Wavefront-Parallel Temporal Blocking on Multi-Core Processors with Shared Caches

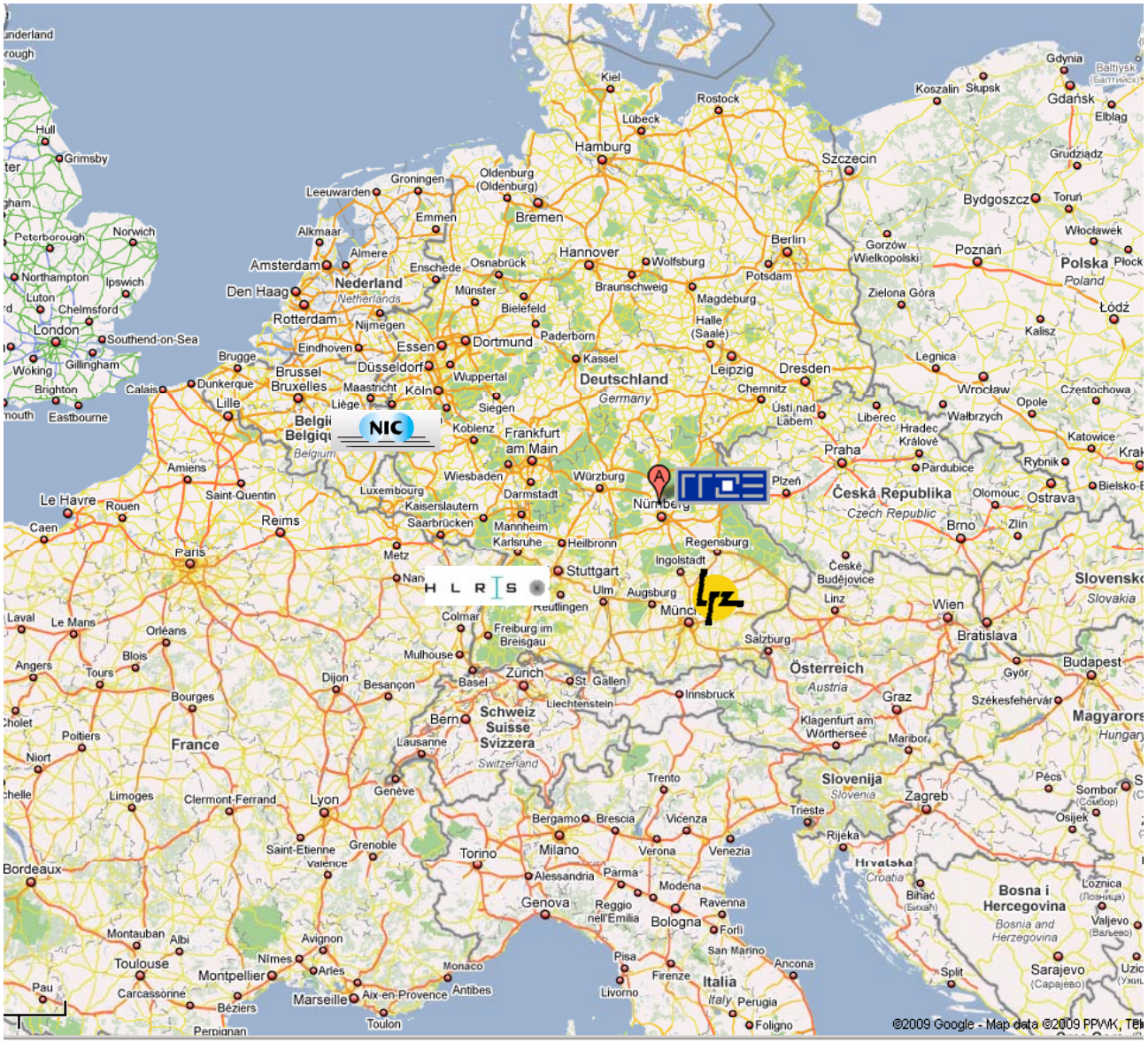
HPC Services @ Erlangen Regional Computing Center (RRZE):

Gerhard Wellein
Georg Hager
Markus Wittmann
Johannes Habich
Thomas Zeiser
Jan Treibig
Gerald Schubert
Michael Meier
Holger Stengel

LANL PAL, Aug 26th, 2009

RRZE

- **RRZE = Erlangen Regional Computing Center**
  - ≈ 100 employees and students,
    8 in HPC Services
  - 14 TFlop/s clusters
    & some “hot silicon”
  - “IT Service Provider for FAU”



- **FAU = Friedrich-Alexander University of Erlangen-Nuremberg**
  - Second largest university in Bavaria
  - 26000 students
  - 12000 employees
  - 550 professors
  - 260 chairs

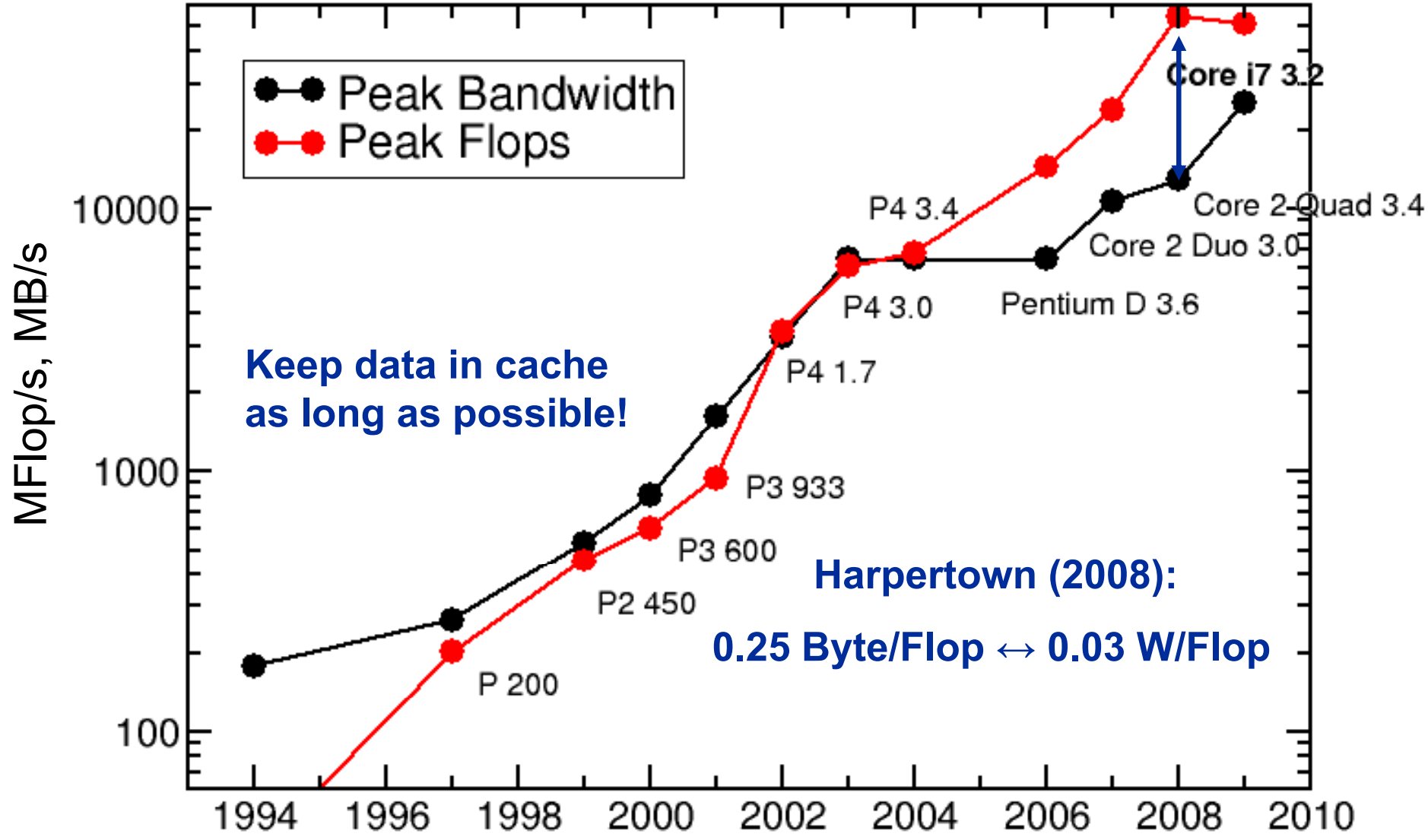
HPC Services @ RRZE:
Tasks

- **Procuring and running RRZE's HPC systems**
  - Fundraising, tenders, selection process
  - Installation, system administration
  - Scientists are closely involved in procurements
  - Guiding scientists to optimal use of local and remote HPC resources
- **Integration into research and teaching at FAU**
  - >70 publications (since 1999)
  - >60 talks (since 1999)
  - Regular HPC tutorials, workshops, courses
  - Regular lecture
    (*Programming Techniques for Supercomputers*)
  - Supervision of bachelor's and master's theses

Publication lists
http://www.blogs.uni-erlangen.de/hager/topics/Publications
http://grid.rrze.uni-erlangen.de/~unrz143/publications.html

Research

- **Research areas**
  - General and architecture-specific performance optimization
  - Programming techniques for HPC
  - Assessment of novel processors and systems for HPC
  - Lattice-Boltzmann methods
- **Some recent projects**
  - HQS@HPC: Highly correlated quantum systems on high performance computers
  - JaDa: Exact diagonalization of Large Sparse Matrices
  - Lattice-Boltzmann methods (LBM)
    - Optimized Implementations of the lattice Boltzmann method in 3D
    - Improving computational efficiency of LBM on complex geometries
    - Performance Evaluation of Numeric Compute Kernels on NVIDIA GPUs
    - SKALB: Lattice-Boltzmann methods for scalable multi-physics applications
  - Optimizing data access on systems with non-uniform memory access characteristics and/or aliasing issues
  - Evaluation of Windows Compute Cluster Server (CCS)
    - Only production-grade Windows cluster in Bavaria

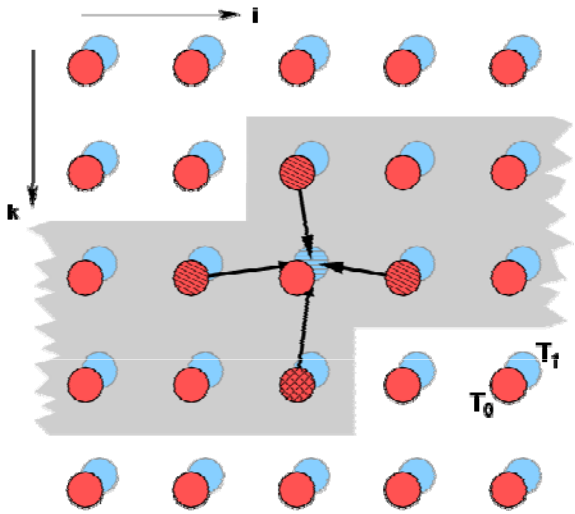


Single-socket Intel chips: Peak Performance & Memory Bandwidth over time

- **Jacobi iteration as a paradigm for regular stencil based iterative methods: Basics and baseline implementation in three spatial dimensions (3D)**
- **Conventional temporal blocking in 3D**
- **Wavefront approach**
- **Outlook**

Jacobi Solver

Basics: 2 arrays; naïve version




```
do k = 1 , Nk
  do j = 1 , Nj
      do i = 1 , Ni
          y(i,j,k) = a*x(i,j,k) + b*
                     (x(i-1,j,k)+x(i+1,j,k)+
                      x(i,j-1,k)+x(i,j+1,k)+
                      x(i,j,k-1)+x(i,j,k+1))
enddo; enddo; enddo
```

- Performance metric: Million Lattice Site Updates per second (MLUPs)
  - Equivalent MFLOPs: 8 FLOP/LUP * MLUPs
- Bandwidth requirements: 16 Byte / Lattice Site Update (LUP) if:
  - N*N*(8Byte)*2 < Cache size → Cache size = 2 MB → N ~ 350
  - No Read for Ownership (RfO) on y ("nontemporal stores")
- Performance estimate: **B_M / (16 Byte/LUP)**
  (B_M : attainable memory bandwidth as measured with STREAM;
  B_M = 8 GByte/s → 500 MLUPs)

Jacobi solver

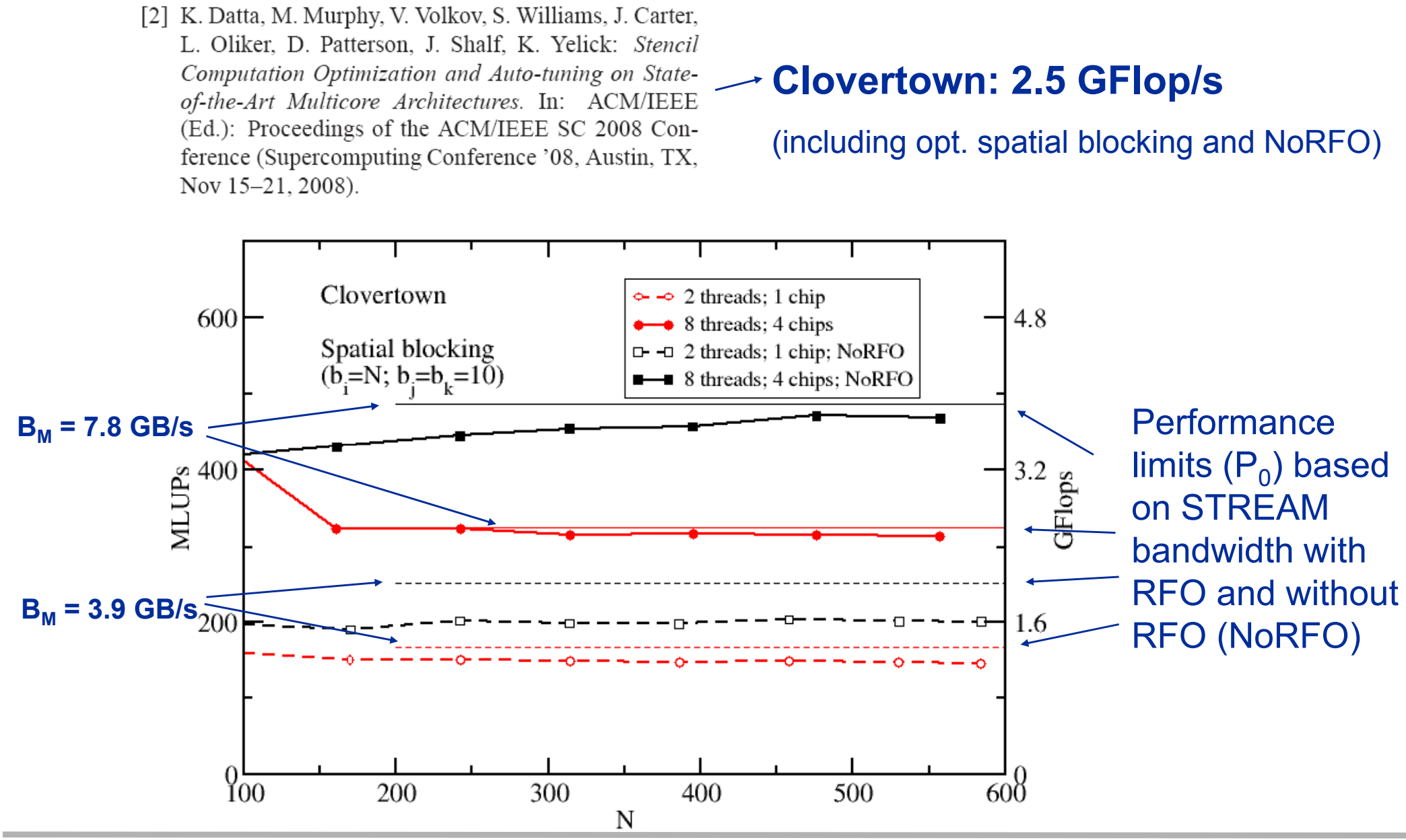
Benchmark architectures: Intel Quad-/Hexa-Cores

		Clovertown	Nehalem	Dunnington
Type		Xeon 5345 @2.33 GHz	"Core i7" @2.66 GHz	Xeon 7460 @2.66 GHz
L1 group	size [kB]	32	32	32
	TRIAD GB/s	3.9	11.6	3.0
L2 group	L2 size [MB]	4	0.25	3
	# cores	2	1	2
	TRIAD GB/s	4.0	see L1	3.5
L3 group / socket	L3 size [MB]	-	8	16
	# cores	4	4	6
	TRIAD GB/s	4.0	16.6	3.5
System	# sockets	2	2	4
	raw bw [GB/s]	21.3	51.2	34.0
	TRIAD GB/s	7.8	32.7	13.2

Stream TRIAD (array size: 20,000,000; nontemporal stores via compiler)

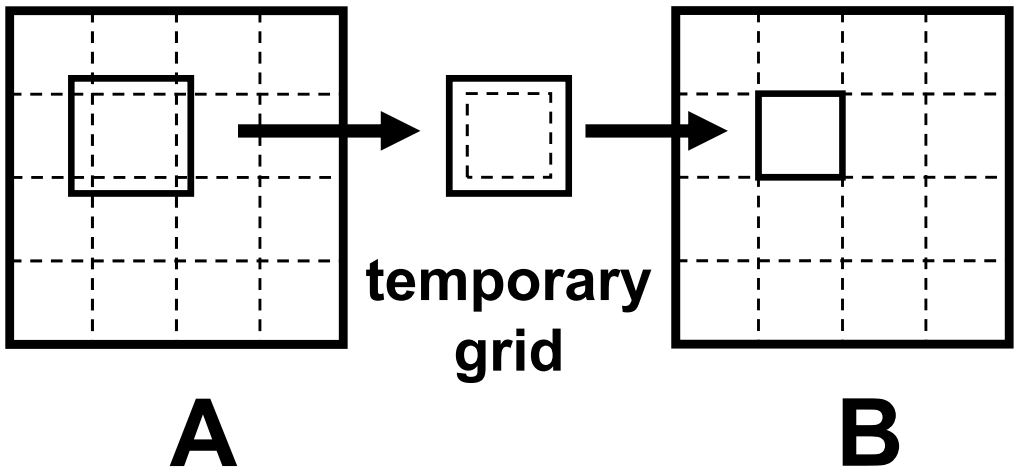
Nehalem: Early access; pre-production; SMT disabled

Jacobi solver
Baseline



Conventional temporal blocking approaches:

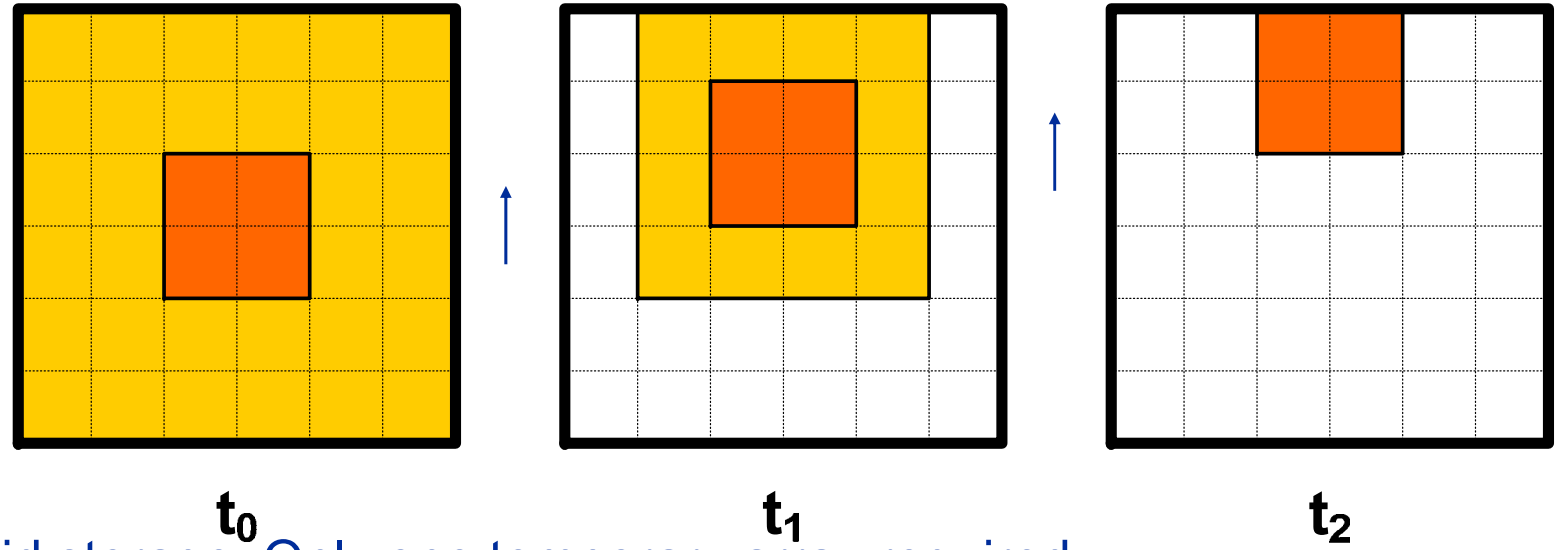
Load a small block into cache and do multiple updates on it



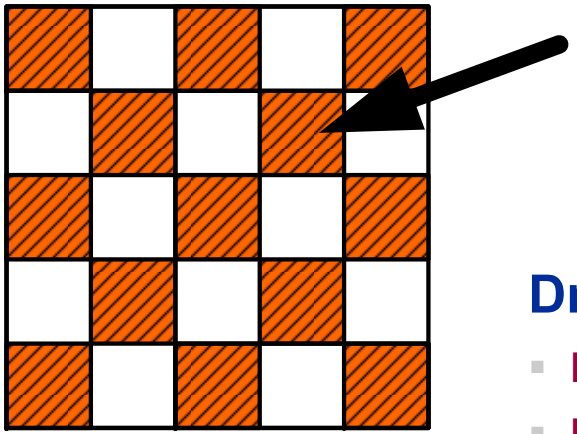
Conventional temporal blocking

A "good" halo implementation

- Load $(N_b*t_b)x(N_b*t_b)$ block & perform t_b time steps on N_b X N_b block
- $N_b=2$; $t_b=2$:



- Compressed grid storage: Only one temporary array required
- "Checkerboard alignment" of rows ensures aligned loads
- Non-diagonal shift ensures alignment of loads with store

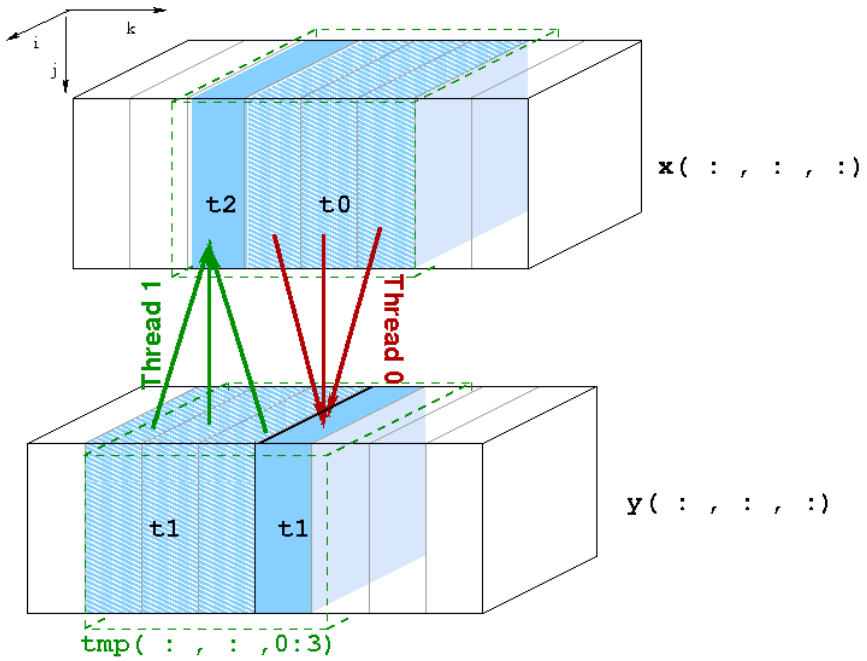


- destination cell for compressed grid
- stencil update is auto-vectorizable

Drawbacks:

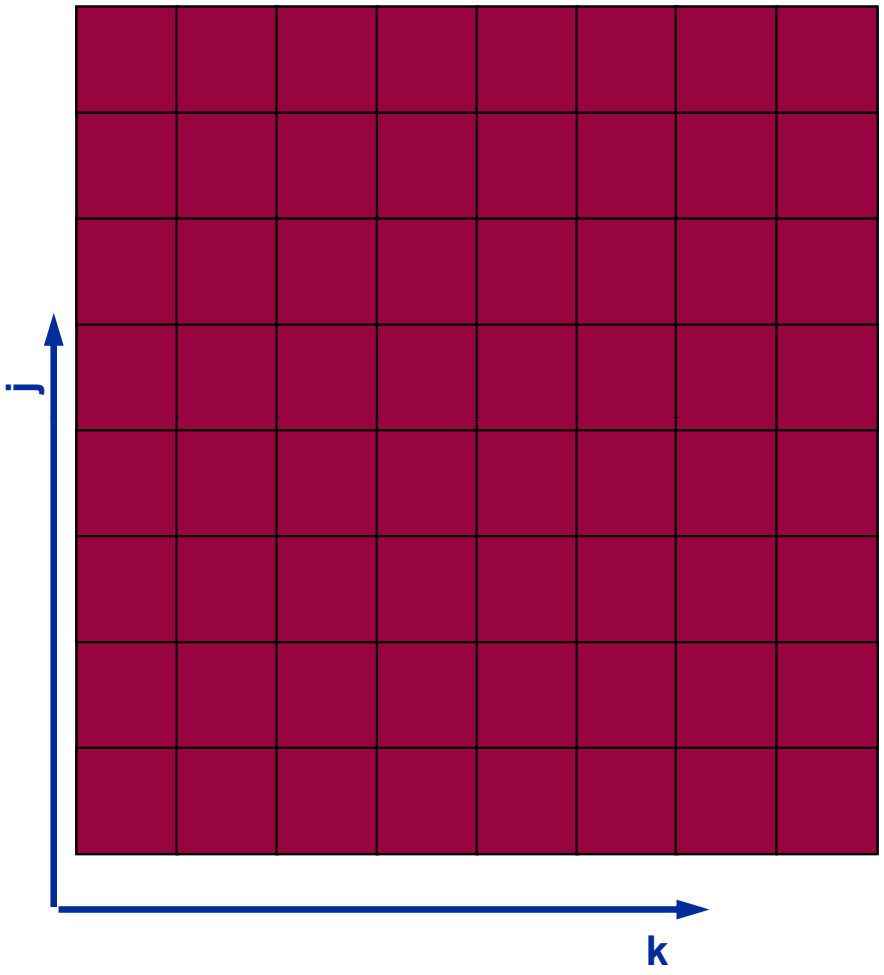
- **No overlap of computation and mem. traffic**
- **Halo overhead**

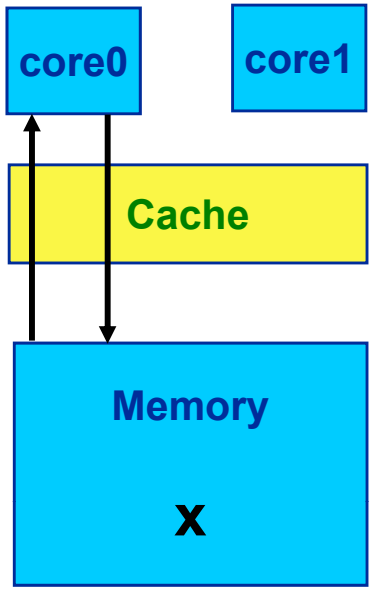
Temporal blocking by pipelining: Wavefront parallelization



Jacobi Solver

Standard sequential implementation

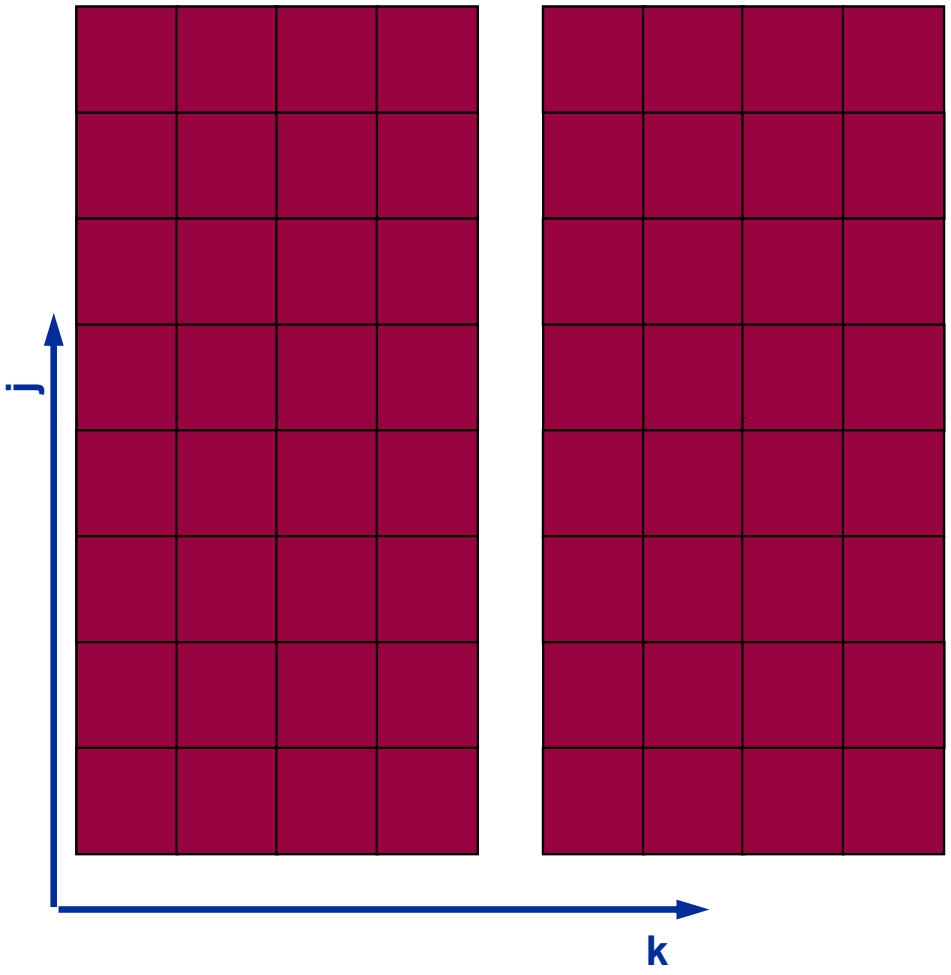


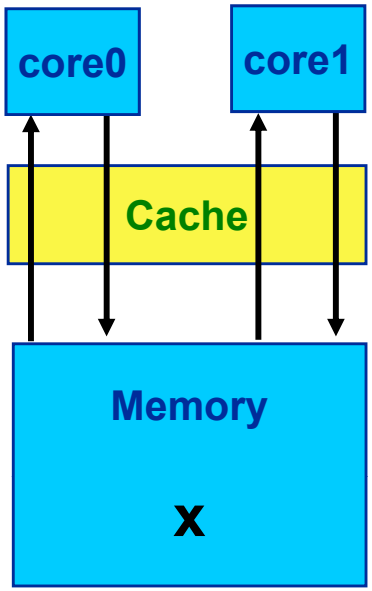



```
do t=1,tMax

   do k=1,N
      do j=1,N
         do i=1,N
            y(i,j,k) = …
         enddo
      enddo
   enddo

enddo
```

Jacobi solver

Standard naive shared memory parallel

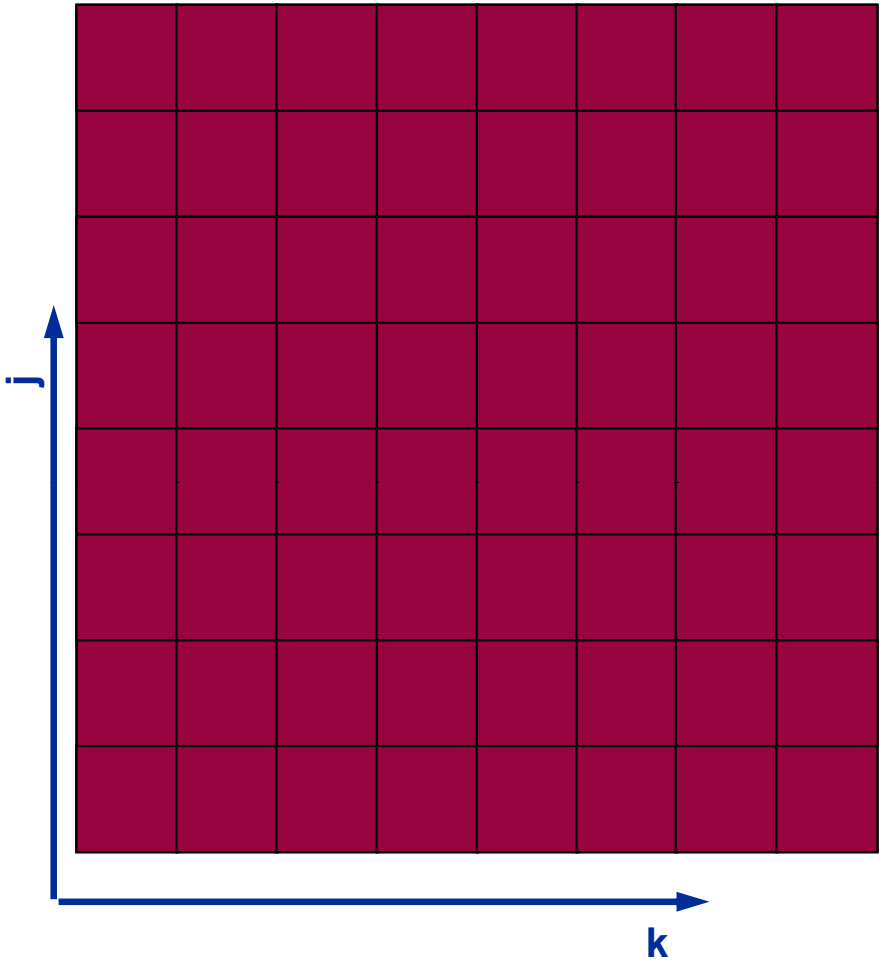


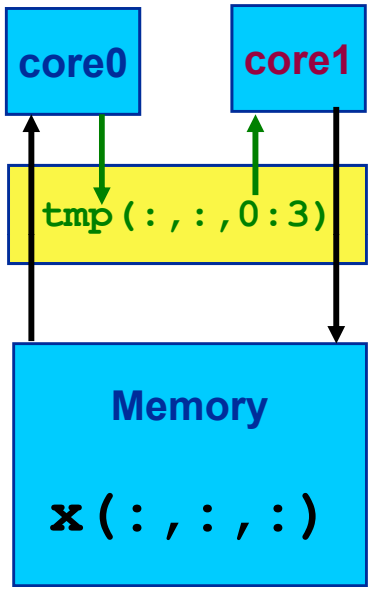



```
do t=1,tMax
!$OMP PARALLEL DO private(…)
  do k=1,N
     do j=1,N
        do i=1,N
           y(i,j,k) = …
        enddo
     enddo
  enddo

enddo
```

Jacobi solver

Propagating two wavefronts!





`y(:,:,:)` is obsolete!

Save main memory data transfers for `y(:,:,:)`!

Use small buffer `tmp(:,:, 0:3)` which fits into cache

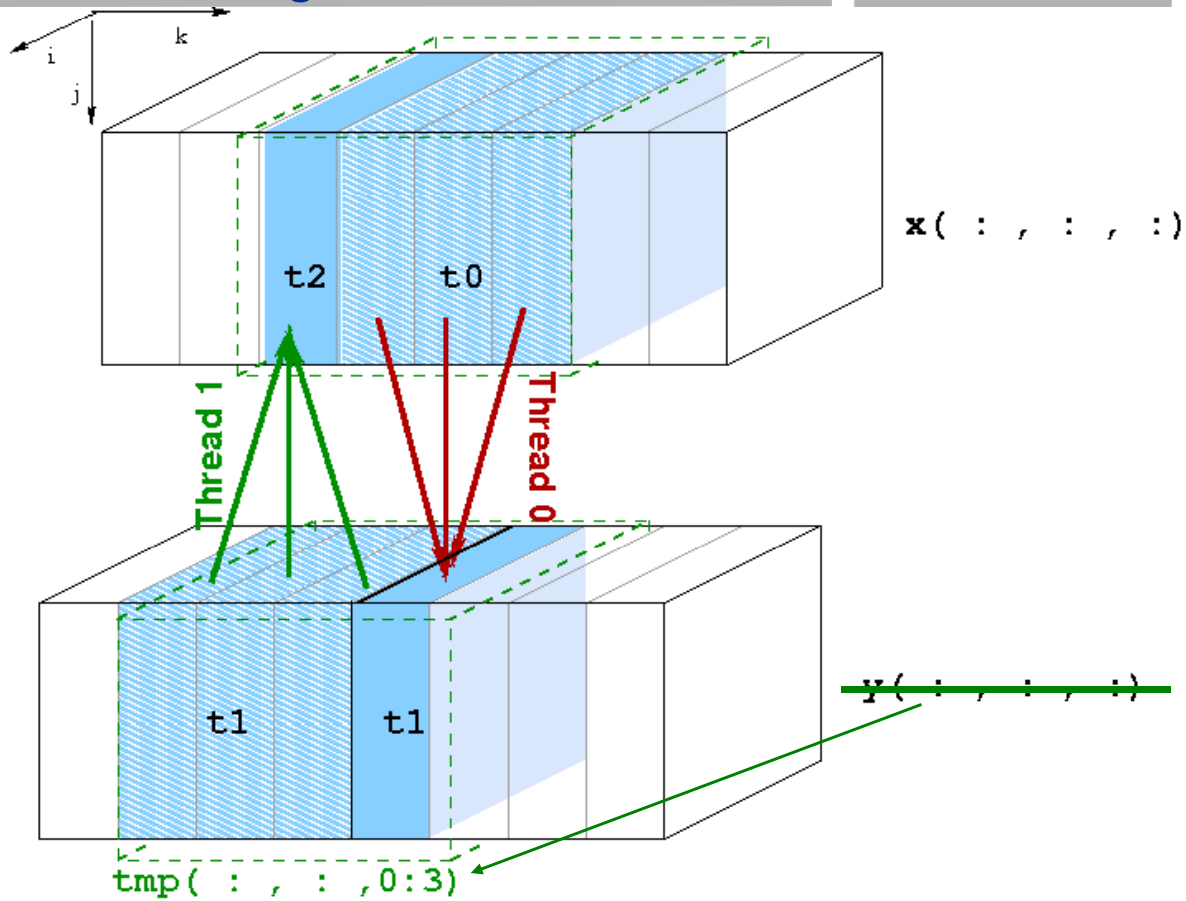
Sync threads/cores after each k-iteration

Core 0: `x(:,:,k-1:k+1)`$_t$ → *`tmp(:,:,mod(k,4))`*

Core 1: *`tmp(:,:,mod(k-3,4):mod(k-1,4))`* → `x(:,:,k-2)`$_{t+2}$

Jacobi solver

Wavefront parallelization: Temporal blocking



`y(:,:,:)` is obsolete!

Use small buffer
`tmp(:,:,0:3)`
which fits into the cache

↓

Save main memory data transfers for `y(:,:,:)` !

↓

16 Byte / 2 LUP !
(instead of 32)

Compare with baseline: Maximum speedup of 2 can be expected

(assuming infinitely fast cache and no overhead for OMP BARRIER after each k-iteration)

Jacobi solver

Wavefront parallelization: Temporal blocking

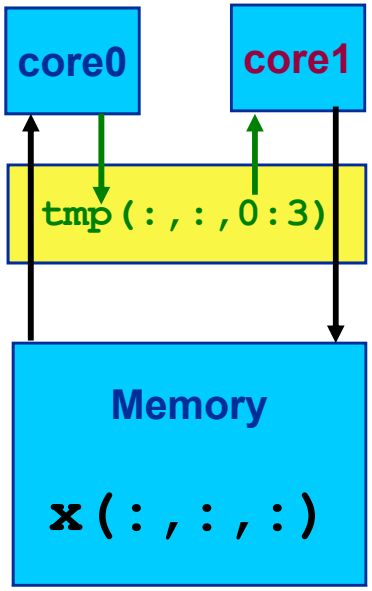
Thread 0: `x(:,:,k-1:k+1)`$_t$ → `tmp(:,:,mod(k,4))`

Thread 1: `tmp(:,:,mod(k-3,4):mod(k-1,4))` → `x(:,:,k-2)`$_{t+2}$

Performance model including finite cache bandwidth (B_C):

Time for 2 LUPs:

$$T_{2LUP} = 16\ \text{Byte}/B_M + x * 8\ \text{Byte}/B_C = T_0\ (1 + x/2 * B_M/B_C)$$



Minimum value: x =2

Speedup vs. baseline: $\mathbf{S_W = 2*T_0/T_{2LUP}}$
$\mathbf{= 2 / (1 + B_M/B_C)}$

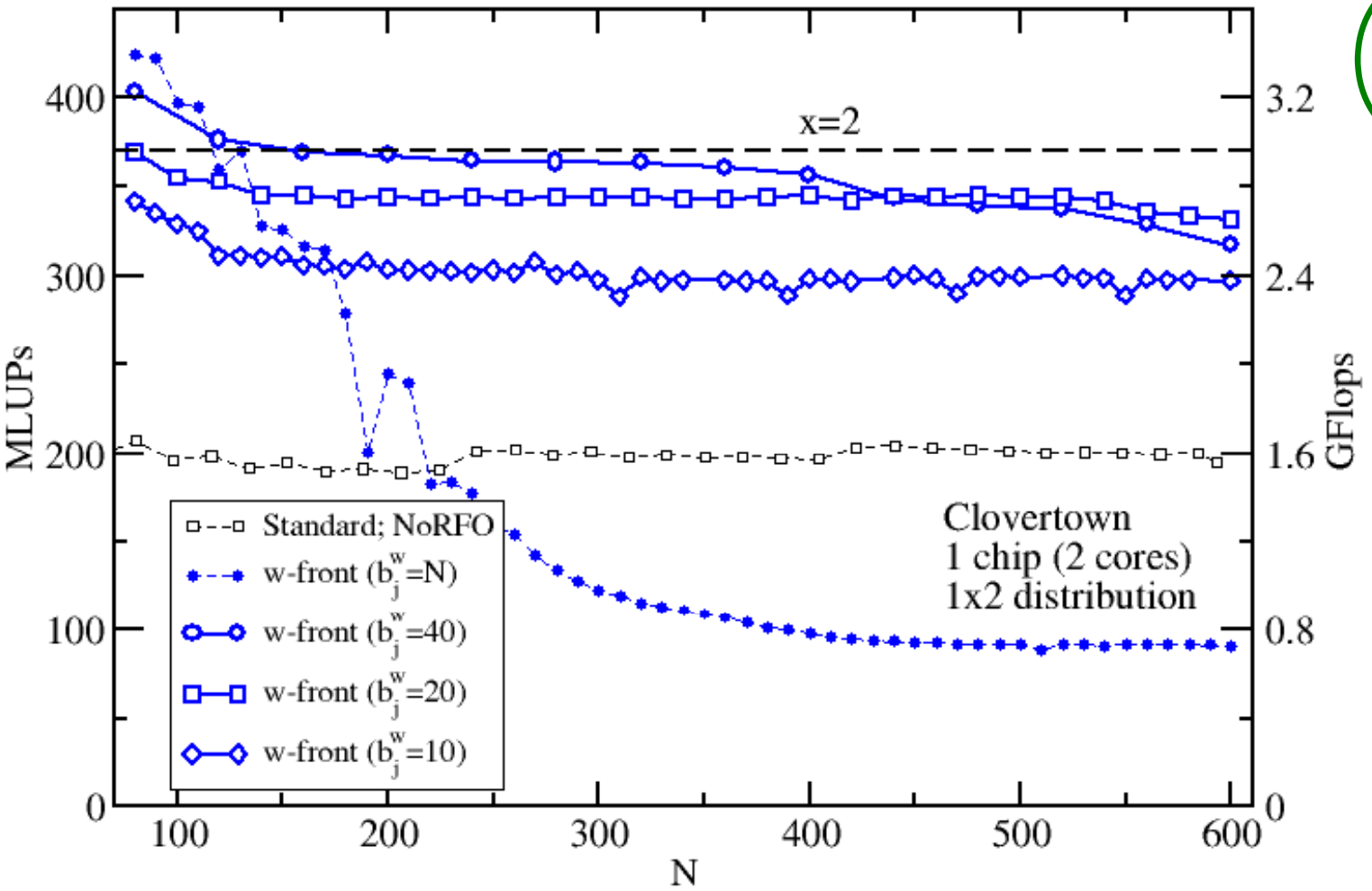
B_C and B_M are measured in saturation runs:

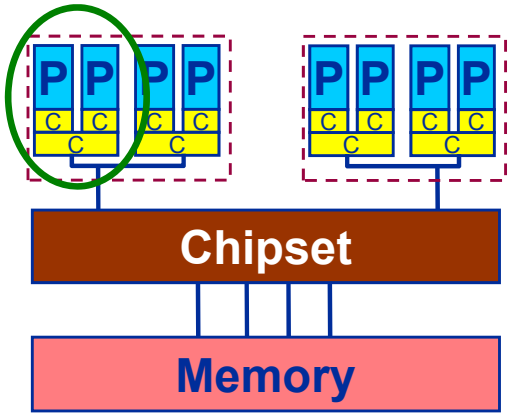
Clovertown: $B_M/B_C = 1/12$ → $S_W = 1.85$

Nehalem: $B_M/B_C = 1/4$ → $S_W = 1.6$

Jacobi solver

Wavefront parallelization: L2 group Clovertown



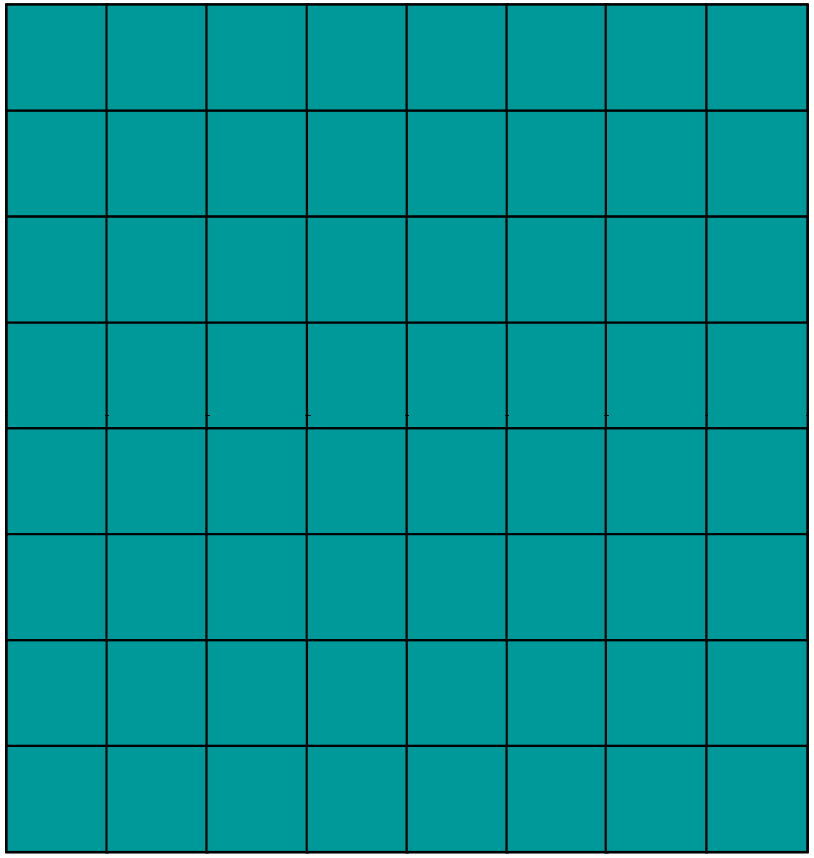


Implement blocking in j direction (b_j^w) to ensure that `tmp(:,:,0:3)` & 4 planes of x stay in cache!

→ Speedup: **~1.7-1.8x (prediction: S_W = 1.85)**

Jacobi solver

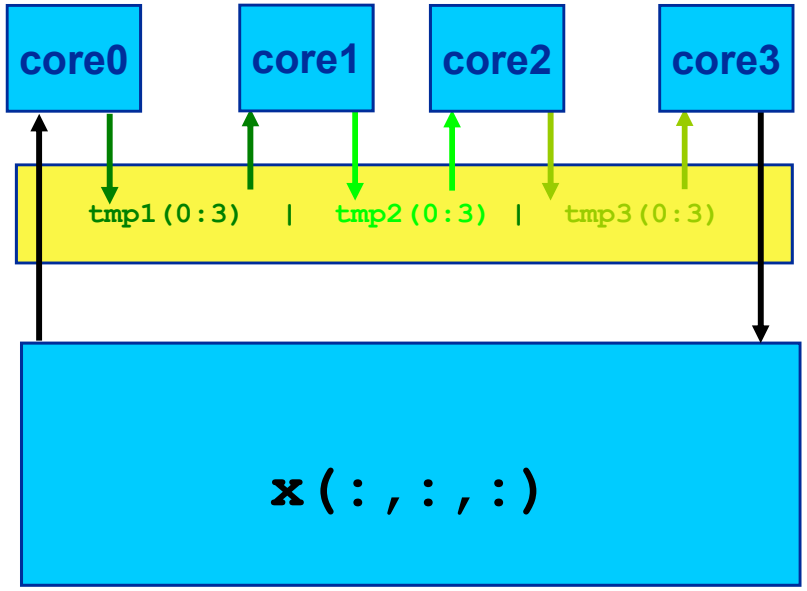
Propagating four wavefronts on native quadcores (1x4)



Running t_b wavefronts requires t_b temporary arrays `tmp` to be held in cache

Massive use of cache bandwidth!

t_b=4 → 1 x 4 distribution



Jacobi solver

Wavefront parallelization: New choices on native quad-cores

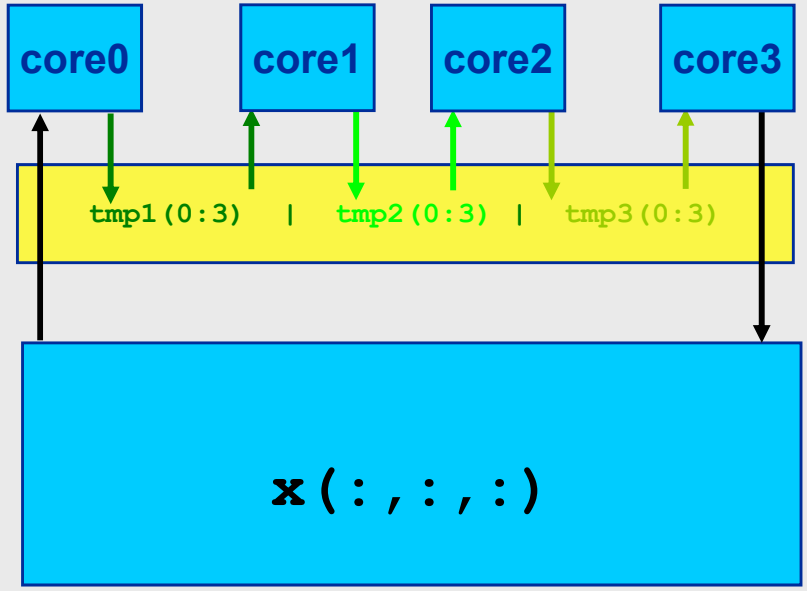
Thread 0: `x(:,:,k-1:k+1)`$_t$ → *tmp1(mod(k,4))*

Thread 1: *tmp1(mod(k-3,4):mod(k-1,4))* → *tmp2(mod(k-2,4))*

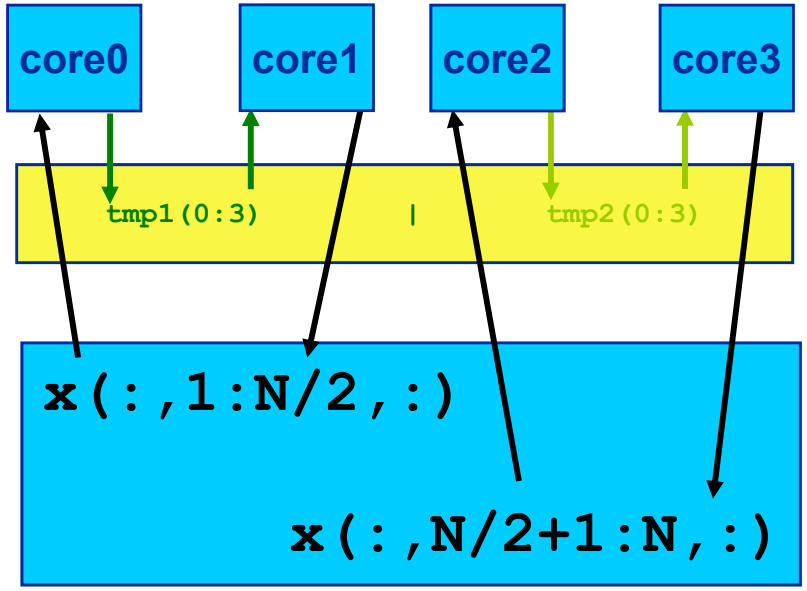
Thread 2: *tmp2(mod(k-5,4:mod(k-3,4))* → *tmp3(mod(k-4,4))*

Thread 3: *tmp3(mod(k-7,4):mod(k-5,4))* → `x(:,:,k-6)`$_{t+4}$

1 x 4 distribution

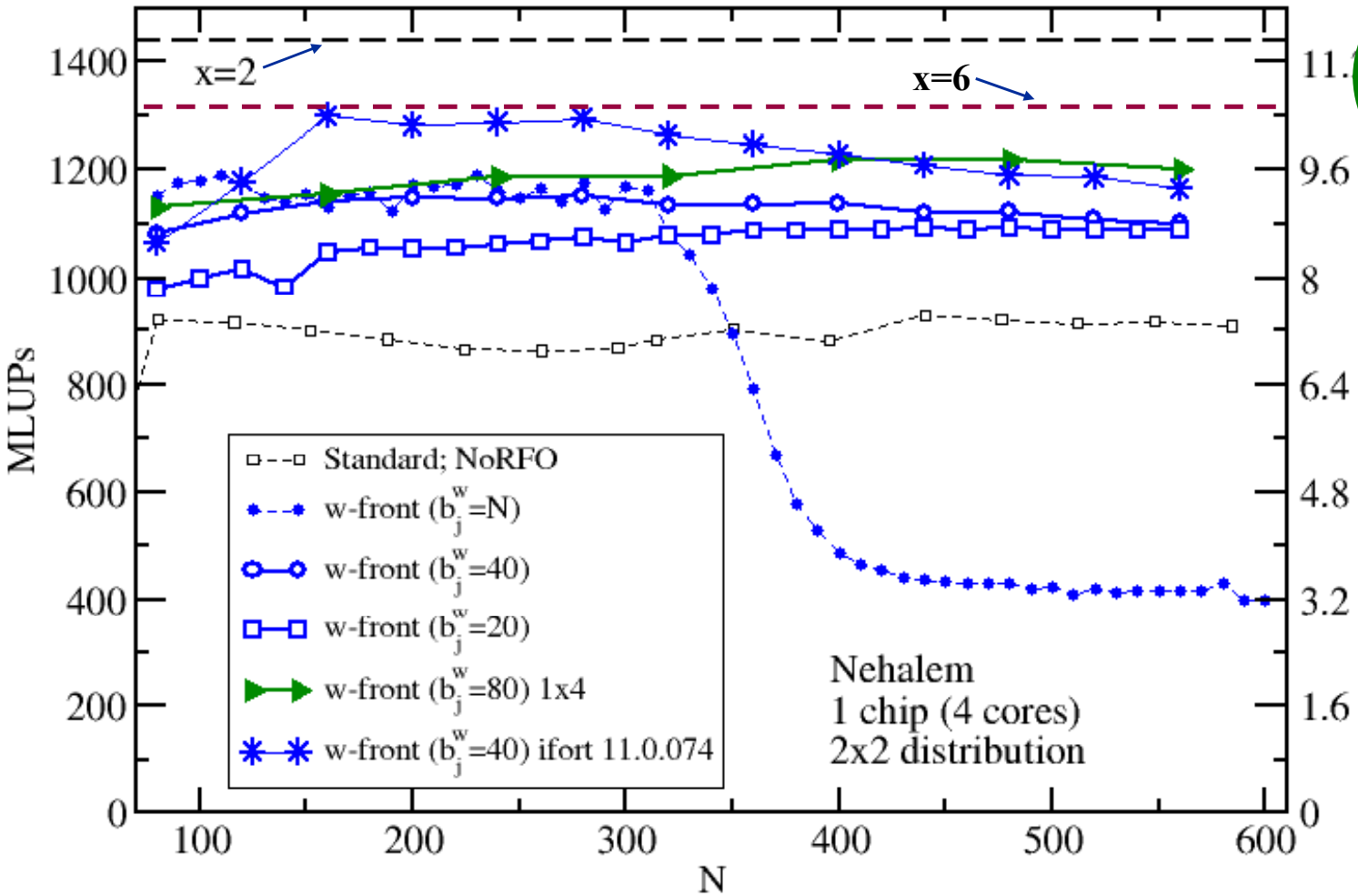


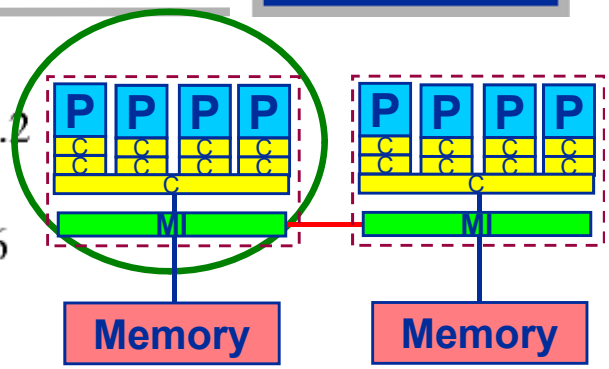
2 x 2 distribution



Jacobi solver

Wavefront parallelization: L3 group on Nehalem





400^3 bj=40	MLUPs
1 x 2	786
2 x 2	1230
1 x 4	1254

Performance model indicates some potential gain → new compiler tested

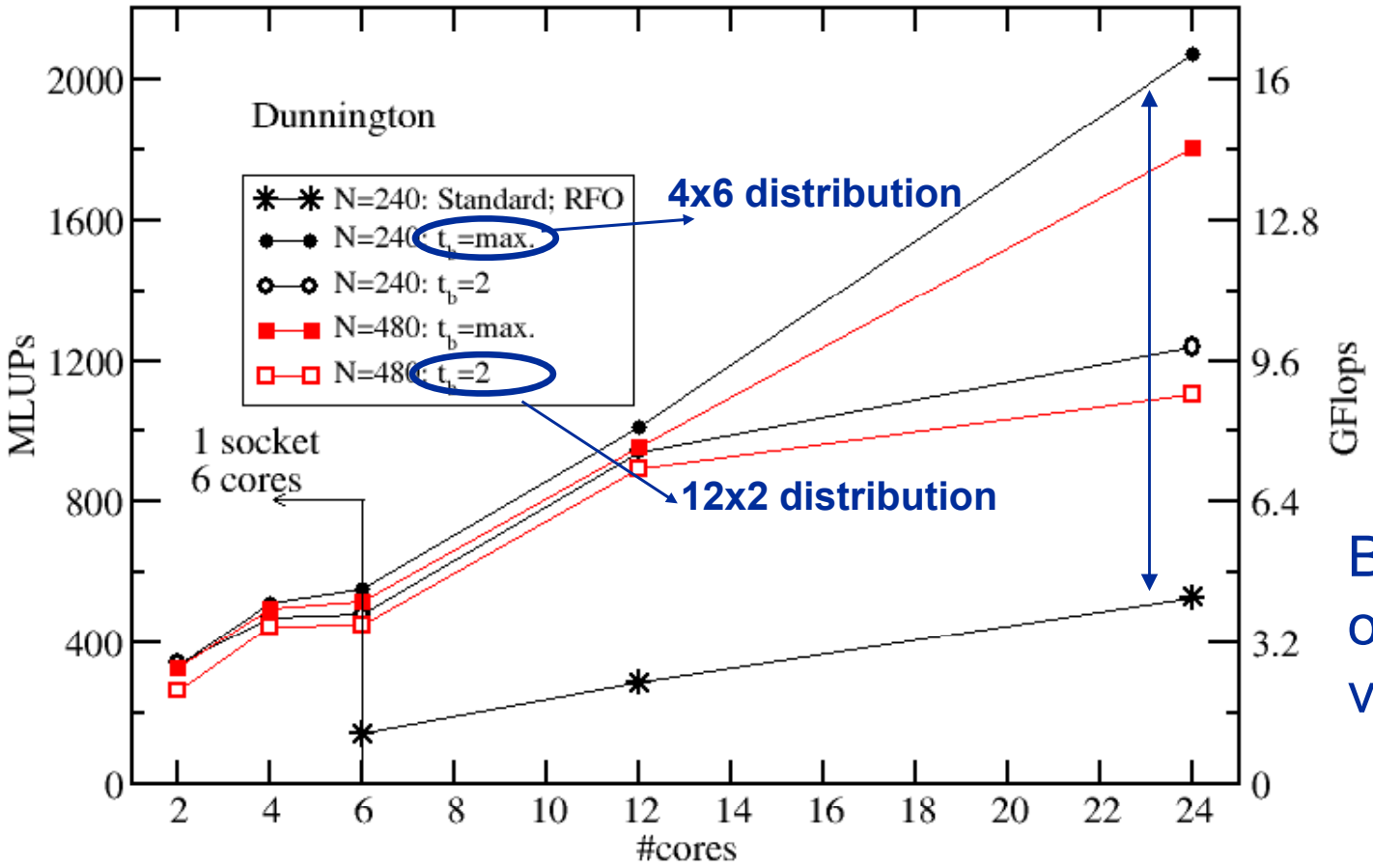
Only marginal benefit when using 4 wavefronts (Model: S_W=1.45):

↔ A single copy stream does only achieve about half max memory bandwidth on Nehalem

↔ L3 bandwidth becomes bottleneck

Jacobi solver

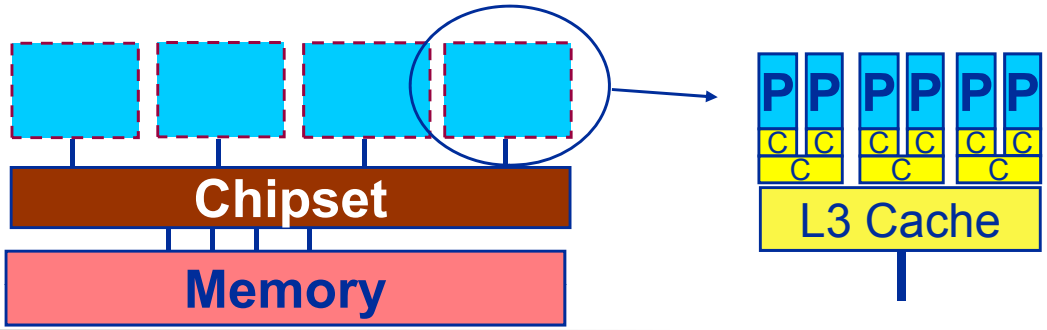
Wavefront parallelization: full Dunnington node



Speed-Up of WF:

3,…,4 x !

Bad performance of baseline NoRFO version!

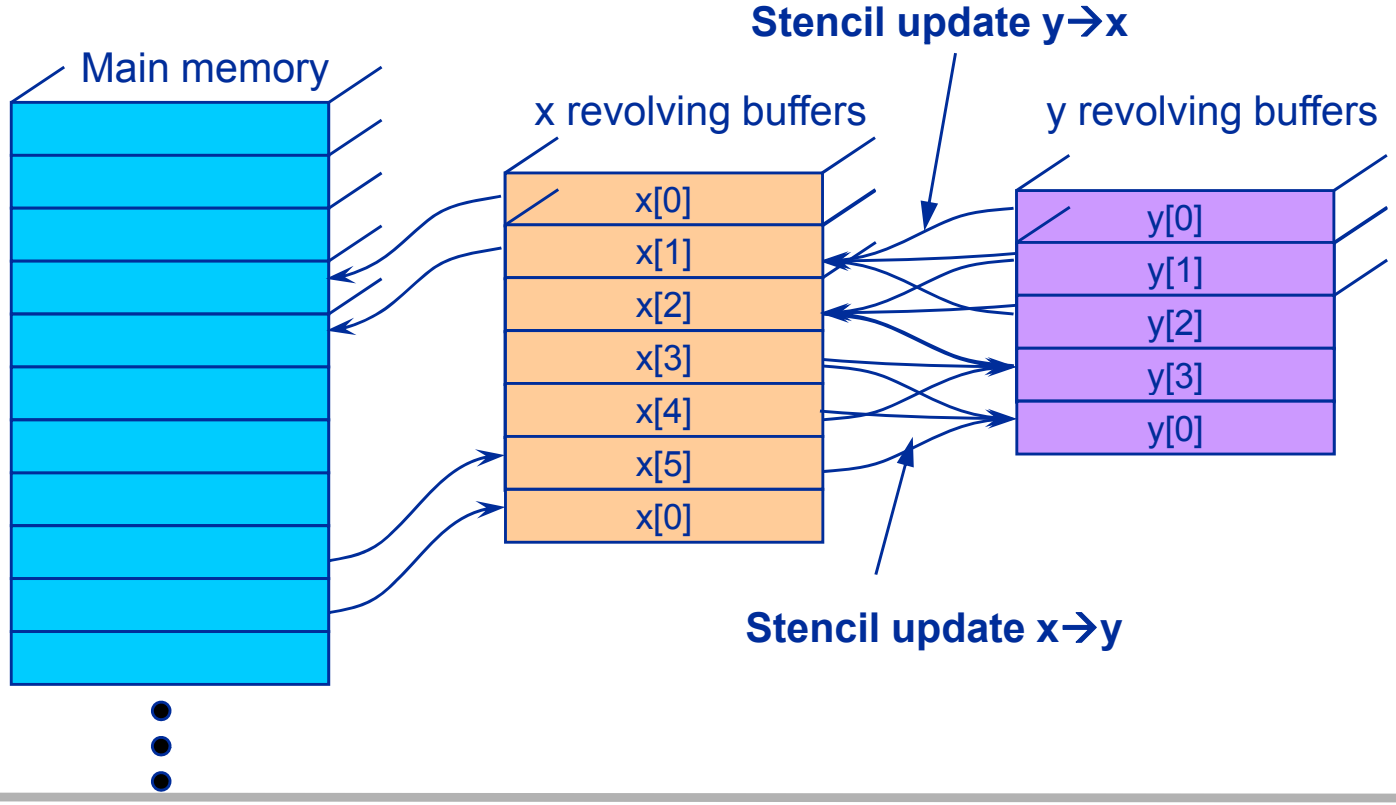


4 socket HexaCore system

Maximum number of useful wavefronts: t_b=6

How about the Cell?

- **Cell has no multiple cores per local store**
- **But:**
  - DMA engine enables overlap of computation and data transfer
  - One SPE can do multiple updates on a grid tile
  - Cell is bandwidth-starved: 0.03 W/F to main memory
- **"Revolving buffers" scheme (one SPE):**



How about the Cell?
Preliminary performance data

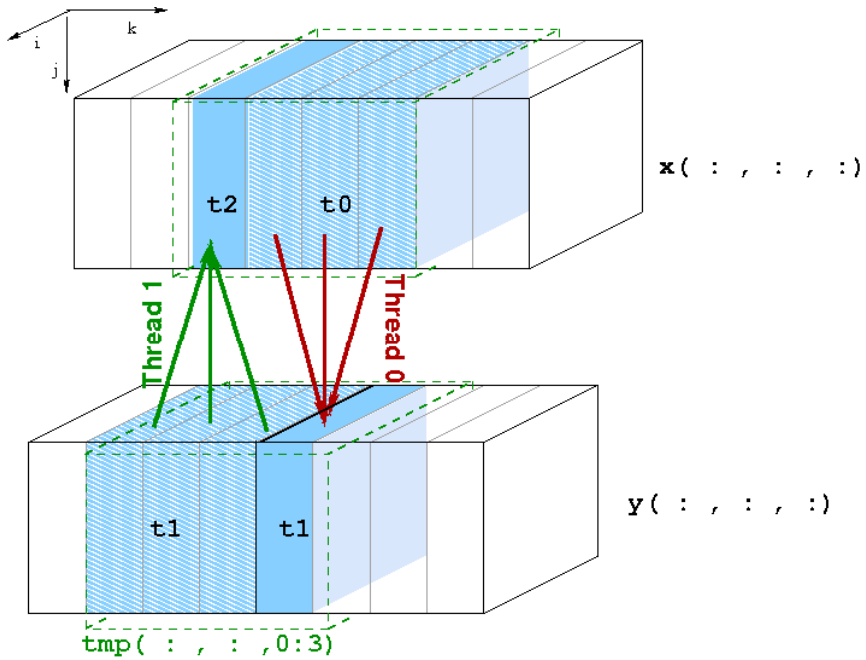
- **Baseline**
  - Datta et al. SC08: 1 GLUPs per Cell chip → 125 MLUPs per SPE
  - This amounts to a main memory bandwidth of 16 GB/s (which is the STREAM BW)

- "Wavefront" code
  - 1 SPE: 200-230 MLUPs (depending on geometry)
    This amounts to a LS bandwidth of 10-11 GB/s (5 LD, 1 ST)
    and a main memory BW of 1.6-1.8 GB/s

  - 8 SPEs: Eight times that, i.e. 13-15 GB/s
  - 16 SPEs on one memory interface: 130 MLUPs per SPE, i.e. 16.6 GB/s

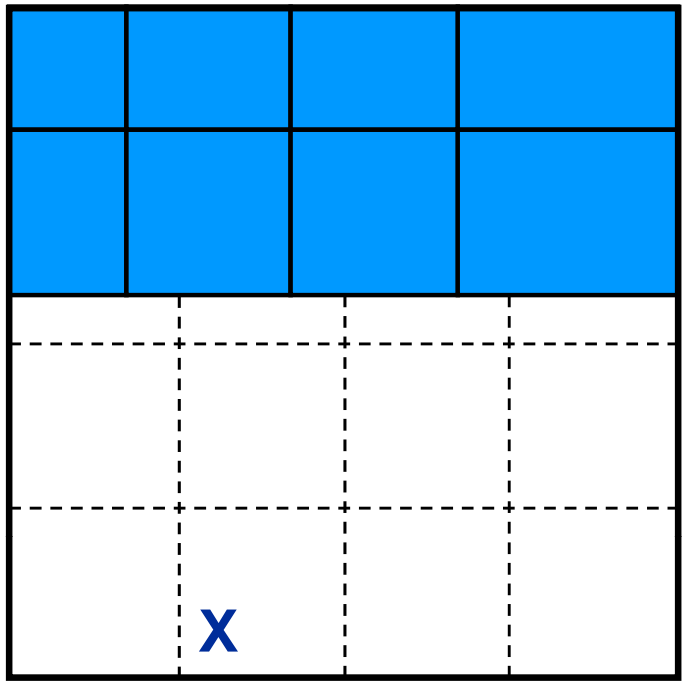
  - So the code is compute bound (but only just barely) on one chip
  - Some optimization potential is still there (SW pipelining, interleaving of updates)

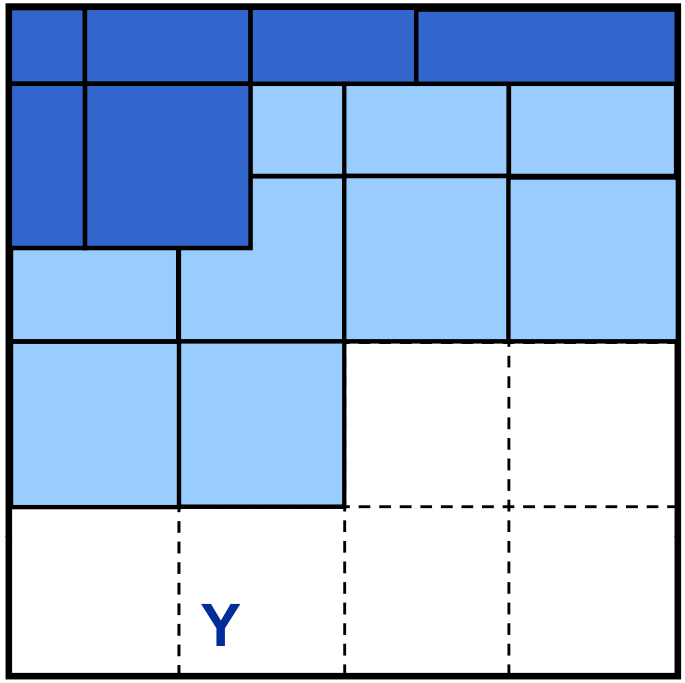
Temporal blocking by pipelining: Some further thoughts

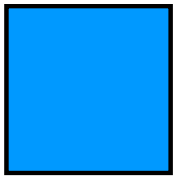


Pipelined temporal blocking

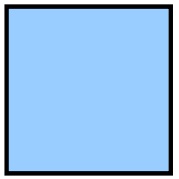
The other approach



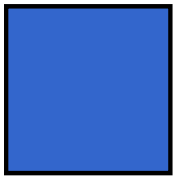




thread 1
$t_1 \rightarrow t_2$



thread 0
$t_0 \rightarrow t_1$



thread 2
$t_2 \rightarrow t_3$

One long pipeline (all cores of a node) advances through the lattice, each update is shifted by (-1,-1,-1)

Advantages

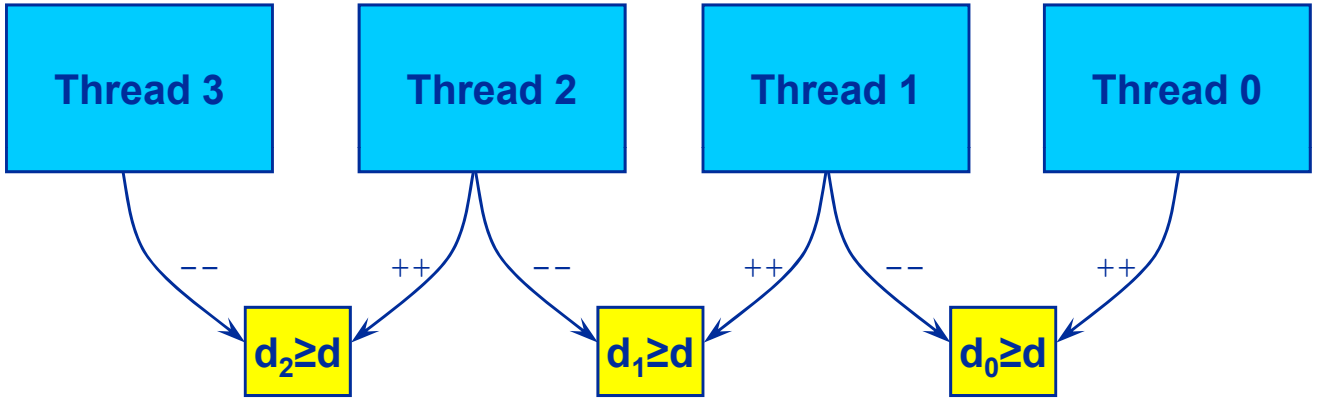
- **Freestyle spatial blocking**
- **No explicit boundary copies**
- **Multiple updates per core**

Drawbacks

- **Elimination of y array more complex**
- **Shift reduces cache reuse**
- **Huge parameter space**

More things to consider

- **Thread synchronization after tile update can be costly**
  - If tiles are small or number of threads is large
  - OpenMP: Barrier overhead ≈ 1µs per socket (Intel compiler, Core2)
  - Solution: Relaxed pairwise synchronization between threads

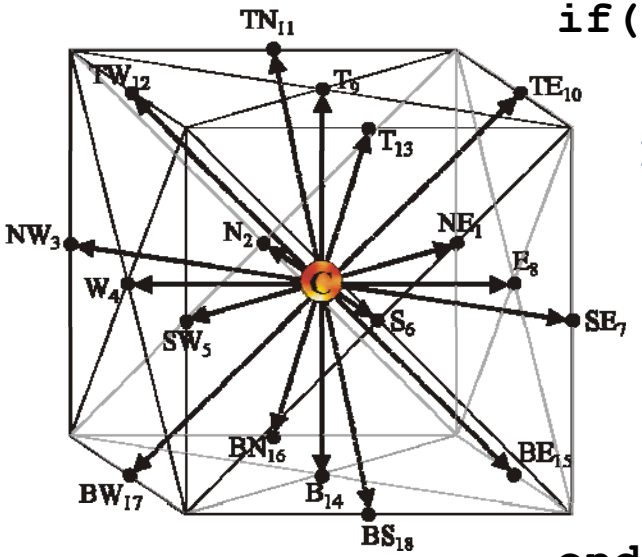


- **Distributed-memory parallelization**
  - Hybrid OpenMP/MPI code
  - Multi-layer halo required
  - Commmunication vs. computation tradeoffs
  - Work in progress

Even more…

- **Can this be done with Lattice-Boltzmann as well?**

```
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 (≈200 flops)
            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
```

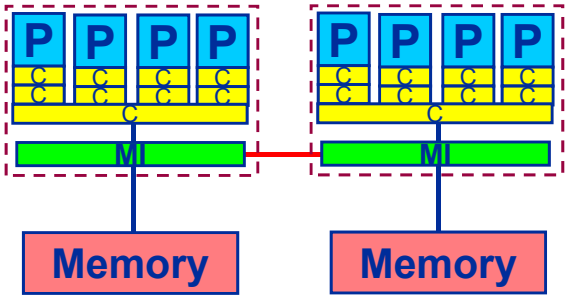


2*19 cache lines are touched for a single cell update, but accessed contiguously

Alignment challenge for GPUs, Cell, …

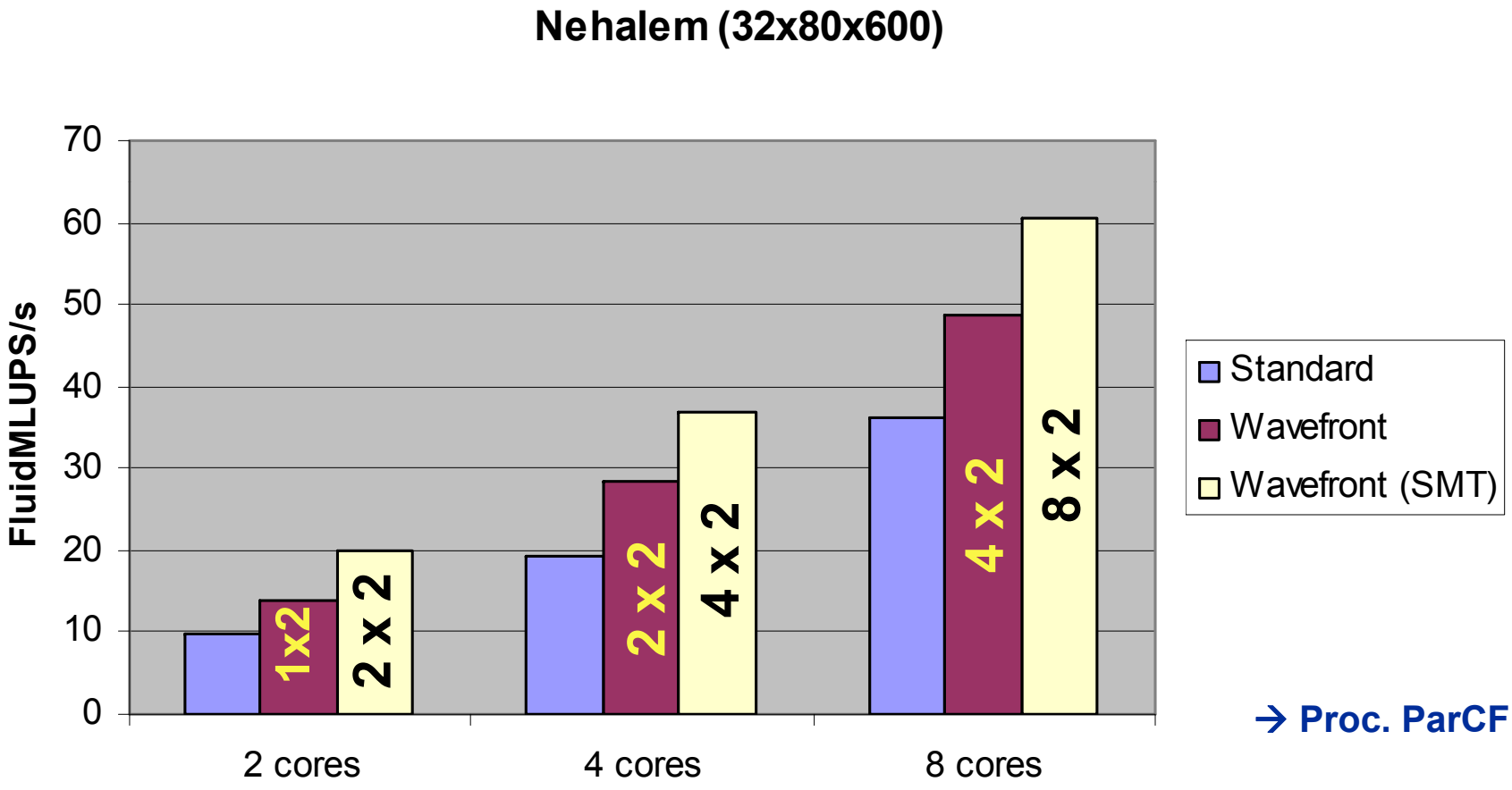
LBM wavefront blocking

Scalability on Nehalem: Standard vs. wavefront



2 wavefronts with 1, 2 and 4 groups

SMT with 2,4 and 8 groups helps a lot!



→ **Proc. ParCFD 2009**

Temporal blocking by pipelined parallelization:
Status

- **No in-cache optimizations (alignment, SIMD) implemented for wavefront blocking of Jacobi and LBM so far**
  - … but for “the other approach” – results are promising
- **Pipelined parallelization for stencil codes beneficial if**
  - Multi-Core Chip is bandwidth starved (one core can sustain main memory bandwidth)
  - Large shared (on-chip) cache is available
  - Benefit of SMT strongly dependent on computational kernel (good for LBM)
- **Easy to implement and parallelize but hybrid approach is required if used in a larger application, e.g. as a smoother in MG**
- **First tests with Gauss-Seidel kernel underway**

THANK YOU

Backup →

- **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)

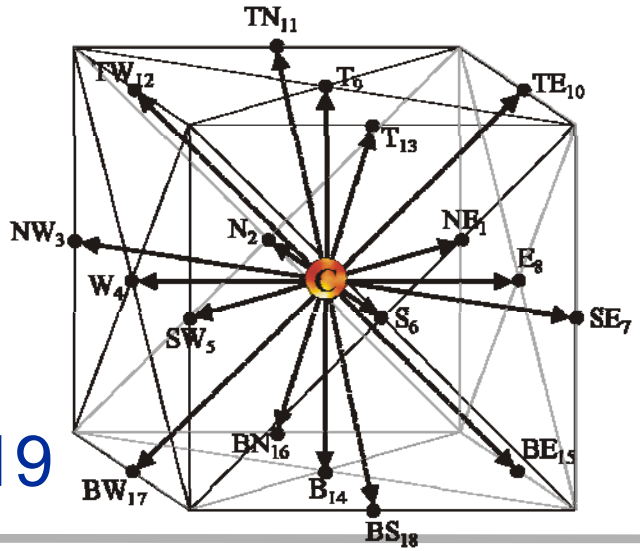
$$\partial_t f_\alpha + \xi_\alpha \cdot \nabla f_\alpha = -\frac{1}{\lambda}[f_\alpha - f_\alpha^{(eq)}]$$

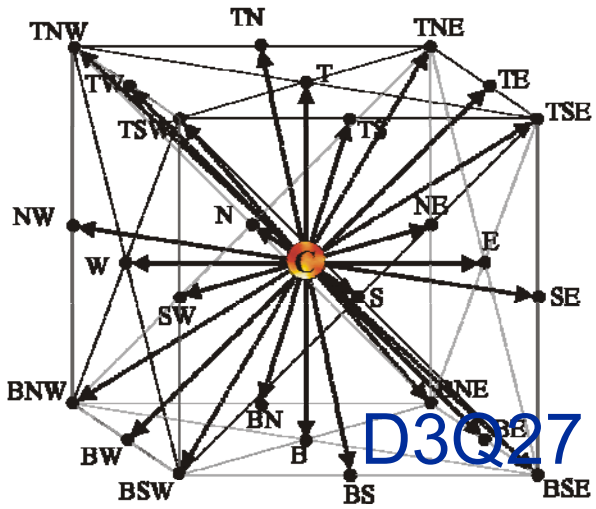
$$f_\alpha(\vec{x},t) = f(\vec{x},\xi_\alpha,\mathrm{t})$$
$$f_\alpha^{(eq)}(\vec{x},t) = f^{(0)}(\vec{x},\xi_\alpha,\mathrm{t})$$

ξ_α :

discretization scheme

D3Q19





D3Q27

Lattice Boltzmann method

A simple but efficient kernel 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)

Lattice Boltzmann method

A simple performance model for the standard implementation

- Standard performance measure for LBM:
  Million fluid cell updates per second: MFLUPS/s

- MFLOP/s is not a good idea: 150 – 400 FLOP for kernel:
  - Implementation
  - Compilerversion & Compileroptions
  - Divide?!

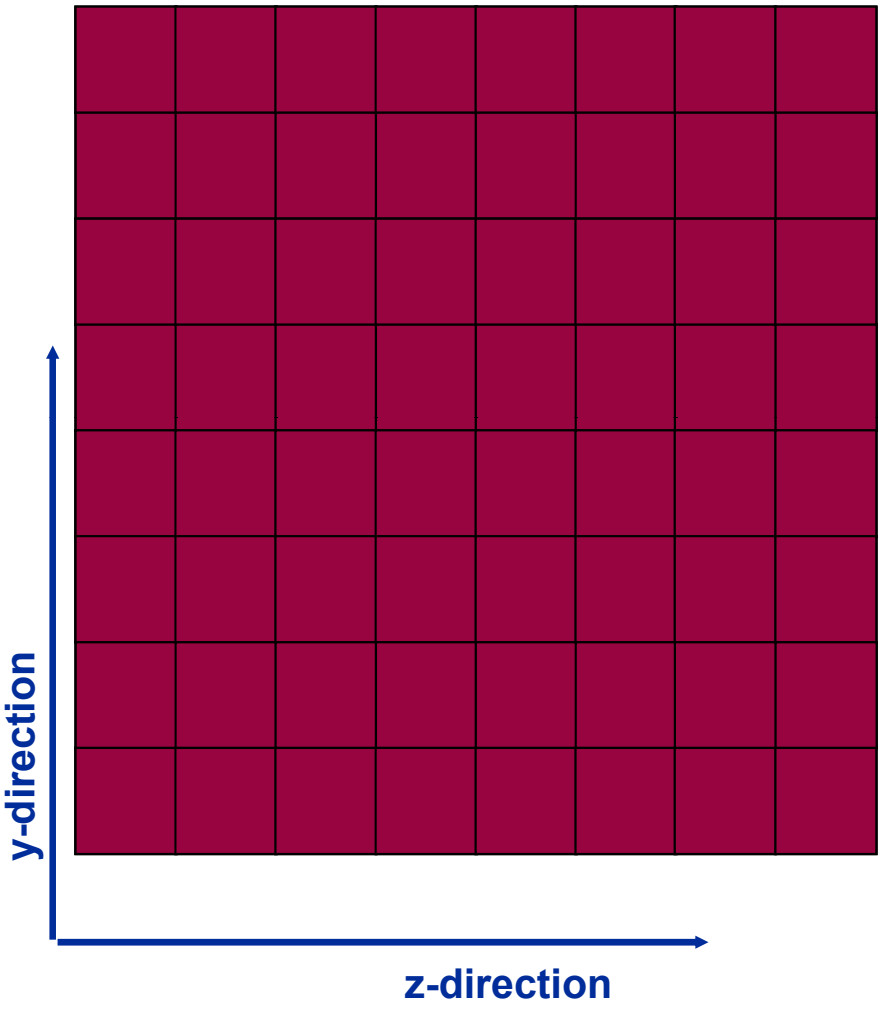
- Estimate maximum performance on basis of attainable main memory bandwidth (MBW):
  - Data transfers / FLUP = 3*19*8Byte = 456 Byte

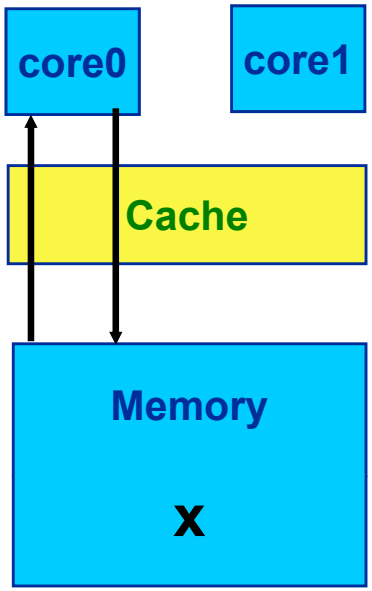
  - Performance estimate [MFLUPS/s]: MBW [MByte/s] / 456 Byte

  - MBW is determined through low level kernel, e.g. stream

LBM & multicore architectures

Standard sequential implementation

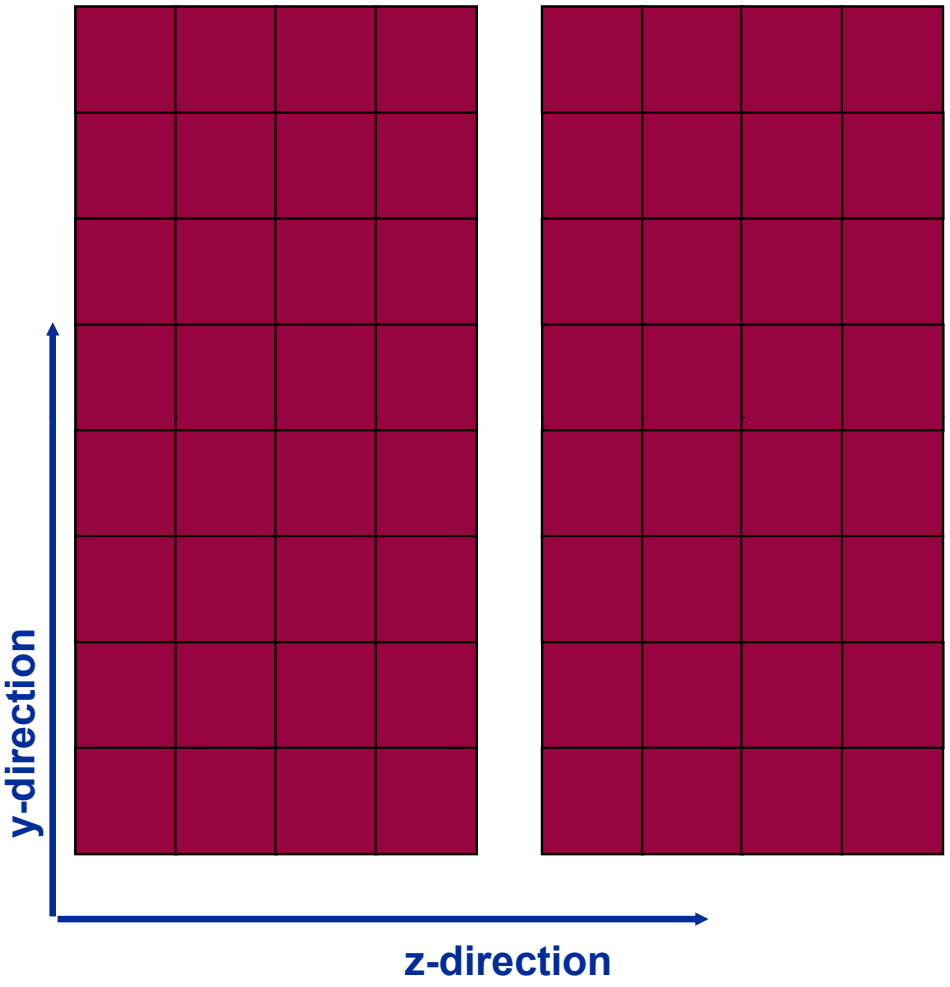


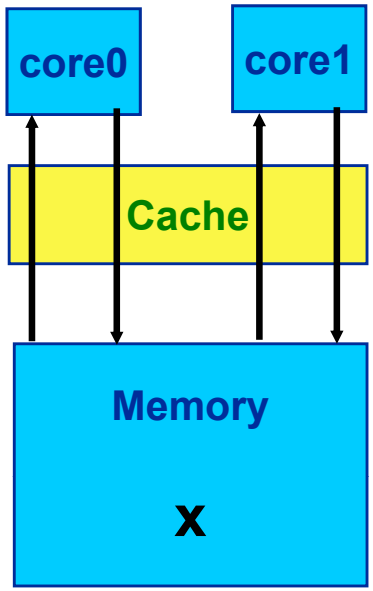



```
do z=1,zMax
   do y=1,yMax
      do x=1,xMax
         if( fluidcell(x,y,z) ) then
         endif
      enddo
   enddo
enddo
```

LBM & multicore architectures

Standard naive shared memory parallel

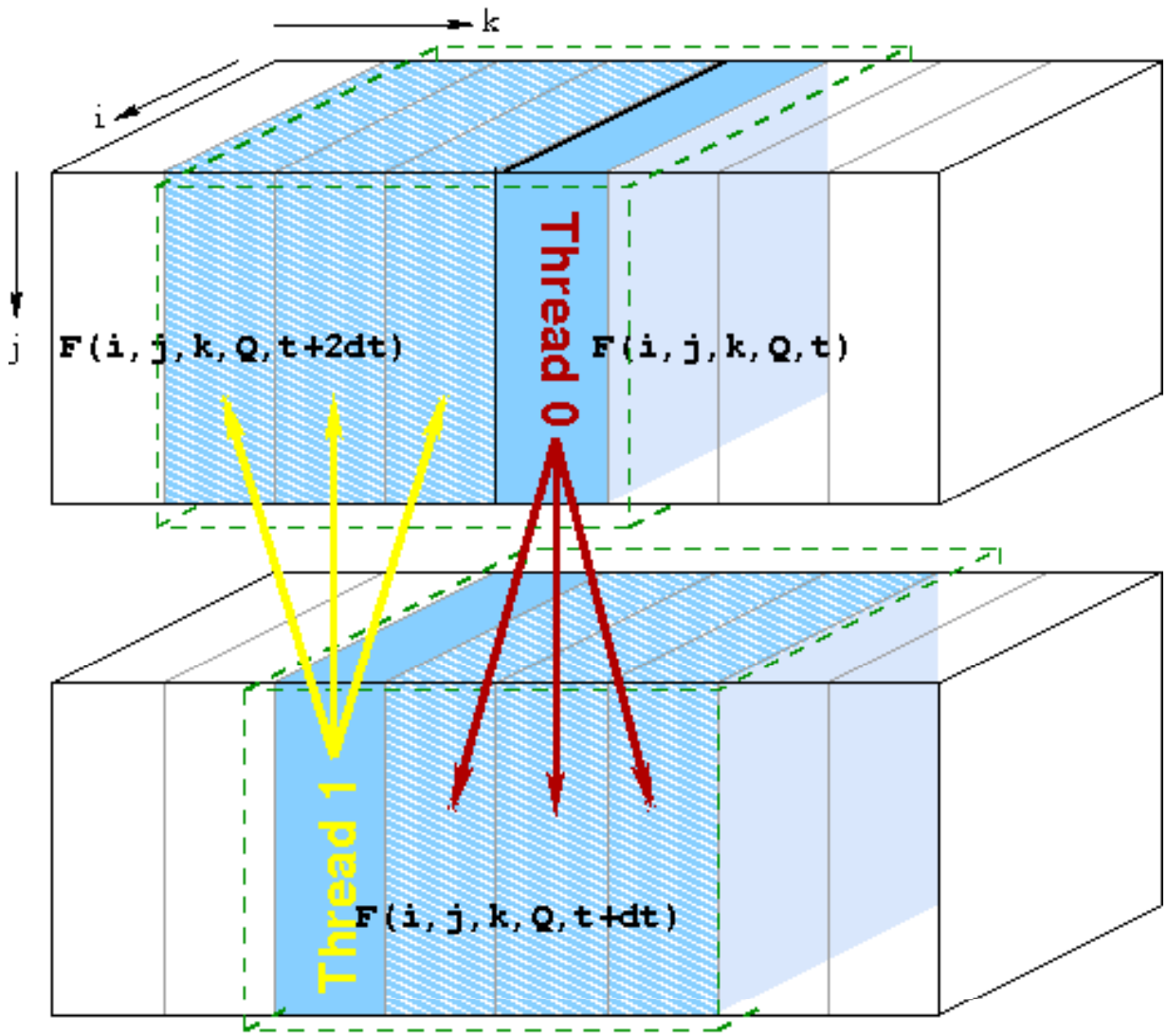





```
!$OMP PARALLEL DO private(…)
do z=1,zMax
   do y=1,yMax
      do x=1,xMax
         if( fluidcell(x,y,z) ) then
         endif
      enddo
   enddo
enddo
```

LBM & mutlicore architectures

A different parallel approach: Propagating wavefronts!



k ←→ z

j ←→ y

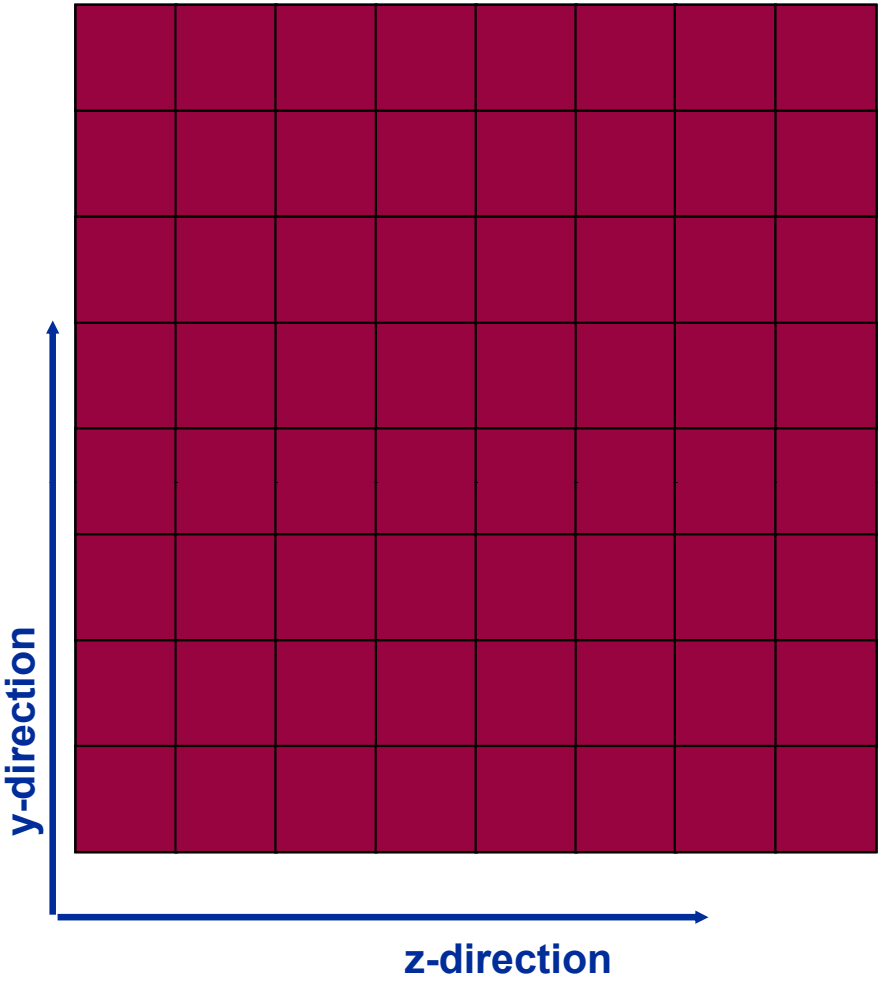
i ←→ x

core0: $\mathbf{F(:,:,z-1:z+1,0:18,1)_t}$ → *tmp(:,:,0:18mod(k,4))*

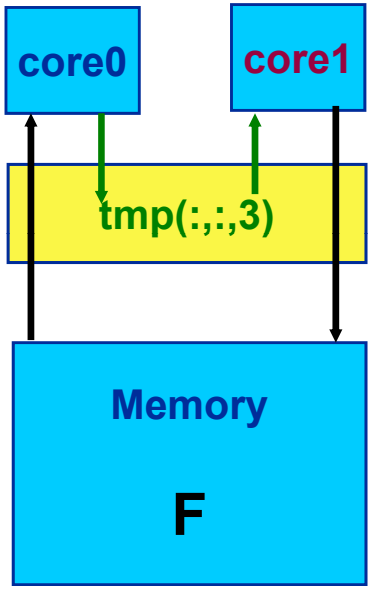
core1: *tmp(:,:,0:18,mod(z-3,4):mod(z-1,4))* → $\mathbf{F(:,:,z-2,0:18,1)_{t+2}}$

LBM & mutlicore architectures

Propagating two wavefronts!



`F(:,:,:,:,1)` is obsolete!



Use small buffer `tmp(:,:,0:18, 0:3)` which fits into the cache

Save main memory data transfers for `F(,1)` !

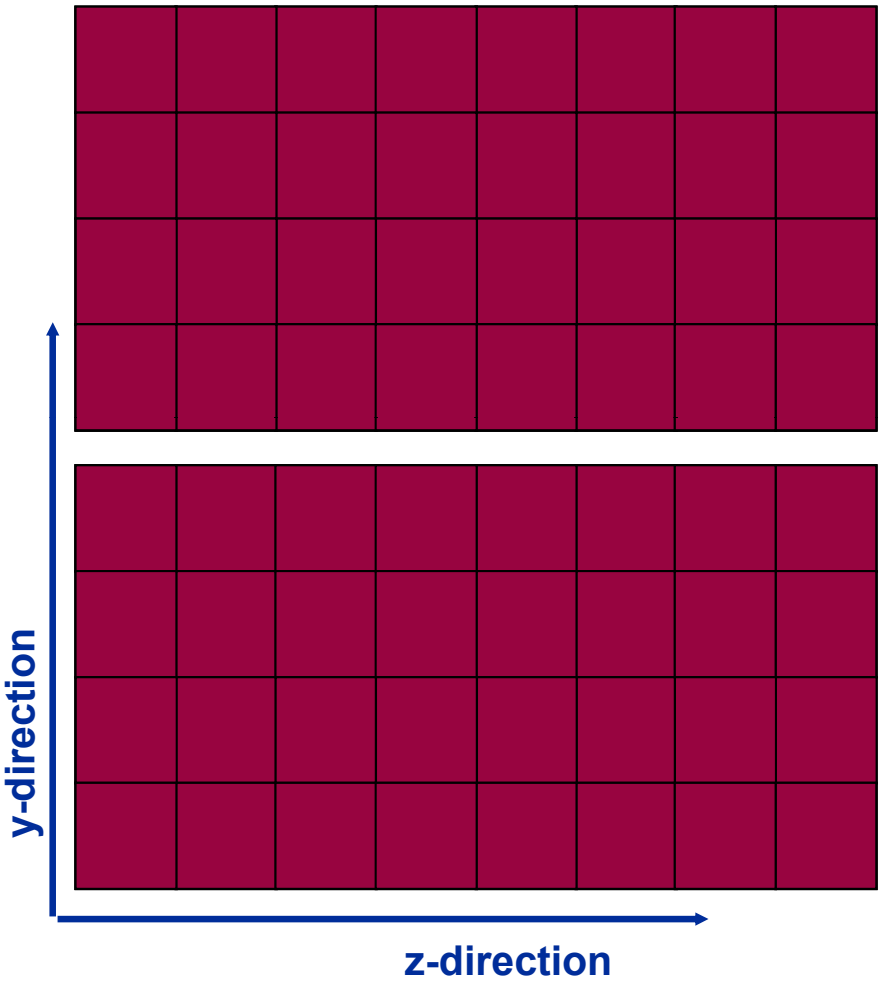
Sync threads/cores after each z-iteration

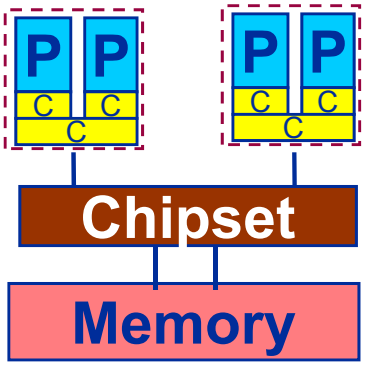
core0: $\mathbf{F(:,:,z-1:z+1,0:18,0)}_{t}$ → *tmp(:,:,0:18,mod(k,4))*

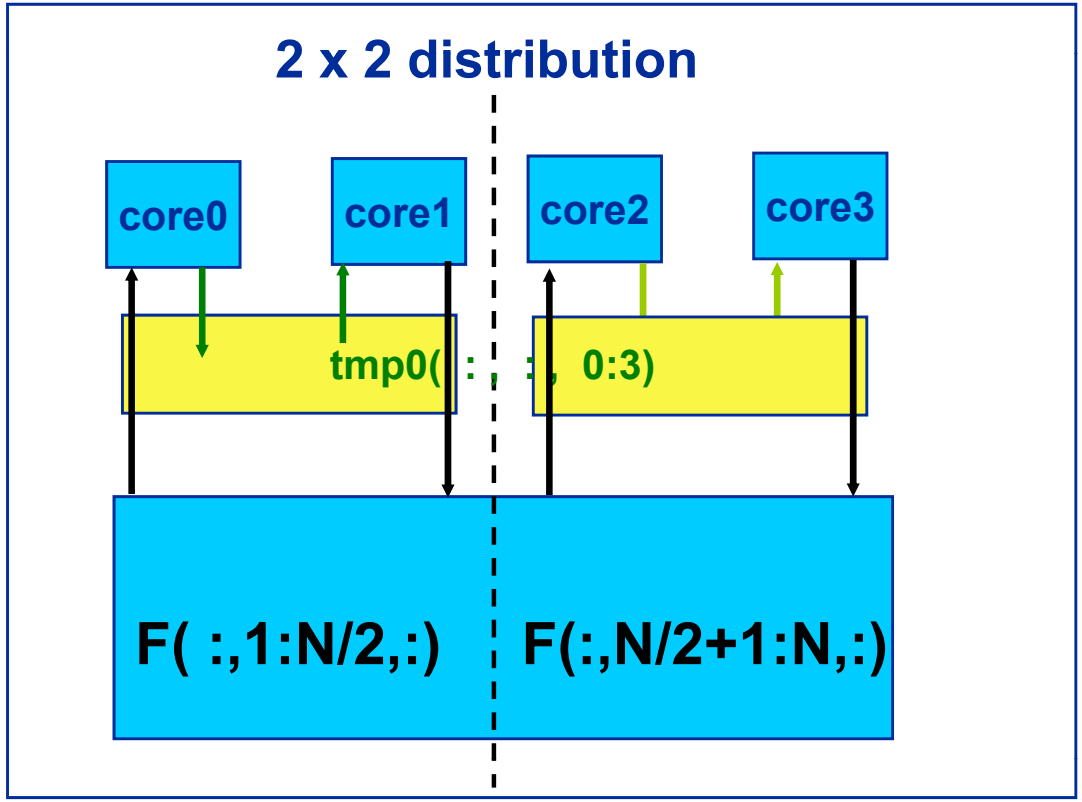
core1: *tmp(:,:,0:18,mod(z-3,4):mod(z-1,4))* → $\mathbf{F(:,:,z-2,0:18,0)}_{t+2}$

LBM & multicore architectures

Propagating two groups of wavefronts (2x2)

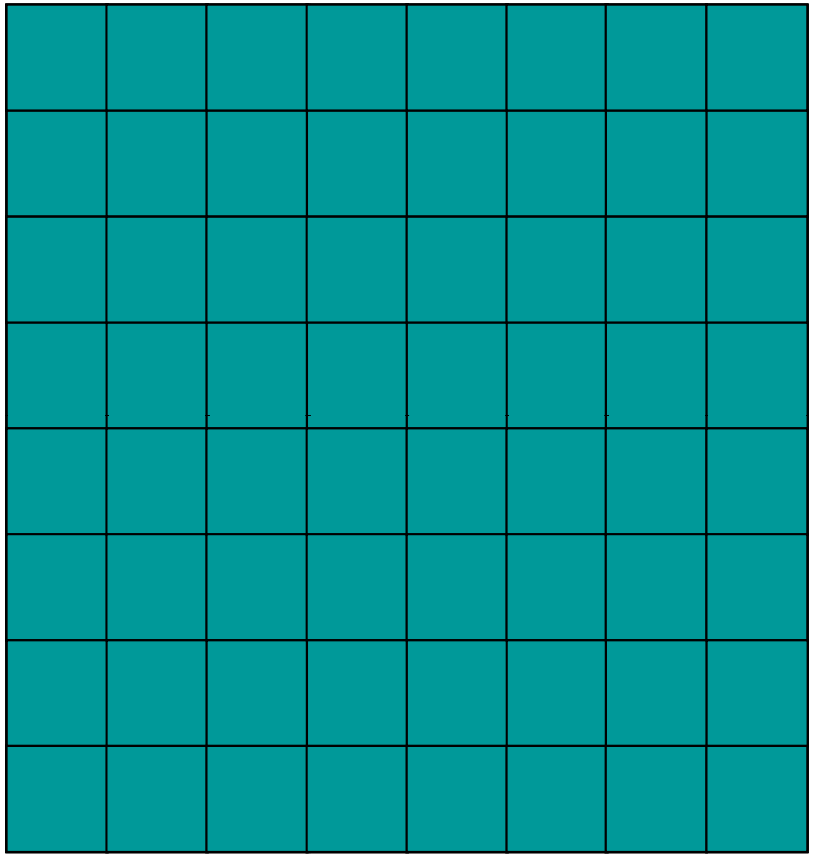






LBM & multicore architectures

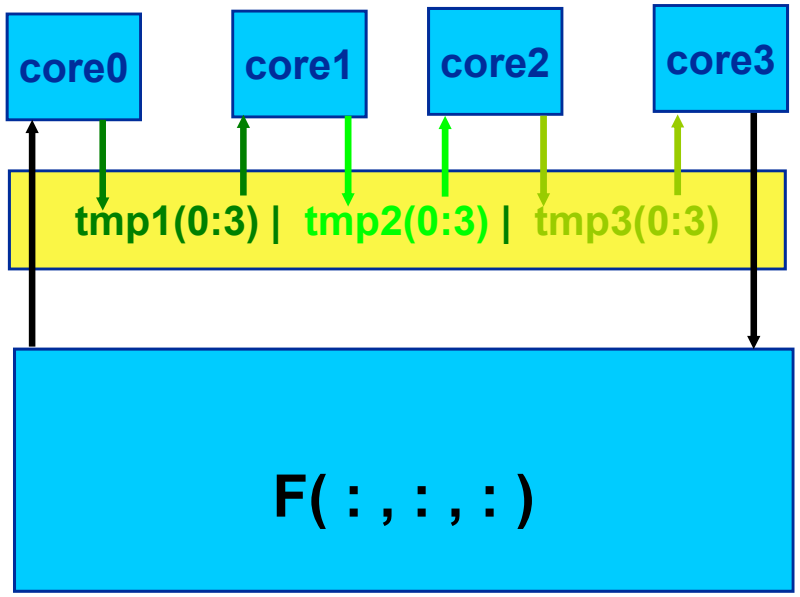
Propagating four wavefronts on native quadcores (1x4)



Running `tb` wavefronts requires `tb-1` temporary arrays `tmp` to be held in cache!

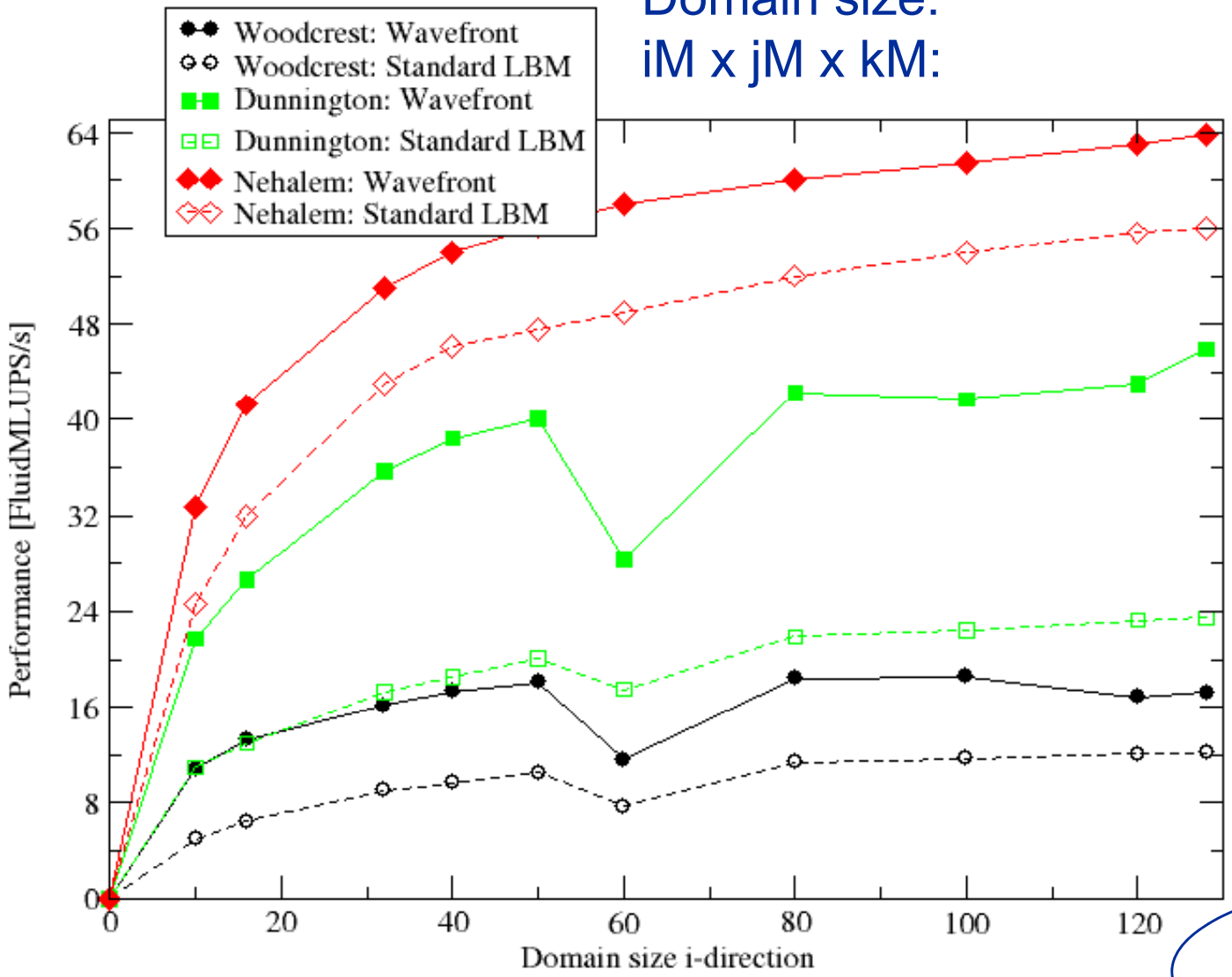
Extensive use of cache bandwidth!

1 x 4 distribution



LBM & multicore architectures

Full node results with 2 wavefronts



iM

Domain size:
iM x jM x kM:

kM=600

jM = 32 (Woodcrest),

jM = 48 (Dunnington),

jM = 64 (Nehalem)

Cache size needs to be larger than:

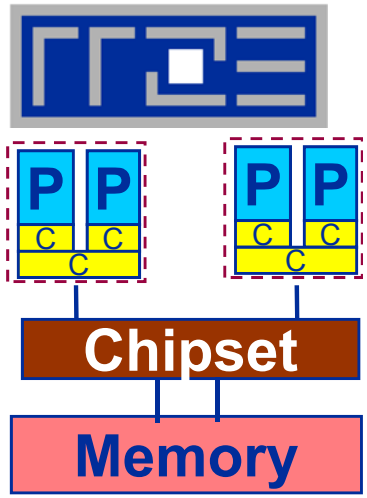
8 * 19 * iM*jM* 8 Bytes

(= 4.7 MB for iM=120; jM=32)

Spatial blocking in j-direction required!

LBM & multicore architectures

Scalability on Woodcrest: Standard vs. wavefront



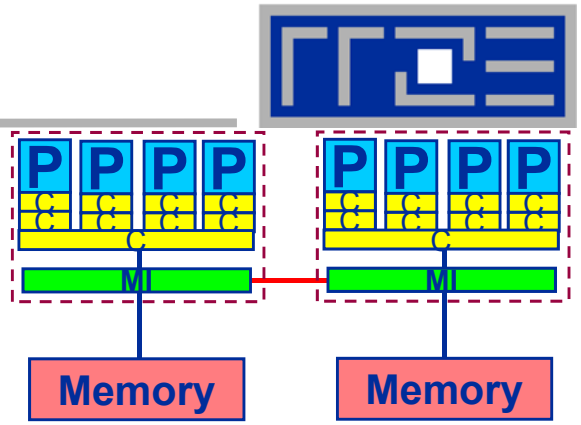
2 wavefronts in 1 and 2 groups

Speed-up: 1.7 x

Domain: 80x32x600

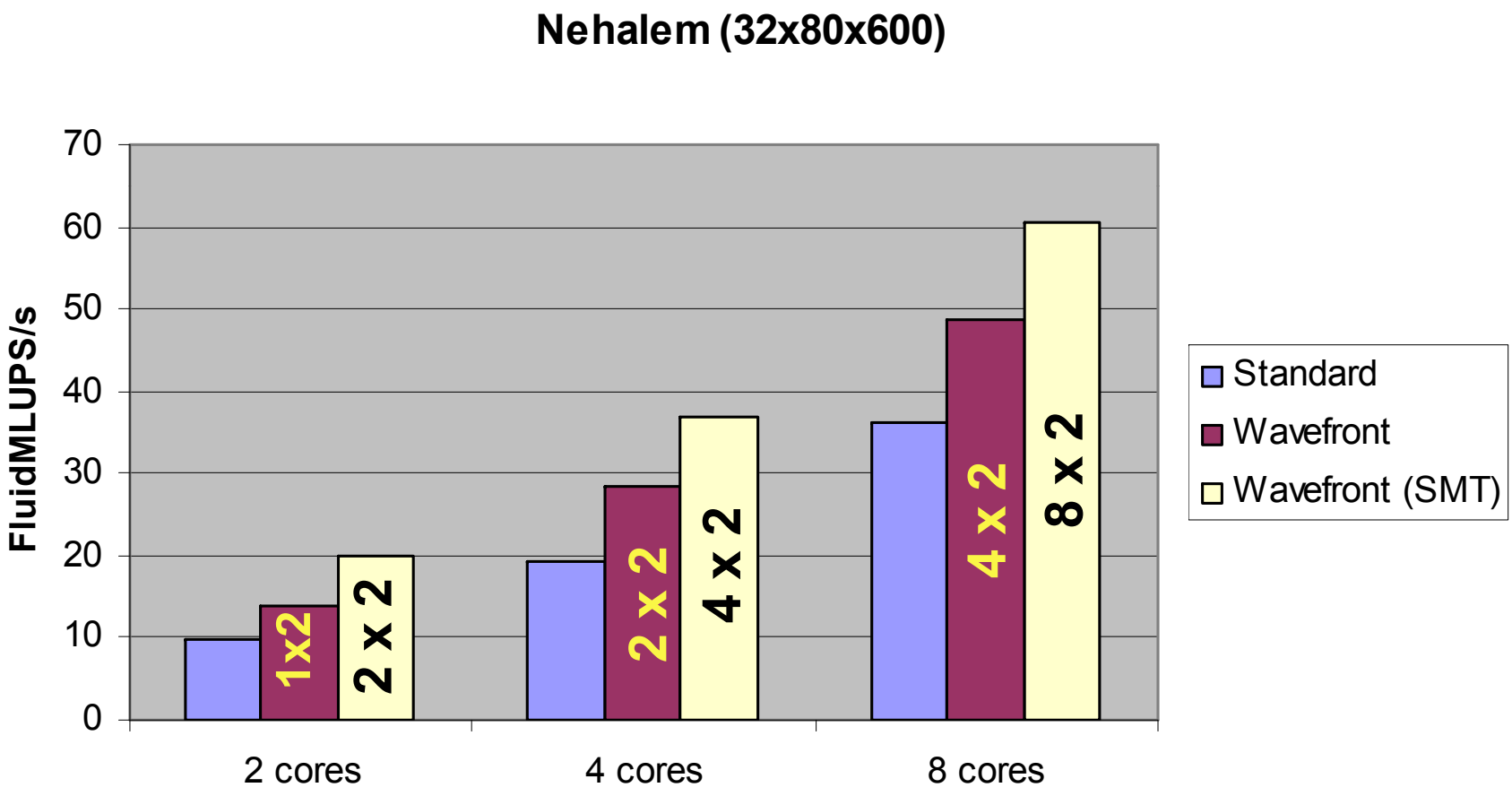
LBM & multicore architectures

Scalability on Nehalem: Standard vs. wavefront



2 wavefronts with 1, 2 and 4 groups

SMT with 2,4 and 8 groups helps a lot!



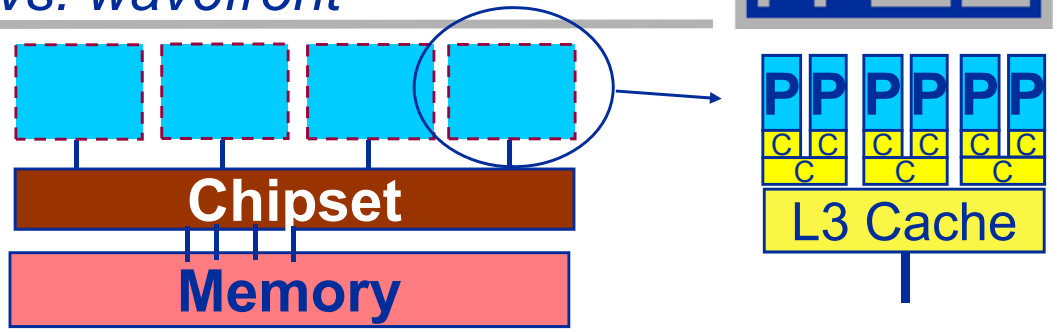
LBM & multicore architectures

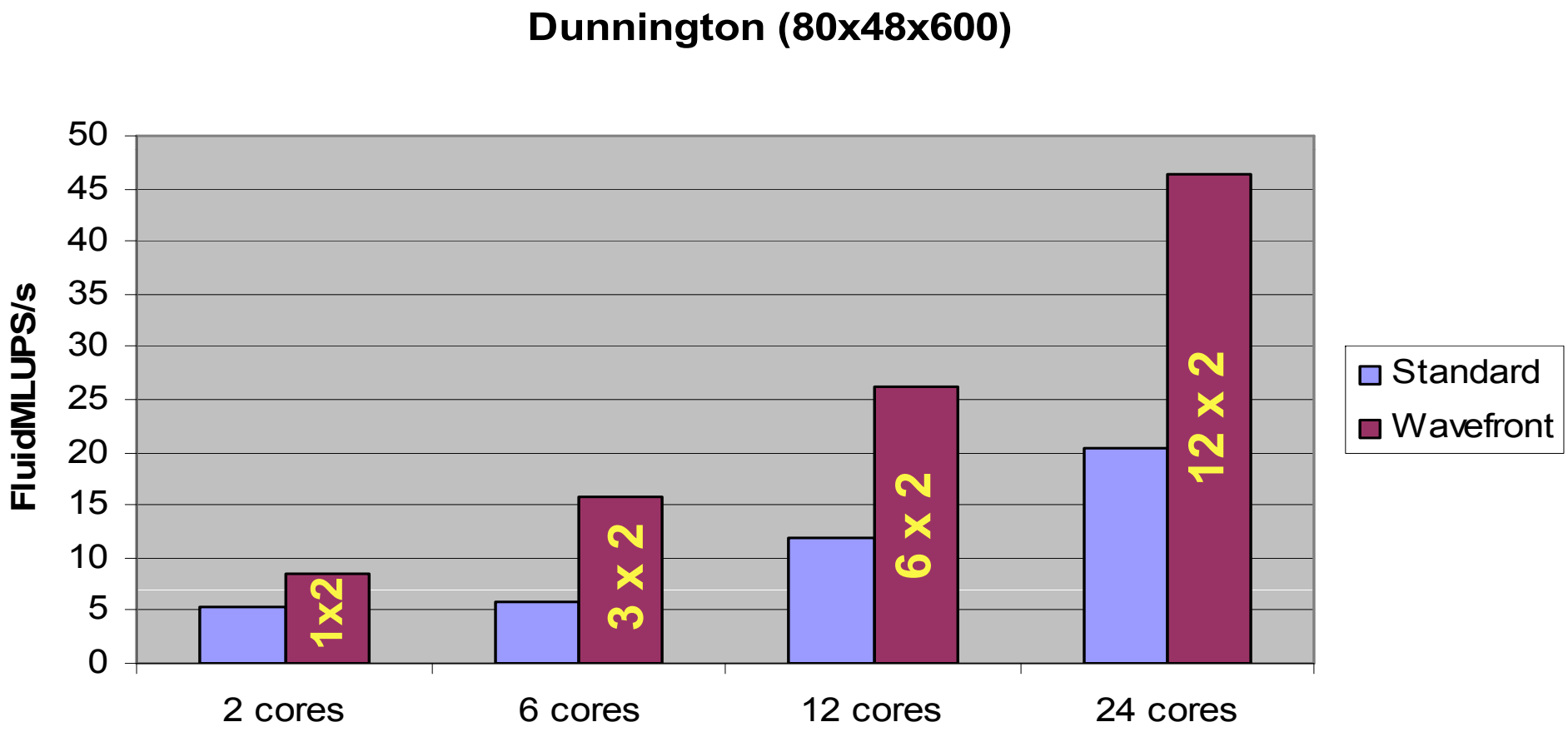
Scalability on Dunnington: Standard vs. wavefront

2 wavefronts with

1, 3, 6 and 12 domains

2.2x speed-up





LBM & wavefront approach
Status & Outlook

- To do:
  - Blocking in j-/y-direction in progress
  - No in-cache optimizations implemented / explored so far
  - Perfomance model

- Wavefront parallelization for LBM beneficial if
  - Multi-Core Chip is bandwidth starved
    (one core can sustain main memory bandwidth)
  - Large shared (on-chip) cache is available

- Can easily be implemented for other stencil based methods:
  - Jacobi solver/smoother → COMPSAC2009
  - Gauß-Seidel → first tests

- Easy to implement and parallelize but **hybrid MPI/OpenMP** approach is required if used in a massively parallel production code.