LibGeoDecomp Developers’ Guide

Andreas Schäfer

Friedrich-Alexander-Universität Erlangen-Nürnberg

Seminar, 2012.01.11
1. Introduction

2. API

3. Architecture

4. Domain Decomposition

5. Source Code Layout
Applications

- stencil codes dominate scientific simulations
- optimized code:
  - highly complex
  - 10x faster than naïve (on single node)
- Library for Geometric Decomposition codes
- handles parallelization (and IO, loadbalancing...)
- portable (e.g. accelerators)
- modular design
- faster than naïve code
- slower than optimized, custom code?
High Level Architecture

- basic design pattern
- **Model**: Cell
  - simulation model, user supplied
- **Parallelization**: Simulator
  - handles cell updates and workflow
- **Balancer**: LoadBalancer
  - drives parameters of Simulator to equalize load
Main Utility Classes

- **Grid**: 2D or 3D array of Cells
  - a.k.a. simulation space
- **Initializer**: sets up grids, implemented by user
- **Writer**: regular output of grid
**Example: Jacobi Solver**

class Cell {
public:
    typedef Topologies::Cube<3>::Topology Topology;

    static inline unsigned nanoSteps() { return 1; }

    inline explicit Cell(const double& v=0) : temp(v) {}

    template<typename COORD_MAP>
    void update(const COORD_MAP& neighborhood,
                const unsigned& nanoStep) {
        temp = (neighborhood[Coord<3>(0, 0, -1)].temp +
                neighborhood[Coord<3>(0, -1, 0)].temp +
                neighborhood[Coord<3>(-1, 0, 0)].temp +
                neighborhood[Coord<3>(1, 0, 0)].temp +
                neighborhood[Coord<3>(0, 1, 0)].temp +
                neighborhood[Coord<3>(0, 0, 1)].temp) * (1.0/6.0);
    }

double temp;
};
```cpp
void runSimulation() {
    int outputFrequency = 10;
    StripingSimulator<Cell> sim(
        new CellInitializer(MPILayer().size()),
        MPILayer().rank() ? new TracingBalancer(new NoOpBalancer()),
        1000,
        MPI::DOUBLE);
    if (MPILayer().rank() == 0)
        new TracingWriter<Cell>(&sim, outputFrequency);

    sim.run();
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    runSimulation();
    MPI_Finalize();
    return 0;
}
```
class CellInitializer : public SimpleInitializer<Cell> {
public:
    CellInitializer(int num) : SimpleInitializer<Cell>(Coord<3>(64, 64, 64 * num), 100) {}

virtual void grid(GridBase<Cell, 3> *ret) {
    CoordBox<3> box = ret->boundingBox();
    Coord<3> offset =
        CoordDiagonal<3>()(this->gridDimensions().x() * 5 / 128);
    int size = this->gridDimensions().x() * 50 / 128;

    for (int z = 0; z < size; ++z) {
        for (int y = 0; y < size; ++y) {
            for (int x = 0; x < size; ++x) {
                Coord<3> c = offset + Coord<3>(x, y, z);
                if (box.inBounds(c))
                    ret->at(c) = Cell(0.99999999999);
            }
        }
    }
};
Outlook

1. Introduction
2. API
3. Architecture
4. Domain Decomposition
5. Source Code Layout
API: Cell Interface

- **update()**: performs actual cell update
  - old state given via parameters
  - cells need to copy unchanged values from old self
  - Neighborhood interfaces is **sloooood**
  - new interfaces in the work
    - abstracted away by update functors
  - how to handle static data with MPI or accelerators?

- **nanoSteps()**: number of sweeps per physical timestep
  - e.g. LBM: 1. flow, 2. collision

- **Topology**: topology of the simulation space
  - 2D vs. 3D
  - constant boundary (cube) vs. periodic (torus)
Outlook

1. Introduction
2. API
3. Architecture
4. Domain Decomposition
5. Source Code Layout
Simulators

- **common pseudo-code:**
  ```
  init_grid_decomposition()
  initializer.initGrid()
  for t in timesteps do
    for n in nanosteps do
      update_all_cells()
      sync_ghostzones()
      balance_load()
    end
  end
  notify_writers()
  end
  ```

- **SerialSimulator:** simple, but serial code
- **StripingSimulator:** parallelized via MPI, scales badly
- **HiParSimulator:** new flagship parallelization
  - rewritten, not yet finished
  - hierarchical design
  - state-of-the-art feature set
HiParSimulator

features:
- modular/flexible design
- overlapping communication and calculation
- load balancing
- memory conservation

partitions split grid coords into sub-domains according to weight vector
controlled by PartitionManager
Steppers store grid fragments, handle actual cell updates
1. update ghost
2. send ghost zones async.
3. recv ghost zones async.
4. update kernel
5. wait for ghost zones

PatchAccepter/PatchProvider
- serialize/buffer ghost zones
- move them between Steppers and MPILayer
Steppers

- VanillaStepper: Andi
- OpenMPStepper: Siegfried
- CudaStepper: Andi
- MulticoreStepper: Arne
- FPGAStepper: Franz
- CellStepper: Max/Franz
Simulator provides weight vector and load vector
- each node corresponds to an element in each vector
- the faster the node, the larger should its region be
- region size corresponds to weight
- LoadBalancer tries to equalize loads by tuning weights
- Simulator adapts decomposition according to new weights
Utility Classes

Storage
- Grid: n-dimensional array of cells
- DisplacedGrid: Grid with origin other than 0
- SuperVector: std::vector plus convenience methods
- SuperMap: same for std::map
- SuperSet: same for std::set

Geometry
- Coord: n-dimensional coordinate
- CoordBox: describes cuboid of coordinates
- Region: set of coordinates
- Topologies: handling of toroidal and cuboid topologies
integer coordinates
1-3 dimensions
vector arithmetic
example code

Coord<3> a(1, 2, 3);
Coord<3> b = 5 * a;
Coord<3> c = b - a;
c == Coord<3>(4, 8, 12);
**CoordBox**

- has origin and dimensions
- primary use: iteration
- currently: ugly Java’esque `hasNext()`, `next()` iterator

```cpp
CoordBox<3> box(Coord<3>(3, 4, 5), Coord<3>(2, 3, 3));
box.size() == 18;
CoordBoxSequence<3> seq = box.sequence();
while (seq.hasNext())
    std::cout << seq.next() << " \n";
```

- output

```
(3, 4, 5)
(4, 4, 5)
(3, 5, 5)
(4, 5, 5)
(3, 6, 5)
(4, 6, 5)
(3, 4, 6)
(4, 4, 6)
...```

Region

- stores `Coords`, not cells
- run-length compression
- `Streaks` store horizontal row of coordinates
example code

Region<3> r;
    r << Coord<3>(1, 1, 2);
    r << Streak<3>(Coord<3>(1, 2, 2), 5);
    r << Coord<3>(2, 1, 2);
    for (Region<3>::iterator i = r.begin(); i != r.end(); ++i)
        std::cout << *i << "\n";
    for (StreakIterator<DIM> i = r.beginStreak();
        i != r.endStreak(); ++i)
        std::cout << *i << "\n";

output

(1, 1, 2)
(2, 1, 2)
(1, 2, 2)
(2, 2, 2)
(3, 2, 2)
(4, 2, 2)
(origin: (1, 1, 2), endX: 3)
(origin: (1, 2, 2), endX: 5)
PartitionManager

- handles domain decomposition within HiParSimulator
- decomposition technique defined via template parameter
- main methods:
  - `getRegion()`: area of node $n$
  - `ownRegion()`: area of current node, expanded by $width$
  - `rim()`: inner and outer ghostzones
  - `innerSet()`: own region minus rim
PartitionManager (cont.)

```
0
\|--1
|   \|--2
|     \|--3
|         \|--4
\|--ownRegion(0)
```
PartitionManager (cont.)

ownRegion(1)
PartitionManager (cont.)

getRegion(4, 0)
PartitionManager (cont.)

ownRegion(0)
PartitionManager (cont.)

\[
\text{ownRegion}(3)
\]
PartitionManager (cont.)

rim(0)
PartitionManager (cont.)

rim(2)
PartitionManager (cont.)

innerSet(0)
PartitionManager (cont.)

innerSet(1)
PartitionManager (cont.)

innerSet(2)
PartitionManager (cont.)

innerSet(3)

2

innerSet(3)
Domain Decomposition

- 2D and 3D codes
- space filling curves
- graph partitioners
- wide ghost zones

→ geometry handling is hard
Ghost Zones

example:

- $8 \times 8$ grid
- shared among 5 nodes
Ghost Zones - How to Find Them?

![Diagram of ghost zones]

LibGeoDecomp Developers’ Guide
Andreas Schäfer
Ghost Zones - How to Find Them?
Ghost Zones - How to Find Them?
Solution: Set-like operations for Region

- supports Boolean operators and
- expansion by $n$ cells
- finding ghost zones:

```c
// (initialized according to image)
Region<2> r[5];
Region<2> ghosts[5];
for (int i = 0; i < 5; ++i)
    ghosts[5] = r[1].expand(1) & r[i];
```
Coding Style

- 1TBS (One True Brace Style) and the KDE Libs style
  - class TheThing;
  - void fooBar()
  - int myVar
  - indentation: 4 spaces, no tabs

- comments:
  - Doxygen API doc
  - self-documenting code
    - use descriptive names
    - avoid deep nesting by method extraction
  - focus on WHY, not what or how
Building

- building, testing:
  1. `cd libgeodecomp/`
  2. `./configure`
  3. `rake test`

- configure can activate/deactivate certain components
- `rake test` will only run tests belonging to the current dir

- unit test:
  - ...show how a given class is meant to be used
  - ...help us ensure the code still works
  - test coverage is good, let’s keep it this way
Directories

- `src/examples`: simple demo apps
- `src/io`: generic output modules and initializers
- `src/loadbalancer`: take a guess...
- `src/misc`: grids, coords and friends
- `src/mpilayer`: MPI communication and typemap creation
- `src/parallelization`: simulators
- `src/parallelization/hiparsimulator`: utility classes for the HiParSimulator
- `src/parallelization/hiparsimulator/partitions`: domain decomposition (e.g. SFCs, striping)
- `src/foo/test/bar`: unit tests for directory foo
  - bar is either `unit` for normal tests or `mpi_n` for tests which need n MPI procs
That's It!

Any Questions?
Send me an e-mail!