

# EWOMP'02

●●● Roma 18-20 settembre 2002

## Achieving high performance in a LBM code using OpenMP

**Federico Massaioli**

[f.massaioli@caspur.it](mailto:f.massaioli@caspur.it)

**Giorgio Amati**

[g.amati@caspur.it](mailto:g.amati@caspur.it)



# Scope of the work

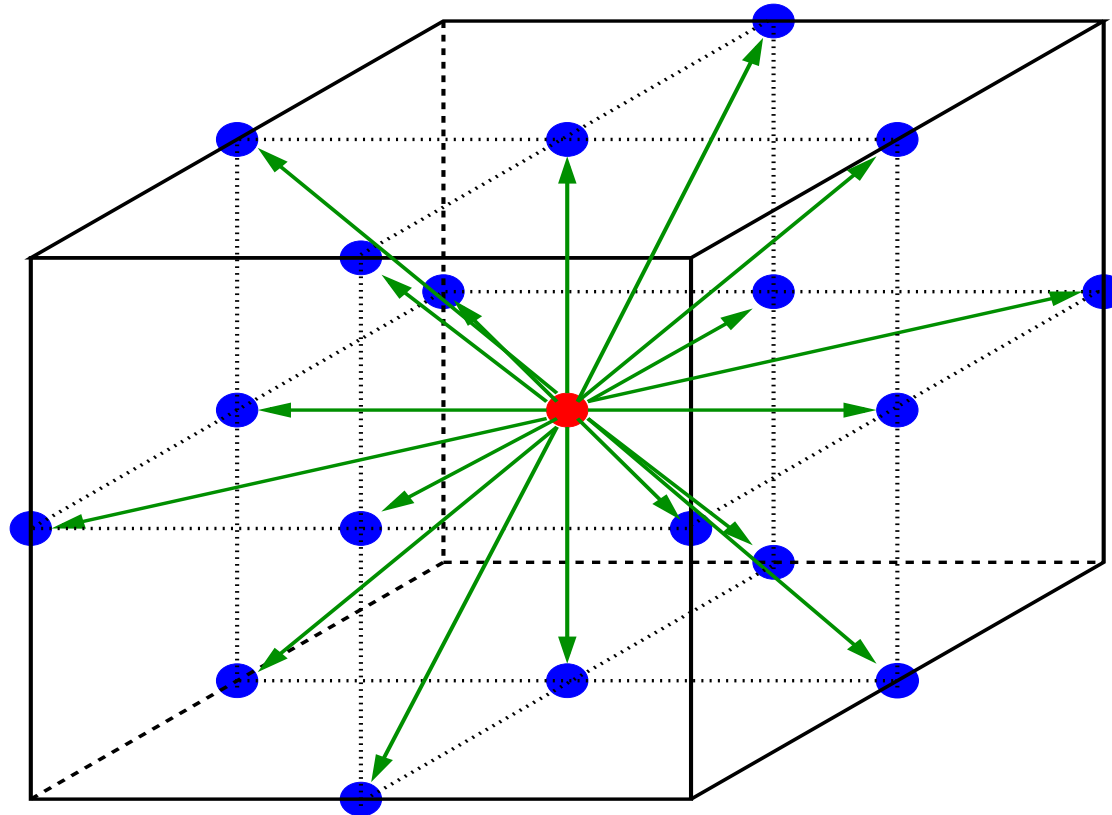
- Application: CFD
- Code: a simple, regular (but real!) FORTRAN77 LBM code, coming from vector world (IBM 3090/600 VF)
- Target platform: 16-way Nighthawk II nodes,  
Power3@375MHz + 16GB RAM  
L1 cache: 64 KB, 128-way set associative  
L2 cache: 8 MB, direct mapped  
Memory B/W: up to 16 GB/s
- Toolbox: xlf (7.1, 8.1), KAP/Pro ToolSet (Guide & Assure, rel 4.0), PMAPI, HPMLib
- Future targets: p690 Regatta Power4 nodes



# Lattice Boltzmann Method (LBM)

- Computes fluid flows with a kinetic method
- Explicit integration of a simplified Boltzmann equation:
  - Particles moves between sites of a uniform lattice
  - Particles jump from site to site according to a fixed, discrete velocities set
  - Particles collide when they meet at a site
- Integrated variables: particle distributions over the velocities set at the lattice sites

# LBM Lattice



D3Q19

Suitable lattices must satisfy some symmetry properties

# LBM Master Equation

Streaming



$$f_i(\vec{x} + \vec{c}_i; t + 1) =$$

$$f_i(\vec{x}; t) - \omega \cdot \left( f_i(\vec{x}; t) - f_i^{eq}(\rho(\vec{x}; t); \vec{u}(\vec{x}; t)) \right)$$

Collision

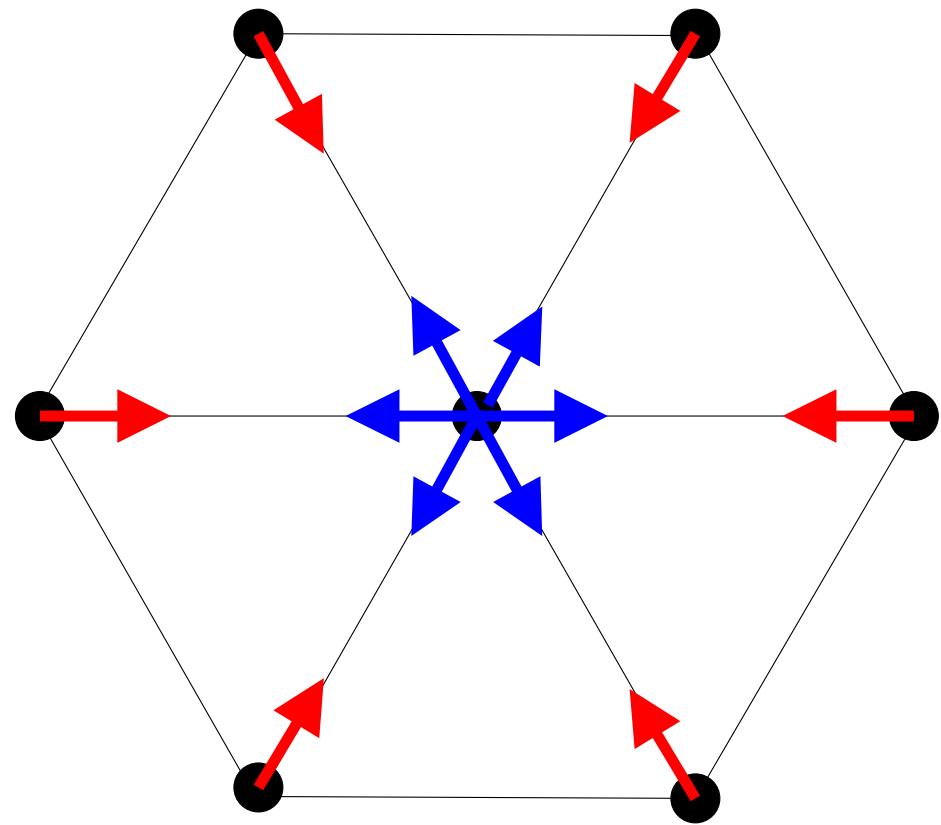
Fluid Variables:

$$\rho(\vec{x}, t) = \sum_i f_i(\vec{x}, t)$$

$$\vec{u}(\vec{x}, t) \rho(\vec{x}, t) = \sum_i f_i(\vec{x}, t) \vec{c}_i$$



# Collision



# Collision

$$f_i(\vec{x}; t^*) = f_i(\vec{x}; t) - \omega \cdot \left( f_i(\vec{x}; t) - f_i^{eq}(\rho(\vec{x}; t); \vec{u}(\vec{x}; t)) \right)$$

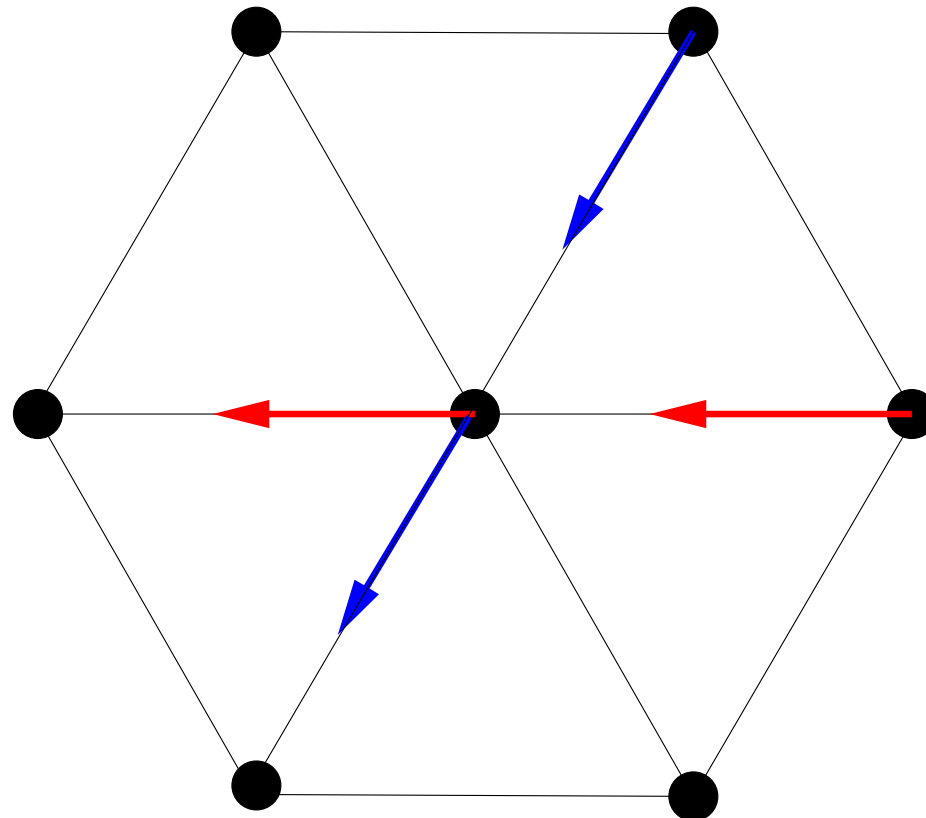
Collision = relaxation toward local equilibrium:

$$f_i^{eq}(\rho, \vec{u}) = d_i \rho \cdot \left( 1 + \frac{c_{i\alpha} u_\alpha}{c_s^2} + \frac{u_\alpha u_\beta}{2c_s^2} \left( \frac{c_{i\alpha} c_{i\beta}}{c_s^2} - \delta_{\alpha\beta} \right) \right)$$

- Completely local
- Extensive reuse of partial subexpressions in computing  $f_i^{eq}$
- A single inlined loop over the lattice, no function calls
- Load to store ratio is 1:1
- Computationally intensive (250-300 floating point operations per gridpoint in 3D)



# Streaming



# Streaming

Streaming = an exercise in moving data

$$f_i(\vec{x} + \vec{c}_i; t + 1) = f_i(\vec{x}; t^*)$$

- Nearest-neighbour interactions
- Stresses memory subsystem, no floating point operations
- Each population moves independently
- Be careful with data dependencies for in-place streaming, e.g.:

- D3Q19:

$$\{\vec{c}_i\} = \{(0,0,0), (\pm 1,0,0), (0,\pm 1,0), (0,0,\pm 1), (\pm 1,\pm 1,0), (\pm 1,0,\pm 1), (0,\pm 1,\pm 1)\}$$

- D3Q15:

$$\{\vec{c}_i\} = \{(0,0,0), (\pm 1,0,0), (0,\pm 1,0), (0,0,\pm 1), (\pm 1,\pm 1,\pm 1)\}$$



# Our target applications

- Turbulent channel flow
  - Looking for statistics of wall bounded turbulent flow
  - Huge number of timesteps  $> 5'000'000$ , many cases
  - Typical grid size:  $256*128*128$  gridpoint
  - Typical memory usage: 600 MB
- Two-phase flows
  - Looking for universal scaling laws in binary fluids
  - Moderate number of timesteps  $< 50'000$
  - Ideal grid size:  $512*512*512$  gridpoint
  - Memory usage: 32 GB



# Original code

- D3Q19 lattice
- Each population is represented in a separate 3-indexes array (explicit streaming parallelism)
- Collision step is implemented separately from streaming (more efficient on vector architectures, explicit collision parallelism)
- All the populations arrays are in separate common blocks
- 2 extra gridpoints for boundary conditions

# Serial code: collision step

```
do k = 1,n
  do j = 1,m
    do i = 1,l
      x01 = a01(i,j,k)
      x02 = a02(i,j,k)
      x03 = a03(i,j,k)
      x04 = a04(i,j,k)
      rho = (x01+x02)+(x03+x04)
      x05 = a05(i,j,k)
      ...
      rho = (x05+x06)+(x08+x09)+rho
      ...
      rhoinv= 1.0/rho
      vx = ((x01+x02)+(x03+x04)+x05-x10-x11-x12-x13-x14)*rhoinv
      .....
      e01 = rp2*(+vxmy+qxmy)
      e02 = rp2*(+vxmz+qxmz)
      .....
      a16(i,j,k)=omega1*x16+(omega*e16)
      a17(i,j,k)=omega1*x17+(omega*e17)
      a18(i,j,k)=omega1*x18+(omega*e18)
      a19(i,j,k)=omega1*x19+(omega*e19)
    end do
  end do
end do
```

# Serial code: streaming step

```
do 01 k = 1, n
do 01 j = 1, m
do 01 i = 1, 1, -1          ! c_1 = (1,0,0)
01   a01(i,j,k) = a01(i-1,j,k)

do 18 k = n, 1, -1
do 18 j = 1, m
do 18 i = 1, 1            ! c_18 = (0,-1,1)
18   a18(i,j,k) = a18(i,j+1,k-1)
```

# OpenMP LBM, basic: collision

```
C$OMP PARALLEL DO, PRIVATE(j,i,x01,x02,...,e01,e02,...)
do k = 1,n
  do j = 1,m
    do i = 1,l
      x01 = a01(i,j,k)
      x02 = a02(i,j,k)
      x03 = a03(i,j,k)
      x04 = a04(i,j,k)
      rho = (x01+x02)+(x03+x04)
      x05 = a05(i,j,k)
      ...
      rh0 = (x05+x06)+(x08+x09)+rho
      ...
      rhoinv= 1.0/rho
      vx = ((x01+x02)+(x03+x04)+x05-x10-x11-x12-x13-x14)*rhoinv
      .....
      e01 = rp2*(+vxmy+qxmy)
      e02 = rp2*(+vxmz+qxmz)
      .....
      a16(i,j,k)=omega1*x16+(omega*e16)
      a17(i,j,k)=omega1*x17+(omega*e17)
      a18(i,j,k)=omega1*x18+(omega*e18)
      a19(i,j,k)=omega1*x19+(omega*e19)
    end do
  end do
end do
```



# OpenMP LBM, basic: streaming

- Usual wisdom: outermost loop is the one to make parallel (less overhead)
- For D3Q19 lattice:
  - 8 populations have zero z velocity component, i.e. no data dependencies on outer loop (k index)
  - 10 populations move along z axis, so they have data dependencies in the outer loop
- Basic parallelization:
  - 8 PARALLEL DOs
  - 10 PARALLEL SECTIONs

# OpenMP LBM, basic: streaming

Data dependency (10 loops)

```
C$OMP SECTIONS
```

```
C$OMP SECTION
```

```
DO 18 K = N, 1, -1
```

```
DO 18 J = 1, M
```

```
DO 18 I = 1, L
```

```
18      A18(I,J,K) = A18(I,J+1,K-1)
```

```
C$OMP SECTION
```

```
.....
```

```
C$OMP END SECTIONS
```

No data dependency (8 loops)

```
C$OMP DO
```

```
DO 10 K = 1, N
```

```
DO 10 J = 1, M
```

```
DO 10 I = 1, L
```

```
10      A10(I,J,K) = A10(I+1,J,K)
```



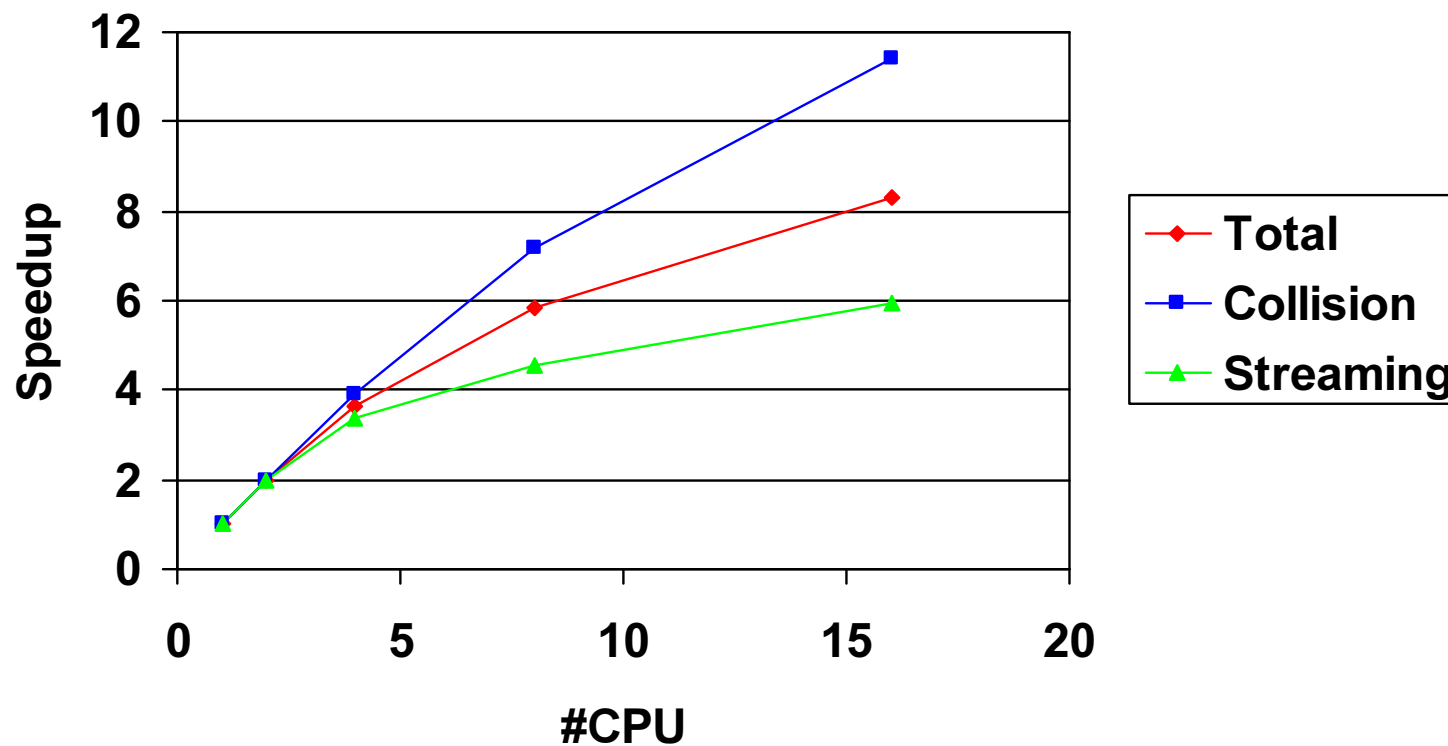
# OpenMP LBM, basic: performance

256×128×128, double prec., 500 timesteps

Threads	Total time	Total speedup	Collision time	Collision speedup	Streaming time	Streaming speedup
1	1882''	1	1248''	1	573''	1
2	957''	1.97	628''	1.99	291''	1.97
4	519''	3.63	321''	3.89	171''	3.35
8	323''	5.83	174''	7.17	125''	4.58
16	227''	8.29	110''	11.4	96''	5.97

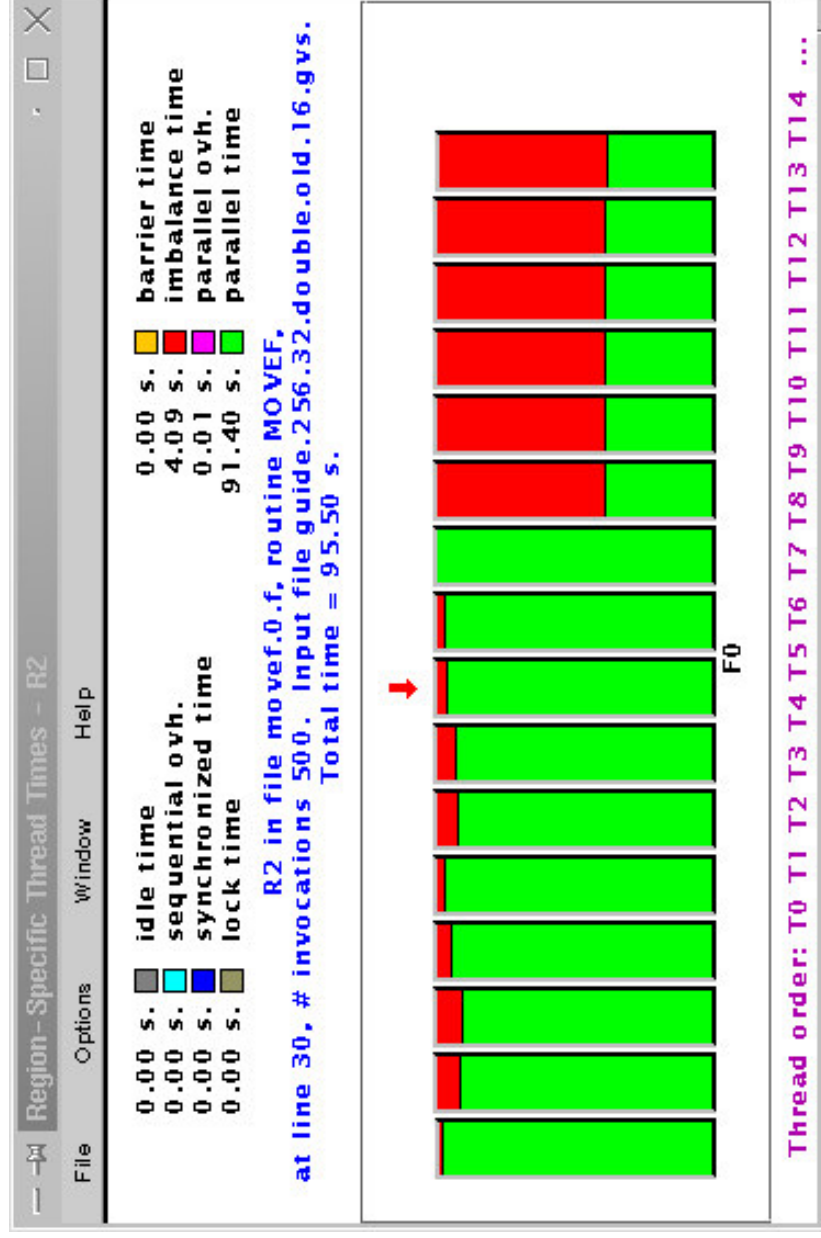
# OpenMP LBM, basic: speedup

256×128×128, double prec., 500 timesteps



# Streaming unbalance

SECTIONS cause load unbalance between threads



# OpenMP LBM, balanced: streaming

Let's change from:

```
C$OMP SECTION
```

```
    DO 18 K = N, 1, -1
```

```
        DO 18 J = 1, M
```

```
            DO 18 I = 1, L
```

```
18                A18(I,J,K) = A18(I,J+1,K-1)
```

```
C$OMP END SECTION
```

to:

```
C$OMP DO
```

```
    DO 18 I = 1, L
```

```
        DO 18 K = N, 1, -1
```

```
            DO 18 J = 1, M
```

```
18                A18(I,J,K) = A18(I,J,K+1)
```



# OpenMP LBM, balanced: how it works

- Usual wisdom:
  - outermost loop is the one to make parallel (less overhead)
  - loops should be nested in order of decreasing stride
- Guide is a source to source compiler
  - the loop is transformed in a parameterized subroutine
  - Guide runtime will call the subroutine from different threads
- The compiler can restructure the subroutine with the most efficient loop order
  
- Better parallelization
  - 18 PARALLEL DOs, no thread unbalances
  - Could destroy cache coherence for small lattice sizes (not an issue here!)

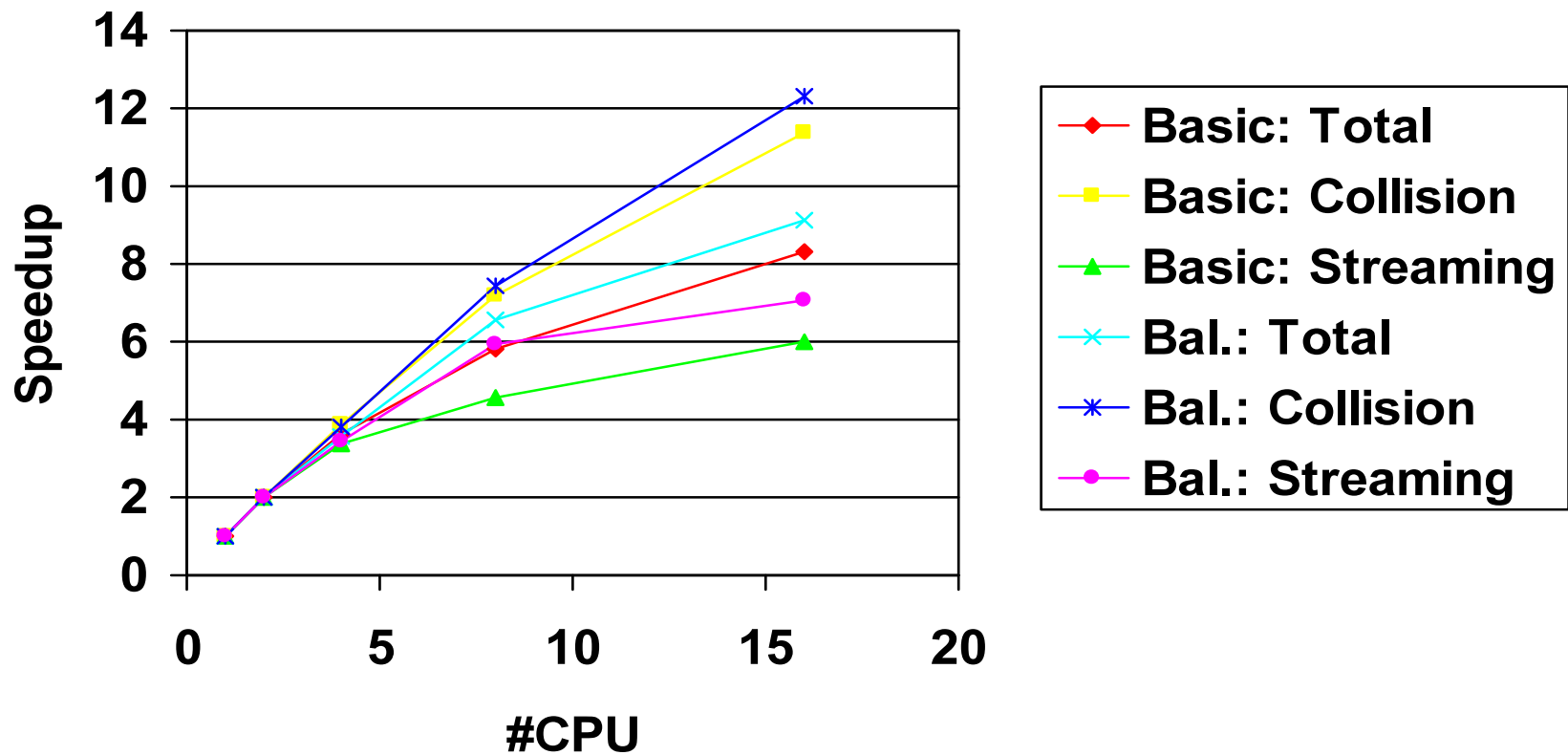
# OpenMP LBM, balanced: performance

256×128×128, double prec., 500 timesteps

Threads	Total time	Total speedup	Collision time	Collision speedup	Streaming time	Streaming speedup
1	1928''	1	1253''	1	614''	1
2	980''	1.97	629''	1.99	312''	1.97
4	541''	3.56	329''	3.81	178''	3.45
8	294''	6.56	168''	7.46	103''	5.96
16	211''	9.14	102''	12.3	87''	7.06

# OpenMP LBM, balanced: speedup

256×128×128, double prec., 500 timesteps





# OpenMP LBM, balanced: performance

256×128×128, **single** prec., 500 timesteps

Threads	Total time	Total speedup	Collision time	Collision speedup	Streaming time	Streaming speedup
1	1461''	1	1074''	1	347''	1
2	751''	1.95	540''	1.99	183''	1.87
4	384''	3.80	271''	3.96	94''	3.69
8	209''	7.00	138''	7.78	55''	6.31
16	130''	11.2	74''	14.5	42''	<b>8.62</b>

# OpenMP LBM, balanced: performance

256×128×128, double prec., 500 timesteps

Threads	Total time	Total speedup	Collision time	Collision speedup	Streaming time	Streaming speedup
1	1928''	1	1253''	1	614''	1
2	980''	1.97	629''	1.99	312''	1.97
4	541''	3.56	329''	3.81	178''	3.45
8	294''	6.56	168''	7.46	103''	5.96
16	211''	9.14	102''	12.3	87''	7.06



# OpenMP LBM: what's wrong?

- It's a memory bandwidth problem:
  - streaming tops at a sustained 6.5 GB/s
  - sustained b/w appears saturated for more than 8 CPUs
  - Less of an issue during collision, most time spent in computations



# OpenMP LBM: from balanced to fused

- Hiding streaming under collision
  - Each population is read and written once per site per timestep
  - Memory accesses are thus halved
  - It introduces data dependencies on every loop!!!
- Data dependencies removal:
  - From Eulerian to Lagrangian approach
  - Population data “at rest” in memory
  - Mapping of array indexes to space coordinates changes at every time step

# OpenMP LBM, fused: the code

```
C$OMP PARALLEL DO, PRIVATE(j,i, xp1, xm1, ..., x01, x02, ..., e01, e02, ...)
do k = 1, n
  zp1 = -(itime-k)+int((itime+n-k)/(n+1))*(n+1)
  zm1 = (itime+k)-int((itime+k-1)/(n+1))*(n+1)
  do j = 1, m
    yp1 = -(itime-j)+int((itime+m-j)/m)*m
    ym1 = (itime+j)-int((itime+j-1)/m)*m
    do i = 1, l
      xp1 = -(itime-i)+int((itime+l-i)/l)*l
      xm1 = (itime+i)-int((itime+i-1)/l)*l
      x01 = a01(xp1, ym1, k)
      x02 = a02(xp1, j, zm1)
      x03 = a03(xp1, yp1, k)
      x04 = a04(xp1, j, zp1)
      rho = (x01+x02)+(x03+x04)
      .....
      rhoinv= 1.0/rho
      vx = ((x01+x02)+(x03+x04)+x05-x10-x11-x12-x13-x14)*rhoinv
      .....
      e01 = rp2*(+vxmy+qxmy)
      e02 = rp2*(+vxmz+qxmz)
      .....
      a01(xp1, ym1, k) = omega1*x01+(omega*e01)
      .....
    end do
  end do
end do
```

# OpenMP LBM, fused: performance

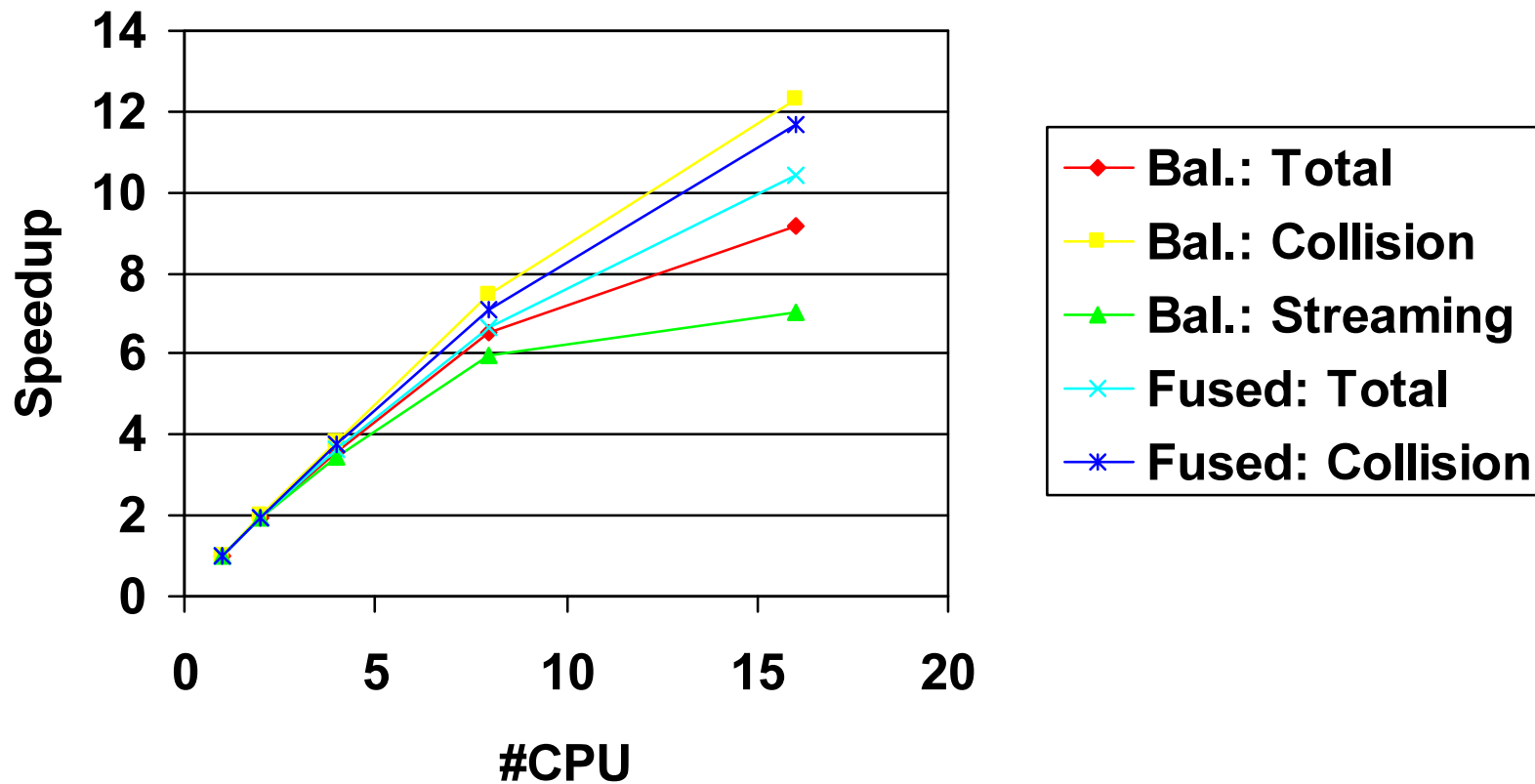
256×128×128, double prec., 500 timesteps

**30 % faster!**

Threads	Total time	Total speedup	Collision time	Collision speedup	Streaming time	Streaming speedup
1	1365''	1	1345''	1	N/A	N/A
2	706''	1.93	688''	1.95	N/A	N/A
4	347''	3.65	357''	3.77	N/A	N/A
8	206''	6.63	189''	7.12	N/A	N/A
16	131''	10.04	115''	11.7	N/A	N/A

# OpenMP LBM, fused: speedup

256×128×128, double prec., 500 timesteps



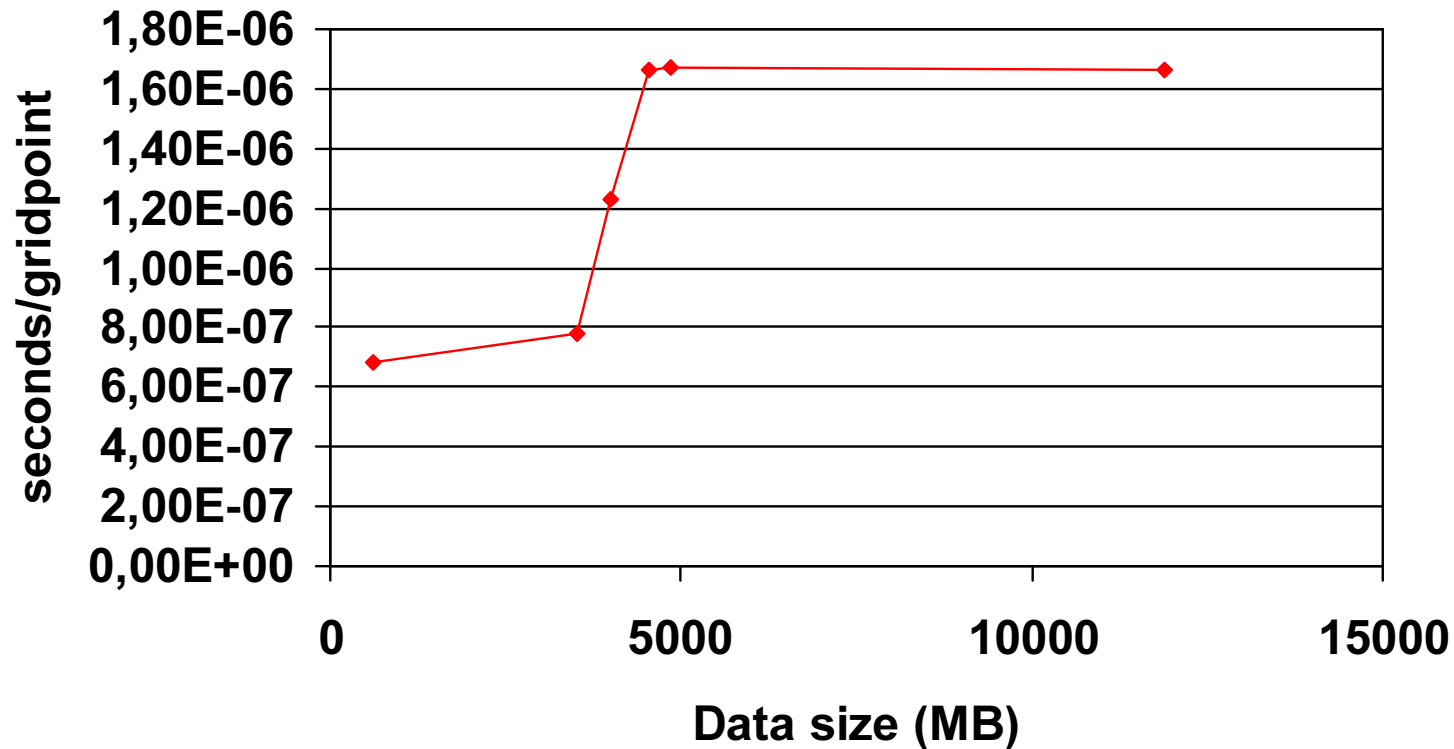


# OpenMP LBM, fused: the sad surprise

Boldly going to bigger grids...

Grid size	Memory size	Time/gridpoint (seconds)
<b>256×128 ×128</b>	<b>608 MB</b>	<b><math>0.68 \times 10^{-6}</math></b>
<b>460×230 ×230</b>	<b>3527 MB</b>	<b><math>0.78 \times 10^{-6}</math></b>
<b>480×240 ×240</b>	<b>4007 MB</b>	<b><math>1.23 \times 10^{-6}</math></b>
<b>500×250 ×250</b>	<b>4529 MB</b>	<b><math>1.66 \times 10^{-6}</math></b>
<b>512×256 ×256</b>	<b>4864 MB</b>	<b><math>1.67 \times 10^{-6}</math></b>
<b>690×345 ×345</b>	<b>11905 MB</b>	<b><math>1.66 \times 10^{-6}</math></b>

# OpenMP LBM, fused: degradation



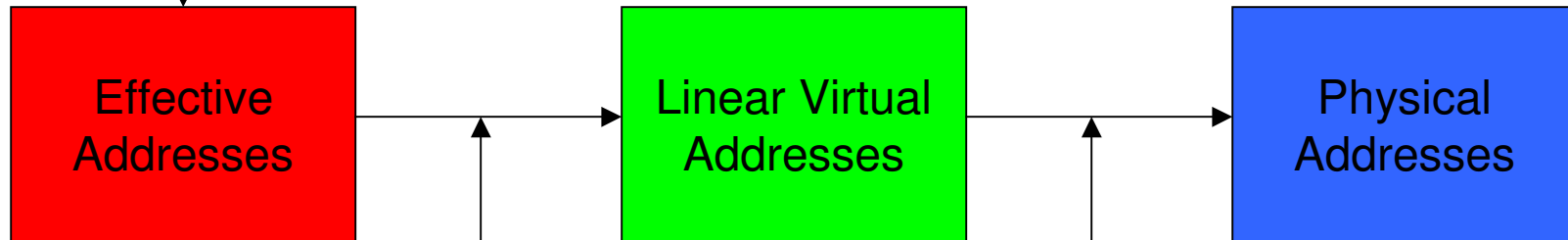
**Damned MPI version doing much better!!!**

# OpenMP LBM, fused: hpmcount report

```
PM_IC_MISS () : 87741
PM_DSLB_MISS () : 6888933063
PM_LD_MISS_L1 () : 1073877255
PM_CYC () : 442229523117
PM_STQ_FULL () : 3922479382
PM_LQ_FULL () : 172627390887
PM_ST_DISP () : 29048384100
PM_TLB_MISS () : 576496069
```

# POWER3: address translation

256 MB segments  
+ offset



Segment  
Lookaside  
Buffer  
16 entries

Translation  
Lookaside  
Buffer  
256 entries,  
2-way set assoc.

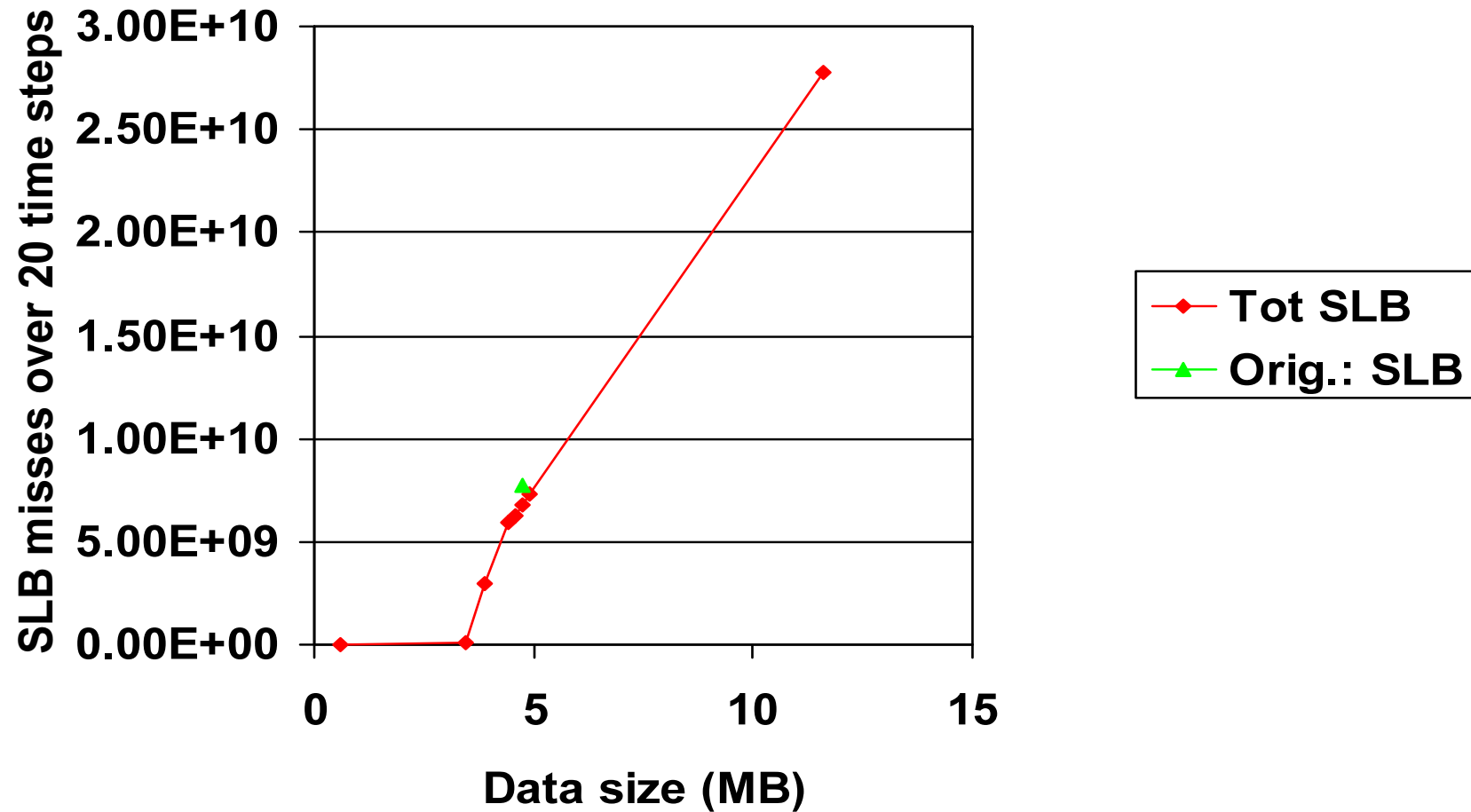
**THRASHING  
AHEAD!**



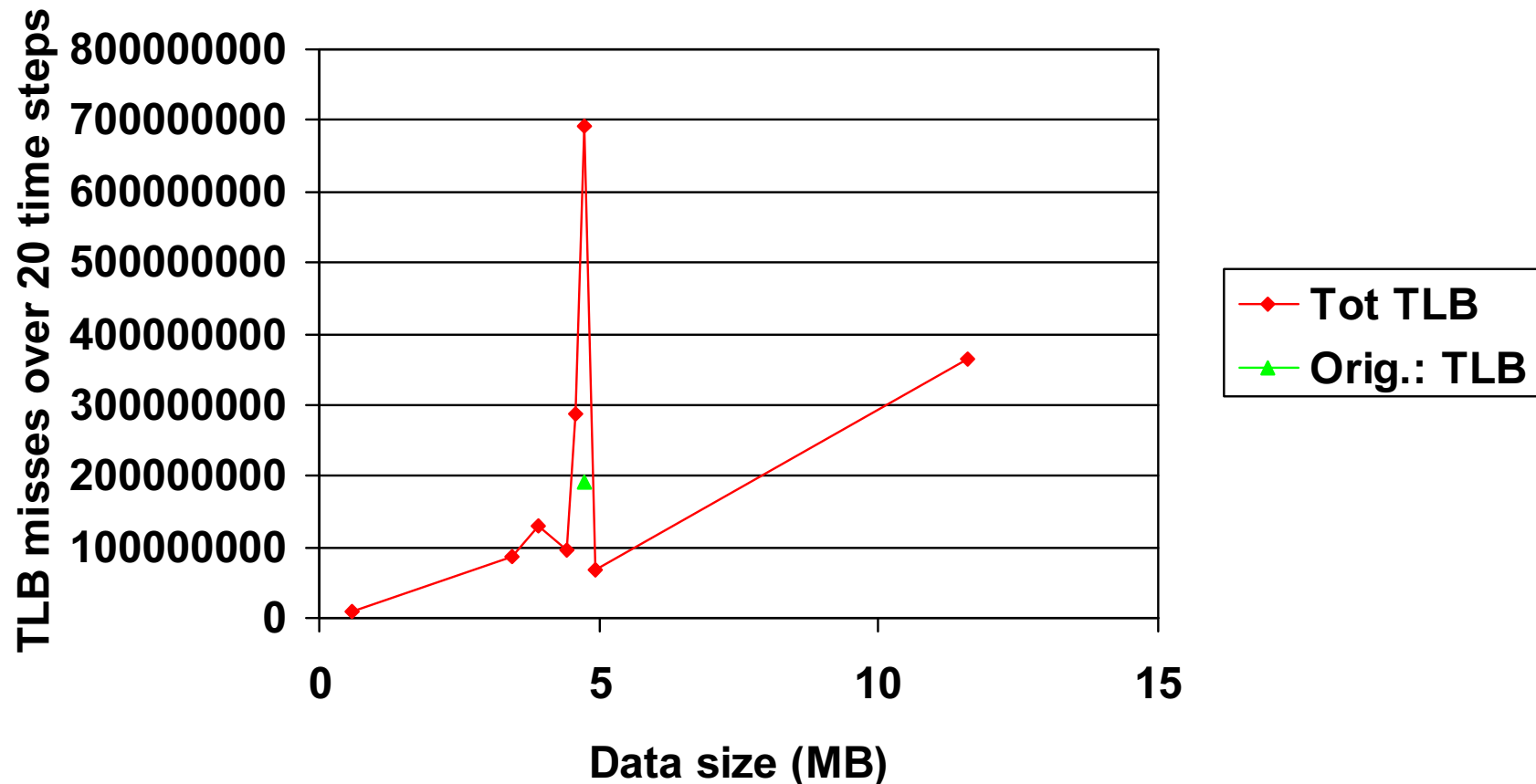
# Why are SLB and SLB so busy?

- LBM can use a lot of segments
  - For every collision loop iteration, 19 different memory streams are read and written to
  - As the grid grows, they will eventually be in 19 different segments
  - The problem affects every version, but you'll never notice if your code is not fast enough to make bigger grids affordable
- TLB thrashing
  - Original code accesses the 19 streams at fixed distance in memory: compiler can reduce thrashing
  - Fused implementations accesses the 19 streams with varying distances: it causes TLB thrashing to oscillate
  - Original code has less, but slower anyway

# LBM, Fused: SLB misses



# LBM, Fused: TLB misses



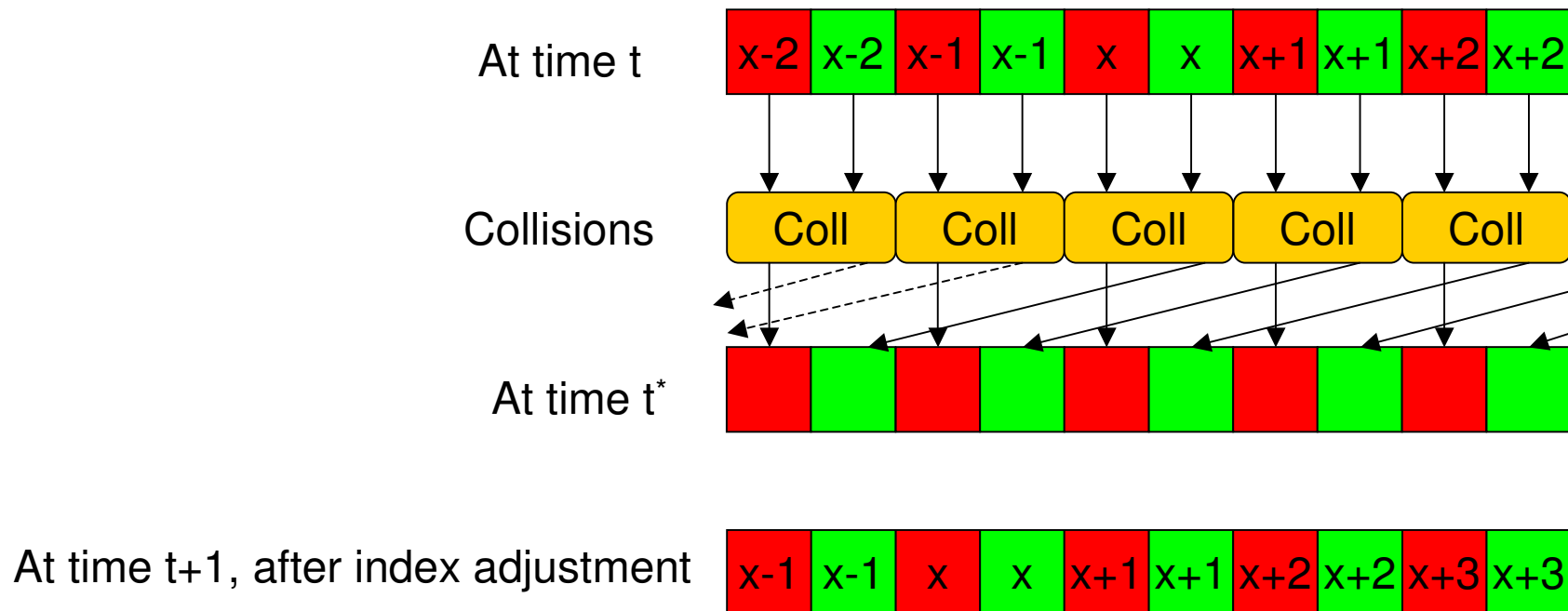


# Lessening SLB and TLB stress

- The trivial approach: all the population in a single array:  
`axx(ipop, i, j, k)`
- Not viable
  - Fused implementation: after a while, 19 different segments would be anyhow accessed
  - Standard implementation: streaming would be horribly inefficient (just one memory location exploited for every cache line)
- How to bundle together populations preserving fused implementation advantages?
- Key observations
  - Relative motion of particles at each time step are small and fixed
  - Neighboring lattice sites in the x direction (1st array index), fall mostly in the same cache line
  - Neighboring lattice sites in the y direction (2nd array index) are already in cache or will be needed shortly

# OpenMP LBM, bundled: the concept

- A simple 1 D model: red and green particles have +1 and -1 velocity
- Particle at the same site are bundled in neighboring memory locations
- Red particles: logical streaming (Lagrangian view)
- Green particles: relative streaming (partly Eulerian view)





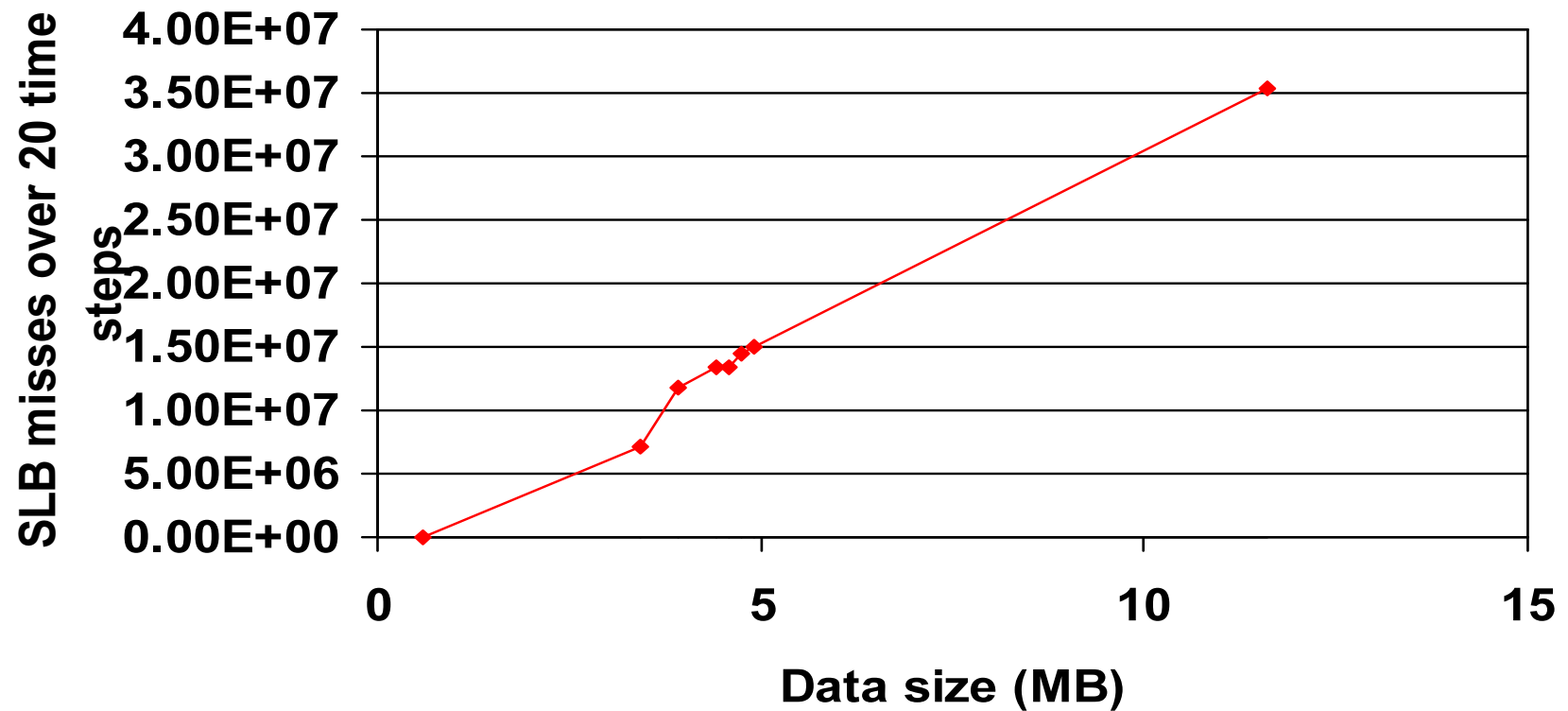
# OpenMP LBM, bundled: details

- Populations are bundled in pairs
  - Populations in a bundle differ in one velocity component
  - x-bundles: populations with opposite x velocity component
  - y-bundles: population with zero x velocity and opposite y velocity component
  - No bundles along z axis (that would add data dependencies to the parallel loop)
- Relative streaming is cache friendly:
  - Almost always in the same line for the 5 x-bundles
  - On already cached lines for the 3 y-bundles
- Memory streams are reduced to 11
- $0.70 \times 10^{-6}$  s per gridpoint (fused:  $0.68 \times 10^{-6}$  s on small grids)

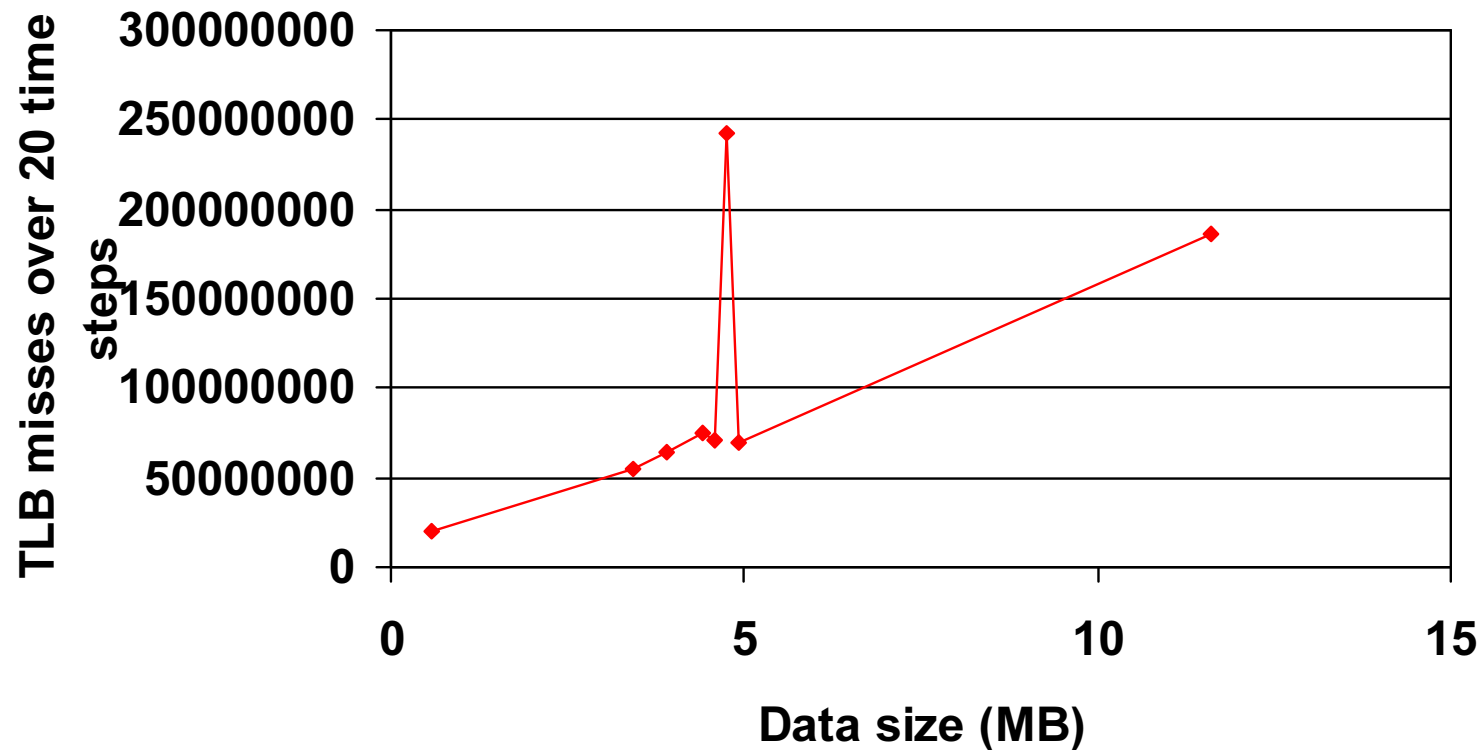
# OpenMP LBM, bundled: hpmcount report

```
PM_IC_MISS () : 1060403
PM_DSLB_MISS () : 40949773
                    (down from 6888933063)
PM_LD_MISS_L1 () : 1034166888
PM_CYC () : 201653713863
PM_STQ_FULL () : 1431015224
PM_LQ_FULL () : 19021872135
PM_ST_DISP () : 33687740796
PM_TLB_MISS () : 259407152
                    (down from 576496069)
```

# LBM, Bundled: SLB misses



# LBM, Bundled: TLB misses





# OpenMP LBM, bundled: performance

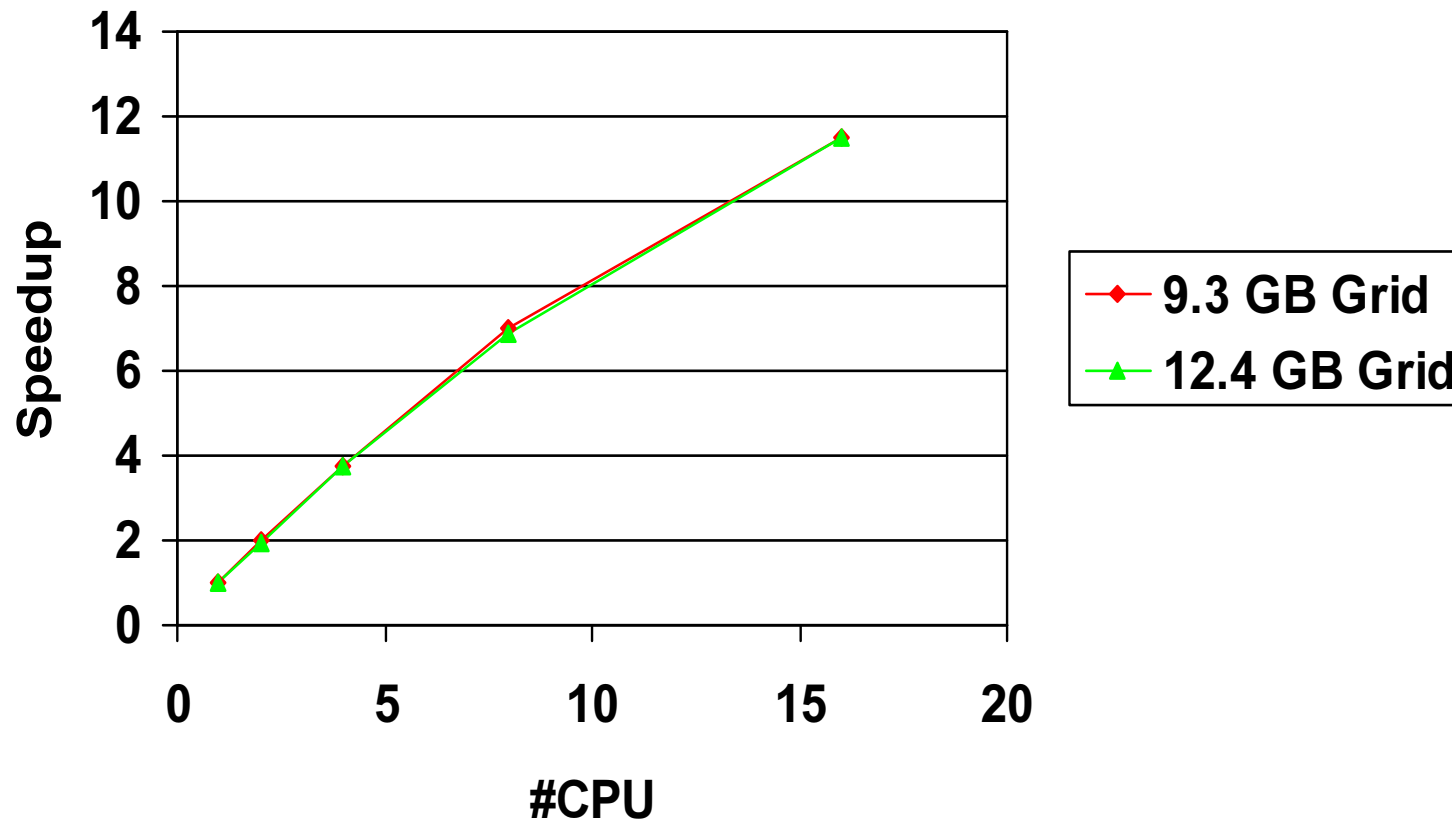
Case A:  $640 \times 320 \times 320$ , double prec., 9.3 GB , 500 timesteps

Case B:  $704 \times 352 \times 352$ , double prec., 12.4 GB , 500 timesteps

Threads	Case A time	Case A speedup	Case B time	Case B speedup
<b>1</b>	<b>23006''</b>	<b>1</b>	<b>29946''</b>	<b>1</b>
<b>2</b>	<b>11686''</b>	<b>1.97</b>	<b>15425''</b>	<b>1.94</b>
<b>4</b>	<b>6117''</b>	<b>3.76</b>	<b>7957''</b>	<b>3.77</b>
<b>8</b>	<b>3298''</b>	<b>6.97</b>	<b>4383''</b>	<b>6.85</b>
<b>16</b>	<b>2005''</b>	<b>11.5</b>	<b>2601''</b>	<b>11.5</b>



# OpenMP LBM, bundled: speedup





# What about the Power4?

- Power4 SLB is a fully-associative, 64 entries cache
- TLB has 1024 entries, 4-way set associative
- Speed of the fused implementation does not decrease as grid size increases
- Was the bundled implementation a useless effort on an obsolescent architecture?
- It could be useful...
  - Fused implementation DSM unfriendly: for big grids CPUs will fetch data from remote memory banks
  - Bundled implementation reduces the number of memory banks accessed by the CPU at the same time
  - Bundled implementation has fewer memory streams, so hardware prefetch could be more effective

# Power4 (preliminary!!): fused vs. bundled

Case A: 256×128×128, double prec., 0.6 GB , 500 timesteps

Case B: 768×384×384, double prec., 16 GB , 500 timesteps

Threads	Case A fused time (speedup)	Case A bundled time (speedup)	Case B fused time (speedup)	Case B bundled time (speedup)
1	780'' (1)	592'' (1)	22772'' (1)	16350'' (1)
2	412'' (1.89)	322'' (1.84)	11463'' (1.99)	8443'' (1.94)
4	202'' (3.86)	168'' (3.52)	5579'' (4.08)	4371'' (3.74)
8	105'' (7.43)	97'' (6.10)	3111'' (7.32)	2337'' (7.00)
16	55'' (14.2)	58'' (10.2)	1685'' (13.5)	1352'' (12.1)
32	32'' (24.4)	44'' (13.5)	975'' (23.36)	939'' (17.4)



# Conclusions

- Even a simple code can exhibit interesting behavior and issues
- Some performance problems can't be easily detected with traditional profiling tools
- Scaling up grid size and CPU number raises new problems
- Some problems don't show up until you go parallel
- OpenMP vs. MPI: deeper analyses needed
- Future work
  - We think there is space for more serial optimization on Power4
  - Performance issues for 32 processors on Regatta must be investigated
  - More detailed analysis of OpenMP runtimes behavior