Parallel Rsync (my way)

rsync is sooo cool, chances are, if you need to copy some files for whatever reason from one linux machine to another or even from one directory to another, rsync has everything you need. one thing though is terribly missing: parallelism

when you copy files with rsync you often see an io performance (by using iotop for example) that is far below what your disks or your network connection are capable of.

1. when you copy a directory containing many small files locally, rsync is slowed down by all the metadata operations it does (copying all the permissions, checking each file for changes by checking file dates etc.)
2. when you copy files across a network, you are slowed down by a single threaded ssh process which can only use one cpu core for encrypting and decrypting data on that connection.

my solution to that: run multiple rsync processes in parallel and leverage the power of several cpu cores in parallel.

here is a bash script function I wrote which i can use in scripts to copy files from A to B through multiple connections.

Use this at your own risk If you are interested in understanding what it does (and i strongly suggest you get interested in that before using this blindly!) you can read through my ideas on how to ideally parallelize rsync below the script.

the bash function

what this function does is as follows: it runs a find across the source directory and gets a list of all files and directories within it. it then extracts a list of directories with a maximum directory depth of $3 (the 3rd argument to the function). it then queues these directories to sync them and runs $4 (the fourth argument) rsync processes in parallel to do so using xargs.

once all directories have been synced it runs a single rsync thread as you would to just simply copy the files single threaded. only now we already copied all the files. this step is more some sort of a safety measure to make sure we really copied everything and that all file attributes are correct.

once this step is passed it runs another sanity check and compares md5 sums between all source and target files. this might take very long and is not really necessary but since i programmed this function for an archive script that will copy files to an archive before they are deleted from the source i wanted to be 100% sure everything went okay :)

how many jobs should run in parallel and how many directories deep you want to parallellize your jobs really depends on your sepcific situation. if you have several terabytes of data and you do a complete sync it makes sense to dive deeper into the structure than when you just want to update an already existing copy of the same data, in that case it might be faster to only dive 1 to 2 levels deep into your structure or even not use this script at all, when most of the time is spend by “creating incremental file list”. really, read what's behind the script further down to understand how to parametrize it and how to modify it to adjust it to your specific situation in the second version I have added the possibility to optionally pass a 5th and 6th argument. A filename can be passed as $5. If the file does not exist, the initial directory list which resulted from the `find` call at the begining of the script is saved to it. If the file exists, prsync will read the contents of the file and use it as directory list instead of re-running the whole `find` operation. a second filename can be passed optionally as $6. prsync will save its progress to that file. if prsync is re-run, this file will be checked before the start of each rsync progress. in case the directory that was supposed to be rsynced is already on the list, it will be skipped. this can prevent re-running rsync for a large number of already synced directories to speed up resuming after an interrupted previous prsync run.

these two optional options should only be used if the source does not change between prsync runs. It is specially beneficial if the source storage in unstable and may crash after a certain period of time. using these two files will help to prevent unnecessary file scanning and comparing when resuming the prsync operation after a crash and hence help to advance the progress faster by minimizing unnecessary load on the storage.

the code

prsync.sh

```
#
# Parallel Rsync function 2020 by Pascal Suter @ DALCO AG, Switzerland
# documentation and explanation at
http://wiki.psuter.ch/doku.php?id=parallel_rsync
#
# version 1: initial release in 2017
# version 2: May 2020, removed the need to escape filenames by using
#            null delimiter + xargs to run commands such as mkdir and
rsync,
#            added ability to resume without rescanning (argument $5)
and to skip
#            already synced directories (argument $6)
#

psync() {
    # $1 = source
    # $2 = destination
    # $3 = dirdepth
    # $4 = numjobs
    # $5 = dirlist file (optional) --> will allow to resume without re-
scanning the entire directory structure
        # $6 = progress log file (optional) --> will allow to skip
previously synced directory when resuming with a dirlist file
    source=$1
    destination=$2
    depth=$3
    threads=$4
    dirlistfile=$5
    progressfile=$6
    # gets directory listing form remote or local using ssh and find
    dirlist(){
        #$1 = path, $2 = maxdepth
        path=$1
        echo "$path" | grep -P "^[^@]*@[^:]*:" > /dev/null
        if [ $? -eq 0 ]; then
            remote=`echo "$path" | awk -F : '{print $1}'`
            remotepath=${path:$((${#remote}+1))}
            ssh $remote "find $remotepath/./ -maxdepth $2 -type d |
perl -pe 's|^.*?/\./|\1|'"
        else
            find $1/./ -maxdepth $2 -type d | perl -pe 's|^.*?/\./|\1|'
        fi
    }

    # get a sorted list of md5sums of all files in a directory (remote
via ssh or local)
    md5list(){
        #$1 = path
        path=$1
        echo "$path" | grep -P "^[^@]*@[^:]*:" > /dev/null
        if [ $? -eq 0 ]; then
            remote=`echo "$path" | awk -F : '{print $1}'`
            remotepath=${path:$((${#remote}+1))}
            ssh $remote "cd $remotepath; find -type f -print0 | xargs
-0 -P $threads -n 1 md5sum | sort -k 2"
        else
            cd $path; find -type f -print0 | xargs -0 -P $threads -n 1
md5sum | sort -k 2
        fi
    }

    # generate a list of directories to sync
    if [ -z "$dirlistfile" ]; then
        rawfilelist=$(dirlist $source $depth)
    else
        # dirlist filename was passed check if it exists and load
dirlist from there, otherwise create it and save the dirlist to the
file
        if [ -f $dirlistfile ]; then
            rawfilelist=$(<$dirlistfile)
        else
            rawfilelist=$(dirlist $source $depth | tee $dirlistfile)
        fi
    fi

    # separate paths less than DIRDEPTH deep from the others, so that
only the "leafs" get rsynced recursively, the rest is synced without
recursion
    i=$(($depth - 1))
    parentlist=`echo "$rawfilelist" | sed -e '/^\(.*\/\)\{'$i'\}.*$/d'`
    filelist=`echo "$rawfilelist" | sed -e '/^\(.*\/\)\{'$i'\}.*$/!d'`

    # create target directory:
    path=$destination
    echo "$path" | grep -P "^[^@]*@[^:]*:" > /dev/null
    if [ $? -eq 0 ]; then
        remote=`echo "$path" | awk -F : '{print $1}'`
        remotepath=${path:$((${#remote}+1))}
        echo -n -e "$remotepath\0" | ssh $remote "xargs -0 mkdir -p"
    else
        echo -n -e "$path\0" | xargs -0 mkdir -p
    fi

    #sync parents first
    echo
"======================================================================
====="
    echo "Sync parents"
    echo
"======================================================================
====="
    function PRS_syncParents(){
        source=$2
        destination=$3
        progressfile=$4
        if [ -n "$progressfile" ] && grep -q -x -F "$1" $progressfile ;
then
            echo "skipping $1 because it was synced before according to
$progressfile"
        else
            echo -n -e "$1\0" | xargs -0 -I PPP rsync -aHvx --numeric-
ids --relative -f '- PPP/*/' $source/./'PPP'/ $destination/
2>/tmp/debug
            status=$?
            if [ -n "$progressfile" ]; then
                echo "$1" >> "$progressfile"
            fi
            return $status
        fi
    }
    export -f PRS_syncParents
    echo "$parentlist" | tr \\n \\0 | xargs -0 -P $threads -I PPP
/bin/bash -c 'PRS_syncParents "$@"' _ PPP "$source" "$destination"
"$progressfile"
    status=$?
    if [ $status -gt 0 ]; then
        cat /tmp/debug
        rm /tmp/debug
        echo "ERROR ($status): the was an error when syncing the parent
directories, check messages and try again"
        return 1
    fi
    #sync leafs recursively
    echo
"==========================================================================
====="
    echo "Sync leafs recursively"
    echo
"==========================================================================
====="
    function PRS_syncLeafs(){
        source=$2
        destination=$3
        progressfile=$4
        if [ -n "$progressfile" ] && grep -q -x -F "$1" $progressfile ;
then
            echo "skipping $1 because it was synced before according to
$progressfile"
        else
            echo -n -e "$1\0" | xargs -0 -I PPP rsync -aHvx --relative
--numeric-ids $source/./'PPP' $destination/ 2>/tmp/debug
            status=$?
            if [ -n "$progressfile" ]; then
                echo "$1" >> "$progressfile"
            fi
            return $status
        fi
    }
    export -f PRS_syncLeafs
    echo "$filelist" | tr \\n \\0 | xargs -0 -P $threads -I PPP
/bin/bash -c 'PRS_syncLeafs "$@"' _ PPP "$source" "$destination"
"$progressfile"
    status=$?
    if [ $? -gt 0 ]; then
        cat /tmp/debug
        rm /tmp/debug
        echo "ERROR: there was an error while syncing the leaf
directories recursively, check messages and try again"
        return 1
    fi
    #exit # uncomment for debugging what happenes before the final
rsync

    #run a single thread rsync across the entire project directory
    #to make sure nothing is left behind.
    echo
"==========================================================================
====="
    echo "final sync to double check"
    echo
"==========================================================================
====="
```

```
    rsync -aHvx --delete --numeric-ids $source/ $destination/
    if [ $? -gt 0 ]; then
        echo "ERROR: there was a problem during the final rsync, check
message and try again"
        return 1
    fi

    exit # comment out if you want to really do the md5 sums, this may
take very long!

    #create an md5 sum of the md5sums of all files of the entire
project directory to comapre it to the archive copy
    echo
"======================================================================
====="
    echo "sanity check"
    echo
"======================================================================
====="
    diff <( md5list $source ) <( md5list $destination )
    if [ $? -gt 0 ]; then
        echo "ERROR: the copy seems to be different from the source.
check the list of files with different md5sums above. Maybe the files
where modified during the copy process?"
        return 1
    fi

    echo "SUCCESS: the entire directory $project has successfully been
copied."
}
```

Usage you can run this function like so:

```
source prsync.sh
psync sourceHost:/source/directory target/destination 5 8 /tmp/dirlist
/tmp/progressfile
```

this will copy the /source/directory to /target/destination and it will dive 5 directory levels deep to parallelize rsyncs. it will run 8 rsync processes in parallel. with the optional `dirlist` and `progressfile` files, it will track its progress and skip all directories it has already rsynced when re-running it in case of an interrupted previous run.

catuion this is a work in progress.. I am writing down my notes as I go!

caution please be careful with the instructions below and think it through yourself. I will take no responsibility for any data loss as a result of this article.

here is, how i did it when i needed to copy 40 TB of data from one raidset to another while the server was still online serving files to everybody in the company:

testing

to test this script when modifying, I use a simple test-dataset which I extract to `/tmp/`. I then uncomment the “exit” statement before the “final sync to doublecheck” and run the script like so:

```
prsync.sh /tmp/source /tmp/target 3 1 /tmp/testdirlist /tmp/progressfile
```

to compare the resulting structure i use diff:

```
diff <(find source/|sed -e 's/source//' | sort ) <(find target/ | sed -e
's/target//' | sort)
```

and to delete the temporary files and target folder in order to re-run a fresh sync i run

```
rm -rf /tmp/target/* /tmp/testdirlist /tmp/progressfile
```

Before we get startet

one important note right at the begining: while parallelizing is certainly nice we have to consider, that spinning harddisks don't like concurrent file access. so be prepared to never ever see your harddisks theoretical throughput reached if you copy lots of small files. make sure you don't run too many parallel rsyncs by checking your cpu load with top. if you see the “wa” (waiting) load increase, it means you have too many processes. On the sytem i did this all for, first tried with 80 parallel rsyncs using option 2 below and i had a waiting load of about 50% and a througput of about 20MB/s. i then reduced to 15 parallel rsyncs and the waiting load went down to 25% and the bandwith went up to over 100MB/s. that is on a raid set that achieves a raw throughput of over 500MB/s if streaming performance is measured. just to give you an idea. besides `top` you can also use `iotop` to monitor your overall rsync speed.

Step 1: creat an incremental file list

depending on your needs, there are different options how to do that.

Option 1: rsync --dry-run

one possibly slow option is to do a dry-run of rsync with all your options you want to use and then use the file-list created by the dry-run for your rsync job.

first do the dry run:

```
rsync -aHvx --dry-run --out-format="%n" /source/ /target/ | tee
/tmp/rawfilelist
```

use rsync options like you would for a simple rsync run to copy all your files, but add the `–dry-run` `–out-format=“%n”` options. the out-format option is to make sure you get a simple list of files without the added information about symlinks and hardlinks, that you would get when this option was omitted.

now clean up the resulting file: the problem with the dry-run output is, that you also get directory names before you get the list of the contets of each directory. that's useless if we want to continue later on and run an rsync for each file. so we need to get rid of these directory paths. this will obviously lead to empty directories not being copied, we can fix that later on by running a simple single thread rsync at the end to fix things like exactly that :) so here we go.. let's clean up the filelist (you can do this inplace, but you might just want to use this line and pipe it directly into parallel further down the road)

```
cat /tmp/rawfilelist | sed -e 's/.*\/$//' | sed -e 's/sent .* bytes\/sec$//' | sed -e 's/^total .* (DRY RUN)$//' | sed -e 's/sending incremental file list//' | sed -e '/^$/d' > /tmp/filelist
```

Option 2: using find

after waiting too long for Option 1 to finish on a system that carried tons of backups of other systems, i tried this option:
if you have tons of files and want to skip the lengthy process of producing a file list via rsync, you can create a list of directories using find and then simply run an rsync per directory. this will give you the full parallelism at the begining but might end with a few ever lasting rsyncs if you don't dig deep enough when doing your initial directory list. still, this might save alot of time.

```
find /source/./ -maxdepth 5 -type d | perl -pe 's|^.*?/\./|\1|' > /tmp/rawfilelist
```

with the `–maxdepth` option you can set how deep you want to dive into your directory tree.. the goal is to get directories with a rather small number of files so you don't have to wait too long for the last couple of rsyncs to finish. also note the added `/./` at the end of the source path. that's important as we need this to define to which point rsync should be relative. also check out the man page of rsync, i stole the idea from there ;)

now it's time to clean up the list. we need to move all lines that contain less than the `-maxdepth` number of directories to a separate file list as these directories will need to be synced without recursion. i tried doing this with a loop that went through all lines trying to find the respective lines, but it took way too long for a rawfilelist with more than 300'000 entries, so i tried it with sed inplace and it was incredibly fast!

```
cp /tmp/rawfilelist /tmp/parentslist
cp /tmp/rawfilelist /tmp/filelist
sed -i '/^\(.*\/\)\{4\}.*$/d' /tmp/parentlist
sed -i '/^\(.*\/\)\{4\}.*$/!d' /tmp/filelist
```

make sure that the number in the sed regex is your `–maxdepth` number minus 1! now we need to sync the parents without recursion first before continuing to step 2

```
cat /tmp/parentlist | parallel -j 3 'shopt -s dotglob; rsync -aHvx --no-r --relative /tmp/source/./{}/* /tmp/target/'
```

the trick here is to use the –no-r option to remove recursion of whatever rsync parameters you have specified before it. also check out the `shopt` command which results in a * matching also hidden files like `.htaccess` and so on.

Step 2: run Rsync with GNU prallel

now it's time to feed our filelist into rsync and run our parallel sync job. in order to parallelize rsync we use the GNU tool `parallel`. it will take a list of files and run a command in parallel with as many processes as are specified by the `-j` option. in the command string, it will replace `{}` with the contents of the respective line. pretty simple :)

```
cat /tmp/filelist | parallel -j 10 rsync -aHvx --relative /source/./{}
/target/
```

note how, like in the above mentioned Option 2, we use the '/./' separator in the source path to tell rsync where to start with the relative path that it transmits to the client. also make sure you actually use the `–relative` option, otherwise your targets file structure will be very flat :)
note that parallel is thoroug as far as escaping goes. there are no quotes needed even with funny directory names.

Step 3: make sure we didn't miss anything

probably the best feature about rsync is, that it resumes aborted previous jobs nicely and it can be run several times across the same source and target with no harm. so let's use this property to just fix everything we have missed or done wrong by simply running a single thread rsync in the end. now this can take some time, and I know no way around that.

```
rsync -aHvx --delete /source/ /target/
```