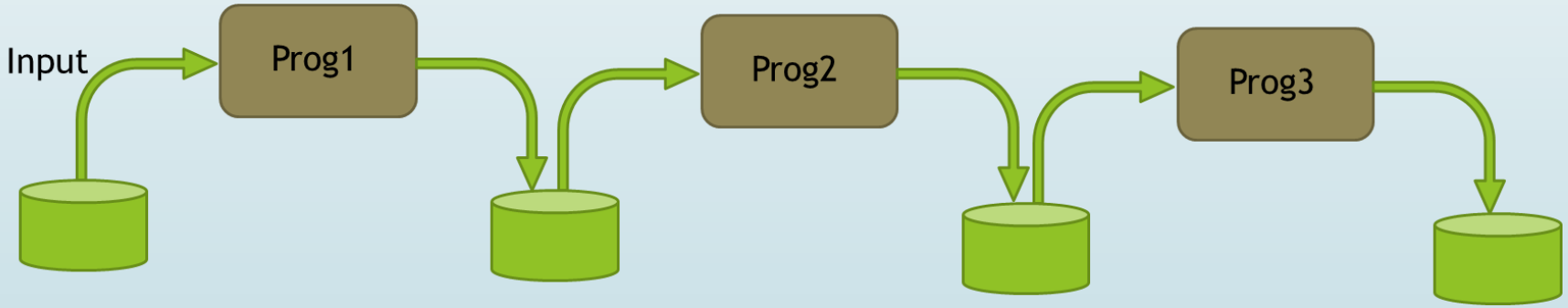
PXFS Project

Alireza Kheirkhahan

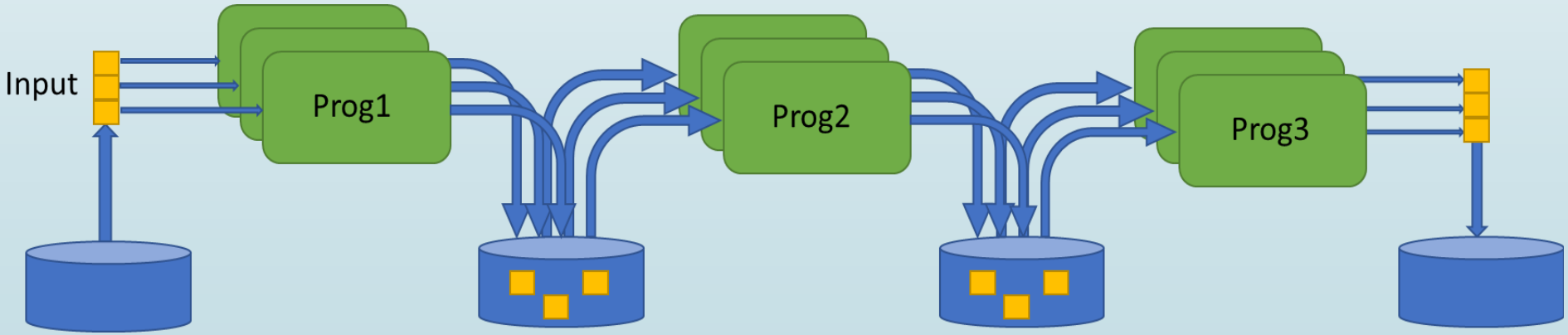
Multistage Applications

- Each stage executed independently following previous step
- The applications may be memory conscious
- Or written in different programming languages



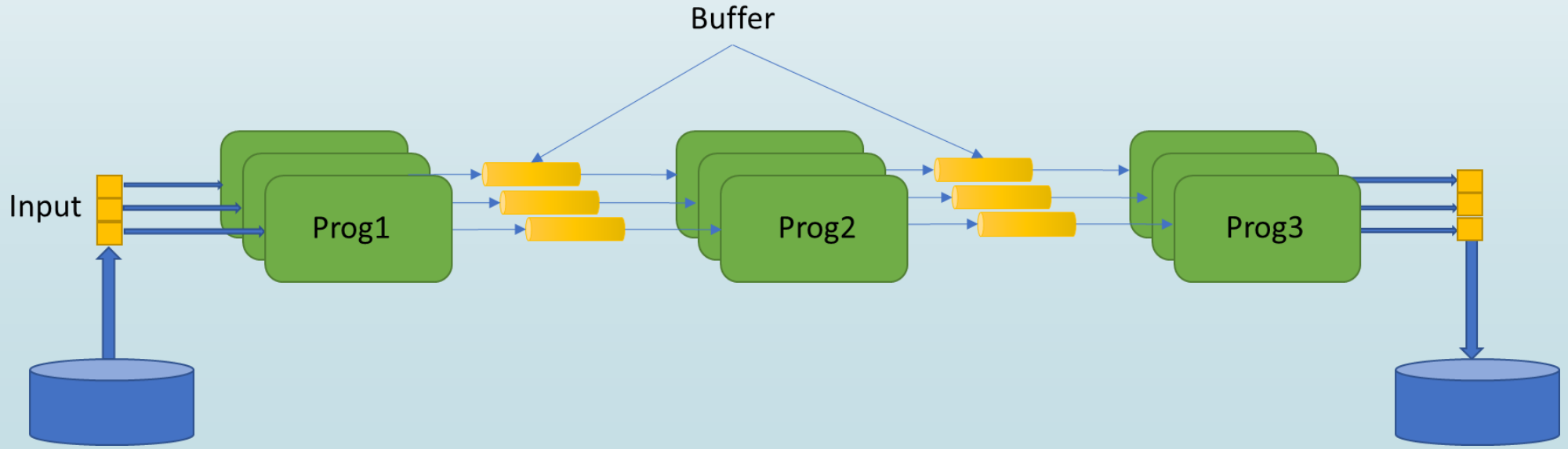
Divide and Conquer

- Given the problem is embarrassingly parallel you can divide input in smaller pieces
- Make it possible to run on a cluster
- Chunk size can be tuned based on input and computation capacity
- It is not enough for I/O intensive applications



Smarter I/O

- Avoid Storing intermediate data on disk
- Bring the application to data
- Try passing the data over the wire when necessary
- Use pre-fetching to speedup the I/O requests.



1- HPXIO

- An HPX component representing a file
- Can have local file, orangefs and cache file as backend
- Capable of performing simple I/O operations in sync and async mode
- It uses same interface independent of backend file system
- Comes with all benefit of HPX components
  - Gid
  - Accessible within all localities
  - Relocation (in some scenarios; When we are dealing with a stateless shared file system as backend)

HPXIO Example

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>

#include <hpxio/server/local_file.hpp>
#include <hpxio/base_file.hpp>

int main(int argc, char* argv[])
{
    hpx::io::base_file f =
        hpx::new_<hpx::io::server::local_file>(hpx::find_here());

    f.open(hpx::launch::sync, "test.txt",  O_WRONLY | O_APPEND | O_CREAT));

    hpx::serialization::serialize_buffer<char> data ("test\n", 5);
    f.write(hpx::launch::sync, data);

    f.close();

    return 0;
}
```

- An interface similar to POSIX Although sometimes we had to change the interface for performance

```
hpx::future<hpx::serialization::serialize_buffer<char> > read(size_t const count)
hpx::serialization::serialize_buffer<char> read(hpx::launch::sync_policy, size_t const count)
```

Implementation Problem

- We need to adopt the legacy application to use HPXIO
- Requires rewriting a big part of application dealing with I/O
- Headache
  - Source code not available
  - Programs are too complicated and rewriting takes a heroic effort
  - Programs written on incompatible languages

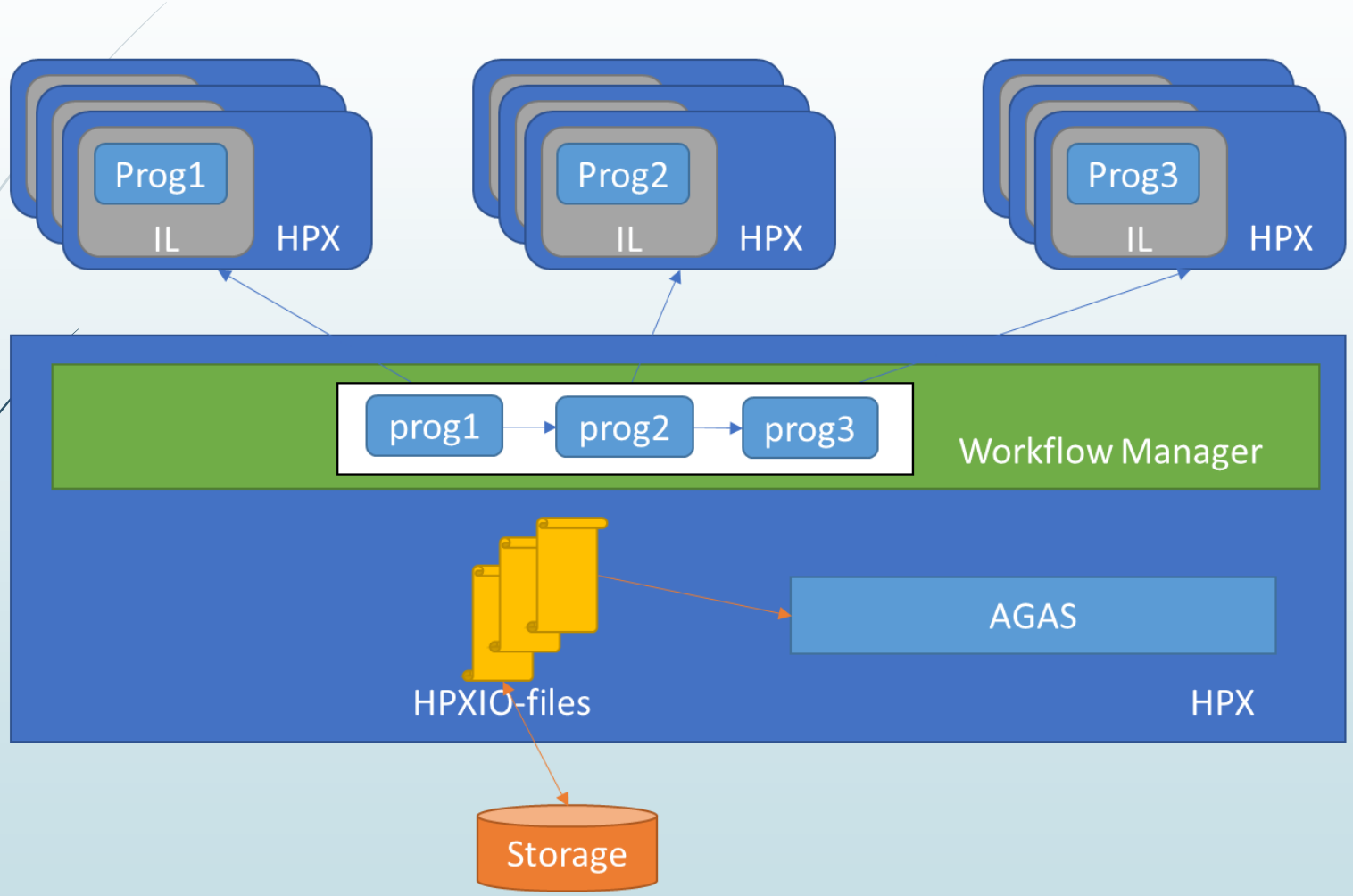
2: Interposition Library

- LD_PRELOADed with application
- When it's loaded read environment variables and connects to running HPX application
- Intercepts all I/O calls from application
- Checks each I/O call and let the irrelevant calls pass through to the OS
- For relevant I/O calls locate the HPXIO object and redirects all call to the object
- Will stay in memory as long as application is running
- The details are avoided

3: Workflow Manager

- Last piece of puzzle which connects all parts together
- Written with HPX, it facilitate HPX features to act as manager
- It's responsibilities includes:
  - Divide the input in chunks and launching application
  - Create and register HPXIO object and pre-fetch data if necessary
  - Setup the environment variables correctly to interposition library be loaded and be able to connect back to HPX.

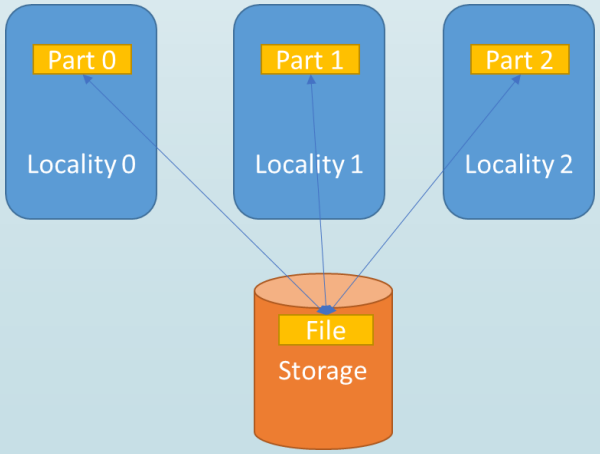
Final Destination



Future Work

Distributed File Objects

- The file image is represented on multiple localities
- Each locality access the portion locally
- Relatively easy to implement since we already have partitioned vector (files are nothing but a vector or char)
- Doesn't have much benefit unless added with Collective I/O



Collective I/O

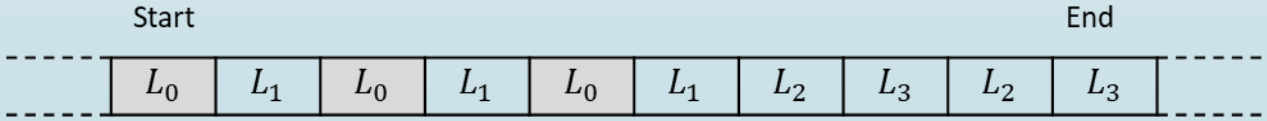
- Built on top of distributed file object
- Each locality has non-interleaving file portion
- Each locality take responsibility of it's portion and access directly to underlying file system
- For other portions, each locality communicate to the responsible locality rather than accessing the file system directly
- It a way of parallel file I/O

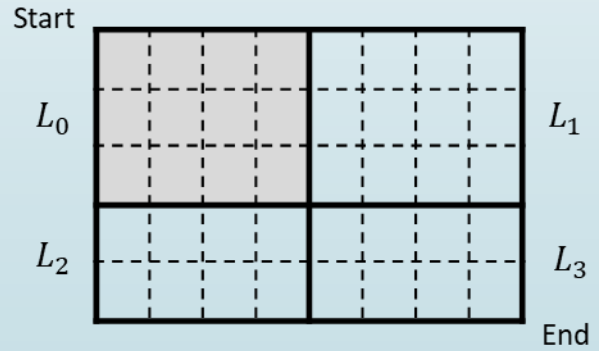
Collective I/O Characteristic

- Difficulties:
  - Some chunks could become hot spot and overwhelm the responsible localities
  - Since each locality is assigned with chunk at random or based on locality number and chunk number, they may be assigned a chunk that is only used by others. Localities may end up always doing remote data transfer for their I/O requests.
  - For sequential access, the localities would work in turns instead of in parallel.
  - Overkill for small files
- Benefits:
  - Since each locality deals with a non-interleaving le chunk, data integrity is preserved between localities. (example)

Collective I/O Example

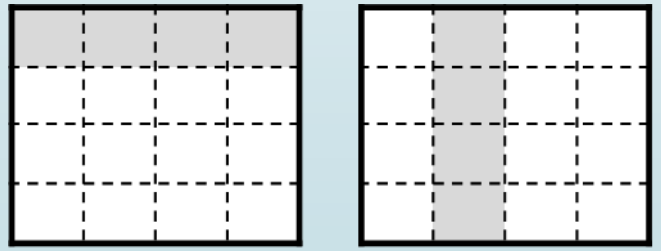
- Example:
  - This two-dimension matrix divided between four localities
  - Each locality wants to write the data back to file
  - The locality could write the file in large parts and risk the data integrity or use several small I/O calls and overwhelm the underlying file system.





File View

- Creates a layer of abstraction on file view
- Could benefit when integrated with collective I/O
- Can create performance problem in cache friendly application.
- Example:
  - Two matrices are stored canonically in two different file and our program want to perform matrix multiplication
  - The read operation on second file is scattered



Questions