Pwrake: A parallel and distributed flexible workflow management tool for wide-area data intensive computing

Masahiro Tanaka
University of Tsukuba
1-1-1 Tennodai, Tsukuba
Ibaraki 3058577 Japan
tanaka@hpcs.cs.tsukuba.ac.jp

Osamu Tatebe
University of Tsukuba
1-1-1 Tennodai, Tsukuba
Ibaraki 3058573 Japan
tatebe@cs.tsukuba.ac.jp

ABSTRACT

This paper proposes Pwrake, a parallel and distributed flexible workflow management tool based on *Rake*, a domain specific language for building applications in the Ruby programming language. *Rake* is a similar tool to *make* and *ant*. It uses a Rakefile that is equivalent to a Makefile in *make*, but written in Ruby. Due to a flexible and extensible language feature, *Rake* would be a powerful workflow management language. The Pwrake extends *Rake* to manage distributed and parallel workflow executions that include remote job submission and management of parallel executions. This paper discusses the design and implementation of the Pwrake, and demonstrates its power of language and extensibility of the system using a practical e-Science data-intensive workflow in astronomical data analysis on the Gfarm file system as a case study. Extending a scheduling algorithm to be aware of file locations, 20% of speed up is observed using 8 nodes (32 cores) in a PC cluster. Using two PC clusters located in different institutions, the file location aware scheduling shows scalable speedup. The extensible Pwrake is a promising workflow management tool even for wide-area data analysis.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Distributed systems; D.4.3 [**File Systems Management**]: Distributed file systems

General Terms

Design,Performance,Measurement

Keywords

workflow, file system, performance evaluation

1. INTRODUCTION

Improvement of network bandwidth in wide area enables large-scale data sharing and analysis. Data-intensive computing especially in wide area is expected to promote so-called e-Science, a scientific research area conducted in collaboration among several institutes. E-Science infrastructure which federates geographically distributed computer resources through the Internet is under research and development. In the astronomy field, the application of e-Science is indispensable to studies that require terabytes or petabytes of observational data taken by several observatories. There have been several activities of data analysis in astronomy using TeraGrid [12] and EGEE [3]. Pegasus [2] provides a mapping task from an abstract workflow defined by a directed acyclic graph (DAG) in XML to concrete workflow executions for DAGMan [1] using underlying cyberinfrastructure. In the case of Montage [7], which is a tool for astronomical image mosaicing, a workflow DAG depends on a set of input data. It means that another DAG needs to be generated for another set of input data. Visual workflow generation systems, including Kepler [5], Taverna [11], and Triana [13], make it easy for users to create and edit workflows by drag and drop of icons. However, it is still error prone and not easy to correctly create a practical complex scientific workflow.

The GXP make [10] is one of research activities to easily write and execute workflows. It utilizes Makefile to define a workflow. The features of Makefile including suffix rules and wildcard expressions are useful to define general workflows also. Users can avoid writing the same kind of tasks repeatedly by writing rules. Even if an execution fails by some reason, the workflow can be restarted without executing completed tasks.

Through a case study of the Montage workflow, it turns out that it requires more flexibility to define a scientific workflow than Makefile. A simple version of the Montage workflow requires dynamic creation of a sub-workflow since a part of workflow is undetermined before execution and it depends on the result of tasks. Such a workflow can be defined as follows; the parent *make* process generates a sub-Makefile during the workflow execution, and executes it as a child process. However, dynamic creation of Makefile is not easy to maintain nor easy to understand the behavior. Furthermore, scientific workflows often require programming features.

Therefore, we propose Pwrake, a parallel and distributed flexible workflow management tool based on a widely-used open-source tool "*Rake*" [8], a build tool in Ruby [9]. A Rakefile has enough flexibility to define the workflow that includes dynamic workflow creation during the workflow execution. This paper describes the design and implementation of Pwrake, and demonstrates its power of language and extensibility of the system using a practical e-Science data-intensive workflow in astronomical data analysis on the Gfarm file system [4] as a case study.

2. RELATED WORK

GXP [10] is a parallel shell tool written in Python, which executes a command line on specified multiple computers at the same time. The GXP make exploits the GNU make for workflow management and uses GXP as the underlying execution engine to distribute tasks across cluster nodes. Task dependencies are defined in Makefile, which is a powerful language to express workflows. It has implicit and explicit rules to execute, variable values, and shell scripts. It is possible to reduce the length of a workflow description dramatically compared to the DAG input file, and to generate a general workflow for applications. This research is inspired by the GXP make.

Swift [14] is a scientific workflow system designed for loosely coupled computations. It uses a statically typed language called SwifScript for workflow definition. Swift dispatches a workflow to another scheduler, such as Karajan, while it is not intended for users to extend the scheduler. Such batch job submission needs granularity of jobs for efficient execution. In contrast, Pwrake is designed for executing workflows consisting of more than thousands of short jobs efficiently without detail configurations.

3. DESIGN AND IMPLEMENTATION

3.1 Rake

Rake [8] is a build tool similar to *make* and *ant*, and is included in Ruby version 1.9 or later. The following is an example of Rakefile:

```
task :default => "hello"

rule ’.o’ => ’.c’ do |t|
  sh "cc -c #{t.source}"
end

file "hello" => ["main.o", "util.o"] do |t|
  sh "cc -o #{t.name} #{t.prerequisites.join(’ ’)}"
end
```

The syntax of Rakefile is same as the Ruby language. This is an internal Domain Specific Language (DSL) which exploits a host language for a particular kind of problem. In this example, the words `task` and `file` are Ruby methods defined in *Rake* for task definition. Each method takes arguments including a target name, a list of the prerequisite tasks, and an action described in a code block embraced by `do` and `end`. When `task` and `file` methods are called, an instance of Rake::Task class and Rake::File class are created, respectively. (Here “Rake::” represents the namespace of the Rake module.)

3.2 Design of Pwrake

The original *Rake* has the MultiTask class for parallel execution of prerequisite tasks in Ruby threads. The Ruby thread is implemented in user level. It means that Ruby threads are executed in a single process by sharing time. Although it is able to utilize multi-core processors by invoking external processes using the MultiTask class, *Rake* has no mechanism for controlling the number of threads nor thread pooling. Furthermore, *Rake* does not have a mechanisms for invoking processes on remote hosts.

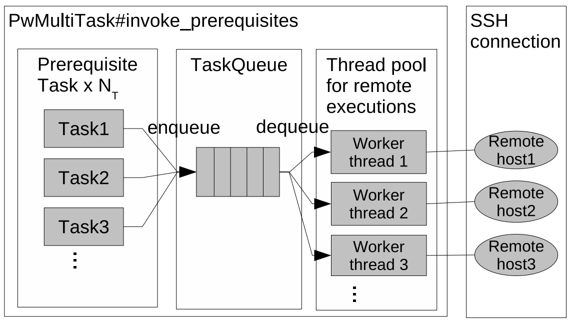

Figure 1: Design of Pwrake parallel and distributed workflow extension. Prerequisite tasks are enqueued to a task queue, which will be dequeued by a worker thread in a thread pool that executes the task in a remote host.

Pwrake is an evolution of *Rake* for distributed parallel workflows. Figure 1 shows the basic design of Pwrake. Pwrake has a task queue and a thread pool to execute tasks in a remote host or another process. Instead of creating a Ruby thread to execute a prerequisite task in the same process, enqueue it to a task queue. Each worker thread in a thread pool dequeues a task from the task queue, and executes it in a remote host. After requesting the execution in a remote host, the worker thread can yield the CPU cycle another worker thread. This design can utilize parallelism of multicores and cluster nodes.

3.3 Implementation of Pwrake

We implemented Pwrake by extending *Rake*. The following new classes are implemented.

PwMultiTask class is the major part of Pwrake for distributed parallel workflows. It inherits the Rake::Task class, and overrides the `invoke_prerequisites` method, which is one of Rake::Task class methods, to implement the mechanism shown in Figure 1. This method operates as follows: First, it creates TaskQueue and enqueues prerequisite tasks to it. Next, it creates a thread pool of worker threads for SSH connections to remote (or local) hosts. Finally, each worker thread dequeues a prerequisite task from the TaskQueue.

SSH class is a class for invoking a process in a remote host using the ssh command. It is created before the execution of a Pwrake workflow for each remote host, more precisely for each core, as specified by a user, and keeps the ssh connection during the workflow execution.

TaskQueue class manages a task queue. Prerequisite tasks will be enqueued, which will be dequeued by a worker thread.

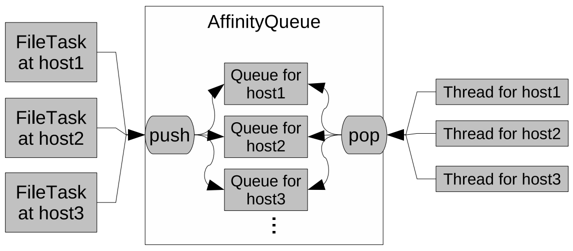

Figure 2: Implementation of Affinity Queue class.

3.4 Extensibility of Pwrake

With the modular design of Pwrake, users can add new functions to Pwrake. This design enables making the best use of modern middleware for distributed computing. In the case study, we use the Gfarm [4] global distributed file system to access data in wide area environment. The Gfarm file system utilizes local storage of compute nodes, which can be distributed to multiple sites. The Gfarm file system has many functions for efficient file access such as automatic file replica selection. However, task scheduling is managed not by a file system, but by a workflow scheduler. The task scheduling that is aware of the file location improves file access performance further. The following two classes are option classes for task scheduling.

AffinityQueue class is a class to dispatch remote processes depending on the location of input files. Figure 2 shows the implementation of AffinityQueue. When an instance of the AffinityQueue is created, it creates a set of queues assigned to every host where parallel processes are executed. AffinityQueue has `push` and `pop` methods as interface. The `push` method takes two arguments; one is a process string and the other is the name of a host where an input file is stored. When the `push` method is invoked, the process is enqueued onto all the queues assigned to the affinity hosts. On the other hand, the `pop` method takes one argument; the name of the host to which the thread is connecting. When the `pop` method is invoked from a worker thread, it dequeues the process from the queue assigned to the host.

PwAffinityMultiTask is a subclass of PwMultiTask class. Using the AffinityQueue class instead of TaskQueue, it dispatches prerequisite tasks to a proper worker thread. Before queuing tasks into AffinityQueue, it investigates the locations of input files. If a task requires multiple input files, it is not obvious which file is better to be chosen. The PwAffinityMultiTask class chooses the first input file passed as an argument so that users can specify.

GfarmSSH class is a subclass of the SSH class. To access the Gfarm file system, every worker thread requires a mount point to the Gfarm file system using `gfarm2fs` command. For this purpose, we extended the SSH class for mounting the Gfarm file system just after the start of an SSH session.

4. ASTRONOMY WORKFLOW

We apply Pwrake to a workflow for Montage, an astronomical image processing tool, as a case study.

Montage [7] is a collection of programs for combining astronomical images to generate a custom mosaic image. It is designed for science uses which require accuracy in astrometry and photometry of astronomical objects. Every task of Montage is written in a C program and can be invoked independently. This design enables parallel execution of independent tasks.

Montage workflow consists of the following tasks. In the first step, the mProjectPP program projects an input image to the coordinate system of the output image. The next step is the correction of sky brightness. In this step, the mDiff program extracts the difference of overlapped area between two images. Then the mFitplane program calculates fitting parameters of the diff images by first order. Using this parameter the mBgModel program calculates fitting parameters all over the target image. Using unified parameter, the mBackground program make brightness correction for each image. Finally the mAdd program integrates into one image.

The mProjectPP part of Rakefile is as follows (line numbers are added for explanation):

```
 1: SRCFITS = FileList["#{INPUT_DIR}/*.fits"]
 2:
 3: file( "pimages.tbl" ) do
 4:   OUTFITS = SRCFITS.map do |i|
 5:     o = i.sub( /^(.*?)([^\/]+).fits/,
 6:                'p/\2.p.fits' )
 7:     file( o => [i, HDR] ) do |t|
 8:       t.rsh "mProjectPP #{i} #{o} #{HDR}"
 9:     end
10:     o
11:   end
12:   pw_multitask( "Proj" => OUTFITS ).invoke
13:   sh "mImgtbl p pimages.tbl"
14: end
```

At line 1, `SRCFITS` is defined as an array of input file names. The whole block at lines 3-14 is the definition of a file task whose target is `pimages.tbl`. At lines 4-11, the `SRCFITS` array is iterated over with the `map` method. At lines 5-6, each input file name (`i`) is substituted to an output file name (`o`) by a regular expression. The `file` method defines a file task (lines 7-9), supplied with prerequisite files (`i`, `HDR`) and a target file (`o`). At line 8, a Rake::Task instance (`t`) receives the method `rsh`, which is extended for Pwrake, supplied with a command string of mProjectPP program as an argument. Note that defined tasks are not executed at the time of definition. Each target file name (`o` at line 10) is collected into the `OUTFITS` array (at line 4). After that, the `pw_multitask` method is called (line 12) with the `OUTFITS` array as prerequisite files. The method creates an instance of the PwMultiTask class. This instance immediately receives the `invoke` method, which is a method to execute tasks. Although the `invoke` method is automatically called from the *Rake* system, it is explicitly called here since such dynamic tasks are not involved in a global dependency. This example is demonstrating that the *Rake* system has the capability to define dynamic workflows. In the `invoke` method, the mProjectPP tasks are executed as prerequisites, and command

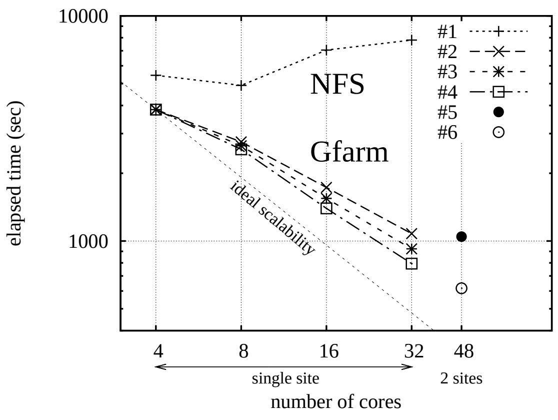

Figure 3: Elapsed time of workflow. See text for measurement details.

strings are added to AffinityQueue along with node information of input files. We note that there is no description on scheduling or file affinity in this Rakefile. We consider that this design makes it easy for users to utilize distributed resources of computers.

5. PERFORMANCE EVALUATION

The performance of Montage workflow with Pwrake is measured using two clusters at University of Tsukuba and at the National Institute of Advanced Industrial Science and Technology (AIST). In each cluster, up to eight nodes are used for the measurement. Each node at University of Tsukuba has CPU AMD Opteron 2218 (2.6 GHz), 4 cores/node, and 4 GB memory. Each node at AIST has CPU Intel Xeon CPU (2.80 GHz), 2 cores/node, and 1 GB memory. The round trip time (RTT) between two clusters is 0.8 msec.

Figure 3 shows the elapsed time of Montage workflow executions from projection of input images by mProjectPP to integration into one image by mAdd, in various configurations using 2MASS all-sky image data. The data set used in the measurement consists of 1,580 files with the total data size of 3.3 GB.

The plot #1-#4 is the measurement using 4-32 cores of one cluster at University of Tsukuba. #1 is the result that uses NFS. Using 16 cores and more, the elapsed time increases. #2-#4 are the results that uses Gfarm. All three plots show scalable performance improvement in terms of the number of cores. #2 uses Gfarm but without data location aware scheduling. #3 uses Gfarm with data location aware scheduling. #4 uses also Gfarm but input data is distributed across compute nodes. #3 and #4 improves 14% and 20%, respectively, than #2 in the elapsed time in the case of 32 cores.

#5 and #6 are the case of 48 cores in two clusters at University of Tsukuba and AIST. Both cases use Gfarm with data location aware scheduling. The difference is the distribution of input data. In the case #5, each cluster has one file replica for each input file. In the case #6, the input files are grouped by celestial coordinate of the image and each group of files is assigned to each host. This spatial grouping method reduces the intermediate file transfer between clusters as described in [6]. It improves 41% of performance.

6. CONCLUSION

This paper proposed design and implementation of Pwrake, a parallel and distributed flexible workflow management tool. Pwrake is extensible and has flexible and powerful workflow language to define scientific workflow. This paper demonstrated its powerful workflow language feature and extensibility by a case study of a practical e-Science data-intensive workflow in astronomical data analysis on the Gfarm file system in wide area environment. Extending a scheduling algorithm to be aware of file locations, 20% of speed up was observed using 8 nodes (32 cores) in a PC cluster. Using two PC clusters located in different institutions, the file location aware scheduling showed scalable speedup. The extensible Pwrake is a promising scientific workflow management tool.

7. ACKNOWLEDGMENTS

This work is a part of research on resource linkage for the formation of research communities (research on data sharing technology), which is a subtheme in the research and development of REsources liNKage for E-scIence (RENKEI) for building next-generation IT infrastructure, funded by the Ministry of Education, Culture, Sports, Science and Technology.

8. REFERENCES

[1] DAGMan (Directed Acyclic Graph Manager). http://www.cs.wisc.edu/condor/dagman/.

[2] E. Deelman, G. Singh, M.-H. Su, J. Blythe, et al. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[3] EGEE. http://www.eu-egee.org/.

[4] Gfarm. http://datafarm.apgrid.org/.

[5] Kepler. http://kepler-project.org/.

[6] L. Meyer, J. Annis, M. Wilde, M. Mattoso, and I. Foster. Planning spatial workflows to optimize grid performance. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 786–790, New York, NY, USA, 2006. ACM.

[7] Montage. http://montage.ipac.caltech.edu/.

[8] Rake. http://rake.rubyforge.org/.

[9] Ruby. http://www.ruby-lang.org/.

[10] K. Taura. Grid Explorer : A Tool for Discovering, Selecting, and Using Distributed Resources Efficiently. *IPSJ SIG Technical Report 2004-HPC-99*, pages 235–240, 2004.

[11] Taverna. http://www.taverna.org.uk/.

[12] TeraGrid. http://www.teragrid.org/.

[13] Triana. http://www.trianacode.org/.

[14] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. *1st IEEE International Workshop on Scientific Workflows*, pages 199–206, 2007.