Using the SPF Algorithm to Optimize the Performance of UNIX Stream Processing in a Distributed Computing Environment

Kenan Kalajdzic <kenan@unix.ba>

Written in March 2012 (original idea 2007)

Abstract

In this document we present the concept of a work-distributing UNIX shell, which relies on the Shortest Path First algorithm to optimally distribute the execution of shell pipelines across multiple computers. The two biggest difficulties in designing such a shell are finding the way to define the meaning of "cost" and calculating the costs of all individual components constituting a shell pipeline. Once the costs are correctly defined and calculated, applying the SPF algorithm is a fairly straightforward process.

1 Introduction

Of all the ideas, which have contributed to the great success of the UNIX operating system, *pipes* are probably among the most significant. They allow the user to quickly connect multiple independent programs into complex commands usually referred to as *pipelines*.

Even though pipes are primarily designed for synchronous sequential processing of data streams, a pipeline consists of multiple processes, which allows for a certain degree of parallelism. With the expansion of computer networks and the invention of the *remote shell* service, it became possible to construct pipelines with commands which may be spread across multiple computers. For example, instead of writing

```
$ tar cf - /home | gzip -c >home.tar.gz
```

a user could execute the `tar` command on the local computer and let a remote host perform the compression with `gzip`:

```
$ tar cf - /home | rsh rhost 'gzip -c' >home.tar.gz
```

Using this technique occasionally as an easy way to speed up the execution of time-consuming pipelines is reasonable. It can, however, not be accepted as a general practice, because it complicates the usage of the command line.

Instead of requiring the user to manually parallelize the pipeline by explicitly inserting invocations to `rsh`, the parallelization should rather be performed automatically by the shell[1]. The user would type in the command line as usual, and the shell would intelligently and optimally distribute all the processes in the pipeline across the available nodes and CPUs. If done correctly, this implicit parallelization would allow millions of shell scripts worldwide to run unmodified on any given configuration of computers and processors.

Consider, for example, a user who wants to rotate a JPEG image and scale it down to 20% of its original size using NetPBM image manipulation tools. To do so, the user could enter the following self-explanatory shell command line:

```
$ jpegtopnm <in.jpg | pnmrotate 90 | pnmscale 0.2 \
  | pnmtojpeg >out.jpg
```

For each of the four commands in the pipeline, the work-distributing shell would decide whether to run it locally or remotely and insert the invocations to `rsh` appropriately, as in the following example:

```
jpegtopnm <in.jpg | rsh remotehost1 'pnmrotate 90' \
| rsh remotehost2 'pnmscale 0.2' | pnmtojpeg >out.jpg
```

The command line is transformed transparently "behind the scenes", so that the user is not aware that some of the commands are being executed on remote computers.

2 Optimal distribution of processes

When a user, working on a single isolated computer, enters the following simple command line

```
$ cmd1 <in | cmd2 | cmd3 >out
```

the shell spawns three processes, P_1, P_2 and P_3, and connects them by pipes as shown in Figure 1. Here we assume that P_1 executes `cmd1`, P_2 executes `cmd2` and P_3 executes `cmd3`. The data stream originating at the input *in* has a unique, precisely defined processing path, which we will conveniently denote by $(in \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow out)$.

[1] In this document, we interchangeably use the terms *shell* and *work-distributing shell* to refer to a command interpreter which has the ability to automatically and intelligently distribute the work across multiple computers.

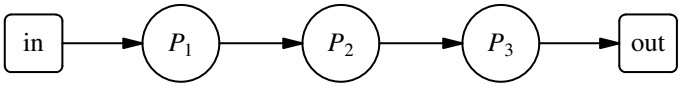

Figure 1: A simple pipeline consisting of three processes running on a single computer

When, beside the local host, the user has access to one remote computer, the shell may distribute the execution of the processes P_1, P_2 and P_3 in several different ways. The graph in Figure 2 shows all the possible data paths between *in* and *out*. The vertices of the graph represent processes executing on one of the two computers, while its edges represent pipes and network links used for data streaming between individual processes. By $P_{i,L}$ we denote that the process P_i is executing on the local host, whereas $P_{i,R}$ means that the process P_i is executing on the remote host. We also assume that *in* and *out* reside on the local host, where the shell is running.

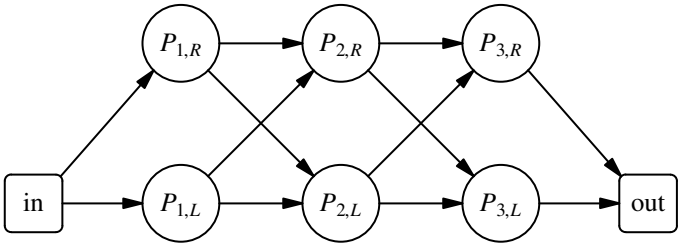

Figure 2: A graph showing all possible paths for distributing processes P_1, P_2 and P_3 across two computers

Without having to worry about where the individual processes will be scheduled to run, the user would enter the exactly same command line as in the case of a single computer:

```
$ cmd1 <in | cmd2 | cmd3 >out
```

This time, however, the shell attempts to distribute the three commands across the two available computers in an optimal way which minimizes the total execution time. This happens in two steps:

1. The shell first assigns costs to all vertices and edges of the graph in Figure 2. The costs of the vertices quantitatively describe how much each process contributes to the overall execution time. The costs of the edges take into account the time required for transferring data between any two processes, along with delays introduced by context switching and a possibly increased system or network load.

2. Once all the individual costs have been assigned, the shell applies the Shortest Path First algorithm to find the data path with the potentially minimal execution time.

After having determined the optimal data path, the work-distributing shell inserts `rsh` invocations as appropriate and executes the transformed pipeline.

2.1 Example 1

In this scenario, we assume that the size of the input *in* is not very large. In such a case the shell may decide that it is beneficial to send the input directly to *rhost* and execute `cmd1` there, while scheduling `cmd2` and `cmd3` for local execution:

```
$ rsh rhost 'cmd1' <in | cmd2 | cmd3 >out
```

This execution pattern corresponds to the data path ($in \rightarrow P_{1,R} \rightarrow P_{2,L} \rightarrow P_{3,L} \rightarrow out$) which is shown in Figure 3.

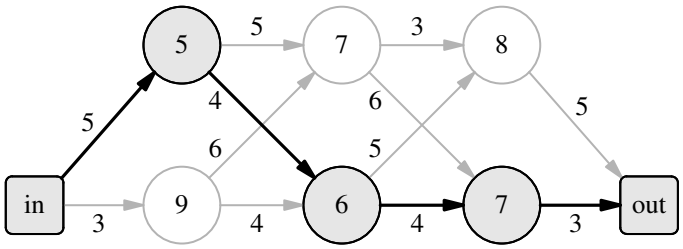

Figure 3: The result of running the SPF algorithm. Process P_1 is executed on the remote host, while P_2 and P_3 are run locally.

2.2 Example 2

Now assume that the remote host is a multiprocessor computer with more processing power than the local host. It may therefore seem reasonable to run all three commands on the remote host by modifying the original pipeline in the following way:

```
$ rsh rhost 'cmd1 | cmd2 | cmd3' <in >out
```

The data path for this execution pattern would be ($in \rightarrow P_{1,R} \rightarrow P_{2,R} \rightarrow P_{3,R} \rightarrow out$).

Nevertheless, using its database of accumulated knowledge the shell determines that `cmd1` features some sort of data compression, so that the size of its output is on average around 30% of the size of its input. So, even

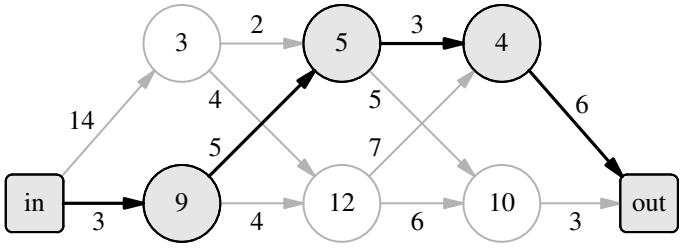

Figure 4: The result of running the SPF algorithm. Process P_1 is run on the local host, while P_2 and P_3 are executed on the remote host.

though the remote host is computationally superior to the local host, if the size of the input is very large, the shell may estimate that it would be infeasible to send the large amount of data over the network to *rhost*. It might therefore decide to run `cmd1` locally and continue with the processing of the pipeline on the remote host. The data path for this scenario would thus be $(in \rightarrow P_{1,L} \rightarrow P_{2,R} \rightarrow P_{3,R} \rightarrow out)$ (Figure 4), which would correspond to the following command line:

```
$ cmd1 <in | rsh rhost 'cmd2 | cmd3' >out
```

2.3 Calculating the costs

The two examples from Sections 2.1 and 2.2 show the results of applying the Shortest Path First algorithm to find the optimal data path for our pipeline. Nevertheless, one important question still remained unanswered:

> *How does the work-distributing shell exactly calculate the individual costs for each vertex and edge of the graph?*

Specifically, how does the shell know that executing process P_3 on the remote host contributes twice as much to the total execution time as transferring data between processes P_1 and P_2 on the local host (Figure 3)?

We might be able to find the answer to this question through extensive research. To appreciate the challenges which this research could bear, let us briefly look into the problem of cost calculation in a little bit more detail.

2.3.1 The complexity of cost calculation

When trying to calculate the costs of processing and data transfers, the work-distributing shell needs to take multiple factors into account:

1. **Speed and number of processors/cores on each node** – Knowing how many processor cores are available on each of the nodes and how fast these are is the essential information which a work-distributing shell would use when deciding how to distribute all the processes in a pipeline. The shell could obtain this information by "asking the system" about it (e.g. reading `/proc/cpuinfo` on Linux systems), or by running benchmarks to obtain some sort of a measure.

2. **Speed of communication links between nodes** – The shell needs this measure to correctly estimate how much time would be required to send data over the network to a remote node. It helps the shell decide whether sending the data is feasible or not in a given situation. Along with the CPU speed, this measure belongs to the most important information upon which a work-distributing shell would base its cost calculations.

3. **Different or unpredictable nature of commands** – This is where things get really complicated. There is a large number of commands which all behave differently and have different processing patterns and requirements. Some commands are CPU-intensive, while others mostly focus on I/O operations. The shell must be able to differentiate between these two categories of programs and "understand" how they will affect the execution time of a pipeline.

   Then, there are differences in the input/output ratio among commands. Some programs, such as `cat` or `pnmrotate` (rotates an image), don't alter the size of the input stream. Others, like `gzip` or `bzip2`, usually produce output which is smaller than their input, while some (e.g. `bunzip2`) inflate their input to produce a bigger output. But knowing that a command compresses its input is, unfortunately, not enough. The compression ratio will most of the time be dependent on the type of input and the algorithm used. For example, trying to compress an already compressed input won't yield any further compression, while compressing a plain text file may produce output of a size somewhere between 30% and 40% of the size of the input, depending on whether the user typed `gzip -1` or `gzip -9`.

   To cope with all these complexities, a work-distributing shell must include a fairly sophisticated semantic layer which would help the shell learn about the behavior of different commands and use the accumulated knowledge to estimate how much a certain command would contribute to the overall execution time of a pipeline. But even with a significant amount of knowledge, a work-distributing shell may not always be able to precisely calculate all the costs. This calls for introduction of a probabilistic model to help the shell reach a reasonably good decision in cases when some information is missing.

4. **Current load of each node** – The current system load may greatly affect the cost calculation for any node. To estimate the load for each individual computer, the shell could use the values of the *load average* for the last minute and the current CPU load for each of the available processors. On Linux systems, for example, these values can be obtained from the files `/proc/loadavg` and `/proc/PID/stat` for each of the processes.

5. **Detecting which operations are inherently local** – A work-distributing shell must have the ability to recognize commands which cannot be executed on a remote node. In the pipeline

   ```
   $ tar cf - /home | gzip -c >home.tar.gz
   ```

   `gzip` can be moved to a remote computer, but `tar` must be executed locally, because it is used to archive the **local** `/home` directory!