NAME

nmrShell − Parallel Execution of Shell−Scripts

SYNOPSIS

nmrShell −par *serverListName* **−sh** *scriptName* [**−path** *remotePath*] [**−to** *uSecDelay*] [**−log**]

DESCRIPTION

nmrShell is a utility for executing a given shell−script simultaneously on a group of computers listed in a text file. The computers can be UNIX workstations connected via a network, or they can be processors within a multi-CPU UNIX machine. Since **nmrShell** is specifically intended to support the use of **NMRPipe** 3D and 4D parallel processing schemes, this document describes the details of such schemes, as well as the use of **nmrShell** to execute them.

When **nmrShell** executes a shell−script on a list of computers, it divides the terminal screen into sub−windows, one for every computer. Each sub−window is labeled with the associated computer's name (hostname) and CPU number (for multi-processor machines). During execution of the shell−script, a given sub−window displays all output messages generated by the corresponding computer. The **−log** option of **nmrShell** allows each computer's output messages to be written to a log file in addition to being displayed on the screen.

nmrShell uses the UNIX remote−shell command "**rsh**" to execute a given shell−script. So, **nmrShell** will only work correctly if the **rsh** command can successfully reach all the computers of interest. Furthermore, the specified shell−script, and all programs and data required by the shell−script, must be accessible to all computers of interest. This usually means that the computers will share data and programs via common directories (NFS, networked file systems).

The remote command used by **nmrShell** has two parts. The first part changes to a particular directory on the remote computer via the UNIX command **cd**. The second part uses the UNIX command **sh** to execute the specified shell−script. By default, the directory used for the remote command is the same as the directory where **nmrShell** is invoked. This implies that the network is configured so that a given shared directory has the same name on all computers of interest, although other configurations can be accommodated.

Since the network and filesystem configurations play such a critical part in **NMRPipe** parallel processing schemes, it is **strongly advised** that suitable conditions should be established and verified before attempting to perform parallel processing. If possible, these conditions should be established with the assistance of your system manager, especially if network security is an issue. Some hints are provided in the **CONDITIONS** section below.

PARALLEL PROCESSING WITH NMRPIPE

Given the appropriate network conditions, ordinary 3D and 4D **NMRPipe** scripts can be used for parallel processing with only *minor modifications*, leading to improved speeds for time−consuming calculations. The method and conditions are explained below; the explanations assume familiarity with **nmrPipe** and **xyz2pipe**.

An **NMRPipe** parallel processing scheme includes the following aspects, described in more detail later:

List of the Computers Involved: the computers to be used for parallel processing should be listed in a text file called the *server list file*. The file can be created with a text editor such as **vi**. The server list file also includes estimates of each computer's relative speed.

Access to Data and Programs: all of the computers must have access to relevant data and programs. This will generally be done via networked file systems, which are disks and directories shared by two or more computers. You can establish these shared directories with your system manager.

Dividing the Processing Task: the input spectral data will be divided into sections by the **xyz2pipe** program, so that each computer will process an appropriate sub−set of the overall data. The division will be performed automatically according to information in the server list file. In practice, a complete processing task is performed via *a single* **NMRPipe** shell−script, which will process a different sub−set of the data depending on which computer executes it.

Execute the Processing: as noted above, a complete **NMRPipe** parallel processing scheme calls for a given shell−script to be executed on a list of computers; this is the purpose of **nmrShell**. As a convenience, **nmrShell** determines the computers to use via the same server list file used by **xyz2pipe**.

Synchronize the Computers: certain schemes may require waiting for all computers to finish one task before continuing with the next one; this is called *synchronization*. **NMRPipe** schemes can be synchronized using the **waitFor** program, which also uses the server list file.

The **NMRPipe** parallel schemes use a simple approach to parallel computing called *static load−balancing.* This means that the amount of data each computer will process is fixed ahead of time. Therefore, a processing scheme will not compensate during execution for possibly changing speeds of a given computer due to other tasks it is performing. In the **NMRPipe** parallel scheme, 3D or 4D data is divided between the computers of interest as a series of appropriate 2D planes, each of which can be processed independently. The number of planes processed by a given computer is based on its relative speed. Note that these planes are "conceptual" ones; they don't necessarily correspond with actual 2D file planes in the input.

According to this scheme, if two computers of equal speed were used to process the X−Axis of a 3D data set with 64 planes, the first computer could process planes 1 to 32 while the second computer processes planes 33 to 64. This partitioning of the data planes for parallel processing is performed automatically by **xyz2pipe**, once it is given a list of computers to use.

As an example, we'll consider building and executing a parallel scheme on two similar computers called "`spite`" and "`spun`", which are Sun Sparc 10 computers in the NIH NMR network. In our example, the data of interest is in a directory called "`/spite2/frankie`". The data directory is on a disk which belongs to the computer `spite`, and the network and file systems are configured so that the data is available *via the same directory name* on the computer `spun`.

First, we'll construct a server list file, called "`pipe.net`" in this example, which lists the computers of interest. Each line in the file lists the name of one computer, along with an arbitrary number indicating its relative speed compared to the other computers in the list; the higher the number, the proportionally faster the computer is assumed to be. So, our server list file `pipe.net`, which we create in the `/spite2/frankie` directory, consists of the following two lines:

```
spite 10
spun  10
```

Note that if we wanted to use additional computers for the parallel scheme, we would simply add their names as additional lines in the server file. If the computers listed are not similar, we could estimate their relative speeds by comparing the time needed to perform the same processing task (such as processing a 2D plane) on each computer.

Once the server list file is created, we can use it in the following one−pass processing scheme, which in this case is saved the the shell−script file `"xy.com"`. The input data is an HNCO FID of (X=1024)(Y=128)(Z=64) total points. Notice that the scheme uses the "**−par**" option of **xyz2pipe**, but it is *completely ordinary* in every other way:

```
#!/bin/csh

xyz2pipe −in fid/test%03d.fid −x −verb −par pipe.net \
| nmrPipe  −fn POLY −time                            \
| nmrPipe  −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5   \
| nmrPipe  −fn ZF                                    \
| nmrPipe  −fn FT                                    \
| nmrPipe  −fn PS −p0 43  −p1 0.0                    \
```

```
| nmrPipe  –fn EXT –left –di –sw                    \
| nmrPipe  –fn TP                                   \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 1 –c 1.0  \
| nmrPipe  –fn ZF                                   \
| nmrPipe  –fn FT                                   \
| nmrPipe  –fn PS –p0 –135 –p1 180 –di              \
| nmrPipe  –fn TP                                   \
| nmrPipe  –fn POLY –auto                           \
| pipe2xyz –out ft/test%03d.ft2 –y
```

The "**–par** `pipe.net`" option of **xyz2pipe** will cause the processing scheme to process only a sub–set of the input data, rather than the entire 3D FID. This option will also temporarily adjust the data parameters that are sent down the pipeline, so that at the end of the pipeline, **pipe2xyz** will store the sub–set result into the appropriate part of the overall output.

If this shell–script is executed on computer `spite`, it will process the first half of the data (XY planes 1 to 32), since `spite` is listed first in the server list file. Correspondingly, when *the same shell script* is executed on `spun`, it will process the second half of the data (XY planes 33 to 64), because `spun` is listed second in the server list. If this shell script is executed on any other computer, it will not process any of the data, since only `spite` and `spun` are listed in the server file `pipe.net`.

USING NMRSHELL TO EXECUTE A SCHEME

The parallel processing scheme above could be executed "manually", by logging in to both computers separately, and running the shell–script `xy.com` on each one. However, this manual procedure becomes cumbersome if more than two computers are involved. As a more convenient alternative, we can use **nmrShell**, since it will execute a shell–script on a given list of computers. For example, to execute script `xy.com` on the computers listed in `pipe.net`, we could use the following single command from the `/spite2/frankie` directory:

```
nmrShell –par pipe.net –sh xy.com
```

When we type in this command, if all goes well the terminal screen will be divided into sub–windows for each computer, and status messages will appear on the screen as each computer processes its portion of the data:

```
spite 0: –---------------------------------------|
| XYZ2Pipe Partition: Plane 1 to 32 of 64        |
| XY Plane 32 of 64                              |
|                                                |
|------------------------------------------------|
spun 0: –----------------------------------------|
| XYZ2Pipe Partition: Plane 33 to 64 of 64       |
| XY Plane 64 of 64                              |
|                                                |
|------------------------------------------------|
```

PARALLEL EFFICIENCY

In the test above, both computers working in parallel completed the processing task in about 300 sec, while computer `spite` working alone can complete the overall processing in about 570 sec. In the ideal case, N similar computers working together could the reduce the time required by a factor of N; in our test, with N = 2 computers, the ideal time would therefore be 570/2 sec = 285 sec. We can express the efficiency of the parallel scheme as the ratio of the ideal time to the actual time. So, for this test case, we had an efficiency (100%)(285 sec)/(300 sec), roughly 95%.

During the parallel processing, computer `spun` must use the network to read and write data stored on computer `spite`. This is generally more time–consuming than reading and writing data on "local" disks. Furthermore, the computers in the parallel scheme must access the same disk, and this contention for disk access can also cause delays.

In practice, it is the extra time required for network transmission of data, combined with possible contention for disk access, which determines the efficiency of **NMRPipe** parallel schemes. This means that the parallel scheme will only be efficient if the time required to access and transmit data over the network is small compared to the time actually spent processing the data.

As a consequence, processing schemes which include computationally intense functions (e.g. automated baseline correction or linear prediction) are more likely to profit from parallel processing than "less intense" processing schemes. It also means that adding more computers to a parallel scheme will not always be effective, since the increasing overhead of data access can eventually cancel the benefits of the additional processors.

SYNCHRONIZING COMPLETE PROCESSING SCHEMES

In the parallel scheme above, two computers were used to share the X−Axis and Y−Axis processing of a 3D data set. In a complete scheme, the Z−Axis should be processed as well. However, in our example, the Z−Axis cannot be processed until both computers have finished their portion of the X−Axis/Y−Axis processing, since the Z−Axis won't be complete until all the XY planes in the result have been created. Once both computers are finished with the XY processing, we can build a scheme to parallel process the Z−Axis just by adding the "**−par**" option to an ordinary scheme, for example:

```
xyz2pipe −in  ft/test%03d.ft2 −z −verb −par pipe.net \
| nmrPipe  −fn SP −off 0.5 −end 0.98 −pow 1 −c 0.5   \
| nmrPipe  −fn ZF                                     \
| nmrPipe  −fn FT                                     \
| nmrPipe  −fn PS −p0 0.0 −p1 0.0                     \
| pipe2xyz −out ft/test%03d.ft3 −z
```

In order to combine this Z−Axis scheme with the X−Axis/Y−Axis scheme, we must *synchronize* the overall script so that no computer will begin the Z−Axis processing until all computers have finished the X−Axis/Y−Axis processing. This can be done via the program **waitFor**. **waitFor** uses a crude but practical technique for synchronization, which works by creating or checking for the existence of text files which signal completion of a given task on a given computer.

For example, if the following command is executed on computer `spite`, two things will happen:

```
waitFor −par pipe.net −ext part1
```

First, a file called "`spite.0.part1`" will be created; the existence of this file implies that computer `spite` has completed the first part of some processing task. Second, the **waitFor** program will pause until text files corresponding to the other computers listed in `pipe.net` have also been created. So, in our example above, **waitFor** running on `spite` would pause until the file "`spun.0.part1`" was created.

If the same command is executed on computer **spun**, the complementary events will occur; the file "`spun.0.part1`" will be created, and **waitFor** will pause until file "`spite.0.part1`" exists. In this way, the actions of the computers executing the same **waitFor** command will be synchronized, since each computer will pause until the others have also executed the **waitFor** command.

So, we can create a complete, synchronized parallel processing scheme, just by inserting a **waitFor** command between the parts of the scheme which must be coordinated across computers. As a defensive measure, we'll insert a version of the **waitFor** command at the very start of the scheme, which will delete any synchronization files that may be left over from previous scripts (the **−init** option of **waitFor** deletes old synchronization files without pausing; we could also use the UNIX file delete command **rm**):

```
#!/bin/csh

waitFor –par pipe.net –ext part1 –init

xyz2pipe –in fid/test%03d.fid –x –verb –par pipe.net \
| nmrPipe  –fn POLY –time                             \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 2 –c 0.5   \
| nmrPipe  –fn ZF                                     \
| nmrPipe  –fn FT                                     \
| nmrPipe  –fn PS –p0 43  –p1 0.0                     \
| nmrPipe  –fn EXT –left –di –sw                      \
| nmrPipe  –fn TP                                     \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 1 –c 1.0   \
| nmrPipe  –fn ZF                                     \
| nmrPipe  –fn FT                                     \
| nmrPipe  –fn PS –p0 –135 –p1 180 –di               \
| nmrPipe  –fn TP                                     \
| nmrPipe  –fn POLY –auto                             \
| pipe2xyz –out ft/test%03d.ft2 –y

waitFor –par pipe.net –ext part1

xyz2pipe –in  ft/test%03d.ft2 –z –verb –par pipe.net \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 1 –c 0.5   \
| nmrPipe  –fn ZF                                     \
| nmrPipe  –fn FT                                     \
| nmrPipe  –fn PS –p0 0.0 –p1 0.0                     \
| pipe2xyz –out ft/test%03d.ft3 –z
```

As before, the scheme can be saved as a shell–script ("`xyz.com`" here), and executed in parallel by **nmrShell** to produce the output that follows:

```
nmrShell –par pipe.net –sh xyz.com

spite 0: ----------------------------------------|
| XYZ2Pipe Partition: Plane 1 to 32 of 64        |
| XY Plane 32 of 64                              |
| XYZ2Pipe Partition: Plane 1 to 64 of 128       |
| ZX Plane 64 of 128                             |
|------------------------------------------------|
spun 0: -----------------------------------------|
| XYZ2Pipe Partition: Plane 33 to 64 of 64       |
| XY Plane 64 of 64                              |
| XYZ2Pipe Partition: Plane 65 to 128 of 128     |
| ZX Plane 128 of 128                            |
|------------------------------------------------|
```

Note that this parallel version of a 3D script is not very different from an ordinary one; it includes only two extra commands ("`waitFor –init`" and "`waitFor`") and two additions of the option "`–par pipe.net`".

RESTRICTIONS

The parallel processing method has some important restrictions which should be noted:

In–Place Processing: parallel scripts **cannot** use in–place processing modes when processing the Z-Axis or A-Axis. This restriction may be lifted in future releases.

Z–Axis Transpose: parallel scripts **cannot** make use of the Z–Axis transpose function ZTP.

CUSTOM REMOTE COMMANDS

As mentioned in the introduction, **nmrShell** uses the UNIX commands **cd** and **sh** to execute a script on a remote machine. The default remote command assumes that the remote machines are single–processor computers, and that the data directory has the exact same name on all machines of interest. As a background to accommodate other situations, and to satisfy advanced UNIX Users, the low–level details of the command generating procedure are presented here.

nmrShell uses the following template to generate commands:

`rsh –n` *host* "cd *workingDir*; sh *scriptName cpuID"*

host: comes from a given entry in the server list file.

workingDir: is the directory where **nmrShell** is invoked, as determined by the **getwd(3)** library function. The "**–path**" option of **nmrShell** is used to specify a different directory.

scriptName: is specified via **nmrShell**'s "**–sh**" option.

cpuID: is a number used to distinguish between different processors on the same multi–processor machine. It is determined from the server list file, according to how many times the same host name appears. It is zero by default. Because of its position in the command line, the *cpuID* becomes an argument to the shell–script specified by *scriptName*.

As a specific example, **nmrShell** would actually use commands like the following to start remote scripts, via the **popen(3)** library function:

```
rsh –n spite "cd /spite2/frankie; sh xyz.com 0"
rsh –n spun  "cd /spite2/frankie; sh xyz.com 0"
```

Remote commands can be customized for each computer by adding information to the server list file; any text which follows the "*hostName hostSpeed*" data will be used in place of "**sh**" in the remote command for that machine. For example, if we wanted to accommodate different directory names on each remote computer, we could use the following server list file:

```
spite 10 cd /net1/frank;  sh
spun  10 cd /net/frankie; sh
```

along with the following **nmrShell** command:

```
nmrShell –path ./ –par pipe.net –sh xyz.com
```

In this case, **nmrShell** will generate the following remote commands instead of the default ones (note that the "`–path ./`" option is used to "neutralize" the first **cd** in the remote command):

```
rsh –n spite "cd ./; cd /net1/frank; sh xyz.com 0"
rsh –n spun  "cd ./; cd /net/frankie; sh xyz.com 0"
```

MULTI–PROCESSOR COMPUTERS

With the help of custom remote commands, the static load–balancing schemes described so far can also be used to perform and control parallel spectral processing on a multi–CPU computer. As an example, we'll consider a four–processor SGI Challenge, with the hostname `charybdis`. If an ordinary processing script is executed on this computer, the operating system itself will distribute the various instances of **nmrPipe** so that they automatically run on different processors; however this distribution may not be the ideal one. So, the goal of our scheme for this machine will to encourage an even distribution of work among the processors.

On the SGI Challenge, each CPU behaves as if it has the same name (`charybdis`) but a different *CPU ID* (0, 1, 2, or 3). So, schemes for this situation reflect the fact that we have to use a combination of both a name and an ID number to refer to a given CPU. In our scheme, we'll use the special SGI command "**runon**", which executes a command on a particular processor. It has the form:

`runon` *processorID command*

During processing, each CPU will be responsible for processing an equal sub–set of the data, as if it were an independent computer on the network. So, each CPU will have its own entry in the server list file, with the same name and speed as the others. Additionally, in order to have **nmrShell** execute the scheme on each processor, the server list file will be adjusted to include the appropriate **runon** command as a custom option:

```
charybdis 10 runon 0 csh
charybdis 10 runon 1 csh
charybdis 10 runon 2 csh
charybdis 10 runon 3 csh
```

Since the hostname `charybdis` appears more than once in the server list, **nmrShell** will automatically use different ID numbers as the last argument of its remote commands. So, the server file contents listed above would lead to remote commands like the following:

```
rsh –n charybdis "cd /data; runon 0 csh xyz.com 0"
rsh –n charybdis "cd /data; runon 1 csh xyz.com 1"
rsh –n charybdis "cd /data; runon 2 csh xyz.com 2"
rsh –n charybdis "cd /data; runon 3 csh xyz.com 3"
```

In each case, the shell–script "`xyz.com`" will be invoked with an ID number as its first argument, which is accessed as "`$1`" in the script. This argument will be used to tell the programs **waitFor** and **xyz2pipe** which processor they are running on. So, once the proper server file is created, we can change a parallel scheme for use on a multi–processor computer just by adding the option "`–cpu $1`" in the appropriate places:

```
#!/bin/csh

waitFor –par pipe.net –ext part1 –init –cpu $1

xyz2pipe –cpu $1 \
    –in fid/test%03d.fid –x –verb –par pipe.net      \
| nmrPipe  –fn POLY –time                            \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 2 –c 0.5   \
| nmrPipe  –fn ZF                                    \
| nmrPipe  –fn FT                                    \
| nmrPipe  –fn PS –p0 43  –p1 0.0                    \
| nmrPipe  –fn EXT –left –di –sw                     \
| nmrPipe  –fn TP                                    \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 1 –c 1.0   \
```

```
| nmrPipe  –fn ZF                                    \
| nmrPipe  –fn FT                                    \
| nmrPipe  –fn PS –p0 –135 –p1 180 –di               \
| nmrPipe  –fn TP                                    \
| nmrPipe  –fn POLY –auto                            \
| pipe2xyz –out ft/test%03d.ft2 –y

waitFor –par pipe.net –ext part1 –cpu $1

xyz2pipe –cpu $1 \
   –in  ft/test%03d.ft2 –z –verb –par pipe.net       \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 1 –c 0.5   \
| nmrPipe  –fn ZF                                    \
| nmrPipe  –fn FT                                    \
| nmrPipe  –fn PS –p0 0.0 –p1 0.0                    \
| pipe2xyz –out ft/test%03d.ft3 –z
```

Note that this same script can also be used on single–processor computers; in these cases, the script will be executed with argument of zero (that is, $1 will have the value 0).

OTHER CUSTOM COMMANDS

In both the default and custom examples so far, **nmrShell** always uses the remote shell command **rsh -n** to execute commands. Custom commands can override use of **rsh** by including the "`!`" character at the start of the custom command. This may be useful in situations were the remote shell command is different than **rsh -n**, or in cases where remote execution of commands is not really needed.

If a custom command in the server file begins with the "`!`" character, **nmrShell** will execute the command directly (without **rsh**), according to the following template:

command *scriptName workingDir host cpuID*

For example, this server list file replaces the default command sequence with the UNIX **echo** command:

```
spite 10 ! echo
spun  10 ! echo
```

According to the template, this server file would give rise to commands like the following, which **nmrShell** will execute via the **popen(3)** library function:

```
echo xyz.com /net/frankie spite 0
echo xyz.com /net/frankie spun 0
```

SUMMARY

The examples so far lead to this summary of the features of parallel **NMRPipe** scripts compared to ordinary ones:

1. Parallel scripts require a server list file, which lists the names and relative speeds of the computers of interest.

2. Parallel scripts will begin with "**waitFor –init" or with a UNIX rm** command to delete old synchronization files from previous tasks.

3. Parallel scripts will invoke **xyz2pipe** with the parallel server list option "**–par**".

4. Parallel scripts will use a **waitFor** command between each part of the overall scheme, to synchronize the computers before continuing with the next stage of the processing.

5. Parallel scripts will be executed simultaneously on a group of computers, either manually or by the **nmrShell** command.

6. Parallel scripts must be executed on all computers listed in the server file in order to generate a complete result.

7. Parallel schemes for multi−processor computers will use custom commands in the server list file. They will also use "**−cpu" arguments for xyz2pipe** and "**waitFor**".

8. Parallel scripts **will not** use the ZTP transpose function, or in−place processing for the Z−Axis or A−Axis.

CONDITIONS FOR PARALLEL PROCESSING

Some hints for establishing and verifying the conditions for **NMRPipe** parallel processing schemes are listed here. If possible, consult with your system manager before proceeding, especially if network security is an issue, or if you are unfamiliar with UNIX.

Hint 1. The rsh command must be the correct one: some computers have more than one command called **rsh**; for instance, older versions of the SGI implementation of UNIX have a "restricted shell" command as well as a "remote shell" command, both called **rsh**. Use your computer's manual pages to find out where the "remote shell" resides. Then, make sure your default version of "rsh" corresponds to "remote shell", and not something else. You can use the UNIX command "**which**" to find which command is currently being used:

```
which rsh
```

If the default version of **rsh** is the wrong one, you can adjust your `.cshrc` file to include an alias to the correct version, for example:

```
alias rsh  /usr/ucb/rsh
```

Hint 2. The remote shell must have permission to run: each computer maintains a general list of other hosts and users allowed to execute remote commands. Each user can also have a private list of hosts, in the file "`.rhosts`" in the user's home directory. The format of these authorization lists is described in the **rsh** manual pages. Our test cases made use of the "`.rhosts`" file, which contains trusted hostname/username pairs. In order to have remote commands work correctly on all machines, we had to list both the long and short versions of each hostname in "`.rhosts`", for example:

```
spite delaglio
spite.niddk.nih.gov delaglio
spun delaglio
spun.niddk.nih.gov delaglio
charybdis delaglio
charybdis.niddk.nih.gov delaglio
```

To test that **rsh** is working correctly, you can try sending a simple remote command to each computer, for instance the **ls** command to list files:

```
rsh spite ls
```

```
rsh spun ls
rsh charybdis ls
```

If authorization is not established correctly, **rsh** will respond with the message `Permission denied`. If the remote command fails in a different way (perhaps with a "`core dumped`" error message) the next hint may apply.

Hint 3. The remote login procedure can't be interactive: some users have log–in procedures which require information to be entered at the keyboard (for instance, terminal type). Since **nmrShell** uses a non-interactive procedure to execute remote commands, it will fail if the remote log–in requires interactive input. So, the log–in procedure must be adjusted to avoid asking for input when it is run non–interactively. For example, in a "`.login`" C–shell script, the following lines could be used to enclose interactive commands so that they are not executed by the remote shell:

```
if ($?prompt) then
   ...interactive commands...
endif
```

Hint 4. The programs must exist: the remote computers must have access to the correct versions of **nmrPipe** and the related programs. You can test this by executing the **which** command remotely on each machine, as in these commands and their corresponding output:

Command: `rsh spite which nmrPipe`
Output: `/u/delaglio/bin/nmrPipe`

Command: `rsh spun which nmrPipe`
Output: `/u/delaglio/bin/nmrPipe`

Command: `rsh charybdis which nmrPipe`
Output: `/u/delaglio/bin/sgi/nmrPipe`

If this doesn't work, verify that a version of **NMRPipe** software is installed on the remote computer, and that the remote log–in procedure includes the software in its search path.

Hint 5. The data and directories must exist: the data and directories must be available and accessible. This means that the data directories will have to be NFS–mounted on the network. In these cases, it is most convenient if a given directory is mounted with the exact same path name on every computer. If this is not the case, the server list file can be adjusted so that the directory to use for each remote computer is specified explicitly. You can confirm that data is available by trying to list it via remote shell. For instance, to confirm that data in directory "`/spite2/frankie`" is available on remote machines, commands like the following can be used:

```
rsh spite ls /spite2/frankie
rsh spun ls /spite2/frankie
rsh charybdis ls /spite2/frankie
```

Hint 6. Perform a simple test with nmrShell: before proceeding with an NMR processing scheme, try using **nmrShell** with a much simpler shell–script, just to confirm that it can execute remote commands correctly, configure the terminal screen, etc. A good choice would be a shell–script which lists files. We made a script called "`ls.com`":

```
#!/bin/csh
ls –la
```

and executed it this way:

```
nmrShell –par pipe.net –sh ls.com
```

If **nmrShell** fails to execute its shell–script, first try invoking it with an explicit path name; this may avoid any problems due to mismatches between actual and symbolic link names for directories on different machines:

```
nmrShell –par pipe.net –sh ls.com –path /spite2/frankie
```

If this doesn't work, and no useful diagnostic messages are produced, try adding the **–log** option, and then examine the resulting log files.

If the terminal doesn't seem to be configured correctly, check the terminal configuration and the TERM environment variable. You may have to use UNIX commands like **reset** or **resize** to configure the terminal properly. **Hint 7. Killing nmrShell may not kill remote processes:** the remote processes may continue running, or they may stay in a suspended state. You can try to kill the remote processes with the server list option of the **killShell** program, for example:

```
killShell xyz.com -par pipe.net
```

OPTIONS

–par *serverListName*

This required argument specifies the name of the server list file, which lists the hostnames of the computers of interest, one per line. The format of a line in the file is:

hostName relativeSpeed [remoteCommand]

Blank lines and lines beginning with the "#" are ignored. The *relativeSpeed* field of the entry is not used by **nmrShell**, but it must be present for consistency with other programs. The optional *remoteCommand* is used to override the default command used on the remote machine; this option can be used to accommodate different directory names on each on each computer, or to execute a command on a specific CPU of a multi–processor machine. If the *remoteCommand* argument begins with the "`!`" character, it will be executed directly, otherwise it will be executed via **rsh**.

–sh *scriptName*

This required argument specifies the name of the shell–script to be executed on the remote computers in the server list. The implies that the shell–script is available in the appropriate directory of every remote computer. It also implies that all data, programs, and commands required by the shell–script are available. If this shell–script is an **NMRPipe** processing scheme, it should be identified as a C–shell by including "`#!/bin/csh`" as the first line of the file.

–path *remotePath*

This option is used to override the default directory which will be used on the remote computers. By default, the directory name where **nmrShell** is invoked will be used as the directory name for the remote command.

–log When this option is included, status messages from each computer will be written to a corresponding text file as well as to the screen. The text file names have the form "*hostName.cpuID*.log", for example "`charybdis.2.log`". This option is useful for tracing problems with remote command execution.

–to *uSecDelay*

This option adjusts the time delay used to read and display status messages from the remote computers. It is intended primarily for development and testing.

The following is an example of 3D processing with Linear Prediction (LP) in two of the dimensions. It is suitable for both single–processor computers and multi–CPU machines. Note that the "`–cpu $1`" options can safely be left out if no multi–processor machines are involved.

```
#!/bin/csh

# xyz.lp.com: Four-Part Parallel Linear Prediction
#  Invoked by "nmrShell" with the Processor
#  ID Number as Argument 1 ($1).

# Remove files used to synchronize parts 1, 2, and 3:
#
waitFor –par pipe.net –init –ext part1 –cpu $1 –verb
waitFor –par pipe.net –init –ext part2 –cpu $1 –verb
waitFor –par pipe.net –init –ext part3 –cpu $1 –verb

# Execute Part 1 of the scheme: X–Axis Processing
# Wait for all computers to finish part 1.
#
xyz2pipe \
  –in fid/test%03d.fid –x –verb –par pipe.net –cpu $1 \
| nmrPipe  –fn SOL –head 1 –mir                         \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 2 –c 0.5  \
| nmrPipe  –fn ZF                                       \
| nmrPipe  –fn FT                                       \
| nmrPipe  –fn PS –p0 43  –p1 0.0                      \
| nmrPipe  –fn EXT –left –di –sw                        \
| pipe2xyz –out ft/test%03d.dat –x

waitFor –par pipe.net –ext part1 –cpu $1 –verb

# Execute Part 2 of the scheme: Z–Axis Processing
# Wait for all computers to finish part 2:
#
xyz2pipe \
   –in ft/test%03d.dat –z –verb –par pipe.net –cpu $1 \
| nmrPipe  –fn SP –off 0.5 –end 0.95 –pow 1 –c 0.5  \
| nmrPipe  –fn ZF –auto                                 \
| nmrPipe  –fn FT                                       \
| nmrPipe  –fn PS –p0 0.0 –p1 0.0 –di                   \
| pipe2xyz –out ft/test%03d.ft3 –z

waitFor –par pipe.net –ext part2 –cpu $1 –verb

# Execute Part 3 of the scheme: Y–Axis LP Processing
# Wait for all computers to finish part 3:
#
xyz2pipe \
   –in ft/test%03d.ft3 –y –verb –par pipe.net –cpu $1 \
| nmrPipe  –fn LP –fb                                   \
| nmrPipe  –fn SP –off 0.5 –end 0.98 –pow 1 –c 0.5  \
```

```
| nmrPipe  –fn ZF –auto                             \
| nmrPipe  –fn FT                                   \
| nmrPipe  –fn PS –p0 –135 –p1 180 –di              \
| pipe2xyz –out ft/test%03d.dat –y

waitFor –par pipe.net –ext part3 –cpu $1 –verb

# Execute Part 4 of the scheme: Z–Axis LP Re-Processing
#
xyz2pipe \
   –in ft/test%03d.dat –z –verb –par pipe.net –cpu $1 \
| nmrPipe  –fn HT –auto                             \
| nmrPipe  –fn PS –inv –hdr                         \
| nmrPipe  –fn FT –inv                              \
| nmrPipe  –fn ZF –inv                              \
| nmrPipe  –fn SP –inv –hdr                         \
| nmrPipe  –fn LP –fb                               \
| nmrPipe  –fn SP –off 0.5 –end 0.99 –pow 1 –c 0.5  \
| nmrPipe  –fn ZF –auto                             \
| nmrPipe  –fn FT                                   \
| nmrPipe  –fn PS –hdr –di                          \
| pipe2xyz –out ft/test%03d.ft3 –z
```

SEE ALSO

getwd(3), **killShell**(1), **nmrPipe**(1), **rsh**(1C), **which**(1), **popen**(3), **xyz2pipe**(1)

BUGS

nmrShell may not always configure or restore the terminal screen properly, especially if the terminal type environment is not established correctly.

The **nmrShell** terminal display is incorrect if the screen is too small to support enough sub–windows for each computer in the server list; at least four lines per computer are required.

Parallel processing options are not yet supported by conversion programs such as **bruk2pipe** and **var2pipe**.

Parallel schemes cannot make use of the **nmrPipe** in–memory 3D transpose function **ZTP.**.

DIAGNOSTICS

If remote commands fail, **nmrShell** may not be able to display diagnostic messages on the screen. In these cases, try using the **–log** option, and look for diagnostic messages in the log files instead.

LEGAL NOTICE