Running a bash function in parallel using xargs

This post outlines how to run a bash function in parallel using xargs. (Note, you could optionally use "parallel" instead of xargs, but I see no advantages/disadvantages at this stage)

It may not be the best way, or the right way, and it may have unforeseen consequences, so I'd welcome any feedback on better practice.

Rationale

We often run scripts with for or while loops. In the simplest case if the operation within the loop is self contained, it's very easy to make it parallel.

E.g.

```
# written in confluence, might not actually run
for file in *.csv ; do
  cat $file | csv-slow-thin > ${file%.csv}.processed.csv
done
```

Becomes:

```
# written in confluence, might not actually run
echo 'file=$1;cat $file | csv-slow-thin > ${file%.csv}.processed.csv' >
do-slow-thing-imp
chmod 777 do-slow-thing-imp
cat *.csv | xargs -n1 -P8 do-slow-thing-impl
rm do-slow-thing-imp
```

But it's clunky to write a script file like that.

Better to make a function as follows, but the specific method in the code block below doesn't work

```
# written in confluence, might not actually run
function do-slow-thing
{
 file=$1
 cat $file | csv-slow-thin > ${file%.csv}.processed.csv
}
cat *.csv | xargs -n1 -P8 do-slow-thing #but this doesn't work
```

The following is the current best solution I'm aware of:

Note: set -a could be used to automatically export all subsequently declared vars, but it has caused problems with my bigger scripts

Note: set -a might have platform specific functionality. On Dmitry's machine it exports vars and functions, whereas on James' machine it exports vars only

Note: the use of declare -f means you don't need to work out a priori which nested functions may be called (e.g. like errcho in this example)

```
#!/bin/bash
export readonly name=$( basename $0 )
function errcho { (>&2 echo "$name: $1") }

export readonly global_var=hello
function example_function
{
        passed_var=$1
        errcho "example_function: global var is $global_var and passed var
is $passed_var"
}

errcho "first run as a single process"
example_function world

errcho "run parallel with xargs"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f) ;
example_function {}"
```

Note: if using comma_path_to_var, you can use --export to export all of the parsed command line options

No need to read beyond this point, unless you want to see the workings that lead up to this, including options that don't work.

The problem exposed and the solution

The following code is tested, try it, by copying into a script and running the script

```
#!/bin/bash

name=$( basename $0 )
function errcho { (>&2 echo "$name: $1") }

global_var=hello
function example_function
{
        passed_var=$1
        errcho "example_function: global var is $global_var and passed var
is $passed_var"
}

errcho "first run as a single process"
example_function world
```

Single process works fine, output:

> *xargs_from_func: first run single threaded*
>
> *xargs_from_func: example_function: global var is hello and passed var is world*

Lets try multiple processes with xargs. Add the following line to the end of the script:

```
errcho "run parallel with xargs, attempt 1"
(echo oranges; echo apples) | xargs -n1 -P2 example_function
```

The problem is that example_function is not an executable:

> *xargs_from_func: run parallel with xargs, attempt 1*
>
> *xargs: example_functionxargs: example_function: No such file or directory*
>
> *: No such file or directory*

Instead, let's run "bash" which is an executable:

```
errcho "run parallel with xargs, attempt 2"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "example_function
{}"
```

The new bash process doesn't know the function:

> *xargs_from_func: run parallel with xargs, attempt 2*
>
> *bash: example_function: command not found*
>
> *bash: example_function: command not found*

So let's declare it:

```
errcho "run parallel with xargs, attempt 3"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f
example_function) ; example_function {}"
```

Getting close, but our example_function refers to another of our functions, which also needs to be declared:

> *xargs_from_func: run parallel with xargs, attempt 3*
>
> *bash: line 3: errcho: command not found*
>
> *bash: line 3: errcho: command not found*

We can do that one by one, or declare all our functions in one go:

```
errcho "run parallel with xargs, attempt 4"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f
example_function) ; $(declare -f errcho) ; example_function {}"

errcho "run parallel with xargs, attempt 5"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f) ;
example_function {}"
```

The function itself is now working, but all the global variables are lost (including "global_var" and also the script name:

> *xargs_from_func: run parallel with xargs, attempt 4*
>
> *: example_function: global var is and passed var is oranges*
>
> *: example_function: global var is and passed var is apples*
>
> *xargs_from_func: run parallel with xargs, attempt 5*
>
> *: example_function: global var is and passed var is oranges*

> *: example_function: global var is and passed var is apples*

We can add these explcitly, one by one, e.g.:

```
errcho "run parallel with xargs, attempt 6"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f) ;
global_var=$global_var ; example_function {}"
```

...but it's extremely hard to work out which functions call which functions and which of all functions called use which global variables.

Leads to very hard to trace bugs in real world examples.

> *xargs_from_func: run parallel with xargs, attempt 6*
>
> *: example_function: global var is hello and passed var is oranges*
>
> *: example_function: global var is hello and passed var is apples*

So the final solution I've arrived at is to pass everything through by using "set":

```
errcho "run parallel with xargs, attempt 7"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(set) ;
example_function {}"
```

This spits out a lot of extra garbage because it includes an attempt to reallocate readonly variables:

> *xargs_from_func: run parallel with xargs, attempt 6*
>
> *bash: line 1: BASHOPTS: readonly variable*
>
> *bash: line 1: BASHOPTS: readonly variable*
>
> *bash: line 8: BASH_VERSINFO: readonly variable*
>
> *bash: line 8: BASH_VERSINFO: readonly variable*
>
> *bash: line 38: EUID: readonly variable*
>
> *bash: line 38: EUID: readonly variable*
>
> *bash: line 68: PPID: readonly variable*
>
> *bash: line 79: SHELLOPTS: readonly variable*
>
> *bash: line 87: UID: readonly variable*
>
> *bash: line 68: PPID: readonly variable*
>
> *bash: line 79: SHELLOPTS: readonly variable*
>
> *xargs_from_func: example_function: global var is hello and passed var is oranges*
>
> *bash: line 87: UID: readonly variable*
>
> *xargs_from_func: example_function: global var is hello and passed var is apples*

...but notice that it did work.

For some reason the following doesn't hide the readonly errors:

```
errcho "run parallel with xargs, attempt 7"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(set) > /dev/null
; example_function {}"
```

...and I've tried various combos of putting the dev/null inside the $(), and redirection of stderr.

I think, therefore, the best approach is to explictly declare each global using export, and to either explicitly export each function, or use the

"declare -f" statement at the xargs call

That looks like this:

```
#!/bin/bash
export readonly name=$( basename $0 )
function errcho { (>&2 echo "$name: $1") }

export readonly global_var=hello
function example_function
{
        passed_var=$1
        errcho "example_function: global var is $global_var and passed var
is $passed_var"
}

errcho "first run as a single process"
example_function world

errcho "run parallel with xargs"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f) ;
example_function {}"
```

Note: the readonly is not strictly necessary for this example, but is good practice if it is a readonly variable.

The whole script together:

```
#!/bin/bash

name=$( basename $0 )
function errcho { (>&2 echo "$name: $1") }

global_var=hello
function example_function
{
        passed_var=$1
        errcho "example_function: global var is $global_var and passed var
is $passed_var"
}

errcho "first run as a single process"
example_function world


errcho "run parallel with xargs, attempt 1"
(echo oranges; echo apples) | xargs -n1 -P2 example_function

errcho "run parallel with xargs, attempt 2"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "example_function
{}"

errcho "run parallel with xargs, attempt 3"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f
example_function) ; example_function {}"

errcho "run parallel with xargs, attempt 4"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f
example_function) ; $(declare -f errcho) ; example_function {}"

errcho "run parallel with xargs, attempt 5"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f) ;
example_function {}"

errcho "run parallel with xargs, attempt 6"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(declare -f) ;
global_var=$global_var ; example_function {}"

errcho "run parallel with xargs, attempt 7"
(echo oranges; echo apples) | xargs -n1 -P2 -i bash -c "$(set) ;
example_function {}"
```

Some external references:

http://stackoverflow.com/questions/1305237/how-to-list-variables-declared-in-script-in-bash

http://stackoverflow.com/questions/11003418/calling-functions-with-xargs-within-a-bash-scrip