An Introduction to fileUtils

Feiyi Wang (Ph.D)
National Center for Computational Sciences
Oak Ridge National Laboratory

March 3, 2015

Outline

1. Overview
   - Motivation
   - Solution Pattern: Master/Slave and Work Stealing
   - The Theory: Distributed Termination Detection

2. libcircle API and Examples

3. Design Considerations
   - Treewalk and Progress
   - Division of Labor: the Granularity
   - Parallel Integrity Check
   - Preserving Attributes

4. Performance Discussion

5. Summary
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 \texttt{cp}, \texttt{find} ...
- multi-threaded: \texttt{bbcp}, \texttt{xdd}
- cross-cluster: \texttt{grid-ftp}

What is fileUtils?

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

Origin: \textit{LaFon, Misra and Brinhurst: “On distributed File Tree Walk of Parallel File System”}.

fileUtils suite

- \texttt{dcmp} - compare files
- \texttt{dcp} - copy files
- \texttt{dfind} - find files by path name
- \texttt{drm} - remove files
- \texttt{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 its 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


- 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.
- 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.
- A machine only forwards the token when it is passive (no work)
- Any time a machine sends work to a machine with lower rank, it colors itself as black.
- Both initially at $n_0$, or upon receiving a token:
  - if a machine is white, it sends the token unchanged.
  - 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.
Overview

The Theory: Distributed Termination Detection

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 condition cannot be met. The token forwarding continues.

```
libcircle API and Examples

libcircle API

```
### dwalk Callback

```c
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 an 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.
**Design Considerations**  
**Preserving Attributes**

---

## Pythonic API: `BaseTask`

```python
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

```python
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

```bash
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:


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.
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.