Adding in memory MapReduce operations and IO interfaces to Chapel

Tim Zakian   Brad Chamberlain   Michael Ferguson

August 9, 2013

Abstract

As big data becomes more and more prevalent in day-to-day computing, there is a need to be able to deal with large sets of data. However conventional systems such as Hadoop and Hive rely heavily on a divide and conquer method distributed throughout a cluster of computers. This method has been proven to be extremely efficient at a number of tasks, however this method can lead to problems when dealing with data in a contextual, iterative or interactive environment. In this document we not only offer an implementation of a framework that allows in memory MapReduce operations in Chapel and shows the benfits that can be acheived this way, but also offers up an API so that it can integrate seamlessly with other filesystems.

1 Introduction

Chapel is an emerging parallel language being developed by Cray Inc. with the goal of improving programmer productivity on large-scale systems as well as on the desktop. It has been developed with the goal being to a large extent the standard HPCesque battery of programs which in the large majority of cases do not involve heavy string based processing, and up until recently this held true with the vast majority of HPC applications.

However, with the event of big data, and especially looking at iterative programs we see that in many cases these programs can benefit from an in memory MapReduce as demonstrated in [2], in which the files data is read in and worked on the node that hosts that data - retaining the data on that node until it is told that it is no longer needed - and doing MapReduce operations using these nodes.

We assert that Chapel could work quite well in this framework. In this document we explore how to go about adding MapReduce operations to Chapel as well as how this might benefit the user not only in terms of speed, but also in terms of user productivity.

2 API

There are four APIs exposed.

- A Systems Level API that is the glue code between the Chapel runtime and the filesystem. (`qio_plugin_hdfs.c/h`)
- A user-level IO API that interfaces with the system API, as well as provides the interface for the MapReduce API. (`HDFS.chpl`)
- A way to parse from the IO interface in Chapel to records in Chapel. (`RecordParser.chpl`)
- A MapReduce API that interfaces with the IO system and RecordParser in Chapel. (`HDFSiterator.chpl`)

2.1 Runtime API

Throughout this section, the names that we use in the description of the function pointer are the names of those function pointers in the `qio_file_functions_t` struct. The Runtime API currently consists of Several functions:

- `readv` Implements a function that has the same semantics as `readv` in POSIX.1-2008. It takes in a user defined filesystem struct which contains all the information that the file system needs in order to work. The `iovec` argument is already allocated.

```
typedef qioerr (*qio_readv_fptr)
                (void*, // plugin file pointer
                const struct iovec*,
                // Data to write into
                int,
                // number of elements in iovec
                ssize_t*);
                // Amount that was written into iovec
```

- `writev` Implements the same sematics as `writev` in POSIX.1-2008.

```
typedef qioerr (*qio_writev_fptr)
                (void*, // plugin fp
                const struct iovec*,
                // data to write from
                int,
                // Number of elements in iovec
                ssize_t*);
                // Amount written on return
```

- `preadv` Implements a function which does a positional read of the file and puts the results in the `iovec` argument which is already allocated.

```
typedef qioerr (*qio_preadv_fptr)
                (void*, // plugin fp
                const struct iovec*,
                // Data to write into
                int,
                // number of elements in iovec
                off_t,
                // Offset to read from
                ssize_t*);
                // Amount that was written into iovec
```

- pwritev does a positional write of the file that is referenced by the plugin filepointer from the contents stored in the argument to iovec. It returns the number of bytes written on return.

```
typedef qioerr (*qio_pwritev_fptr)
                 (void*,//plugin fp
                 const struct iovec*,
                 // data to write from
                 int,
                 // Number of elements in iovec
                 off_t,
                 // offset to write
                 ssize_t*);
                 // Amount written on return
```

- seek Implements the semantics of lseek in POSIX.1-2008.

```
typedef qioerr (*qio_seek_fptr)
                 (void*,  // plugin fp
                  off_t,  // offset to seek from
                  int,    // Amount to seek
                  off_t*);// Offset on return from seek
```

- filelength Returns the length of the file in bytes that is referenced by the plugin filepointer.

```
typedef qioerr (*qio_filelength_fptr)
                 (void*,     // file information
                  int64_t*); // length on return
```

- getpath Returns the path to the file that is referenced by the plugin filepointer.

```
typedef qioerr (*qio_getpath_fptr)
                 (void*, // file information
                  const char**);// string/path on return
```

- open Opens the file on the configured filesystem pointed to by the path passed in as the second argument. The plugin filepointer (which is user defined) is finished being populated here. The user also at this point needs to also set the flags for the file (for more information see 2.5 and 2.6). open expects a configured filesystem passed in as the last argument to the function.

```c
typedef qioerr (*qio_open_fptr)
                (void**,  // the plugin fp on return
                 const char*, // pathname to file
                 int*, // flags out
                 mode_t,  // mode
                 qio_hint_t, // Hints for opening the file
                 void*);  // The configured filesystem
```

- `close` Closes the file pointed to by the plugin filepointer. Has the same semantics as `close` in POSIX.1-2008.

```c
typedef qioerr (*qio_close_fptr)
                (void*); // file fp
```

- `fsync` Provides the same functionality as `fsync` in POSIX.1-2008.

```c
typedef qioerr (*qio_fsync_fptr)
                (void*); // file information
```

- `getcwd` replicates the semantics of `getcwd` in POSIX.1-2008.

```c
typedef qioerr (*qio_getcwd_fptr)
                (void*, // file information
                 const char**); // path on return
```

- The struct `qio_file_functions_t` represents all the functions needed within QIO in order to implement the functionality needed for file IO in Chapel. This is loaded into the QIO representation of the file at initialization and the user is responsible for populating this struct before passing it into the QIO runtime code.

```c
typedef struct qio_file_functions_s {
  qio_writev_fptr  writev;
  qio_readv_fptr   readv;

  qio_pwritev_fptr pwritev;
  qio_preadv_fptr  preadv;

  qio_close_fptr   close;
  qio_open_fptr    open;

  qio_seek_fptr   seek;

  qio_filelength_fptr filelength;
  qio_getpath_fptr getpath;

  qio_fsync_fptr fsync;
  qio_getcwd_fptr getcwd;

  void* fs; // Holds the configured filesystem

} qio_file_functions_t;
```

Where the `void* fs` in `qio_file_functions_t` holds the configured file system. This way we can support calling functions that are not dependent upon opening the file on a file system (*e.g.* calling `getpath` or `getcwd`).

The various types of information needed by the file system in order to read and write files is passed around as a user defined struct in a `void*` (the plugin fp). This way we can support any filesystem since we no longer have to worry about the number of arguments to these functions. Therefore the library writer is responsible for packing the arguments into - and extracting the arguments from - the plugin fp.

The library writer is also responsible for writing the wrapper functions around the calls to the filesystem so that it conforms with this API. This way we can report appropriate errors as well as supporting as many filesystems as possible.

The only other function needed in order to create an interface with QIO is to implement a function that populates the `qio_file_functions_t` struct (the number of arguments to such a function can be arbitrary and user defined). Hereafter we will call this function `create_qio_functions`. This will then be an interface to the module level code. The way in which we pass this information through to the QIO runtime is via the function in runtime called `qio_file_open_access_usr` which is called by `open` in the Chapel module level code and is the last argument to the function (the rest of the arguments are the same as `qio_file_open_access`).

This now brings us to a discussion of the module level API to the runtime.

The module-level API for the runtime is almost trivial and depends only on `create_qio_functions`. The way the library writer would interface with the runtime at the module level would be along the lines of

```
extern proc create_qio_functions(...):
                 qio_file_functions_t;

proc myOpen(...): file {
  var err: syserr = ENOERR;
  ...
  var fsfns = create_qio_functions(...);
  ...
  err = qio_file_open_access(..., fsfns);
  ...
  }
```

2.2 User API for HDFS

2.2.1 Types

The types defined by the HDFS module are as follows:

```
record hdfsChapelFile {
  var files: [rcDomain] file;
}
```

Which is a wrapper around a replicated array of files; one per locale. (*i.e.* a "Global file" in a sense)

```
record hdfsChapelFileSystem {
  var home: locale;
  var _internal_file: [rcDomain] c_ptr;
  // contains hdfsFS
}
```

This is almost the same as `hdfsChapelFile`, except this time instead of replicating a file across each locale, it replicates the configured file system across each locale.

```
record hdfsChapelFile_local {
  var home: locale = here;
  var _internal_:qio_locale_map_ptr_t
      = QIO_LOCALE_MAP_PTR_T_NULL;
}
```

Represents a mapping of a localeId to a specific byte range in the file.

```
record hdfsChapelFileSystem_local {
  var home: locale;
  var _internal_: c_ptr;
}
```

Represents a configured file system pointer.

2.2.2 Functions

```
hdfsChapelConnect(name: string, port: int): fs;
```

Connects to HDFS with name `name` and port `port` and replicates across all locales on the machine. This way there is a valid way to reference this other then from the locale it was called on.

```
fs.hdfsChapelDisconnect();
```

Disconnects (on each locale) from the file system `fs` connected to by `hdfsChapelConnect`

```
fs.hdfsOpen(filename: string,
                flags: iomode): hdfsChapelFile;
```

Opens a file with path `pathname` and in mode `flags` on each locale from the file system that was connected to via `hdfsConnect`. The only possible iomodes are `iomode.r` and `iomode.cw` (due to HDFS constraints).

```
hdfsChapelFile.hdfsClose();
```

Closes the files created by `fs.hdfsOpen`.

```
hdfsChapelFile.getLocal: file;
```

Returns the file for the current locale that you are on when this function is called.

```
hdfs_chapel_connect(path:string, port: int):
                  hdfsChapelFileSystem_local;
```

Same as `hdfsChapelConnect` except that this only creates a valid file system on the locale it was on when it was called.

```
hdfsChapelFileSystem_local.hdfs_chapel_disconnect();
```

Disconnects from HDFS.

```
getHosts(f: file);
```

Returns a C array of structs of the form

```
{(locale_id, start_byte, length), ...}
```

which can be accessed via

```
getLocaleBytes(g: hdfsChapelFile_locale, i: int);
```

The only other things added to the current IO functionality is a convenience function

```
hdfsChapelFile.hdfsReader(...): channel;
```

Which takes in the same arguments as the standard `file.reader` function.

After this, all other functionality is supported. An example of this is:

```
use HDFS;

var gfl: hdfsChapelFile;
var hdfs: fs;

hdfs = hdfsChapelConnect("default", 0);
gfl  = hdfs.hdfsOpen("/tmp/advo.txt", iomode.r);

for loc in Locales {
  on loc {
    var r = gfl.hdfsReader(start=50);
    // same as:
    // var r = gfl.getLocal.reader(start=50);
    var str: string;
    r.readline(str);
    writeln("on␣locale␣", here.id, "␣string:␣" + str);
    r.close();
  }
}
on Locales[2] {
  gfl.hdfsClose();
}

on Locales[1] {
  hdfs.hdfsChapelDisconnect();
}

/* outputs:

   on locale 0 string: 0325

   on locale 1 string: 0325

   on locale 2 string: 0325

   on locale 3 string: 0325

 */
```

2.3 Record parser API

The API for the record parser works with the IO interface in Chapel and *is not* dependent upon using HDFS or anything else except for the functions provided in `IO.chpl` and regexp support. The API is as follows:

```
new recordReader(type recordType, reader: channel,
                 regex: string): recordReader;
```

Creates a record reader that parses into the record of type `recordType` from the channel `reader` using the regexp `regex`

```
new recordReader(type recordType,
                 reader: channel): recordReader;
```

The same as the first one, however this time the regex is inferred from the field names in the record `recordType`. The regex created this way is very lax in terms of how much whitespace there is between records. This could lead to naming problems as well as there might be problems parsing into the record.

```
recordReader.get(): recordType
```

Returns one record and advances the position in the file to the end of where it read. Will return error if it cannot return one.

```
recordReader.stream(): iter(recordType)
```

Returns a stream of records of type `recordType` until the regex no longer matches. At end, leaves the channel position at the place where it read to.

An example of how to use this is as follows:

```
use RecordParser;

var fl = open("test.txt", iomode.r);
var ch = fl.reader();

record Test {
  var name: string;
  var id:   int;
}

var regex = "Name:␣(.*)\\s*Id:␣(.*)\\n\\n";

var M = new RecordReader(Test, ch, regex);

writeln("get()␣=␣", M.get());

writeln("Now␣testing␣stream()");

for m in M.stream {
  writeln(m);
}

ch.close();
fl.close();

/*  Outputs:
    get() = (name = one, id = 1)
    Now testing stream
    (name = two, id = 2)
    (name = three, id = 3)

    For test.txt =
    Name: one
    Id: 1

    Name: two
    Id: 2

    Name: three
    Id: 3
*/
```

2.4 MapReduce API

The API here is the simplest of them all and consists of only one function

```
HDFSiter(path: string, type recordType,
         regex: string): iter(RecordType)
```

This is a leader-follower iterator that is locale aware in terms of data locality (*e.g.* if blocks 0 and 1 reside on locales 0 and 1 respectively, it will read on and work on locales 0 and 1 while using those blocks).

Many times what things might look like is:

```
use HDFSiterator;

record someRecord {
  var id1: real;
  var id2: real;
}

var regex = "ID1(.*)\\s*ID2(.*)\\n";

forall r in HDFSiter("/tmp/test.txt", someRecord, regex) {
  <do something with r in here>
}
```

2.5 Flags in QIO

The flags for QIO are fairly straightforward and consist of:

- `QIO_FDFLAG_READABLE` Specifies that this file has been opened in a mode that supports reading.
- `QIO_FDFLAG_WRITABLE` Specifies that this file has been opened in a mode that supports writing
- `QIO_FDFLAG_SEEKABLE` Specifies that this file is seekable.

2.6 Hints in QIO

There are 5 types of hints:

- `IOHINT_NONE` Normal operation. We expect to use this most of the time.
- `QIO_HINT_RANDOM` We expect random access to this file.
- `QIO_HINT_SEQUENTAL` We expect sequential access to this file.
- `QIO_HINT_CACHED` We expect the entire file to be cached and/or pulled in all at once.
- `QIO_HINT_PARALLEL` We expect many channels to work on this file concurrently.

3 Examples

The examples here show how one might go about interfacing with the various APIs provided in section 2 and while they are not supposed to be comprehensive, the hope is that this section will provide enough guidance to get started.

3.1 System API

In this section we'll walk through a simple example of implementing `preadv` for HDFS.

```
qioerr hdfs_preadv (void* file,
                    const struct iovec *vector,
                    int count, off_t offset,
                    ssize_t* num_read_out)
{
  ssize_t got;
  ssize_t got_total;
  qioerr err_out = 0;
  int i;

  STARTING_SLOW_SYSCALL;

  err_out = 0;
  got_total = 0;
  for(i = 0; i < count; i++) {
    got = hdfsPread(to_hdfs_file(file)->fs,
                    to_hdfs_file(file)->file,
                    offset + got_total,
                    (void*)vector[i].iov_base,
                    vector[i].iov_len);
    if( got != -1 ) {
      got_total += got;
    } else {
      err_out = qio_mkerror_errno();
      break;
    }
    if(got != (ssize_t)vector[i].iov_len ) {
```

```
28      break;
29     }
30    }
31
32    if( err_out == 0 &&
33     got_total == 0 &&
34     sys_iov_total_bytes(vector, count) != 0 )
35     err_out = qio_int_to_err(EEOF);
36
37    *num_read_out = got_total;
38
39    DONE_SLOW_SYSCALL;
40
41    return err_out;
42   }
```

In lines 1-5, we take in the file pointer for our filesystem as a `void*`, a vector of preallocated `iovec` buffers, the number of these vectors, the offset to read from and the output to tell the runtime how much we were able to read. `qioerr` is a QIO defined struct, and consists of an error number field and a message field (*i.e.* an `int` and a `const char*`) and is a `syserr` in Chapel code.

In lines 11-30 we call `STARTING_SLOW_SYSCALL` which at this time does nothing - and is meant to signify that we are doing a system call that might block - however in the future this might do something to migrate threads, and therefore errno's might go away if you tried accessing them after `DONE_SLOW_SYSCALL`. Inside the for loop we simply populate our vector of buffers until we cannot read anymore, or until we have filled up all the buffers.

In lines 32-42 we check to see if we have read anything, if we don't have an error and we didn't read anything and the vectors total length is not 0, then we have encountered an unexpected EOF, and set the error to `EEOF`. We then simply set the amount we've read and return.

3.2 Other APIs

In this example, we walk through the implementation of the HDFS leader-follower iterator which uses the User API along with the RecordParser API. The code for the leader is:

```
1
2   iter HDFSiter(param tag: iterKind,
3                 path: string, type rec, regex: string)
4    where tag == iterKind.leader {
5
6     type hdfsInfo = 2*int(64);
7
8     var workQueue: [LocaleSpace] domain(hdfsInfo);
9
10    // ----- Create a file per locale ----
11    var hdfs = hdfsChapelConnect("default", 0);
12    var fl   = hdfs.hdfsOpen(path, iomode.r);
13
14    // -------- Get locales for blocks ------
15    var fll = fl.getLocal();
16    var (hosts, t) = getHosts(fl1);
17
18    for j in 0..t-1 {
19      var h = getLocaleBytes(hosts, j);
20      writeln(h);
21      var r: hdfsInfo;
22      r(1) = h.start_byte;
23      r(2) = h.len;
24      workQueue[h.locale_id] += r;
25    }
26
27    coforall loc in Locales {
28      on loc {
29        forall block in workQueue[loc.id] {
30          var rr = fl.hdfsReader(start=block(1));
31          var N  = new RecordReader(rec, rr, regex);
32          for n in N.stream_num(block(1), block(2)) {
33            yield n;
34          }
35        }
36      }
37    }
38
39    fl.hdfsClose();
40    hdfs.hdfsChapelDisconnect();
41  }
```

In lines 7-11 We first create an associative domain over our locales with tuples of the form `(start_byte, length)` (in essence creating a work-queue for each locale), we then connect to HDFS and open files on each of our locales.

In lines 14-24 we then get our local file and get a C array of structs of the form `(locale_id, start_byte, length)` and the number of elements in that array are returned as the second element of the tuple. We then go through each element in the array and add it to our work-queue for that locale.

In lines 26-35, we go on each locale and in parallel run through the work queue for that locale, returning records in parallel.

4 Future Work

Future work in this area has a couple different areas.

4.1 Locality

Right now, it is up to the programmer to create the ability to query where data is stored as well how they wish to represent this data and make it viewable in Chapel. However in the future it would be nice to have a locality API much like the current file IO API and therefore make it easier for the library writer to interface with Chapel level programs in such a way that they can leverage as much benefit from a different filesystem as possible.

Also, it would be nice to have a way to deal only with data on a certain locale when dealing with regular filesystems (such as LUSTRE, or other RAID

0 type filesystems) the goal being so you could do something along the lines of

```
// Connect to, and open a filesystem here
...
forall stripe in file.stripes by 3 {
  // Do something with every third stripe
}
```

Which gives the ability to easily reason about and use concurrency in RAID 0 type filesystems.

4.2 Replication of Files

Right now due to the way HDFS is structured, when `hdfsOpen` is called, it creates a file per locale. This is not ideal, but at least at this point the only way to be able to handle all cases in HDFS since we do not know beforehand where the various blocks of the file will reside.

On other filesystems, this could be different, and therefore the possibility to minimize the number of files created becomes a possiblity. As well as this also might allow us to represent this information in a more compact and meaningful way.

Along with these options is the option to make this file creation and replication lazy in the sense that files will only be created on a locale iff a file is needed on that locale. Otherwise, that locale will not get a file. This coupled with caching the file once we've opened it on a locale would also offer a speed improvement.

5 Integrating with Other File Systems

To integrate with other filesystems, the user must modify a couple areas. For example if we wanted to add Ceph [1], we would need to edit and change the following:

- Create a plugin `qio_plugin_ceph.c` that is placed under `runtime/src/qio/foreignFS/ceph`
- Add the header file `qio_plugin_ceph.h` to `runtime/include/qio`
- Add a `Makefile.foreignFS-ceph` to `runtime/etc` that will link in what you need at the compile time of the program (this will normally include include and libraries).
- You will need to add and replicate (renaming appropriately) the various makefiles in `runtime/src/qio/foreignFS/hdfs`
- After this you should be able to make the `ceph` plugin by setting `CHPL_FOREIGN_FS=ceph` and remaking the runtime.

References

[1] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.