Efficient implementation of simple lattice Boltzmann kernels

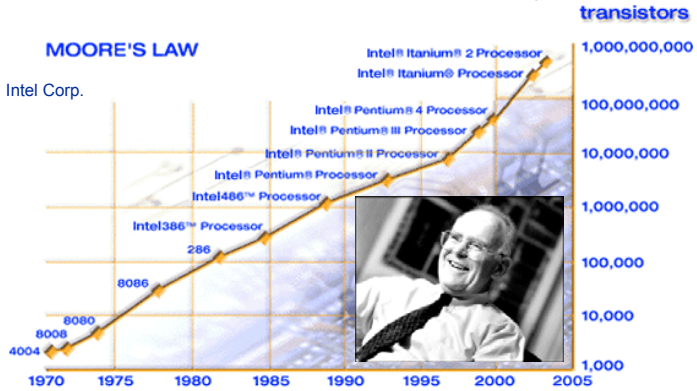
G. Wellein, T. Zeiser, G. Hager, S. Donath
HPC Services
Regionales Rechenzentrum Erlangen
Friedrich-Alexander-University Erlangen-Nuremberg
Germany

International Conference on
Parallel Computational Fluid Dynamics 2007 – Short Course

Survey

Introduction
Ever growing processor speed & Moore´s Law

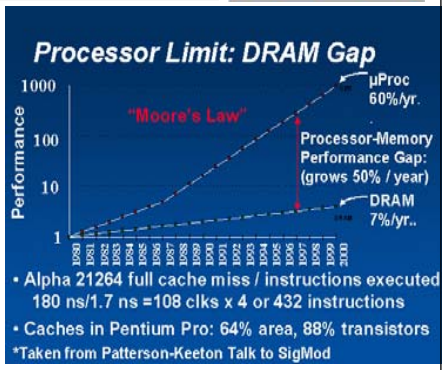
- **1965 G. Moore (co-founder of Intel) claimed**
  #transistors on processor chip doubles every 12-24 months



- **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?**

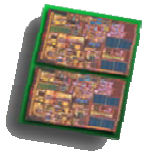
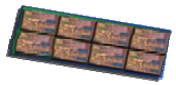
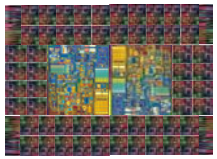
Introduction
Why worry about performance: Memory gap

- **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

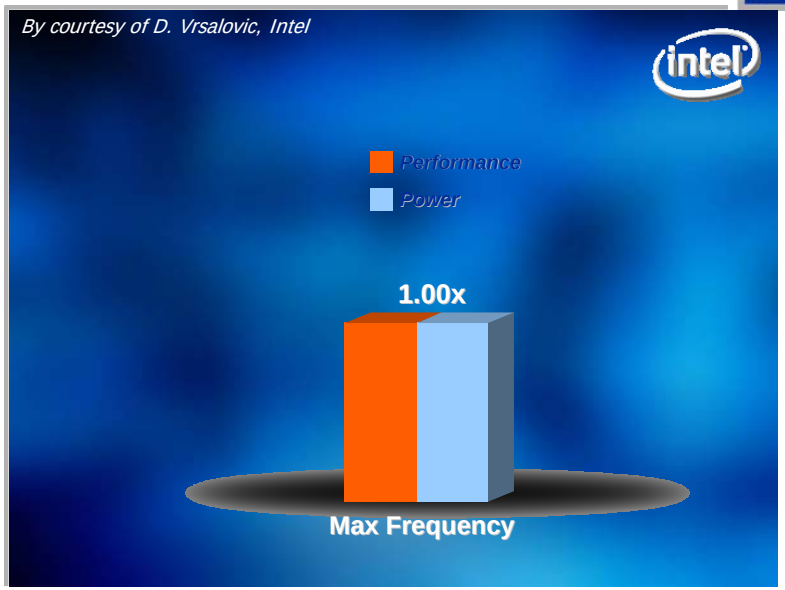


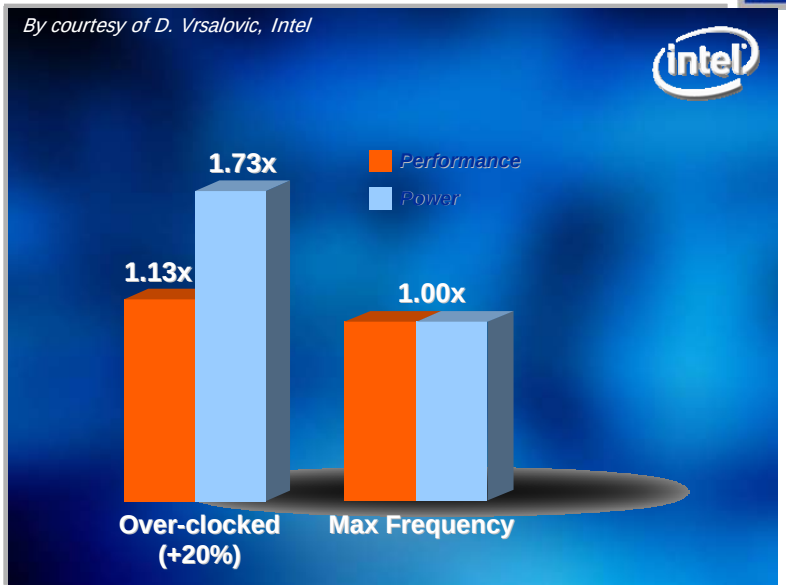
Optimization of main memory access is mandatory for most applications

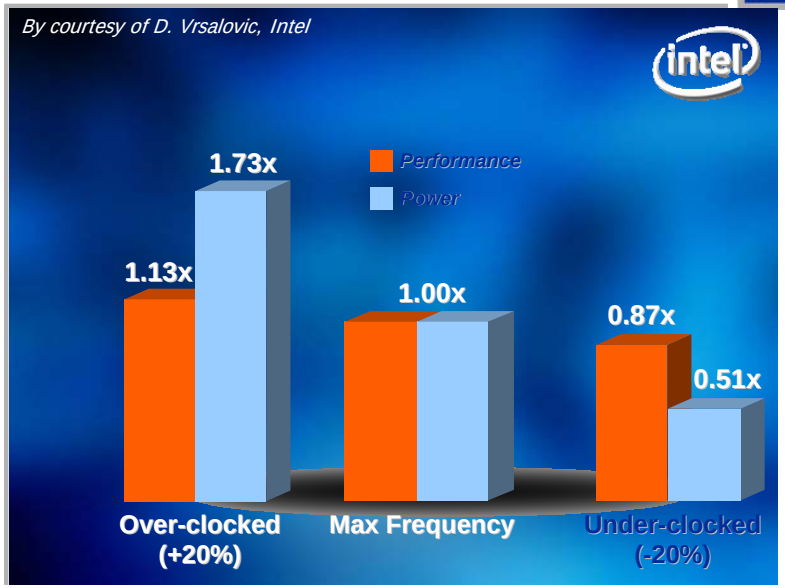
- **Multi-Core Processors – The game is over…**
  - Problem: Moore's law is still valid but increasing clock speed hits a technical wall (heat)
  - Solution: Reduce clock speed of processor but put 2 (or more) processors (cores) on a single silicon die
  - We will have to use many less powerful processors in the future

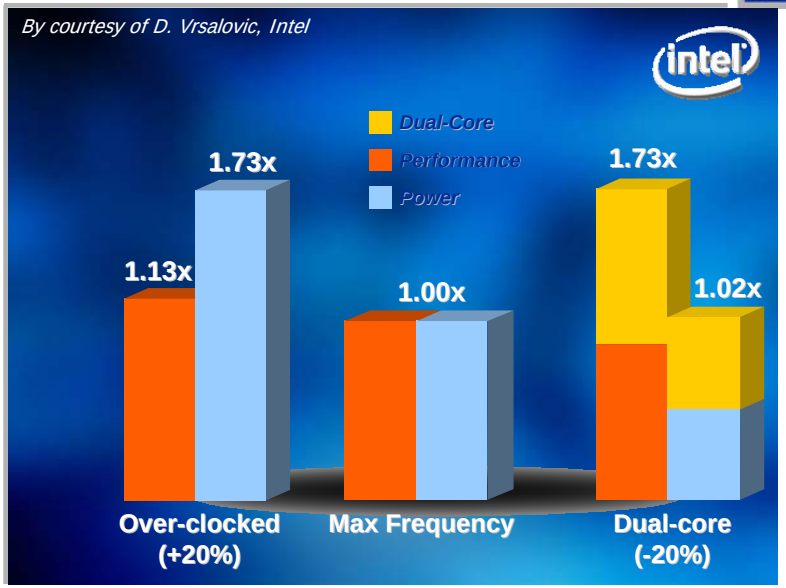
Parallelization will be mandatory for most applications in the future







Why worry about performance: Why Multi-Core?



Programming basics

- **FORTRAN or C is recommended**
- **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

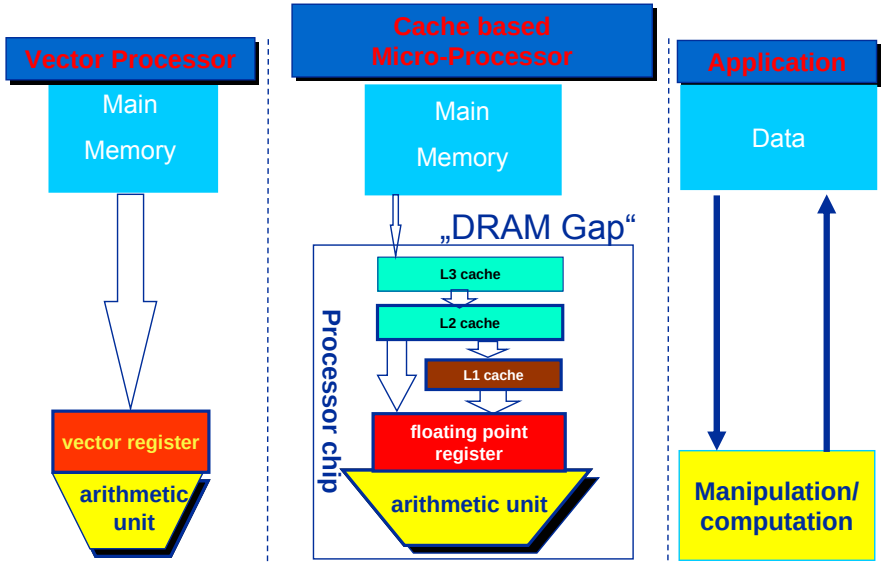
```
real*8 a(0:3,0:3)  ! FORTRAN Version
```

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!

- **Use appropriate compiler flags to ensure optimization**, e.g. for Intel compiler: `-O3 -xW -fno-alias` (if no pointer aliasing is used)

Memory hierarchies of modern processors

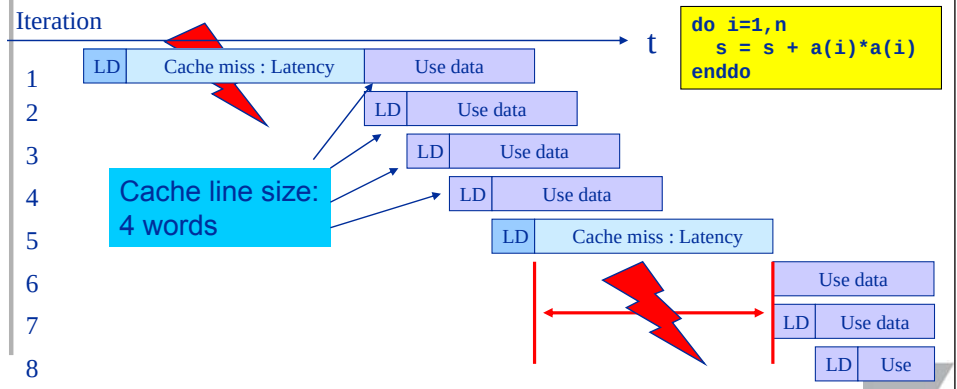


Memory hierarchies
Cache-based dual-core architectures

*SSE2		Intel Xeon5100	AMD Opteron	Intel Itanium2
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
#Registers		**8 / 16***	**16 / 32***	**128**
L1	**Size**	32 kB	64 kB	*16 KB*
	BW	96 GB/s	41.6 GB/s	*51.2 GB/s*
	Latency	2 cycles	3 cycles	*1 cycle*
L2	**Size**	**4 MB (2 cores)**	**1 MB**	256 KB
	BW	**96 GB/s**	**41.6 GB/s**	51.2 GB/s
	Latency	**7 cycles**	**~13 cycles**	5-6 cycles
L3	**Size**			**6 / 12 MB**
	BW			**51.2 GB/s**
	Latency			**12-13 cycles**
Memory	**BW**	**10.6 GB/s**	**6.4 GB/s**	**8.5 GB/s**
	Latency	**~200 ns**	**< 100 ns**	**~200 ns**

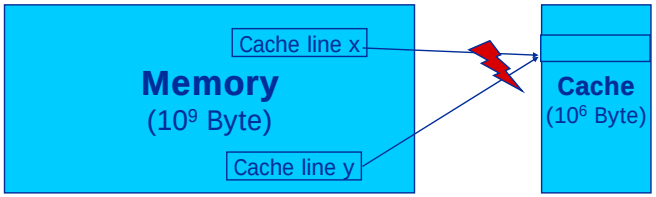
Memory hierarchies
Cache structure

- Caches are organized in cache lines that are fetched/stored as a whole (e.g. 128 bytes=16 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



Memory hierarchies:
Cache structure

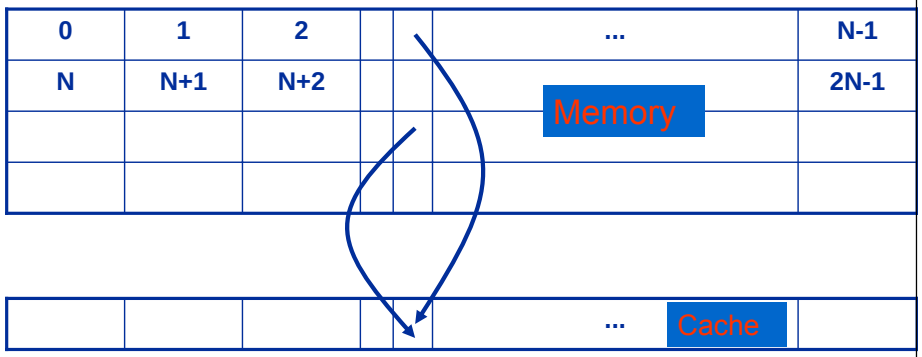
- Cache line data is always consecutive
  - Cache use is optimal for contiguous access (stride 1)
  - Non-consecutive access reduces performance (worst case: ≥ cache line size)
  - **Ensure spatial locality by blocking or optimizing data layout**
- Caches (~MB) must be mapped to memory locations (~GB)
  - **Cache multi-associativity enhances utilization**



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 can be stored at 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

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

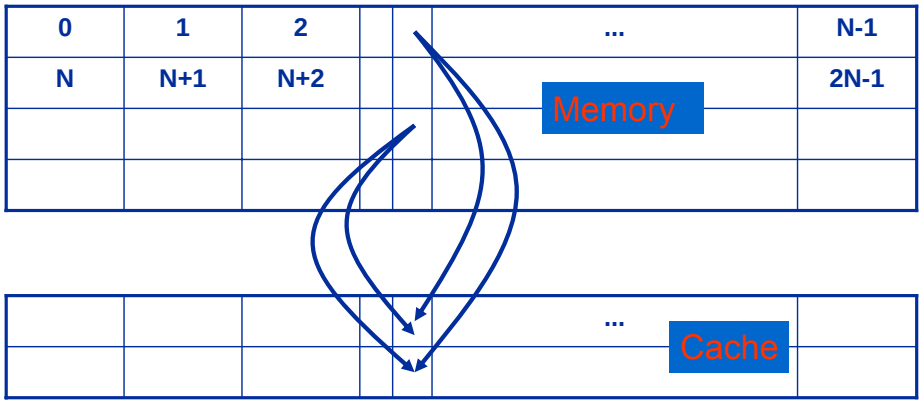
Memory hierarchies
Cache mapping – Associative caches

- *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+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

  - If all m locations are taken, one has to be overwritten on next cache load using different strategies:
    - least recently used (LRU)
    - Random
    - not recently used (NRU)

Memory hierarchies
Cache mapping – Associative caches



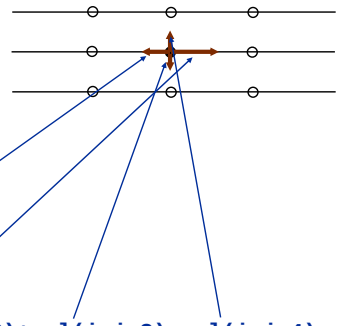
Example: 2-way associative cache. Each memory location can be mapped to two cache locations:
E.g. Size of main memory= 1 GByte; Cache Size= 256 KB -> 8192 memory locations are mapped to two cache locations

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!**

Memory hierarchies
Cache thrashing - Example

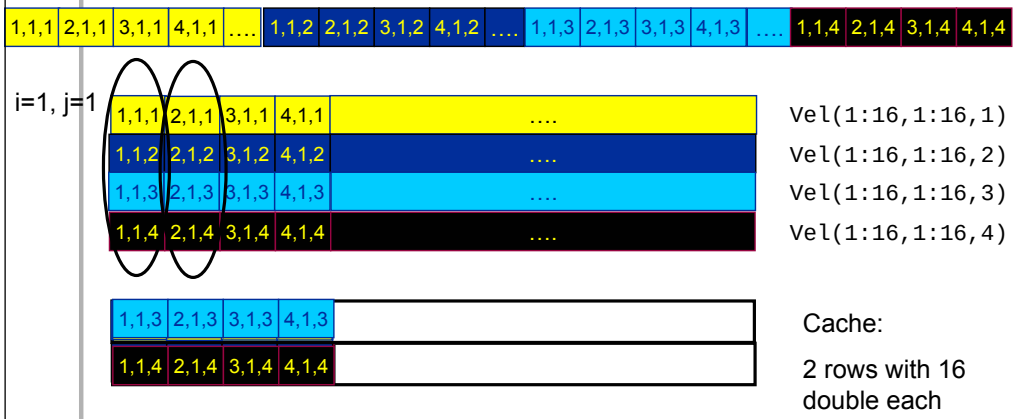
Example: 2D – square lattice
At each lattice point the 4 velocities for each of the 4 directions are stored



```
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

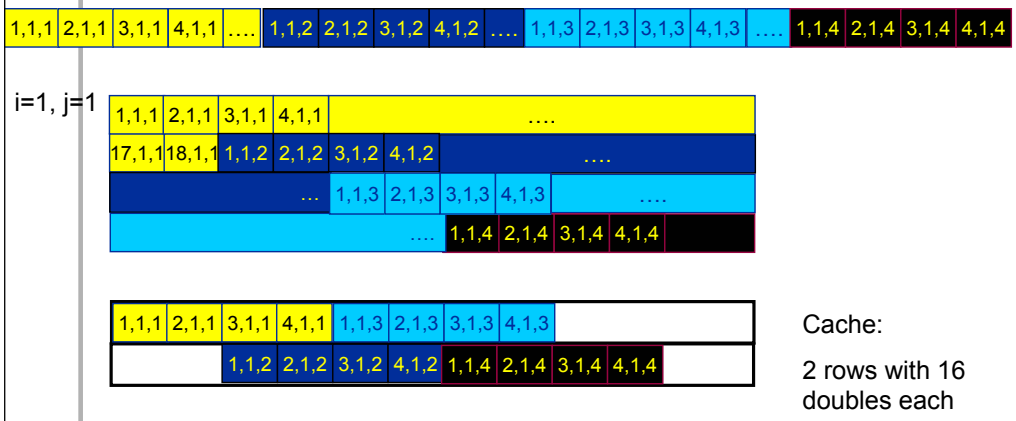
Memory to cache mapping for **`vel(1:16, 1:16, 4)`**
Cache: 256 byte (=32 double) / 2-way associative / Cache line size=32 byte



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

Memory hierarchies
Cache thrashing - Example

Memory to cache mapping for **`vel(1:18, 1:18, 4)`**
Cache: 256 byte (=32 doubles) / 2-way associative / Cache line size=32 byte



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

Implementation of the

3D lattice Boltzmann method

Lattice Boltzmann method
Discretization

- **Boltzmann Equation**

$$\partial_t f + \xi \cdot \nabla f = -\frac{1}{\lambda}[f - f^{(0)}]$$

ξ ... particle velocity
$f^{(0)}$ ... equilibrium distribution function
λ ... relaxation time

- **Discretization of particle velocity space**
  (finite set of discrete velocities)

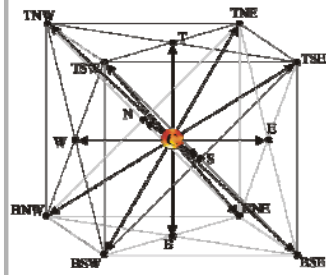
$$\partial_t f_\alpha + \xi_\alpha \cdot \nabla f_\alpha = -\frac{1}{\lambda}[f_\alpha - f_\alpha^{(eq)}]$$

$$f_\alpha(\vec{x},t) = f(\vec{x},\xi_\alpha,t)$$
$$f_\alpha^{(eq)}(\vec{x},t) = f^{(0)}(\vec{x},\xi_\alpha,t)$$

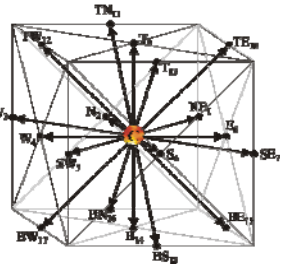
ξ_α – determined by discretization scheme

Lattice Boltzmann method
Discretization schemes

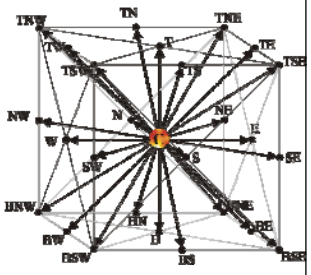
- **Different discretization schemes in 3D**
  - **Numerical accuracy and stability**
  - **Computational speed and simplicity**



D3Q15



D3Q19



D3Q27

We choose D3Q19 because of good balance between stability and computational efficiency

Lattice Boltzmann method
Stream and collide steps

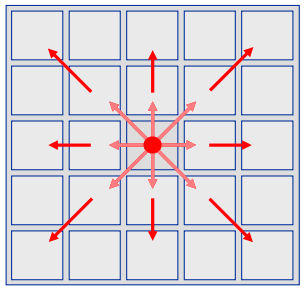
- **Discretization in space *x* and time *t*:**

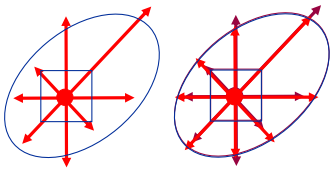
Relaxation of distribution function: Mainly floating point operations

collision step: $\tilde{f}_\alpha(x_i,t) = f_\alpha(x_i,t) - \omega\,[f_\alpha(x_i,t) - f_\alpha^{(eq)}(x_i,t)]$

streaming step: $f_\alpha(x_i + \vec{e}_\alpha \delta t, t+\delta t) = \tilde{f}_\alpha(x_i,t)$

Propagation of distribution function: Moving data only



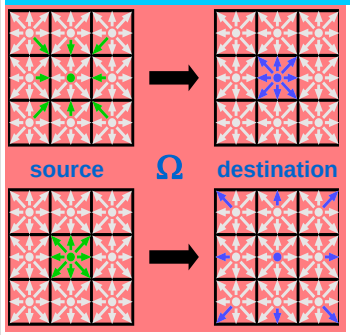


Lattice Boltzmann method
Stream and collide steps

- **Discretization in space *x* and time *t*:**

collision step: $\tilde{f}_\alpha(x_i,t) = f_\alpha(x_i,t) - \omega\,[f_\alpha(x_i,t) - f_\alpha^{(eq)}(x_i,t)]$

streaming step: $f_\alpha(x_i + \vec{e}_\alpha \delta t, t+\delta t) = \tilde{f}_\alpha(x_i,t)$



Stream-Collide (Pull-Method)

Get the distributions from the neighboring cells in the source array and store the relaxated values to one cell in the destination array

Collide-Stream (Push-Method)

Take the distributions from one cell in the source array and store the relaxed values to the neighboring cells in the destination array

We choose Collide-Stream in what follows

Lattice Boltzmann method: Basic implementation strategy

- Use "*full matrix*" representation: **F(0:18,x,y,z,0:1)**
  - **F(α,x,y,z,0) = $f_\alpha(x,y,z,t)$ & F(α,x,y,z,1) = $f_\alpha(x,y,z,t+1)$ (t odd; $\vec{x}$=(x,y,z))**
  - For complex geometries, *vector-like representations* might be interesting as they can block out the solid parts
  - However, the connectivity of the cells has to be stored additionally (therefore, you need at least 1/3 of solid to save memory)

Basic Optimizations

- Analyze relaxation step: Many operations can be eliminated (common sub-expressions, zero-velocity components): # of floating point operations depends on compiler & optimization level!
- **Combine Collide & Stream step in a single loop to minimize data transfer!**

Lattice Boltzmann method Basic implementation strategy

```
double precision F(0:18,0:xMax+1,0:yMax+1,0:zMax+1,0:1)
do z=1,zMax
   do y=1,yMax                          LD 1-2 Cachelines (cont. access)
      do x=1,xMax
         if( fluidcell(x,y,z) ) then
            LOAD F(0:18,x,y,z,t)                          } Collide Step
            Relaxation (complex computations)             }
            SAVE F( 0,x  ,y  ,z  ,t+1)                    }
LD & ST     SAVE F( 1,x+1,y+1,z  ,t+1)                    }
19 Cachelines SAVE F( 2,x  ,y+1,z  ,t+1)                  } Stream Step
            SAVE F( 3,x-1,y+1,z  ,t+1)                    }
            …                                             }
            SAVE F(18,x  ,y-1,z-1,t+1)                    }
         endif
      enddo                  If cache line of store operation is not in
   enddo                     cache it must be loaded first (RFO)!
enddo
```

#load operations: 19*xMax*yMax*zMax + 19*xMax*yMax*zMax

#store operations: 19*xMax*yMax*zMax

Lattice Boltzmann method Performance (1)

- Standard performance measure for LBM:
  Million fluid cell updates per second: MFLUP/s
  (or *Million Lattice cell updates per second: MLUP/s*)

Performance estimation

- Assumption:
  200 Floating point operations per fluid cell update
- If data transfer is infinite fast (all data fits into cache):
  - Max. MFLUP/s = PeakPerformance / (200 Flop/cell update)
  - Peak Performance = 4,000 MFLOP/s → Max. 20 MFLUP/s
  - In reality complex kernels do not achieve peak performance and transfer speed is finite even for caches → ~10-15 MFLUP/s

Lattice Boltzmann method Performance (2)

- If data must be transferred from and to main memory in each time step:
  - Assumption: full use of each cache line loaded
  - Data to be transferred for a single fluid cell update: (2+1)*19*8 Byte → 456 Bytes/(fluid cell update)
  - Max. MFLUP/s= MemoryBandwidth / (456 Bytes/(fluid cell update))
  - MemoryBandwidth = 6,000 MByte/s → Max. 13 MFLUP/s
  - In reality even simple kernels only achieve 50%-60% of theoretical main memory bandwidth → ~ 6-8 MFLUP/s
- Crossover between cache and memory bound computations:
  *2 *19 *xMax³ * 8Byte ~ L2/L3 cache size* → xMax ~ 14-15 (for 1 MB cache)

Optimization of data access for

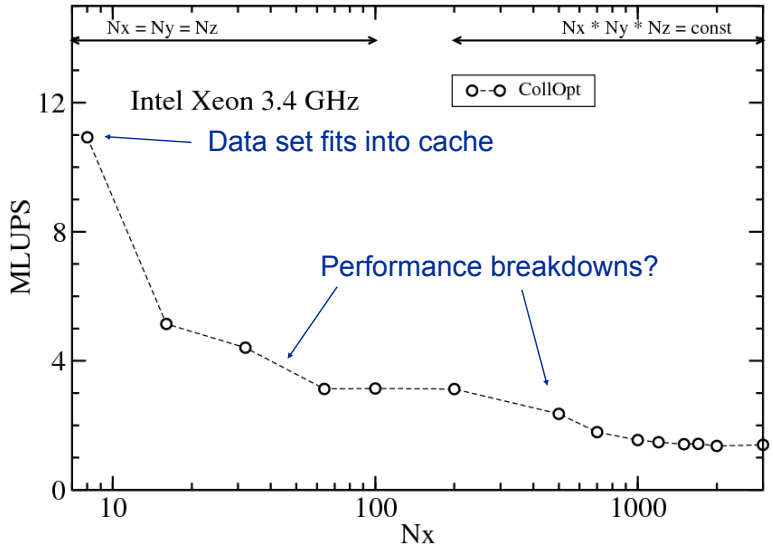
3D lattice Boltzmann method

Optimization of data access
Data layout

- In our case we have a 5-dimensional array F
  - `x,y,z` the three spatial directions
  - `i` the discrete velocity directions
  - `t` the source/destination pointer (current / next time step)
- In principle, any permutation of these indices can be used…
  …but which is the most efficient?
- In the following, it is assumed that by increasing the first index by one, you go to the physically next location in memory (FORTRAN)
  - `F(i,x,y,z,t)` „collision optimized“ („array of structures”)
  - `F(x,y,z,i,t)` „propagation optimized“ („structure of arrays”)

Optimization of data access
Collision optimized layout: F(0:18,x,y,z,t)

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



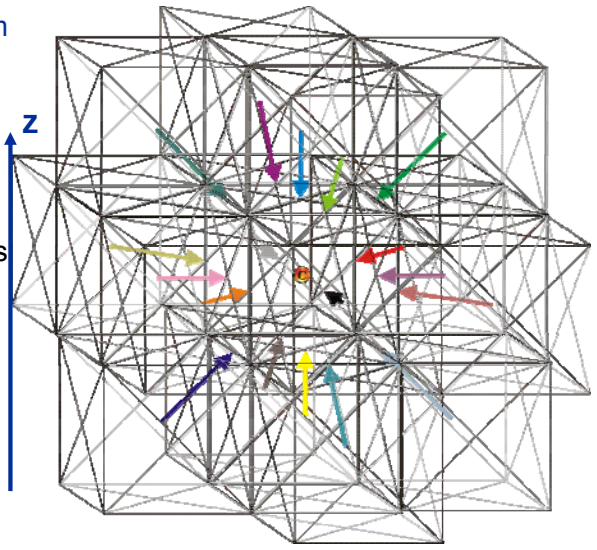
Optimization of data access
Collision optimized layout: F(0:18,x,y,z,t)

A single cell 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 * (xMax+2)*(yMax+2) Bytes

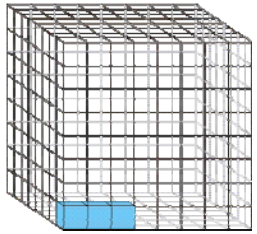
Cache lines must be reloaded if
Mem_z ~ L2/L3 cache

xMax=yMax ~ 33
(1 MByte cache)



Optimization of data access
Spatial blocking

- Increase spatial locality by spatial blocking



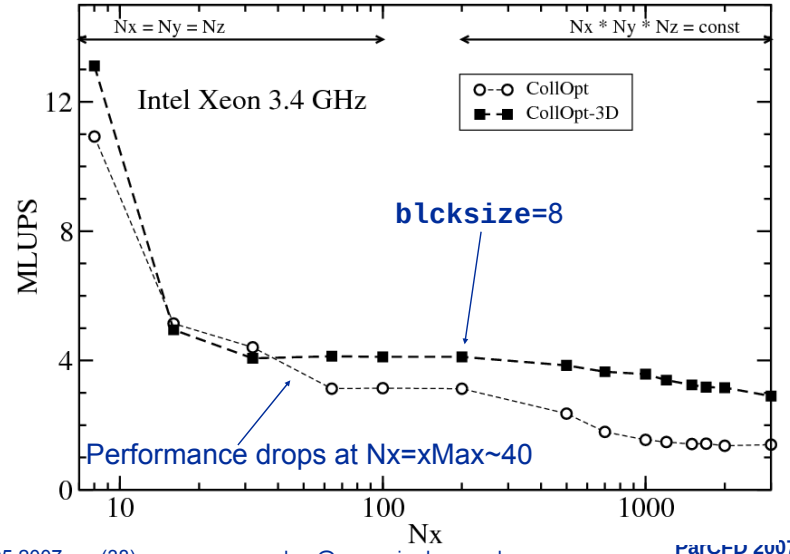
```
real(8) F(0:18,0:xMax+1,…,0:1)
do zz=1,zMax,blcksize
 do yy=1,yMax,blcksize
  do xx=1,xMax,blcksize

   do z=zz,min(zMax,zz+blcksize-1)
    do y=yy,min(yMax,yy+blcksize-1)
     do x=xx,min(xMax,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***blcksize**^{2} Byte < L2/L3 → **blcksize** ~25-30

Optimization of data access
Collision optimized layout: F(0:18,x,y,z,t)

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



Optimization of data access
Propagation optimized layout : F(x,y,z,0:18,t)

```
double precision F(0:xMax+1,0:yMax+1,0:zMax+1,0:18,0:1)
do z=1,zMax
   do y=1,yMax
      do x=1,xMax
         if( fluidcell(x,y,z) ) then
            LOAD F(x,y,z, 0:18,t)
            Relaxation (complex 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
```

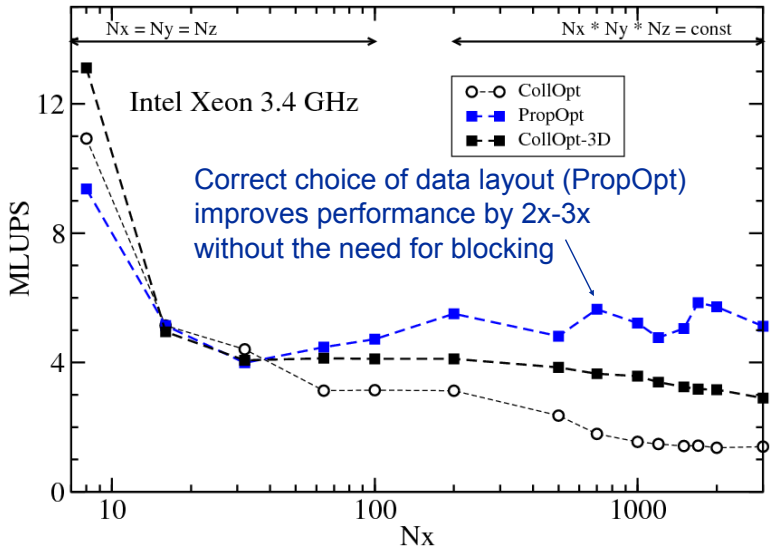
2*19 caches lines are touched for a single cell update – however they are accessed contiguously

High spatial data locality if 38 cache lines stay in the cache!

(38 * 128 Byte ~ 5 kByte << L2/L3 caches)

Optimization of data access
Impact of data layout: F(0:18,x,y,z,t) vs. F(x,y,z,0:18,t)

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



Optimization of data access
Propagation optimized layout: Cache thrashing

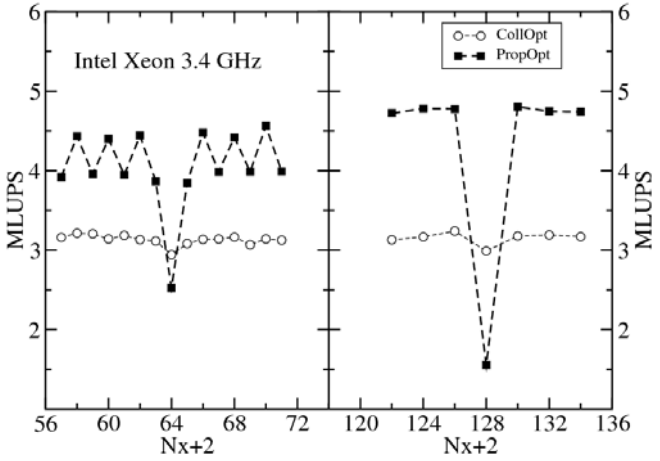
- Avoid powers of 2 in the leading dimensions of multidimensional array by array padding!

Use

`F(0:128,0:128,…)`

instead of

`F(0:127,0:127,…)`



xMax=yMax=zMax=Nx

Optimization of data access
Splitting innermost loops is essential!

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

do k=1,zMax; do j=1,yMax
   do i=1,xMax                                          (1)
      dens(i)=sum( F(i,j,k,:,t) )
      ux(i)=F(i,j,k,1,t)-F(i,j,k,5,t)+ …
      uy(i)= … ; uz(i)= …
   end do
   ! update (relax and advect) first half of directions (2)
   do i=1,xMax
      ! calculate common coefficients
      …
      sum15_1 = 3/4*coefA*(ux(i)+uy(i))
      sum15_2 = coefA*(ux(i)+uy(i))*(ux(i)+uy(i)) + usqnA
      F(i+1,j+1,k,1,t+1) = F(i,j,k,1,t)*coefA+sum15_1+sum15_2
      F(i-1,j-1,k,5,t+1) = F(i,j,k,5,t)*coefA-sum15_1+sum15_2
      …
   end do
   ! update (relax and advect) second half of directions (3)
   do i=1,xMax; similar calculations; end do
end do; end do
```


Optimization of data access
Splitting innermost loops is essential!

Woodcrest:
Intel Xeon 5150

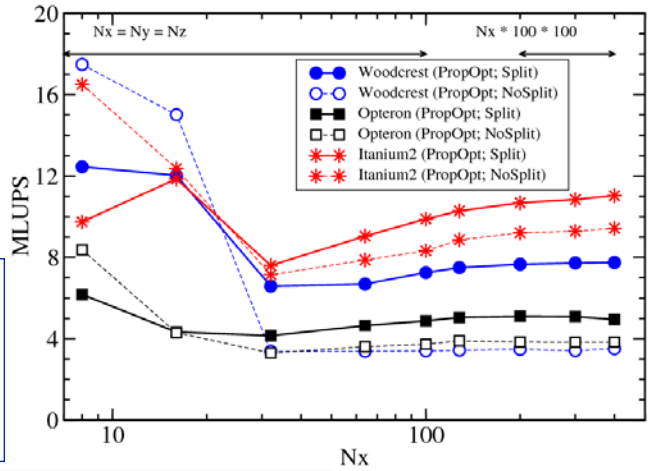
2.67 GHz
4 MB L2 (2 cores)

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

Summary – Efficient LBM implementation

- 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
- Not covered in this tutorial
  - Spatial blocking techniques for complex geometries
  - Temporal blocking techniques
  - Parallelization

Literature

Performance of LBM:

T. Pohl, N. Thürey, F. Deserno, U. Rüde, P. Lammers, G. Wellein, and T. Zeiser, In: IEEE/ACM: *Proceedings of the IEEE/ACM SC2004 Conference* (2004), pp. 1-13.
Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures

T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde, Parallel Processing Letters, Vol. 13, No. 4 (2003), pp. 549-560.
Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes

G. Wellein, T. Zeiser, G. Hager, and S. Donath, Computers & Fluids, Vol. 35 (2006), pp. 910-919.
On the Single Processor Performance of Simple Lattice Boltzmann Kernels

M. Schulz, M. Krafczyk, J. Tölke, E. Rank, in: M. Breuer, F. Durst, C. Zenger (Eds.), High Performance Scientific and Engineering Computing, Springer, Berlin (2001), pp. 115–122.
Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high performance computers

Literature

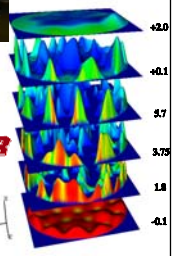
Performance of LBM:

J. Habich, Bachelor Thesis, Erlangen 2006.
Improving computational efficiency of Lattice Boltzmann methods on complex geometries

S. Donath, Bachelor Thesis, Erlangen 2004.
On Optimized Implementations of the Lattice Boltzmann Method on Contemporary Architectures

K. Iglberger, Bachelor Thesis, Erlangen 2003.
Performance Analysis and Optimization of the Lattice Boltzmann Method in 3D

Find these and more interesting Bachelor and Master Theses at:
http://www10.informatik.uni-erlangen.de/en/Publications/Theses/

Thank you!

Bayerisches Staatsministerium für Wissenschaft, Forschung und Kunst

KONWIHR

http://www.rrze.uni-erlangen.de/hpc/

Acknowledgement:
R. Vogelsang, R. Wolff (SGI)
W. Oed (CRAY)
Th. Schoenemeyer (NEC)

Acknowledgement:
M. Brehm et al. (LRZ)
U. Küster, P. Lammers (HLRS)
H. Cornelius, H. Bast, A. Semin (Intel)

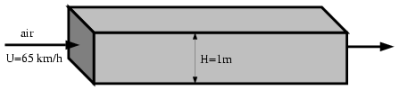
Efficient implementation of lattice Boltzmann kernels – Part II

T. Zeiser, G. Wellein, G. Hager, S. Donath
HPC Services
Regionales Rechenzentrum Erlangen
Friedrich-Alexander-University Erlangen-Nuremberg
Germany

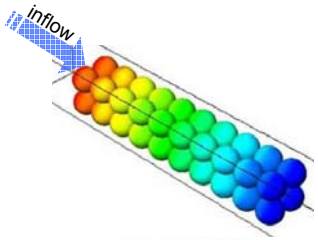
International Conference on Parallel Computational Fluid Dynamics 2007 – Short Course

Motivation

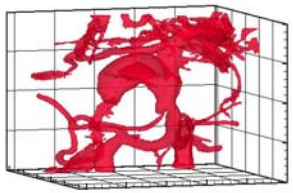
- **Turbulent channel flow**
  - → full array data structure ideal !
  - → **follow ideas of Part I**



- **Porous media / packed bed**
  - → **significant surface**
    handling of
    boundary conditions !?
  - → **significant amount of solid**
    full array = waste of memory !?



- **Blood flow**
  - → **only tiny fraction of bounding box is of interest**
    full array = waste of memory !!



Blood flow geometry by courtesy of K. Beronov, LSTM-Erlangen

The "universal object-oriented approach"

- **Use abstraction with fine-granular classes and derived objects for the different types of cells and algorithmic steps you have**
  - **fluid, solid, different types of boundary cells, ….**
- **Gives a lot of flexibility, allows optimal adaptation to the needs, but can also introduce much overhead.**
- **Seems to be a good idea for fast prototyping but not for a high performance production environment.**

Full arrays: dealing with solid cells & bounce back

- **Put everything into the relaxation-propagation loop: nested *if* will prevent much optimization!**

```
do z=1,zMax
  do y=1,yMax
    do x=1,xMax
    if( fluidcell(x,y,z) ) then
        ! do calculations for relaxation
        …
        ! do propagation / bounce back
        if ( fluidcell(x+1,y,z) then
          F(x+1,y,z,EAST,t+1)=…
      else
          F(x,y,z,WEST,t+1)=…
        endif
        if ( fluidcell(x,y+1,z) then
          F(x+1,y,z,NORTH,t+1)=…
      else
          F(x,y,z,SOUTH,t+1)=…
        endif
        …
      endif
    enddo
  enddo
enddo
```

- **Do uniform relaxation/propagation first and correct afterwards: waste of CPU cycles and additional memory traffic …**

```
! first step: uniform relaxation/propag.
do z=1,zMax; do y=1,yMax; do x=1,xMax
  ! do calculations for relaxation
  ! do uniform propagation
  F(x+1,y,z,EAST,t+1)=…
  F(x+1,y,z,NORTH,t+1)=…
  …
enddo; enddo; enddo
! second step: bounce back correction
do z=1,zMax; do y=1,yMax; do x=1,xMax
   if( .not. fluidcell(x,y,z) ) then
     F(x-1,y,z,WEST,…)=F(x,y,z,EAST,…)
     F(x+1,y,z,EAST,…)=F(x,y,z,WEST,…)
     F(x,y+1,z,NORTH,…)=…F(x,y,z,SOUTH,…)
   …
enddo; enddo; enddo
```

- Alternatively, just looping over the first solid cell layer is possible for the 2nd step…

Beyond full arrays

Full arrays:

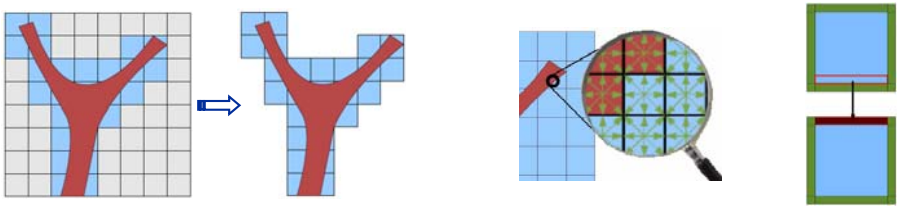
- **waste of memory if fluid cells make only a small fraction of the total cells in the bounding box**
- **efficiency strongly depends on boundary handling**

Alternatives:

- **use "patches"**
- **use "unstructured/sparse" approach**
- **…**

Principle idea of “patch” approach

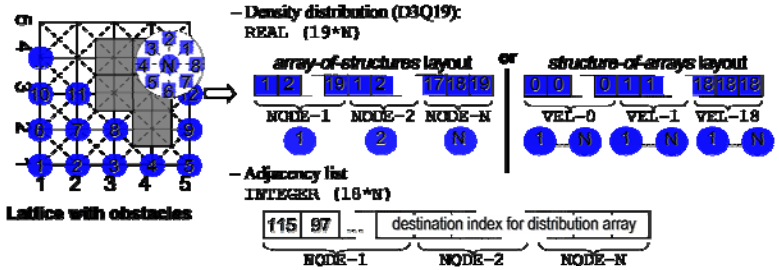
- Divide the domain into small patches
  - Use full arrays within the each patch
  - Use ghost cells for exchange between patches
  - only allocate patches which contain fluid areas
  - a patch should be much smaller than the total domain to block out most of the solid
  - but large enough to allow efficient processing
- Very much like parallelization with domain decomposition (but many patches go to the same processor)



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

References: IWTM/FHG, LSS/Uni-Erlangen, …

Unstructured approach: LB-DC implementation



- Long 1-D list of all fluid cells
  - 1 loop instead of 3 nested loops
    → good for vectorization, low loop overhead
  - indirect data access via adjacency list
    → overhead *but* allows flexible ordering of cells
    → moves effort from flow solver to preprocessor
  - additional memory requirements for adjacency list
    → but only fluid cells allocated

Schematic data structure by courtesy of J. Bernsdorf, NEC CCRLE

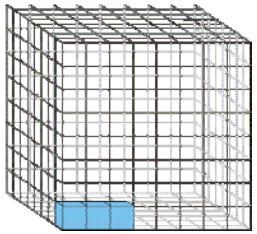
References: C. Pan et al. Comp.Phys.Comm. (2004), Krafczyk et al. (2000,2001,2002),…

Optimization of data locality

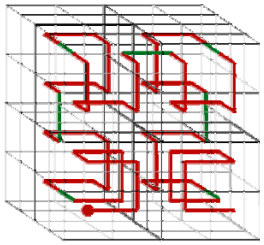
- Processing of 1-D list no longer depends on lexicographic ordering owing to (necessary) use of adjacency list

→ added flexibility, e.g.

- “implicit” blocking (without loop overhead)
- use of space filling curves
- …



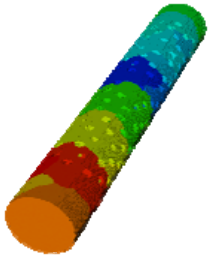
→ No need for the solver to know about the ordering!



→ Automatic handling of bounce back by adjacency list!

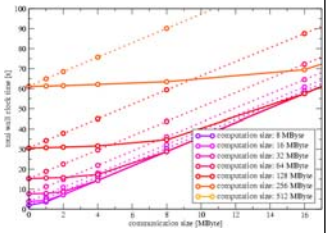
Parallelization by domain decomposition

- Cartesian cutting is no longer appropriate
- “unstructured grid” → use graph pratitioner (e.g. metis)
- or just cut the 1-D list into equal chunks



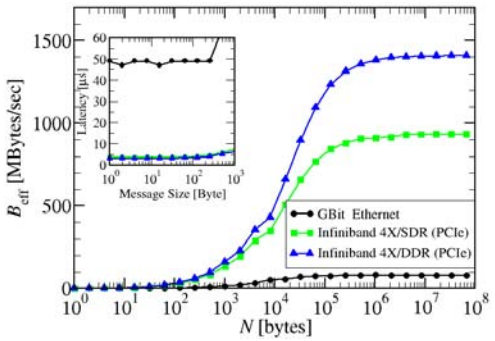
→ avoiding **load imbalance** is more important than minimizing communication

→ Trying to overlap communication and computation is generally not worth the effort as most MPI implementation do not support it out of the box (i.e. data transfer only takes place while within an MPI call).



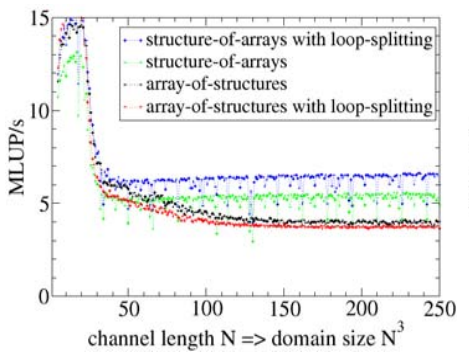
Why is *load balance* more important than *communication*?

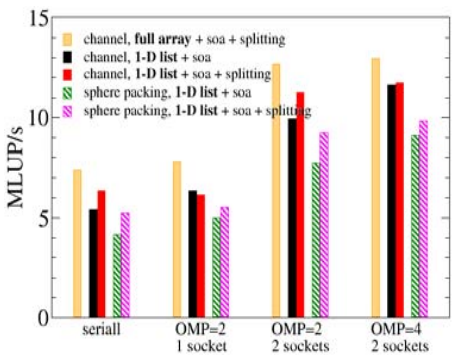
- Let's assume
  - 10^5 cells/processor (40 MB only)
  - 10 MLUPs/processor (very fast)
  - →0.01s per timestep
  - 6 communication partners per partition each with 2500 cells (6x 200kB)
  - →0.0002s/exchange for IB
- Communication makes up only a tiny fraction of the total time



Characteristics of interconnects according to Ping-Pong benchmark

Is the 1-D list approach competitive?



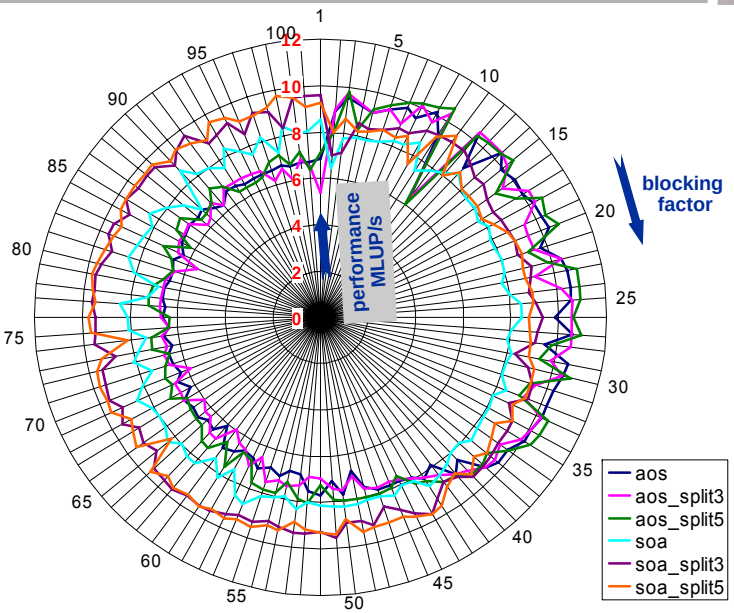


Measured on "Woody" cluster at RRZE (HP DL140G3: 2 socket Xeon 5160)

- The same basic optimization strategies apply:
  - data layout
  - splitting of loops
  - danger of cash thrashing

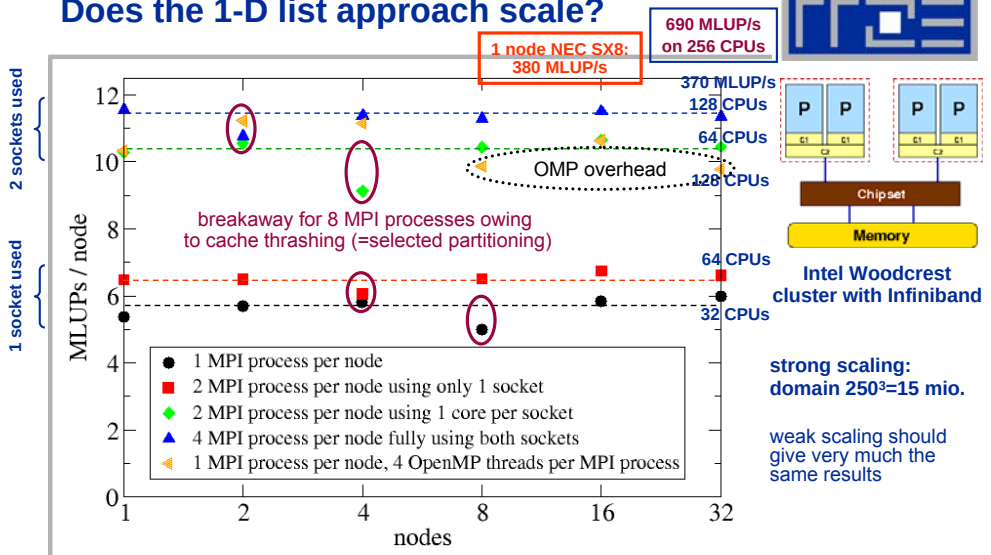
- Only little performance lost even for empty channel
- Complex geometry causes little extra overhead
- OpenMP compensates some overhead

Optimization of blocking factor



Measured on "snode2xx" (2 socket Opteron nodes with Dual Core 2.0 GHz, OMP=4)

Does the 1-D list approach scale?



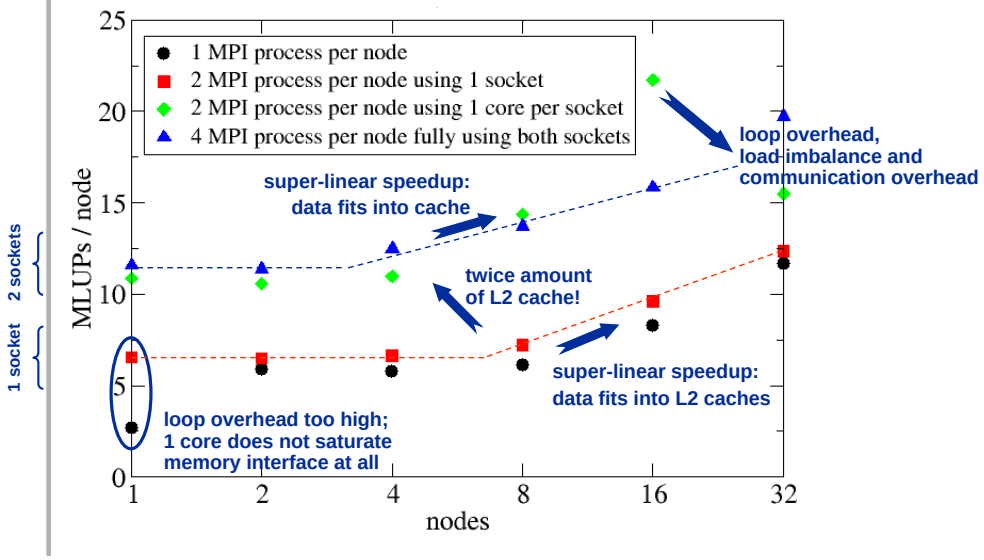
Intel Woodcrest cluster with Infiniband

strong scaling: domain 250^3=15 mio.

weak scaling should give very much the same results

- the LB implementation scales; but the memory interface does not
- stick with pure MPU; hybrid (MPI+OpenMPI) does not give a benefit

If the domain is too small ...

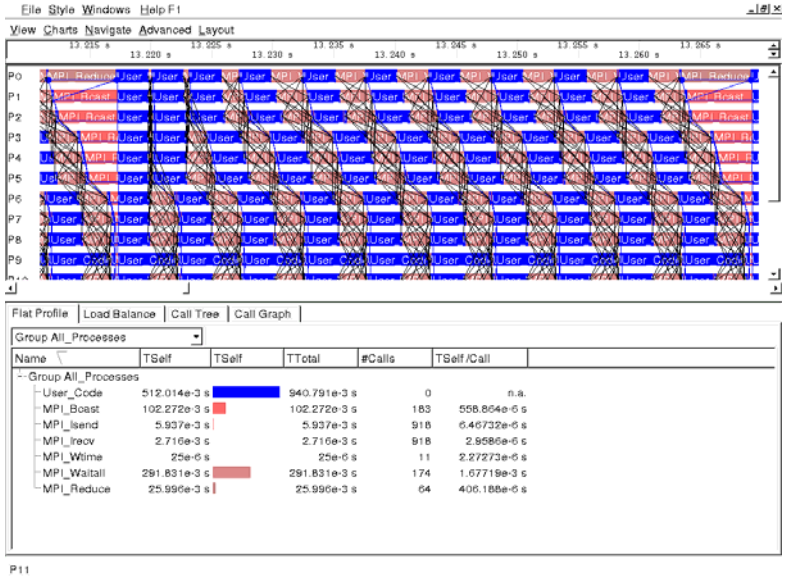


Summary

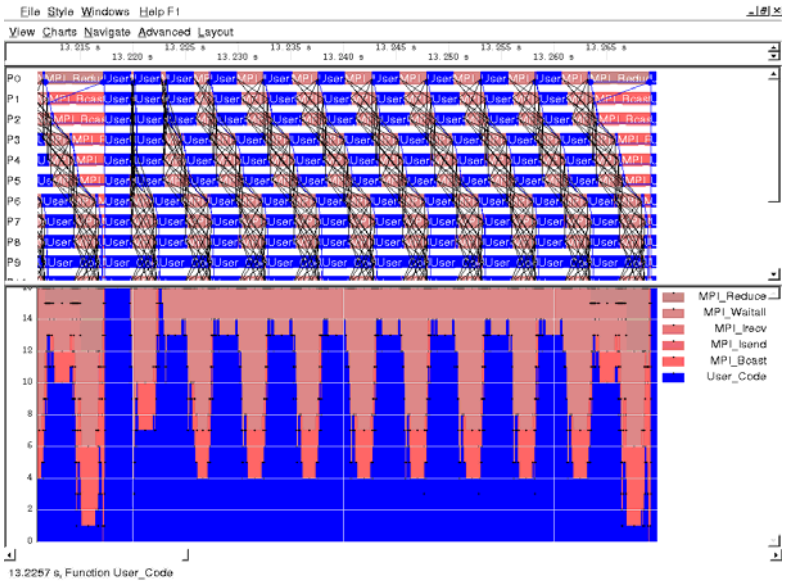
- Optimization strategies from simple full array kernels can be applied to list-based sparse data structures.
- List-based sparse data structure provide additional opportunities for data access optimizations.
- Most of them are exploited at the preprocessing stage, thus, they do not increase the complexity of the LBM solver.
- With regard to speed, they are competitive with full array kernels even for channel configurations.
- More complex geometries lead to significant advantage.

- Performance characteristics are predictable or at least explainable.
- Tools can assist to point out bottlenecks.

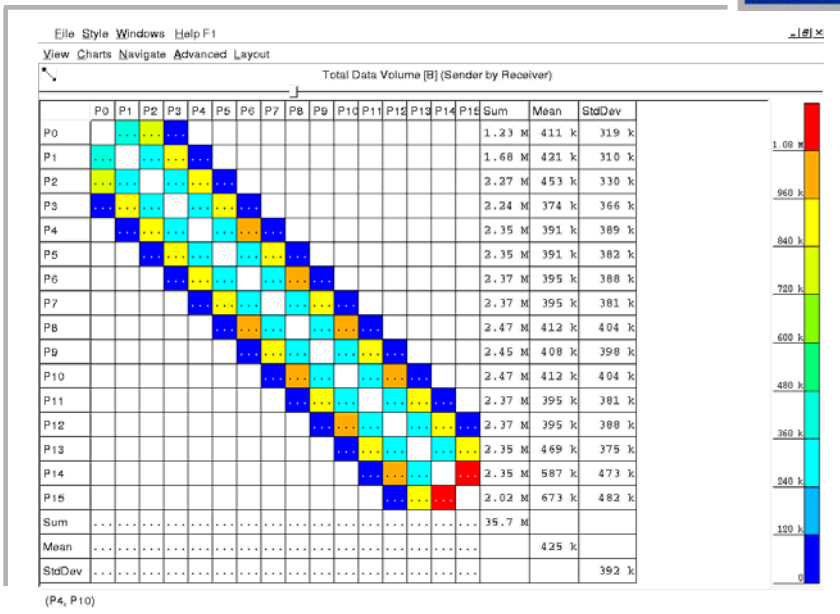
Intel Trace Collector/Analyzer (I)



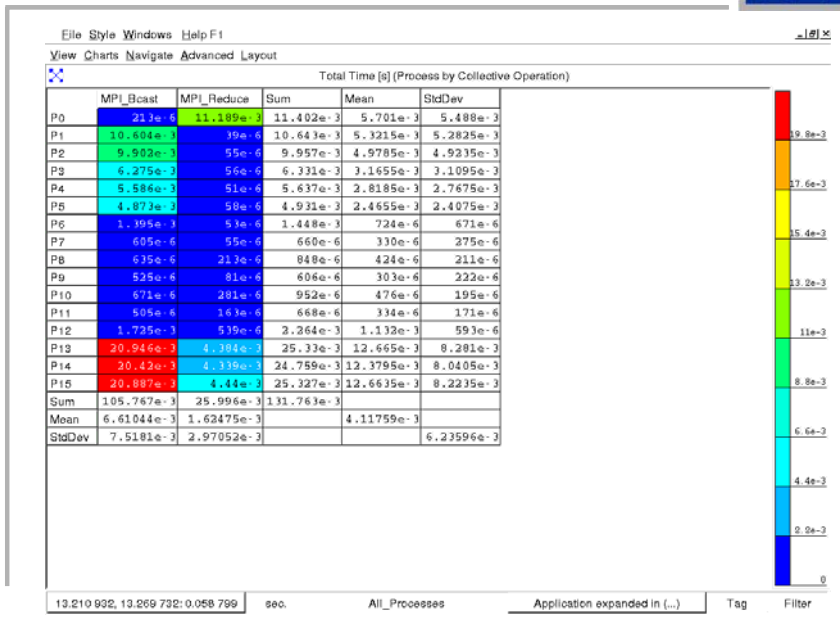
Intel Trace Collector/Analyzer (II)



Intel Trace Collector/Analyzer (III)

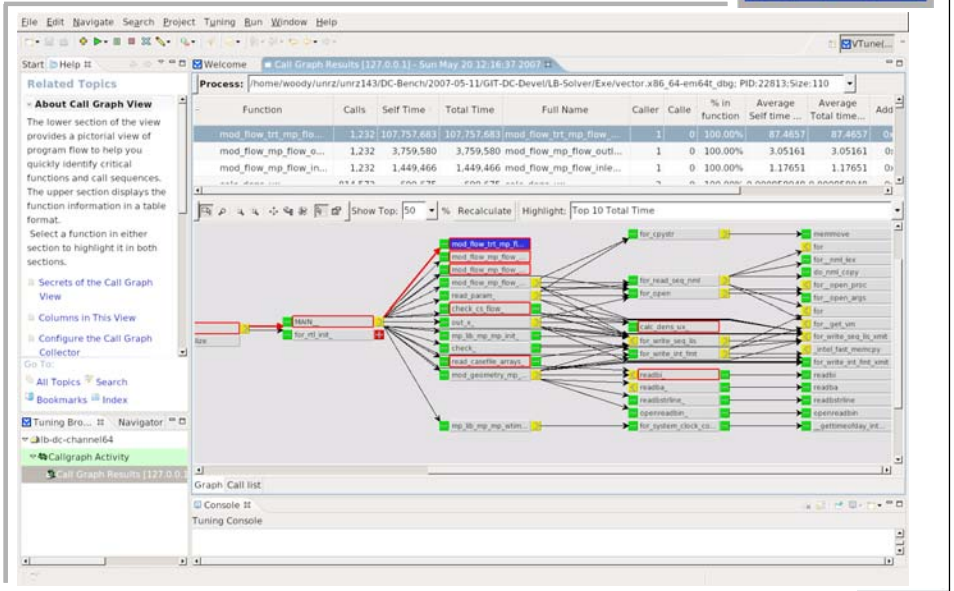


Intel Trace Collector/Analyzer (IV)

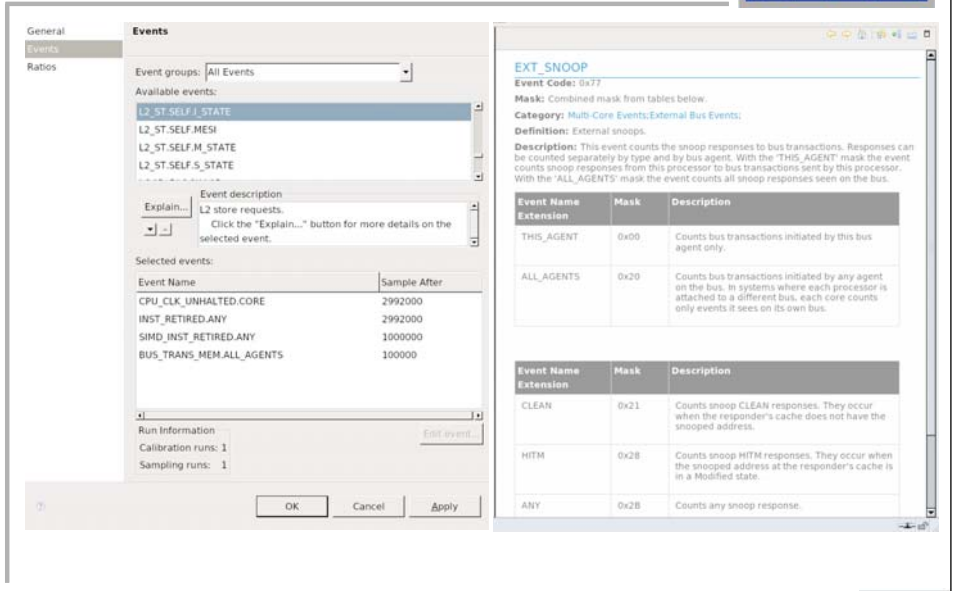


	MPI_Bcast	MPI_Reduce	Sum	Mean	StdDev
P0	21e-6	11.189e-3	11.402e-3	5.701e-3	5.488e-3
P1	10.604e-3	19e-6	10.643e-3	5.3215e-3	5.2825e-3
P2	9.902e-3	55e-6	9.957e-3	4.9785e-3	4.9235e-3
P3	6.275e-3	56e-6	6.331e-3	3.1655e-3	3.1095e-3
P4	5.586e-3	51e-6	5.637e-3	2.8185e-3	2.7675e-3
P5	4.873e-3	58e-6	4.931e-3	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	1.735e-3	539e-6	2.264e-3	1.132e-3	593e-6
P13	20.946e-3	4.304e-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

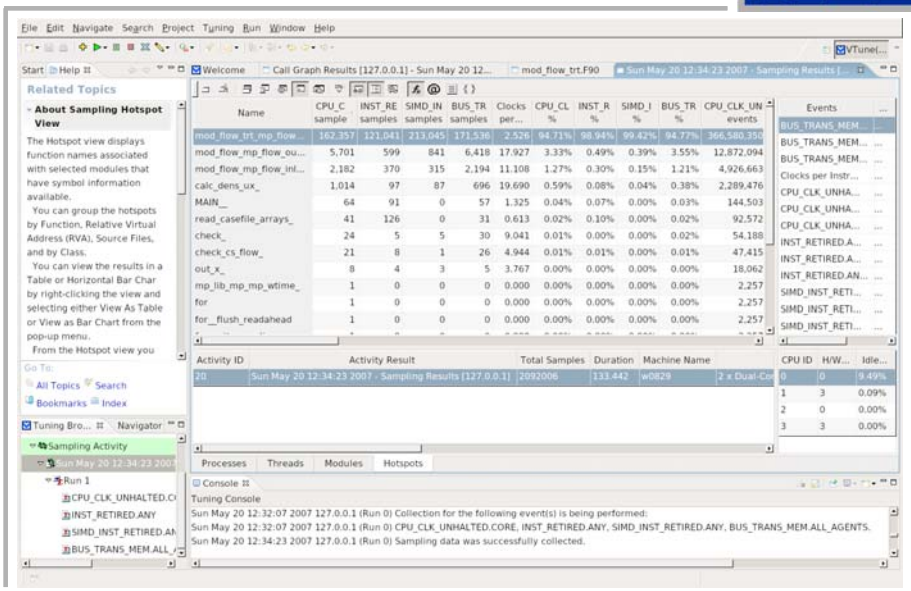
Intel VTune (I)



Intel VTune (II)



Intel VTune (III)



Intel VTune (IV)