社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－HPC－99 (40)
2004／8／1

Grid Explorer : A Tool for Discovering, Selecting, and Using Distributed Resources Efficiently

KENJIRO TAURA[†,††]

We describe GXP, a shell for distributed multi-cluster environments. With GXP, users can quickly submit a command to many nodes simultaneously (approximately 600 milliseconds on over 300 nodes spread across five local-area networks). It therefore brings an interactive and instantaneous response to many cluster/network operations, such as trouble diagnosis, parallel program invocation, installation and deployment, testing and debugging, monitoring, and dead process cleanup. It features (1) a very fast parallel (simultaneous) command submission, (2) parallel pipes (pipes between local command and all parallel commands), and (3) a flexible and efficient method to interactively select a subset of nodes to execute subsequent commands on. It is very easy to start using GXP, because it is designed not to require cumbersome per-node setup and installation and to depend only on a very small number of pre-installed tools and nothing else.

1. Introduction

Working with a large number of distributed resources is troublesome. Among many difficulties, one that everybody immediately faces is the lack of tools efficiently supporting 'everyday' operations, such as parallel command submissions, multi-nodes file replications, and process cleanup. Of course, there are many popular tools that execute a *single* such operation, such as rsh and ssh[8] for command submissions, and rcp, scp, rsync, and cvs for file replications. There are also Grid-oriented version of some of such tools including globus (globus-run)[4] and GridFTP. However, when it comes to efficiently manipulating many (say, > 100) nodes spread across multiple administration domains, we need a substantial amount of effort to combine them.

While there have been much progress in Grid middleware including job schedulers[3], Grid-enabled job submission[4], and GridRPC[9], relatively little attention has been paid to improving efficiency of daily operations. This potentially keeps potential application developers away from the Grid, and makes them stick to more comfortable single cluster or SMPs, despite their small scale. We believe improving our daily experience on the Grid will accelerate research and development on all areas of Grid software.

† Department of Information and Communication Engineering, Faculty of Engineering, University of Tokyo
†† Japan Science and Technology Agency

To this end, we are developing Phoenix Grid Tools, a set of tools to improve user's daily experience on the Grid. This paper describes one of such tools, GXP (which stands for Grid Explorer), an interactive shell for the Grid. Elsewhere, we have described early versions of a similar tool, VPG[7] and MPSH[1], and a high performance file replication tool NetSync[5]. GXP inherits many of the features of VPG and MPSH and improves upon them in many areas. Through our experiences with GXP, we feel GXP is a powerful tool to enhance the productivity of distributed operation and programming, and its power comes from the ability to quickly and instantaneously perform simple tasks involving many (> 100) nodes. Moreover, we noticed that many of simple parallel tasks (e.g., parameter sweep or master-worker style computation) can be comfortably accomplished solely with this tool + simple and ad-hoc scripting with no or little network programming. Section 2.6 gives several interesting examples.

Rest of the paper is organized as follows. Section 2 describes design of GXP. Section 3 shows performance measurement. Section 4 mentions related work and Section 5 states conclusion and future work.

2. GXP Design

2.1 Design Constraints

GXP is designed from the beginning to meet the following constraints.

- **Overcome Connection Restrictions:** It works in typical network configuration

where many of inter-subnet or inter-cluster connections are blocked by firewalls and NATs. It finds their ways to reach necessary nodes, trying nested logins as necessary, without assuming too much help given by the user.

- **No Per-Node Installations, No Daemons:** It does not require permanent daemons specific to GXP on each of the resources. This significantly reduces its initial setup cost. Moreover, it does not require explicitly installing GXP program on any of the remote resources. Once installation on the user's home node has been done, s/he is ready to use GXP and it is installed on each of the designated resources automatically. Currently the source file of GXP is a single python file (about 2,500 lines), so installing it on the user's home node takes a single file copy or download to whatever places the user wants.
- **Minimum Prerequisites:** Along the same line, GXP is designed so that it depends only on a small number of pre-installed software that are considered "standard" on Unix platforms. It is thus likely the case that these prerequisites are already met in the user's environment. We detail the current prerequisites in Section 2.3.
- **Fault Tolerance:** It has a simple fault tolerance that does not stuck on dead nodes but leaves them behind.

2.2 Using GXP

Using GXP involves the following steps.

- **Preparation:** Write a configuration file describing a small number of "key nodes" and the user's login names. This is necessary only for the first time or when it must be modified. Details are in Section 2.3.
- **Explore Phase:** Launch GXP, which brings up a GUI like Figure 1. With this GUI, the user can explore the network, discovering other nodes, selecting nodes the user wants to use later, and building a connection tree among them. Details are in Section 2.4.
- **Shell Phase:** The user enters an interactive shell in which s/he can dynamically select nodes to execute commands on and issue command lines to the selected nodes, as many times as desired. Details are in Section 2.5. The user can switch back and forth between the explore phase and the shell phase.

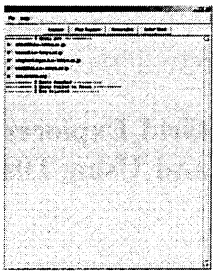

Fig. 1 GXP GUI

2.3 Preparation

GXP configuration file basically specifies a list of nodes the user wants to use, along with login names suitable for each of them. However, literally listing *all* nodes is already overwhelming and error-prone for users. It is particularly so if the user uses different sets of nodes for different jobs. We would like to retain configuration files mostly static, while allowing selection of desired resources per session.

To this end, GXP can start with a small number of manually specified nodes and *find their neighbors automatically.* We currently use NIS for neighbor discovery where available. As a consequence, a typical GXP configuration file only specifies one node for each LAN (NIS domain, to be more precise). When a user has an access to a remote cluster or a LAN, s/he typically remembers a designated gateway host to login, from which other hosts in the same cluster or the LAN can be reached. GXP configuration file naturally fits this model.

2.4 Explore Phase

This phase is an interactive process in which the user, through the GUI, is presented with a list of node names and their status, checks nodes s/he wants to use, and launches an "Explore" command, which tries to reach them.

See Figure 1 again. GXP displays all nodes whose names have been found, along with their statuses on their left column. A status is either *reached,* indicated by a character `'o'` in the column, or *not yet reached* indicated by a checkbutton. Initially no nodes have been reached (except for the local node, which is not displayed). All nodes that have been reached are connected by available remote shell sessions. They form a tree whose root is the user's home node and children of a node p are the nodes directly reached from p by one of the available remote shell command. While our current im-

plementation recognizes only ssh and rsh as a remote shell command, GXP can use any command that can remotely invoke an arbitrary command line and gives the local process handles to the standard input/output/error of the remote process. We hereafter call this tree *the login tree*. It is initially a singleton tree whose only node is the user's local host.

When users wish to reach additional nodes, they check their buttons and presses the "Explore" button in the top row. Then GXP tries to expand the login tree to include them. On each of the newly reached nodes, GXP searches for names of its neighbor nodes and newly discovered names are displayed in the GUI. The user repeats this process (i.e., check buttons and then press the Explore button) until s/he reaches all the wanted nodes.

2.5 Shell Phase

When the user presses "Enter Shell" button in the GUI, GXP enters a shell phase and prompts the user for a command. Table 1 summarizes the list of available commands. Among them, the most basic is the 'e(xec)' command which executes a given command on all the "selected" nodes. We detail later how selected nodes are determined. For now, it suffices to say all nodes that GXP reached are selected by default. Thus, the command

`e hostname`

executes `hostname` command on all nodes, displaying all reached host names on the user's terminal.

An argument of `'e'` command can be an arbitrary shell command syntax including pipes, redirections, environment variables, and command substitution. In addition, `'e'` command accepts an extended syntax, which we call "parallel pipes" discuss below.

2.5.1 Parallel Pipes

Running an identical or a similar command on all nodes is already useful in many circumstances, but through our experiences, we found that the ability to connect inputs/outputs of parallel commands to local commands makes GXP much more powerful. This is a natural extension to Unix pipes, so we call this facility *parallel pipes*.

The full syntax of the `'e'` command is as follows.

e L {{ P }} R

where L, P, and R are all Unix shell command syntax. The behavior is as follows (see Figure 2).

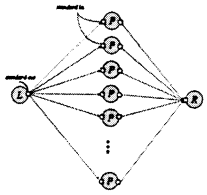

Fig. 2 Behavior of Parallel Pipe $L\{\{P\}\}R$

- L and R are executed locally.
- P is executed in all selected nodes.
- Standard outputs of L is sent to the standard input of each of P. That is, the output is broadcast.
- Standard outputs of P are merged and sent to the standard input of R. This merge is undeterministic and by default, the granularity is a single line (i.e., a single line is not intervened by other characters).

Here are some default rules.

- When L is omitted, it defaults to a ':' command (a command that immediately terminates).
- When R is omitted, it defaults to cat, which effectively displays the outputs of P.
- When both L and R are omitted, the user can simply abbreviate {{ P }} to P.

Simple examples are given below.

- `e {{ hostname }} sort` will list all selected host names in an alphabetical order. This is often easier to read for human beings and more appropriate for configuration files that must list machine names (e.g., MPI's machinesfile). `e {{ echo 'hostname':2 }} sort` will exactly generate a MPI's machinesfile (two CPUs for each node).
- `e cat` *file* `{{ cat >` *remotefile* `}}` will effectively copy a local file named *file*, to each of the selected nodes under name *remotefile*. This is very useful to copy sources/configuration files/input data of parallel applications.

  `e tar cvf -` *directory* `{{ tar xvf - }}` will do the same thing for a directory.

We will see more interesting combinations later in Section 2.6.

2.5.2 Node Selection

Another feature that turned out to be vital is an efficient mechanism to select nodes on which parallel commands are run. Nodes on which a command should be run differ depending on tasks and stages. For example,

e *cmd*	execute *cmd* on selected nodes
l *cmd*	execute *cmd* on the local node
cd *dir*	change directory to *dir* on selected nodes
lcd *dir*	change directory to *dir* on the local node
export *var=val*	set environment variable *var* to *val* on selected nodes
smask	select nodes whose last command status are zero
rmask	select all nodes
bomb	clean up processes (see text)

Table 1 Summary of GXP commands in shell phase.

- for many of the file system operations such as file/directory transfers and compiling applications, a single node should be selected for each (NFS-shared) file system.
- in heterogeneous environments, we may sometimes need to work separately for each architecture, this time on Linux, this time on Solaris, etc.
- it will be common to drop busy nodes from the selection.
- for testing and debugging, a small number of nodes are often selected.
- for testing and debugging, only nodes in a single cluster are often selected.
- for production runs, as many nodes as possible will be selected.

It is difficult to anticipate in advance what kind of node selections will become useful, so the users must be able to select nodes interactively as the needs arise.

For this purpose, we introduce a builtin command called *smask* (*set mask*). Its effect is to select nodes on which the last command succeeded (i.e., exited with status zero). Therefore, to select some nodes, the user performs the following steps.

(1) issues a command that should succeed on (and only on) nodes the user like to select, and

(2) issues smask command. Then, subsequent commands will be executed on the nodes on which the first command succeeded.

(3) To doublecheck, after a selection has been made, issuing **e hostname** will show the nodes actually selected.

GXP's prompt shown in Figure3 displays the number of nodes on which the last command succeeded, therefore the user often can have some confidence about the selection before issuing **smask**.

smask - does the reverse. It will select nodes on which the last commands failed. Command *rmask* (*reset mask*) command will revert to the default selection of all reached nodes.

```
GXP[32/124/211]>>>
```

Fig. 3 An example GXP prompt. The three numbers separated by slashes represent, from left to right, the number of nodes on which the last command succeeded, the number of nodes currently selected, and the number of nodes reached.

Here are some examples.

- To select nodes in a particular cluster, the following is often adequate.
  ```
  GXP[96/96/96]>>> e hostname|grep domain
  GXP[32/96/96]>>> smask
  GXP[32/32/96]>>>
  ```
  The first command succeeds only on nodes whose names contain a string *domain*. So if the user knows a string that discriminates a cluster, it succeeds on the desired cluster. By looking at the prompt at the second line, the user will learn it succeeded on 32 nodes. If it matches the user's knowledge about the number of nodes in the cluster, the user will have confidence before actually issuing **smask** command.
- The following command will select Linux nodes.
  ```
  GXP[211/211/211]>>> e uname|grep Linux
  GXP[160/211/211]>>> smask
  GXP[160/160/211]>>>
  ```
- A more tricky but frequently used technique is to select a single node for each file system. Suppose for the sake of simplicity that the current working directory of all nodes are the user's home directory, which may or may not be shared between nodes. Typically, nodes within a single cluster share a home directory, and nodes across clusters do not. We would like to elect a single node from each shared home directory, to perform subsequent file operations safely. An interesting trick is to use mkdir command.
  ```
  GXP[211/211/211]>>> e mkdir xxxx
  ... many error messages
  saying directory already exists ...
  ```

```
GXP[5/211/211]>>> smask
GXP[5/5/211]>>>
```

Command 'mkdir' should succeed for one node per a physically distinct home directory. We regularly use this technique to deliver files to all nodes, compile sources on each cluster, etc.

- Combinations of Unix commands give us powerful ways to select nodes dynamically. For example, the following will select nodes based on their load averages.

```
GXP[211/211/211]>>> uptime \
| awk '{if($(NF-2) > 0.5)print "H"}' \
| grep H
GXP[1/211/211]>>> smask
GXP[1/211/211]>>>
```

Command **uptime** will display the host's load average of the past one minute in the third from the right column (obtained via `$(NF-2)` expression in the awk command). The first command will succeed on nodes whose load averages are higher than 0.5. Such selections are often useful in cluster troubleshooting.

2.5.3 Process Cleanup

One of the biggest headaches in developing cluster/Grid software is process cleanup. Due to software bugs or operation errors, processes that should terminate might keep running, processes that terminated might leave as zombies, etc. Although fixing software bugs so that they almost never happen is an ultimate solution, in practice, we sometimes have to act retroactively or periodically so as to clean up (i.e., kill) whatever processes should have terminated. Doing so on every single cluster node is a nightmare. GXP supports a command, called *bomb*, which kills all processes of the user, except those constituting the current GXP session. In our experiences, this command is vital for making cluster/Grid programming productive.

2.6 Experiences

We have found thorough our experiences a number of interesting ways to use parallel pipe constructs. The essence is that, the parallel pipe construct of GXP establishes communication channels between processes on behalf of the user and make them available through standard input/output. Therefore, it often happens that no network programming is necessary to implement a simple coordination between processes, and even existing Unix tools fit for a purpose. This is again analogous to Unix redirections/pipes where the programmer can manipulate files and communicate with other processes without knowing the details of how they are done at the lower level. GXP brings this concept to distributed/parallel programming setting, where standard input/output of processes connect to those of processes of other nodes or even of other clusters, hiding much more complexities than the regular Unix setting. Such applications/templates include:

- A master-worker scheduler that distribute commands to available resources. As opposed to executing the same command on *all* available resources, this application executes a given command on a single available resource. Communication between master and workers are done through the parallel pipe.
- An MPI-like parallel program launcher in which processes should know each other's contact address (hostname and its listening port number) to begin with. Gathering announcements of contact addresses and broadcasting them to all processes are done through the parallel pipe.

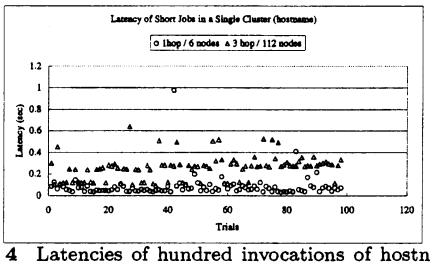

Fig. 4 Latencies of hundred invocations of hostname commands in a single cluster (average: 82 msec on 6 nodes and 260 msec on 112 nodes).

3. Basic Performance Measurement

Our primary interest is latencies of short jobs. Figure 4 is the record of one hundred invocations of **hostname** commands, inside a single cluster. The latency is the time between the point the command was issued at the local host and the point all standard output have been sent to the user's terminal.

In this single cluster setting, GXP maintains a perfect quintanary tree, so it reaches 6 nodes with $\leq$ 1 hop, 31 nodes with $\leq$ 2 hops, and 156 nodes with $\leq$ 3 hops. Since the cluster's node count is 112, we experimented with 6 and 112 nodes. We also had experimented with 31 nodes, but the result was not essentially different from the 112 nodes case. Mean values are

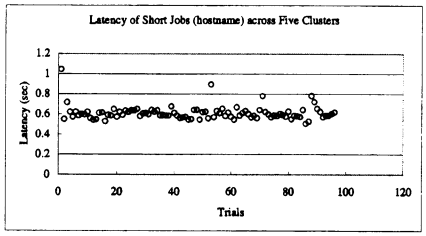

Fig. 5 Latencies of hundred invocations of hostname commands in five clusters (average: 611 msec on 327 nodes).

82 msec on 6 nodes and 260 msec on 112 nodes.

Figure 5 shows the case where nodes are spread across clusters. GXP built what seemed the "best" tree with some user interactions. That is, the local host directly reached each of the cluster gateways, from which all other cluster nodes are reached. The node count was 327 and the most distant node was five hops away from the local host. The average latency was 611 msec.

These numbers indicate GXP actually maintains an interactive response even for very shot jobs, at least up to this number of nodes.

4. Related Work

Prior to GXP and MPSH, MPD reports the best result[2] regarding quick process invocation on a large number of nodes (approximately two seconds on a 211 nodes cluster). The target of MPD is a single cluster environment. Gfpmd[6] aims at fast process invocation across clusters. An interesting difference is that whereas MPD/Gfpmd assume a daemon is permanently servicing *all* (or at least many) users on each node, GXP assumes *each* user brings them up in the Explore phase. This allows us to get rid of authentication on individual job submissions, even when machines are spread across multiple administration domains. This model also significantly reduces installation cost.

5. Conclusion and Future Work

We described GXP, a shell for Grid environment. Getting started with GXP requires a setup only on a local host, so it has a very low entry barrier. It features a fast command submission (611 msec on 327 nodes across five LANs), a flexible model of node selection, and a powerful parallel pipe syntax. These features together enhance the productivity of many interactive cluster/Grid operations. GXP makes simple coordination of processes trivial with no or little network programming. GXP will be available for download by the end of 2004 3Q from the author's home page.

Acknowledgement

We are very grateful to the group of GXP initial users for their discussion and feedback. The ideas of writing master-worker programs in GXP is largely due to my colleague Yoshikazu Kamoshida.

References

1) Ando, M., Taura, K. and Chikayama, T.: A Command Shell for Supporting Parallel Job Submission in Grid Environment, *Proceedings of Symposium on Advanced Computing Systems and Infrastructure (SACSIS2004)*, Vol. 2004, pp. 225–232 (2004). (in Japanese).
2) Butler, R., Gropp, W., and Lusk, E.: A Scalable Process-Management Environment for Parallel Programs, available from http://www.mtsu.edu/ rbutler/.
3) Buyya, R., Abramson, D. and Giddy, J.: Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid, *HPC Asia 2000*, pp. 283–289 (2000).
4) Globus Home Page: http://www.globus.org/.
5) Hoshino, T., Taura, K. and Chikayama, T.: An Adaptive File Distribution Algorithm for Wide Area Network, *Proceedings of Workshop on Adaptive Grid Middleware* (2003).
6) Iwasaki, S., Matsuoka, S., Soda, N., Hirano, M., Tatebe, O. and Sekiguchi, S.: Implementation and Evaluation of a Scalable Job Management Architecture for Large-Scale PC Cluster on the Grid Environment, *Proceedings of Hokke 2002* (2002). (in Japanese).
7) Kaneda, K., Taura, K. and Yonezawa, A.: Virtual private grid: a command shell for utilizing hundreds of machines efficiently, *Future Generation Computer Systems*, Vol. 19, No. 4, pp. 563–573 (2003).
8) OpenSSH: http://www.openssh.com/.
9) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *Journal of Grid Computing*, Vol. 1, No. 1, pp. 41–51 (2003).