Bash Survival Guide for HPC

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

History of `bash`

The concept of the *shell* environment has been around for a long time (long = lifetime of `UNIX` operating system):

- `bash` is a twist on one of the earliest `UNIX` shells (the so-called Bourne shell, `sh` on most systems).
- `bash` is a superset of the (limited) Bourne shell syntax
- There is quite an extended family of shells, a good link for which can be found here:

  http://en.wikipedia.org/wiki/Comparison_of_command_shells

bash Features

Subset of important features of bash:

- Basically a command line interface (CLI) to the operating system
- Command line completion (via TAB key)
- Command line editing, history
- Output redirection
- Control constructs
- ...

Basically all of the shells give you the ability to enter text-based commands/queries much faster than using a GUI (if your particular OS even supports a GUI).

bash Invocation/Startup Scripts

bash will read from various startup scripts depending on how it was invoked:

- Interactive login shell: uses /etc/profile, then ~/.bash_profile, ~/.bash_login, ~/.profile in that order (if they are present at all).
- Interactive non-login shell: ~/.bashrc (suppressed by --norc option, or --rcfile file alternative)
- Non-interactive: uses $BASH_ENV (subject to substitution, if result is a file, executes the file)
- as sh: attempts to duplicate Bourne shell behavior, reads and executes /etc/profile and ~/.profile, in order
- via rshd: will read and execute ~/.bashrc (if it is able to determine that is was launched via rsh)

CLI Navigation

`bash` supports a rich set of line editing features making it easier for you to move the cursor around the command line, recall past commands and edit them, etc. The default syntax follows that of the **emacs** text editor (you can customize the syntax, of course, see **man bash**). Very briefly, the arrow keys can typically be used to navigate the current command line (left and right arrow keys, up and down to scroll though past commands), plus some common key bindings (there are many more than this small sample):

C-a	go to the beginning of the line
C-e	go to the end of the line
C-f,C-b	go forward,back one character
M-f,M-b	go forward,back one word

where **C-** denotes the **Control** key on the keyboard, and **M-** the **META** key - not many keyboards have **META** keys these days, in which case **ESC** is used instead.

CLI History

Unsurprisingly, the shell records a history of what commands you execute. The **history** command will list the commands in your shell's history with an identification number that you can use as shorthand for recalling them:

```
[rush:~]$ !!                # repeat very last command
ls -l
[rush:~]$ history           # spits out last HISTSIZE commands
[rush:~]$ !1574             # pick one by number, preface with !
who am i
jonesm   pts/23       2012-05-25 12:53 (cash.ccr.buffalo.edu)
[rush:~]$ !1562:s/mix/idle/   # pick one, use a regex to modify on the fly
snodes all general-compute idle| less
```

Environment Variables

Environment variables are heavily used by all shells to set the user environment and control its behavior, and are also frequently used by applications. For now, here are just a few of the simplest of the defaults:

$HOME , your home directory location.

$PATH , search path for commands

$LD_LIBRARY_PATH , search path for dynamic libraries

$MANPATH , search path for `man` pages

$PS1 , command prompt syntax

Note that the **modules** environment package used in CCR is designed to allow you to more easily manipulate the environment (especially $PATH, $LD_LIBRARY_PATH, and $MANPATH) for various software packages.

You can set environment variables very simply by assignment or using the `export` command, the difference being whether the value is passed on to sub-shells (i.e. if you run another script or executable) - you can also run a program with specific settings using the `env` command:

```
# N.B., these variables are set only in the current shell
srun -l hostname -s | sort -n | awk '{print $2}'> $MY_NODEFILE
NP=`cat $MY_NODEFILE | wc -l`            # counts lines in $MY_NODEFILE
NODES=`cat $MY_NODEFILE | uniq`          # counts unique lines in $MY_NODEFILE
# this applies to all children of this shell
#  (Intel MKL thread core affinity setting)
export KMP_AFFINITY=nowarnings,compact
# this runs a shell script using a local setting for $HOME
#   (MATLAB compiled binaries like to abuse $HOME directories ...)
env HOME=/tmp ./run_extract.sh
```

You can add common settings to your $HOME/.bashrc file if you want them to apply to all of your logins/shells. Using `export` without any arguments will print the current settings for your shell (as will the `env` command).

The First Line

Shell scripts can be no more than an ordered list of commands to run. By convention, the first line declares the shell:

```
#!/bin/bash
#
# pound symbol generally starts a comment, except for that first line
echo 'Hello, world!'
```

Variables

Variables in `bash` are easily declared:

```
# some variable declarations:
str1="Goodbye, world!"  # note quotes to protect the space
str2=`date`             # backticks to execute commands,
                        #   capture the output
echo -n '$USER ='       # single quotes do not exand $USER
                        #   and -n does not break line
echo "$USER"            # double quotes will expand $USER
# preceding two lines print out $USER =johndoe for userid johndoe
# there are a bunch of predefined variables, $USER is one
echo "${str1}Aargh!"    # braces are handy to protect variables
str3=""
if [ -n "$str3" ]; then # -n indicates non-empty argument
  echo '$str3 is not empty!'
fi
```

That last bit is a good example of why variables are often enclosed in quotes (they do not have to be), in case they evaluate as the empty string.

Command Substitution

Command substitution is an invaluable way to access the output of a command from the environment. You can do this either with backticks, `'command'`, or subshells, `$(commands)`:

```
files="$(ls)"                           # listing of cwd
html_files=`ls /var/www/html`           # listing of /var/www/html
index=`expr 4 * 2 + 1`                  # use expr to evaluate math
```

Note that `$()` is very easy to nest, which gives it an edge over backticks. Newlines are not allowed, so they are stripped out in both methods.

Arithmetic

`bash` has no notion of floating-point math, although you can resort to external tools like `bc` to work around that lack. Integer arithmetic is done using backticks, double parentheses, or `let`:

```
z=`expr $z + 1` # expr command does the work
z=$(($z+1))
z=$((z+1))       # parameter de-referencing optional inside double parentheses
(( z += 1 ))     # shortcut
let z=z+1        # let command
let "z += 1"     # quotes let you use spaces
```

I/O Redirection

Very handy topic - how to redirect input/output in `bash`. First, though, there are three default “files”:

- **stdin** (keyboard), descriptor 0
- **stdout** (screen), descriptor 1
- **stderr** (error messages to the screen), descriptor 2

Redirection just means capturing output from a file, command, or script (or portion of a script) and sending it elsewhere as input. Note that descriptors 3-9 are also available for use.

Generally you can redirect input with <, and output with | (pipe) or > (file), but there are quite a few more variations best illustrated by example:

```
echo 'blah' > filename          # overwrites filename
echo 'blah' >> filename         # appends filename
./somecode 1> filename          # filename capture stdout from somecode
./somecode 2> filename          # filename captures stderr from somecode
./somecode &> filename          # filename captures both stdout and stderr
./somecode > filename 2>&1      #  same as above, old (Bourne) style
./othercode < filename          # othercode gets input from filename
./code < inname &> outname      # combinations are ok
command1 | command2 | command3  # pipes are similar to > but more general
                                #  for chaining commands together
cat *.txt | sort | uniq > result # see?
```

if--fi

bash has a conditional construct, of course:

```
if conditionA
then
   statement0
   statement1
   ...
elif conditionB
then
   morestatements1
   morestatements2
   ...
else
   yetmore0
   yetmore1
   ...
fi
```

We will go into the sytnax for the conditions next, but note that the conditional branches are often written more concisely using the ";" separator:

```
if conditionA; then
   listofstatements
...
fi
```

Is There in Truth No Beauty?

The test command (and its shorthand version) gets used quite often in conditionals (and in general):

```
# first form
test operand1 operator operand2
#  example:
if test -n "$SLURM_JOBID"; then
  echo "Hey, I am in a Slurm environment!"
fi
# second form
[ operand1 operator operand2 ]
# example:
if [ -n "$SLURM_JOBID" ]; then
  echo "Hey, I am still in a Slurm environment!"
fi
# I cheated a bit, -n is a unary operator ...
```

Operators

I am not going to show the full set of test operators (see `man bash`), but here is a quick subset:

Operator	Operands	
-n	1	nonzero length
-z	1	zero length
-f	1	file given by operand exists
-d	1	directory given by operand exists
==	2	string equality
!=	2	
<,>	2	string lexical
-eq,-ne	2	integer (in)equality
-lt,-gt	2	integer comparisons
-le,-ge	2	integer comparisons
||	2	logical OR
&&	2	logical AND

Note the absence of floating-point operations (serious shortcoming).

`for` Loops

First of the main loop types in `bash`, `for` loops repeat over a simple index of space separated items:

```
for name in "$LIST"
do
  echo "working on $name"
  bunchofstatements1
done
```

So the loop counter, `name` is set to each item in the list as the loop is executed. Note that the familiar form from C for integer indices is also valid:

```
for (( expr1; expr2; expr3 )); do listofstatements ; done
# for example:
for ((i=1; i<10; i++))
do
  echo "i = $i"
done
```

Note that `bash` is pretty slow (most interpreted languages are slow), so the C-like loop syntax is not encouraged.

Globbing

You can make handy use of substitution by “globbing,” or using * to match all filenames.

```
# simple globbing example
for fname in *.c          # grabs all .c files in current directory
do
  grep 'double' $fname   # pull all out all the doubles
done
```

while Loops

While loops are just about what you would expect:

```
while list; do listofstatements; done
# simple example
while [ "$counter" -lt "$UPPER_LIMIT" ] ; do
   echo "counter = $counter"
   counter=`expr $counter + 1`  # counter=$(($counter+1)) should also work
                                # let "counter += 1"       should also work
done
```

There is also an `until` variant of `while` that has the same syntax (just negates the test).

Conditional Execution

A very typical `bash`-ism is conditional execution, namely use the return code from a command (you know that all commands have return codes to indicate success or failure, right?):

```
command1 && command2             # command2 iff command1 returns true (0)
command1 || command2             # command2 iff command1 returns false(non-zero)
command1 && command2 || command3 # command2 iff command1 returns true
                                 #  command3 iff command1 returns false
#
rm -f $thisfile && echo "$thisfile was removed." || echo "$thisfile not removed."
#
# can check exit status of last command using $?
echo $?
```

Functions

bash has support for functions, general syntax:

```
#
# general function syntax
# one way:
function_name {                # body of function inside braces
  statements;                  #  should end with semicolon or newline
}
#
# another way:
function functionname ()       # using the function builtin requires ()
{
  statement1
  statement2
  ...
  statementN
  return                       # or an explicit return builtin
#
# handy function for DOS/windows shell users
dir ()
{
   ls -F --color=auto -lF --color=always "$@" | less -r   # $@ is $1, $2, ...
}

}
```

Return code is the the return code from the last statement in the function.

Function Parameters

You can pass arguments to functions (they behave like mini-scripts). They become positional parameters that you can reference as $1, $2, ...

```
#
# fun with function (or script itself) parameters
#
echo "My name is $0"            # name of the script
echo "I have $# arguments"      # number of arguments is $#
if [ -n "$1" ]
then
  echo "First argument is $1"   # 1st argument
fi
#
# An often used form for scripts to check argument number
#
if [ ! $# -eq 2 ]; then
  echo "Usage: $0 arg1 arg2"
  exit 1
fi
```

Note that $0 remains unaffected by calling a function.

Bit More Function Trivia

Functions can be recursive, and can have their own local variables through the **local** builtin.

Shell Limits

`bash` shell limits are controlled through the **ulimit** builtin function. Easiest way to see that is by example:

```
[rush:~]$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 2066355
max locked memory       (kbytes, -l) 33554432
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) unlimited
cpu time               (seconds, -t) 900
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

The values can be numerical or one of **hard, soft** or **unlimited**. The flags `-H, -S` can be used to specify the hard and soft limits. Note that the stack limit is of particular interest to us in HPC, and very frequently needs to be increased from its (small) default value.

Aliases

Aliases are basically keyboard shortcuts that you can set up and use - they do not do any expansion, nor can they be recursive:

```
#
# from my ~/.bashrc
#
export EDITOR=vi
#
# Aliases
#
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
alias xe='xemacs'
alias myps='ps -u jonesm -Lf'
alias myjobs='qstat -a | grep jonesm'
alias proj='cd /projects/jonesm'
alias wien='cd /projects/jonesm/d_wien/wien2k/u2'
alias mod='module'
alias ml='module load'
alias mu='module unload'
alias mls='module list'
```

Loading Other Scripts

You can load other `bash` scripts (or just files containing `bash` commands) using the **source** command.

```
#
# loads Intel compiler environment
. /opt/intel/composerxe/bin/compilervars.sh intel64
```

Sometimes you will see the shorthand **.** used instead of **source** (same meaning).

Regular Expressions

Regular expressions (REs) occur very frequently in UNIX, especially in sed, awk, and grep. If you have done any significant work in Perl, of course, you are likely also well versed in REs.
REs are sets of (meta)characters used for pattern matching in text searches and string manipulation, and contain one (or more) of:

- **characters** - retain their literal meaning
- **anchors** - designate position in the text line, ^ (beginning) and $ (end)
- **modifiers** - expand or narrow the range of text to search, includes , brackets, and the backslash

More on modifiers in REs:

- `*`, matches any number of the character string or RE preceding it
- `.`, ("dot") matches any non-newline character
- `^`, ("carat") matches beginning of line, but also for negation (context dependent)
- `$`, matches end of line
- `[,]`, brackets enclose a set of characters to match, including negation with `^` and ranges with -
- `\`, backslash escapes a special character (e.g. $)
- `\<,\>`, escaped angle brackets match word boundaries
- ?, question mark matches zero or one of the preceding RE, used mostly for single characters
- +, matches one or more of the previous RE (like `*`, but does not match zero occurrences)
- `\{,\}`, escaped curly braces indicate number of previous REs to match
- (), parentheses enclose a group of REs
- `|`, the "or" RE operator used in matching alternates

RE Examples

Used in the context of `grep` just as an example:

```
grep "2224*" textfile             # matches 222 followed by any number of 4s
                                  #   e.g. 222, 2224, 22244, ...
grep "24." textfile               # matches 24 followed by any character, but not 24
grep "^A" textfile                # matches any line beginning with A
grep "A$" textfile                # matches any line ending with A
grep "^$" textfile                # matches blank lines
grep "[abc123]" textfile          # matches any of the character a,b,c,1,2,3
grep "[a-c1-3]" textfile          # same thing using ranges
grep "[^a-c]" textfile            # matches all characters except a-c
grep "[Yy][Ee][Ss]" textfile      # case insensitive match for YES, yes, Yes, ...
grep "\$\$" textfile              # match $$
grep "\<yes\>" textfile           # matches only distinct word yes
```

REs are a rich topic in their own right ...

Here Documents

So-called “here documents” are ways to pass text on-the-fly into programs or other code blocks. It’s another form of I/O redirection, but a very handy one for running programs:

```
./a.out<<ENDOFINPUT   # nothing special about this string, just needs to be unique
inputline1            #  and matched below.  These lines appear as standard input
inputline2            #  to the program a.out
...
ENDOFINPUT
#
# you can prevent substitution by quoting the string
#
./a.out<<'EOI'
line that needs $input that contains $
EOI
cat <<'EOF'>newscript.sh  # very handy to automate script generation
#!\bin/bash
#
echo "$0 $@"
...
EOF
```

Basic Job/Process Control

You have the ability to stop and continue your processes from a fairly low level to more sophisticated tools like `screen`. The basics are controlled through the notion of a **job**, created by putting a process asynchronously into the background using the & operator, or **C-z** (control-z), the `bg` and `fg` commands place the process into background or foreground, `jobs` will list all jobs:

```
[rush:~]$ emacs test.c
^Z                             # control-z suspends job, jobid given in square brackets
[1]+  Stopped                  emacs test.c
[rush:~]$ bg                   # put into background, now running separately from terminal
[1]+ emacs test.c &
[rush:~]$ jobs                 # lists running jobs
[1]+  Running                  emacs test.c &
[rush:~]$ fg
emacs test.c                   # bring back to foreground in terminal
[rush:~]$ kill %1              # die job 1, die - note % for jobid
```

`kill` terminates jobs and processes (no % sign for process ids), if you have a stuck process (identified with the `ps` commands) you may need to specify a higher kill level like `kill -9 pid`.

Stuff Not Covered Thus Far

In the context of a `bash` "survival guide," I skipped quite a bit of stuff:

- Argument processing (**getopts, shift**) ...
- List-oriented data structures
- Built-in functions (only talked about a small subset)

More Resources on `bash`

- Man page for your particular `bash` (definitive for your version): man bash
- `bash` Reference Manual: `http://www.gnu.org/software/bash/manual`
- Beginners guide to `bash` (LDP): `http://tldp.org/LDP/Bash-Beginners-Guide`
- Advanced `bash` Scripting (LDP): `http://tldp.org/LDP/abs`

Viewers

You do not need (and often do not want) to use an editor just to look at a file (or a long listing of results, files in a directory, etc.), so Linux/UNIX provides viewing utilities that you can use:

- `head` - prints the first N (default 10) lines of a file, e.g., `head -30 filename`
- `tail` - prints the last N (default 10) lines of a file, e.g., `tail -30 filename`
- `cat` - **cat**enates a file, i.e. just spits it out to standard output (**-n** shows line numbers, **-t** shows non- printing characters and tabs)
- `more` - file filter, with pagination commands like **vi**
- `less` - alternative to `more` intended to be faster (does not attempt to parse the whole file) and with more freedom of movement. Cf. **man less**

sed

sed, the **s**tream **ed**itor, is one of the oldest of the UNIX editors (and therefore probably the least user-friendly). It has a lot of power, though, and you will still find it heavily utilized in shell scripts. Some very simple examples:

```
[rush:~]$ echo 'Hello, world!'      # self explanatory (note single quotes)
                                   #
[rush:~]$ echo 'Hello, world!' | sed s/Hello/Goodbye/
Goodbye, world!
[rush:~]$ echo 'Hello, world!' | sed "s/Hello/Goodbye, cruel/"
Goodbye, cruel, world!             # note quote usage
```

grep

`grep` is a handy little utility to perform searches on files for lines containing a given string (the origin of the name dates back to UNIX's original command editor, `ed`, namely `g/re/p`.

Simple examples:

```
[rush:~]$ w > tmp.w            # dump output of w to a temporary file, cf. 'man w'
[rush:~]$ grep $USER tmp.w     # my current logins to this machine
[rush:~]$ grep joeuser tmp.w   # joeuser's logins to this machine
[rush:~]$ w | grep $USER       # we don't need no stinkin' temporary files ...
```

`man grep` for handy options (notable, **-i** and **-n**).

awk

`awk` is a very powerful utility - it harbors a complex programming environment in its own right, but you are free to use only as much of it as you want or need to at a given time. The general syntax for `awk` looks like:

```
awk '/pattern/ {action}' file
```

a given action will be performed on every record which matches the given `/pattern/`. A very handy list of the fields in the record are given as variables, $0 (entire record), $1 (first field), $2 (second field) .... Let's try an example:

```
[rush:~]$ squeue > tmp.q                       # record current state of the queues
[rush:~]$ grep xdtas tmp.q                     # our friend grep
[rush:~]$ awk '/xdtas/ {print $0}' tmp.q       # same thing with awk
                                               # more interesting example
[rush:~]$ awk 'BEGIN {nodes=0} /xdtas/ {nodes += $7} END {print nodes}' tmp.q
                                               # add cores, running/pending jobs?
```

Exercise

Modify the preceding `awk` example to also count and report the number of jobs and cores as well as nodes.

find

`find` is quite a handy command for searching for files in complex directory layouts - see **man find** for various options, the very basic syntax looks like:

```
find [path] [expression]
```

where the expression involves options for just about every possible file property that you might imagine - here are a few examples:

```
# find all files herein with suffix .nc
[rush:/scratch/jonesm]$ find . -name \*.nc -print
# narrow the search down a bit
[rush:/scratch/jonesm]$ find . -name 2010-\*.nc -print
# long list each one
[rush:/scratch/jonesm]$ find . -name 2010-\*.nc -exec ls -l {} \;
# show me files in danger of being scrubbed from my panasas scratch directory
[rush:~]$ find /panasas/scratch/jonesm -atime +23
```

Note that the `xargs` command is also helpful when you want to do something useful with copious amounts of output from a command like `find`.

Other utilities

There is a zoo of other useful commands/utilities, here is a sampling based on what I have found to be useful:

`file` determines a file's type, typically text, data, or executable, but can also identify other common file types (e.g., it will recognize windows file types).

`ldd` prints shared library dependencies, useful for checking to see that all dependencies are satisfied.

`cut` slices files in a similar fashion to `awk`, but does so in a less complex way. You can specify bytes (**-b**), delimiters (**-d**), character position (**-c**), or fields (**-f**) from which to extract.

`paste` merges lines from files (or standard input).

`join` more flexible version of `paste` for sources containing common fields.

`sort` handy sorting utility (lexical, **-n** numeric, etc.).

`split` splits files into pieces, by lines (**-l**) or by bytes (**-b**) or a maximum of some bytes per line (**-C**).

`tr` translates or deletes characters, very useful for rapidly cleaning up files (common example, `tr -d '\r' < infile.csv > outfile.csv`, to remove windows carriage returns from a file)

`uniq` omit (or report) repeated lines.

`wc` newline (**-l**), word (**-w**), character (**-m**), or byte (**-c**) counts.

More CLI Resources

- List of CLI related "cheat sheets:"

  http://www.cyberciti.biz/tips/linux-unix-commands-cheat-sheets.html

- The Linux documentation project:

  http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/GNU-Linux-Tools-Summary.html

Simple Tool Script

Shell scripts are quite commonly used when deploying other tools - here is an example of one that is used to wrap a Java application called Panoply[1]

```
[rush:~]$ ls panoply/PanoplyJ
colorbars  jars  overlays  Panoply.jar  panoply.sh  README.txt
[rush:~]$ cat panoply/PanoplyJ/panoply.sh
#!/bin/sh
#

SCRIPT=`readlink -f $0`
SCRIPTDIR=`dirname $SCRIPT`

java -Xmx1G -Xms128m -jar $SCRIPTDIR/jars/Panoply.jar "$@"
```

[1] http://www.giss.nasa.gov/tools/panoply

Example: Slurm `bash` Script

```
#!/bin/bash
#SBATCH --nodes=10
#SBATCH --ntasks-per-node=16
#SBATCH --constraint=CPU-E5-2660
#SBATCH --time=72:00:00
#SBATCH --mail-type=END
#SBATCH --mail-user=jonesm@buffalo.edu
#SBATCH --output=slurmWIEN.out
#SBATCH --job-name=Gd1212-sup222
#
# hybrid mpi/openmp capable version
# (still do not recommend using OpenMP, though)
#
#module use /projects/jonesm/modulefiles
module load wien2k/2k.12.1b
module list
```

```
# Use local /scratch whenever possible - NOTE:
#  this requires that the number of k-points breaks
#  down evenly over the number of cores, if not, they
#  need a shared $SCRATCH
#
export SCRATCH=$SLURMTMPDIR
#export SCRATCH=/panasas/scratch/jonesm/nPuIn3AFM
export | grep SLURM
export | grep WIENROOT
export | grep WIEN_DMFT_ROOT
echo "Allocated Nodes:"
export PBS_NODEFILE=tmp.pbsnodes
srun -l hostname -s | sort -n | awk '{print $2}'> $PBS_NODEFILE
cat $PBS_NODEFILE
ALLCORES=`cat $PBS_NODEFILE`
NNODES=`cat $PBS_NODEFILE | sort | uniq | wc -l`
NCORES=`cat $PBS_NODEFILE | wc -l`
```

```
#
# decomposition - for pure k-point parallelism, set NMPI_PERNODE=0
#
CASE=${PWD##*/} # grabs current directory name through parameter expansion
NK=`wc -l $CASE.klist | awk '{print $1}'`
let NK-=2
NODESPERK=`echo "$NNODES/$NK" |bc`
NMPI_PERNODE=16
NCPUSPERNODE=`cat /proc/cpuinfo | grep processor  | wc -l`
if [ $NMPI_PERNODE -gt 0 ]; then
   NOMP_PERTASK=`echo $NCPUSPERNODE/$NMPI_PERNODE |bc`
else
   NOMP_PERTASK=1
fi
NMPI=`echo ${NNODES}*${NMPI_PERNODE} |bc`
echo "Number of k-points: $NK"
echo "Nodes per k-point: $NODESPERK"
echo "MPI tasks/node = $NMPI_PERNODE"
echo "Total MPI tasks = $NMPI"
echo "OpenMP threads/task = $NOMP_PERTASK"
export OMP_NUM_THREADS=$NOMP_PERTASK
#export OMP_NUM_THREADS=1
export I_MPI_DEBUG=4
export I_MPI_PIN_DOMAIN=omp
export KMP_AFFINITY=verbose,compact
```

```
# .machines file
[ -e .machines ] && \rm .machines
if [ $NMPI -ge 1 ]; then
   echo -n "lapw0:" > .machines
   for str in `head -1 $PBS_NODEFILE`; do  # lapw0 likes to fail on > 1node
     echo -n "$str:$NMPI_PERNODE  " >> .machines
   done
   echo >> .machines
   let k_cntr=0
   for str in `cat $PBS_NODEFILE | sort | uniq`; do
     #echo "str = "$str
     #echo "k_cntr = "$k_cntr
     let k_rem=`echo "$k_cntr%$NODESPERK" | bc`
     #echo "k_rem = "$k_rem
     if [ $k_rem == 0 ]; then
        echo -n "1:" >> .machines
     fi
     echo -n "$str:$NMPI_PERNODE " >> .machines
       let k_cntr++
     if [ $k_cntr == $NODESPERK ]; then # start a new k-point entry
        echo " " >>.machines
        let k_cntr=0
     fi
   done
elif [ $NOMP_PERTASK -gt 1 ]; then
   cat $PBS_NODEFILE | uniq | awk '{printf "1:%s\n", $1}' > .machines
else # pure k-point run, 1 thread per k-point
   cat $PBS_NODEFILE | awk '{printf "1:%s\n", $1}' > .machines
fi
echo "granularity:1" >> .machines
echo "extrafine:1" >> .machines
runsp_lapw -so -cc 0.0001 -i 40 -NI -p
```

```
[rush:/projects/jonesm/d_wien/wien2k/u2/d_coffey/Gd1212-sup222]$ cat .machines
lapw0:f16n11:16
1:f16n11:16
1:f16n13:16
1:f16n16:16
1:f16n28:16
1:f16n32:16
1:f16n33:16
1:f16n34:16
1:f16n35:16
1:f16n36:16
1:f16n37:16
granularity:1
extrafine:1
```