An Introduction to fileUtils

Feiyi Wang (Ph.D)

National Center for Computational Sciences
Oak Ridge National Laboratory

March 3, 2015

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Outline

Motivation

- Most traditional file system tools are serialized.
- Some are multi-threaded, bounded by single host performance.
- What we need: parallelization that can go beyond single host.

Existing tools:

- traditional `cp`, `find` ...
- multi-threaded: `bbcp`, `xdd`
- cross-cluster: `grid-ftp`

What is fileUtils?

One of a suite of parallel tools produced by collaboration between LLNL, LANL and ORNL.

Origin: *LaFon, Misra and Bringhurst: "On distributed File Tree Walk of Parallel File System".*

fileUtils suite

- `dcmp` - compare files
- `dcp` - copy files
- `dfind` - find files by path name
- `drm` - remove files
- `dtar` - create file tape archives

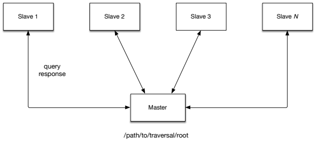
General Idea

From tools perspective, we need a parallel tree-walk algorithm. The essence of such algorithm is to **efficiently** visit each file in parallel. If such general problem can be resolved, then it can be applied to:

- file copy
- file delete (purge)
- file checksum (ls -l)
- file find
- ...

How to distribute the workload?

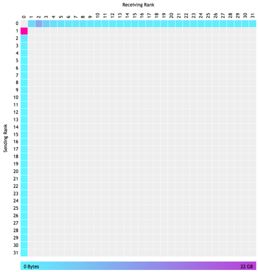
A simple but naive solution:



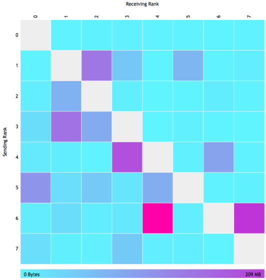
Problem:

- centralized
- unbalanced

Jharrod Lafon: centralized heat map



Jharrod Lafon: distributed heat map



Pattern: Work Stealing

Key Ideas

- Each worker maintains it own work queue
- When local work queue is processed, it picks a random worker, and asks for more work items.

Without a master process, how do we know when to terminate?

Distributed Termination Detection

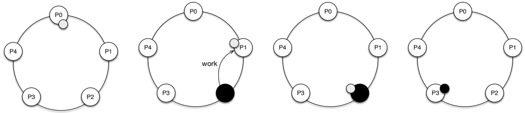
Edsger W. Dijkstra: *Derivation of a termination detection algorithm for distributed computations*. June 10, 1983.

1. The system in consideration is composed of N machines, $n_0, n_1, \ldots, n_{N-1}$, logically ordered and arranged as a ring. Each machine can be either white or black. All machines are initially colored as white.
2. A **token** is passed around the ring. machine n's next stop is $n + 1$. A token can be either white or black. Initially, machine n_0 is white and has the white token.
3. A machine only forwards the token when it is passive (no work)
4. Any time a machine sends work to a machine with lower rank, it colors itself as black.
5. Both initially at n_0, or upon receiving a token:
    1. if a machine is white, it sends the token unchanged.
    2. if a machine is black, it makes the token black, makes itself white, and forward the token.

Termination condition: white n_0 receives a white token.

Understanding the Algorithm

- Stable state is reached when all machines are passive.
- Edge case: a system is composed of one machine: it will send a white token to itself, thus it meets the termination condition, also it reaches the stable state.
- Even a machine becomes passive at time *t* and forward the token, it can become active again upon receiving works from others.
- When a black token returns to machine n_0 or a white token returns to a black machine n_0, a termination conition can not be met. The token forwarding continues.



Initial State | Send work to j < i turn black | Color token black if itself is black | P2 forwards the token, color itself white

`libcircle` API

dwalk

```
// Initialize state
CIRCLE_init(0, NULL, CIRCLE_SPLIT_EQUAL);

// Register callback
CIRCLE_cb_create(& walk_stat_create);
CIRCLE_cb_create(& walk_stat_process);
CIRCLE_cb_reduce_init(& reduce_init);
CIRCLE_cb_reduce_op(& reduce_exec);
CIRCLE_cb_reduce_fini(& reduce_init);

// After setting up, execute
CIRCLE_begin();

// Finally, clean up
CIRCLE_finalize();
```

dwalk Callback

```
void walk_stat_create(CIRCLE_handle * handle) {
    handle->enqueue(CURRENT_DIR);
}

void walk_stat_process(CIRCLE_handle * handle) {
    struct stat st;
    handle->dequeue(path);
    int status = lstat(path, &st)
    if (S_ISDIR(st.st_mode)) {
        DIR * dirp = opendir(path);
        while (1) {
            struct dirent * entry = readdir(dirp);
            handle->enqueue(entry->d_name);
        }
        ...
        closedir(dirp)
    }
}
```


Parallel Copy: A More Involved Example

In a nutshell, there are four stages of parallel copy:

tree walk recursively walk the tree hierarchy until you reach to the leaf node, which is the actual files to be copied.

OR
walk the tree first before doing actual copying.

copy breaking up a large file into chunks and enq for processing.

clean up set permission, owner, timestamps etc.

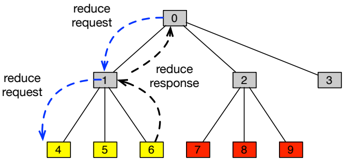
compare check the data integrity.

Tree Walk and Progress Report

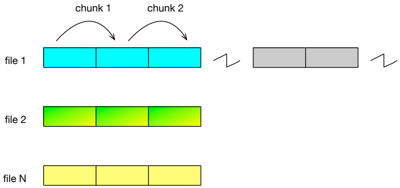
User wants to know the progress, in particular when doing a large data transfer that could take more than a few hours. For example, during Spider 1 to Spider 2 transition.

Yet, this can be difficult in a fully distributed task setup environment.

Solution
`reduce()` callback



Copy and Parallel Granularity



Verification

In the past:

- fileUtils provides a dcmp utility that can do source and destination comparison.
- dcp used to have a internal compare function, which was later deemed unreliable.

The design issues:

- We need to close the destination file handle to make sure the data is committed, from application point of view.
- We do NOT want to re-read the source from the disk.
- We want to parallelize the verification process, if possible.

Preserving Attributes

There are 4 types of attributes we need to consider:

- ownership
- permission bits
- timestamp
- extended attributes

The extended attributes are important for preserving Lustre stripe information. The basic steps:

- `mknod()` while doing the treewalk.
- `llistxattr()` to get list of names of the attributes.
- `lgetxattr()` and `lsetxattr()` to get and set the attributes.

Pythonic API: BaseTask

```
class BaseTask:
    __metaclass__ = ABCMeta

    def __init__(self, circle):
        self.circle = circle
        self.rank = circle.rank

    @abstractmethod
    def create(self):
        pass

    @abstractmethod
    def process(self):
        pass

    @abstractmethod
    def reduce(self):
        pass

```


Pythonic API Example

pcp main

```
    circle = Circle(reduce_interval=5)

    # first task
    treewalk = PWalk(circle, src, dest)
    circle.begin(treewalk)
    circle.finalize()

    # second task
    pcp = PCP(circle, treewalk, src, dest)
    circle.begin(pcp)
    circle.finalize()

    # third task
    pcheck = PCheck(circle, pcp)
    circle.begin(pcheck)
    circle.finalize()

```

DCP Usage

```
mpirun -np 8 dcp -R -p /my/src/dirA /my/dest/dirB
```

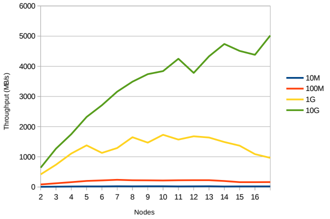
- -R: copy directory recursively
- -p: preserve original file attributes (owner, group, permission) as well as Lustre striping information.

For more complete description and batch script example:

https://www.olcf.ornl.gov/kb_articles/transferring-data-with-dcp/

Performance

DCP performance depends on a variety of factors: number of parallel processes, number of files, depth of directory, file size, and current I/O loads etc.



Throughput DCP Atlas 1000 Files

Summary

- `fileUtils` builds on the fundamental concept of doing workload distribution by *work stealing*.
- `fileUtils` can also be seen as an example of running*embarrassingly parallel* jobs on a large-scale MPI-based platform.
- With the right amount of abstraction - the circle API and associated services may have the potential to provide a Hadoop (map/reduce) like interface for the scientists.