Efficient Parallel Shell

Georges-André Silber
Centre de Recherche en Informatique, École des mines de Paris
35, rue Saint-Honoré, 77305 Fontainebleau cedex, France
E-Mail: Georges-Andre.Silber@ensmp.fr

ABSTRACT

We propose a slightly modified Unix shell where the user can explicitly or implicitly execute commands on different computers of a cluster. The shell redirects output and input of those different commands to and from the right hosts, for instance when the user uses the redirections `|` (pipe), `>` (write to a file) and `<` (read from a file). We use a simple model of remote execution implemented as a portable *Remote Execution Daemon* (RED) that has to be executed on each node of the cluster and that can be seen as a highly simplified operating system. We show with a simulation that those very simple shell constructs coupled with a RED could lead to significant performance results on clusters.

Keywords: Unix shell, remote execution, parallel execution, pipelined execution.

1. INTRODUCTION

A Unix shell offers several ways to express parallelism. For instance, `A|B|C` describes a pipelined execution of processes `A`, `B` and `C` that collaborate in parallel to produce a result; the operator `&` launch a process without waiting for its result, giving the possibility to launch other processes in parallel.

On computers with a processor executing a single thread of instructions, processes are not executed in parallel. They are only executed in parallel if the computer has several processors or at least several execution units and if the operating system is capable of distributing the processes among execution units. Grouping some computers with the same architecture into a cluster is a widely used way of running parallel programs. But, it does not lead directly to a parallel execution of processes, because those processes must be executed remotely on computers of the cluster. Programmers usually have to write specific programs using TCP/IP sockets or message passing libraries like MPI to obtain a parallel execution of communicating processes.

Transparent remote execution of processes on computers of a cluster can be obtained with a *single system image* operating system [1, 5]. It gives the user the illusion that the cluster is a single computer with several execution units. The underlying operating system is responsible for processes allocation on computers, load-balancing, input/output redirections, file systems management, etc.

We propose a different approach where a modified Unix shell allows the user to explicitly or implicitly launch commands on different computers of a cluster. The shell redirects output and input of those different commands to and from the right hosts, for instance when the user uses the redirections `|` (pipe), `>` (write to a file) and `<` (read from a file). An important aspect of our work is that the user has the choice to name the host where he wants to run a task or to let the shell decide for him where to run the task. Another aspect is that the user does not have to transfer the file he wants to execute to the right hosts, the shell taking care of that for him.

Our approach requires a modification of the shell but no modification of the operating system. It uses a simple model of remote execution materialized as a *Remote Execution Daemon* (RED) [7] that has to be executed on each node of the cluster. This daemon implements a simple service for remote program execution and file storage and can be seen as a highly simplified operating system. Our shell process is in fact a client of many RED running on distant hosts.

RED and the modified shell are highly portable on POSIX systems. We show with a simulation that those very simple constructs coupled with a simple RED could lead to significant performance results on clusters.

The organization of this paper is as follows. First, we present our extensions to the Unix shell. Second, we present performance results for a simple pipeline in two cases: CPU-bounded and IO-bounded application. We show that in every case our approach leads to speedups. Third, we give a sketch of the implementation of our extensions in GNU BASH using a C implementation of a RED using XML-RPC [10, 4]. Finally, before we conclude, we discuss some related works.

2. PARALLEL SHELL

Our reference shell is GNU BASH and we based our extensions on the documentation that can be found in [2]. We describe our parallel shell in an informal way, without giving many implementation details. We are going to see in section 4 that our extensions can be easily implemented.

We use the first restriction that all computers share the same architecture. The second restriction is that the executable file and the required dynamic libraries of the command that is executed on a remote host must be present on the local host. They are transferred on demand by the shell on the remote host. With this last restriction, the computers of the cluster does not need to share a file system.

2.1 Local shell and remote hosts

When a shell is started, we consider that it has a list of available remote hosts that have the same architecture and that accept commands (typically, the computers of the cluster). We consider in a first approach that this list is read from a file when the shell starts and that it cannot change during shell execution.

The shell sends a message to each host of this list, getting a token for each host. Those tokens represent keys to private virtual file systems that are created on each host for the client shell when it asks for a token. Those file systems can be empty and there is no guarantee of persistence for the files stored on the remote host. The private virtual file systems on the remote hosts can be implemented in memory or in actual disks. The client shell is the only one that can access its private file system.

2.2 Simple commands

A simple shell *command* is a sequence of words separated by blanks, terminated by one of the shell's control operators (a newline or one of the following: '`||`', '`&&`', '`&`', '`;`', '`;;`', '`|`', '`(`', or '`)`'). The first word generally specifies a command to be executed, with the rest of the words being that command's arguments.

In our parallel shell, the first word, the command to be executed, can be prefixed by a sequence of characters of the form `host:`, where `host` is the computer where the command has to be executed. For instance, the command

```
node0:command a b c
```

runs the `command` program on the host `node0` with the arguments `a`, `b`, and `c`. The remote command does not see the local files. All files used by `command` must be on `node0`. The shell is responsible for transferring the `command` executable and its associated dynamic libraries to `node0` if they are not already there. The command can be the path where to find the command locally. For instance, the user can write:

```
node0:/usr/bin/command a b c
```

and the local command `/usr/bin/command` is going to be transfered to `node0`, creating on the fly the directories if needed.

In a natural way, standard input stream is taken from the terminal that launched the command and standard output and error streams are redirected to the terminal. A command executed this way does not receive any PID (Process IDentifier), because no process is created on the local host. It only receives a job number in the BASH sense and can then be manipulated with BASH built-in commands.

2.3 Redirections

We add a special case for file names that are used in redirections. The user can prefix filenames with a `host:`, where `host` is the remote computer where the file must be written or read. For instance, the command

```
node0:command > node1:file1 < node2:file2
```

use the remote file `file1` as input stream of the remote command `command` that writes its result in the remote file `file2`. To copy the resulting file locally, it is sufficient to write

```
cat > localfile < node2:file2
```

2.4 Pipeline

The way we handle pipelines is one of the main aspect of our work. A pipeline is a sequence of simple commands separated by '`|`'. The format for a pipeline is

```
command1 [| command2 ...]
```

where the output of each command in the pipeline is connected via a pipe to the input of the next command. That is, each command reads the previous command's output. We modify the pipe semantic the following way: if at least one command of the pipe is a remote execution, the pipe becomes a direct network connection. We are going to see in section 3 that this aspect is crucial in terms of performance.

2.5 Predefined parameter to get a remote host name

We add a new special parameter of the shell, the parameter `$:` that contains the name of a remote host accepting commands. Two consecutive uses of `$:` do not necessarily give the same result. By default, we consider a round-robin policy where we just pick the next remote host on the static list. Other policies can be implemented, to take for instance load-balancing aspects into account.

With this special parameter, it is possible to put the name of a distant host in a variable like

```
MYHOST=$:
```

and to use it the following way

```
$MYHOST:command
```

The user can also directly write

```
$::command
```

leaving to the shell the placement decision.

We also add a variable called REDHOSTS that contains the list of all available hosts. In the following, we give an example where this special variable is used to run a single command on all available hosts.

2.6 Examples

With our parallel shell, it is possible to write commands with explicit placement like:

```
producer | node1:task1 | node2:task2 > node3:dfile
cat < node3:dfile > lfile
```

where `node1`, `node2` and `node3` are remote computers. The data produced by the process `task1` running on `node1` is sent directly to the standard input of the process `task2` running on `node2`, without returning to the host running the process `producer`.

The special parameter `$:` is convenient to exhibit an implicit placement like:

```
DESTHOST=$:
producer | $::task1 | $::task2 > $DESTHOST:dfile
cat < $DESTHOST:dfile > lfile
```

where the shell decides where to run the different tasks.

It is also possible to run a single command on multiple hosts at the same time:

```
for node in $REDHOSTS
do
  $node:task &
done
wait
```

Note the '&' character that is necessary to run all the processes in parallel. The `wait` command is a BASH command that waits for the completion of all processes running in the background [2].

3. PERFORMANCE RESULTS

The results presented here come from a simulation we ran before we began the actual implementation of our extensions into GNU BASH and the development of RED with XML-RPC. We wanted to validate our ideas by experiments that are now motivating our developments. We show that our approach gives significant performance results when executing communicating processes in parallel under the form of a pipeline. We provide in [6] a file containing all C source codes and shell scripts we have used during our experiments.

We ran experiments using the following pipeline, where `x` is the number of packets of 1024 bytes that are produced by the program `producer` and consumed by the program `consumer`. Each packet contains random data.

```
producer x | task w | task w | task w | consumer x
```

The standard output of `producer` is transmitted to the standard input of a program `task`, that executes the following steps: 1) read a packet of 1024 bytes, 2) iterate `w` times over this packet, executing two arithmetic operations on each byte, and 3) write the packet to the standard output. The program `consumer` just read `x` packets, one at a time, and does not write anything on standard output.

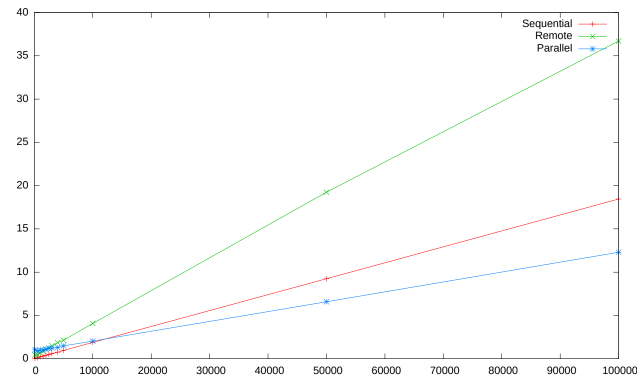


Figure 1: Sum of user and system time in seconds (y axis) for an increasing number of 1024 bytes data packets exchanged (x axis). IO-bound version of the pipeline (2048 iterations per packet).

Sequential experiments were done using a Pentium 4 computer running Debian GNU/Linux. Its processor is clocked at 2.4 GHz and has 2 GB of RAM (this computer is called `stockholm`). The +-line called Sequential in Figure 1 gives the running time (user and system time) in seconds of the previous pipeline for numbers of packets `x`. The parameter `w` is set to 2 ($1024 \times 2 = 2048$ iterations per packet). We call this version the IO-bound version because most of the running time is taken by input/output operations. Figure 2 represents the same experiment with a parameter `w` set to 1024 ($1024 \times 1024 = 1048576$ iterations per packet). We call this version the CPU-bound version because most of the running time is taken by computing operations.

Our next experiments consisted in the evaluation of two distributed execution schemes: one that uses the `rsh` command, and another that uses a parallel execution scheme that can be expressed with our shell extensions. We used for these experiments three more computers: a computer with two Opteron 244 processors at 1.8 GHz with a 8 GB shared RAM (`surville`), a computer with a Pentium 4 computer at 2.4 GHz with 2 GB of RAM (`saigon`), and a computer with a Pentium 4 computer at 3 GHz with 2 GB of RAM (`nantes`). The four computers are connected using a 100 Mbit/s Ethernet switch that is the main switch of our laboratory. We are not exactly in a cluster, but the differences between the schemes of executions that we present should remain in a

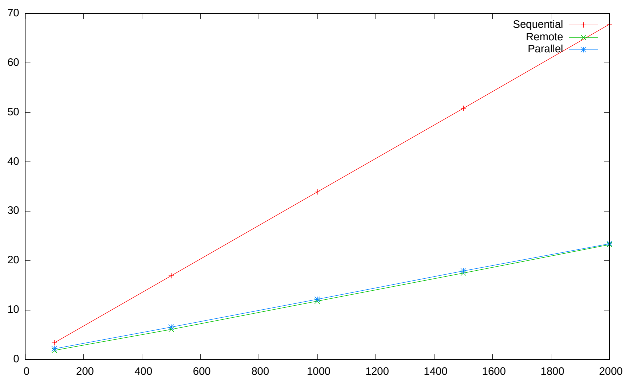


Figure 2: Sum of user and system time in seconds (y axis) for an increasing number of 1024 bytes data packets exchanged (x axis). CPU-bound version of the pipeline (1048576 iterations per packet).

true cluster environment. All four computers run Debian GNU/Linux.

The first distributed scheme of execution, that we call the *remote* execution scheme, is as follows:

```
producer x | (rsh surville task w)
           | (rsh saigon task w)
           | (rsh nantes task w) | consumer x
```

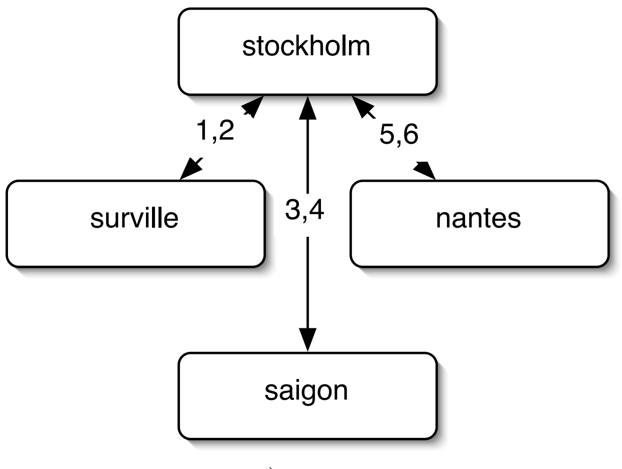
where `producer` and `consumer` are executed on the host `stockholm` and the three `task` processes are remotely executed on the hosts `surville`, `saigon`, and `nantes`. Three bi-directional network connections are established between the host `stockholm` and the hosts `surville`, `saigon`, and `nantes`. The order of data movements are depicted in Figure 3 a).

We can see in Figure 3 that the remote scheme of execution transforms the host `stockholm` in a potential bottleneck for data transfers. This is verified in Figure 1 where we can see that the $\times$-line representing the execution times is far above the sequential execution for the IO-bound case. On the contrary, this scheme of execution leads to a linear speedup depicted in Figure 2 for the CPU-bound case.

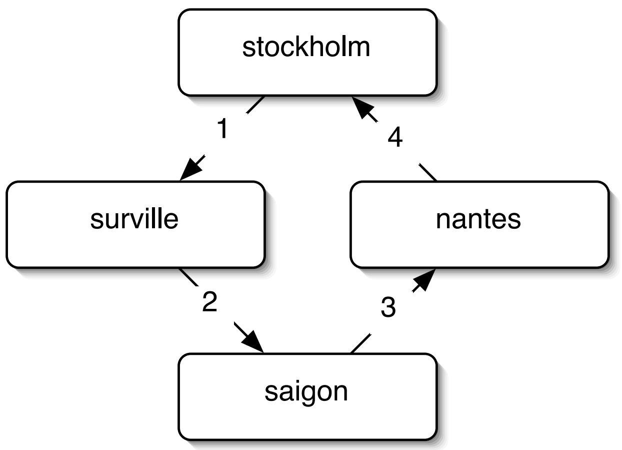
The second distributed scheme of execution, that we call the *parallel* execution scheme, is as follows:

```
producer x | surville:task w | saigon:task w
           | nantes:task w | consumer x
```

where an explicit direct network connection is made between `stockholm` and `surville`, `surville` and `saigon`, `saigon` and `nantes`, and `nantes` and `stockholm`. The order of data movements are depicted in Figure 3 b). We can see that this parallel scheme of execution has no bottleneck. This is verified in Figure 1 where we can see that the $*$-line representing the execution times becomes better than the sequential execution for the IO-bound case, when the number of packets



a) remote



b) parallel

Figure 3: Communication schemes for the a) remote and the b) parallel executions of the pipelined command.

exchanged exceed 10^5. This scheme is equivalent to the remote execution scheme for the CPU-bound case depicted in Figure 2.

These encouraging results confirmed that our very simple extensions can lead to significant performance results with a very easy syntax. So, we decided to implement those extensions as discussed below.

4. IMPLEMENTATION IN GNU BASH USING RED-XML

Our developments are twofold. First, we are implementing in C a simple daemon for remote execution and storage, following the RED interface described in [7]. Second, we are modifying the source code of GNU BASH version 3.0 to add our extensions. The use of those extensions is going to be an option when BASH is launched.

Our implementation of RED is on top of a regular Linux system and consists in an HTTP server waiting for messages in XML-RPC format. Each kind of message represents a call to one method of the RED interface. Basically, this interface allows to transfer files and to remotely execute commands, with flexible input/output redirections in sockets. The main point is that this interface permits the implementation of each extension of our parallel shell under the form of one or several calls to methods of one or several RED servers. In fact, the conception of the RED interface has been driven by the needs that appear during the conception of our parallel shell.

5. RELATED WORK

The most recent work that shares some ideas with our work is [9]. Compared to our work, they do not address the problem of pipelines and they do not give a simple and coherent shell syntax to place explicitly tasks on remote computers. The work in [8] is more an effort to build a Single System Image than a parallel shell. It does not address explicit placement nor pipelines. The work in [3] focus on Grid environments and its main purpose is to provide a work around to build virtual grids composed of hundred of heterogeneous machines. It does not provide any implicit placement of processes nor any mechanism to automatically transfer on-demand executable files to the remote hosts. To finish, none of the works cited above give any real or simulated performance result for the execution time of a pipelined application.

6. CONCLUSION AND FUTURE WORK

We introduced a new syntax to express explicit or implicit simple parallelism on clusters with Unix shell constructs. We gave significant performance results that motivate our approach. We are now implementing the RED interface using XML-RPC and we are modifying GNU BASH with our parallel extensions.

This simple system is not as powerful and generic as a message passing interface like MPI nor a Single System Image operating system, but it is far more simple to use and to implement, and it does not require any modification in existing programs nor operating systems. Furthermore, people familiar with the shell should be able to use those new constructs in a very natural way.

We plan to extend our framework towards heterogeneous networks of workstations. An easy but not very elegant way to achieve this is to store locally several versions of the commands and to transfer the proper version at the right time.

7. REFERENCES

[1] Barak, A., and La'adan, O. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems 13*, 4 (Mar. 1998), 361–372.

[2] Free Software Foundation (FSF). GNU BASH. Web. `http://www.gnu.org/software/bash/bash.html`.

[3] Kaneda, K., Taura, K., , and Yonezawa, A. Virtual Private Grid : A Command Shell for Utilizing Hundreds of Machines Efficiently. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid)* (may 2002).

[4] Laurent, S. S., Dumbill, E., and Johnston, J. *Programming Web Services with XML-RPC*. O'Reilly, 2001.

[5] Morin, C., Gallard, P., Lottiaux, R., and Valée, G. Towards an Efficient Single System Image Cluster Operating System. *Future Generation Computer Systems 20*, 2 (2004).

[6] Silber, G.-A. Experiments for coset-2 workshop. Web. `http://www.cri.ensmp.fr/people/silber/metacc/red/coset02.tar.gz`.

[7] Silber, G.-A. Remote Execution Daemon (RED): A Simple Service for Remote Execution and Storage. Tech. Rep. E-267, CRI/ENSMP, Apr. 2005. `http://www.cri.ensmp.fr/classement/doc/E-267.pdf`.

[8] Tan, C., Tan, C., and Wong, W. Shell over a Cluster (SHOC): Towards Achieving Single System Image via the Shell. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2002)* (sep 2002), pp. 28–36.

[9] Truong, M., and Harwood, A. Distributed shell over peer-to-peer networks. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications* (Las Vegas, Nevada, 2003), pp. 269–275.

[10] Wikipedia, the Free Encyclopedia. Xml-rpc. Web. `http://en.wikipedia.org/wiki/XML-RPC`.