GXP : An Interactive Shell for the Grid Environment

Kenjiro Taura (University of Tokyo/JST)

***Abstract*— 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. We describe how GXP achieves these features and demonstrate through examples how they make many otherwise boring and error-prone tasks simple, efficient, and fun.**

I. 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 [1] 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 (globusrun) [2] 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.

Let us take a parallel command submission for example. When developing distributed applications, we often need to launch an identical or a similar command on all nodes involved in a computation. An obvious example is the case when we develop a parallel software running on the Grid. We must not only be able to quickly run the executable on many nodes, but also be able to, among others, copy sources and/or configuration files, compile them, and copy the executables. When using clusters, we frequently have to kill processes on each of the nodes. Each of such operations needs to run an identical or a similar command on some selected nodes.

One could imagine an ad-hoc solution in which the user writes a small script that issues rsh or ssh as necessary. Besides consuming user’s time of writing such scripts in the beginning, this approach quickly becomes unmanageable in large scale by several reasons. First, some nodes may not be directly reachable from the user’s home node, due to firewalls or NATs. In such circumstances, it is not enough to issue ssh/rsh command to each node. The user instead needs nested logins. Second, directly issuing a remote login command to all nodes from the user’s home does not scale even if it is possible at all. It would incur unacceptable latencies, especially for short jobs. Finally, in large scale environments it is usual that some nodes are dead, so we must have a mechanism to avoid being stuck on them.

While there have been much progress in Grid middleware including job schedulers [3], Grid-enabled job submission [2], and GridRPC [4], 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.

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, an interactive shell for the Grid. Elsewhere, we have described early versions of a similar tool, VPG [5] and MPSH [6], and a high performance file replication tool NetSync [7]. GXP inherits many of the features of VPG and MPSH and improves upon them in many areas. They include ease of use and initial setup, amount of manual configurations, dynamic node selections, and flexibility of communication between processes (parallel pipes). 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 III shows several interesting examples including a simple job scheduler and a flexible parallel program launcher.

Rest of the paper is organized as follows. Section II describes design of GXP. Section III demonstrates several example tasks of which GXP can significantly enhance the productivity. Section IV shows performance measurement. Section V mentions related work and Section VI states conclusion and future work.

II. GXP DESIGN

A. 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 II-C.
- **Fault Tolerance:** It has a simple fault tolerance that does not stuck on dead nodes but leaves them behind.

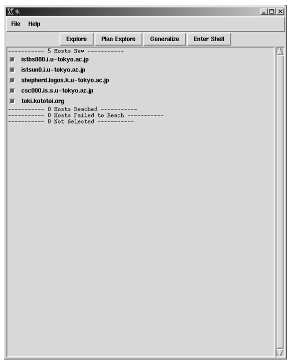
B. Using GXP

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 II-C.
- **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 II-D.
- **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 II-E. The user can switch back and forth between the explore phase and the shell phase.

C. 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.



```
----------- 5 Hosts New -----------
istbs000.i.u-tokyo.ac.jp
istsun0.i.u-tokyo.ac.jp
shepherd.logos.k.u-tokyo.ac.jp
csc000.is.s.u-tokyo.ac.jp
toki.kototoi.org
----------- 0 Hosts Reached -----------
----------- 0 Hosts Failed to Reach -----------
----------- 0 Not Selected -----------
```

Fig. 1. GXP GUI

```
hosts = [
    "gw.www.abc.p.u-tokyo.ac.jp",
    "gway.p.u-tokyo.ac.jp",
    "gateway.xxx.q.u-tokyo.ac.jp",
    "pub.yyy.w.u-tokyo.ac.jp",
    "public.zzz.org"
    ]

default_user = "tau"
user_map = [
    ("abc.p.u-tokyo.ac.jp", "tau"),
    ("p.u-tokyo.ac.jp", "taue")
    ]
```

Fig. 2. An Example Configuration File

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.

Suffixes of host names can be used to specify a login name common for a set of nodes. Figure 2 specifies an example configuration file. This is derived from the author's actual configuration file (with node names changed for our security) from which we reach more than 300 nodes (500 CPUs) spread across 5 clusters.

D. 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.

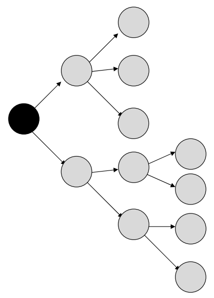


Fig. 3. A Login Tree. A node is a host, the root node the local host, and an edge an rsh/ssh connection.

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 (Figure 3) and children of a node p are the nodes directly reached from p by one of the available remote shell command. While our current implementation 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.

Issues involved in this phase are:

1) Bootstrapping or Automatic Installation: As described, GXP does not require the user to install it on each of the remote nodes. It instead makes a temporal copy of the GXP program file (about 2,500 lines of python code as of writing) to a remote node each time GXP logs in the node. The temporal file gets removed when GXP program quits a session, unless it unexpectedly crashes. Note that the program is not really installed in the literal sense that it permanently stays in the remote nodes' disks. It is instead temporarily copied for every GXP session, to every single node. We made this decision to make upgrading GXP software trivial.

An interesting bootstrapping issue that arises is how to minimize pre-installed applications this automatic installation process depends on. We copy the program only assuming the availability of python and a remote shell command, and nothing else, both on the local and the remote side. Logically, installing the program on a remote node involves the following.

- Check if the destination directory exists and create it if it does not. Currently, we use `gxp_work` directory under the user's home directory. The entire directory can be removed when there is no active GXP sessions of the user.
- Generate a unique temporary file name under the destination directory.
- Copy the program under the generated file name.

We execute them by letting python interpreter read its program via its standard input (i.e., giving '-' as a program file name). We feed its standard input with a copy of the script, specifying options so it executes the above operations, writes the name of the installed file to its standard output, and then quit. The local host waits for the file name to appear in the standard out, and considers the login failed if it does not come back in a specified time.

2) Searching for the Login Tree: When an Explore button is pressed, the local node makes a plan on how to reach requested nodes (i.e., nodes whose buttons are checked). In graph terminology, the problem is how to extend the current login tree so that the extended tree spans all the checked and reached nodes. A login attempt is made along the new edges of the extended tree. Among many criterion defining good trees, the most important criteria is of course to draw edges on which a login will succeed. Most nodes in a LAN/cluster are typically not directly accessible from external hosts, so ideally, such edges should not be chosen without being tried. Yet, it appears difficult to quickly perform this kind of selections without user's help. In our earlier work [6], we proposed a simple configuration file syntax to concisely represent network configurations. We believe the approach was valid, but also felt that as a user-friendly tool, we should seek an alternative too.

At present, GXP implements a very naive heuristic to draw edges between nodes. As the strongest rule, GXP never selects an edge on which a login previously failed in the current session, so by repeating explores, a single host is tried to reach along different paths. Otherwise, the heuristic works as follows. Requested nodes are chosen in a random order, and for each node, its parent is chosen in the current login tree. Once chosen, the tree data structure is extended by adding the requested node and then we proceed to the next requested node. To choose the parent, nodes already in the current login tree (including ones added in the current planning process) are scored based on a weighted sum of the following three scores.

- A score based on similarity of its name to the requested node.
- A score based on its number of children. When it becomes closer to a predetermined constant (currently five), the node will get a higher score.
- A score based on its depth. Nodes at lower depths get higher scores.

These heuristics are useful to get nodes within a LAN/cluster connected together, while maintaining the shape of the tree close to a perfect k-ary tree (where k is currently five).

However, this still does not help in finding the "gateway" node of a LAN/cluster. That is, which nodes should be first reached from outside the LAN/cluster? We do not have a sufficiently reliable and portable solution yet. Asking the user to supply information is one possibility. GXP's current workaround is to exploit the fact that the user normally writes only gateways in their configurations, so GXP initially knows a small number of nodes. Assuming this, we currently ask the user to *reach all the gateways first* during the Explore phase, and then explore inside clusters/LANs. This perfectly works most of the time because gateways are normally globally reachable. Even if some are not, since GXP does not perform failed logins again, simply retrying Explore will hopefully find the right path after several attempts. Of course this will become infeasible once the number of clusters become large and many of their gateways are open only to certain selected hosts.

3) Secure Single Sign-On: To login many machines, it is clearly undesirable for users to type their passwords/passphrases each time they must authenticate themselves to a new machine. It is instead highly desirable to have a single sign-on capability in which they have to type their passphrases only once to access all machines they have accesses to.

GXP's single sign-on capability is limited by that of the underlying remote shell commands (ssh and rsh) and configurations. GXP currently never carries credentials over network. Thus, if an explore phase attempts to login a machine and is asked to input a password or a passphrase, that login attempt simply fails.

As of writing, GXP provides no particular help on this issue. We believe typical users won't have much trouble with this, because:

- Inside a cluster, users can normally hop between nodes via rsh without passwords (.rhosts or .hosts-equiv authentication). This is unlikely to change because among others, MPI typically needs them. Therefore the main issue is how to authenticate a user to a gateway from outside the LAN.
- One simple approach is to reach all gateways directly from the local host with ssh public key authentication, if this is possible. Use ssh-agent and ssh-add so that ssh invoked by GXP does not ask passphrases.
- Although not implemented yet, GXP could forward ssh-agent connection when invoking ssh where allowed. This single sign-on feature built in ssh allows a remote gateway to authenticate the user to a third gateway. This would work up to not too many number of gateways.
- If all the above are somehow not possible or undesirable, setup public/private keys with empty passphrases and occasionally refresh them.

E. 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 I 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
```

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 I
SUMMARY OF GXP COMMANDS IN SHELL PHASE.

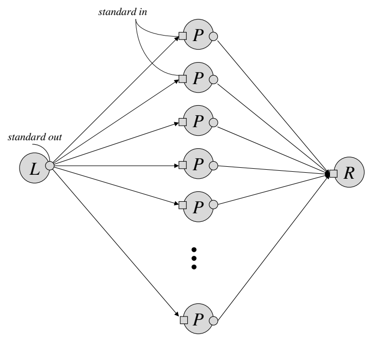


Fig. 4. Behavior of Parallel Pipe $L\{\{P\}\}R$

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.

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 4).

- 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 III.

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,

- 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 Figure5 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

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

Fig. 5. 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.

will revert to the default selection of all reached nodes.

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 exist ...
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.

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. More precisely, the `bomb` command does the following on each node.

- Run `ps` command with `-efl` option, which lists all processes along with their parents and their effective user id.
- Climb the process tree from the current (GXP) process, until we encounter a process with a different effective user id. In the end of this process, we normally encounter a remote login daemon (e.g., sshd) that born the current GXP session on the node. We call the last process of the same effective user id the root process.
- Kill all processes of the same user id, except the children of the root process.

In our experiences, this command is vital for making cluster/Grid programming productive.

III. Illustrating Applications

We have shown so far individual GXP commands and their uses. In this section, we show several examples showing how GXP enhances the productivity of network/distributed/parallel programming. 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.

A. Parameter Sweep : Static Case

Consider executing many command lines using many nodes. A single command line should be executed on an arbitrarily selected single node. Typically, these command lines run the same sequential program with different arguments. This model of parallel execution is often used in Grid environments and called embarrassingly parallel, parameter sweep, task farming, and so on. We hereafter call a single command line a single *task*. The goal is to execute all tasks as fast as possible using many nodes.

```
host1: ./app 0 1 2
host2: ./app 2 3 4
host3: ./app 3 4 5
...
```

Fig. 6. An example task file

GXP's model of execution is to run a *single* command line on *all* nodes, so it is slightly different from the parameter sweep model. Fixing the gap needs a little scripting. To make our problem setting more specific, consider in this section a simple case where the work assignment is static. That is, we can determine which tasks should be executed on which nodes before executing the first task. In practice, we often need more dynamic work assignments, which will be discussed later. In the simple setting we consider for now, we can implement the parameter sweep model as follows.

- Tasks are described in a file, called a *task file*, a single task per line.
- In addition, each line of the task file is labeled with the name of the host that should execute it. Figure 6 shows an example task file, where a label is put on the first field of each line.
- The task file is broadcast to all nodes using the parallel pipe syntax (i.e., `cat taskfile {{ ... }}`).
- Running on each node is a simple script that filters lines based on the label and executes whatever passed the filter. In this example, the following simple awk script suffices.

```
awk '{ if ($1 == h) system($2); }' \
        FS=":" h='hostname'
```

To summarize, the following GXP command line executes all commands listed in the task file, distributing work according to labels in the `taskfile`.

```
e cat taskfile {{ \
        awk '{ if ($1 == h) system($2); }' \
        FS=":" h='hostname' }}
```

B. Parallel Program Launcher

In the above example, the communication pattern of parameter sweep applications exactly fit to the parallel pipe model of GXP. The communication occurs only between an individual worker and the coordinator (master). GXP routes these messages on its established channels, thereby freeing the programmer from network programming altogether.

Many parallel programs, however, have more complex communication patterns and hence need direct any-to-any communication. Consider launching a parallel program whose participating nodes communicate with each other via sockets (e.g., mpirun). A common problem is *port assignment and advertisement.* Since it is not practical to statically fix the port number of the processes, launching a parallel program often involves a protocol to announce individual processes' listening port number. A bootstrapping problem occurs as to how to announce these port numbers. Normally, a process that launches these remote processes serves as a hub. GXP's parallel pipes is ready for this purpose and makes writing such protocols trivial as follows.

- Each node emits a single line describing its ⟨ *hostname, port number* ⟩ pair to its standard out.
- These lines are collected (via the parallel pipe syntax `{{ ... }}` ...).
- The only problem is how to feed this information *back to* each node. For this purpose, we use Unix's FIFO (a.k.a. named pipe). These lines are written to a FIFO, read by a local process and broadcast to all processes, again by the parallel pipe syntax.

To summarize, the following command line lets all processes know each other's hostname and port number.

```
e cat q {{ ./proc ... }} head -np > q
```

where `q` is a named pipe and `np` the number of processes.

Using this framework, we wrote a simple network diagnosis/measurement tool called "all-to-all-connector" which simply brings up a process on each node and reports which pairs (out of N^2, where N is the number of processes) can be directly connected via the `connect` system call. It took 73 lines of python script. It will be easy if not trivial to extend this program so that it measures latencies between nodes.

A particularly instructive in this example is the fact that, using FIFO this way, we can establish a bidirectional channel between the local node and remote nodes. This gives a very powerful framework which encompasses master-worker style computation, where communication occurs only between a worker and the master.

C. Parameter Sweep : Dynamic Work Assignment

We have addressed earlier how to run parameter sweep parallel applications assuming static work assignment. In this paragraph, we extend the model to work assignment based on dynamic availability of nodes. The framework we consider is one in which each worker individually indicates to the master its availability. The master then knows the pool of available workers. A master sends a description of a task to each of available workers, until all tasks have been finished.

In the simplest model, a worker considers it being available when there is no unfinished task currently assigned to it. More sophisticated criterion are possible, such as ones based on the host load average, memory availability, and so on, but their differences basically boil down to the differences in the definition of "availability." Thus, this section sticks to the simple greedy model in which a worker simply repeats getting a task and executing it, and the master simply dispatches tasks to workers in FIFO manner.

Following the idea of static work assignment case discussed in Section III-A, what we need in this case is a way to dynamically label tasks based on availability of workers. A worker's availability is naturally indicated by a message from the worker to the master, which in GXP amounts to a single `printf` which writes its host name to the standard output. Using the idea introduced in Section III-B, we can feed it back to the master via Unix named pipe. The master reads a single line from a task file, which in this case only describes tasks (command lines, without destination host names), *and* the named pipe. This effectively synchronizes an available worker with a single task. When the matching has been done, a line describing destination host and the task is emitted to the master's standard out. To summarize, we run the following parallel pipe.

```
e master q taskfile \
  {{ worker 'hostname' }} cat > q
```

where

- `master` is a program that repeats reading a single line from each of `taskfile` and `q`, combining them in a single line, and writing it to the standard out.
- `worker` is a program that initially writes its name (given in the command line argument) to the standard out, and then repeats reading a line from standard input and executing tasks labeled with its name, ignoring other lines.

Currently, `master` takes 89 lines of python code and `worker` 109 lines, with simple support for status reporting and logging.

We have used this very simple code to run $11,750$ tasks over 350 Xeon CPUs, each taking $1,000$ to $4,000$ seconds. During the run one node got rebooted, and a power outage occurred which leads to disconnecting the master from the network. GXP at present has no provision for fault tolerance of individual jobs, so when a node crashes or a network disconnects during a single command execution, the application must reboot and continue. The master code is designed so that it does not reissue the finished task, so when the above events occurred, we simply killed all running processes and restarted the master and workers. The work continued without duplication and finally completed.

This experience indicates GXP is useful for parallel processing over multiple clusters. Of course, the claim is limited to the extent that the life time of the application is not too long to make node/network failures real issues. Nevertheless, we believe such a quick and ad-hoc parallelization will attract many application developers.

IV. Basic Performance Measurement

Our primary interest is latencies of short jobs. Figure 7 is the record of one hundred invocations of `hostname` commands, inside a single cluster. The cluster's spec is in Table II. 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 ≤ 1 hop, 31 nodes with ≤ 2 hops, and 156 nodes with ≤ 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 82 msec on 6 nodes and 260 msec on 112 nodes.

Figure 8 shows the case where nodes are spread across clusters. Nodes that participated are listed in Table III. GXP

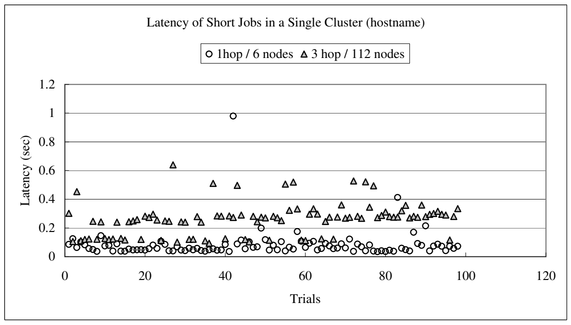


Fig. 7. Latencies of hundred invocations of hostname commands in a single cluster (average: 82 msec on 6 nodes and 260 msec on 112 nodes).

Name	IBM BladeServer
CPU	dual Xeon 2.4GHz/2.8GHz
Nodes	112
Memory	2GB/node
Network	Dell PowerConnect Gigabit Ethernet
OS	Linux

TABLE II
THE CLUSTER USED IN THE EXPERIMENT OF FIGURE 7

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.

V. Related Work

GXP is primarily related to (1) work on fast process invocations/management in distributed environment, and (2) work on shell in distributed environment.

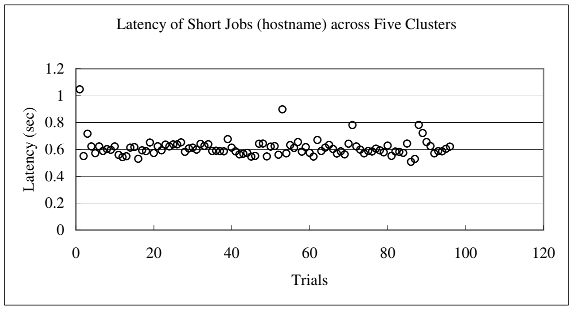


Fig. 8. Latencies of hundred invocations of hostname commands in five clusters (average: 611 msec on 327 nodes).

OS	Nodes
Linux	1
Linux	112
Linux	64
Solaris	119
Linux	31

TABLE III
FIVE LANS USED IN THE EXPERIMENT OF FIGURE 8

A. Fast Process Invocation/Management

To the best of our knowledge, MPD reports the best result [8] 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 [9] aims at fast process invocation across clusters. Their main design choices are similar to those of MPD. There are several interesting differences in design choices of MPD/Gfpmd and GXP, as discussed below.

- MPD/Gfpmd assume a daemon is permanently running on each node, whereas GXP assumes no permanent daemons except for standard remote login daemons. GXP essentially lets individual users invoke *user level, per session* daemons (through remote logins performed by the Explore phase), each time the user starts a GXP session. Our approach significantly reduces installation and management burden with extra session start up cost.
- While the startup time may be inconvenient for the user, we believe it is difficult to avoid in our target environment where nodes may belong to different administrative (security) domains and different users may have accesses to different set of domains. MPD seems to assume daemons operate in a single administrative domain. Gfpmd requires authentication for individual job submission. GXP abandons the idea of having a single daemon for all users. By maintaining a separate tree for individual users, we get rid of authentication on individual job submissions on Grid environment. Another benefit is since it completely runs in the user level (except using standard remote shell daemons), separations between users is simply made up to the operating system. GXP does not have to be a highly trusted piece of software, as was the case in one-daemon-for-all approach. Another side benefit is GXP allows incremental deployment by individual users, whereas the daemon approach must persuade system administrators to be deployed.
- MPD uses a ring topology to connect daemons, Gfpmd a hierarchy of rings, whereas GXP uses a (quintanary by default) tree. We believe our better latency result (611 msec on 327 nodes across five clusters, compared to about two seconds on 211 nodes in a single cluster [8]) is in large part due to this process topology (i.e., a shorter hop count). Gfpmd paper [9] reports that Gfpmd spends a large amount time in building a job-specific network connections for standard I/O and signal delivery. GXP does not make new connections for each command, but uses the same connections (login tree) for all commands.

- GXP, designed as a shell in the first place, supports a flexible and interactive way of node selections (`smask` command), for which MPD/Gfpmd have no particular support.

Scalable parallel Unix commands [10] are built on MPD and provide a set of Unix-like tools written in MPI. They show many interesting examples where parallel programs can be used not only for high performance computation, but also for interactive cluster operations. GXP shares this spirit. Their focus is the design of an individual Unix-like command in parallel environment. On the other hand, GXP stresses the power of combining existing serial Unix commands through our proposed parallel pipes. We also noticed through our experience the importance of efficient and interactive node selections, for which GXP proposes a powerful and interactive method.

B. Shells for Distributed Environments

GXP was the last descendant of our earlier work, VPG [5] and MPSH [6]. VPG implemented a transparent job submission across firewall/NAT. It also featured an elaborate self-stabilization after a network or a node fault occurs, but assumed a somewhat cumbersome per-cluster installation and configuration. MPSH realized many of the GXP's features including automatic installation on individual cluster nodes, fast parallel command submissions, and ease of configurations. GXP's main new features are parallel pipes and node selections.

VI. 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, as demonstrated by parameter sweep scheduler and an MPI-like parallel program launcher. GXP will be available for download by the end of 2004 3Q from the author's home page.

Acknowledgement

We thank William Gropp for the discussion at IWIA conference and pointing me at relevant work. The ideas of writing master-worker programs in GXP is largely due to my colleague Yoshikazu Kamoshida.

References

[1] "OpenSSH home page," http://www.openssh.com/.
[2] "Globus home page," http://www.globus.org/.
[3] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: an architecture of a resource management and scheduling system in a global computational grid," in *HPC Asia 2000*, 2000, pp. 283–289.
[4] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "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.
[5] K. Kaneda, K. Taura, and A. Yonezawa, "Virtual private grid: a command shell for utilizing hundreds of machines efficiently," *Future Generation Computer Systems*, vol. 19, no. 4, pp. 563–573, 2003.
[6] M. Ando, K. Taura, and T. Chikayama, "A command shell for supporting parallel job submission in grid environment," in *Proceedings of Symposium on Advanced Computing Systems and Infrastructure (SACSIS2004)*, vol. 2004, no. 6, 2004, pp. 225–232, (in Japanese).
[7] T. Hoshino, K. Taura, and T. Chikayama, "An adaptive file distribution algorithm for wide area network," in *Proceedings of Workshop on Adaptive Grid Middleware*, 2003.
[8] R. Butler, W. Gropp, , and E. Lusk, "A scalable process-management environment for parallel programs," available from http://www.mtsu.edu/ rbutler/.
[9] S. Iwasaki, S. Matsuoka, N. Soda, M. Hirano, O. Tatebe, and S. Sekiguchi, "Implementation and evaluation of a scalable job management architecture for large-scale pc cluster on the grid environment," in *Proceedings of Hokke 2002*, 2002, (in Japanese).
[10] E. Ong, E. Lusk, and W. Gropp, "Scalable unix commands for parallel processors: A high-performance implementation," available from http://www-unix.mcs.anl.gov/ gropp/bib/papers/index.htm.