Large Survey Database

A Distributed Framework for Storage and Analysis of Large Datasets

Mario Juric[1,2]

[1]Harvard-Smithsonian Center for Astrophysics / Harvard University, [2]Hubble Fellow

Summary

The Large Survey Database (LSD) is a Python framework and DBMS for distributed storage, cross-matching and querying of large survey catalogs (>10^9 rows, >1 TB). The primary driver behind its development is the analysis of Pan-STARRS PS1 data. It is optimized for fast queries and parallel sweeps of positionally and temporally indexed datasets. It transparently scales to more than >10^2 nodes, and can be made to function in "shared nothing" architectures.

An LSD database consists of a set of vertically and horizontally partitioned tables, physically stored as compressed HDF5 files. Vertically, we partition the tables into sets of related columns ('*column groups*'), grouping together logically related data (e.g., astrometry, photometry). Horizontally, the tables are partitioned into partially overlapping ``cells'' by position in space (lon, lat) and time (t). This organization allows for fast lookups based on spatial and temporal coordinates, as well as data and task distribution. The design was inspired by the success of Google BigTable (Chang et al., 2006).

Our programming model is a pipelined extension of MapReduce (Dean and Ghemawat, 2004). An SQL-like query language is used to access data. For complex tasks, MapReduce ``kernels'' that operate on query results on a per-cell basis can be written, with the framework taking care of their distribution, scheduling, and execution. The combination leverages the users' familiarity with SQL, while offering a fully distributed computing environment.

LSD adds little overhead compared to direct Python file I/O. In tests, we sweeped through 1.1 Grows of PanSTARRS+SDSS data (220GB) less than 15 minutes on an 8-core machine. In a cluster environment, we achieve bandwidths of 14Gbits/sec (I/O limited). Based on current experience, we believe LSD should scale to be useful for analysis and storage of LSST-scale datasets.

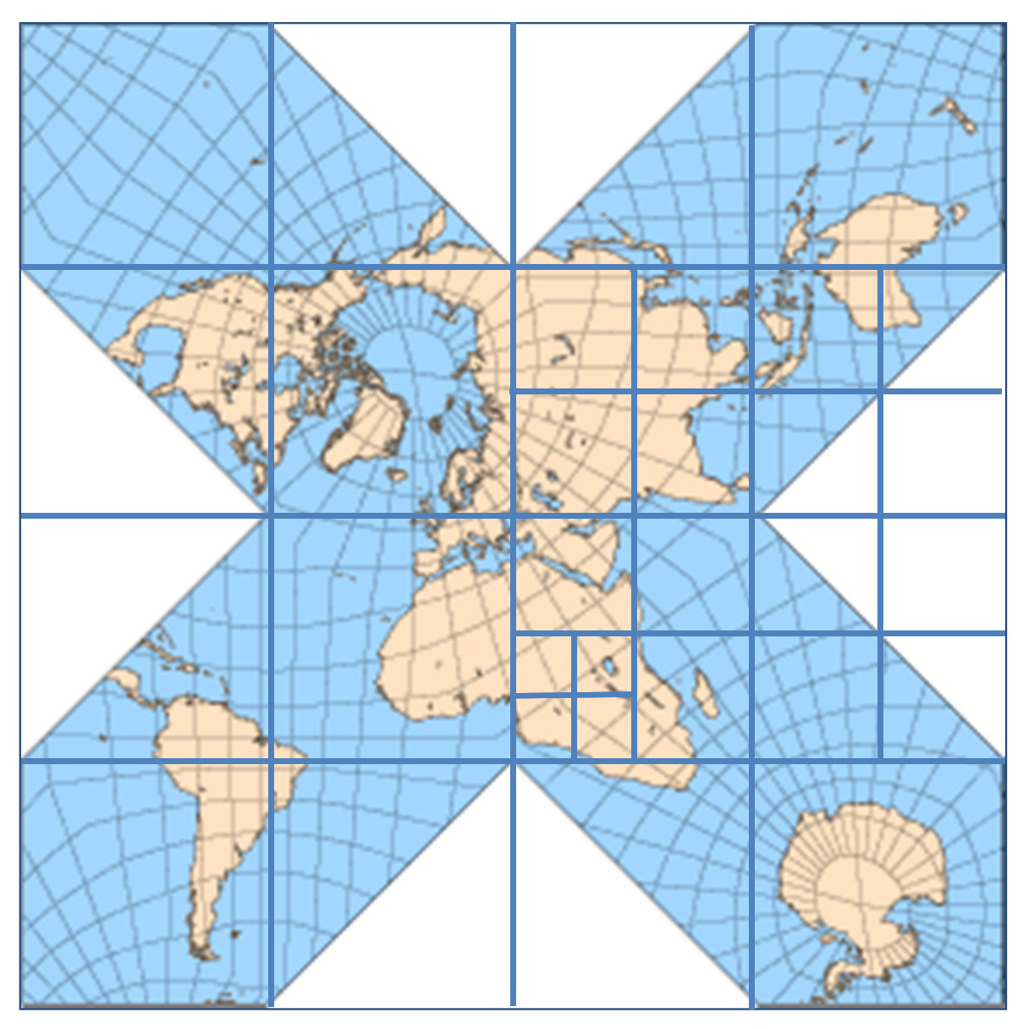
Develpment versions of LSD can be downloaded from **http://mwscience.net/lsd**.

LSD: A Spatially and Temporally Partitioned Database

LSD is optimized for fast queries and efficient parallel iteration through positionally (lon, lat) and temporally (time) indexed sets of rows. Its design and some of the terminology have been inspired by Google's BigTable distributed database and the MapReduce programming model. LSD is a system now capable of efficiently sweeping through PanSTARRS PS1 3π catalogs (~10^{10} rows), that should scale to LSST-sized datasets (>10^{12} rows) and be possible to distribute over large (100-1000) clusters of machines.

LSD tables are internally split vertically into column groups (*cgroups*): groups of columns with related data (e.g., astrometry, photometry, survey metadata, etc.). They're further partitioned horizontally into equal-area space and time cells (HEALPix pixels on the sky and equal time interval). The horizontal partitioning maps to a directory structure on the disk: every space/time cell maps to a unique directory. The data in each cell are stored in *tablets:* compressed, checksummed, HDF5 tables in the cell's directory, one per column group, and accessed using PyTables. LSD design allows this directory structure to be distributed onto a cluster of computers with no common storage, where each node stores and operates on a subset of cells. While not currently implemented, this capability is planned in the near future.

Each tablet also contains a copy of all rows that are within a margin (typically, 30 arcsec) outside the cell's boundaries (the *"neighbor cache"*). This allows for efficient neighbor lookup (or, for example, for the application of spatial matched filters) without the need to access tablets in neighboring cells. To facilitate parallelization and distribution, all LSD operations are always internally performed on cell-by-cell basis, with no inter-cell communication.



An illustration of "Butterfly HEALPix" projection, and the hierarchical spatial partitioning of database tables into cells

LSD Query Syntax

1.) SQL-like, case-insensitive keywords

2.) Column specifications are Python expressions; free to call Python functions from within query clauses

```
SeLEct
    ra, dec, g, r, sdss.g as sg, sdss.r as sr,
    sg-sr as sgr, ffitskw(chip_hdr(chip_id), "ZPT_OBS") as zpt
FROM
    ps1_obj, ps1_det, ps1_exp, sdss(outer)
WHERE
    sr < 22.5
INTO
    mytable
```

4.) The WHERE clause is a Python expression, with column data given in NumPy arrays.

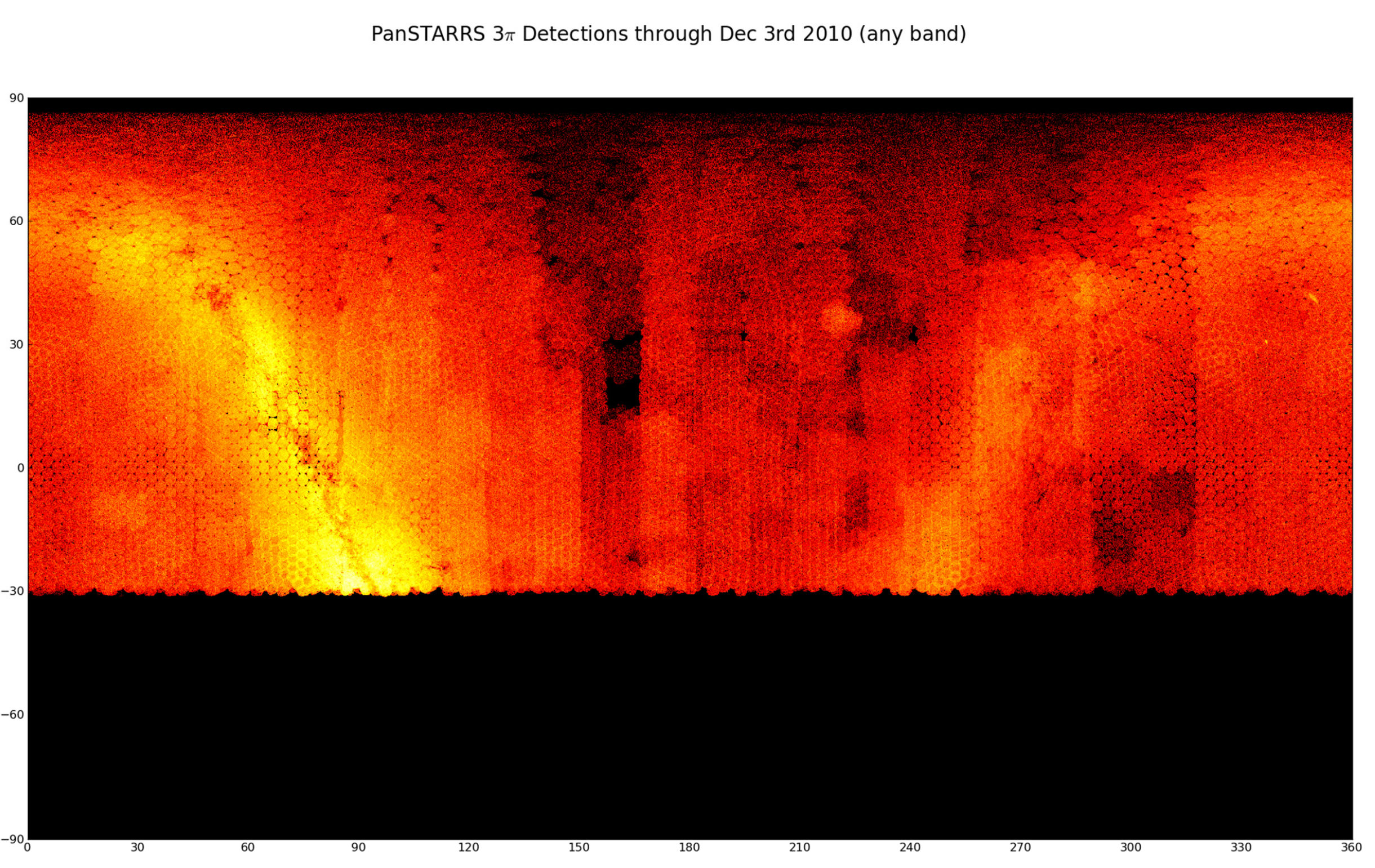
3.) Implicit natural JOINs between tables, outer JOINs supported

Distributed Computing on LSD: LSD/MapReduce

The computation over results of LSD queries may be distributed over hundreds of computational nodes by writing computational ***kernels*** conforming to the MapReduce programming model.

A MapReduce code to visualize survey footprint, utilizing a single mapper, is shown below.

```
import lsd
import pyfits
import numpy as np

def coverage_mapper(qresult, dx, filter):
        for rows in qresult:
                lon, lat = rows.as_columns()

                # Pixel indices
                i = (lon / dx).astype(int)
                j = ((90 - lat) / dx).astype(int)

                # Cut out the patch with data
                (imin, imax, jmin, jmax) = (i.min(), i.max(), j.min(), j.max())
                w = imax - imin + 1
                h = jmax - jmin + 1
                i -= imin; j -= jmin

                # Binning
                sky = np.zeros(w*h)
                idx = np.bincount(j + i*h)
                sky[0:len(idx)] = idx
                sky = sky.reshape((w, h))

                yield (sky, imin, jmin)

dx = 1./3600.
width  = int(np.ceil(360/dx))
height = int(np.ceil(180/dx)+1)

sky = np.zeros((width, height))

db = lsd.DB("db")
q = db.query("select ra, dec from ps1_det")
for (patch, imin, jmin) in q.execute([(_coverage_mapper, dx, filter)]):
        sky[imin:imin + patch.shape[0], jmin:jmin + patch.shape[1]] += patch

pyfits.writeto(output, sky[::-1,::-1].transpose(), clobber=True)
```



PanSTARRS PS1 source map constructed using the MapReduce code above

PyMR: A Pipelined Distributed MapReduce Engine for Python

PyMR is an implementation of the MapReduce programming model for Python programs running on Beowulf clusters. PyMR requires no special cluster-wide setup and can be launched via whatever scheduler is available (e.g. PBS, LSF, SGE, or others). This makes the deployment of PyMR possible for non-privileged users. PyMR transparently handles MapReduce task startup, distribution, communication, and the return of the results to the application.

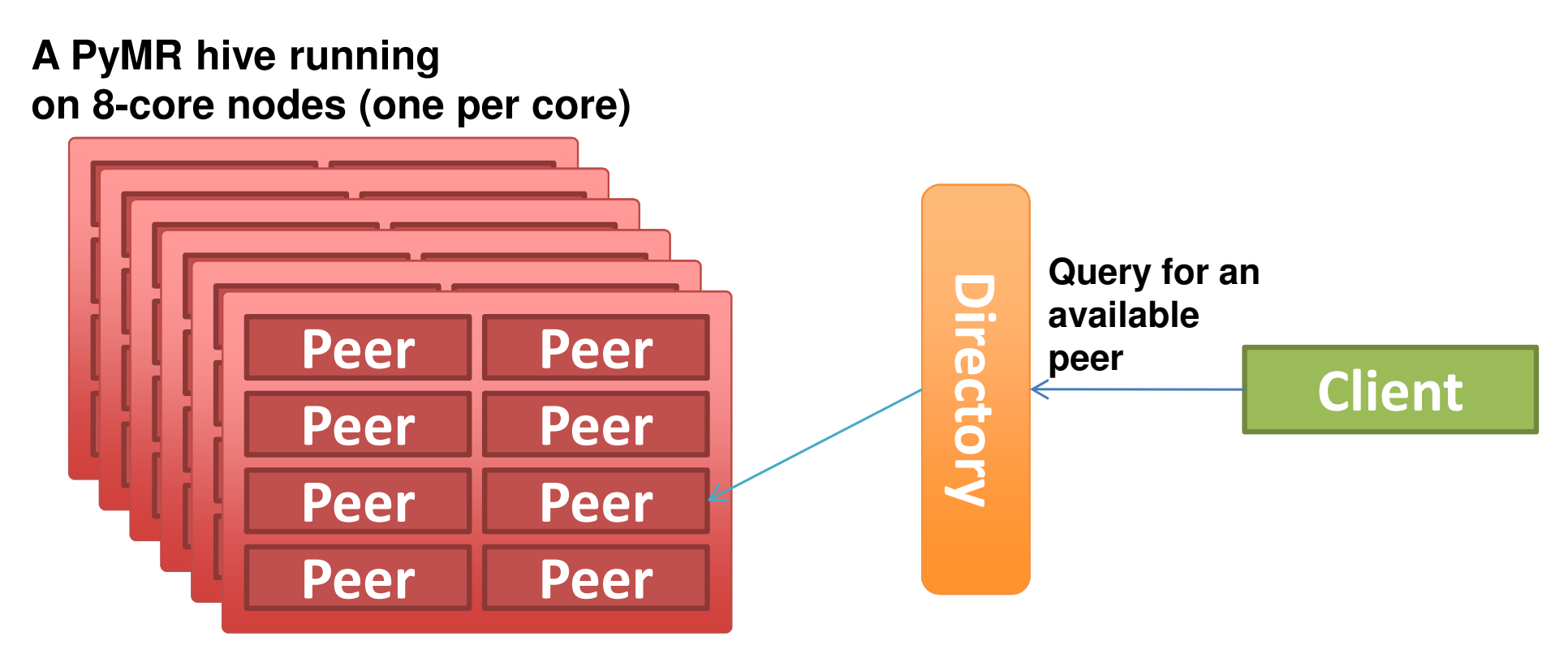
In contrast to Hadoop and similar solutions, the running of MapReduce jobs on PyMR is *pipelined*; each PyMR MapReduce stage transmits the results to the next one in "push" fashion, and subsequent stages start as soon as there's data available to be processed. This provides a substantial speed up for certain types of tasks.

Top right: A **PyMR *"hive"*** consists of tens to hundreds of ***peers***, running on nodes of the cluster, typically with one peer per core. Peers are contact points for PyMR clients, and launchers of PyMR workers (see below). All peers are registered in a well known ***directory***, and listen for task submission from the clients. The ***client*** consults the directory to find an available peer and submit a new map reduce task.

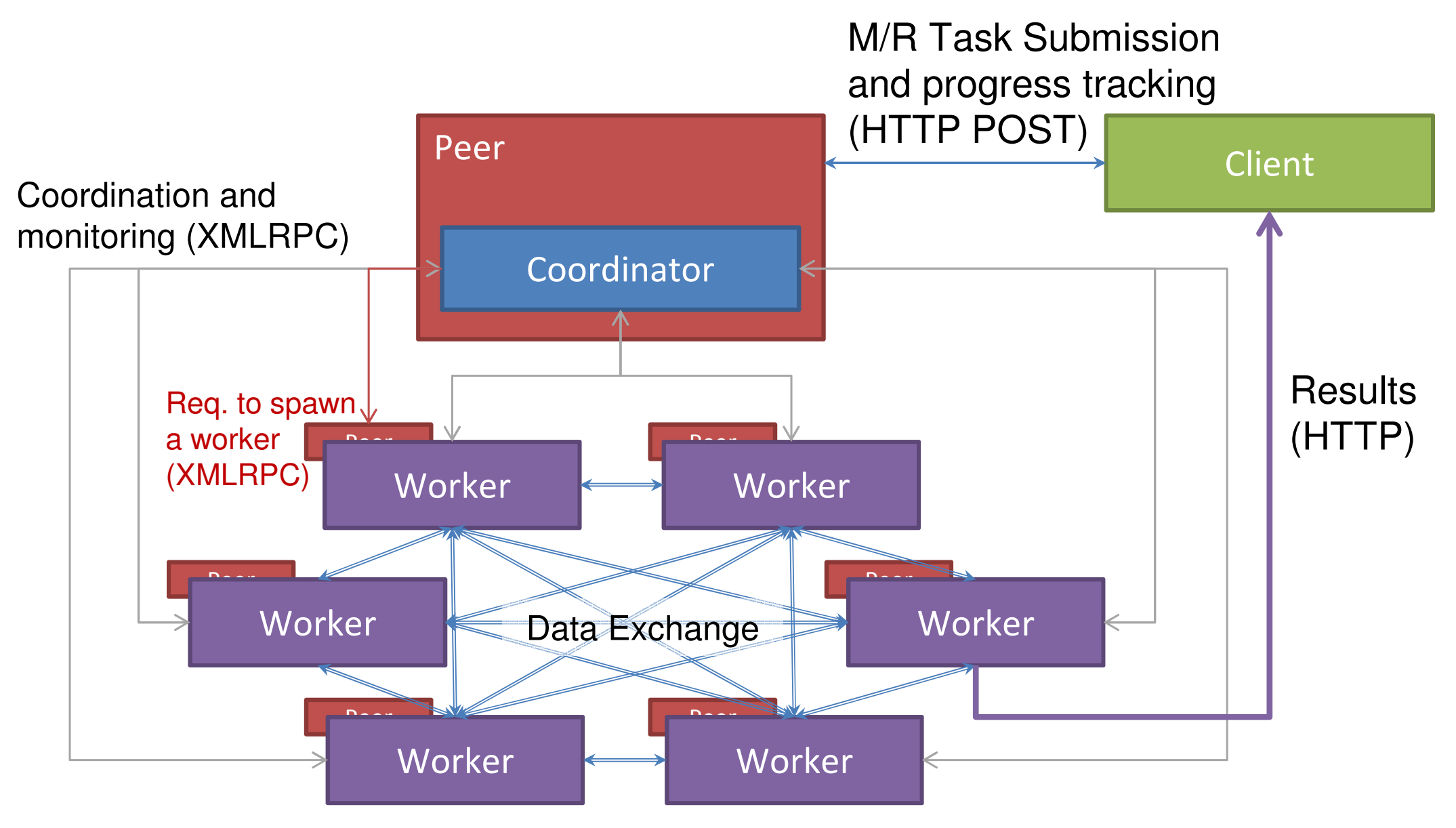
Middle right: A new task is submitted via a HTTP POST request from the client. The submission consists of the chain of MapReduce kernels to execute, and the Python source file containing them. The peer responds by spawning a ***coordinator*** thread, that manages the spawning and communication between ***workers*** – the processes actually executing the M/R task. The workers, form the ***"swarm"*** for the task, exchange control and progress messages with the coordinator (XMLRPC), but transmit data via long-lived direct worker-to-worker TCP connections. The final result is made available to the client via HTTP.

Bottom: Internally, each worker runs a ***gatherer*** thread, that receives and aggressively buffers the incoming data, one or more ***MapReduce*** threads, that perform the actual computation by running the user's kernels, a ***scatterer*** thread, distributing the emitted (key, value) pairs to the workers that will handle them, and a command & control ***monitor*** thread. The multi-threaded design ensures the I/O and computation run in parallel with minimum interference, maximizing the I/O throughput.

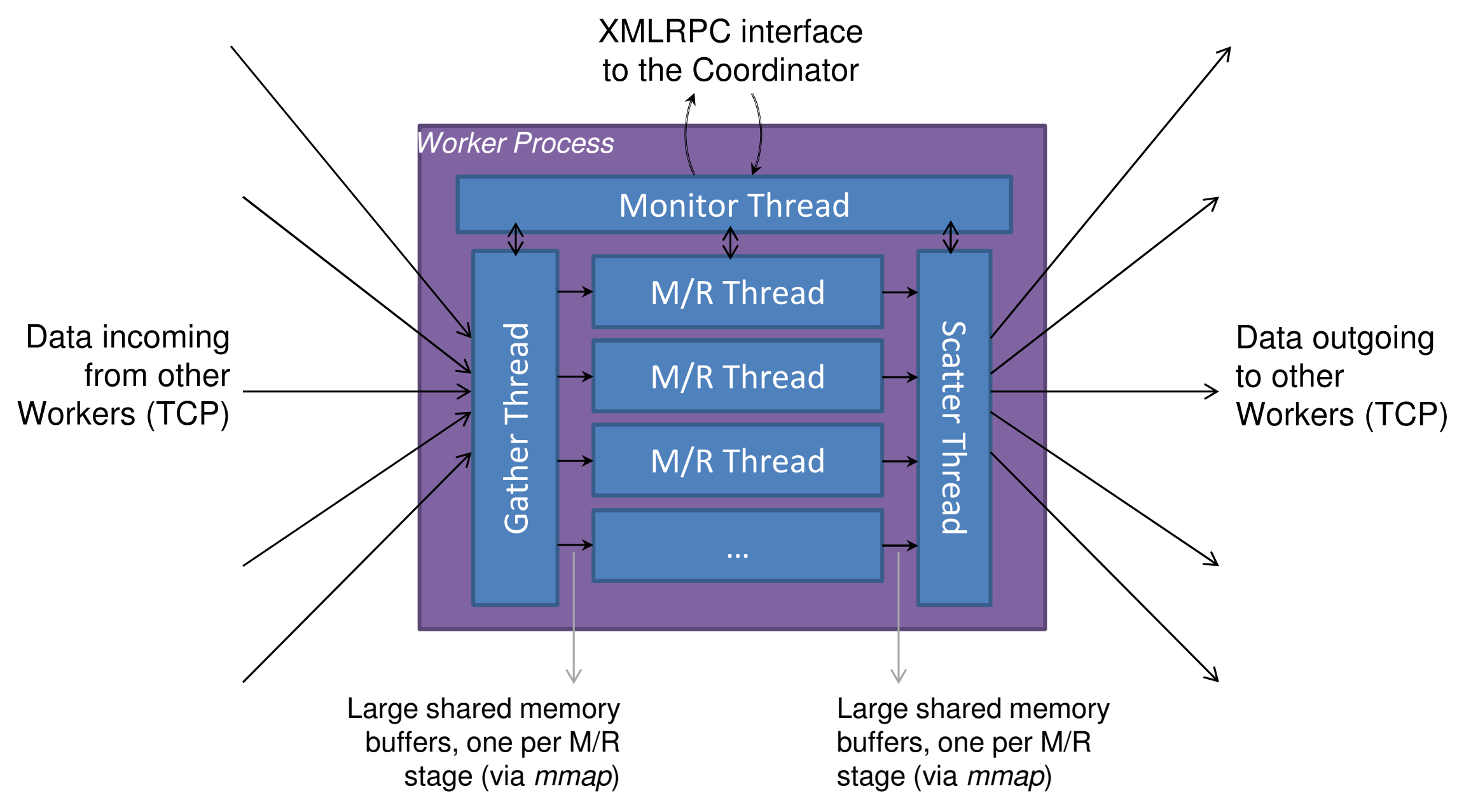
<u>All of this is fully hidden from the user, as seen from the source-code example (eg, see the listing to the left)</u>


A PyMR hive running on 8-core nodes (one per core)
Peer
Directory
Query for an available peer
Client

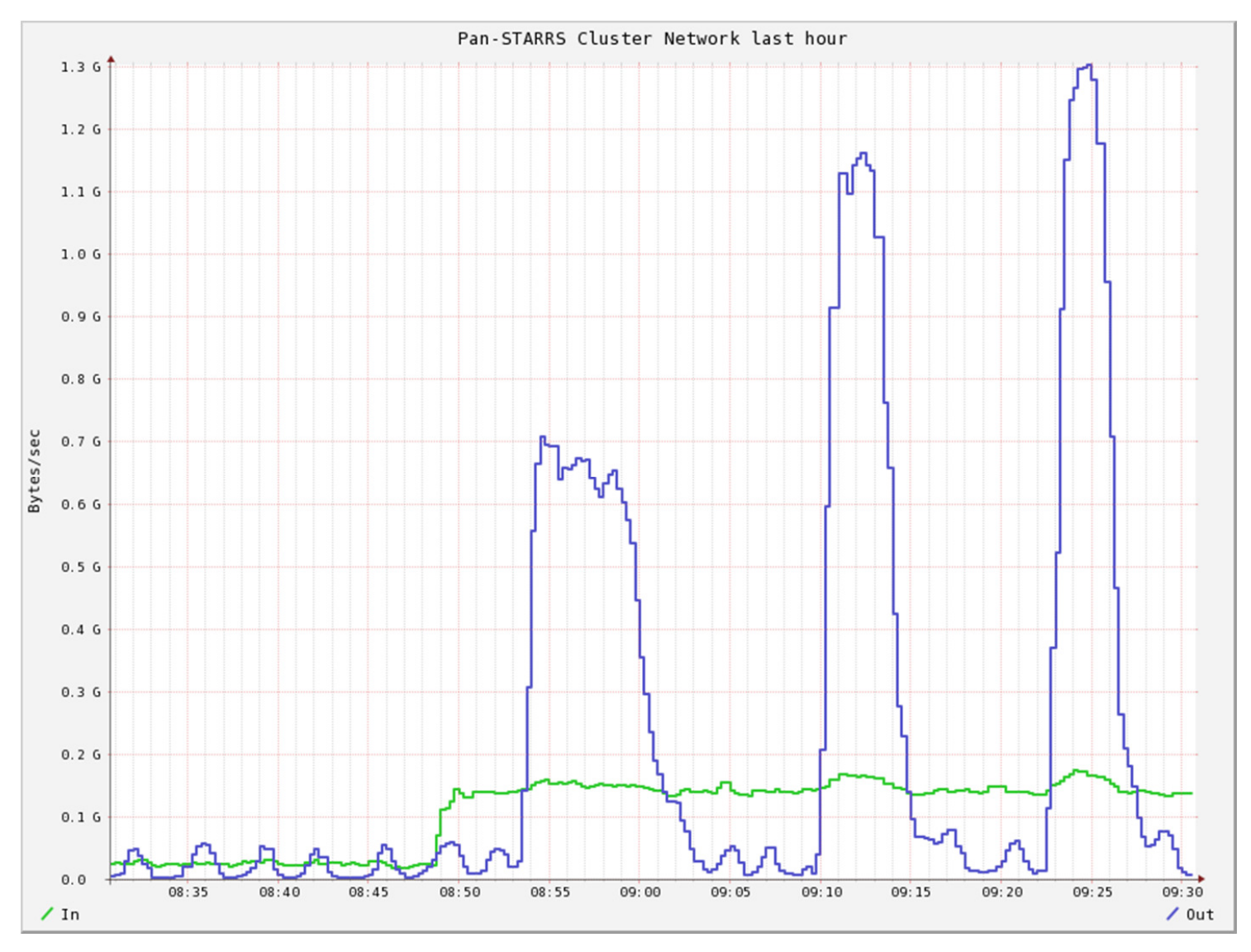
A diagram of a PyMR hive


M/R Task Submission and progress tracking (HTTP POST)
Peer
Client
Coordination and monitoring (XMLRPC)
Coordinator
Req. to spawn a worker (XMLRPC)
Worker
Data Exchange
Results (HTTP)

A diagram of a running PyMR swarm


XMLRPC interface to the Coordinator
Worker Process
Monitor Thread
Gather Thread
M/R Thread
M/R Thread
M/R Thread
...
Scatter Thread
Data incoming from other Workers (TCP)
Data outgoing to other Workers (TCP)
Large shared memory buffers, one per M/R stage (via mmap)
Large shared memory buffers, one per M/R stage (via mmap)

Internal structure of a PyMR worker process



The 96-worker run operated on 28 distinct nodes giving peak throughput of ~50 Mbytes/node.

The I/O throughput on Harvard FAS Odyssey computer cluster with a centralized LustreFS storage grid having a theoretical bandwidth of 20Gbps. The blue line represents reads, green line are the writes.

The lsd-footprint code (listed on the left) was run with 40, 80 and 96 workers for the first, second and third bump, respectively.