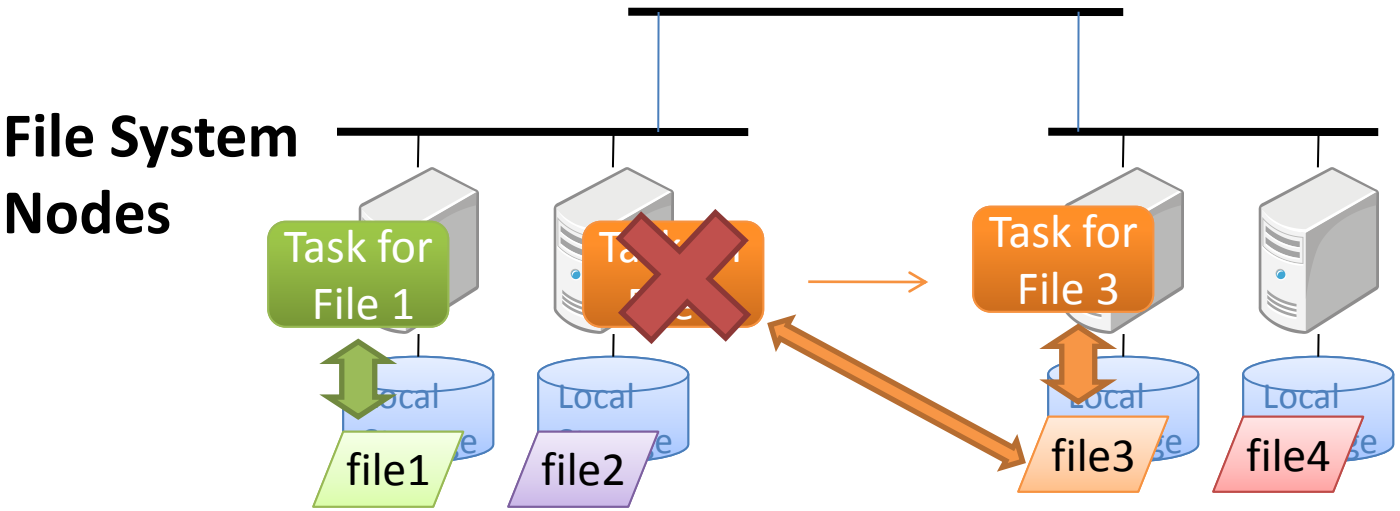
Gfarmハンズオン:
Pwrakeによる分散ワークフロー実行

田中昌宏　2010年7月2日

Gfarmにおけるタスク実行

- Gfarmの特徴：ファイルシステムノードでタスク実行が可能
- ファイルのローカリティを考慮したタスク配置により、分散処理のスケーラビリティが向上
- 適切にタスク配置を行うツールが必要



Pwrake

- Parallel Workflow extension for ***Rake***
- 分散ワークフロー実行ツール

- ***Rake*** = Ruby版make
  - Ruby言語による柔軟なワークフロー記述が可能
- **Pwrake** における拡張
  - SSHによるリモート実行
  - Gfarmファイルシステムへの自動マウント
  - ローカリティを考慮したタスク配置

Pwrake ハンズオン

- Pwakeによる分散処理を、実際に実行していただきます。
- 時間の都合上、詳細な説明は省きます。
- インストール方法、RakeおよびPwrakeの処理記述方法については、付録に掲載します。

前準備

- 端末からクラスタへログイン
- コマンドラインから赤字の部分を入力してください
- SSHの設定

```
# パスフレーズなしで鍵ペアを作成
$ ssh-keygen
# 公開鍵を計算ノードに配置
$ cd ~/.ssh
$ cat id_rsa.pub >> authorized_keys
# 接続できることを確認
$ ssh les03 hostname
```

- Ruby, Rake, Pwrake はインストールしてあります。
  （付録1にインストール方法を記載）

例1

- 各入力ファイルについて、下記の情報を、出力ファイルに記録
  - 入力ファイルが格納されているホスト名
  - タスクを実行しているホスト名
  - ワーキングディレクトリ名

例1：ファイルの展開

- Gfarmマウントポイントの下の、Gfarmユーザディレクトリに移動

```
$ cd {Gfarmマウントポイント}/{Gfarmユーザディレクトリ}
```

- 例1のファイルセットを展開

```
$ tar xzf /home/tanaka/ex1.tgz
$ cd ex1
$ ls
README  Rakefile  nodes  regist.rb
```

例1: 入力ファイルの準備

- 入力ファイルを生成し、指定ノードに登録

```
$ ruby regist.rb
$ ls
README      test10.in  test6.in
Rakefile    test2.in   test7.in
nodes       test3.in   test8.in
regist.rb   test4.in   test9.in
test1.in    test5.in
```

例1: タスク定義

```
$ cat Rakefile

SRC = FileList["*.in"]   # 入力ファイル
OUT = SRC.ext("out")     # 出力ファイル

# ルール定義
rule ".out" => ".in" do |t|
  # プロセスをリモート実行
  t.rsh "(
    echo 'filenode: '`gfwhere #{GfarmSSH.gf_pwd}/#{t.prerequisites[0]}`
    echo 'hostname: '`hostname`
    echo 'pwd:      '`pwd`
    ) > #{t.name}"
end

# 前提タスク(上記のルール定義)を並列実行
pw_multitask :default => OUT
```

例1：Pwrake の実行

- 実行ノードファイル

```
$ cat nodes
les02.omni.hpcc.jp
les03.omni.hpcc.jp
les04.omni.hpcc.jp
les05.omni.hpcc.jp
```

- Pwrakeの実行

```
$ pwrake FS=gfarm NODEFILE=nodes
```

(実行状況が表示されます)

  - Pwrakeオプション(詳細は付録3に記載):

    FS=ファイルシステム(gfarmまたは指定なし)

    NODEFILE=実行ノードのホスト名が書かれたファイル

    LOGFILE=ログファイル

例1: 実行結果

```
$ ls *.out
test1.out   test2.out  test4.out  test6.out  test8.out
test10.out  test3.out  test5.out  test7.out  test9.out

$ tail *.out
==> test1.out <==
filenode: les02.omni.hpcc.jp
hostname: les02.omni.hpcc.jp
pwd:      /tmp/tanaka000/home/tanaka/handson/example1

==> test10.out <==
filenode: les03.omni.hpcc.jp
hostname: les03.omni.hpcc.jp
pwd:      /tmp/tanaka001/home/tanaka/handson/example1
…
```

- 入力ファイルの格納ノードと、タスクの実行ノードが一致。(終わりの方のタスクでは、一致しない場合もある)
- 自動的にGfarmファイルシステムをマウントし、カレントディレクトリに移動している

例2：ソースコードのコンパイル

```
$ cd ..
$ tar xzf /home/tanaka/ex2.tgz
$ cd ex2
$ ls
README  include  lib  libtool
$ cd lib/libgfarm/gfarm
$ ls
```

例2:タスク定義(抜粋)

```
$ cat Rakefile
…
SRC = FileList[“liberror.c”, ..]    # 入力ファイル
LOBJ = SRC.ext(‘lo’)                # 出力ファイル

# ルール定義:
rule ".lo" => ".c" do |t|
  # プロセスをリモート実行
  t.rsh "#{LTCOMPILE} -c #{t.prerequisites[0]}"
end

# 前提タスク(上記のルール定義)を並列実行
pw_multitask "libgfarmcore.la" => LOBJ do |t|
  sh "#{LTLINK} -o #{t.name} #{t.prerequisites.join(' ')}"
end

# デフォルトターゲット
task :default => "libgfarmcore.la"
```

例2：Pwrake の実行

- 実行ノードファイル

```
$ cat nodes
```

- Pwrakeの実行

```
$ pwrake FS=gfarm NODEFILE=nodes
```

- ターゲットファイルの確認

```
$ ls -l libg*
libgfarmcore.la
```

付録1．インストール方法

（インストール先によっては、ルートの権限が必要）

Rakeのインストール

- Ruby 1.9.x には、Rakeが添付されているため、インストールは不要
- Ruby 1.8.x には、Rakeが添付されていないため、別途インストールが必要
  - ソースからインストール

```
wget http://rubyforge.org/frs/download.php/56872/rake-0.8.7.tgz
tar xvzf rake-0.8.7.tgz
cd rake-0.8.7
ruby install.rb
```

  - Rubygems : Rubyパッケージングシステム

```
gem install rake
```

  - Redhat系

```
yum install rubygem-rake
```

  - Debian系

```
apt-get install rake
```

Pwrakeのインストール

- ソースの取得
  - Gitが使える場合
    - `git clone git://github.com/masa16/Pwrake.git`
  - Githubのウェブページから取得
    - http://github.com/masa16/Pwrakeにアクセスし、Download のリンクをクリック
    - アーカイブを展開
- インストール
  - ソースのディレクトリに移動
  - `ruby setup.rb`

付録2．Rakeの簡単な説明

Rake

- Ruby版のmake
- タスク定義文法
  - 独自の文法ではなく、Rubyの文法を利用。
  - このようにホスト言語を利用した言語は、**内部DSL** (Internal Domain-Specific Language)と呼ばれる。
  - Rubyスクリプトとして実行されるため、Rubyの言語仕様をフルに利用できる。
  - ビルドツールとしてだけではなく、ワークフロー定義にも有用

Rakefile 記述例: ソースコードのビルド

```
SRCS = FileList["*.c"]
OBJS = SRCS.ext("o")

rule  "*.o" => "*.c" do |t|
  sh "cc -o #{t.name} #{t.prerequisites[0]}"
end

file "prog" => OBJS do |t|
  sh "cc -o prog #{t.prerequisites.join(' ')} "
end

task :default => "prog"
```

タスク定義

```
task :default => "prog"
```

- task
  - タスクを定義するための、Rakeで定義されたメソッド。
- :default
  - コロン(:)で始まる文字列は、Rubyにおいてシンボルを表すリテラル。
  - タスク名として、task メソッドの引数として与えられる。
  - Rakeにおいて、:default タスクは、最終ターゲット表す。
- :default => “prog”
  - => は、Rubyにおいて、key-value 引数を表す。
  - :default (最終ターゲット)が、”prog” タスクに依存することを表す。

ファイルタスク定義

```
file "prog" => OBJS do |t|
  ..
end
```

- file
  - FileTask を定義するための、Rakeで定義されたメソッド。ファイルのタイムスタンプによって実行される点が task と異なる。
- do |t| .. end
  - Rubyコードブロック。中括弧 {} でも記述できる。コードブロックとは無名関数のようなものであるが、変数のスコープはブロックの外側と共有している。
  - コードブロックは、file メソッドに渡され、タスクの依存関係の順に実行される。
  - |t| は、コードブロックへ渡される引数を表す記法。Rakeのタスク定義では、t として、Rake::Taskクラスのインスタンスが渡される。

外部コマンド起動

```
sh "cc –o prog #{t.prerequisites.join(' ')}"
```

- sh
  - 外部コマンドを起動するための、Rakeで定義されたメソッド。
- 2重引用符 ".."
  - 文字列を表すRubyのリテラル。バックスラッシュ記法 と式展開が有効になる。
  - #{..}により、括弧内の式を評価して埋め込む。
- t.prerequisites
  - t は、Rake::Task クラスのインスタンス。prerequisites メソッドは、自身が依存するタスクを配列として返す。
- join(' ')
  - Arrayクラスのメソッド。要素を文字列として連結する。引数の文字列が、各要素の間に入る。

ルールの記述

```
rule "*.o" => "*.c" do |t|
  sh "cc -o #{t.name} #{t.prerequisites[0]}"
end
```

- rule
  - ルールを記述するためのメソッド。
- "*.o" => "*.c"
  - 拡張子が .o のファイルが、拡張子が .c のファイルに依存することを表す。
- t.name
  - タスクの名前を返すメソッド
- t.prerequisites[0]
  - タスクが依存するファイルの最初の要素

ファイルリスト

```
SRCS = FileList["*.c"]
OBJS = SRCS.ext("o")
```

- `FileList["*.c"]`
  - “*.c”で展開されるファイルリストを配列で返す。
- `SRCS.ext("o")`
  - SRCS配列のそれぞれの要素について、拡張子を .o に置き換えた配列を返す。*.o ファイルは最初は存在しないので、FileListは使えない。

タスクの動的生成

- ルールでは書けないような複雑な場合でも、プログラミングによって、タスクを生成できる。

```
files = FILES["*.o"]
for i in 0..1
  task files[i] do |t|
    sh "cc #{t.name}"
  end
end
```

付録3．Pwrake 拡張

Pwrakeにおける拡張

- （仕様は変更される可能性があります）
- **`t.rsh`**
  - 外部コマンドをリモートで実行する
  - Taskクラスインスタンスのメソッド
  - 現在はSSHのみサポート
- **`pw_multitask A => B`**
  - タスク(A)が依存する複数のタスク(B)を並列に実行する。並列数は、使用コア数によって決まる。

Pwrake実行オプション

```
pwrake FS=gfarm NODEFILE=.. LOGFILE=..
```

- FS
  - gfarm ： SSH接続の際に、Gfarmファイルシステムをマウントし、ローカルのカレントディレクトリに相当するディレクトリに移動する。
  - 指定なし ： SSH接続した後、ローカルのカレントディレクトリの絶対パスに移動
- NODEFILE
  - 使用するノードの情報として、各行にホスト名とコア数をスペース区切りで記述したファイルを指定する。
- LOGFILE
  - ログを出力するファイル名を指定。ファイルは、./log ディレクトリに保存される。
- そのほか、Rakeのオプションが使用できる。