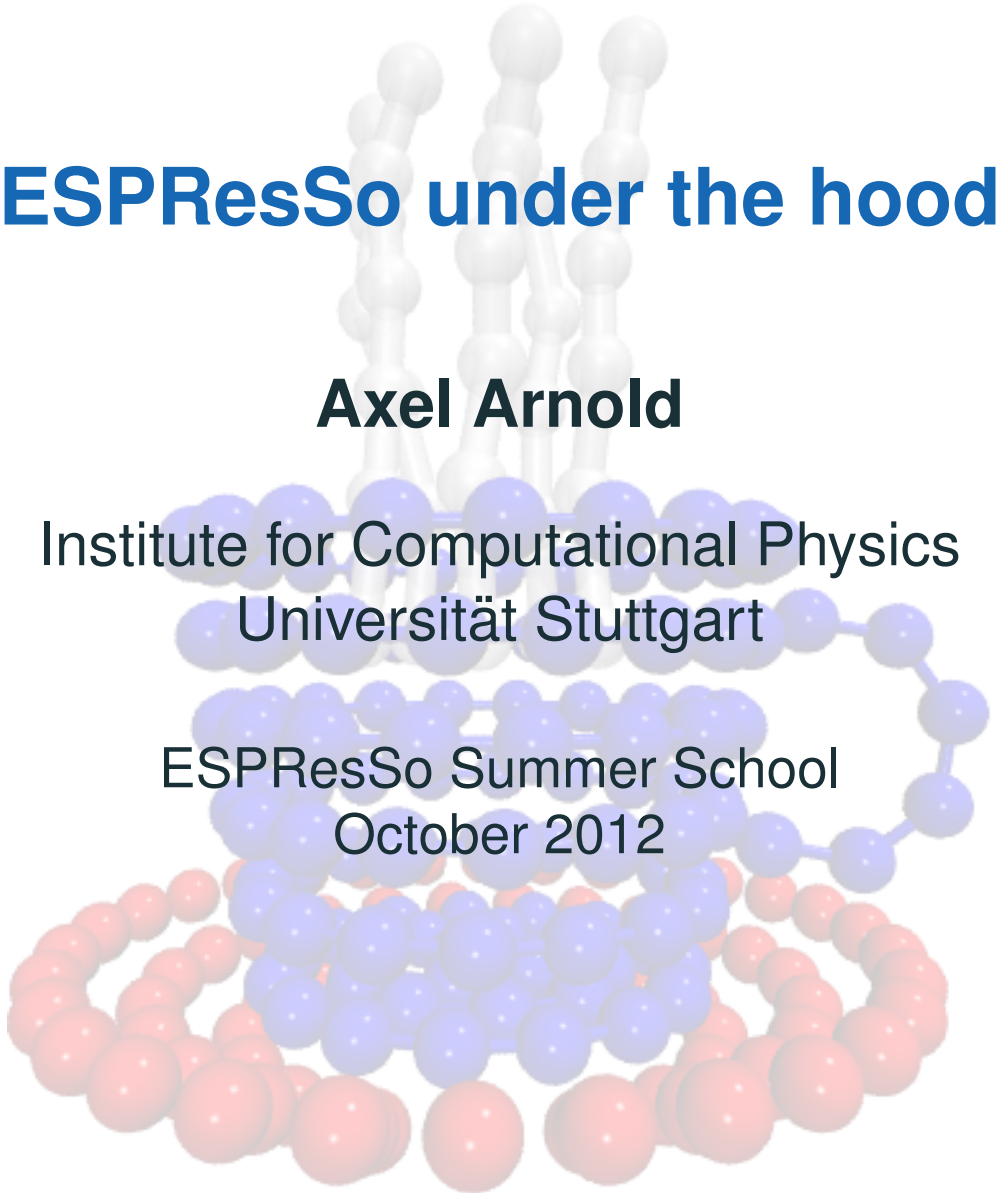
ESPResSo under the hood

Axel Arnold

Institute for Computational Physics
Universität Stuttgart

ESPResSo Summer School
October 2012



Working on the current code: git

Getting the current code

```
git clone git://git.savannah.nongnu.org/espressomd.git
```

- creates source directory `espressomd`
- contains ESPResSo' whole history

History

```
git log
```

```
...
commit 085b7fb0510d05dd5e2cd6fb73983e3eb067cc8d
Author: Axel Arnold <arnolda@icp.uni-stuttgart.de>
Date:   Tue Nov 13 15:01:09 2001 +0000

    MD using TCL.
```

- commits identified by unique number

Working on the current code: git

Getting updates

```
git pull
```

- gets the latest updates
- use `git stash` to temporarily stow away local changes
- `git stash pop` reapplies these local changes
- **CONFLICT** $\Longrightarrow$ manually merge changes if necessary
- differences marked as

```
<<<<<<< Updated upstream
  data->LJ_eps     = 1.0*eps;
=======
  data->LJ_eps     = 2.0*eps;
>>>>>>> Stashed changes
```

Working on the current code: git

Status

```
git status
```

```
# On branch master
nothing to commit (working directory clean)
```

Locally committing changes

```
git add/rm/mv <file>
```

- `add`: mark changes to include into commit
- `rm / mv`: (re-)move a file in filesystem and commit

```
git commit
```

- commits *marked* changes
- opens editor to ask for description of changes
- pull before committing to avoid clashes of commits

Working on the current code: git

Formatting patches

```
git format-patch HEAD^
```

- creates file 0001-commit-description.patch:

```
From b96cf9fa49bb9e7f1794bdf0777a59af04ddcc7e Mon Sep 17 00:00:00 2001
From: Axel Arnold <arnolda@icp.uni-stuttgart.de>
Date: Sun, 7 Oct 2012 18:33:13 +0200
Subject: [PATCH] Changed LJ energy scale

---
 src/lj.c |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/src/lj.c b/src/lj.c
...
```

- send this file to the ESPResSo mailing list

Sending pull requests

- get a public git repository at `github.com`
- send a pull request to espressomd via the web front end

The directory tree

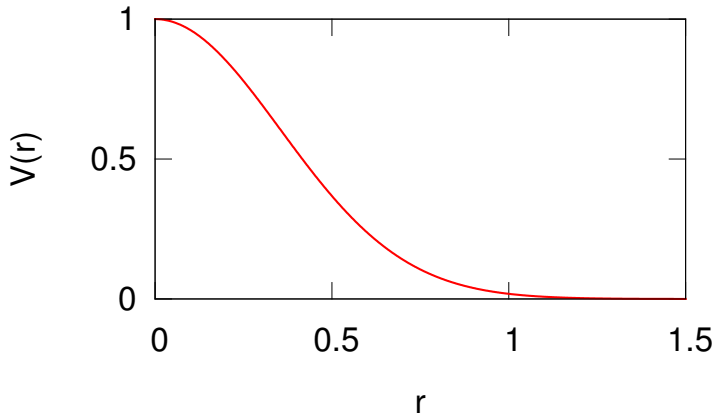
- `src` — source code
  compiles without Tcl, but requires a *carefully* written SIMD C-program to interface
- `src/tcl` — Tcl interface code
- `scripts` — Tcl code that is integral part of ESPResSo, e.g. the `blockfile` command
- `packages` — Tcl packages such as the membrane tools
- `samples` — example scripts
- `testsuite` — short scripts testing particular features. These should make sure features do not break in later releases. `make check` runs all these tests. Do this before sending patches!
- `tools` — some special helpers
- `doc/ug` — LaTeXdocumentation

Relevant source files

Header file + implementation (.h / .c)
Tcl interfaces under `tcl` directory with "_tcl" appended

- `integrate` — main integration loop
- `forces / pressure / energy` — interaction calculations
- `particle_data` — setting / getting particle properties
- `interaction_data` — setting / getting interactions
- `harmonic / hertzian / lj` — simple examples of interactions
- `constraint` — spatial potentials / confinement
- `statistics` — analysis routines
- `global` — `setmd` variables
- `initialize` — hooks that are called if something has changed (global variables, integration loop starts,...)
- `communication` — implements the master-worker communication
- `utils.h` — useful helpers (arrays, rounding,...)

Adding a new (non-bonded) potential



We implement a Gaussian potential

$$V(r) = \begin{cases} \epsilon\, e^{-\frac{1}{2}\left(\frac{r}{\sigma}\right)^2} & r < r_{\text{cut}} \\ 0 & r \geq r_{\text{cut}} \end{cases}$$

- choose a name for `inter`: gaussian
- choose a guard: GAUSSIAN
- choose a template (HERTZIAN or LENNARD_JONES)

What to do

- calculate potential and force: `gaussian.h`
  `add_gaussian_pair_force` and `hertzian_pair_energy`
- set parameters: `gaussian.c` and `gaussian.h`
  `hertzian_set_params`
- make the parameters exist: `interaction_data.h`
  `struct IA_parameters`
- integrate with interactions: `interaction_data.c`
  - include header `gaussian.h`
  - initialize and copy parameters: `initialize_ia_params`
  - make cutoff known: `recalc_maximal_cutoff_nonbonded`
- integrate with force and pressure: `forces.h`
  `calc_non_bonded_pair_force_parts`
- integrate with energy calculation: `energy.h`
  `calc_non_bonded_pair_energy`

What to do

- parse and write the parameters: `tcl/gaussian_tcl.c` and `tcl/gaussian_tcl.h`
  `tclcommand_inter_parse_gaussian` and `tclprint_to_result_GaussianIA`
- add to the interaction parser: `tcl/interaction_data_tcl.c`
  `tclcommand_inter_parse_non_bonded` (macro REGISTER_NONBONDED) and `tclprint_to_result_NonbondedIA`
- add `gaussian.c`, `gaussian.h`, `tcl/gaussian_tcl.h` and `tcl/gaussian_tcl.h` to the build system: `src/Makefile.am`
- add `GAUSSIAN` to the config system: `features.def`
- document Gaussian potential: `doc/ug/inter.tex`

Tcl integration: parsing

```
#include "parser.h"

int some_parser(Tcl_Interp * interp, int argc, char ** argv)
{
  int order; double sig;
  if (argc < 3 ||
      (! ARG_IS_I(1, order)) ||
      (! ARG_IS_D(2, sig))) {
    Tcl_AppendResult(interp, "we need 2 parameters: "
                     "<order> <sigma>", (char *) NULL);
    return TCL_ERROR;
  }
  return TCL_OK;
}
```

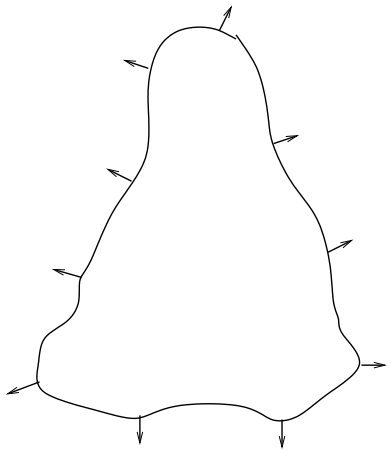
- you will probably need parameters
- parameters in `argc`/`argv`-form just as in `main()`
- `Tcl_Interp` represents the Tcl interpreter

Tcl integration: return value

```
#include "parser.h"

int printSmthToResult(Tcl_Interp *interp, int i, double d)
{
  char buffer[TCL_DOUBLE_SPACE + TCL_INTEGER_SPACE];
  Tcl_ResetResult(interp);

  sprintf(buffer, "%d ", i);
  Tcl_AppendResult(interp, "my result is ",
    buffer, (char *) NULL);
  Tcl_PrintDouble(interp, d, buffer);
  Tcl_AppendResult(interp, " ", buffer, (char *) NULL);

  return TCL_OK;
}
```

- return value is constructed and stored in the interpreter
- this will respect the setting for `tcl_precision`
- make your buffer large enough!

Adding a new constraint



- constraints are external potentials acting on the particles, mostly geometric obstacles
- defined by distance to surface
- uses any standard short-ranged potentials
- choose a name for `constraint`: tux
- choose a constant name: CONSTRAINT_TUX
- use wall constraint (CONSTRAINT_WAL) as template

What to do

- add your constant to `interaction_data.h`
- write `calculate_tux_dist()` in `constraint.c`
- add it to `add_constraints_force()` and `add_constraints_energy()` in `constraint.c`
- integrate in `constraint`-parser in `tcl/constraint_tcl.c`:
  - `tclcommand_constraint_parse_wall`: parser, integrate in `tclcommand_constraint`
  - add to `tclprint_to_result_Constraint`
- if *possible*, also add to: `lb-boundaries.c` and `polymer.c`

Adding analysis routines

What to do

- add calculation to `statistics.c` and `statistics.h`
- in `tcl/statistics_tcl.c`:
  - add parser `tclcommand_analyze_parse_something`
  - register with `tclcommand_analyze` (macro REGISTER_ANALYSIS)

How to get particle properties?

- write a parallel routine (few so far)
- use `partCfg` on the master node
- use `n_configs` and `configs` array to access older positions stored via `analyze push/append`

Particles in ESPResSo

The particle struct

```
typedef struct {
  ParticleProperties p;
  ParticlePosition r;
  ParticleMomentum m;
  ParticleForce f;
  ParticleLocal l;
  IntList bl;
} Particle;
```

- `ParticleProperties`: constants like mass, charge,... present also in ghosts
- `ParticlePosition`: always up-to-date in ghosts, *almost* folded
- `ParticleMomentum`: up-to-date in ghosts if `ghosts_have_v`=1
- `ParticleForce`: ghost force is added to real particle
- `ParticleLocal`: only available with real particles
- bond list `bl`: at real particles only, dynamic integer list

Serial access to particles in ESPResSo

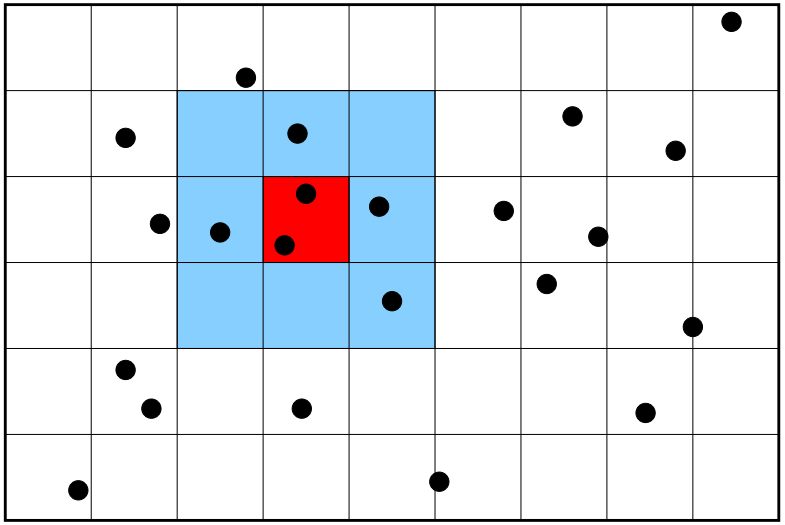
- only readable, writing does not affect the simulation

```
double q_tot = 0;
updatePartCfg(WITHOUT_BONDS);
for (int j=0; j<n_total_particles; j++)
  q_tot += partCfg[j].p.q;
```

- analysis often not time-critical $\Longrightarrow$ serial code sufficient
- `updatePartCfg()` loads particles into `partCfg` on master node
- positions are *unfolded*
- particles can carry bond information (parameter WITH_BONDS) or all appear unbonded (WITHOUT_BONDS)
- `sortPartCfg()` sorts particles if ids are contiguous:

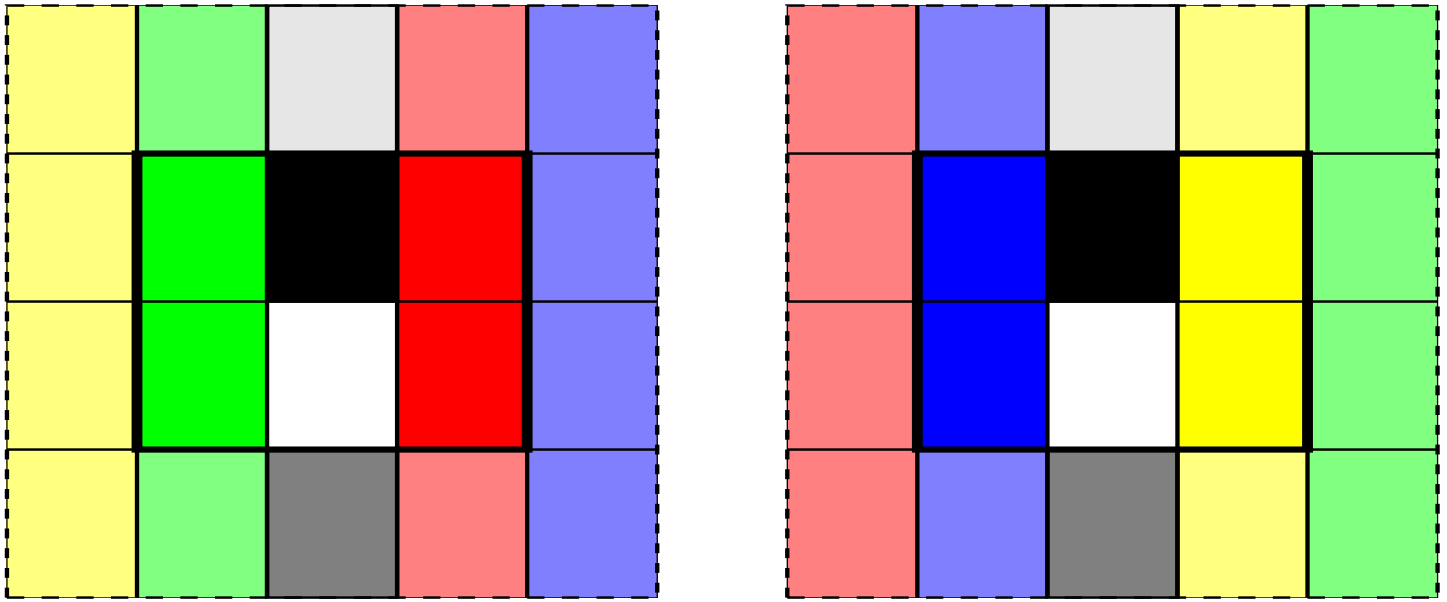
```
if (!sortPartCfg() || n_part <= 42) {
  /* throw error, particles are not contiguous */
}
double q_fortytwo = partCfg[42].p.q;
```

Distributed storage: domain decomposition



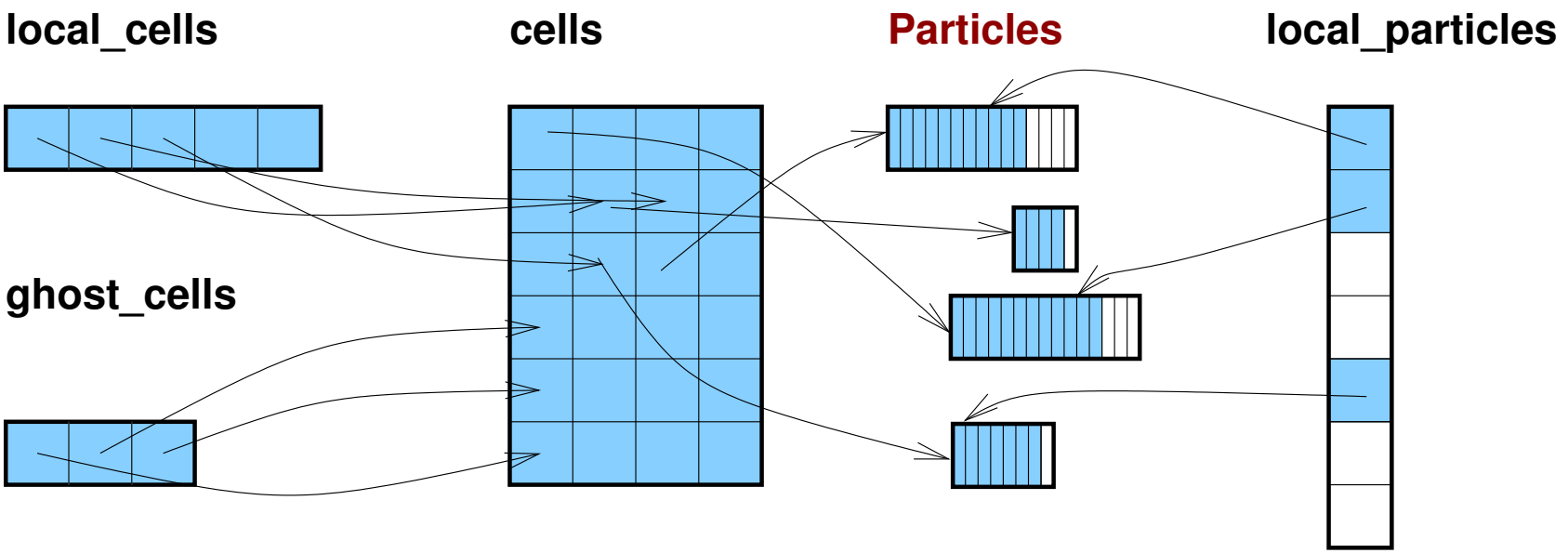
- choose cell size larger than interaction range
- automatically chosen using `recalc_maximal_cutoff`
- only nearest neighbor cells interact
- skin avoids moving particles continously
- speed up short-ranged interactions
  from $\mathcal{O}(N^2)$ to $\mathcal{O}(N\rho r_{\text{cut}}^3)$

Parallelization: ghost particles



- mirror images across processors
- send positions, receive forces
- local mirroring simplifies p. b. c.
- up to 8 copies of a cell on 1 processors

Accessing the distributed particles



- Particles are stored in arrays, the *cells*
- local cells contain real particles
- ghost cells contain *partial* copies from other processors
- `local_particles` points to real and ghost particles

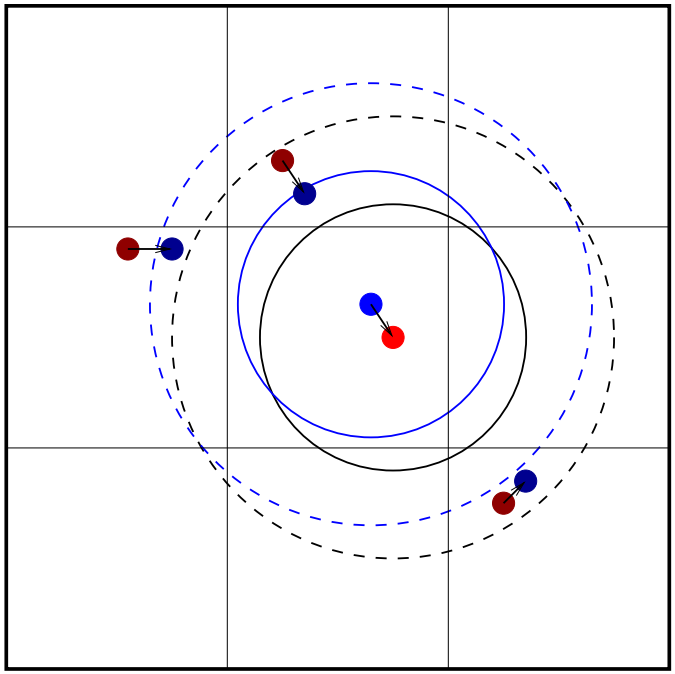
Looping over the particles

loop over real particles

```
Cell *cell;
Particle *p;
int np, c, i;
for (c = 0; c < local_cells.n; c++) {
  cell = local_cells.cell[c];
  p  = cell->part;
  np = cell->n;
  for (i = 0; i < np; i++) {
    do_something(p[i].p.id);
  }
}
```

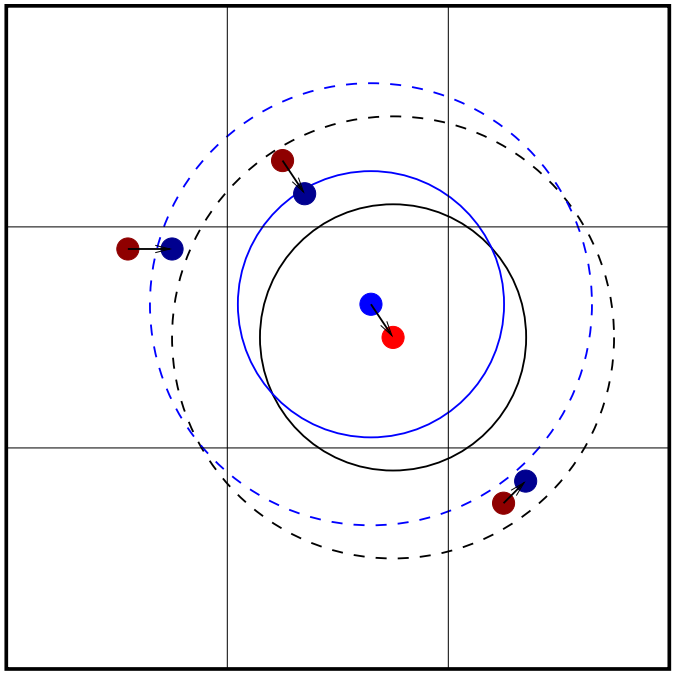
- loop over ghosts analogously with `ghost_cells`
- if domain decomposition, then mesh of cells
  - check `cell_structure.type == CELL_STRUCTURE_DOMDEC`
  - global variable `dd.cell_grid` describes cell arrangement
  - neighbors of all cells are stored in `dd.cell_inter`

Verlet lists



- keep a list of neighbors of each particle
- list range also larger by skin than the interaction range
- updated simultaneously with domain decomposition
- further reduce time by factor $\frac{\pi}{6} = 0.52$

Verlet lists



- Verlet list stored in `dd.cell_inter[i].vList`
- again *only* with domain decomposition
- check `cell_structure.type == CELL_STRUCTURE_DOMDEC` and `dd.use_vList != 0`

Bonds

```
typedef struct {
  int type, num;
  union {
    struct { ... } fene;
    struct { ... } harmonic; ...
  } p;
} Bonded_ia_parameters;
extern Bonded_ia_parameters *bonded_ia_params;
```

Particle's `bl`:

id_1	$part_1$	...	$part_n$	id_2	$part_1$	...	$part_n$	...	0

e. g.

1	5	0

1	5	3	2	5	0

- type is constant from `interaction_data.h`
- id_n is index in table `bonded_ia_params`
- bonds are stored with *one* particle
- use `local_change_bond()` from `particle_data.h`

integrate.c — velocity Verlet integrator

- main integration loop `integrate_vv`
- implements velocity Verlet integrator:

$$v_i\left(t+\frac{1}{2}\delta t\right) = v_i(t) + \frac{1}{2}\delta t F_i(t)/m_i$$

$$r_i(t+\delta t) = r_i(t) + \delta t v_i\left(t+\frac{1}{2}\delta t\right)$$

calculate forces $F_i(t+\delta t) \implies$ `forces.c:force_calc`

$$v_i(t) = v_i\left(t+\frac{1}{2}\delta t\right) + \frac{1}{2}\delta t F_i(t+\delta t)$$

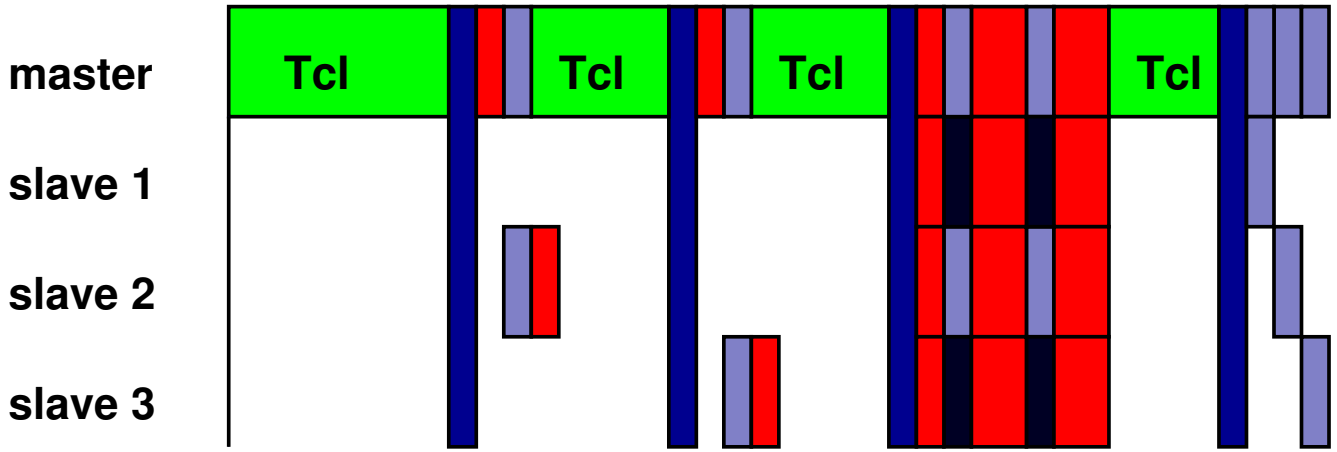
- a number of algorithms hook into the loop (LB, MEMD, virtual particles, AdResS,...)
- be careful: order *does* matter

Tcl integration: adding a global variable

What to do

- add define *with logically following number* to end of defines in `global.h`
- add entry to `fields` in `global.c` *in the place corresponding to the chosen number*
  - address of the physical location of the variable
  - type (at present int or double)
  - vector length, 1 for a scalar
  - name on Tcl level
  - number of letters necessary to uniquely determine this variable
- by default, the variable is read only
- to make it writeable, you need a *write callback*
  - place it in a new .c/.h-file
  - make it accessible by adding it to `register_global_variable` in `tcl/initialize_interpreter.c`

Going parallel: master-slave communication



- Tcl-script is read and executed only on the master node
- master-slave communication in `communication.c`
- other nodes wait in `mpi_loop` (`MPI_BCast`)
- master initiates parallel region using `mpi_issue_request`
- `mpi_loop` calls function from table `slave_callbacks`

Example: adding a particle property

- choose a name: mass
- choose a guard: MASS
- choose communication pattern struct: `ParticleProperty`
- set default value in `init_particle` in `particle_data.c`
- add to struct and export setter function `set_particle_mass` in `particle_data.h`
- write `set_particle_mass`, using `mpi_send_mass`
- `particle_node` contains particle-node mapping
- add `mpi_send_mass` to `communication.h`
- add to `tclcommand_part_print` and `tclprint_to_result_Particle`
- write parser `tclcommand_part_parse_mass` and add to `tclcommand_part_parse_cmd`

Code documentation: inline doxygen

particle_data.h

```
/**  bonds_flag value for updating particle config
     without bonding information */
#define WITHOUT_BONDS 0

/** Capacity of \ref particle_node / \ref local_particles. */
extern int  max_particle_node;

/** Implementation of the tcl command \ref tcl_part.
    This command allows to modify particle data. */
int part(ClientData data, Tcl_Interp *interp,
 int argc, char **argv);
```

- in header file for exported functions, variables and defines
- otherwise in c-file
- allows for cross-referencing, dedicated pages...
- html+LaTeX-like keywords