading the data first**
  - "read for ownership" (**RFO**)
  - unless "non temporal stores" can be used (requires e.g. vectorizable code and proper alignment)
- **Caches (~MB) must be mapped to memory locations (~GB)**



Memory hierarchies: cache mapping

- **"Cache mapping":**
  - pairing of memory locations with cache line
  - e.g. mapping 1 GB of main memory to 512 KB of cache
- **Directly mapped cache:**
  - every memory location can be mapped to exactly one cache location
  - if cache size=n, i-th memory location is mapped to cache location $mod(i,n)$
  - 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!
  - no penalty for stride-one access
- **Set-associative cache:**
  - m-way associative cache of size m x n: each memory location i can be mapped to the m cache locations $j*n$+mod(i,n), j=0..m-1
  - if all m locations are taken, one has to be overwritten on next cache load; different strategies (least recently used (LRU), random, not recently used (NRU)

Memory hierarchies: associative caches



Example: 2-way associative cache. Each memory location can be mapped to two cache locations.

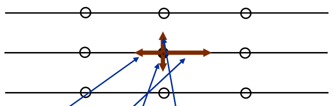
e.g. size of main memory= 1 GByte; Cache Size= 256 KB
→ 8192 memory locations are mapped to two *cache locations*

Memory hierarchies: cache thrashing

- **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.**
- ***Effective* cache size is usually less than *m* x *n* for real-world applications.**
- **Warning: Using powers of 2 in the leading array dimensions of multi-dimensional arrays should be avoided! "Padding" may help.**
  See example on the following slides.

Memory hierarchies: cache thrashing example

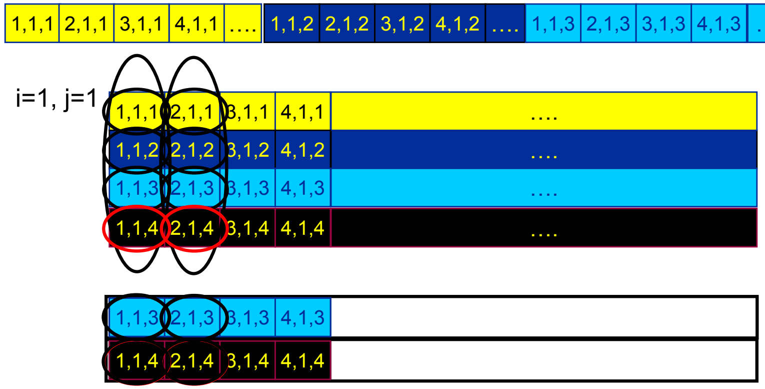
Example: 2D – square lattice
At each lattice point the 4 velocities for each of the 4 directions are stored.



```
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
```

Memory hierarchies: Cache thrashing example

memory-to-cache mapping for `vel(1:16, 1:16, 4)`
Hypothetic 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 | …. `Vel(1:16,1:16,1)`
1,1,2 | 2,1,2 | 3,1,2 | 4,1,2 | …. `Vel(1:16,1:16,2)`
1,1,3 | 2,1,3 | 3,1,3 | 4,1,3 | …. `Vel(1:16,1:16,3)`
1,1,4 | 2,1,4 | 3,1,4 | 4,1,4 | …. `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

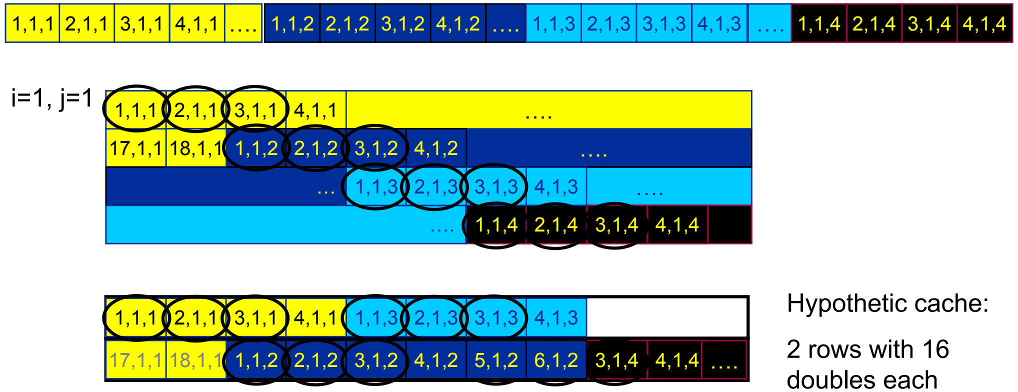
Hypothetic cache:

2 rows with 16 double each

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

Memory hierarchies: cache thrashing solved

Memory-to-cache mapping for `vel(1:18, 1:18, 4)`
Hypothetic 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
17,1,1 | 18,1,1 | 1,1,2 | 2,1,2 | 3,1,2 | 4,1,2 | 5,1,2 | 6,1,2 | 3,1,4 | 4,1,4 | ….

Hypothetic cache:

2 rows with 16 doubles each

(most) cache lines need only be loaded **once** from memory to cache!
→ "padding" of the array solved the cache thrashing issue

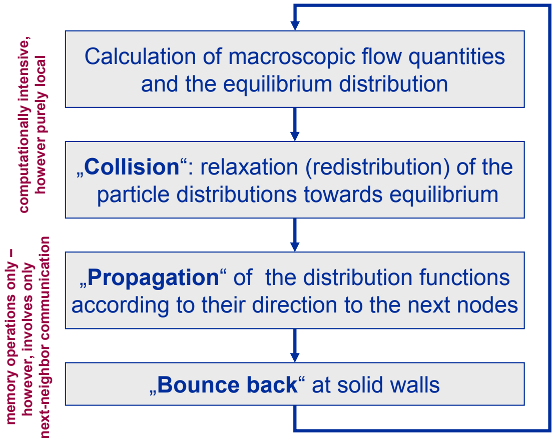
Performance modeling

expected performance vs. sustained performance

How close to the "optimum" am I ?

Lattice Boltzmann method: basic algorithm

The evolution of the particle distribution functions f_i at each lattice node is calculated in discrete time steps.


computationally intensive, however purely local
memory operations only – however, involves only next-neighbor communication

Calculation of macroscopic flow quantities and the equilibrium distribution

„**Collision**“: relaxation (redistribution) of the particle distributions towards equilibrium

„**Propagation**“ of the distribution functions according to their direction to the next nodes

„**Bounce back**“ at solid walls

Computationally: (D3Q19)

- **explicit Jacobi-like iteration scheme**
- **19 double precision floating point values stored per node**
- **19-point stencil; data exchange with nearest neighbors**
- **~ 200 Flops per node update** (for BGK or TRT collision model)
- ***algorithmic balance:*** 456 bytes / 200 flops = **2.3 B/flop**
- ***system balance* of a 2-socket Nehalem node (2.66 GHz, no SMT):** 32 GB/s *sustained* stream bandwidth / 85.12 GFlop/s = **0.37 B/flop**

→ **we will usually be *memory bound***

Estimation of performance: in-cache case

our performance metric for lattice Boltzmann codes:

- **million (fluid) lattice node updates per second – MLUP/s**

performance estimation

- **general assumptions**
  - D3Q19 lattice
  - 200 floating point operations per fluid node update (reasonable for BGK/TRT)
- **Case 1: data transfer is infinite fast (all data fits into cache):**
  - max. MLUP/s = PeakPerformance / (200 Flop/node update)
  - single 2.66 GHz Intel Core i7 core = 10,640 MFLOP/s → max. 53 MLUP/s
  - in reality even simple kernels do not achieve peak performance and transfer speed is finite even for caches

  → thus, this upper limit is more hypothetical than real

  → in-cache performance depends on many small details and can vary e.g. significantly from compiler release to compiler release

Estimation of performance: memory-bound case

- **Crossover between cache and memory bound computations**
  - Complete domain no longer fits into outermost cache `(N=Nx=Ny=Nz)` `2 *19 * N^3 * 8 Byte` ~ L2/L3 cache size
    → 1 MB cache: N ~ 14-15; 4 MB cache: N ~ 23-24
- **Cache 2: data must always be transferred from/to main memory**
  - assumption: full use of each cache line loaded
  - data to be transferred for a single fluid node update: (including RFA) `(2+1) * 19 * 8 Byte` → 456 Bytes/(node update)
  - max. number of lattice site updates per second MLUPs: MaxMLUPs = MemoryBandwidth / (456 Bytes/node update)
  - (effective) MemoryBandwidth = 3-12 GByte/s → MaxMLUPs ~ 6-26 MLUPs
- **Verification of efficiency of data access using hardware perf counter**
  - *number of cache misses* is a bad performance metric (at least on Intel Xeon systems due to hardware prefetcher, etc.)
  - more reliable: *memory bus (FSB) utilization* and *number of buss accesses*
  - minimum number of bus accesses: N^3 * timesteps * 456 / 64

size of a cache line

Implementation and optimization of a 3-D lattice Boltzmann kernel

- **common sense optimizations**
- **minimizing data access**
- **effect of data layout**

Basic optimizations: preliminaries (general)

General guidelines – some common sense optimizations

1. **do less work**
   - eliminate common sub-expressions
   - avoid branches; move if tests outside of inner loops
2. **avoid expensive operations**
   - a division is much more expensive than a multiplication
   - does the compiler realize that **`x**2.0`** is just **`x*x`**
   - trigonometric expressions, etc.
   - "strength reduction"; tabulating values; …
3. **shrink working set**
   - use appropriate data types (e.g. float vs. double)
   - only calculate / store what is really required
4. **use appropriate compilers and compiler flags**
   - optimization; inlining; …
   - tell the compiler if data references disjoint locations (i.e. no aliasing)

further reading

G. Hager and G. Wellein: *Part 1: Architectures and Performance Characteristics of Modern High Performance Computers. Part 2: Optimization Techniques for Modern High Performance Computers.* In Fehske et al., Lecture Notes Phys. 739, pp 681-730 and pp. 731-767 (2008), ISBN: 978-3-540-74685-0. (Springer, Berlin)

Basic optimizations: preliminaries (LBM specific)

1. **analyze relaxation step (I)**
   - many operations can be eliminated (common sub-expressions, zero-velocity components, etc.) — do not rely on compiler!
   - # of floating point operations depends on compiler & optimization level

```
! Calculate ux velocity component
ux=0.d0
do i=0,18
   ux=ux+ex(i)*f(i,x,y,z,t)
enddo
! worst case: loop, 38 LD; 19 MultAdd
```

```
ux=f(E ,x,y,z,t)+f(NE,x,y,z,t)+
   f(SE,x,y,z,t)+f(TE,x,y,z,t)+
   f(BE,x,y,z,t)-f(W ,x,y,z,t)-
   f(NW,x,y,z,t)-f(SW,x,y,z,t)-
   f(TW,x,y,z,t)-f(BW,x,y,z,t)
! 10 LD; 9 Add
```

   - even worse: (C++) function calls which are not inlined; e.g. **`getF(…)`**

2. **analyze relaxation step (II)**
   - compare **`usq=ux**2.0+uy**2.0+uz**2.0`** ←→ **`usq=ux*ux+uy*uy+uz*uz`**

3. **combine *collide & stream* step in a single loop to minimize data transfer!** Otherwise data may have to be transferred twice.

4. **e.g. for Intel compiler: `-O3 -xSSE4.2 -fno-alias`**
   (`-fno-alias` of course only if no pointer aliasing is used)

Basic optimizations: initial implementation

- **Starting point: straight forward "full matrix" approach with toggle index and marker-and-cell flag field**
  - separate storage for even/odd time steps ("source/destination") to avoid data dependencies
  - discrete velocity as additional array index
  - ghost layer (to avoid special algorithm at domain boundaries)

→ **5-D array *f(Q, x,y,z, t)***

- **In the following, it is assumed that by increasing the first index by one, you go to the physically next location in memory (FORTRAN, column-major).**

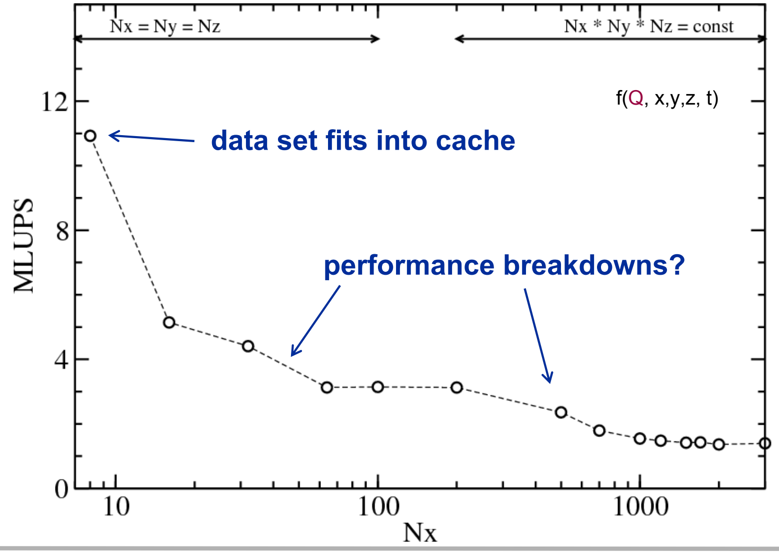
```
real*8 f(0:18,0:Nx+1,0:Ny+1,0:Nz+1,0:1)
do z=1,Nz; do y=1,Ny; do x=1,Nx
  if( fluidcell(x,y,z) ) then
    LOAD f( 0,x,y,z,t)
    LOAD f( 1,x,y,z,t)
    …
    LOAD f(18,x,y,z,t)
    Relaxation (complex computations)
    SAVE f( 0,x  ,y  ,z  ,t+1)
    SAVE f( 1,x+1,y  ,z  ,t+1)
    SAVE f( 2,x  ,y+1,z  ,t+1)
    SAVE f( 3,x-1,y+1,z  ,t+1)
    …
    SAVE f(18,x  ,y-1,z-1,t+1)
  endif
enddo; enddo; enddo
```

#load operations: 19*Nx*Ny*Nz + 19*Nx*Ny*Nz
#store operations: 19*Nx*Ny*Nz

If cache line of store operation is not in cache it must be loaded first (RFO) !

Performance characteristics of initial code

Intel Xeon 3.4 GHz (1 MB L2; 5.3 GByte/s) → max. 5-6 MLUP/s

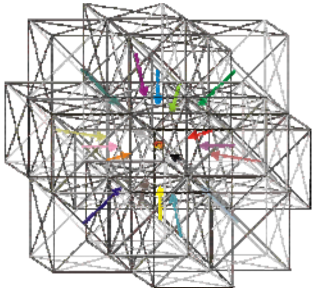


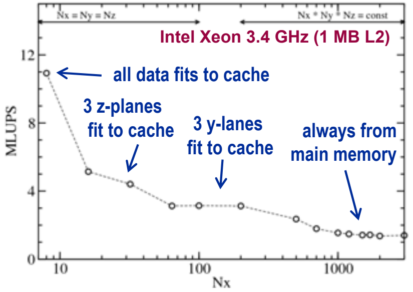
data set fits into cache

performance breakdowns?

Explanation of the performance breakdowns

- **A single node is updated by 18 write accesses from 3 successive z-planes.**
- **total amount of data for 3 successive z-planes:**
  Mem_z ~ 3 * 2 * 19 *8 * (Nx+2)*(Ny+2) Bytes
- **cache lines must be reloaded if Mem_z ~ L2/L3 cache**
- **1 MB cache → Nx=Ny ~ 33**
- **next breakdown if only 3 y-lines fit into cache**

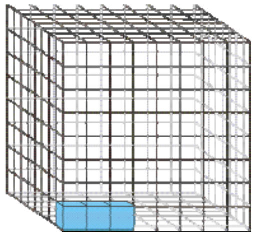




Drawing by courtesy of S. Donath, LSS/RRZE

Optimization of data access: spatial blocking

- **Increase spatial locality by spatial blocking**



```
real(8) f(0:18,0:Nx+1,…,0:1)
do zz=1,Nz,blcksize
 do yy=1,Ny,blcksize
  do xx=1,Nx,blcksize

   do z=zz,min(Nz,zz+blcksize-1)
    do y=yy,min(Ny,yy+blcksize-1)
     do x=xx,min(Nx,xx+blcksize-1)
            if( fluidcell(x,y,z) ) then
            …………
            endif
     enddo
    enddo
   enddo

   enddo
 enddo
enddo
```

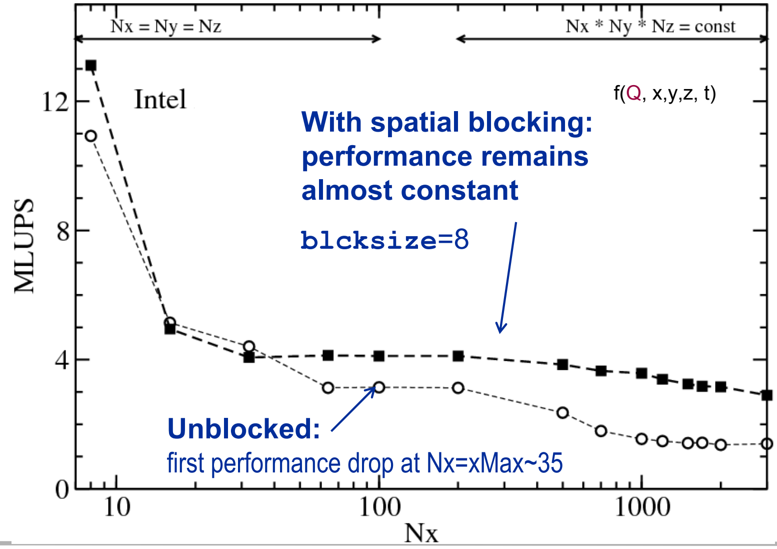
Correct choice of `blcksize`?

- 2*19*`blcksize`^{3} Byte < L2/L3
  → `blcksize` ~8-10
- 2*19***3***`bcksize`^{2} Byte < L2/L3
  → `blcksize` ~25-30

Schematic animation by courtesy of S. Donath, LSS/RRZE

Performance characteristics with 3-D blocking

Intel Xeon 3.4 GHz (1 MB L2; 5.3 GByte/s) → max. 5-6 MLUP/s



With spatial blocking: performance remains almost constant

`blcksize=8`

Unblocked:
first performance drop at Nx=xMax~35

Optimization of data access: data layout

- **Starting point: straight forward "full matrix" approach with toggle index and marker-and-cell flag field**
  - separate storage for even/odd time steps ("source/destination") to avoid data dependencies
  - discrete velocity as additional array index
  - ghost layer (to avoid special algorithm at domain boundaries)

→ **5-D array:** **F(0:18, 0:xMax+1, 0:yMax+1, 0:zMax+1, 0:1)**

- **By increasing the first index by one, you go to the physically next location in memory (FORTRAN, column-major).**
- **In principle, any permutation of these indices can be used… …but which is the most efficient?**
  - `F(Q, x,y,z, t)` **„collision optimized"** **(„array of structures")**
  - `F(x,y,z, Q, t)` **„propagation optimized"** **(„structure of arrays")**

Effect of different data layouts

array-of-structures

```
real*8 f(0:18,0:Nx+1,0:Ny+1,0:Nz+1,0:1)
do z=1,Nz; do y=1,Ny; do x=1,Nx
  if( fluidcell(x,y,z) ) then
    LOAD f( 0,x,y,z,t)
    LOAD f( 1,x,y,z,t)
    …
    LOAD f(18,x,y,z,t)
    Relaxation (complex computations)
    SAVE f( 0,x   ,y   ,z   ,t+1)
    SAVE f( 1,x+1,y   ,z   ,t+1)
    SAVE f( 2,x   ,y+1,z   ,t+1)
    SAVE f( 3,x-1,y+1,z   ,t+1)
    …
    SAVE f(18,x   ,y-1,z-1,t+1)
  endif
enddo; enddo; enddo
```

„collision“

stride-1 memory access

„propa-gation“

large distances in memory; cache lines rarely reusable ⇩ loop blocking mandatory !?

structure-of-arrays

```
f(0:Nx+1,0:Ny+1,0:Nz+1,0:18,0:1)
do z=1,Nz; do y=1,Ny; do x=1,Nx
  if( fluidcell(x,y,z) ) then
    LOAD f(x,y,z, 0,t)
    LOAD f(x,y,z, 1,t)
    …
    LOAD f(x,y,z,18,t)
    Relaxation (computations)
    SAVE f(x   ,y   ,z   , 0,t+1)
    SAVE f(x+1,y+1,z   , 1,t+1)
    SAVE f(x   ,y+1,z   , 2,t+1)
    SAVE f(x-1,y+1,z   , 3,t+1)
    …
    SAVE f(x   ,y-1,z-1,18,t+1)
  endif
enddo; enddo; enddo
```

19 different cache lines; but re-usable

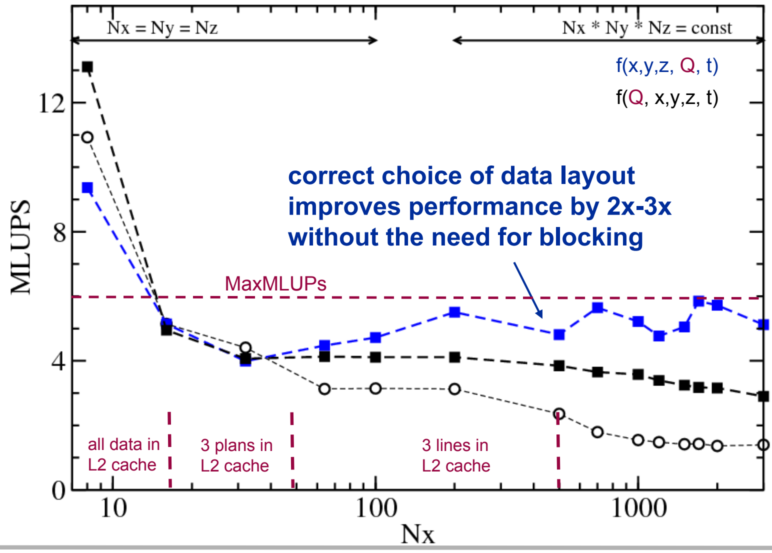
19 cache lines; but all are reusable

Data layout has a significant effect on cache based systems:

- **structure-of-arrays layout has high spatial data locality built-in if 38 cache lines stay in the cache!** (38 * 128 Byte ~ 5 kByte << L2/L3 caches)
- ***but* watch possibility of cache thrashing for *structure-of-arrays* layout**

Performance characteristics of 2nd data layout

Intel Xeon 3.4 GHz (1 MB L2; 5.3 GByte/s) → max. 5-6 MLUP/s



correct choice of data layout improves performance by 2x-3x without the need for blocking

f(x,y,z,Q,t) structure-of-arrays; without any blocking

f(Q,x,y,z,t) array-of-structures with/without spatial blocking

Optimization of data access: cache thrashing

- **Generally avoid powers of 2 in the leading dimension of (multidimensional) arrays.**
  → use “array padding” if necessary

!

In case of the structure-of-arrays data layout:

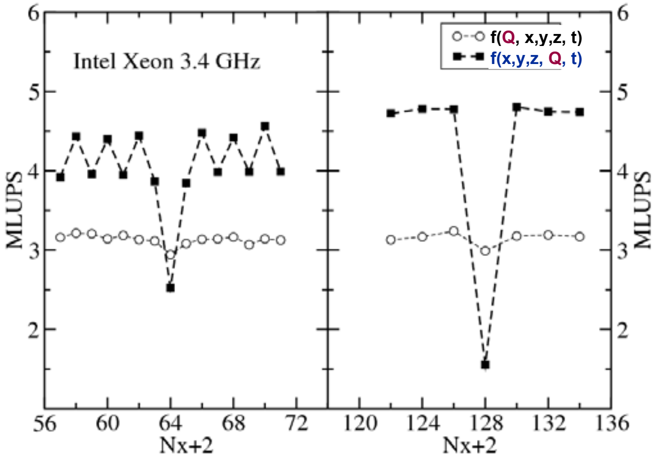
Use

`F(0:128,0:128,…)`

instead of

`F(0:127,0:127,…)`

The array-of-structures data layout generally does not suffer from cache thrashing as the leading dimension is 19.



Optimized data movement: splitting innermost loop

```
0  ! temporary arrays to hold pre-calculated values
   real(8) :: ux(Nx), uy(Nx), uz(Nx), dens(Nx)

   do z=1,Nz; do y=1,Ny
      do x=1,N
         dens(x) = sum( f(x,y,z,:,t) )
1        ux(x)  = f(x,y,z,NE,t) + f(x,y,z,SE,t) + …
         uy(x)  = … ; uz(x) = …
      end do
      ! update (relax and advect) first fraction of directions
      do x=1,Nx
          feq_com=…; t2x2=…; fac2=…; ! calc common coefficients
          ui= ux(x)+uy(x); …
2         sym = omega_h(f(x,y,z,NE,t)+f(x,y,z,SW,t)
                - fac2*ui*ui-t2x2*feq_com)
          asy = asyo_h*(f(x,y,z,NE,t)-f(x,y,z,SW,t)-3.d0*t2x2*ui)
          f(x+1,y+1,z,NE,t1) = f(x,y,z,NE,t) - sym - asy
          f(x-1,y-1,z,SW,t1) = f(x,y,z,SW,t) - sym + asy
          …
      end do
3     ! update (relax and advect) 2nd/3rd fraction of directions
      do x=1,Nx; similar calculations; end do
   end do; end do
```

Performance characteristics with loop splitting

Woodcrest:
Intel Xeon 5150

2.67 GHz
4 MB L2 (2 cores)

FSB1333
(10.6 GB/s)

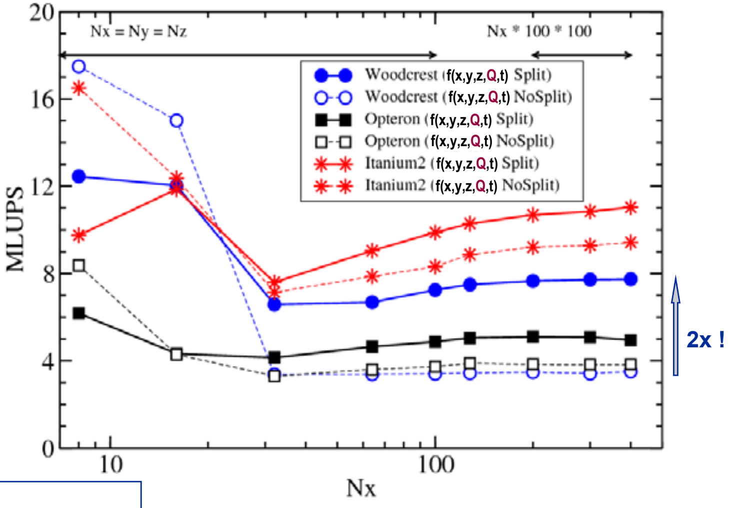
Opteron:
AMD Opteron 270

2.0 GHz
1 MB L2 (per core)

6.4 GB/s

Itanium2:
Intel Itanium2/Madison9M

1.6 GHz / 6 MB L3 / 8.5 GB/s

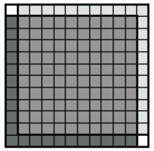


most significant performance gain on Intel Xeon (Woodcrest) system

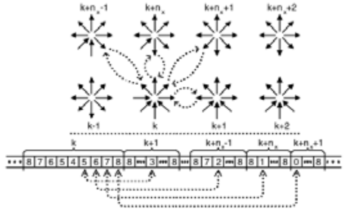
Reducing the memory footprint

further reading

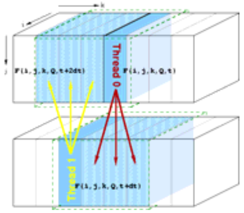
- **Compressed grids**
  - T. Pohl, M. Kowarschik, J. Wilke, K. Igelberger, U. Rüde: *Optimization and profiling of the cache performance of parallel lattice Boltzmann codes.* Par. Proc. Lett., 13:4 (2003) 549-560.



- **Intelligent in-place swapping**
  - K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, J. Westerholm: A*n efficient swap algorithm for the lattice Boltzmann method.* Comp. Phys. Comm 176 (2007) 200–210.



- **Multi-core aware wave socket parallelization**
  - G. Wellein, et al.; *in preparation, 2009.* cf. my talk on Friday in the HPC session!



→ **All three methods basically eliminate the toggle index and thus reduce the memory footprint by approx. 50%.**

Summary of basic part on efficient LBM coding

- **Efficient code implementation requires insight into memory hierarchy of modern processors.**
- **Data layout analysis and/or spatial blocking is mandatory to optimize data transfer between main memory and processor.**
- **Optimizing single processor performance and parallelization are tightly connected to the use of multi-core processors.**
- **Parallel computing does *not* supersede sequential optimization .**

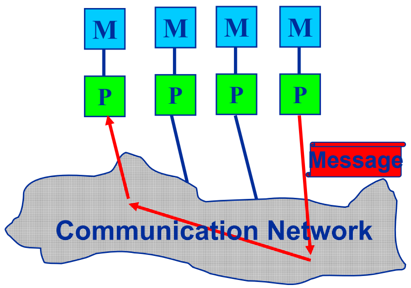
Parallelization

- **OpenMP and ccNUMA**
- **MPI**
- **how to decompose the domain**
- **limits of scalability: strong/weak scaling; Amdahl's/Gustafsson's law**

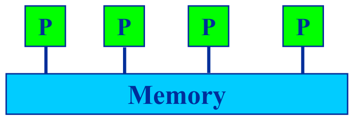
Two paradigms for parallel programming

Partially based on LRZ/RRZE course on parallel programming. By courtesy of R. Bader.

- **Distributed Memory**
  - data exchange between processes: send/receive messages via library (MPI = "Message Passing Interface")
  - explicit programming required
  - up to very large processor numbers possible



- **Shared Memory**
  - common address space for a number of CPUs
  - access efficiency may vary
    - SMP, (cc)NUMA
  - many programming models (e.g. pthreads, OpenMP)
  - potentially easier to handle
    - req. hardware and OS support!



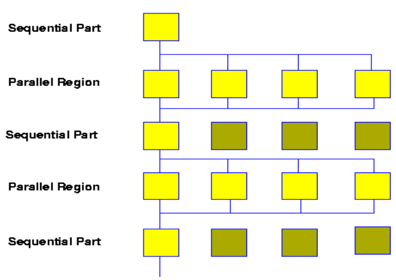
Shared memory parallelization with OpenMP

Partially based on LRZ/RRZE course on parallel programming. By courtesy of R. Bader.

- **based on compiler directives** (extending the C/Fortran standard)
- **requires shared memory (i.e. all threads can see the same data)**
- **(in principle) allows gradual transformation from serial to parallel**
- **Attention:**
  - memory bandwidth often does not scale with number of processors
  - watch *ccNUMA effects* and avoid *false sharing*!

further reading
http://openmp.org/wp/openmp-specifications/

OpenMP Execution Model

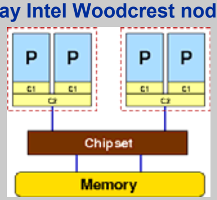


```
!$OMP PARALLEL DO DEFAULT(NONE)     &
!$OMP&SHARED(omega,f,obs,Nx,Ny,Nz)  &
!$OMP&PRIVATE(x,y,z,ux,uy,uz,dens)
do z=1,Nz ! outer loop parallelized
  do y=1,Ny; do x=1,Nx
    if( obs(x,y,z) ) then
      …
      relaxation (computations)
      …
    endif
  enddo; enddo;
enddo
```

Performance issues with OpenMP: ccNUMA and first-touch

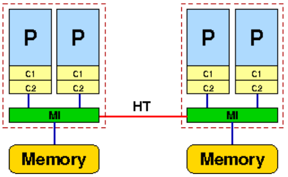
- **First-touch page allocation leads to network contention if initialization of data is done on a single CPU only.**
- **Only correct parallel initialization and block-static scheduling can achieve sufficient scalability.**

2-way Intel Woodcrest node



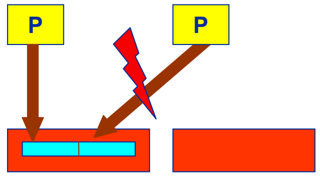
- **Uniform memory access through chipset.**
- **No placement issues,**
- **but limited bandwidth.**

2-way AMD Opteron node (similar on Intel Nehalem)

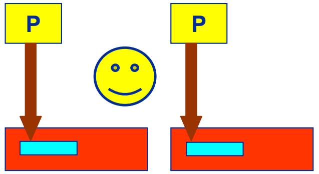


- **Cache coherent non-uniform memory access on-chip memory controller.**
- **Placement issues,**
- **better scalable bandwidth in multi-socket nodes.**

- **Data access on ccNUMA system without correct data placement:**

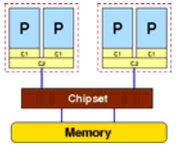


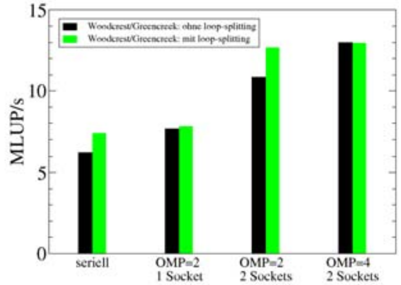
- **Data access on ccNUMA system with correct data placement:**



Scalability examples of OpenMP parallel LBM

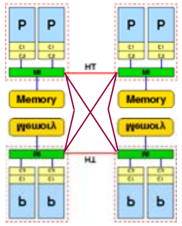
2-way Intel Core2 node

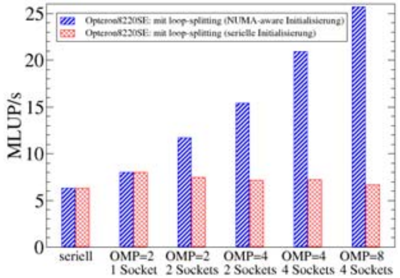




- **Highly optimized code version does not see any performance increase when using the second core of a socket.**

4-way AMD Opteron node





- **Significant performance increase only if proper data placement is ensured by NUM-aware initialization.**

Schematic NUMA-aware implementation

```
! allocate memory
allocate( f(Nx, Ny, Nz, 0:18, 0:1)

! ensure proper location of memory
!$OMP PARALLEL DO
!$OMP&SCHEDULE(STATIC)
do z=1,Nz
  do y=1,Ny
    do x=1,Nx
      f(x,y,z,:,:) = 0.d0
    enddo
  enddo
enddo
!$OMP END PARALLEL DO
...
```

!

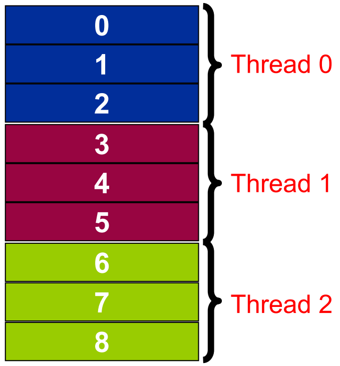
- What about calloc !?
- What about C++ new operator !?
- Linux filesystem buffer cache !?
- How to ensure that threads do not migrate between cores !?

```
...
! Iteration loop
do iterat=1, tEnd
   ...
   ! do relaxation and propagation
   !$OMP PARALLEL DO
   !$OMP&SHARED(Nx,Ny,Nz,omega,f,…)
   !$OMP&PRIVATE(x,y,z,ux,uy,uz,…)
   !$OMP&SCHEDULE(STATIC)
   do z=1,Nz
     do y=1,Ny; do x=1,Nx
          Some complex calculations
       enddo
     enddo
   enddo
   !$OMP END PARALLEL DO
   ...
enddo
...
```

Thread assignment: OpenMP vs. CUDA

OpenMP (on CPUs)

- divide domain into huge chunks
- avoid false sharing
- switching between threads rather expensive



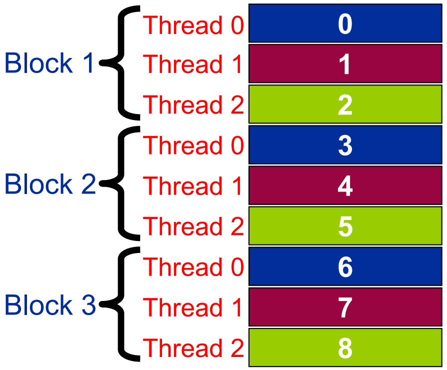
0 1 2 Thread 0

3 4 5 Thread 1

6 7 8 Thread 2

CUDA (on GPUs)

- ensure proper alignment
- divide domain into small pieces
- always many more threads in flight than GPU cores available to hide latency



Block 1: Thread 0 0, Thread 1 1, Thread 2 2

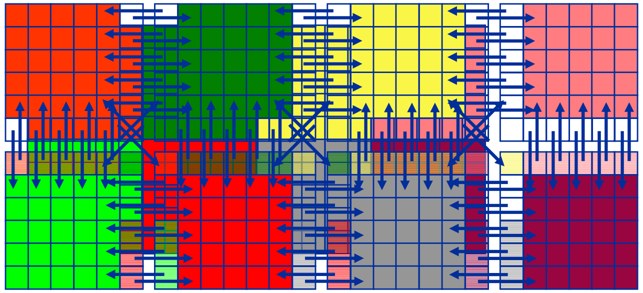
Block 2: Thread 0 3, Thread 1 4, Thread 2 5

Block 3: Thread 0 6, Thread 1 7, Thread 2 8

Parallelization for distributed memory systems

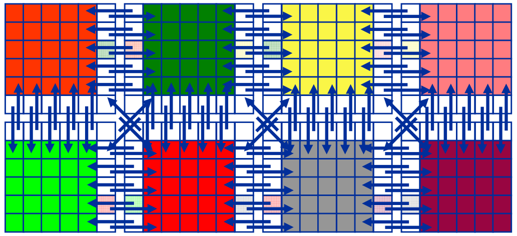
MPI parallelization

- domain decomposition
- ghost layers / halo cells
- explicit data exchange (sending/receiving of messages)
- can be used on any parallel computer (i.e. on shared and distributed memory systems)



Parallelization for distributed memory systems

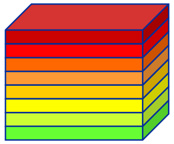
```
! initialize your parallel machine
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
 ...
call MPI_BCAST(var,cnt,type,root,comm,ierr)

do iterat=1, steps
  ! do relaxation&propagation
  ! exchange data between partitions
  call MPI_ISEND(buf,cnt,type,to,tag,com,req,ierr)
  call MPI_IRECV(buf,cnt,type,from,tag,com,req,ierr)
  call MPI_WAITALL(cnt,req,status,ierr)
enddo
call MPI_FINALIZE(ierr)
```

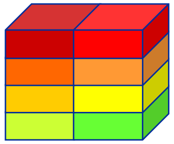


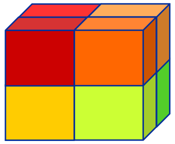
MPI distinguishes

- point-to-point and collective operations
- *point-to-point* involves exactly two partners; can be blocking or non-blocking (send & recv)
- *collective* operations always involve all partners and are blocking (e.g. bcast, reduction, alltoall, barrier, …)
- data types and operations must match

How to decompose a simple domain?







Partially based on LRZ/RRZE and HLRS courses. R. Bader & Rabenseifner.

Splitting in

- **one dimension:**
  - communication $= n^2*2*w*1$
- **two dimensions:**
  - communication $= n^2*2*w*2 / p^{1/2}$
- **three dimensions:**
  - communication $= n^2*2*w*3 / p^{2/3}$

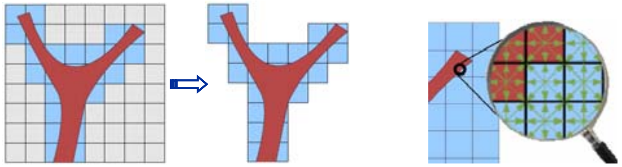
w = width of halo
n^3 = size of matrix
p = number of processors
cyclic boundary
→ two neighbors
in each direction

optimal for p>11

How to decompose a complex domain?

Patch approach

- Patches are only allocated for regions which contain fluid nodes.
- Multiple patches can be assigned to one processor.

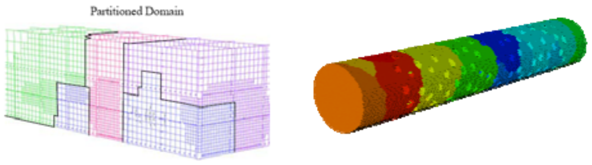


References: IWTM/FHG, LSS/Uni-Erlangen, iRMB/TU-Braunschweig …

Schematic sketches by courtesy of J. Götz, LSS, Uni-Erlangen

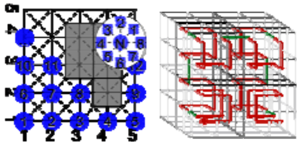
Graph-based distribution

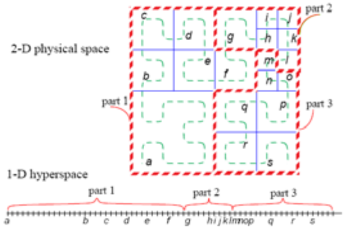
- e.g. using METIS on node or patch level



Cutting of a space filling curve

- e.g. in combination with a sparse approach using explicit adjacency information instead of index shift operations





Schematic sketches by ILBDC and Aftosmis, et al. AIAA2000-0808

Performance model for communication

Partially based on LRZ/RRZE and HLRS courses. R. Bader & Rabenseifner.

- **simplest model:**

  transfer time = latency + message length / bandwidth

- **latency: startup for message handling**
- **bandwidth: transfer of bytes**

- **n messages:**

  transfer time = n * latency + total message length / bandwidth

  → send one big message instead of several small messages!
  → reduce the total amount of bytes!
  → bandwidth depends on protocol

Load balance or minimal communication?

- **Let's assume**
  - 10^5 cells/processor **(40 MB only)**
  - 10 MLUPs/processor **(reasonably fast)**

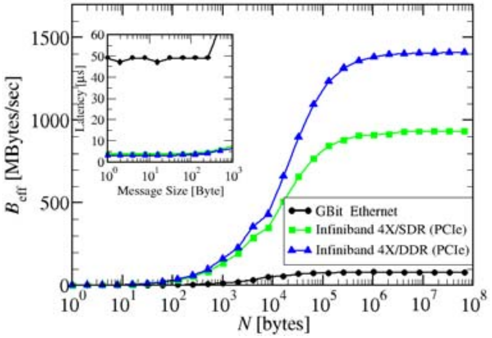
  →0.01s per timestep

  - 6 communication partners per partition each with 2500 cells (6x 200kB)

  →0.0002s/exchange for Infiniband

- **Communication makes up only a tiny fraction of the total time.**

→ optimize for load balance



Characteristics of interconnects according to Ping-Pong benchmark

Recomputation versus communication

- How to implement, if same data can be computed on several / all processes
  - parallel equivalent computation
  - single computation + broadcast
    while other processes can do other work
  - single computation + broadcast
    while other processes idle (worst solution!)

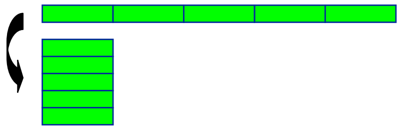
additional computation

communication & additional synchronization

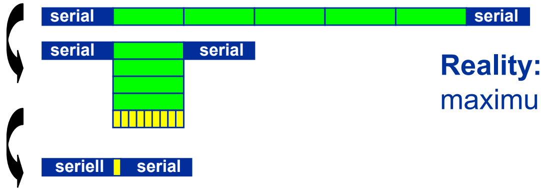
- Normally replicate and recompute the values
  - Consider how many calculations you can execute while only sending 1 Bit from one process to another
    (6 µs, 1.0 GFlop/s → 6000 operations)
  - Sending 16 kByte (20x20x5) doubles
    (with 300 MB/s bandwidth → 53.3 µs → 53,300 operations)
  - very often blocks have to wait for their neighbors
  - but extra work limits parallel efficiency

Partially based on LRZ/RRZE and HLRS courses. R. Bader & Rabenseifner.

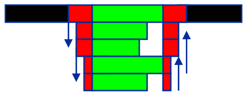
Limits of parallel scalability



Ideal world:
all work is perfectly parallelizable



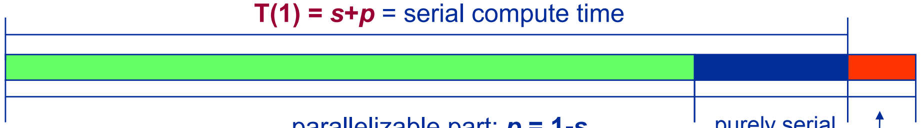
Reality: purely serial parts limit maximum speedup



Even worse: Communication processes hurt scalability even further

Calculating speedup in a simple model

T(1) = *s*+*p* = serial compute time

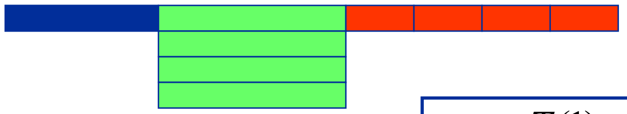


parallelizable part: *p* = 1-*s*

purely serial part *s*

fraction *k* for communication between each two workers

parallel: T(*N*) = *s*+*p*/*N*+*Nk*



General formula for speedup

$$S_p^k = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N} + Nk}$$

special case k=0:
Amdahl's Law
"strong scaling"

$$\lim_{N \to \infty} S_p^0(N) = \frac{1}{s}$$

for k=0

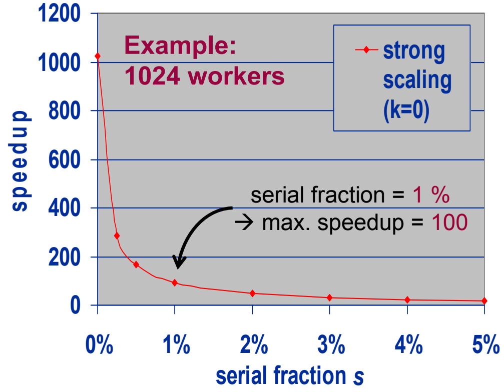
parallel efficiency

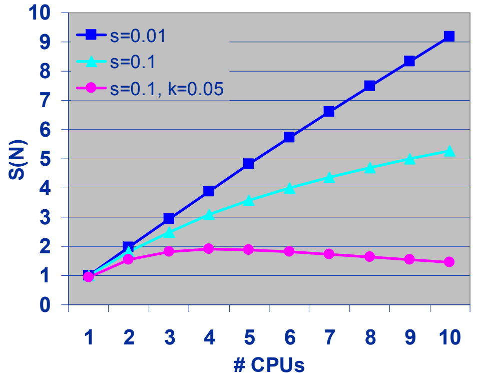
$$\varepsilon_p(N) = \frac{S_p^k(N)}{N}$$

Understanding parallelism: Amdahl´s Law

Amdahl's law poses a serious limit on the use of parallel resources!

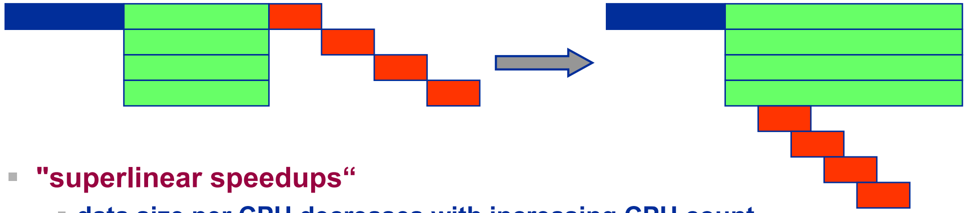
- in reality the situation gets even worse
  - load imbalance
  - communication overhead, task interdependencies
  - shared resources to be used sequentially (e.g. IO), etc.





Shifting the limits of scalability

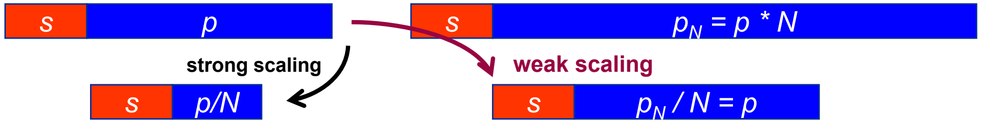
- **Communication is not necessarily purely serial**
  - non-blocking networks can transfer many messages concurrently – factor *Nk* in denominator becomes *k* (technical measure)
  - sometimes, communication can be overlapped with useful work (depends on implementation, algorithm):



- **"superlinear speedups"**
  - **data size per CPU decreases with increasing CPU count → working set may fit into cache at large CPU counts**
  - **communication must be negligible to see this effect**
- **the larger the problem, the larger is often the parallel fraction**
  - i.e. the sequential part *s* gets smaller and the Amdahl limit is shifted
  - communication overhead may scale with a smaller power than problem size

Increasing parallel efficiency ("weak scaling")

- **When scaling a problem to more workers, the "amount of work" (i.e. usually the domain size) will often be scaled as well**
  - perfect situation: runtime stays constant while N increases
  - often *p* scales with problem size while *s* stays constant



"Performance Scale-up" = $\frac{\text{work/time for problem size N with N workers}}{\text{work/time for problem size 1 with 1 worker}}$

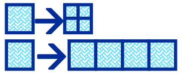
Gustafsson's Law ("weak scaling")

$$P_s(N) = \frac{s+pN}{s+p} = s+pN = N+(1-N)s = s+(1-s)N$$

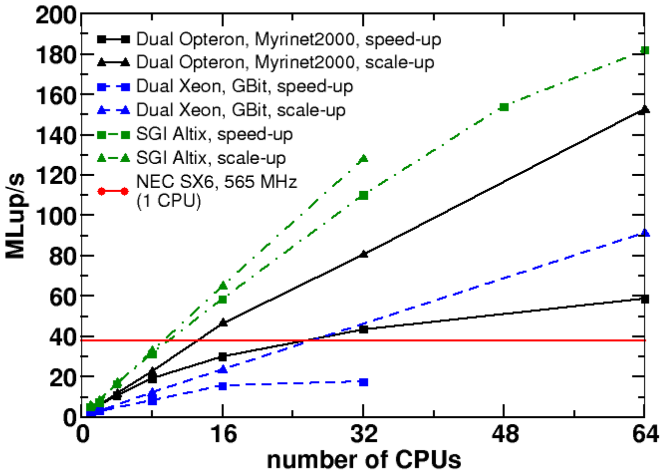
- linear in N – but closely observe the meaning of the word "work"!
- number of timesteps required may increase if domain size is increased! → runtime per iteration is constant but time-to-solution increases ☹

Some final remarks on parallelization (I)

- **strong scaling (speed-up)**
- **weak scaling (scale-up)**

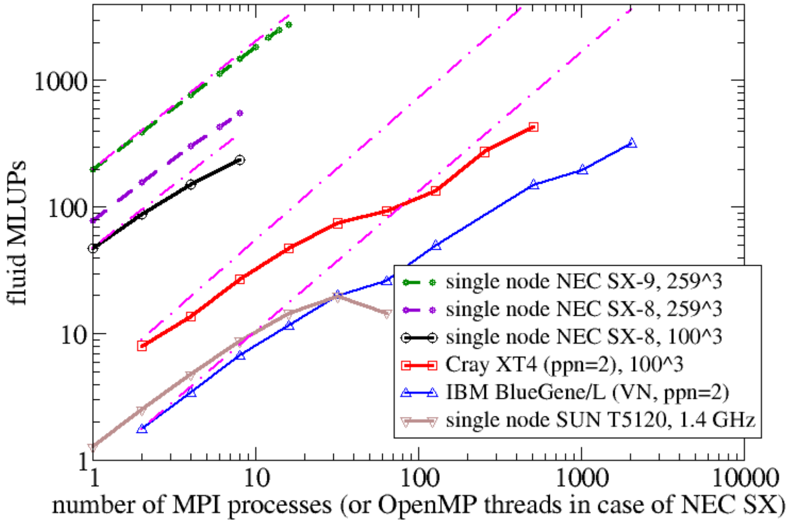


Domain size 256x129x128 for speed-up; 128^3 per processor for scale up



- **The benchmarked systems are outdated, but the message is still true:**
- **cheap GBit-ethernet is enough for weak scaling;**
- **strong scaling is the only true challenge.**

Some final remarks on parallelization (II)



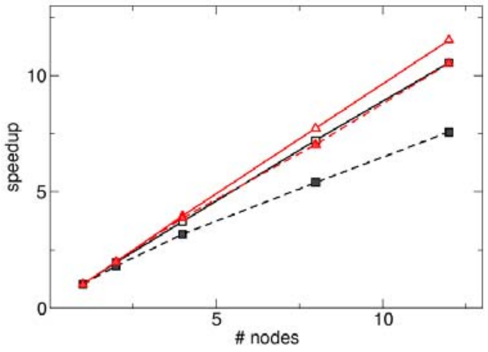
Single core performance still matters

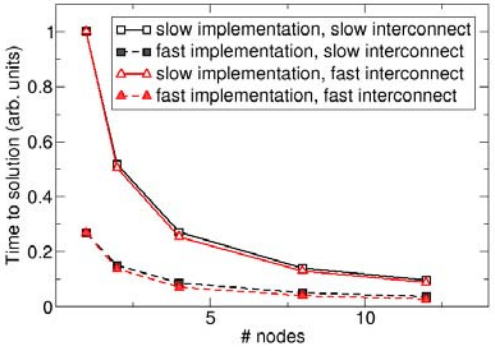
…

- **pretty linear scaling for all systems !?**
- **yes, but with a slope <1 !**

Some final remarks on parallelization (III)

- Fooling the masses by choosing the right metric ☺





- Looks like the "red triangles" performance best !?
  At least in terms of speedup …

- … but not in terms of time-to-solution.
- A slow code scales better – but as an engineer you are interested in quick simulation results and not in speedup.
- Thus, do not (only) publish speedups.

Tools

- make
- version control systems
- MPI tracing

Automatic recompilation of changed files: make

- **Rules** say when and how to remake targets, which depend on certain prerequisites, using given shell commands.
  - **explicit and/or implicit rules**
- **Variable** definitions and expansions in a makefile work similar to shell variables.
- **Directives** tell make to do something special while reading a makefile, like including other makefiles, deciding whether to use or ignore some part of the makefile, . . .
- **Comments** are started with a # character. The # and the rest of the line are ignored. If you want a literal #, escape it with a backslash: \#.

- Most important automatic variables
  - $@ name of the target, which caused the rule to be processed.
  - $< name of the first prerequisite.
  - $ˆ names of all prerequisites separated by spaces.
  - $? space-separated names list of all prerequisites newer than the target

Example of a makefile

```
CC = icc
FC = ifort
LD = ${FC}
CFLAGS = -O3
FFLAGS = -g
ifeq (${LD}, icc)
  LIBS += -L${IFORT_BASE}/lib -lifcore
endif

c_objects =
f_objects = lbmain.f90 geometry.f90 collprop.f90

# the rules
.PHONY: clean
lbmSolver: ${c_objects} ${f_objects}
<TAB>   ${LD} ${LDFLAGS} -o $@ $ˆ ${LIBS}
${f_objects}: %.o: %.f90
<TAB>   ${FC} -c ${FFLAGS} $<
clean:
<TAB>   -rm -f lbmSolver *.o *.mod

# explicit dependencies
lbmain.o: geometry.mod collprop.mod
```

define the compilers and their flags.

conditionally append additional settings

special directive

explicit rule

rule for a class of files

ignore errors for rm

Version control systems

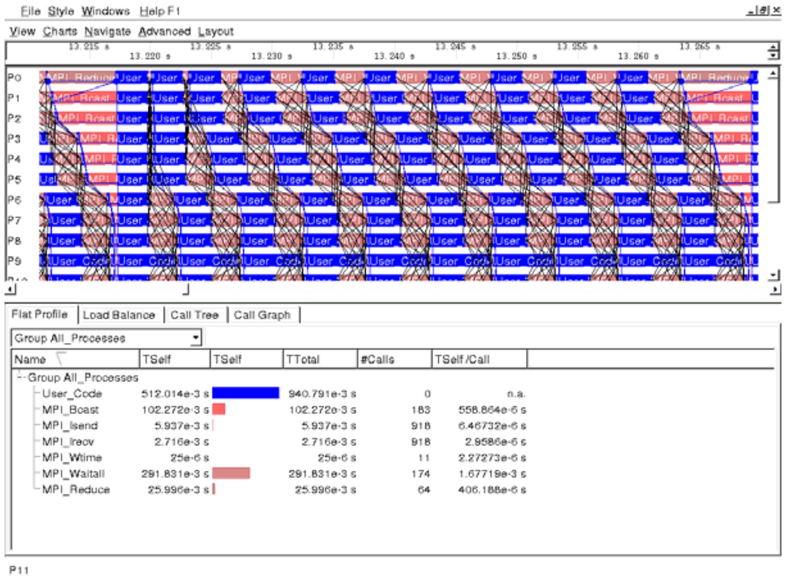
- RCS – Revision Control System
  - a simple dinosaurian; suitable for keeping track of OS configuration files but not for programming projects; history kept in a subdirectory
- **CVS – Current Version System**
  - extensive free documentation: http://cvsbook.red-bean.com/
  - MS Windows GUI: http://www.tortoisecvs.org/
- **SVN – Subversion**
  - extensive free documentation: http://svnbook.red-bean.com/
  - MS Windows GUI: http://tortoisesvn.tigris.org/

- **central repository**
- **basic support for branching and merging**

- **Distributed VCS**
  - git http://git-scm.com/
  - mercurial/Hg http://www.selenic.com/mercurial/
  - bazaar http://bazaar-vcs.org/
  - …

- **each copy is a full independent repository**
- **full offline operation**
- **advanced features for merging, branching, bisection, …**

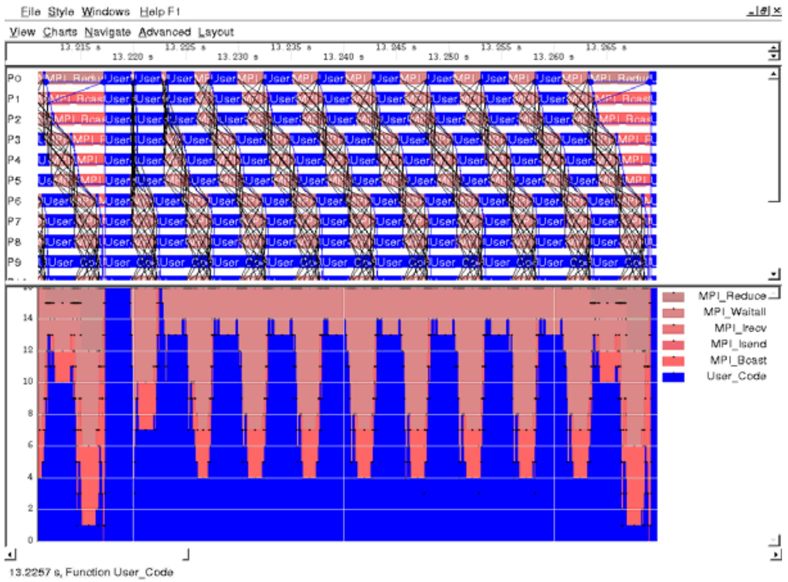
Intel Trace Collector/Analyzer (I)



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
User_Code	512.014e-3 s		940.791e-3 s	0	n.a.
MPI_Bcast	102.272e-3 s		102.272e-3 s	183	558.864e-6 s
MPI_Isend	5.937e-3 s		5.937e-3 s	918	6.46732e-6 s
MPI_Irecv	2.716e-3 s		2.716e-3 s	918	2.9586e-6 s
MPI_Wtime	25e-6 s		25e-6 s	11	2.27273e-6 s
MPI_Waitall	291.831e-3 s		291.831e-3 s	174	1.67719e-3 s
MPI_Reduce	25.996e-3 s		25.996e-3 s	64	406.188e-6 s

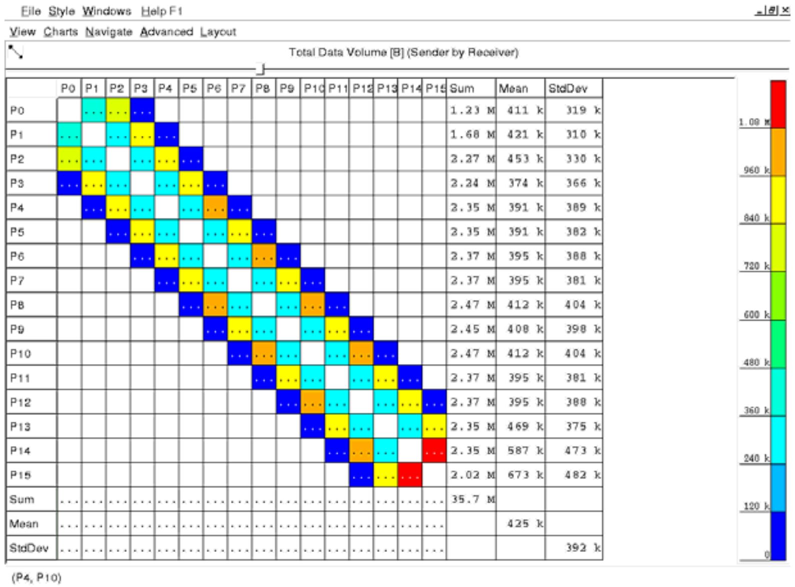
P11

Intel Trace Collector/Analyzer (II)



13.2257 s, Function User_Code

Intel Trace Collector/Analyzer (III)

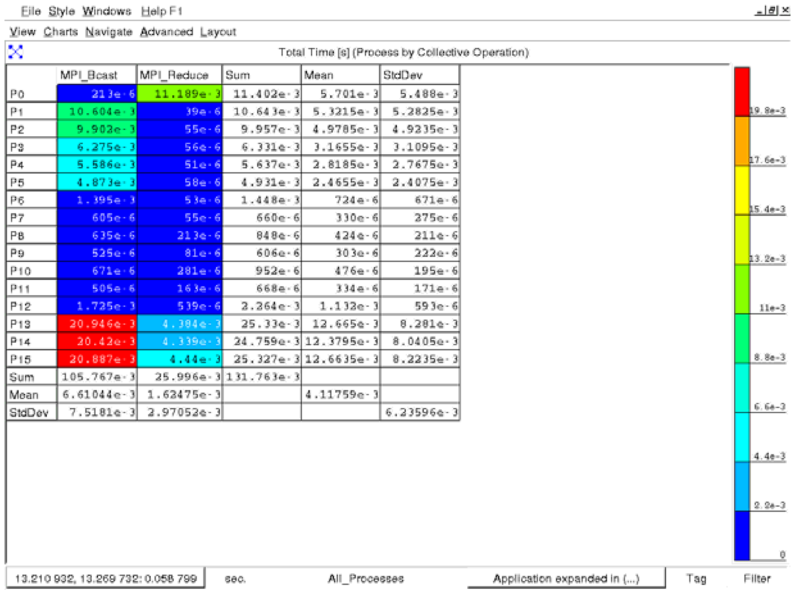

File Style Windows Help F1
View Charts Navigate Advanced Layout
1.08 M
960 k
840 k
720 k
600 k
480 k
360 k
240 k
120 k
0

Total Data Volume [B] (Sender by Receiver)

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Sum	Mean	StdDev
P0		...	...	...													1.23 M	411 k	319 k
P1	...		...	...	...												1.68 M	421 k	310 k
P2	...	...		...	...	...											2.27 M	453 k	330 k
P3	...	...	...		...	...	...										2.24 M	374 k	366 k
P4		...	...	...		...	...	...									2.35 M	391 k	389 k
P5			...	...	...		...	...	...								2.35 M	391 k	382 k
P6				...	...	...		...	...	...							2.37 M	395 k	388 k
P7					...	...	...		...	...	...						2.37 M	395 k	381 k
P8						...	...	...		...	...	...					2.47 M	412 k	404 k
P9							...	...	...		...	...	...				2.45 M	408 k	398 k
P10								...	...	...		...	...	...			2.47 M	412 k	404 k
P11									...	...	...		...	...	...		2.37 M	395 k	381 k
P12										...	...	...		...	...	...	2.37 M	395 k	388 k
P13											...	...	...		...	...	2.35 M	469 k	375 k
P14												...	...	...		...	2.35 M	587 k	473 k
P15													...	...	...		2.02 M	673 k	482 k
Sum	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	35.7 M		
Mean	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...		425 k	
StdDev	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...			392 k

(P4, P10)

Intel Trace Collector/Analyzer (IV)

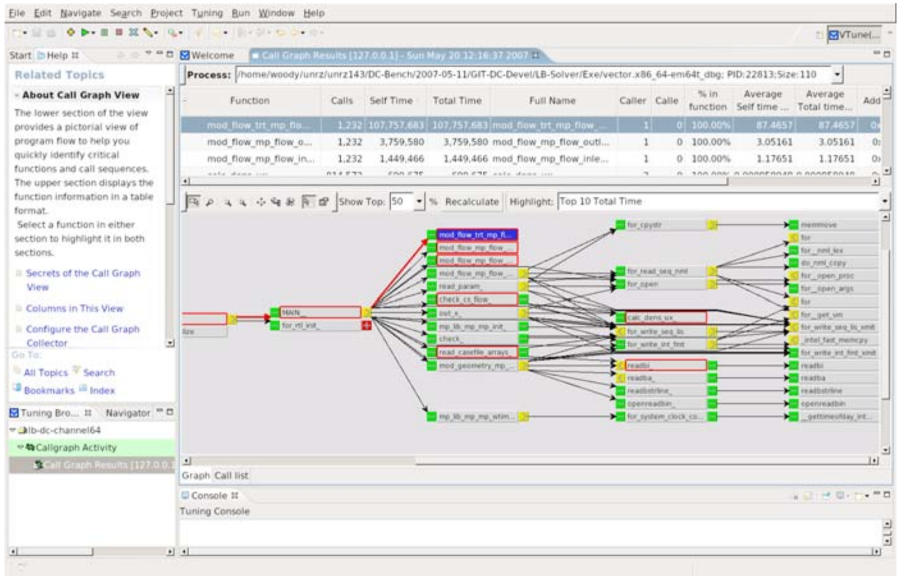

File Style Windows Help F1
View Charts Navigate Advanced Layout
19.8e-3
17.6e-3
15.4e-3
13.2e-3
11e-3
8.8e-3
6.6e-3
4.4e-3
2.2e-3
0

Total Time [s] (Process by Collective Operation)

	MPI_Bcast	MPI_Reduce	Sum	Mean	StdDev
P0	213e-6	11.189e-3	11.402e-3	5.701e-3	5.488e-3
P1	10.604e-3	39e-6	10.643e-3	5.3215e-3	5.2825e-3
P2	9.902e-3	55e-6	9.957e-3	4.9785e-3	4.9235e-3
P3	6.275e-3	56e-6	6.331e-3	3.1655e-3	3.1095e-3
P4	5.586e-3	51e-6	5.637e-3	2.8185e-3	2.7675e-3
P5	4.873e-3	58e-6	4.931e-3	2.4655e-3	2.4075e-3
P6	1.395e-3	53e-6	1.448e-3	724e-6	671e-6
P7	605e-6	55e-6	660e-6	330e-6	275e-6
P8	635e-6	213e-6	848e-6	424e-6	211e-6
P9	525e-6	81e-6	606e-6	303e-6	222e-6
P10	671e-6	281e-6	952e-6	476e-6	195e-6
P11	505e-6	163e-6	668e-6	334e-6	171e-6
P12	1.725e-3	539e-6	2.264e-3	1.132e-3	593e-6
P13	20.946e-3	4.384e-3	25.33e-3	12.665e-3	8.281e-3
P14	20.42e-3	4.339e-3	24.759e-3	12.3795e-3	8.0405e-3
P15	20.887e-3	4.44e-3	25.327e-3	12.6635e-3	8.2235e-3
Sum	105.767e-3	25.996e-3	131.763e-3		
Mean	6.61044e-3	1.62475e-3		4.11759e-3	
StdDev	7.5181e-3	2.97052e-3			6.23596e-3

13.210 932, 13.269 732: 0.058 799 | sec. | All_Processes | Application expanded in (...) | Tag | Filter

Intel VTune (I)



Related Topics

About Call Graph View

The lower section of the view provides a pictorial view of program flow to help you quickly identify critical functions and call sequences. The upper section displays the function information in a table format.

Select a function in either section to highlight it in both sections.

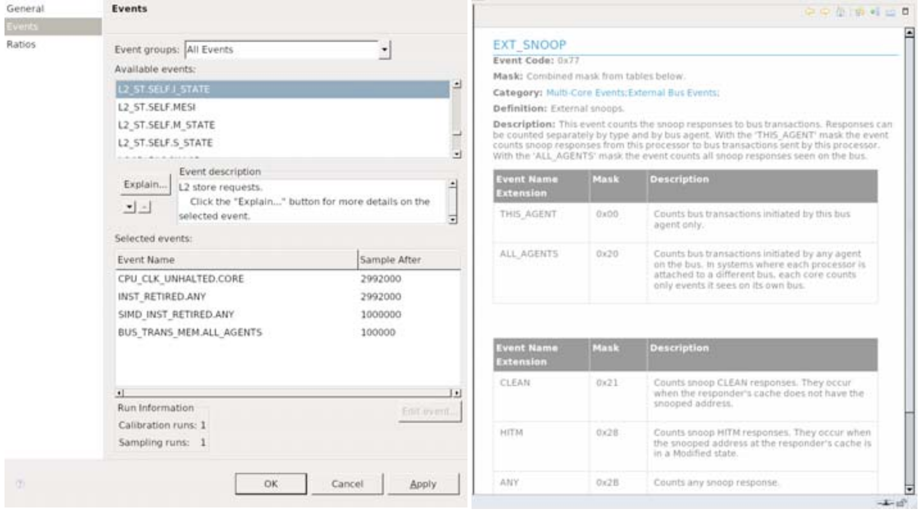
- Secrets of the Call Graph View
- Columns in This View
- Configure the Call Graph Collector

Go To: All Topics Search Bookmarks Index

Process: /home/woody/unrz/unrz143/DC-Bench/2007-05-11/GIT-DC-Devel/LB-Solver/Exe/vector.x86_64-em64t_dbg: PID:22813:Size:110

Function	Calls	Self Time	Total Time	Full Name	Caller	Calle	% in function	Average Self time ...	Average Total time...	Add
mod_flow_trt_mp_flo...	1,232	107,757,683	107,757,683	mod_flow_trt_mp_flow_...	1	0	100.00%	87.4657	87.4657	0
mod_flow_mp_flow_o...	1,232	3,759,580	3,759,580	mod_flow_mp_flow_outl...	1	0	100.00%	3.05161	3.05161	0
mod_flow_mp_flow_in...	1,232	1,449,466	1,449,466	mod_flow_mp_flow_inle...	1	0	100.00%	1.17651	1.17651	0

Intel VTune (II)



Events

Event groups: All Events

Available events:

- L2_ST.SELF.I_STATE
- L2_ST.SELF.MESI
- L2_ST.SELF.M_STATE
- L2_ST.SELF.S_STATE

Event description

L2 store requests.

Click the "Explain..." button for more details on the selected event.

Selected events:

Event Name	Sample After
CPU_CLK_UNHALTED.CORE	2992000
INST_RETIRED.ANY	2992000
SIMD_INST_RETIRED.ANY	1000000
BUS_TRANS_MEM.ALL_AGENTS	100000

Run Information

Calibration runs: 1

Sampling runs: 1

EXT_SNOOP

Event Code: 0x77

Mask: Combined mask from tables below.

Category: Multi-Core Events;External Bus Events;

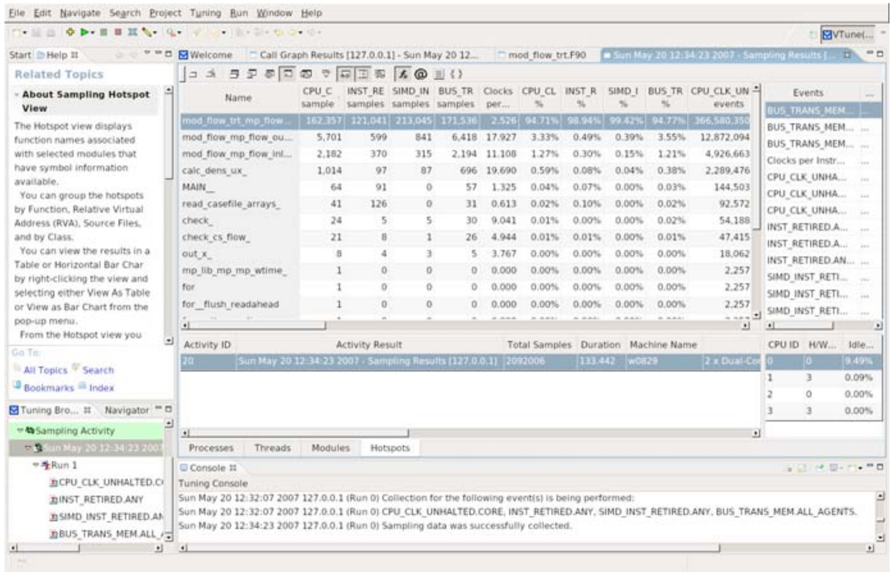
Definition: External snoops.

Description: This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. With the 'THIS_AGENT' mask the event counts snoop responses from this processor to bus transactions sent by this processor. With the 'ALL_AGENTS' mask the event counts all snoop responses seen on the bus.

Event Name Extension	Mask	Description
THIS_AGENT	0x00	Counts bus transactions initiated by this bus agent only.
ALL_AGENTS	0x20	Counts bus transactions initiated by any agent on the bus. In systems where each processor is attached to a different bus, each core counts only events it sees on its own bus.

Event Name Extension	Mask	Description
CLEAN	0x21	Counts snoop CLEAN responses. They occur when the responder's cache does not have the snooped address.
HITM	0x28	Counts snoop HITM responses. They occur when the snooped address at the responder's cache is in a Modified state.
ANY	0x2B	Counts any snoop response.

Intel VTune (III)



Related Topics

About Sampling Hotspot View

The Hotspot view displays function names associated with selected modules that have symbol information available.

You can group the hotspots by Function, Relative Virtual Address (RVA), Source Files, and by Class.

You can view the results in a Table or Horizontal Bar Char by right-clicking the view and selecting either View As Table or View as Bar Chart from the pop-up menu.

From the Hotspot view you

Go To: All Topics Search Bookmarks Index

Name	CPU_C sample	INST_RE samples	SIMD_IN samples	BUS_TR samples	Clocks per...	CPU_CL %	INST_R %	SIMD_I %	BUS_TR %	CPU_CLK_UN events
mod_flow_trt_mp_flow...	162,357	121,041	213,045	171,536	2.526	94.71%	98.94%	99.42%	94.77%	366,580,350
mod_flow_mp_flow_ou...	5,701	599	841	6,418	17.927	3.33%	0.49%	0.39%	3.55%	12,872,094
mod_flow_mp_flow_inl...	2,182	370	315	2,194	11.108	1.27%	0.30%	0.15%	1.21%	4,926,663
calc_dens_ux_	1,014	97	87	696	19.690	0.59%	0.08%	0.04%	0.38%	2,289,476
MAIN__	64	91	0	57	1.325	0.04%	0.07%	0.00%	0.03%	144,503
read_casefile_arrays_	41	126	0	31	0.613	0.02%	0.10%	0.00%	0.02%	92,572
check_	24	5	5	30	9.041	0.01%	0.00%	0.00%	0.02%	54,188
check_cs_flow_	21	8	1	26	4.944	0.01%	0.01%	0.00%	0.01%	47,415
out_x_	8	4	3	5	3.767	0.00%	0.00%	0.00%	0.00%	18,062
mp_lib_mp_mp_wtime_	1	0	0	0	0.000	0.00%	0.00%	0.00%	0.00%	2,257
for	1	0	0	0	0.000	0.00%	0.00%	0.00%	0.00%	2,257
for__flush_readahead	1	0	0	0	0.000	0.00%	0.00%	0.00%	0.00%	2,257

Events	...
BUS_TRANS_MEM...	...
BUS_TRANS_MEM...	...
BUS_TRANS_MEM...	...
Clocks per Instr...	...
CPU_CLK_UNHA...	...
CPU_CLK_UNHA...	...
CPU_CLK_UNHA...	...
INST_RETIRED.A...	...
INST_RETIRED.A...	...
INST_RETIRED.AN...	...
SIMD_INST_RETI...	...
SIMD_INST_RETI...	...
SIMD_INST_RETI...	...

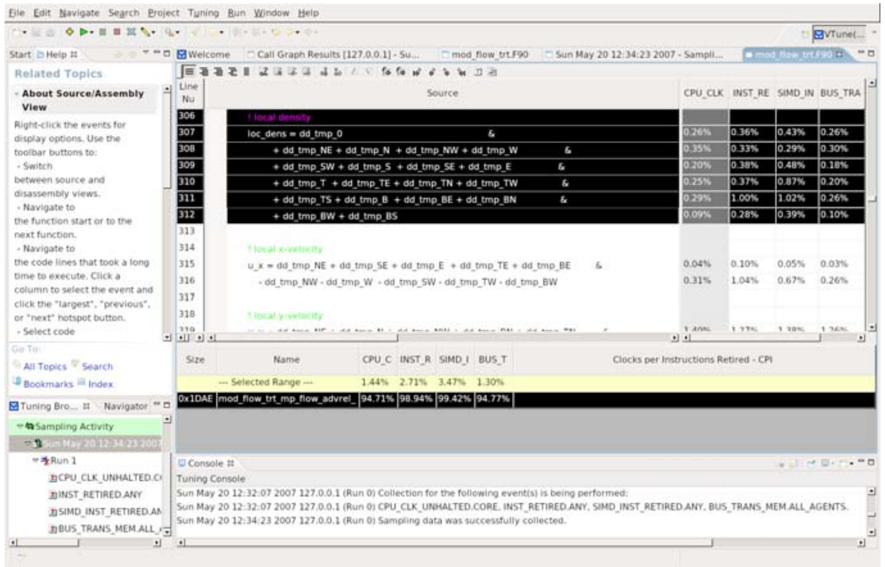
Activity ID	Activity Result	Total Samples	Duration	Machine Name	
20	Sun May 20 12:34:23 2007 - Sampling Results [127.0.0.1]	2092006	133.442	w0829	2 x Dual-Co

CPU ID	H/W...	Idle...
0	0	9.49%
1	3	0.09%
2	0	0.00%
3	3	0.00%

Tuning Console

```
Sun May 20 12:32:07 2007 127.0.0.1 (Run 0) Collection for the following event(s) is being performed:
Sun May 20 12:32:07 2007 127.0.0.1 (Run 0) CPU_CLK_UNHALTED.CORE, INST_RETIRED.ANY, SIMD_INST_RETIRED.ANY, BUS_TRANS_MEM.ALL_AGENTS.
Sun May 20 12:34:23 2007 127.0.0.1 (Run 0) Sampling data was successfully collected.
```

Intel VTune (IV)



About Source/Assembly View

Right-click the events for display options. Use the toolbar buttons to:
- Switch between source and disassembly views.
- Navigate to the function start or to the next function.
- Navigate to the code lines that took a long time to execute. Click a column to select the event and click the "largest", "previous", or "next" hotspot button.
- Select code

Source	CPU_CLK	INST_RE	SIMD_IN	BUS_TRA
! local density				
loc_dens = dd_tmp_0 &	0.26%	0.36%	0.43%	0.26%
+ dd_tmp_NE + dd_tmp_N + dd_tmp_NW + dd_tmp_W &	0.35%	0.33%	0.29%	0.30%
+ dd_tmp_SW + dd_tmp_S + dd_tmp_SE + dd_tmp_E &	0.20%	0.38%	0.48%	0.18%
+ dd_tmp_T + dd_tmp_TE + dd_tmp_TN + dd_tmp_TW &	0.25%	0.37%	0.87%	0.20%
+ dd_tmp_TS + dd_tmp_B + dd_tmp_BE + dd_tmp_BN &	0.29%	1.00%	1.02%	0.26%
+ dd_tmp_BW + dd_tmp_BS	0.09%	0.28%	0.39%	0.10%
! local x-velocity				
u_x = dd_tmp_NE + dd_tmp_SE + dd_tmp_E + dd_tmp_TE + dd_tmp_BE &	0.04%	0.10%	0.05%	0.03%
- dd_tmp_NW - dd_tmp_W - dd_tmp_SW - dd_tmp_TW - dd_tmp_BW	0.31%	1.04%	0.67%	0.26%
! local y-velocity				

Size	Name	CPU_C	INST_R	SIMD_I	BUS_T	Clocks per Instructions Retired - CPI
	--- Selected Range ---	1.44%	2.71%	3.47%	1.30%	
0x1DAE	mod_flow_trt_mp_flow_advrel_	94.71%	98.94%	99.42%	94.77%	

```
Sun May 20 12:32:07 2007 127.0.0.1 (Run 0) Collection for the following event(s) is being performed:
Sun May 20 12:32:07 2007 127.0.0.1 (Run 0) CPU_CLK_UNHALTED.CORE, INST_RETIRED.ANY, SIMD_INST_RETIRED.ANY, BUS_TRANS_MEM.ALL_AGENTS.
Sun May 20 12:34:23 2007 127.0.0.1 (Run 0) Sampling data was successfully collected.
```

Quick analysis of a user code (I)

- **Compiled using `-O3 -fomit-frame-pointer -g -p -xW`**
- **Set `GMON_OUT_PREFIX` and run with the provided input**
  - → display gathered timing data with `gprof`
  - → “stream” requires significant time
- **look at source code**
  - → separate routines for *collision*, *propagation* and periodic boundary conditions; *no toggle arrays*
  - → “collision optimized” data layout
- **Let’s flip the array of the distribution function: f(i,x,y,z) ➔ f(x,y,z,i)**
  - → used `cpp` for most of the work; had to do small changes in the MPI communication manually (i.e. one send/recv pair per direction instead of just one big pair)
- **Again a run with the provided input**
  - → significant speedup (see next slide); results still identical
  - → invested time for analysis + optimization: <15 min

Quick analysis of a user code (II)

`gprof` profile for 1 of 8 MPI processes **routine**	**calls**	**original [s]**	**optimized [s]**	**ratio**
d3q19	1	1447	849	0,59
stream	1117	480	134	0,28
mcollid	3001	428	430	1,00
collid	1117	160	198	1,24
lbm	1	147	144	0,98
analyze	558	146	146	1,00
onedimen	1674	22	21	0,95
bcperio	1	13	21	1,62
…				
total		2873	1927	0,67

total elapsed time		**10:16 min**	**4:12 min**	**0,41**

The END

These slides (including updates if necessary) are also available online at:

`http://www.konwihr.uni-erlangen.de/projekte/workshop-lattice-boltzmann-methods/`