1. What is LibGeoDecomp?

LibGeoDecomp (Library for Geometric Decomposition codes) is a generic library for parallelizing stencil codes. Stencil codes are time and space discrete simulations on regular grids. The library itself is written in C++ and based on class templates. Users supply their simulation model as a class which models the behavior of the simulation cells. The library takes care of the parallelization, output, checkpointing etc. Its main features are:

- parallelization via MPI
- accelerator offloading (CUDA)
- 2D and 3D stencils
- latency hiding via
- wide ghostzones
- overlapping communication and calculation
- dynamic load balancing
- parallel IO, customizable formats
- parallel visualization via VisIt
- multiple domain decomposition techniques (e.g. SFCs or weighted recursive bisection)

2. Use Case: Simulation of Dendritic Growth

A first real world application using LibGeoDecomp is the simulation of dendritic growth in molten metal alloys. The crystalline structure of metal components is crucial to their durability. Modelling the crystallization process yields insights in how to improve manufacturing processes, e.g. turbine blades for jet engines.

The simulation model is developed by Markus Pitzing and Klemens Reuther at the Department of Metallic Materials, Friedrich-Schiller-Universität Jena.

This snapshot shows a dendrite simulation on a grid of 420x420x420 cells. This is, to the best of our knowledge, one order of magnitude larger than any previously published dendrite simulation.

3. Architecture

A distinct trend of the last years is that current super computers exhibit at least two levels of parallelism: system wide via MPI and on the node level via multi core CPUs or accelerators (GP GPUs, Cell, FPGAs). Our LibGeoDecomp tackles this challenge via a hierarchical parallelization. It can use different plug-ins on each level in order to adapt to the current system's architecture.

4. Example Code: Jacobi Solver

```cpp
#include "libgeodecomp.h" using namespace LibGeoDecomp;

class Cell {
public:	#endif
    typedef Topologies::Torus<3>::Topology Topology;
	n inline explicit Cell(const double & v=0) : temp(v) {}

t public:

    ~Cell() {}

t protected:

    Cell& operator=(const Cell&);

t private:

    int rank = 0;
    int largest = 0;
    int candidate = 0;

    double temp;

    void update(double & temp)
    {
        neighborhoo[d][Coord<3>(0,0,1)].temp =
            (neighborhoo[d][Coord<3>(0,0,1)].temp +
                neighborhoo[d][Coord<3>(0,1,0)].temp +
                neighborhoo[d][Coord<3>(1,0,0)].temp +
                neighborhoo[d][Coord<3>(0,0,0)].temp) / 7.0;

        return temp;
    }

t int main(int argc, char * argv[])
    {
        MPI::Initialize(argc, argv);
        MPI::Barri

        return 0;
    }
```

The Jacobi iteration is a good benchmark since it allows comparisons with competing frameworks. It is particularly challenging because of its low FLOPs/byte ratio. This example uses a simple interface to access neighboring cells. A new prototypic interface directly uses pointers and allows the user to write vectorized code.

5. Performance

3D Jacobi evaluation on a GPU (Nvidia Tesla C2050 with Fermi chipset) and an MPI cluster (IBM BladeCenter, 26 blades, each with 2 AMD Opteron 2216HE dual cores @2.4 GHz, 10 GB/s Infiniband). Performance on the GPU is outstanding (fastest code as far as we know), efficiency on the blade center is low because the CPU code (unlike the GPU code) is not yet optimized for memory bound codes. The efficiency increases if the model performs more FLOPS per byte.