Effective Parallelisation with Kokkos

Shendruk Group Meeting

Timofey Kozhukhov, NBIA - July 2026

UNIVERSITY OF COPENHAGEN

View these slides online

Table of contents

What have I been up to since my PhD?

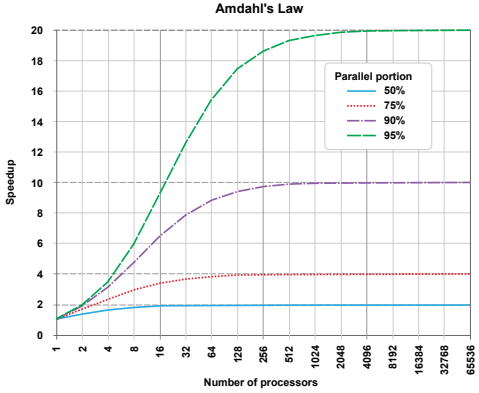
- **Submitted anchored colloids paper**
  - arXiv:2503.19664 - Pending resubmission
- **Studied new technologies and design patterns for simulation and analysis**
- **Designed and implemented MPCD code for Kristian's group: `raMPCD`**
  - Design goal: high level expansion by academics, while retaining performance
  - Implemented core MPCD functionality
  - ***Current focus:*** Finishing implementing arbitrary MD-like solutes (*i.e.* swimmers)
  - ***Here this week to:*** Produce design document for oriented fluids with *Gabriel*
- **Working with PhD students on their projects**
  - Multi-monomer TADPole model — Motile n-Monomer (MnM) swimmers
  - Planned papers:
    - *(Current)* Both: Initial characterisation of MnM
    - Jakub: Polydispersity in MnM length
    - Michela: Polydispersity in MnM chirality, curvature based sorting

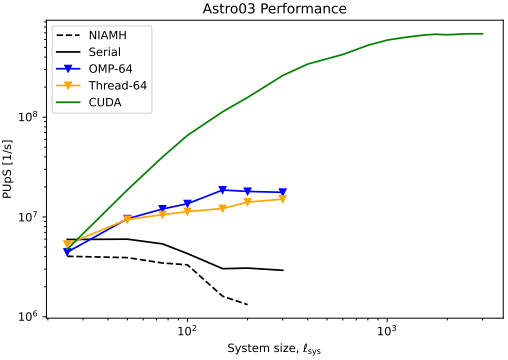
Parallelisation enables faster, larger, and more complex simulations

- Modern computing hardware has many *cores* executing instructions in parallel
- *Parallel programming* is the process of writing code that leverages this hardware
  - As opposed to typical *serial* programming, where instructions execute one at a time
- Generally speaking: Programmer writes *functions* or *kernels* which are passed to *software threads*, and then mapped to *hardware cores* by the OS or GPU drivers
- Usually parallelisation is done in lower level languages (*e.g.*, `C++`, `Fortran`)
  - ...but now, also in higher level languages with libraries (*e.g.* `cupy`, `numba` for `Python`)
  - Analysis scripts with long runtimes are great candidates for parallelisation too!
- Parallelisation allows for:
  - **Speedup** of existing codes
    - More cores = more instructions executed per second = shorter simulation times
  - **Larger system sizes** for the same execution time
    - Shorter simulation times → larger system sizes are possible
  - **More complex simulations** can be implemented without a runtime penalty
    - "Simpler" simulations run faster → complexity can be increased

Parallelisation is not a silver bullet for poor quality codes

- Amdahl's law: Speed-up from parallelisation is sub-linear with thread count
- Parallelisation $\neq$ optimisation





Numerous APIs & architectures for parallelisation exist

- `std::thread` — Raw `C++11` API for CPU parallelisation
  - ✓ Works on any `C++>11` compiler
  - ✓ Can be adapted to any arbitrary problem
  - ✗ Very low-level — Lots of boilerplate, manual management of threads
- `OpenMP` — CPU parallelisation API (`C`/ `C++`/ `Fortran`)
  - ✓ Implemented by all modern compilers
  - ✓ GPU support (that gets better with each version)
  - ✓ Designed to "easily" parallelise existing serial code...
  - ✗ ...but not well suited for **all** codes (*e.g.* ensure independence of parallel regions)
  - ✗ Limited control over shared memory → Error prone (*race conditions*, *deadlocks*, *etc.*)
- `CUDA` — API for programming NVIDIA GPUs (`C`/ `C++`/ `Fortran`)
  - ✓ GPUs are incredibly fast (for "appropriately written" code)
  - ✓ Easily the best GPU-compute API
  - ✗ Necessitates a different *design mentality* (architecture, minimising *warp divergence*, *etc.*)
  - ✗ Low-level **AND** proprietary:
    - ✗ Mixed memory management (*host* vs *device* memory)
    - ✗ Debugging and optimising is a nightmare

Each of these APIs has its own syntax and semantics for parallelisation!

Writing parallel code is more challenging than serial code

- **Algorithm design** — *Partition work* into independent tasks; minimize *synchronisation*
- **Memory hierarchy** — More complex, especially for GPUs
  - GPU: Distinct *device* and *host* memory
    - Copies between device and host memory are very time consuming
    - `CUDA`'s shared *Unified Virtual Memory* (UVM) doesn't entirely solve this
  - Easy to hit *race conditions* with improper memory management
- **Memory access patterns** — *Data locality*, *memory coalescing* and *cache hits*
  - Memory layout often limits performance more than arithmetic or algorithm design
  - *e.g.* Optimal *strides* are architecture dependent
    - *i.e.* "`for x for y`" vs "`for y for x`" when accessing an array `A[x][y]`
  - ***Aside:*** *MPCD is algorithmically memory bound (Halver et al., Parallel Computing, 2023)*
- **Debugging** — Parallel codes are incredibly difficult to debug effectively
  - Numerous threads accessing same variables at the same time
  - *e.g.*, currently *2+ months* debugging one complex bug in `raMPCD`
- **APIs** — All of the above is *API dependent*

We can try solving most of these problems using a library...

Kokkos— A library for performance portability

- A (pseudo-) high-level templated library for C++
- Motivated by the *Exascale computing project* - Designed to write *"performance portable"* scientific code across HPCs with vastly different hardware
  - *"Performance"*: Designed for high performance parallel applications...
  - *"Portable"*: ...where one codebase can run on multiple architectures

⇒ Kokkos is designed in a *"single-source"* manner, where one code can run anywhere...

- ...but, generally speaking, best to use the design mentality of CUDA

- Well established in HPC communities:
  - In development for a decade across multiple US DoE national labs
  - Has contributed to the C++ STL
  - Used in LAMMPS, Trilinos, and other solvers
- Great platform-agnostic debugging and profiling suite

How?

- Kokkos abstractions allow you to account for different hardware
- Provides a unified high-level interface for parallelisation
- Manages memory and memory access patterns for you (*to an extent*)

Kokkos is not a perfect solution

- Kokkos has poor native *vectorisation*
  - All modern processors make use of vectorisation to enable *SIMD parallelism*
  - Kokkos relies on compiler auto-vectorisation, which may not be reliable
  - Kokkos provides some SIMD-friendly types...
    - ...**but** they are unwieldy and do not natively integrate with parallel regions
- Kokkos's technical implementation is detrimental to "modern" CMake processes
  - ...though learning to use CMake is enough of a detriment
- The fundamental Kokkos data structure is arguably more complex than necessary
- No trivial containers for alike-types
  - *i.e.*, groupings of equal length arrays
  - Though Cabana implements a variant of this
- Kokkos is **NOT** portable with respect to MPI

From a Sandia National Labs presentation on osti.gov.

Despite this, Kokkos was the most compelling solution for us.

Kokkos abstracts away hardware details

Parallelisation involves two important hardware-related details:

1. Where is the data stored?
   - RAM/ host memory? VRAM/ device memory? UVM?
   - ➜ Kokkos abstracts this as *memory spaces* ***types***
2. Where *and how* are the kernels executed?
   - Serially? Using OpenMP? CUDA? HIP? SYCL?
   - ➜ Kokkos abstracts this as *execution spaces* ***types***

Kokkos memory space types:

- `Kokkos::HostSpace`
- `Kokkos::CudaSpace`
- `Kokkos::SharedSpace`
- **`Kokkos::DefaultMemorySpace`**

Kokkos execution space types:

- `Kokkos::Serial`
- `Kokkos::OpenMP`
- `Kokkos::Cuda`
- **`Kokkos::DefaultExecutionSpace`**

These are passed as *template arguments* to Kokkos functions and classes

Abstractions allow Kokkos codes to be ran on multiple architectures

Two (non-exhaustive examples) of how you can run code with Kokkos

Compile time:

- Kokkos usually compiled with CMake
- Takes arguments for which memory and execution spaces to build with
  - -DKokkos_ENABLE_OPENMP=ON
  - -DKokkos_ENABLE_CUDA=ON
- Write most of your Kokkos code using *default spaces*
- If using an *embedded build* (*i.e.*, Kokkos is downloaded by CMake), then you can limit the build to only the space you want to run on
  - Effectively switching between spaces at compile time

Run time:

- Kokkos can be built with multiple execution spaces enabled
- If your simulation class is implemented appropriately, then you can switch between execution spaces at run time
  - *e.g.* use program arguments to set the execution space
- *Remember:* Kokkos uses templates, which must be evaluated at compile-time, and Kokkos spaces are *types*
  - Non-trivial implementation required
  - Templated simulation class **with** polymorphic dispatch works

Example: Compile time architecture selection

Run with OpenMP (bash):

```
mkdir build; cd build;  # make build directory (CMake requirement)
cmake -DKokkos_ENABLE_OPENMP=ON ..  # build with OMP backend only
make
./my_program  # will run with OMP
```

Run with CUDA (bash):

```
mkdir build; cd build;
cmake -DKokkos_ENABLE_CUDA=ON ..  # build with CUDA backend (and serial)
make
./my_program  # will run with CUDA
```

Example: Run time architecture selection (1/2)

This is more difficult — In this solution, use polymorphic dispatch with templates for this Assuming Kokkos is build seperately with Serial, `OpenMP`, and `CUDA` enabled:

Simulation class (`C++`):

```
class ISimulationBase {  // interface class
public:
    virtual void run() = 0;  // pure virtual function - intended to be overridden
    virtual ~SimulationBase() = default;  // virtual destructor
};

template <typename ExecSpace, typename MemSpace>  // templated for exec and mem spaces
class Simulation : public ISimulationBase {  // inherit from an interface class
public:
    void run() override;  // launches the simulation - overrides the interface class

    /* ... rest of the class should be implemented using the template arguments ... */
}
```

Shouldn't manually initialise it, instead...

Example: Run time architecture selection (2/2)

This is more difficult — In this solution, use polymorphic dispatch with templates for this Assuming Kokkos is build seperately with Serial, OpenMP, and CUDA enabled:

Inside main function (C++):

```
/* Assuming arguments handled above... */

ISimulationBase* sim;  // base class pointer - Good boys and girls use smart pointers!

if (args.run_mode == SERIAL) {  // factory pattern for different spaces
    sim = new Simulation<Kokkos::Serial, Kokkos::HostSpace>(args);
} else if (args.run_mode == OMP) {
    sim = new Simulation<Kokkos::OpenMP, Kokkos::HostSpace>(args);
} else if (args.run_mode == CUDA) {
    sim = new Simulation<Kokkos::Cuda, Kokkos::CudaSpace>(args);
} /* ...continue factory for other spaces... */

sim->run();  // Polymorphism means this run function will make use of the appropriate
             // spaces
```

Kokkos provides simple interfaces for launching parallel code

Multiple parallel paradigms are available in Kokkos:

1. `Kokkos::parallel_for`
   - Traditional for loop, but each iteration is parallelised (and runs independently)
   - Kokkos handles dispatch of kernel to individual threads on appropriate execution space.
2. `Kokkos::parallel_reduce`
   - Parallel *reduction* operation
   - Similar to `parallel_for`, but kernels compute a result to be combined to a single value
   - *e.g.* compute an average, sum, or other aggregate value across threads
3. `Kokkos::parallel_scan`
   - Performs a *prefix scan* operation
   - Similar to `parallel_reduce`, but returns cumulative results across threads
   - *e.g.*, running totals/ sums across iterations

- `Kokkos::RangePolicy<...>(...)` objects tell Kokkos how to dispatch kernels
  - Template argument gives the *execution space*
  - Object arguments give the *iteration range*

Kokkos expects kernels as functors (or lambdas)

Functor = function object (struct or class)

- *i.e.* a class with an overloaded `operator(int i)` ← Kernel definition goes here!

```
struct AtomForceFunctor {  // Instance of the functor - Can also be a class
    // ALL data used by the functor must be given as members
    ForceType _atomForces;  // The "output array" the functor writes to
    AtomDataType _atomData;  // Misc "input data" used by the functor

    // Constructor - Needs to initialise and copy args to the member variables
    AtomForceFunctor(/* args */){...}

    KOKKOS_INLINE_FUNCTION  // indicate to compiler next function can be run in parallel
    // The operator() function: The kernel that Kokkos runs
    void operator()(const int64_t atomIndex) const {  // atomIndex is the iterator var
        _atomForces[atomIndex] = calculateForce(_atomData);  // what runs on each thread
    }
};
```

Functors are bit verbose, so can also use *lambdas* (a "concise" way of writing functors)

- *Aside:* Fiddly to use when launching parallel regions inside class members

⇒ `raMPCD` typically uses functors for clarity

Example: Launch a parallel_for using previous kernel

```
// Initialise the functor with the data it requires
AtomForceFunctor kernel_functor(atomForces, data);

Kokkos::parallel_for(  // Prepare a parallel_for dispatch
    "AtomForceFunctor",  // Name of the kernel (for Kokkos debugging tools)
    Kokkos::RangePolicy<ExecSpace>(0, numberOfAtoms),  // Range and execspace to use
    kernel_functor  // What kernel Kokkos should dispatch
);
```

Or, using a lambda instead:

```
Kokkos::parallel_for(  // Same dispatch function
    "AtomForceLambda",
    numberOfAtoms,  // alias for RangePolicy<DefaultExecutionSpace>(0, numberOfAtoms)
    KOKKOS_LAMBDA (int64_t atomIndex) {  // lambda equivelant of the previous functor
        atomForces[atomIndex] = calculateForce(atomData);
    }
);
```

Note: Lambdas will only use variables in their *local scope*

Kokkos manages memory for multiple architectures

- Kokkos provides a `Kokkos::View<Type, MemSpace>` class for handling data
  - Templated argument for *arbitrary base data type* (*e.g.* `int*`, `float*`, `float[3]*`)
  - Templated argument for *memory space* (*e.g.*, `Kokkos::HostSpace`)
- Broadly acts similar to a `std::vector`
- Will **store data in optimal stride** for the given memory space!
- Typical to set up *type aliases* with `using`:

```
using host_view_type = Kokkos::View<double**, Kokkos::HostSpace>;
using def_view_type = Kokkos::View<double***>;  // default memory space - equiv to:
using def_view_type = Kokkos::View<double***, Kokkos::DefaultMemorySpace>;
```

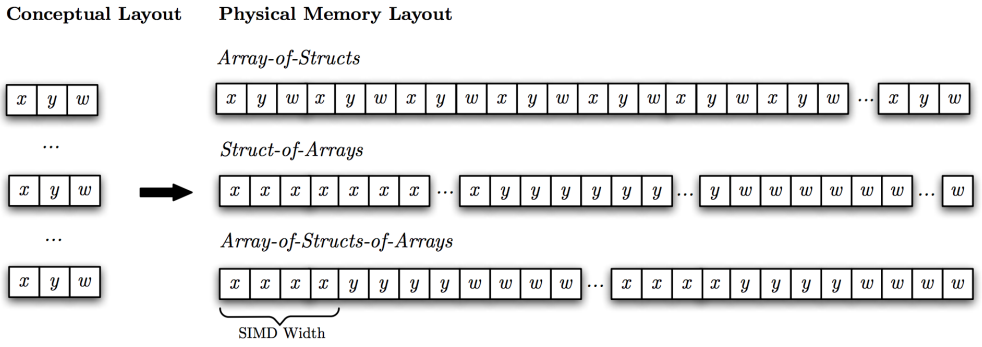
- *GPU relevant:* `Kokkos` also provides (host) *mirror views* linked to a (device) view
  - Designed for easier interaction with GPU/ device memory (initialising, saving to disk, *etc.*)
  - Can trivially `Kokkos::deep_copy()` between the two
  - *Technical aside:* if the original `View` is already on host:
    - The mirror will point to the original `View`
    - Deep copies will exit out

Cabana— A Kokkos-based library for particle-based codes

- The design mentality of writing Kokkos codes is very CUDA-like
  - Inconvenient for writing particle based codes in a language without meta-programming...
- Enter: Cabana! A Kokkos dependent library for particle-based simulation
- Cabana provides:
  - Data structures for handling particles
    - Cabana::Tuple
    - Cabana::AoSoA
    - Cabana::Slice
  - Algorithms for particle handling
    - Various sorting algorithms
    - Grid- and verlet- list construction
    - *MPCD relevant:* Binning particles to grids
  - Particle-based parallel dispatch paradigms
    - Cabana::simd_parallel_for
    - Cabana::neighbor_parallel_for
    - Cabana::neighbor_parallel_reduce

Array-of-Structs-of-Arrays (AoSoA) provide an efficient storage mechanism for particles

- There are different ways of handling lists of objects/ structures (*AoS* vs *SoA*)
- `Cabana` provides an easy to use modern hybrid method that combines two common approaches: the *Array-of-Structs-of-Arrays* (*AoSoA*)



- Good for optimising data locality and coalescing for particle-based applications

Concluding remarks

- Parallel software can enable opportunities for faster and more complex science
- `Kokkos` provides a good interface to simplify parallel development
- `Cabana` builds atop this to simplify particle-based simulation

Recommended further resources:

- General parallelism: "Parallel and High Performance Computing" by R. Robey and Y. Zamora, Manning, 2021
- Kokkos: Official Kokkos tutorials
- Cabana: ECP 2021 Cabana tutorial

novo nordisk
fonden

View these slides online

Thank you for listening!