DIY2: Data Parallel Out-of-Core Library

Dmitriy Morozov
dmorozov@lbl.gov

Tom Peterka
tpeterka@mcs.anl.gov

Motivation

Abstracts matter: think blocks, not processes
• block is the unit of decomposition; flexible size, shape, and placement;
• block-level addressing: user should worry about algorithmic logic, not implementation details;
• decompose problem into blocks, both local and global operations at block level.

Simple things should be simple: examples include
• exchanging particles efficiently using swap-reduce;
• partitioning the data using a kd-tree;
• sorting the data.

Performance portability to emerging architectures:
• Intel Knights Landing (manycore) processor will be native on Cori;
• MPI-threading will be essential;
• threading should be effortless in the data parallel setting.

Out-of-core processing:
• a lot of analysis is memory-bound, but simulations often need all the available memory (problem for in situ analysis);
• great deal of similarity between parallel and IO-efficient algorithms (both value locality and seek to minimize data movement);
• next generation supercomputers, e.g., Cori at NERSC, will have burst buffers (already a testbed on Edison).

Features

Decompositions and neighborhoods:
• regular decomposition
• kd-tree decomposition

(Performance) portability:
• to use multiple threads per process;
• to vary the amount of memory usage

Separate domain decomposition and block-to-process assignment:

diy::ContiguousAssigner assigner(world.size(), nblocks);
diy::RegularDecomposer<Bounds>::BoolVector diy::RegularDecomposer<Bounds>::BoolVector
diy::RegularDecomposer<Bounds>::BoolVector

diy::decompose(fs, rank, domain, assigner, create, share_face, wrap, ghosts);

Global communication (+ partners):
• k-ary swap-reduce()

• k-ary merge-reduce()

Results

Serialization mechanism central to DIY2. Used both for
• enqueue/dequeue communication mechanism;
• serializing individual blocks to move in and out of core.

How serialization works:
• by default, compiles the contents of the object;
• Specialize diy::Serialization<T> to create custom serialization for a class;
• Serializations provided for many STL containers, e.g., std::vector, std::map, std::set, etc.

Structure

struct Point ( float x, y, z ;)

struct Tet ( int verts[4]; int tets[4]; )

struct Tessellation {

  void save(BinaryBuffer& bb, const Tessellation& t) {
      save(bb, t.verts); save(bb, t.tets);
  }

  void load(BinaryBuffer& bb, Tessellation& t) {
      load(bb, t.verts); load(bb, t.tets);
  }

}

Serialization

• no need to deal with MPI datatypes
• no built-in object tracking (making what used to be a data race)
• but can easily implement it manually (e.g., to serialize a graph)

Density of 512 particles-ordered into a 1283 grid with different threading options

8 processors – complete OpenMP threading;

128 processors – serial OpenMP threading.

Out of core

Compilation of a signed distance function to material scanned
at ALS, mcs.anl.gov; 2560 processors, 4k blocks, using DIY2 on a single node, in two rounds.

Density of 512 particles-ordered into a 1283 grid with different threading options

8 processors – complete OpenMP threading;

128 processors – serial OpenMP threading.

Legend records the number of integers per processor.

Compilation of the distance function of 512 particles, 4k
blocks in 4096 grids, using 12 processors. It takes 1.8 seconds to complete the computation using 4096 processors.

3.1 Giga point/sec.