parallel programming project - 2d incompressible fluid

Nicole Vadot

initial serial code

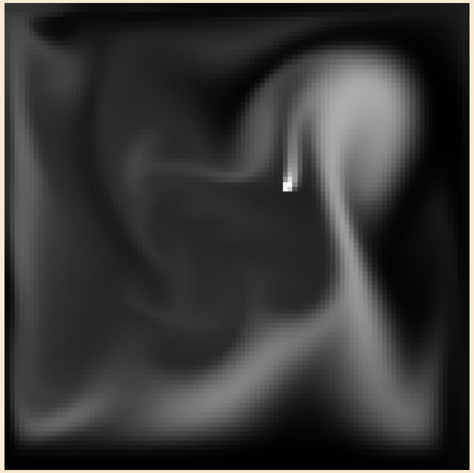
$$\nabla \cdot v = 0$$

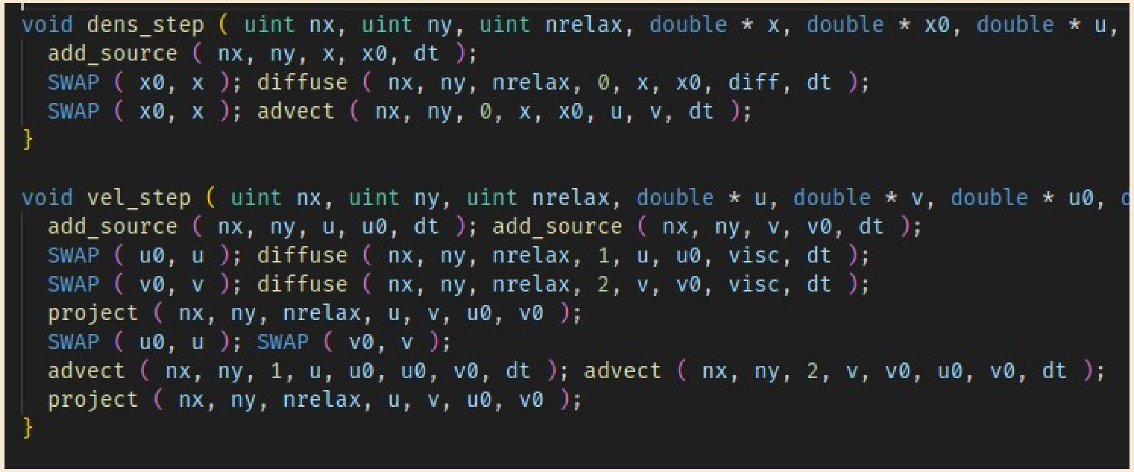
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

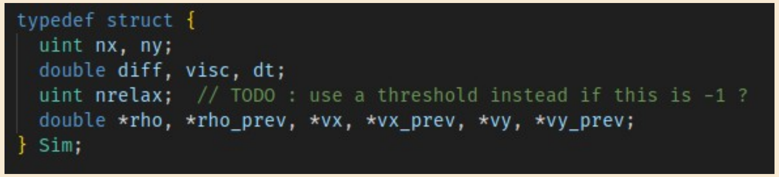
→ diffusion → Gauss-Seidel relaxation
→ advection → particle method
→ incompressibilty → "projection" (also GS)

→ complexity ~nx*ny*nrelax
→ nrelax = cst to prevent implicit dependence on nx, ny (no threshold)





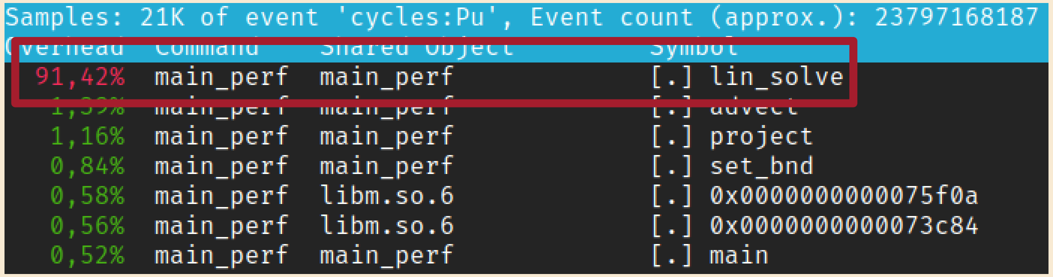
```
void dens_step ( uint nx, uint ny, uint nrelax, double * x, double * x0, double * u,
  add_source ( nx, ny, x, x0, dt );
  SWAP ( x0, x ); diffuse ( nx, ny, nrelax, 0, x, x0, diff, dt );
  SWAP ( x0, x ); advect ( nx, ny, 0, x, x0, u, v, dt );
}

void vel_step ( uint nx, uint ny, uint nrelax, double * u, double * v, double * u0, d
  add_source ( nx, ny, u, u0, dt ); add_source ( nx, ny, v, v0, dt );
  SWAP ( u0, u ); diffuse ( nx, ny, nrelax, 1, u, u0, visc, dt );
  SWAP ( v0, v ); diffuse ( nx, ny, nrelax, 2, v, v0, visc, dt );
  project ( nx, ny, nrelax, u, v, u0, v0 );
  SWAP ( u0, u ); SWAP ( v0, v );
  advect ( nx, ny, 1, u, u0, u0, v0, dt ); advect ( nx, ny, 2, v, v0, u0, v0, dt );
  project ( nx, ny, nrelax, u, v, u0, v0 );
}
```



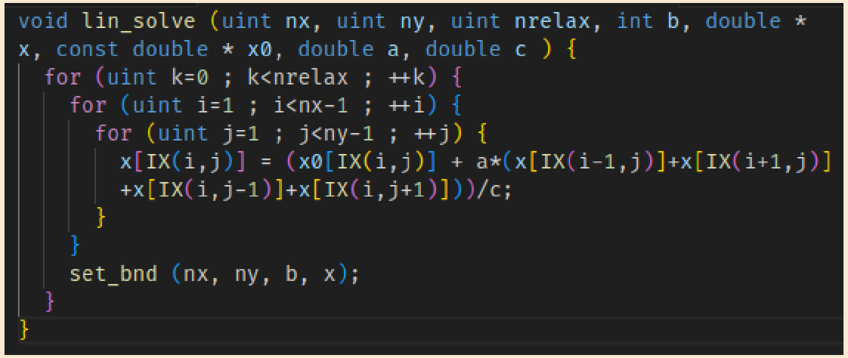
```
typedef struct {
  uint nx, ny;
  double diff, visc, dt;
  uint nrelax;  // TODO : use a threshold instead if this is -1 ?
  double *rho, *rho_prev, *vx, *vx_prev, *vy, *vy_prev;
} Sim;
```

where to optimize ?

```
make -B USE_OMP=0 USE_GS=0 RELEASE=1 main_perf && perf record ./main_perf 100 100
```



```
Samples: 21K of event 'cycles:Pu', Event count (approx.): 23797168187
Overhead  Command    Shared Object        Symbol
  91,42%  main_perf  main_perf            [.] lin_solve
   1,39%  main_perf  main_perf            [.] advect
   1,16%  main_perf  main_perf            [.] project
   0,84%  main_perf  main_perf            [.] set_bnd
   0,58%  main_perf  libm.so.6            [.] 0x0000000000075f0a
   0,56%  main_perf  libm.so.6            [.] 0x0000000000073c84
   0,52%  main_perf  main_perf            [.] main
```

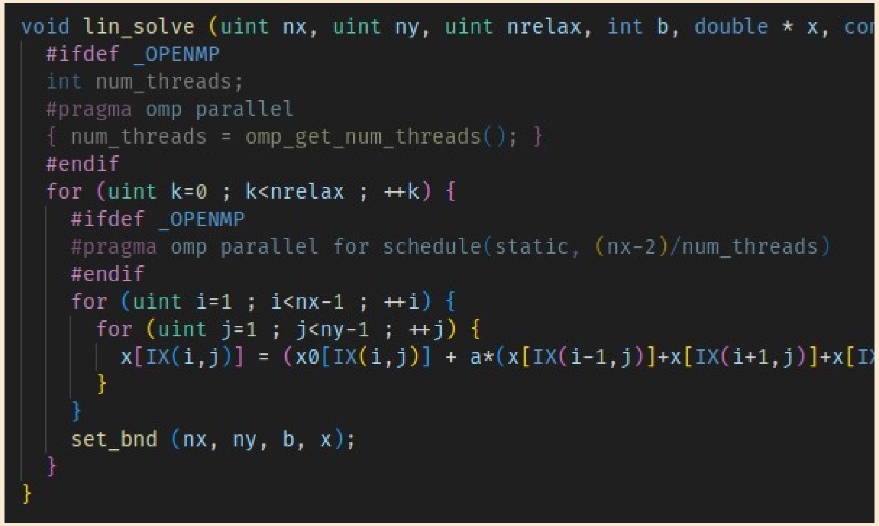


```c
void lin_solve (uint nx, uint ny, uint nrelax, int b, double *
x, const double * x0, double a, double c ) {
  for (uint k=0 ; k<nrelax ; ++k) {
    for (uint i=1 ; i<nx-1 ; ++i) {
      for (uint j=1 ; j<ny-1 ; ++j) {
        x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]
        +x[IX(i,j-1)]+x[IX(i,j+1)]))/c;
      }
    }
    set_bnd (nx, ny, b, x);
  }
}
```

⇒ most of the time is spent running lin_solve

1. optimize lin_solve

```
works:       #pragma omp parallel for
slower ???:  #pragma omp parallel for collapse(2)
~similar:    #pragma omp parallel for schedule(static, (nx-2)/num_threads))
```

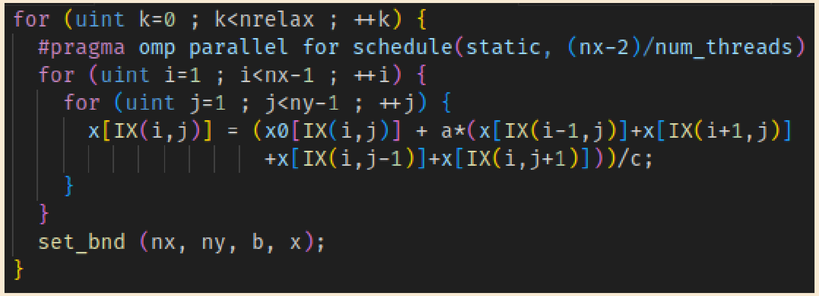


```
void lin_solve (uint nx, uint ny, uint nrelax, int b, double * x, co
  #ifdef _OPENMP
  int num_threads;
  #pragma omp parallel
  { num_threads = omp_get_num_threads(); }
  #endif
  for (uint k=0 ; k<nrelax ; ++k) {
    #ifdef _OPENMP
    #pragma omp parallel for schedule(static, (nx-2)/num_threads)
    #endif
    for (uint i=1 ; i<nx-1 ; ++i) {
      for (uint j=1 ; j<ny-1 ; ++j) {
        x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]+x[I
      }
    }
    set_bnd (nx, ny, b, x);
  }
}
```

```
nx=100, ny=100, nrelax=100
no omp:     ~30ms/step
1 thread:   ~32ms/step   overhead!
2 threads:  ~16ms/step
3 threads:  ~12ms/step
4 threads:  ~9.2ms/step
5 threads:  ~7.9ms/step
6 threads:  ~6.7ms/step
7 threads:  ~5.8ms/step   strong scaling
8 threads:  ~5.9ms/step   limit!
9 threads:  ~7.3ms/step
...
```

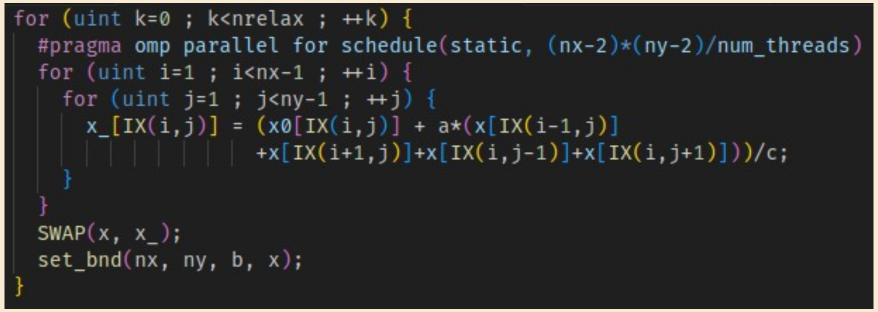
fixing the race condition

→ problem: results are not repeatable !!
  → race condition between the threads



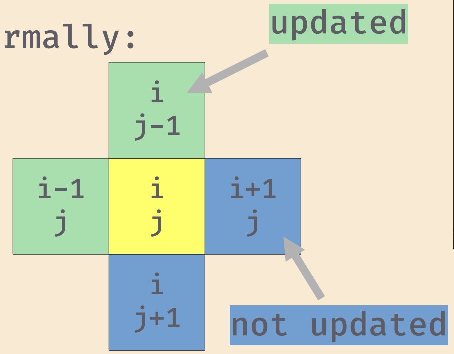
```
for (uint k=0 ; k<nrelax ; ++k) {
  #pragma omp parallel for schedule(static, (nx-2)/num_threads)
  for (uint i=1 ; i<nx-1 ; ++i) {
    for (uint j=1 ; j<ny-1 ; ++j) {
      x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]
                    +x[IX(i,j-1)]+x[IX(i,j+1)]))/c;
    }
  }
  set_bnd (nx, ny, b, x);
}
```

→ use Jacobi method instead of Gauss-Seidel

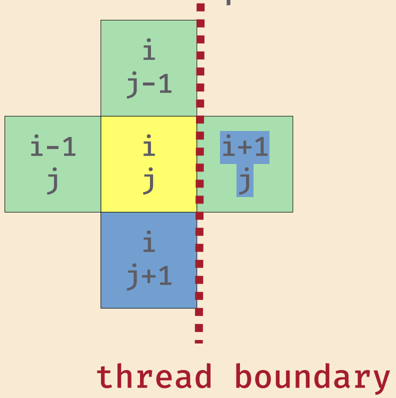


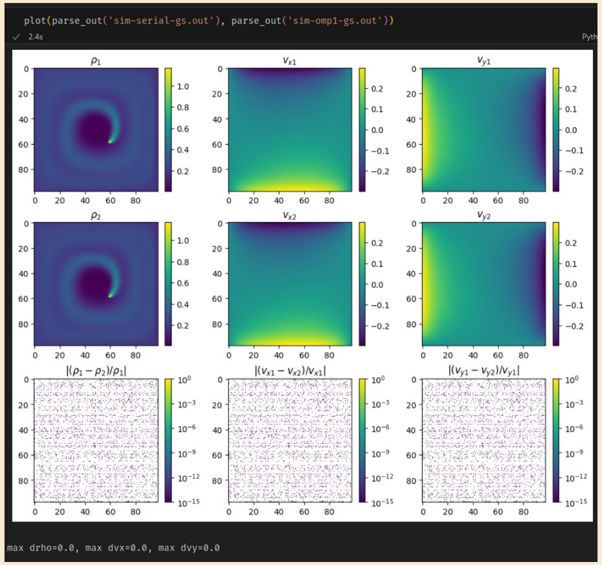
```
for (uint k=0 ; k<nrelax ; ++k) {
  #pragma omp parallel for schedule(static, (nx-2)*(ny-2)/num_threads)
  for (uint i=1 ; i<nx-1 ; ++i) {
    for (uint j=1 ; j<ny-1 ; ++j) {
      x_[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]
                    +x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)]))/c;
    }
  }
  SWAP(x, x_);
  set_bnd(nx, ny, b, x);
}
```

normally:



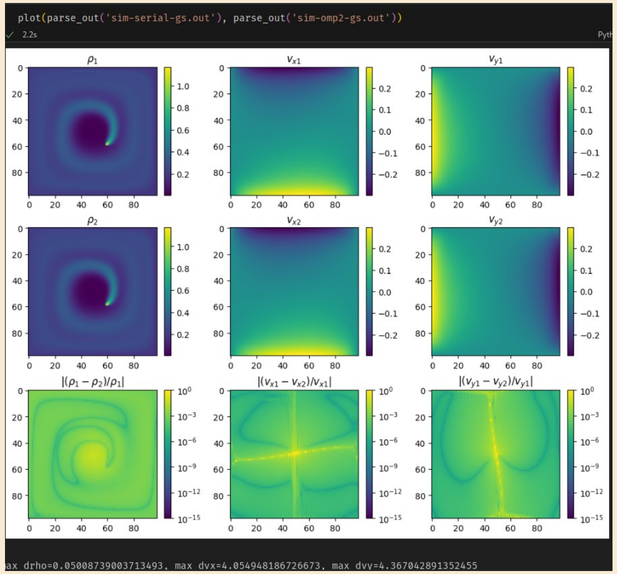
threads overlap!





```
plot(parse_out('sim-serial-gs.out'), parse_out('sim-omp1-gs.out'))
```
2.4s
ρ_1
v_{x1}
v_{y1}
ρ_2
v_{x2}
v_{y2}
$|(\rho_1-\rho_2)/\rho_1|$
$|(v_{x1}-v_{x2})/v_{x1}|$
$|(v_{y1}-v_{y2})/v_{y1}|$

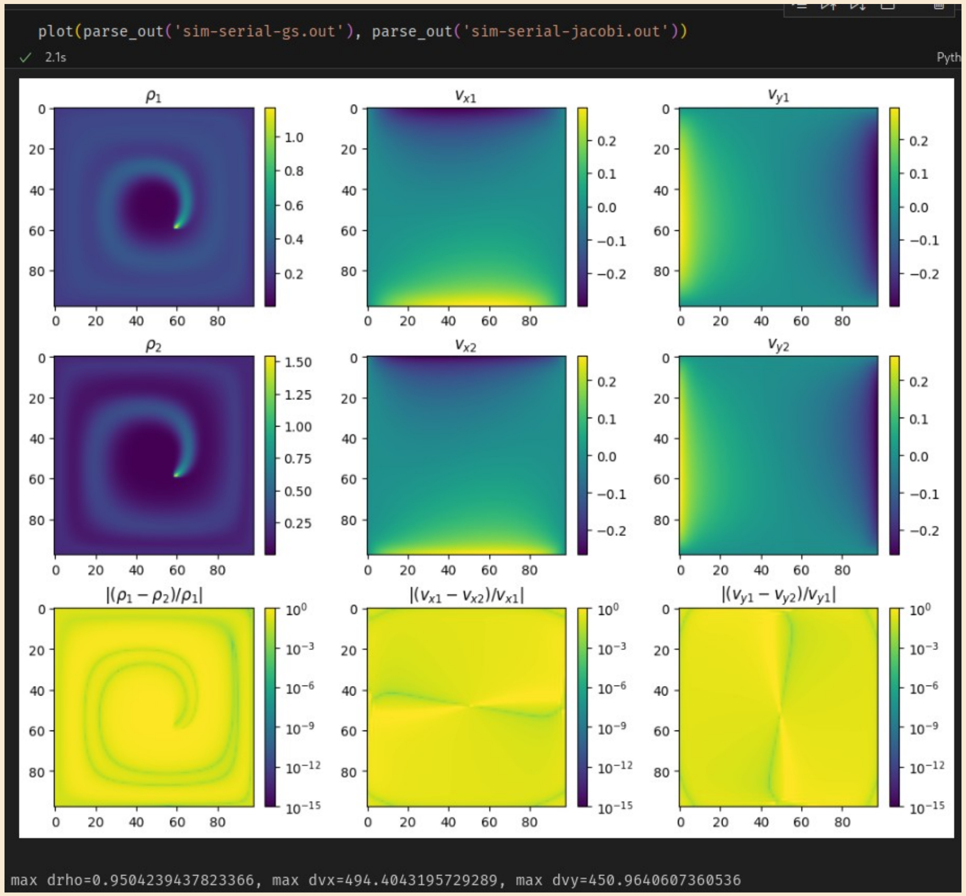
```
max drho=0.0, max dvx=0.0, max dvy=0.0
```



```
plot(parse_out('sim-serial-gs.out'), parse_out('sim-omp2-gs.out'))
```
2.2s
ρ_1
v_{x1}
v_{y1}
ρ_2
v_{x2}
v_{y2}
$|(\rho_1-\rho_2)/\rho_1|$
$|(v_{x1}-v_{x2})/v_{x1}|$
$|(v_{y1}-v_{y2})/v_{y1}|$

```
ax drho=0.05008739003713493, max dvx=4.054948186726673, max dvy=4.367042891352455
```

jacobi vs gauss-seidel



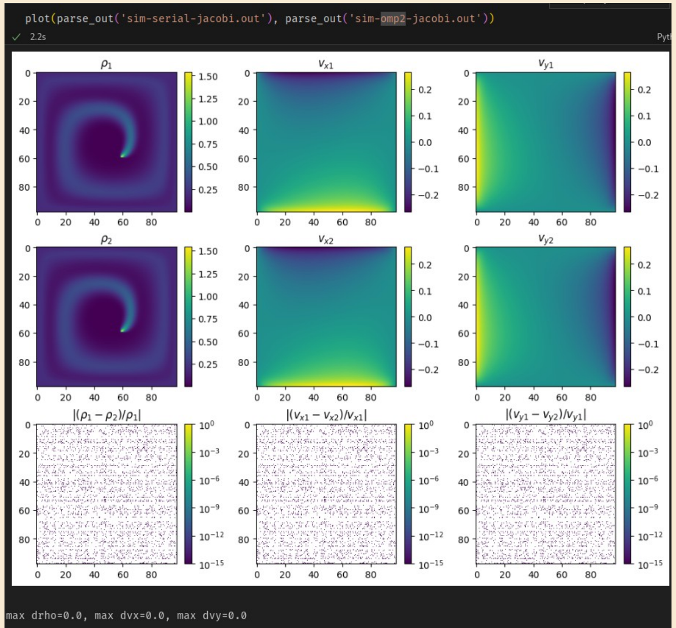
```
plot(parse_out('sim-serial-gs.out'), parse_out('sim-serial-jacobi.out'))
```

✓ 2.1s

```
max drho=0.9504239437823366, max dvx=494.4043195729289, max dvy=450.9640607360536
```

→ at fixed nrelax, not the same! (different convergence rate)

→ jacobi ~5.3ms/step
  gs ~30ms/step
  → ~6x speedup
  → no more read-after-write!



```
plot(parse_out('sim-serial-jacobi.out'), parse_out('sim-omp2-jacobi.out'))
```

✓ 2.2s

```
max drho=0.0, max dvx=0.0, max dvy=0.0
```

timings with jacobi [ms/step]

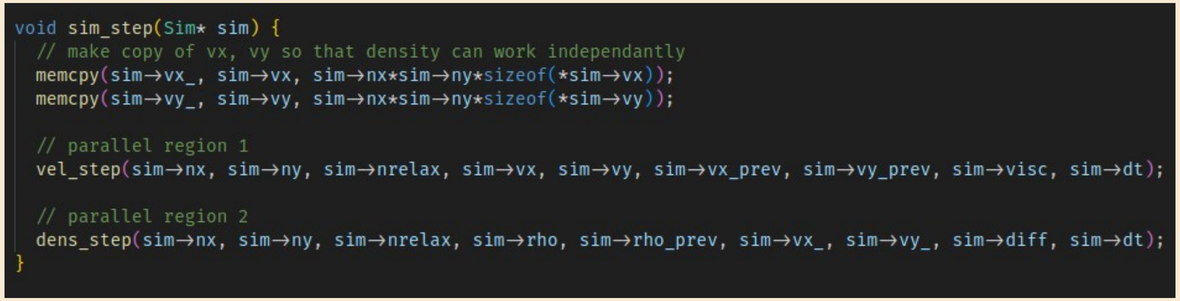
noomp	5.3
1 th.	5.4
2 th.	3.4
3 th.	2.6
4 th.	2.2
5 th.	2.0
6 th.	1.9
7 th.	2.8

2. improve balance between density and velocity computation

→ 5.3ms/step = 1.0ms/step [density]
               + 4.3ms/step [velocity]

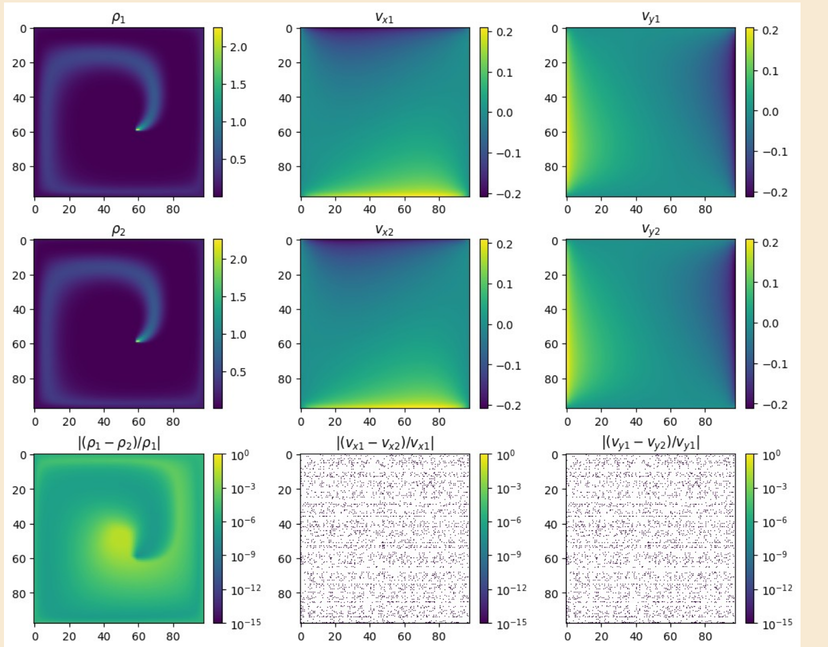
→ serial algo (leapfrog-style): rho → vx,vy → rho → vx,vy → …
→ parallel algo: (rho, vx,vy) → (rho, vx,vy) → …

new serial implementation   vvv



```c
void sim_step(Sim* sim) {
  // make copy of vx, vy so that density can work independantly
  memcpy(sim->vx_, sim->vx, sim->nx*sim->ny*sizeof(*sim->vx));
  memcpy(sim->vy_, sim->vy, sim->nx*sim->ny*sizeof(*sim->vy));

  // parallel region 1
  vel_step(sim->nx, sim->ny, sim->nrelax, sim->vx, sim->vy, sim->vx_prev, sim->vy_prev, sim->visc, sim->dt);

  // parallel region 2
  dens_step(sim->nx, sim->ny, sim->nrelax, sim->rho, sim->rho_prev, sim->vx_, sim->vy_, sim->diff, sim->dt);
}
```

compare with before.. →
difference is ~1e-7
→ seems OK



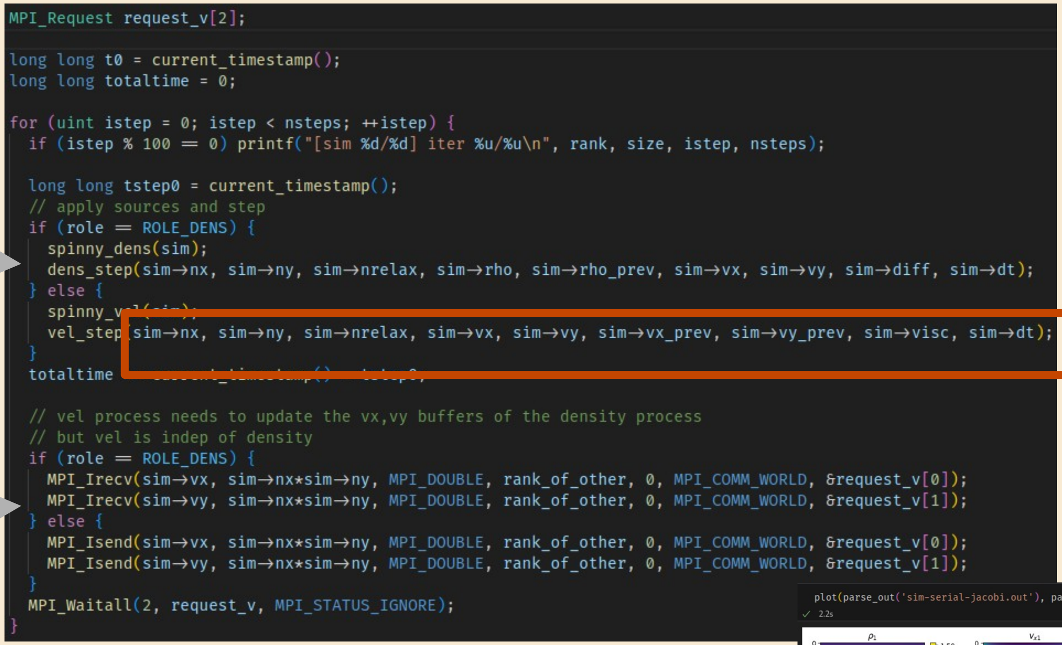
MPI parallel implementation

2 processes: dens and vel, each handles its own evolution

communications are only one-directional! (vel is indep of dens)

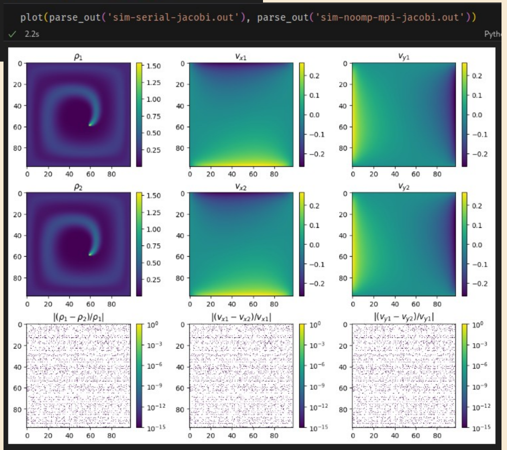
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$



```c
MPI_Request request_v[2];

long long t0 = current_timestamp();
long long totaltime = 0;

for (uint istep = 0; istep < nsteps; ++istep) {
  if (istep % 100 == 0) printf("[sim %d/%d] iter %u/%u\n", rank, size, istep, nsteps);

  long long tstep0 = current_timestamp();
  // apply sources and step
  if (role == ROLE_DENS) {
    spinny_dens(sim);
    dens_step(sim->nx, sim->ny, sim->nrelax, sim->rho, sim->rho_prev, sim->vx, sim->vy, sim->diff, sim->dt);
  } else {
    spinny_vel(sim);
    vel_step(sim->nx, sim->ny, sim->nrelax, sim->vx, sim->vy, sim->vx_prev, sim->vy_prev, sim->visc, sim->dt);
  }
  totaltime += current_timestamp() - tstep0;

  // vel process needs to update the vx,vy buffers of the density process
  // but vel is indep of density
  if (role == ROLE_DENS) {
    MPI_Irecv(sim->vx, sim->nx*sim->ny, MPI_DOUBLE, rank_of_other, 0, MPI_COMM_WORLD, &request_v[0]);
    MPI_Irecv(sim->vy, sim->nx*sim->ny, MPI_DOUBLE, rank_of_other, 0, MPI_COMM_WORLD, &request_v[1]);
  } else {
    MPI_Isend(sim->vx, sim->nx*sim->ny, MPI_DOUBLE, rank_of_other, 0, MPI_COMM_WORLD, &request_v[0]);
    MPI_Isend(sim->vy, sim->nx*sim->ny, MPI_DOUBLE, rank_of_other, 0, MPI_COMM_WORLD, &request_v[1]);
  }
  MPI_Waitall(2, request_v, MPI_STATUS_IGNORE);
}
```

verify identical results [OK] →



```python
plot(parse_out('sim-serial-jacobi.out'), parse_out('sim-noomp-mpi-jacobi.out'))
```
✓ 2.2s

balancing the dens, vel MPI processes (the theory)

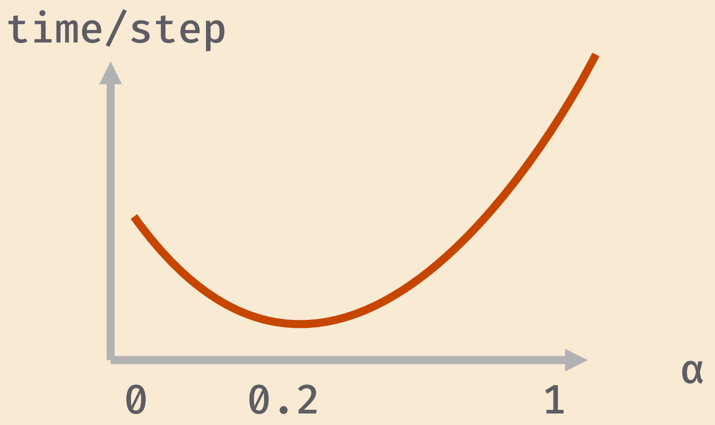
→ 5.3ms/step = 1.0ms/step [density] + 4.3ms/step [velocity]
⇒ time/step is limited by the longest of dens, vel process

→ idea: give more threads to velocity than density to equilibrate?

→ n_t threads available, $0 < \alpha < 1$
- $\alpha * n_t$ go to density
- $(1-\alpha) * n_t$ go to velocity

(+/- some rounding and thresholding)

→ what α to pick?
→ do a scan to minimize time/step
→ intuition -- 4:1 ratio
velocity: 4 calls to lin_solve per step
density: 1 call to lin_solve per step
⇒ $\alpha \sim 1/(4+1) = 0.2$



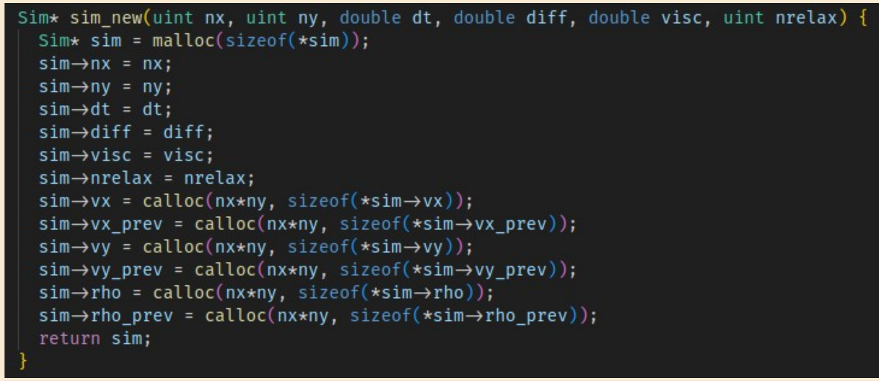
balancing the dens, vel MPI processes (the practice)

→ didn't get it to work :(
enabling OMP on top of MPI results in slower times (and more threads leads to even slower)

→ tried tweaking
- MPI_THREAD_LEVEL, MPI_Init{,_thread}
- OMP parallel for collapse, schedule …
- number of threads, grid size …

further improvements

- → first touch: OK probably



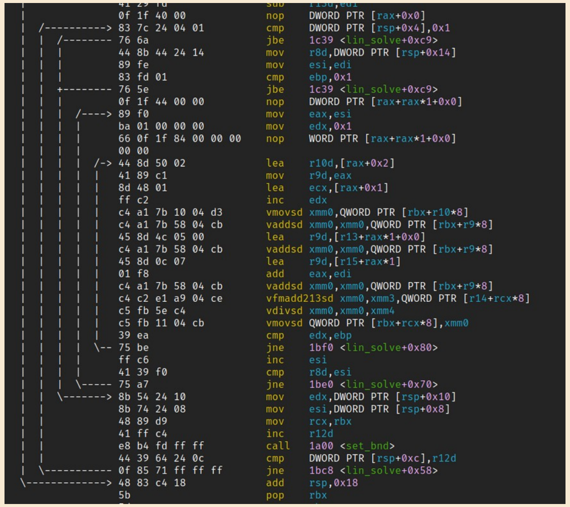
```c
Sim* sim_new(uint nx, uint ny, double dt, double diff, double visc, uint nrelax) {
  Sim* sim = malloc(sizeof(*sim));
  sim->nx = nx;
  sim->ny = ny;
  sim->dt = dt;
  sim->diff = diff;
  sim->visc = visc;
  sim->nrelax = nrelax;
  sim->vx = calloc(nx*ny, sizeof(*sim->vx));
  sim->vx_prev = calloc(nx*ny, sizeof(*sim->vx_prev));
  sim->vy = calloc(nx*ny, sizeof(*sim->vy));
  sim->vy_prev = calloc(nx*ny, sizeof(*sim->vy_prev));
  sim->rho = calloc(nx*ny, sizeof(*sim->rho));
  sim->rho_prev = calloc(nx*ny, sizeof(*sim->rho_prev));
  return sim;
}
```

- → MPI carthesian geometry
  … but, might help on large grids (not the case here, we are 2D)

- → prevent fork/join on every lin_solve
  + would require to also parallelize all other loops
    (or add omp single everywhere else)

- → review vectorized instructions
  … seems OK … maybe cache misses could be reduced?

- → interface parallelized MPI with GUI (once MPI+OMP works)



```
 |          0f 1f 40 00             nop    DWORD PTR [rax+0x0]
 | /----------> 83 7c 24 04 01       cmp    DWORD PTR [rsp+0x4],0x1
 | |  /-------- 76 6a                jbe    1c39 <lin_solve+0xc9>
 | |  |         44 8b 44 24 14       mov    r8d,DWORD PTR [rsp+0x14]
 | |  |         89 fe                mov    esi,edi
 | |  |         83 fd 01             cmp    ebp,0x1
 | |  +-------- 76 5e                jbe    1c39 <lin_solve+0xc9>
 | |  |         0f 1f 44 00 00       nop    DWORD PTR [rax+rax*1+0x0]
 | |  |  /----> 89 f0                mov    eax,esi
 | |  |  |      ba 01 00 00 00       mov    edx,0x1
 | |  |  |      66 0f 1f 84 00 00 00 nop    WORD PTR [rax+rax*1+0x0]
 | |  |  |      00 00
 | |  |  |  /-> 44 8d 50 02          lea    r10d,[rax+0x2]
 | |  |  |  |   41 89 c1             mov    r9d,eax
 | |  |  |  |   8d 48 01             lea    ecx,[rax+0x1]
 | |  |  |  |   ff c2                inc    edx
 | |  |  |  |   c4 a1 7b 10 04 d3    vmovsd xmm0,QWORD PTR [rbx+r10*8]
 | |  |  |  |   c4 a1 7b 58 04 cb    vaddsd xmm0,xmm0,QWORD PTR [rbx+r9*8]
 | |  |  |  |   45 8d 4c 05 00       lea    r9d,[r13+rax*1+0x0]
 | |  |  |  |   c4 a1 7b 58 04 cb    vaddsd xmm0,xmm0,QWORD PTR [rbx+r9*8]
 | |  |  |  |   45 8d 0c 07          lea    r9d,[r15+rax*1]
 | |  |  |  |   01 f8                add    eax,edi
 | |  |  |  |   c4 a1 7b 58 04 cb    vaddsd xmm0,xmm0,QWORD PTR [rbx+r9*8]
 | |  |  |  |   c4 c2 e1 a9 04 ce    vfmadd213sd xmm0,xmm3,QWORD PTR [r14+rcx*8]
 | |  |  |  |   c5 fb 5e c4          vdivsd xmm0,xmm0,xmm4
 | |  |  |  |   c5 fb 11 04 cb       vmovsd QWORD PTR [rbx+rcx*8],xmm0
 | |  |  |  |   39 ea                cmp    edx,ebp
 | |  |  |  \-- 75 be                jne    1bf0 <lin_solve+0x80>
 | |  |  |      ff c6                inc    esi
 | |  |  |      41 39 f0             cmp    r8d,esi
 | |  |  \----- 75 a7                jne    1be0 <lin_solve+0x70>
 | |  \-------> 8b 54 24 10          mov    edx,DWORD PTR [rsp+0x10]
 | |            8b 74 24 08          mov    esi,DWORD PTR [rsp+0x8]
 | |            48 89 d9             mov    rcx,rbx
 | |            41 ff c4             inc    r12d
 | |            e8 b4 fd ff ff       call   1a00 <set_bnd>
 | |            44 39 64 24 0c       cmp    DWORD PTR [rsp+0xc],r12d
 | \----------- 0f 85 71 ff ff ff    jne    1bc8 <lin_solve+0x58>
 \------------> 48 83 c4 18          add    rsp,0x18
                5b                   pop    rbx
```

`objdump -Mintel --visualize-jumps --disassemble=lin_solve main_perf`