Implementing a parallel Benchmark in Julia

Two versions of PageRank

Outline

- Background
- The PageRank benchmark
- Two Julia implementations
- Results

Background

What are we benchmarking?

- Hardware differences?
- Ability to exploit special hardware features, e.g. fast interconnect?
- Performance of programming language?
- Performance of libraries?
- Efficiency of algorithm?
- Quality of benchmark implementation?

Classes of benchmarks

- Goal oriented
- Algorithm oriented
- Code oriented

The PageRank benchmark

What is PageRank?

- A model for importance of web pages
- 1996: Google founders Larry Page and Sergey Brin for ranking web pages
- 1970s: *Very* similar work in bibliometics
- Mathematical structure: Stationary distribution of a Markov chain
- Algorithmic structure:
  - Power iteration on the eigenvalue problem (notice λ = 1 known)

$$(\alpha P + (1-\alpha)ve^T)x = x$$

  - Richardson method on linear system

$$(I - \alpha P)x = (1-\alpha)v$$

- See Gleich (2015)

Why a PageRank Benchmark

- Interest in large data problems (in contrast to physics simulations)
- Mathematically simple
- Exercises a range of (parallel) code pattern
  - Graph generation (local)
  - Sorting (global)
  - Filtering (global)
  - Linear algebra (global)
  - I/O (local)

The PageRank Benchmark Pipeline

- Kernel 0: *Graph generation* (+ write to disk)
- Kernel 1: (Read from disk +) *Sort* (+ write to disk)
- Kernel 2: (Generate adjacency matrix +) *Filter*
- Kernel 3: *PageRank*

Kronecker Graph generation I

- Graphs represented by their adjacency matrices
- Random matrix *X* is a *2x2* with a *single* unit element (multinomial)

$$P(X_{ij} = 1) = p_{ij} \quad \text{for } i, j = 1, 2$$

- Each edge is

$$E = X^{(1)} \otimes \cdots \otimes X^{(n)} = \bigotimes_{i=1}^{n} X^{(i)}$$

- Final graph is

$$G = \sum_{j=1}^{m} E^{(j)}$$

Kronecker Graph generation II

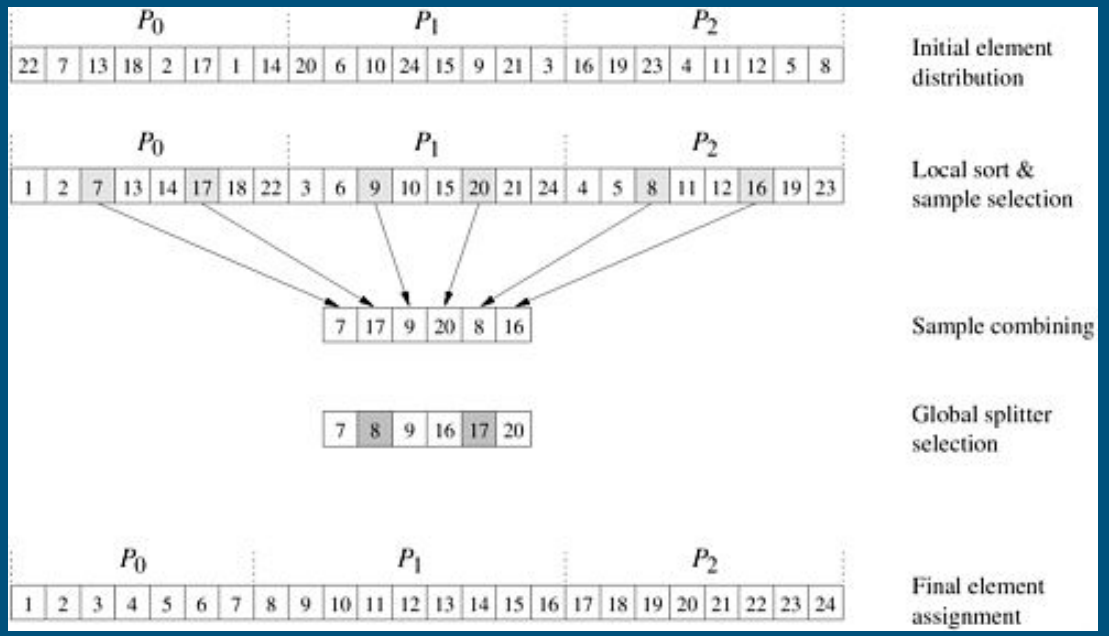
- Critical to exploit structure
- Data structure should only store list of edges
- E.g. $((a_1, b_1), \ldots, (a_n, b_n))$ $(a_1, \ldots, a_n), (b_1, \ldots, b_n)$

Or

- Computations should only track indices of non-zero elements

Sample sort

- Sort edge list by start vertex
- Dominated by all-to-all step



http://parallelcomp.uw.hu/ch09lev1sec5.html

Filtering

- Construct sparse matrix representation of graph (e.g. DCSC)
- Remove edges starting at vertex with highest in-degree

$$\left\{(i,j)\middle|\sum_i G_{ij} = \max_j \sum_i G_{ij}\right\}$$

- Remove edges starting at vertices with unit in-degree

$$\left\{(i,j)\middle|\sum_i G_{ij} = 1\right\}$$

- Scale rows

$$G_{ij} := \frac{G_{ij}}{\sum_j G_{ij}}$$

PageRank

Run 20 iterations of

$$x^{(k+1)} = \alpha P x^{(k)} + (1 - \alpha) v$$

where *v* is constant

Two Julia implementations

DArrays + remote_call

- Client-Server model
- Distributed/Global Arrays/Partioned Global Address space (PGAS)
- Mimics semantics of `Array`
- Internally composed by `remote_call`s
  - Similar to Remote procedure call (RPC) / Remote method invocation (RMI)
- Very flexible -> almost all of Julia's datatypes
- Potential overhead

MPI.jl

- SPMD model
- Julia package wrapper C implementations of MPI
- MPI is a message passing model but also
  - Optimized parallel constructs (all-to-all, gather, scatter, broadcast)
  - Optimized transport layers (shared memory, InfinyBand)
- No global abstraction layer for arrays (e.g. global array/PGAS)
- Fast for bitstypes

Kronecker graph generation in Julia (common)

```
function Base.rand(x::Kronecker)
    n = 2^x.scl
    m = x.edgesPerVertex*n

    rand2x2(_) = rand(x.onenz)
    randnxn(_) = mapreduce(rand2x2, ⊗, 1:x.scl)
    mapreduce(randnxn, add!, 1:m)
end

rand(Kronecker(scl, edgesPerVertex))
```

Kronecker graph generation in C++

```
for (T i = 0; i < M; i++) {
    T ij_i = 0, ij_j = 0;
    for (int ib = 0; ib < SCALE; ib++) {
        T r1 = random();
        T r2 = random();
        T ii_bit = r1 > ab_scaled;
        T jj_bit = r2 > (c_norm_scaled * ii_bit + a_norm_scaled * !ii_bit);
        ij_i += ii_bit << ib;
        ij_j += jj_bit << ib;
    }
    edges.push_back(std::tuple<T,T>(ij_i, ij_j));
}
```

Kernel 0

- DArray + remote_call

```
map(distribute(files)) do fn
    nEdges = EdgesPerWorker + (myid() == lastWorker ? surplus : 0)
    kronGraph500(fn, scl, nEdges)
end
```

- MPI.jl

```
id       = MPI.Comm_rank(state)
nworkers = MPI.Comm_size(state)
nEdges   = EdgesPerWorker + (id + 1 == nworkers ? surplus : 0)
kronGraph500(files[id + 1], scl, nworkers == id + 1 nEdges)
```