Pacific Northwest
NATIONAL LABORATORY

Programming MiniMD with MCL

PPoPP ‘22

Alok Kamatar, Rizwan Ashraf, Ryan Friese,
Roberto Gioiosa, Lenny Guo, Gokcen Kestor

U.S. DEPARTMENT OF ENERGY BATTELLE

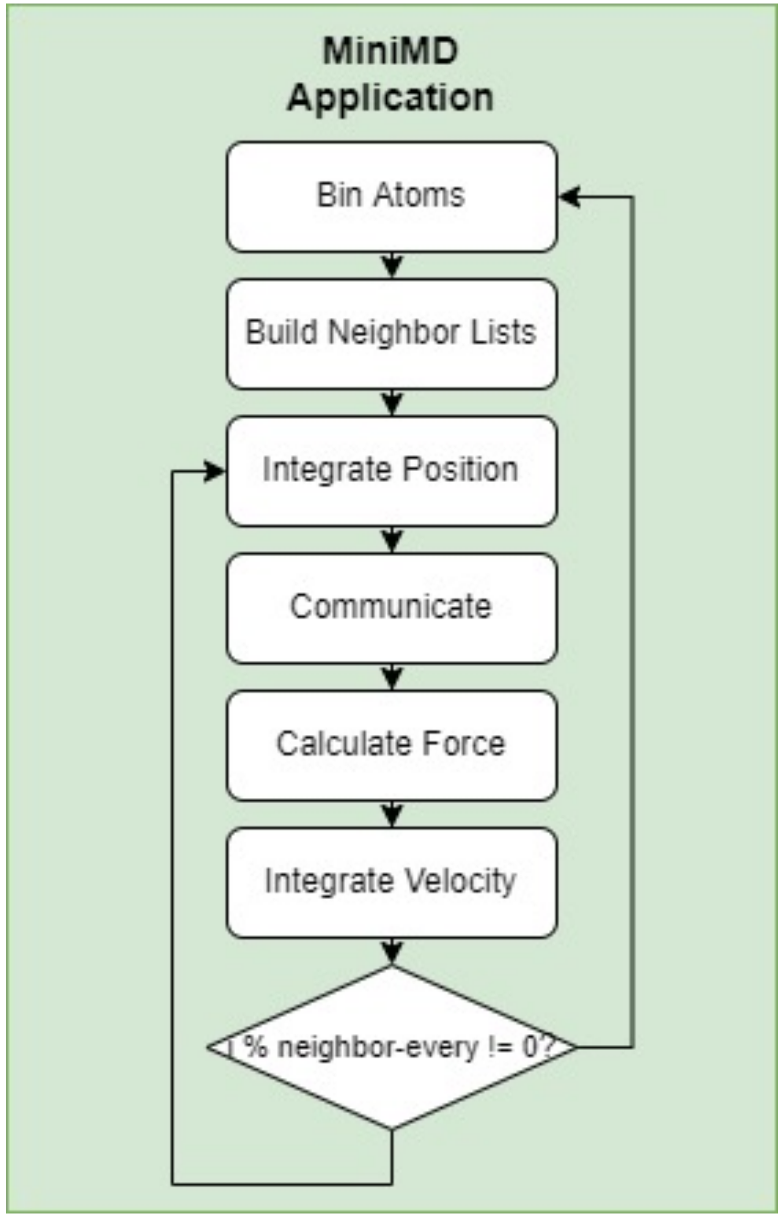
PNNL is operated by Battelle for the U.S. Department of Energy

MiniMD Application Background

- Based off LAMMPS MD code (from Sandia National Lab) - https://github.com/Mantevo/miniMD
- Uses a spatial decomposition to break the problem over multiple processors (or multiple GPUs)
- Uses neighbor lists for the force calculation – reduces work, but relies on random memory accesses
- MCL currently only supports Lennard-Jones interactions for the simulation

MiniMD Structure

- Composed of 6 different kernels/tasks
- MPI - Atoms are divided into different processes -1 process per GPU
- MCL – 1 client process
  - No need for inter-process communication
- Maintain spatial decomposition of atoms - same number of tasks, same memory footprint



MiniMD Application

Bin Atoms

Build Neighbor Lists

Integrate Position

Communicate

Calculate Force

Integrate Velocity

i % neighbor-every != 0?

MiniMD-MCL

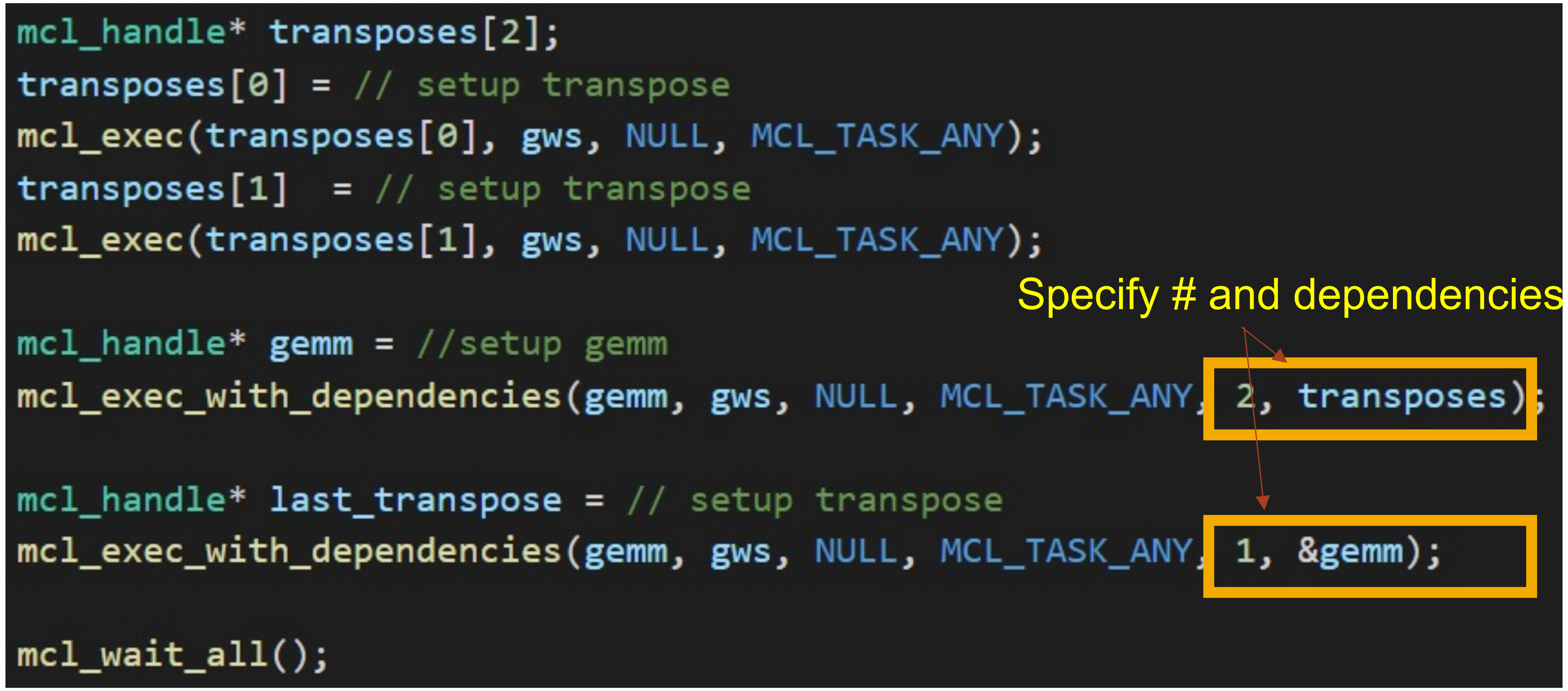
- Advantages:
  - **Kernels reused** from OpenCL
  - **Dynamic Scheduling** – GPU allocation adapts to system use
  - **Portable** –Nvidia GPUs, AMD GPUs, and GPU + Xilinx FPGA
  - **Ease of Use**
- Disadvantages:
  - **Scheduling Overhead**

Preliminaries: Resident Memory

- May need to mark certain data as “resident” to MCL
  - i.e. Multiple tasks use the same piece of data, output of one task is input to another, etc.
- Can mark these arguments with MCL_ARG_RESIDENT.
- MCL_ARG_DYNAMIC – pass data from task to task
- Scheduler manages transfers to correct device
- Allocation **could** exist till:
  - MCL_ARG_DONE flag is passed
  - mcl_unregister_buffer(void* buffer) is called
- Data my be on host, or any device based on scheduler and tasks

Preliminaries: Dependencies

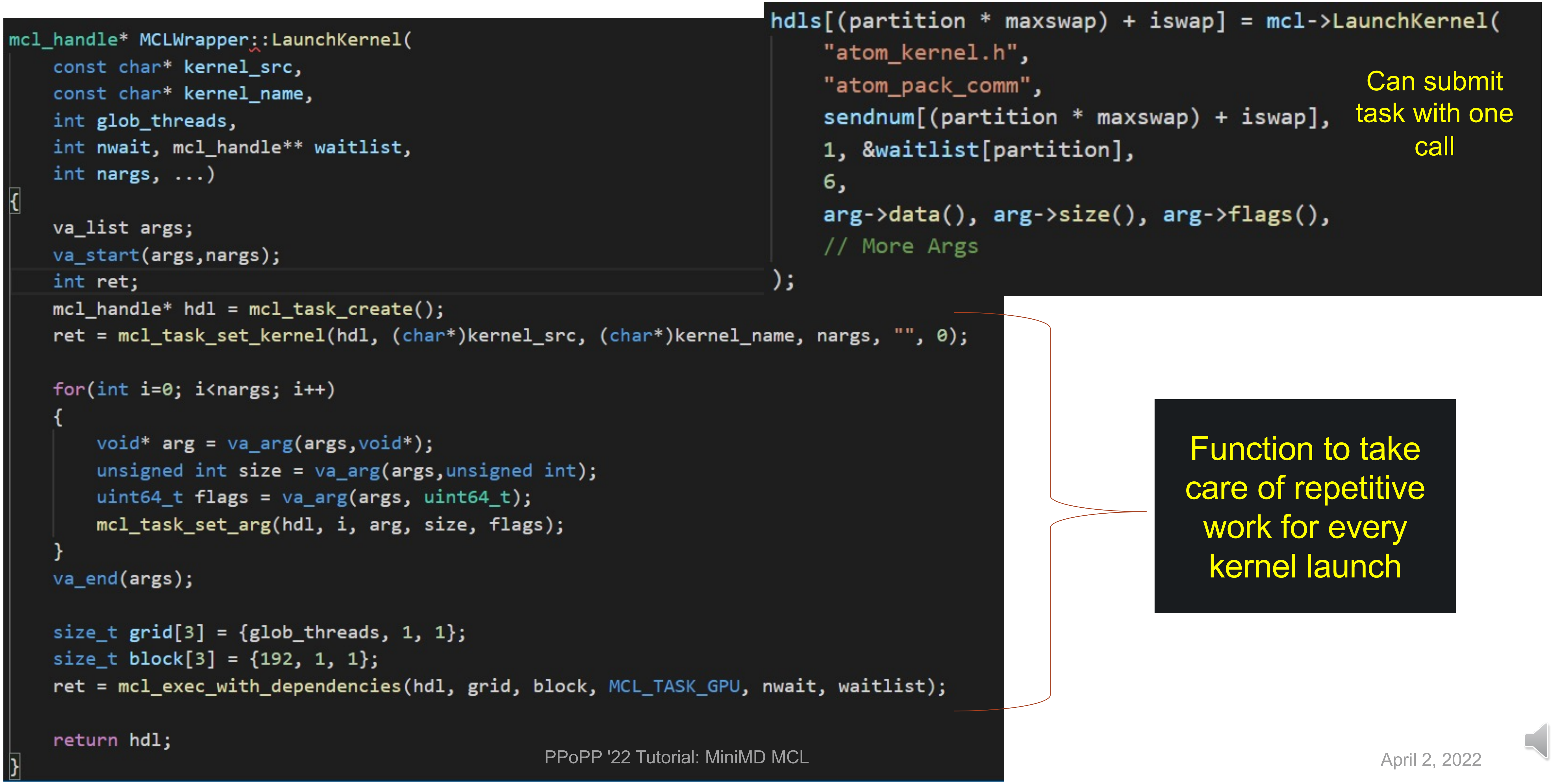
- New API
  - Asynchronous expression of dependencies
  - Quicker turn around between tasks
- Scheduling logic
  - Tasks are scheduled when all immediate dependencies are executing (but before they are finished)



```
mcl_handle* transposes[2];
transposes[0] = // setup transpose
mcl_exec(transposes[0], gws, NULL, MCL_TASK_ANY);
transposes[1]  = // setup transpose
mcl_exec(transposes[1], gws, NULL, MCL_TASK_ANY);

mcl_handle* gemm = //setup gemm
mcl_exec_with_dependencies(gemm, gws, NULL, MCL_TASK_ANY, 2, transposes);

mcl_handle* last_transpose = // setup transpose
mcl_exec_with_dependencies(gemm, gws, NULL, MCL_TASK_ANY, 1, &gemm);

mcl_wait_all();
```

Specify # and dependencies

Preliminaries: Boilerplate



```
mcl_handle* MCLWrapper::LaunchKernel(
    const char* kernel_src,
    const char* kernel_name,
    int glob_threads,
    int nwait, mcl_handle** waitlist,
    int nargs, ...)
{
    va_list args;
    va_start(args,nargs);
    int ret;
    mcl_handle* hdl = mcl_task_create();
    ret = mcl_task_set_kernel(hdl, (char*)kernel_src, (char*)kernel_name, nargs, "", 0);

    for(int i=0; i<nargs; i++)
    {
        void* arg = va_arg(args,void*);
        unsigned int size = va_arg(args,unsigned int);
        uint64_t flags = va_arg(args, uint64_t);
        mcl_task_set_arg(hdl, i, arg, size, flags);
    }
    va_end(args);

    size_t grid[3] = {glob_threads, 1, 1};
    size_t block[3] = {192, 1, 1};
    ret = mcl_exec_with_dependencies(hdl, grid, block, MCL_TASK_GPU, nwait, waitlist);

    return hdl;
}
```

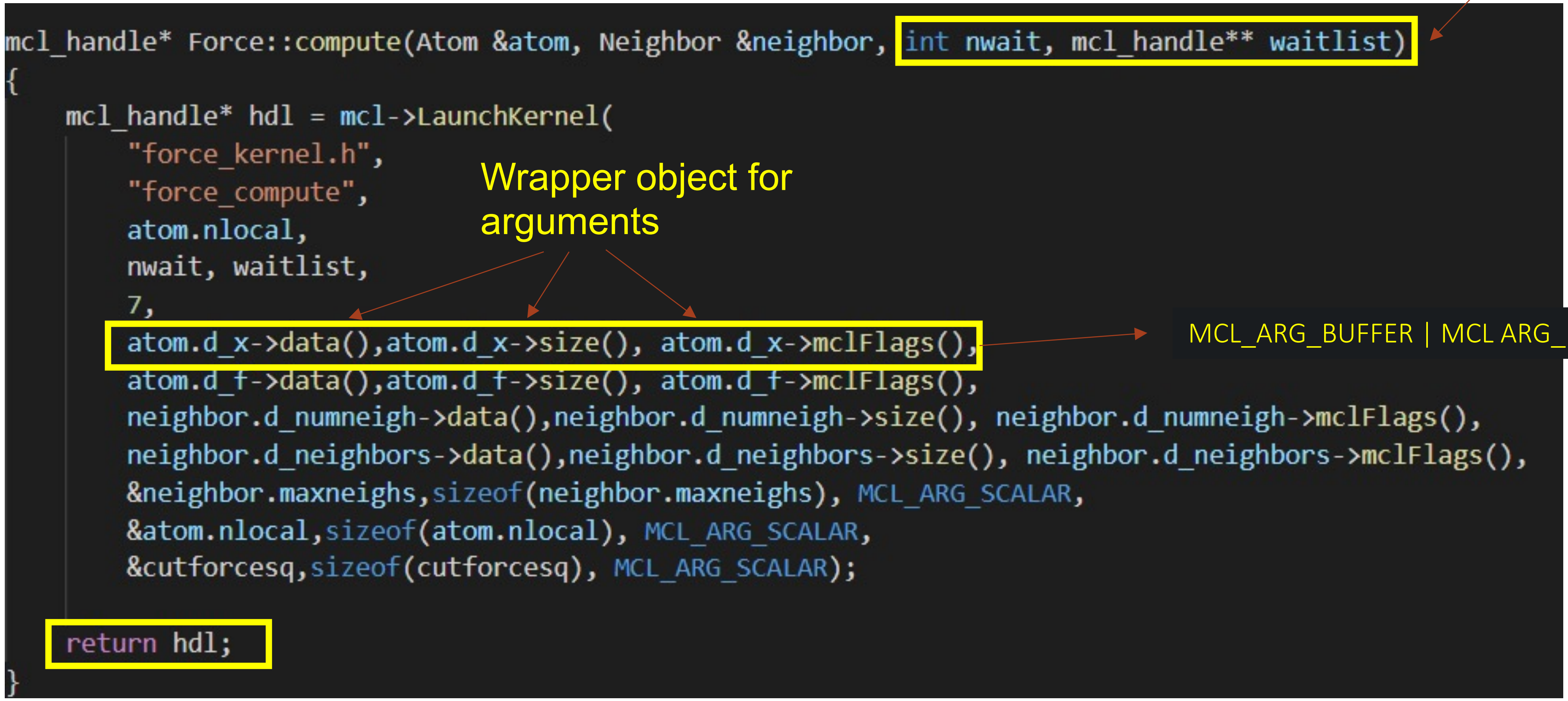
```
hdls[(partition * maxswap) + iswap] = mcl->LaunchKernel(
    "atom_kernel.h",
    "atom_pack_comm",
    sendnum[(partition * maxswap) + iswap],
    1, &waitlist[partition],
    6,
    arg->data(), arg->size(), arg->flags(),
    // More Args
);
```

Can submit task with one call

Function to take care of repetitive work for every kernel launch

Example: Force

Pass around waitlist to manage dependencies

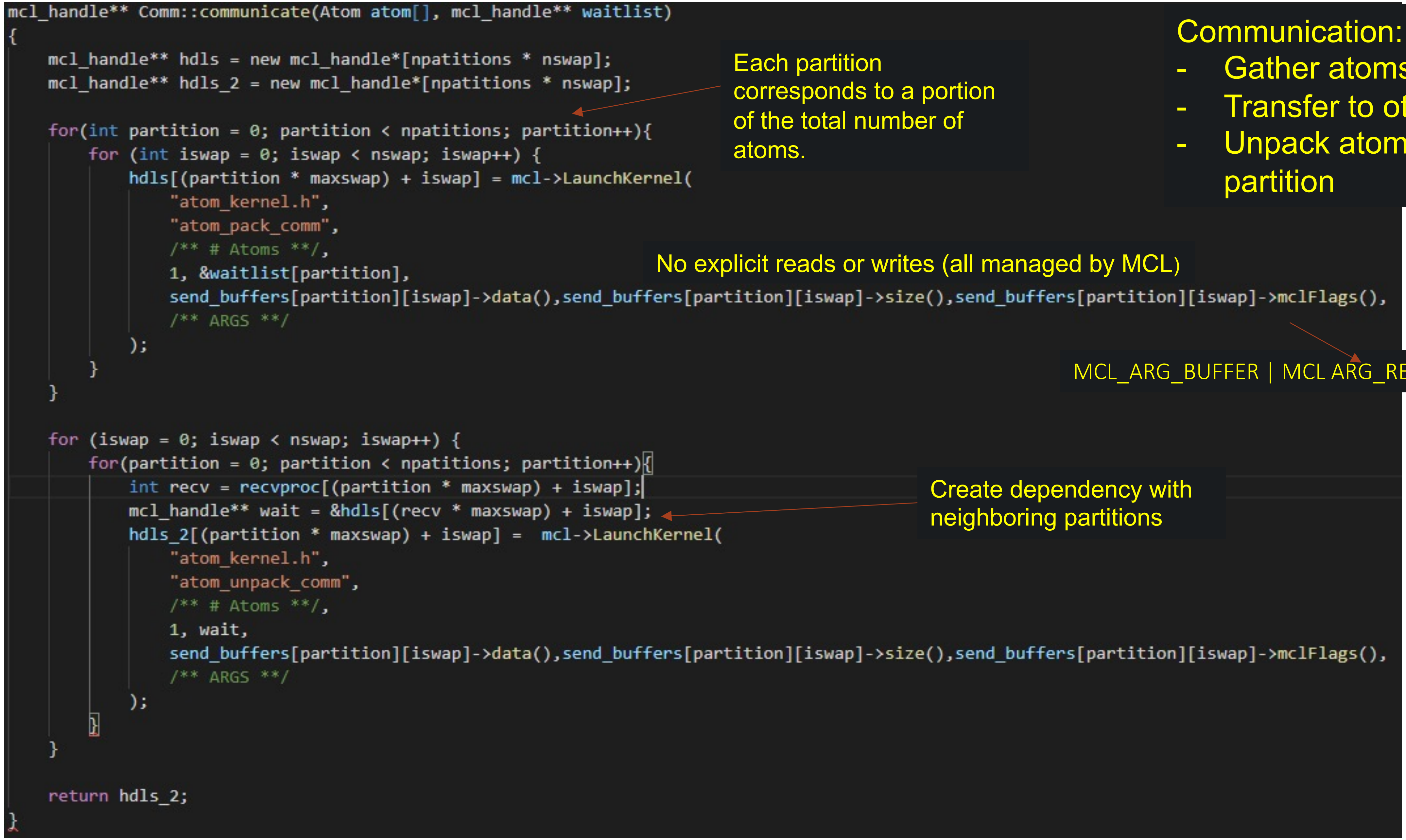


```
mcl_handle* Force::compute(Atom &atom, Neighbor &neighbor, int nwait, mcl_handle** waitlist)
{
    mcl_handle* hdl = mcl->LaunchKernel(
        "force_kernel.h",
        "force_compute",
        atom.nlocal,
        nwait, waitlist,
        7,
        atom.d_x->data(),atom.d_x->size(), atom.d_x->mclFlags(),
        atom.d_f->data(),atom.d_f->size(), atom.d_f->mclFlags(),
        neighbor.d_numneigh->data(),neighbor.d_numneigh->size(), neighbor.d_numneigh->mclFlags(),
        neighbor.d_neighbors->data(),neighbor.d_neighbors->size(), neighbor.d_neighbors->mclFlags(),
        &neighbor.maxneighs,sizeof(neighbor.maxneighs), MCL_ARG_SCALAR,
        &atom.nlocal,sizeof(atom.nlocal), MCL_ARG_SCALAR,
        &cutforcesq,sizeof(cutforcesq), MCL_ARG_SCALAR);

    return hdl;
}
```

Wrapper object for arguments

MCL_ARG_BUFFER | MCL ARG_RESIDENT | MCL_ARG_DYNAMIC

Example: Communication



```
mcl_handle** Comm::communicate(Atom atom[], mcl_handle** waitlist)
{
    mcl_handle** hdls = new mcl_handle*[npatitions * nswap];
    mcl_handle** hdls_2 = new mcl_handle*[npatitions * nswap];

    for(int partition = 0; partition < npatitions; partition++){
        for (int iswap = 0; iswap < nswap; iswap++) {
            hdls[(partition * maxswap) + iswap] = mcl->LaunchKernel(
                "atom_kernel.h",
                "atom_pack_comm",
                /** # Atoms **/,
                1, &waitlist[partition],
                send_buffers[partition][iswap]->data(),send_buffers[partition][iswap]->size(),send_buffers[partition][iswap]->mclFlags(),
                /** ARGS **/
            );
        }
    }

    for (iswap = 0; iswap < nswap; iswap++) {
        for(partition = 0; partition < npatitions; partition++){
            int recv = recvproc[(partition * maxswap) + iswap];
            mcl_handle** wait = &hdls[(recv * maxswap) + iswap];
            hdls_2[(partition * maxswap) + iswap] =  mcl->LaunchKernel(
                "atom_kernel.h",
                "atom_unpack_comm",
                /** # Atoms **/,
                1, wait,
                send_buffers[partition][iswap]->data(),send_buffers[partition][iswap]->size(),send_buffers[partition][iswap]->mclFlags(),
                /** ARGS **/
            );
        }
    }

    return hdls_2;
}
```

Communication:
- Gather atoms on boundary of partition
- Transfer to other partition
- Unpack atoms to proper place in partition

Each partition corresponds to a portion of the total number of atoms.

No explicit reads or writes (all managed by MCL)

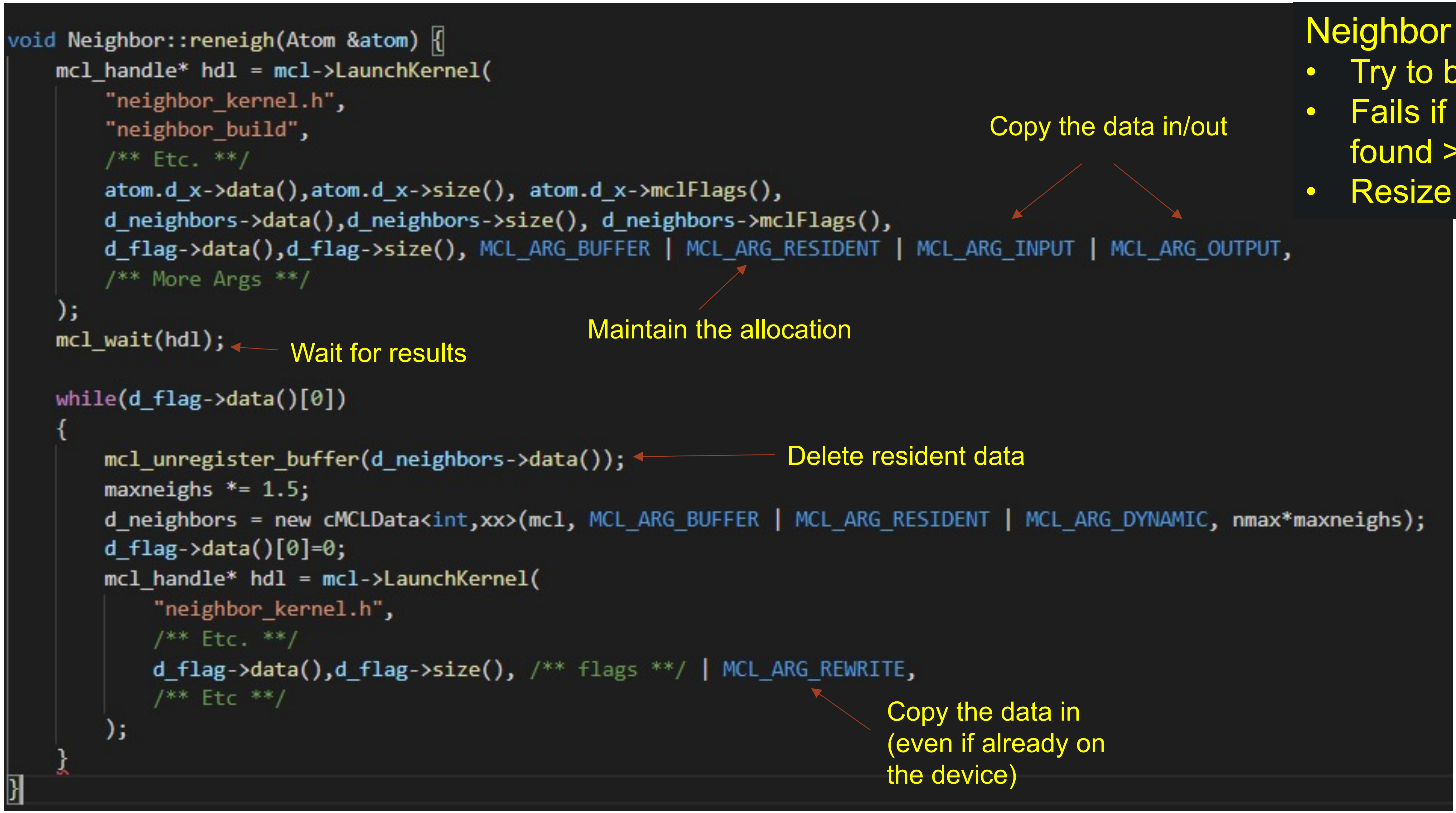
MCL_ARG_BUFFER | MCL ARG_RESIDENT | MCL_ARG_DYNAMIC

Create dependency with neighboring partitions

Example: Neighboring

Neighbor Build:

- Try to build neighbor lists
- Fails if num neighbors found > max neighbors
- Resize and retry on failure



```
void Neighbor::reneigh(Atom &atom) {
    mcl_handle* hdl = mcl->LaunchKernel(
        "neighbor_kernel.h",
        "neighbor_build",
        /** Etc. **/
        atom.d_x->data(),atom.d_x->size(), atom.d_x->mclFlags(),
        d_neighbors->data(),d_neighbors->size(), d_neighbors->mclFlags(),
        d_flag->data(),d_flag->size(), MCL_ARG_BUFFER | MCL_ARG_RESIDENT | MCL_ARG_INPUT | MCL_ARG_OUTPUT,
        /** More Args **/
    );
    mcl_wait(hdl);

    while(d_flag->data()[0])
    {
        mcl_unregister_buffer(d_neighbors->data());
        maxneighs *= 1.5;
        d_neighbors = new cMCLData<int,xx>(mcl, MCL_ARG_BUFFER | MCL_ARG_RESIDENT | MCL_ARG_DYNAMIC, nmax*maxneighs);
        d_flag->data()[0]=0;
        mcl_handle* hdl = mcl->LaunchKernel(
            "neighbor_kernel.h",
            /** Etc. **/
            d_flag->data(),d_flag->size(), /** flags **/ | MCL_ARG_REWRITE,
            /** Etc **/
        );
    }
}
```

Copy the data in/out

Maintain the allocation

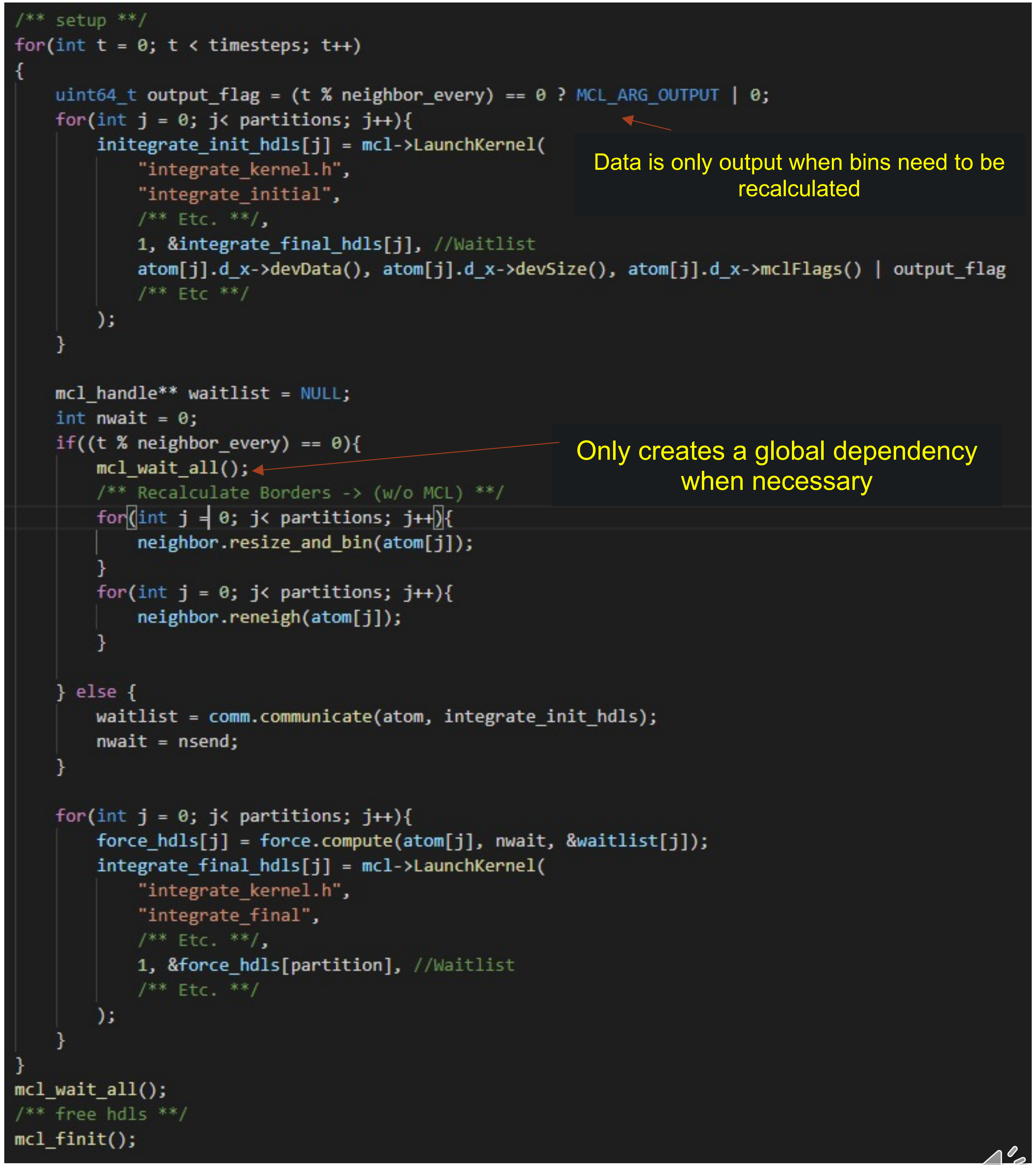
Wait for results

Delete resident data

Copy the data in (even if already on the device)

Example: Integrate

Integrate:
- Step through time calculating position, force, and velocity
- Launch calculations for all partitions asynchronously
- Every ~20 timesteps, recalculate the partitions of the atoms

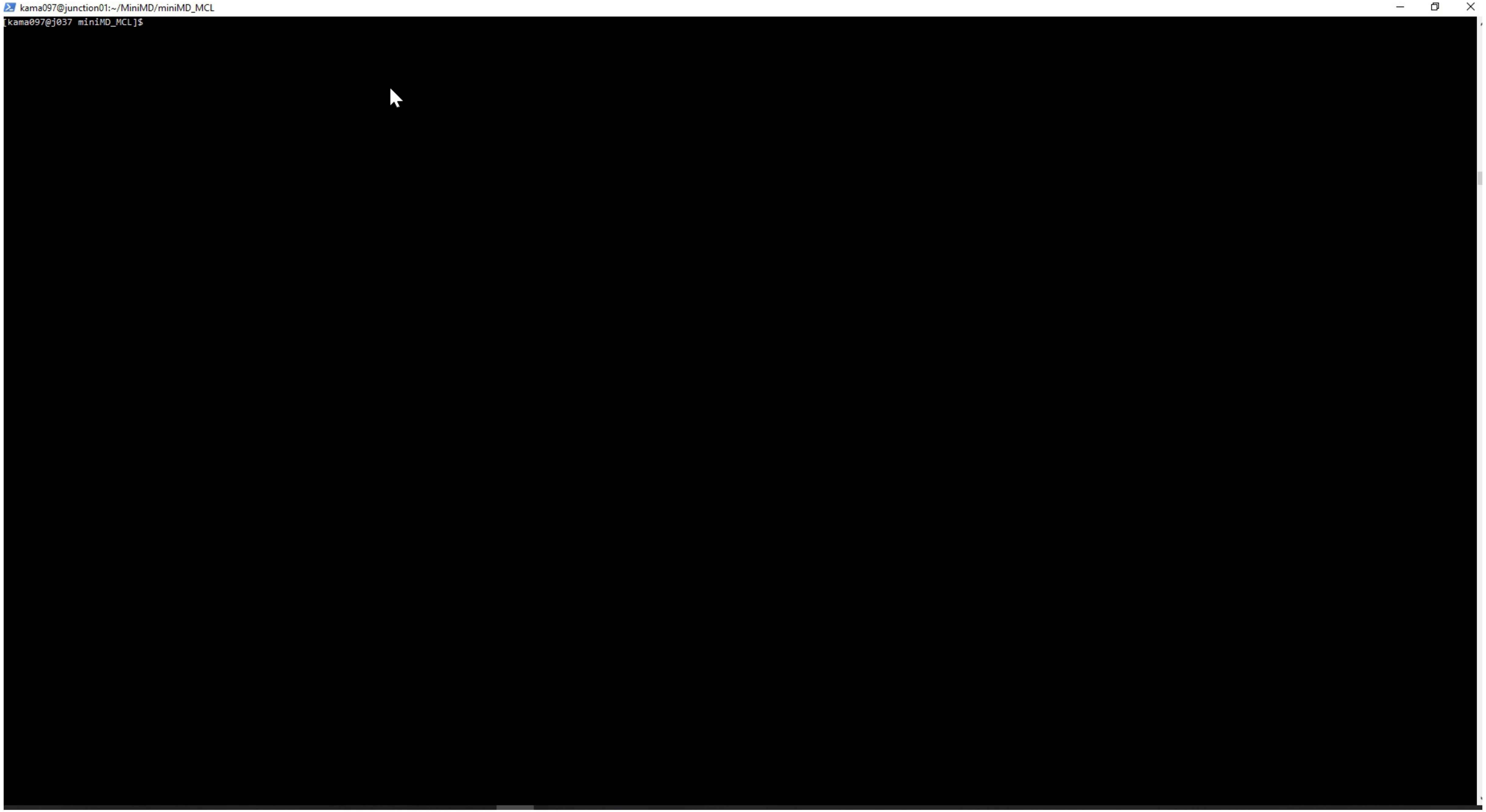


```
/** setup **/
for(int t = 0; t < timesteps; t++)
{
    uint64_t output_flag = (t % neighbor_every) == 0 ? MCL_ARG_OUTPUT | 0;
    for(int j = 0; j< partitions; j++){
        initegrate_init_hdls[j] = mcl->LaunchKernel(
            "integrate_kernel.h",
            "integrate_initial",
            /** Etc. **/,
            1, &integrate_final_hdls[j], //Waitlist
            atom[j].d_x->devData(), atom[j].d_x->devSize(), atom[j].d_x->mclFlags() | output_flag
            /** Etc **/
        );
    }

    mcl_handle** waitlist = NULL;
    int nwait = 0;
    if((t % neighbor_every) == 0){
        mcl_wait_all();
        /** Recalculate Borders -> (w/o MCL) **/
        for(int j = 0; j< partitions; j++){
            neighbor.resize_and_bin(atom[j]);
        }
        for(int j = 0; j< partitions; j++){
            neighbor.reneigh(atom[j]);
        }

    } else {
        waitlist = comm.communicate(atom, integrate_init_hdls);
        nwait = nsend;
    }

    for(int j = 0; j< partitions; j++){
        force_hdls[j] = force.compute(atom[j], nwait, &waitlist[j]);
        integrate_final_hdls[j] = mcl->LaunchKernel(
            "integrate_kernel.h",
            "integrate_final",
            /** Etc. **/,
            1, &force_hdls[partition], //Waitlist
            /** Etc. **/
        );
    }
}
mcl_wait_all();
/** free hdls **/
mcl_finit();
```

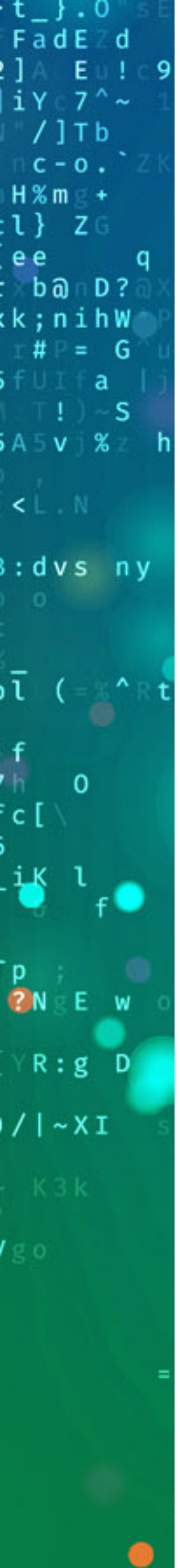
Data is only output when bins need to be recalculated

Only creates a global dependency when necessary

Demo



```
[kama097@j037 miniMD_MCL]$
```



Pacific Northwest
NATIONAL LABORATORY

Thank you