



Performance portability of a lattice Boltzmann code

Federico Massaioli
Giorgio Amati

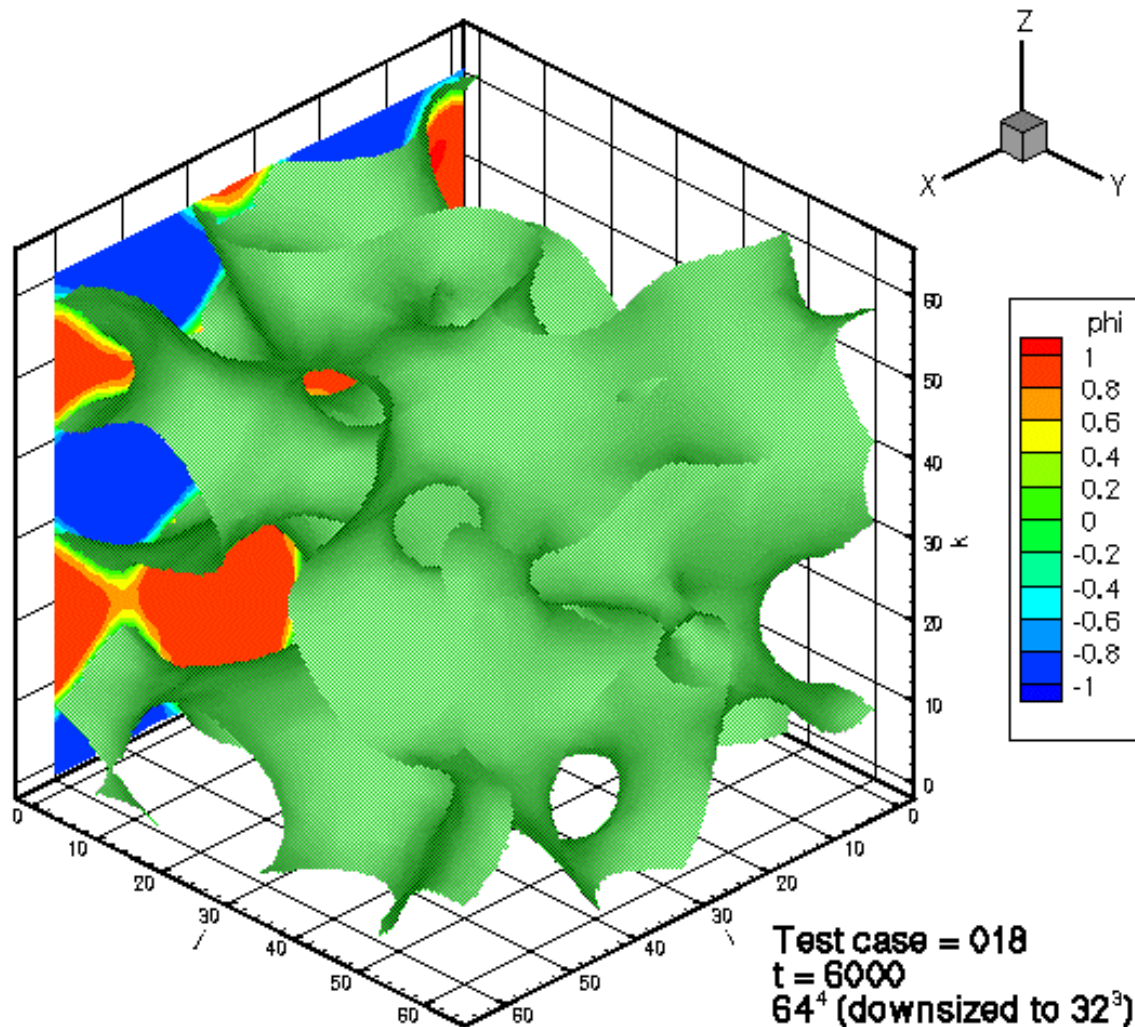
f.massaioli@caspur.it
g.amati@caspur.it



Outline of the talk

- Performance is a key issue for CFD
- We want to share our experiences about computing performance issues in LBM codes
- In particular:
 - Different implementations/algorithms
 - Different architectures/compilers
 - Portability: from cache-based machines to vector machines
- See previous talks at Scicomp6:
 - www.spscicomp.org/ScicomP6/Presentations/Amati
 - www.spscicomp.org/ScicomP6/Presentations/Massaioli

Binary demixing



512³ two-phase
simulations
running in
production since
8 months



Test Case

- 3D-Turbulent channel flow:
 - 256*128*128 grid-points
 - 1'000 time-steps
 - Single & double precision
 - Serial & parallel (OpenMP) performances
- Why?
 - Easier to modify/re-write/measure/dissect
 - Computationally intensive:
 - About 1'000'000'000 floating point operations for each time-step
 - More then 1'000'000 time-step are required to ensure enough statistics



Portability is important:

- IBM Power3@375 Mhz (1.5 Gflops peak)
- IBM Power4@1300 Mhz (5.2 Gflops peak)
- HP EV68@1250 Mhz (5.0 Gflops peak)
- Intel Xeon@2800 Mhz (5.6 Gflops peak)
- Intel Itanium2@1000 Mhz (4.0 Gflops peak)
- Intel Itanium2@900 Mhz (3.6 Gflops peak)
- AMD Athlon MP@1533 Mhz (3.0 Gflops peak)
- NEC SX-6i@500 Mhz (8.0 Gflops peak)

LBM Master Equation

Streaming
↓

$$f_i(\vec{x} + \vec{c}_i; t + 1) =$$
$$f_i(\vec{x}; t) - \omega \cdot \left(f_i(\vec{x}; t) - f_i^{eq}(\rho(\vec{x}; t); \vec{u}(\vec{x}; t)) \right)$$

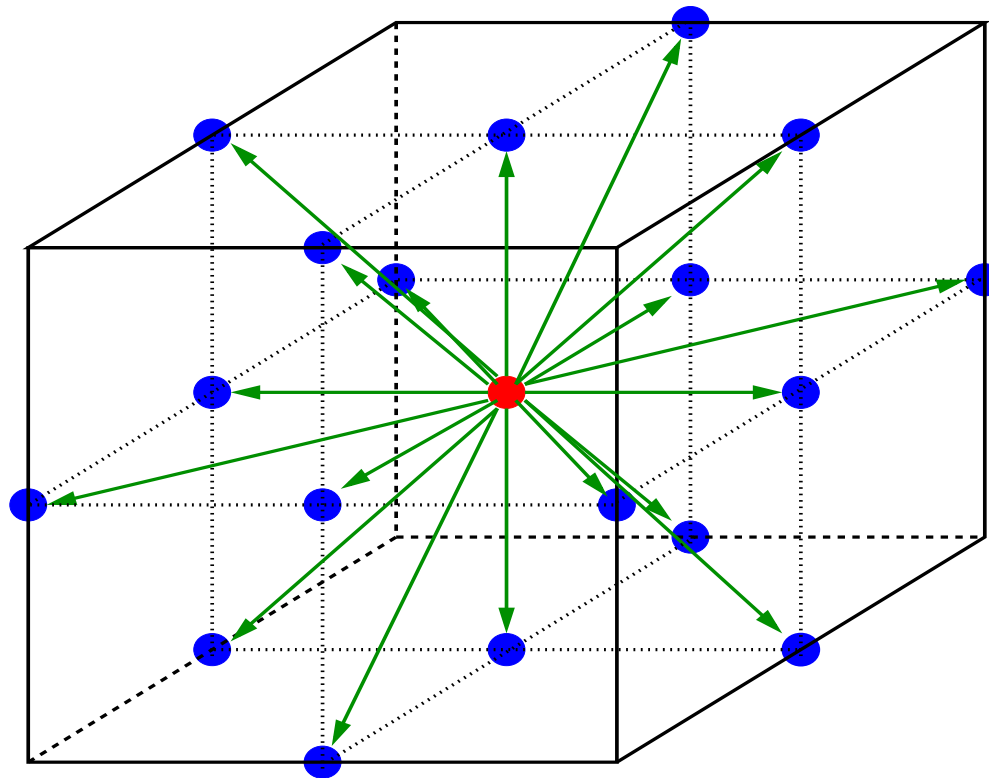
↑
Collision

Fluid Variables:

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

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

3D Lattice (D3Q19)



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 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
- Load to store ratio is 1:1
- Computationally intensive (200-250 floating point operations per gridpoint in 3D)

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:

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



LBM vs. other methods

- Requires more memory (> 4 for incompressible flow)
- Suboptimal “load to store” ratio (1:1)
- Strongly local (kinetic approach): easily parallelizable
- Less data have to be accessed to advance a gridpoint in time
- No need to solve pressure equation!!!
- More and more convenient as processors evolve
- Actual convenience depends on the physical problem



Parallel performances

Original OpenMP code on 32-processor Power4 IBM
Time for 1000 timesteps (seconds, double precision)

# procs	time	Speed-up	Collision	Streaming
Serial	2087	1.00	1369	676
2	1039	2.00	678	338
4	525	3.76	343	171
8	273	7.64	175	91
16	161	13.0	97	57
24	133	15.7	79	49
28	124	16.8	73	46
32	121	17.0	69	44

What's wrong?

Sustained memory bandwidth is limiting speedup
Single precision scale better!!

32 Power4 CPUs	Single precision	Double precision
Total speed-up	23.0 😊	17.0 😞
Collision speed-up	28.1	19.8
Streaming speed-up	16.3	15.4
Streaming Bandwidth	27 Gbyte/sec	26 Gbyte/sec

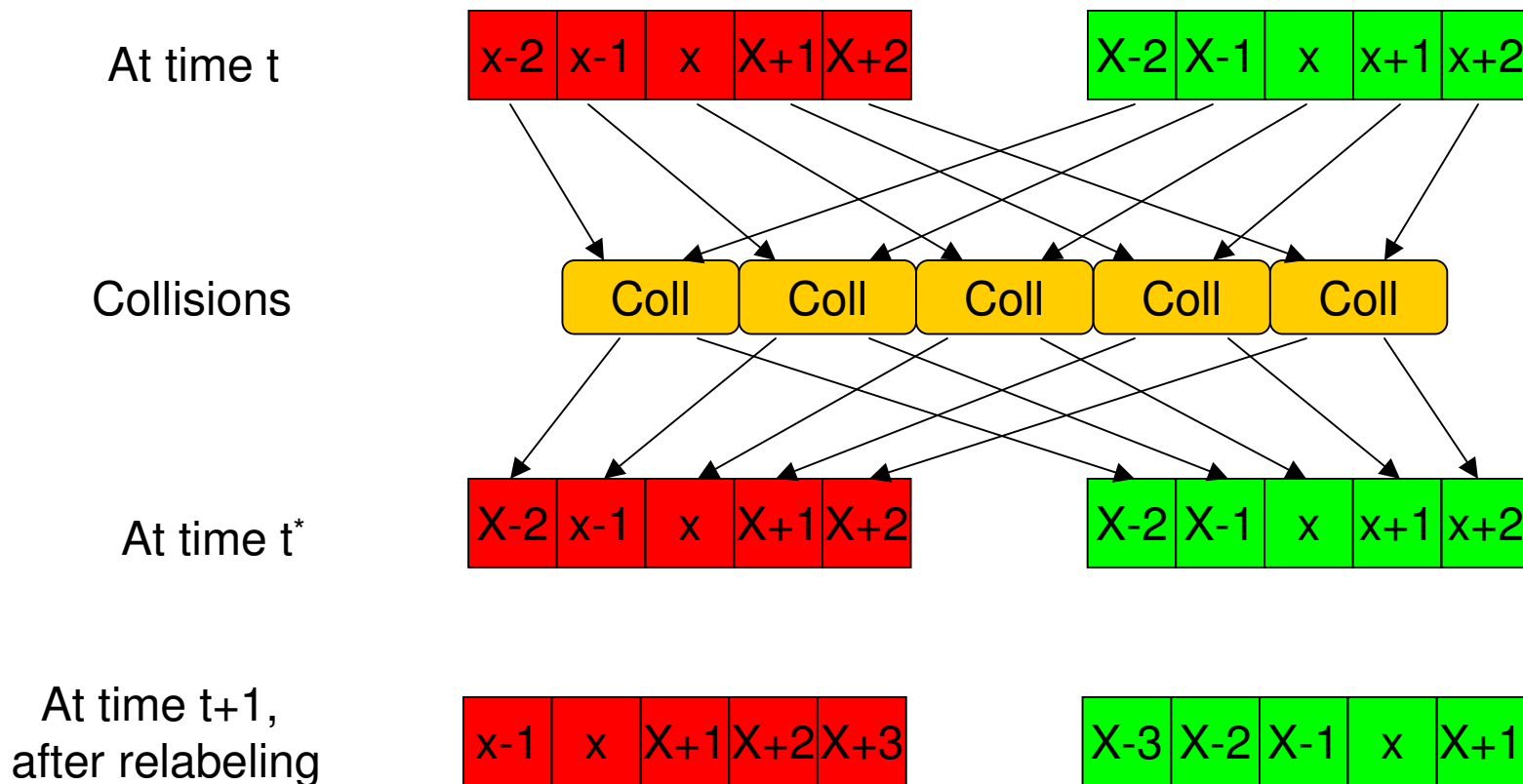


Too many memory accesses

- Each population read/written in streaming step, (again) read/modified/written in collision step
 - Memory accesses not a real issue during collision
- What if we “fuse” streaming and collision steps?
 - Memory accesses would be halved
 - Beware: 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

Fused LBM: the picture

A simple 1 D model: red and green particles have +1 and -1 velocity



Fused LBM collision step

```
omega1 = 1 - omega
do k = 1, N
  zp1 = N - mod(itime+N-k,N+1)           ! Noslip walls
  zm1 = mod(itime+k-1,N+1) + 1
  do j = 1, M
    yp1 = M - mod(itime+M-j,M)          ! Periodic B.C.
    ym1 = mod(itime+j-1,M) + 1
    do i = 1, L
      xp1 = L - mod(itime+L-i,L)        ! Periodic B.C.
      xm1 = mod(itime+i-1,L) + 1
      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
```



Fused LBM: serial performances

Serial time for 1000 timesteps (seconds, double precision)

	Original	Fused
Pwr3	4154	2934
Pwr4	2032	1704
EV68	1869	1046
Xeon	1893	1318
It2@900	1419	1051
It2@1000	1301	959
AthlonMP	2873	1468

Fused LBM: parallel performances

Fused code on 32 IBM

Time for 1000 timesteps

(precision)

20% faster on 1 proc!!!!

42% faster on 32 procs!!!!

# procs	time	Serial	Streaming
serial	1677	1.00	1671
2	852	1.97	849
4	433	3.87	431
8	225		
16	115		
24	92		
28	80		
32	70	24.0	68

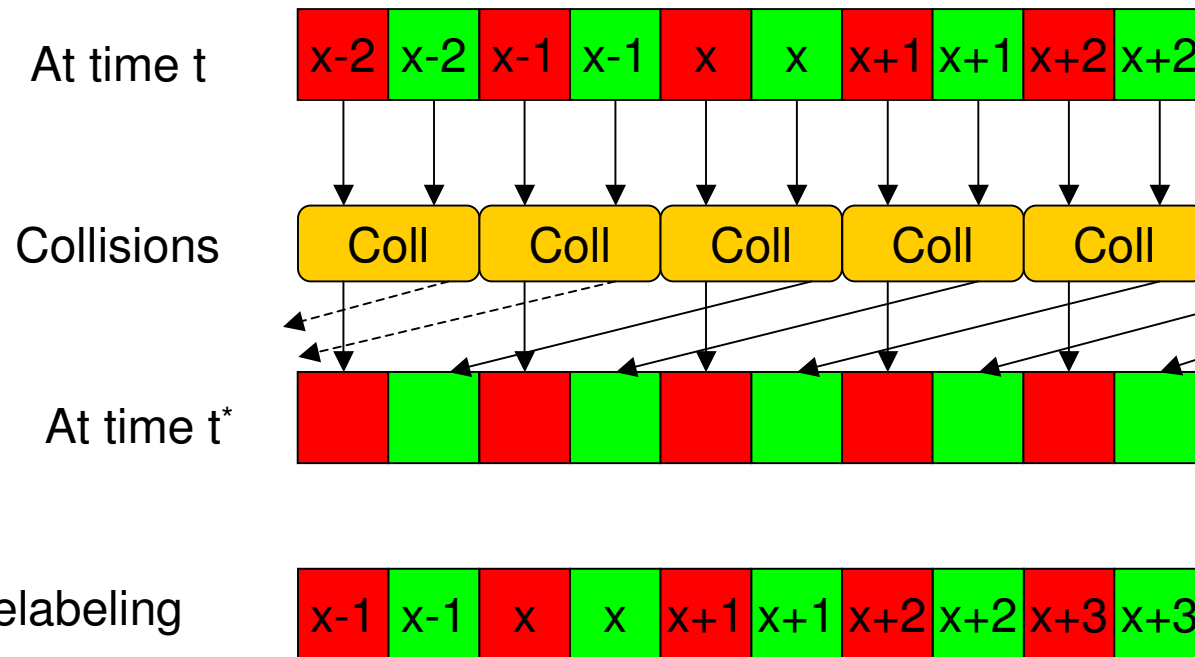


Too many arrays

- 19 memory data streams to/from the CPU are definitely too much...
 - Usual design point: 4 loads / 2 store
 - HW prefetching usually tuned at 4 to 8 data streams
 - Address translation hardware (TLB, SLB) can be stressed
- What if we “bundle” together pairs of populations in the same array?
 - 11 arrays for D3Q19 (not less to avoid data dependencies)
 - The streaming step must be partially restored
- Relative streaming
 - First population in the bundle managed as in “Fused” LBM
 - Second one streamed wrt. the first (cache friendly!!)

Bundled LBM: the picture

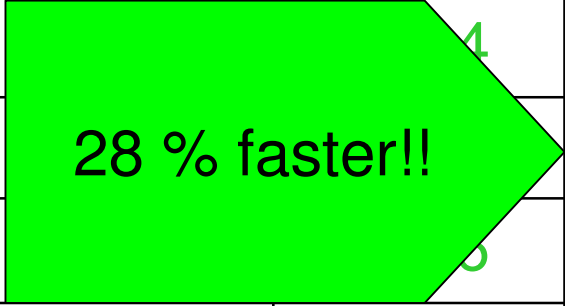
A simple 1 D model: red and green particles have +1 and -1 velocity
 Particle at the same site are bundled in neighboring memory locations
 Red particles: logical streaming
 Green particles: relative streaming



Bundled LBM: serial performances

Time for 1000 timesteps (time in seconds, double precision)

	Original	Fused	Bundle
Pwr3			3083
Pwr4			1231
EV68			1293
Xeon	1893	1318	1121
It2@900	1419	1051	1205
It2@1000	1301	959	1066
AthlonMP	2873	1468	1484



Running on a NEC SX-6i

Time for 1000 timesteps (seconds, double precision)

	Original	Fused	Bundle
Pwr3	4154	2934	3083
Pwr4	2032	1704	1231
EV68	1869	1046	1293
Xeon	fast!	Very slow!!!!!!!	
It2@900			
It2@1000			
AthlonMP			
SX-6i	191	2845	2707



Where's the problem?

- SX-6i: single processor vector system
 - 8 Gflops peak, huge memory bandwidth
 - Vector registers instead of cache
 - Very slow on non-vector code...
- Is the compiler vectorizing?
 - The original version is vector friendly...
 - But “Fused” and “Bundled” are not really different!!!
- The problem lies in address computations
 - MOD() to manage index wrap-around cause troubles
 - Let's split the loop in two!!

VBundled LBM: collision step

```
bound      = mod(itime,L+2)
istart(1)  = 1
istart(2)  = bound
istop(1)   = bound-1
istop(2)   = 1
disp(1)    = bound-(L+2)
disp(2)    = bound
omega1     = 1 - omega
do k = 1,n
  zp1 = N+1-mod(N+1-k+itime,N+2)
  zm1 = mod(k+itime,N+2)
  do j = 1,m
    yp1 = M+1-mod(M+1-j+itime,M+2)
    ym1 = mod(j+itime,M+2)
    yp1m2 = M+1-mod(M+1-j+itime+2,M+2)
    do ii=1,2
      iin = istart(ii)
      iend= istop(ii)
      idisp=disp(ii)
      do i = iin,iend,idisp
        xp1 = i - idisp
        xp1m2 = 1 - idisp -2
        x01 = a0110(1,xp1,ym1,  k)
        x02 = a0211(1,xp1,  j,zm1)
        .....
        a0718(1,    i,  yp1,zp1) = omega1*x07+(omega*e07      )
        a0718(2,    i,yp1m2,zp1) = omega1*x18+(omega*e18      )
      end do
    end do
  end do
end do
```

F. Massaioli & G.Amati, CASPUR 2004
Scicomp09, 23-26 March 2004

VBundled LBM: serial performances

- Time for 1000 timesteps (seconds, double precision)

	Original	Fused	Bundled	VBundled	Gain
Pwr3	4154	2934	3083	2811	32%
Pwr4	2032	1704	1231	1189	41%
EV68	1869	1046	1293	746	60%
Xeon	1893	1318	1121	1114	41%
It2@900	1419	1051	1205	725	49%
It2@1000	1301	959	1066	650	50%
AthlonMP	2873	1468	1484	1270	56%
SX-6i	192	2845	2707	175	8%

What this really means:

Estimated serial time for a 1'000'000 timesteps simulation
(using double precision, in days....)





In parallel, it's much faster...

VBundled OpenMP code on 32-processor Power4 IBM
Time for 1000 timesteps (seconds, single precision)

# procs	time	Speed-up	Collision	Streaming
Serial	996	1.00	956	-
2	480	2.08	464	-
4	242	4.11	232	-
8	118	8.44	113	-
16	59	16.9	56	-
24	47	21.2	44	-
28	41	24.3	37	-
32	37	29.6	31	-



Some conclusions

- Writing a code which is efficient on every architecture is possible (much different picture from 15 years ago!)
 - Reducing memory bandwidth abuse is the crucial issue
 - Doing that without stressing caches, address translation HW, etc... is not intuitive
- Vectorizing a code pays on most architectures
 - If this is done after reducing memory bandwidth abuse
 - Most modern processors/compiler enjoy vector codes (this seems a definite trend!)
- Can we further this approach? (We'll see...)
- A 1'000'000 timesteps channel flow simulation now needs only 19 hours on a 32 processor SMP IBM Power4 → **HAPPY USERS**