DIY2: data parallel out-of-core library

Dmitriy Morozov
dmorozov@lbl.gov

Tom Peterka
tpeterka@mcs.anl.gov

BERKELEY LAB

Argonne NATIONAL LABORATORY

Motivation

Abstractions 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).

Master

`Master` is the core of the library. It owns blocks, moves them and the queues in and out of core, calls back the user's computation routines.

```
std::string      prefix = "./DIY.XXXXXX";
diy::FileStorage storage(prefix);
diy::Master      master(world,            // MPI communicator
                        &create_block,
                        &destroy_block,
                        mem_blocks,       // blocks to keep in memory
                        num_threads,      // number of threads to use
                        &storage,         // external storage
                        &save_block,      // block serialization
                        &load_block);
...

master.foreach(&delaunay);
master.exchange();

void delaunay(void* b_, const diy::Master::ProxyWithLink& cp, void*)
{
  for (size_t i = 0; i < in.size(); i++)
  {
    vector<Point> pts;
    cp.dequeue(in[i], pts);
    // insert points into the tessellation
  }

  // do work

  for (size_t j = 0; j < out.size(); j++)
    cp.enqueue(out[j], outgoing_points[j]);
}
```

`in` and `out` come from the proxy `cp`

Serialization

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, copies 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.

```
struct Point { float x,y,z; };
struct Tet   { int verts[4]; int tets[4]; };

struct Tessellation
{
    std::vector<Point>  points;
    std::vector<Tet>    tets;
};

template<>
struct Serialization<Tessellation>
{
    void save(BinaryBuffer& bb, const Tessellation& t)
    { save(bb, t.points); save(bb, t.tets); }

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

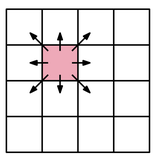
} serialized automatically (binary copy)

- no need to deal with MPI datatypes
- no built-in object tracking (making serialization fast), but can easily implement it manually (e.g., to serialize a graph)

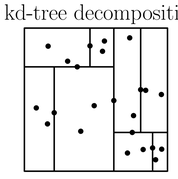
Features

Decompositions and neighborhoods:

- regular decomposition

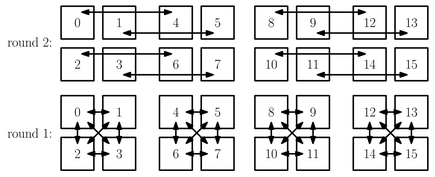


- kd-tree decomposition

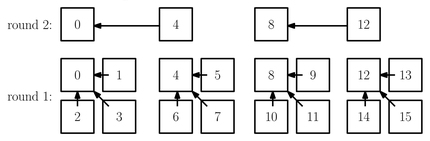


Global communication (+ partners):

- k-ary swap-`reduce()` (can vary k between rounds)



- k-ary merge-`reduce()` (can vary k between rounds)



(Performance) portability:

- to use multiple **threads** per process;
- to vary the amount of **memory usage**.

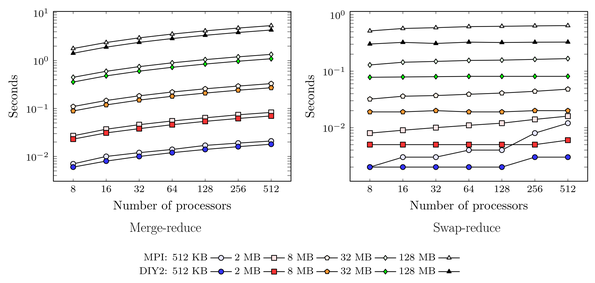
} change a **runtime parameter**

Separate domain decomposition and block-to-process assignment:

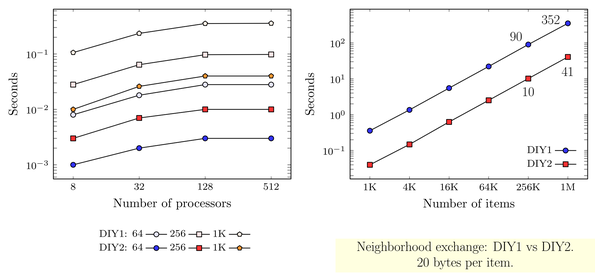
```
diy::ContiguousAssigner   assigner(world.size(), nblocks);

diy::RegularDecomposer<Bounds>::BoolVector          share_face;
diy::RegularDecomposer<Bounds>::BoolVector          wrap(3, true);
diy::RegularDecomposer<Bounds>::CoordinateVector    ghosts(3, 1);
diy::decompose(3, rank, domain, assigner, create,
               share_face, wrap, ghosts);
```

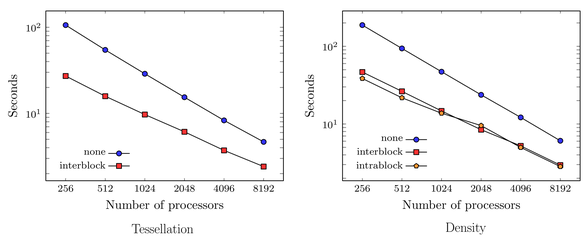
Results



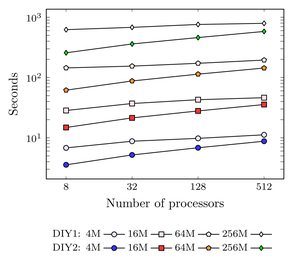
`MPI_Reduce` and `MPI_Reduce_scatter` vs DIY2 merge- and swap-reductions, respectively.



Neighborhood exchange: DIY1 vs DIY2.
20 bytes per item.

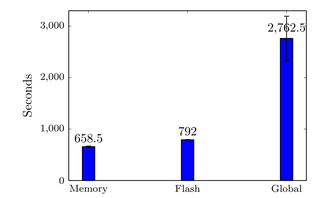


Density of 512^3 particles estimated onto a 1024^3 grid with different threading options.
interblock – completely automated threading by DIY2.
intrablock – manual OpenMP threading.

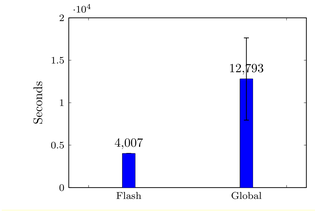


Sorting using DIY1 vs DIY2.
Legend records the number of integers per processor.

Out of core



Computation of a signed distance function to a material, scanned at ALS, on a $2560^2 \times 2160$ grid, broken up into 1024 blocks, using 128 processors and different methods for block storage (all blocks in memory; all but one block per process stored on NVRAM (burst buffers) or on global scratch). After computing the local distances within the block, the sources at boundaries are exchanged and distances are updated repeatedly, until no changes occur. The similarity in the running times between memory and flash is thanks to an optimization: blocks with no incoming queues cannot update and so are skipped.



Computation of the Delaunay tessellation of 1024^3 points, divided into 4096 blocks, using 32 processors. It takes 8.3 seconds to compute the tessellation using 4096 processors.
$(8.3 \cdot 4096/32 = 1062)$