Optimizing bash scripts for multi-core processors

PARALLEL THINKING

You don't need a heavy numeric mystery to benefit from the wonders of parallel processing. This article describes some simple techniques for parallelizing everyday bash scripts. **BY BERNHARD BABLOK**

If you want a piece of software to execute a task in parallel, the first challenge is to split that task into meaningful subtasks, which the computer can then processes simultaneously. Libraries such as OpenMP help programmers achieve this kind of parallelization.

Bash scripts typically don't handle numeric problems, so most programmers don't think of a bash script as a candidate for parallelization. The venerable bash shell, however, is used for other types of jobs that lend themselves to a parallel approach. For instance, a bash script is often employed as a tool for processing multiple files in the same way.

Listing 1 shows a shell function that processes all the arguments in the script one by one and passes the results to a program (*doSomething*). In this scenario, it is easy to imagine the benefits of some parallel-processing techniques.

Brute Force

Minimal changes to the code in Listing 1 produces the parallel-processing alternative shown in Listing 2. Listing 2 starts a

Does It Make Sense?

Before you start parallelizing all your bash scripts, it makes sense to consider whether it is indeed meaningful and achievable. Surprisingly, this question is fairly easy to answer. The *sar u P ALL 1 0* command can help you decide. (The *sar* utility is a part of the sysstat package.)

To run the test, launch your bash script in a second console. The *sar* command outputs the CPU load for all of your system CPUs (Figure 1).

In addition to the *%idle* value, the *%iowait* value is of interest. The *%iowait* value shows whether the processing has stopped because the system is waiting for I/O or some other reason.

The sar values make it easier to make a decision: Parallelization is only worthwhile if some of your processors are waiting while others are gasping for breath (as shown in Figure 1). Typical applications in this category are image and music conversions that generate a fair amount of CPU load, or logfile parsing scripts that use complex regular expressions. I/O-linked processes are not good candidates. Although you can parallelize the copying of 200 files from one directory to another, this strategy will not result in significant time savings if the disk is the bottleneck rather than the CPU.

Typically, if the processing steps depend on one another or if the processing order is important, you will have no viable alternative to a sequential order. A different algorithm might help, but the parallel approach proposed in this article will not result in a significant benefit.

Additionally, administrators should remember that it does not always make sense to fully load a computer. If you need to carry on with your daily chores (mailing, Internet, composing texts, and so on) while running CPU-intensive jobs, remember that sequential processing in the background, which only occupies one core, might be better than a fast alternative that blocks the whole system.

Listing 1: Serial Processing

```
01 doSeriell() {
02   local item
03   for item in "$@" do
04     doSomething "$item"
05   donexyyyyyyyyy
06 }
```

separate process for each argument. In line 6, the script waits for its child processes to terminate. This approach can cause problems: If the system is stressed by excessively large numbers of processes, the overhead will increase because of many context changes. In environments with limited memory, you also might see the machine slow to a crawl as it swaps individual processes. In some situations, however, this simple approach to parallelization makes sense.

Listing 2a is a variant on Listing 2. In this case, the arguments are not known; instead, a separate process (*createWorkItems*) creates them sequentially – this could be a *find* that is run against a very large filesystem. If the generating and processing rates, which depend on the number of available processors, are approximately the same, you will not experience system overload. If this is not the case, you will need a more elaborate solution. The script in Listing 3 distributes the arguments depending on the number of processors and then processes the subsets sequentially. Line 1 of the script determines the number of processors (*PMAX*) for the system. If the process is heavy on the I/O, it might make sense to set the number of processes to a value greater than *PMAX* to allow one process to work while another is waiting for I/O.

Unfortunately, bash only uses single-dimensional arrays, which makes the construct in lines 6 and 13 slightly complicated. For each process, the script creates a long string containing the arguments for the process within an array element (lines 5 through 9). The script then launches *PMAX* parallel processes (lines 11 through 14). Line 12 prevents empty processing (e.g., in the case of just two arguments on a quadcore machine), and *eval* in line 12 makes sure that the shell interprets the quotes in line 6 correctly.

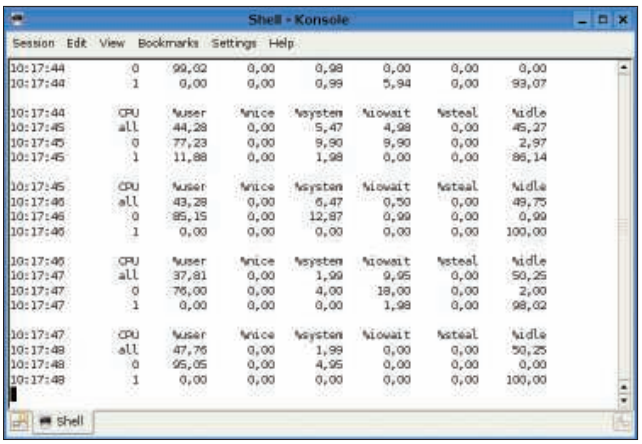

Figure 1: The output from sar shows the load for all CPUs. Parallelization only makes sense if some cores are overworked while others are doing nothing.

Dynamic Dispatcher

The scheme shown in Listing 3 is optimal if the average processing time per item is not subject to major fluctuation. Unfortunately, you can't always rely on this. For example, if you are converting multiple tracks, an unfavorable distribution of long and short tracks could mean that some processes finish sooner than others. Another situation in which this approach might be a problem is the task of converting images from digital cameras. Some cameras create JPG or thumbnail files in addition to RAW files. If every other file uses the RAW format and has to be converted, half of the conversion processes will finish significantly sooner because the processing scheme assigns all the RAW files to one process and all the JPG files to the other.

The processing method in Listing 3 also suffers in cases in which not all arguments are known in advance. If the arguments generated later in the script occur in sequence, it would not make sense to wait for them all to be created and then distribute them over the processes.

The solution for this problem is to use worker processes and a dynamic dispatcher. In this scenario, the script launches a number of worker processes.

Listing 2: Massively Parallel Processing

```
01 doMassiveParallel() {
02   local item
03   for item in "$@" do
04     doSomething "$item" &
05    done
06  wait
07 }
```

Listing 2a: Serial Input Parallel

```
01 doMassiveParallel2() {
02   local item
03   while read item do
04   doSomething "$item" &
05 done
06 wait
07 }
08
09 createWorkItems | doMassiveParallel
```

Listing 3: Parallel with Load Balancing

```
01  ${PMAX:=`ls 1d /sys/devices/system/cpu/cpu* | wc l`}
02
03 doParallel() {
04   local items item currentProcess=0
05   for item in "$@" do
06     items[$currentProcess]="${items[$currentProcess]} \"$item\""
07      shift
08     let currentProcess=$(( (currentProcess+1)%PMAX ))
09   done
10
11   for (( currentProcess=0 currentProcess<PMAX currentProcess++ )) do
12      [ n "${items[$currentProcess]}" ] &&
       eval doSequentiell ${items[$currentProcess]} &
13    done
14  wait
15 }
```

A dispatcher accepts tasks and distributes them as intelligently as possible to the workers. In contrast to the parallel solutions described earlier, in which all worker processes need to receive all their arguments at the start, the dispatcher talks to the workers after they have launched.

Named pipes or FIFOs are used as communications channels. To begin, the dispatcher opens a pipe for each worker and sends new tasks to the pipe (Figure 2). Another pipe that is shared by the dispatcher and the workers is used as the return channel. If a worker has nothing to do, it writes its ID to the pipe. The dispatcher reads an ID from the pipe after each task and sends the next task to this worker.

Listing 4 shows an implementation of this concept. In lines 1 to 4, the program sets a number of constants, if this has not already happened. Normally, the user will only define the *_cmd* variable. The *dispatchWork* function in lines 54 to 72 is the public part of the interface. The function starts by creating a temporary directory for all the pipes in line 55 (referred to as *controlDir* in the script). The *mkfifo* command in line 58 sets up the return channel.

Line 59 needs some explanation. Here, the shell opens the return channel for reading and writing, although read-only access is all it really needs. The problem is that read-only access to a pipe blocks the system call. A similar problem occurs in the *startWorker()* function, which creates a pipe for each worker process (line 37) and opens it for reading and writing (line 40).

The additional *eval* in line 40 is necessary because the bash parser processes the input redirection at a very early stage – before the variable substitution. (This also explains the backslashes before the lesser than and greater than symbols.)

Listing 4 simply contains functions – other scripts include this file and then use the *dispatchWork* function (see Listing 5).

Pitfalls

The script in Listing 4 has a couple of minor issues to contend with: For example, a *kill* command would leave orphaned worker processes (although this

Listing 4: Dynamic Dispatcher

```
001 ${DEBUG:=0}
002 ${_cmd:=echo}
003 ${PMAX:=`ls 1d /sys/devices/system/cpu/cpu* | wc l`}
004 ${FDOFF:=4}
005
006 processWorkItem() {
007   eval $_cmd \"$1\"
008 }
009
010 processWorkItems() {
011   local line workerFifo="$1" dispatcherFifo="$2" id="$3"
    fd
012   exec 3<>"$dispatcherFifo"
013   while ! echo "$id" >&3 do
014     sleep 1
015   done
016   let fd=id+FDOFF
017   while true do
018     read r u $fd line
019     if [ $? ne 0 ] then
020       break
021     fi
022     if [ "$line" = "EOF" ] then
023       break
024     else
025       processWorkItem "$line"
026       while ! echo "$id" >&3 do
027         sleep 1
028       done
029     fi
030   done
031   rm f "$workerFifo"
032 }
033
034 startWorker() {
035   local i fd fifo
036   for (( i=0 i<PMAX ++i )) do
037     workerFifo="$controlDir/worker$i"
038     mkfifo "$workerFifo"
039     let fd=i+FDOFF
040     eval exec $fd\<\> "$workerFifo"
041     processWorkItems "$workerFifo" "$dispatcherFifo" "$i"
    &
042   done
043 }
044
045 stopWorker() {
046   local i fifo
047   for (( i=0 i<PMAX ++i )) do
048     fifo="$controlDir/worker$i"
049     echo "EOF" > "$fifo"
050   done
051   wait
052 }
053
054 dispatchWork() {
055   local idleId dispatcherFifo  controlDir=`mktemp d`
056
057   dispatcherFifo="$controlDir/dispatcher"
058   mkfifo "$dispatcherFifo"
059   exec 3<>"$dispatcherFifo"
060
061   startWorker
062
063   while read r u 0 line do
064    read u 3 idleId
065    echo "$line" >>   "$controlDir/worker$idleId"
066   done
067
068   stopWorker
069
070   rm f "$dispatcherFifo"
071   rm fr "$controlDir"
072 }
```

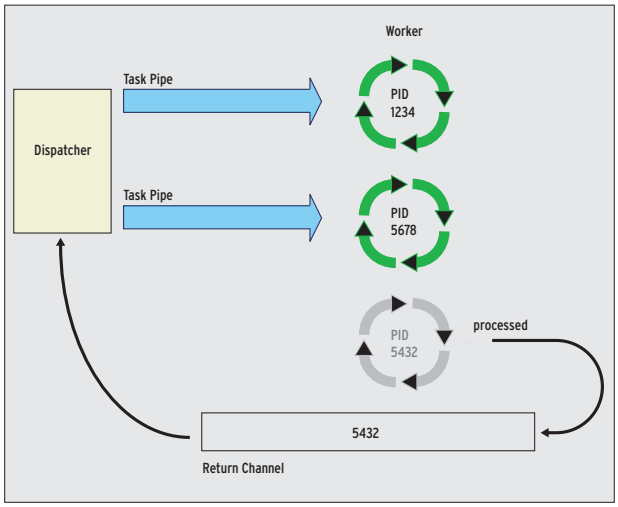

Figure 2: Dispatcher and worker processes use pipes to communicate.

problem could be handled by a timeout variable). Also, if you have more than six processes, the script will use file descriptors (channel numbers) that are greater than *9*. According to the bash manual, you have to be "careful" with this – whatever that means – because bash might already be using these descriptors for internal purposes. As a workaround, you can modify the offset for the channel numbers (line 4).

Other implementations are possible. For example, the dispatcher and workers could use files to communicate. The dispatcher would then write tasks to worker-specific files. Workers would use polling to see whether their worker files exist, process the tasks defined in the files, and then delete the files. At the other end, the dispatcher would check for worker files that are missing and thus would know which workers are idle.

Of course, this solution is typically inefficient because of the need for continuous polling.

A longer version of Listing 4 is available at the *Linux Magazine* website [1]. This expanded version supports calls to *dispatchWork* at the command line:

```
$ dispatchWork c ➘
"doSomething" file1 file2 [...]
```

The longer version also includes comments and some switches for optional debug output that allow administrators to monitor scripts.

Swapping Out to Other Machines

If you aren't satisfied with the efficiencies of parallel processing on a local machine, you can even apply this principle to the network. In that case, a first-level dispatcher could use TCP/IP to talk to multiple second-level dispatchers on various machines. The second-level dispatchers then talk to their local worker processes. This approach is only useful if you have a secure network, of course.

Conclusions

With just a couple of lines of code, you can use the techniques described in this article to parallelize existing shell scripts. Other scripting languages can use this approach; however, some languages offer superior alternatives. For example, Python uses explicit forking (*os.fork()*) in addition to pipes (*os.pipe()*), which allows for a low-level solution that is very close to the efficiency

Listing 5: Dispatcher at Work

```
001 source workDispatcher
002 doDynamic() { _cmd="doSomething" local item for item in "$@"
    do echo "$item" done | dispatchWork }
```

INFO

[1] Dynamic dispatcher source code: *http://www.linux-magazine.com/resources/article_code*

THE AUTHOR

Bernhard Bablok manages a large data warehouse with technical performance data for machines ranging from mainframes to servers at Allianz Shared Infrastructure Services. Besides listening to music, hiking, and biking, Bernhard enjoys anything related to Linux and object-oriented programming.

Benefits

For two benchmark programs, I used the dynamic dispatcher approach described in this article. In the first scenario, the script converts 20 RAW files to TIFF format on an Intel Quadcore machine (Q9450 with a clock speed of 2.67GHz and a 2x 6MB L2 cache).

If you pass all the files to *ufrawbatch* at once, the program takes 132 seconds (iterating autonomously over all the files). The dynamic dispatcher and *PMAX* = 12 and *PMAX* = 4 reduced the run time from 134 to 68 and 35 seconds. The efficiency of this method with four processors is thus approximately 95 percent, or to put it differently, the run time is reduced to almost a quarter.

The difference between this scenario and static parallelization is marginal. The reason for this small difference is that all of the source files are approximately the same size, so all the processors are equally stressed.

The second scenario uses another CPU-intensive method to convert WAV files to MP3, but with more difficult conditions this time. The script reads and writes the files from and to an NFS server with a 100MB network connection. Some interesting observations I made regarding this scenario are that, first, the method scales nicely (with run times of 207, 107, and 55 seconds – that is, 94 percent efficiency for four processors).

Also, the second run, in which the source files were in the NFS server's cache, differed only slightly compared with the local test. Finally, the use of five worker processes rather than four achieves slightly better results.

The effect of additional worker processes is more pronounced in the case of "narrower" data lines. However, sorting the WAV files in descending order of size had a more pronounced effect on throughput. At the end of processing, only one processor was still working on the last file, and this had a disproportionate effect on the run time. More optimization is possible, but enough is enough. In the case of complex simulations with run times of several hours or days, you would definitely want to experiment with additional optimization.

The energy balance of a computer working at full load is slightly better than that of a machine involved in sequential processing with just one core. However, you can save more power by switching off your screen while your computer is busy with complex processing work.