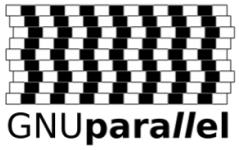
Thai Flowers and Arthur Karapateas

COP5570 Term Group Project
GNU Parallel Reimplementation Report
Fall 2019



Motivation

For our final term project, we re-implemented a parallel process UNIX tool titled GNU Parallel[1]. This tool was originally implemented in PERL, so our project fits under the category of re-developing an existing software application from scratch. Our project also featured the goal of splitting processes in parallel across multiple computers over a network The primary tasks of our project involved reimplementing from scratch the GNU Parallel shell scripting utility that takes a command and applies it in parallel to the provided data and or files, one process per core. GNU Parallel is a useful tool because it takes a program or a command that isn't parallel(or a for loop) and allows you to run the program or command in parallel, handling different sets of data at the same time, as long as the data has no interdependencies.

Originally, we were unaware of this tool's existence. Our plan was to implement a new parallel processing UNIX tool that could split up work across multiple computers on the same network. The goal of the tool was to provide a simple method for people unfamiliar with the complexities of parallelization to be able to simulate parallelized code by running parallel copies of their code at the same time on different data. Upon research we found that an already existing tool titled GNU Parallel was implemented by Ole Tange and released in 2007[1]. The project was officially added to the GNU project on 04-22-2010. This tool was written in PERL and was approximately ~11,000 lines of code. GNU Parallel functions as a form of process-based parallelization that works on files, command line arguments, and input sources. GNU Parallel also allows for remote execution which is one of the tasks we were set on implementing for our tool to split up work across multiple computers on the same network. After the processes are completed the output is printed sequentially. After discovering this tool we thought rather than reinventing a tool that accomplishes the same task, we could reimplement this tool in a separate language. As PERL is a scripting language that can run less efficiently with memory and runtime compared to a compiled language, we opted to provide some form of improvement to the original implementation of the tool by implementing it in a compiled language. Since all other assignments in this course were implemented in C/C++, we thought it would be best to implement in C. Another benefit to implementing this tool in C is that it would provide a cleaner interface for adding future functionality to the tool, as the PERL implementation is running on out dated PERL code.

Project Implementation

Our first task was to implement a bare bones, basic version of a GNU Parallel like program in C. This task was completed, and we followed the GNU Parallel Cheat sheet to see what basic functions should be handled by our program. Parsing through the PERL code to understand what should be done would be challenging for us as neither of us are familiar enough with the PERL language to understand what features to implement.

Based on researching this tool via the GNU Parallel Manual[2], the tool can act as a replacement for xargs and for loops. It also has the ability to split up a stream or file into separate blocks and pass those items to commands all in parallel. This functionality was completed but some aspects of our tool varied in their implementation in comparison to the original GNU Parallel. For instance, we cannot split up the work to do a various specific amount of jobs per CPU, our implementation always splits the processes in a first come first server manner on each CPU core on the machine. Our implementation acts in a one job per CPU core manner, and we can limit the number of cores used with the -jobs option flag. Another task that our GNU Parallel implementation cannot handle is perl regex expression handling, as our code was written in C, this task would prove to be very time consuming to handle properly. The perl implementation can handle this very easily because it can simply take the regex string and run an eval command on the regex expression to handle it.

Input sources in our implementation of GNU Parallel were handled identically to how they are handled in the original implementation of the tool, but our tool lacks additional features for basic input sources, the 4 colon input method and piped-in input. In the 4-colon method each word is a filename that is an input source itself. Our implementation handles the 3 colon input method ":::", that being the command is followed by three colons followed by space separated words, followed optionally by additional input sources. Each word from each input source is substituted into a curly braced substitution pattern that is processed. Our implementation can properly handle all basic replacement string patterns except the perl expressions. The --dryrun option just outputs the parsed/substituted command line.

Example Syntax for Usage: ./parallel --dryrun -j<#jobs> command ::: a b c d

One of the most challenging aspect of our project was getting GNU Parallel to work across multiple computers on the same network. This task was not entirely completed with full functionality as we reached a memory leak hurdle when making our code handle both quoted input and non-quoted input. The implementation for handling quoted input involved performing a form of "sh wrapping" for allowing bash/sh function calls. This sh wrapping was necessary in order to complete the next task of sending our jobs over the network. When our code was modified to perform this task, non-quoted input would result in a memory leak that we were unable to properly resolve. Due to this, in the interest of time we split our implementation of GNU Parallel into 2 separate versions. One version specifically handled non-quoted input and the other handled quoted input that could be properly processed by our form of sh wrapping. After this step we researched what tools we could use to send our tasks to remote systems on the same network. We discovered the

best way to approach this task was to configure password-less ssh login for desired users on all remote systems via a public/private key pair. After configuring this, the next task is to implement a form of "ssh wrapping" to modify our parallel implementation to accept -S <servername> option command with a remote system address to ssh into. After doing this we would be able to send our jobs over the network via ssh. Due to time constraints this task was not fully functioning as we still had to determine how to split the number of jobs per connection, and this task was left for future work.

Replacement string patterns our tool handles based on GNU cheatsheet examples[3]:

Replacement String Patterns:	**Value replaced based on input being dir/subdir/file.ext**
{}	dir/subdir/file.ext
{.}	dir/subdir/file
{/}, {//}, {/.}	file.ext, dir/subdir, file
{#}	Sequence number of job
{%}	Job slot number
{3}	Value from 3rd input source
{3.} {3/} {3//} {3/.}	Combination of value from numbered input source and {.} {/} {//} {/.}

Example Tasks That Our Tool Can Handle

- **Test ability to run simple executable program with 3 different inputs**
  - **parallel -j4 ./fib ::: 30 50 10**
- **Test replacement string parsing and run ls command in current directory**
  - **parallel --dryrun echo '{#} {%} {} --> {.} && {/} && {/.} && {//}' ::: `ls ./*`**
- **Test remote execution but on the same server(to test sh parsing)**
  - **parallel "echo {} ; hostname" ::: a b c**
- **Compress all html files, 2 jobs per "thread":**
  - **parallel --jobs 2 gzip ::: *.html**
- **Convert wav files to mp3, 1 job per cpu(default)**
  - **parallel lame {} –o {.}.mp3 ::: *.wav**
- **Run 2 jobs in parallel sleep and echo**
  - **parallel --jobs 2 "sleep{}; echo{}":::5 4 3 2 1**

Original GNU Parallel vs Our Implementation of GNU Parallel

• Huge number of features • Complex command line parsing and internals • Uses outdated PERL, and implemented in 1 file to aid installing in users personal directories • Heavy bash dependency/assumption • Implemented in PERL	• Extremely basic feature set. • Simple command line parsing and internals • Loose sh dependency via sh wrapping, GNU assumes we are always using bash, and our tool should also work in tcshell. More shell agnostic than original GNU Parallel. • Mostly standards conformant C

Implicit Thesis – GNU Parallel Could Have Been Introduced in 1987 not 2007 (Further Research)

While working on the implementation portion of the project we began to form an implicit thesis behind this project, an implicit proof of concept. We had chosen to use C as our implementation language as we had just completed the shell implementation project, the tasks seemed similar, and we were already comfortable with C as our primary language again thanks to the aforementioned project. We realized that instead of using C++ and attempting to hold to old standards of C (ANSI C in particular) that a tool like GNU Parallel could have possibly been written in 1987 rather than 2007. Provided multicore, or more accurately for the time period multiprocessor, computers and operating systems were available for everyday use (as in not just limited to expensive prototypes in inaccessible laboratories).

Before beginning with our historical exploration to substantiate this hunch we will first mention a few important dates in GNU Parallel history. Despite being a somewhat common/popular GNU tool, it has not been an official GNU project for very long. As of this writing the official website is preparing to celebrate ten years of GNU Parallel as an official project on its anniversary: April 22, 2020.[3] Also, unlike other popular tools by GNU such as bash and gcc there is no codebase stretching back to the early nineties and before under various names and maintainers, the earliest code online for Parallel is from 2007[4]

Another item of note before diving into some information on multiprocessor operating systems is that remote command execution from trusted machines was a common practice by the late eighties or early nineties. We discovered this from an old textbook and reference guide entitled “Unix for Programmers and Users” 2nd ed. by Glass and Ables. In the examples given and in the discussions of internet software used at the time there is no mention of ssh, instead rsh is used, where rsh is short for “remote shell”. Interestingly it uses a passwordless login system for running commands on other machines on the local network, much like how GNU Parallel needs ssh passwordless login enabled in order to distribute its tasks across the network.

The first item of interesting multiprocessor operating system history is the famed failure that preceded Unix: Multics. As can be seen on the “Multicians” Multics fan site it is reported as being a multiprocessor operating system that originated in 1965.[6] There are numerous features listed there that were copied/ported into Unix over time. Interesting examples include a kernel based design, online documentation, and a tree structured file system. However, multiprocessor compatible variants of Unix wouldn’t come for several years after its inception, mostly due to its “unitary” design and resulting rejection of many of the complexities of multics that came from its habit of multiplexing based designs.

The earliest example of such a Unix variety we could find was “MUNIX”, short for “Multiprocessing Unix”. It was released in 1975 as part of a thesis by John Alfred Hawley, a student at the Navy Postgrad school. [7] It was a non-commercial version of unix.

We found this example thanks to a paper from 1983 discussing the implementation of such systems. [8] This proves the topic was popular to discuss at the time, especially due to the list of papers cited of similar, then recent, projects. In a similar vein, a search for "multiprocessor unix" yields a flurry of other research projects implementing such unix systems, many published between 1986 and 1989. A few examples that can be easily located are Mosix, Plurix, and Mach. Mach is a particular interesting example as papers describing it were first published in 1986, in particular the one we cite here describing it as "A New Kernel Foundation for UNIX Development" [9]. As to why this is interesting, the modern Mac OS X and iOS XNU kernel is supposedly based on Mach, however good academic sources for this are a little more difficult to find. See Apple's open source archives for the XNU readme for direct confirmation and the Darwin documentation for more details.

So our argument is based on the following pieces of evidence. By 1987 parallel execution of different processes was available on many unix systems, albeit not necessarily the most mainstream variants. However, it was also a popular research and discussion topic during the time period. In the same era rsh was a commonly used tool for remote command execution, and especially for commands issued without prompting a password on trusted machines on the local network. Also, we know from old Posix standards that the fork and exec model was also common by this time.

From this we can make a simple conclusion: a rudimentary version of parallel, complete with networking features, could have been developed in 1987 instead of 2007. We can only surmise that the interest or imagination needed was not available to create the tool. Perhaps this is due to some barrier of entry, or adoption issue that we have failed to acknowledge.

References

[1] Gnu.org. (2007). *GNU Parallel- GNU Project - Free Software Foundation*. [online] Available at: https://www.gnu.org/software/parallel/ [Accessed 7 Dec. 2019].

[2] Gnu.org. (2007). *GNU Parallel Manual - GNU Project - Free Software Foundation*. [online] Available at: https://www.gnu.org/software/parallel/man.html [Accessed 7 Dec. 2019].

[3] Gnu.org. (2007). *GNU Parallel Manual - GNU Project - Free Software Foundation*. [online] Available at: https://www.gnu.org/software/parallel/parallel_cheat.pdf [Accessed 7 Dec. 2019].

[4] Savannah.gnu.org. (2010). *GNU Parallel - Summary [Savannah]*. [online] Available at: https://savannah.gnu.org/projects/parallel/ [Accessed 7 Dec. 2019].

[5] Glass, & Ables. (1993). *Unix for programmers and users* .. (2nd ed.).

[6] Multicians.org. (n.d.). *Multics History*. [online] Available at: https://multicians.org/history.html#tag1 [Accessed 7 Dec. 2019].

[7] Core.ac.uk. (1975). [online] Available at: https://core.ac.uk/download/pdf/36714194.pdf [Accessed 7 Dec. 2019].

[8] Bach, M. J., & Buroff, S. J. (1984). The UNIX System: MultiprocessorUNIXOperating Systems. *AT&T Bell Laboratories Technical Journal*, *63*(8), 1733–1749. doi: 10.1002/j.1538-7305.1984.tb00062.x

[9] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.3964