

DeBasher: a Flow-Based Programming Bash Extension for the Implementation of Complex and Interactive Workflows with Stateful Processes

Daniel Ortiz-Martínez¹

¹Department of Mathematics and Computer Science, Universitat de Barcelona, Gran Via de les Corts Catalanes,
585, 08007, Barcelona, Spain

Supplementary Materials

Contents

1	Sample Debasher Program	3
2	Sample Debasher Program Using Python, R, Perl and Groovy	4
3	The Telegram Problem	6
4	Telegram Problem Implementation with DeBasher	6
4.1	Original Telegram Problem	6
4.2	Telegram Problem With Two Processes	10
4.3	Telegram Problem With Two Job Steps	11
4.4	Array of Telegram Problems	13
4.5	Imperative Telegram Problem	14
5	Telegram Problem Implementation with Other Tools	15
5.1	CWL	16
5.2	WDL	17
5.3	Nextflow	17
5.4	Bpipe	18
5.5	Snakemake	19
5.6	Toil	19
6	Programs with Arrays: a Code Reuse Example	20
7	Network Reconfiguration Example	21
8	Programs with Cycles	23
9	Dynamic Scheduling Example	25

10 User-Defined Triggers and Interactive Programs	26
11 Runtime Piping	27
12 Scalability Experiments	29
12.1 Experiments Configuration	29
12.2 Results	30
13 Sample Bioinformatics Workflow	32
References	35

1 Sample Debasher Program

To illustrate how a DeBasher program looks like, Figure S1 shows the implementation of the “Hello World!” program¹. Extended information about creating programs with DeBasher can be found in its technical documentation². The code has four functions:

- **hello_world_explain_cmdline_opts**: this function is used to describe or document the options of the **hello_world** process, and more specifically, those that should be specified through the command line by the user. In this case, the process accepts only the **-s** option, which allows to specify the string to be displayed by the program.
- **hello_world_define_opts**: this function should establish the options passed to the **hello_world** function (this differs from the previous function, whose purpose was just to document command line options). In this case, the **-s** option provided in the command line by the user will be passed to the process. If no option is provided, then the “Hello World!” string is used as a default value.
- **hello_world**: the **hello_world** function implements the process of the same name. It reads the option defined by the **hello_world_define_opts** function and print it to the standard output.
- **debasher_hello_world_program**: this function defines the program implemented by the module. In particular, the program executes only the **hello_world** process.

```
1 hello_world_explain_cmdline_opts()
2 {
3     # -s option
4     local description="String to be displayed ('Hello World!' by default)"
5     explain_cmdline_opt "-s" "<string>" "$description"
6 }
7
8 hello_world_define_opts()
9 {
10    # Initialize variables
11    local cmdline=$1
12    local optlist=""
13
14    # Obtain value of -s option
15    local str=$(get_cmdline_opt "${cmdline}" "-s")
16
17    # -s option
18    if [ "${str}" = "${OPT_NOT_FOUND}" ]; then
19        define_opt "-s" "Hello World!" optlist || return 1
20    else
21        define_opt "-s" "$str" optlist || return 1
22    fi
23
24    # Save option list
25    save_opt_list optlist
26 }
27
28 hello_world()
29 {
30    # Initialize variables
31    local str=$(read_opt_value_from_func_args "-s" "$@")
32
33    # Show message
34    echo "${str}"
35 }
36
37 debasher_hello_world_program()
38 {
39     add_debasher_process "hello_world" "cpus=1 mem=32 time=00:01:00"
40 }
```

Figure S1: DeBasher implementation of “Hello World!” program.

DeBasher incorporates functionality to graphically represent the network associated to a given program. Figure S2 shows the graph for the “Hello World!” program, which contains only one process (represented by a rectangle) with one input option (represented by an ellipse). A gray box is used to enclose the two elements.

¹Note: all of the diagrams shown in this document were automatically generated with DeBasher.

²<https://debasher.readthedocs.io/en/latest/>

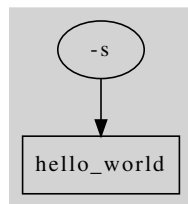


Figure S2: Process graph for the “Hello World!” DeBasher program.

2 Sample Debasher Program Using Python, R, Perl and Groovy

Due to the fact that FBP is language agnostic, process implementation can be done in multiple languages. For this purpose, it is necessary to store the code in a variable with a specific name, that results from concatenating the name of the process and the extension of the files used by the language. The code should ensure that the input and output options of the specific process are read and carefully processed.

Figure S3 shows the implementation of the `hello_world` process using Python, R, Perl and Groovy. To build the complete “Hello World!” program, it would only be necessary to replace the `hello_world` function in Figure S1 by the implementation in the desired language.

Python

```
1 hello_world_py=$(cat <<'EOF'
2 import argparse
3
4 # Create the parser
5 parser = argparse.ArgumentParser()
6
7 # Add the "-s" option with an integer argument
8 parser.add_argument('-s', type=str, required=True, help='String to be displayed')
9
10 # Parse the arguments
11 args = parser.parse_args()
12
13 # Access the value of "-s"
14 s = args.s
15
16 # Print message
17 print(s)
18 EOF
19 )
```

R

```
1 hello_world_r=$(cat <<'EOF'
2 args <- commandArgs(trailingOnly = TRUE)
3
4 # Function to parse arguments
5 parse_args <- function(args) {
6   options <- list()
7   i <- 1
8   while (i <= length(args)) {
9     if (args[i] == "-s") {
10       if ((i + 1) <= length(args)) {
11         options$string <- args[i + 1]
12         i <- i + 1
13       } else {
14         stop("Option -s requires a string argument.")
15       }
16     }
17     i <- i + 1
18   }
19   return(options)
20 }
21
22 # Parse the command-line arguments
23 options <- parse_args(args)
24
25 # Ensure the string was provided
26 if (is.null(options$string)) {
27   stop("You must provide a string with the -s option.")
28 }
29
30 # Print the string
31 print(options$string)
32 EOF
33 )
```

Perl

```
1 hello_world_perl=$(cat <<'EOF'
2 use strict;
3 use warnings;
4 use Getopt::Long;
5
6 # Variables to hold command-line options
7 my $string;
8
9 # Parse command-line options
10 GetOptions("s=s" => \$string) or die "Error in command line arguments\n";
11
12 # Ensure the string was provided
13 die "You must provide a string with the -s option.\n" unless $string;
14
15 # Print string
16 print "$string\n";
17 EOF
18 )
```

Groovy

```
1 hello_world_groovy=$(cat <<'EOF'
2 def parseArgs(args) {
3   def options = [:]
4   args.eachWithIndex { arg, index ->
5     if (arg == '-s' && index < args.size() - 1) {
6       options.string = args[index + 1]
7     }
8   }
9   if (!options.string) {
10     throw new IllegalArgumentException("You must provide a string with the -s option.")
11   }
12   return options
13 }
14
15 def options = parseArgs(this.args)
16
17 println "${options.string}"
18 EOF
19 )
```

Figure S3: Implementation details of the `hello_world` process using Python, R, Perl and Groovy.

3 The Telegram Problem

The Telegram problem is used in [4] to illustrate how FBP programs are designed. More specifically, it involves creating a program that takes input lines of text and produces output lines with maximum word count, ensuring that each line's character count remains under a specified limit. Words must remain intact without splitting, and it is assumed that no word exceeds the line's length.

Figure S4 shows a Bash implementation of the Telegram problem using two functions, one to decompose the input text into words (**decompose**) and another one that takes the words and generates lines with a given maximum length (**recompose**). Both functions internally use the UNIX **awk** command for text processing. Once the two functions have been defined, they are combined by means of a Bash pipeline.

```
1 decompose()
2 {
3     local inf=$1
4
5     awk '{for(i=1;i<=NF;++i) print $i}' "${inf}"
6 }
7
8 recompose()
9 {
10    local char_lim=$1
11
12    awk -v maxlen="${char_lim}" 'BEGIN{len=0}
13    {
14        if(len + length($0) <= char_lim)
15        {
16            if(len > 0) printf " "
17            printf "%s", $0
18            len = len + length($0)
19        }
20        else
21        {
22            printf "\n%s", $0
23            len = length($0)
24        }
25        if(len+1 <= char_lim)
26            len = len + 1
27    },
28    '
29
30    inf=$1
31    char_lim=$2
32
33    decompose "${inf}" | recompose "${char_lim}"
```

Figure S4: Bash implementation of the Telegram problem.

4 Telegram Problem Implementation with DeBasher

This section describes five possible ways to implement the Telegram problem using DeBasher.

4.1 Original Telegram Problem

The original implementation of the Telegram problem from an FBP perspective was proposed in [4]. In particular, the author mentions the necessity of defining four processes that are depicted in Figure S5. Those processes are the following:

- **rseq**: this process takes a file as input (**-f** option) and reads it line by line, sending the file content to the **decomposer** process.
- **decomposer**: this process fragments the content of the input file and fragments it into words that are sent to the **recomposer** process.
- **recomposer**: the **recomposer** process takes as input the stream of words composing the input file provided by the **decomposer** process and the character limit per each line (**-c** option) and generates a stream of text lines whose length is below the character limit. This stream is sent to the **wseq** process.

- **wseq**: finally, the **wseq** process takes the stream of text generated by the **recomposer** process and writes it to a file.

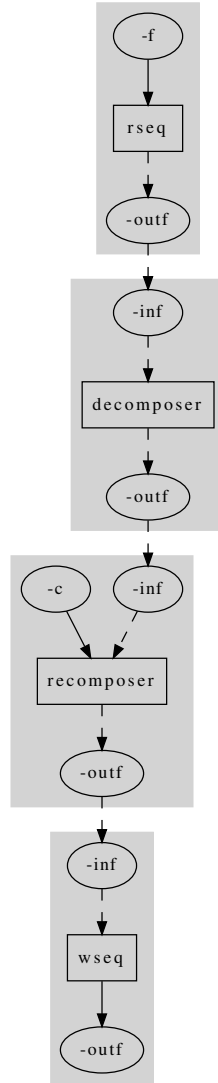


Figure S5: Process graph for the original formulation of the Telegram problem in the context of FBP.

Figure S6 shows the DeBasher code implementing the processes mentioned above for the original implementation of the Telegram problem from an FBP perspective. It is worthy of note that the **decomposer** and **recomposer** code that was shown in Figure S4 for the standard Bash implementation is used here without any modification.

On the other hand, Figure S7 shows how the options for each process are defined. The **define_opt_from_proc_out** function plays a fundamental role to connect a given process with another one. Dashed lines denote a connection between processes made by means of a pipeline.

Finally, once the different processes with their respective options have been defined, combining them into a program is straightforward, as it is shown in Figure S8.

```

1 rseq()
2 {
3     # Initialize variables
4     local outf=$(read_opt_value_from_func_args "-outf" "$@")
5     local file=$(read_opt_value_from_func_args "-f" "$@")
6
7     # Read sequence
8     cat "${file}" > "${outf}" || return 1
9 }
10
11 decomposer()
12 {
13     # Initialize variables
14     local inf=$(read_opt_value_from_func_args "-inf" "$@")
15     local outf=$(read_opt_value_from_func_args "-outf" "$@")
16
17     # Decompose input
18     awk '{for(i=1;i<=NF;++i) print $i}' "${inf}" > "${outf}" ; pipe_fail || return 1
19 }
20
21 recompose()
22 {
23     local char_lim=$1
24     local file=$2
25
26     awk -v char_lim="${char_lim}" 'BEGIN{len=0}
27     {
28         if(len + length($0) <= char_lim)
29         {
30             if(len > 0) printf " "
31             printf "%s", $0
32             len = len + length($0)
33         }
34         else
35         {
36             printf"\n%s", $0
37             len = length($0)
38         }
39         if(len+1 <= char_lim)
40             len = len + 1
41     }' "${file}"
42 }
43
44 recomposer()
45 {
46     # Initialize variables
47     local char_lim=$(read_opt_value_from_func_args "-c" "$@")
48     local inf=$(read_opt_value_from_func_args "-inf" "$@")
49     local outf=$(read_opt_value_from_func_args "-outf" "$@")
50
51     # Recompose input
52     recompose "${char_lim}" "${inf}" > "${outf}" ; pipe_fail || return 1
53 }
54
55 wseq()
56 {
57     # Initialize variables
58     local outf=$(read_opt_value_from_func_args "-outf" "$@")
59     local inf=$(read_opt_value_from_func_args "-inf" "$@")
60
61     # Write sequence
62     cat "${inf}" > "${outf}" || return 1
63 }

```

Figure S6: Process implementation for the original formulation of the Telegram problem in the context of FBP.

```

1 rseq_define_opts()
2 {
3     # Initialize variables
4     local cmdline=$1
5     local optlist=""
6
7     # Define option for rseq FIFO
8     local fifoname="rseq_fifo"
9     define_fifo_opt "-outf" "${fifoname}" optlist || return 1
10
11     # -f option
12     define_cmdline_opt "$cmdline" "-f" optlist || return 1
13
14     # Save option list
15     save_opt_list optlist
16 }
17
18 decomposer_define_opts()
19 {
20     # Initialize variables
21     local cmdline=$1
22     local optlist=""
23
24     # -inf option
25     define_opt_from_proc_out "-inf" "rseq" "-outf" optlist || return 1
26
27     # Define option for decomposer FIFO
28     local fifoname="dc_fifo"
29     define_fifo_opt "-outf" "${fifoname}" optlist || return 1
30
31     # Save option list
32     save_opt_list optlist
33 }
34
35 recomposer_define_opts()
36 {
37     # Initialize variables
38     local cmdline=$1
39     local optlist=""
40
41     # -c option
42     define_cmdline_opt "$cmdline" "-c" optlist || return 1
43
44     # -inf option
45     define_opt_from_proc_out "-inf" "decomposer" "-outf" optlist || return 1
46
47     # Define option for decomposer FIFO
48     local fifoname="rc_fifo"
49     define_fifo_opt "-outf" "${fifoname}" optlist || return 1
50
51     # Save option list
52     save_opt_list optlist
53 }
54
55 wseq_define_opts()
56 {
57     # Initialize variables
58     local cmdline=$1
59     local process_spec=$2
60     local process_name=$3
61     local process_outdir=$4
62     local optlist=""
63
64     # Define name of output file
65     local outf="${process_outdir}/output.txt"
66     define_opt "-outf" "${outf}" optlist || return 1
67
68     # -inf option
69     define_opt_from_proc_out "-inf" "recomposer" "-outf" optlist || return 1
70
71     # Save option list
72     save_opt_list optlist
73 }

```

Figure S7: Option definition for the original formulation of the Telegram problem in the context of FBP.

```

1 debasher_telegram_program()
2 {
3     add_debasher_process "rseq" "cpus=1 mem=32 time=00:05:00"
4     add_debasher_process "decomposer" "cpus=1 mem=32 time=00:05:00"
5     add_debasher_process "recomposer" "cpus=1 mem=32 time=00:05:00"
6     add_debasher_process "wseq" "cpus=1 mem=32 time=00:05:00"
7 }

```

Figure S8: Program definition for the original formulation of the Telegram problem in the context of FBP.

4.2 Telegram Problem With Two Processes

The Telegram problem can be redefined using only two processes, the **decomposer** and the **recomposer** processes. This alternative is less modular than the one shown in the previous section, since the **rseq** and the **wseq** processes are no longer independent. However, this implementation strategy makes more sense for workflow managers that are not able to handle real pipelines (for instance, if regular files were used, **rseq** would just copy the input file into another one).

Figure S9 shows a process diagram, where the two processes that are defined are connected by means of a pipeline denoted by a dashed line.

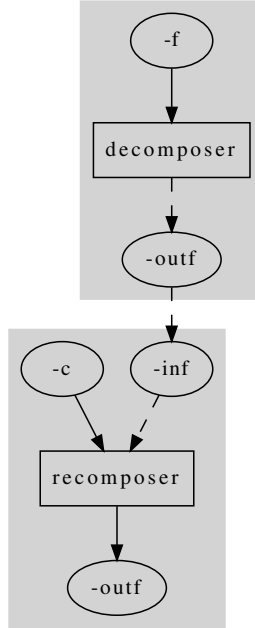


Figure S9: Process graph for the Telegram problem with two processes.

It is illustrative to show an alternative graph expressing the execution process dependencies. Given that both processes are connected by a pipeline, they will start execution simultaneously, which is reflected in the dependency graph shown in Figure S10.

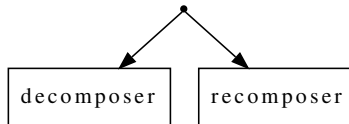


Figure S10: Dependency graph for the Telegram problem with two processes.

When implementing the Telegram problem with two processes using DeBasher, the previous process implementation given in Figure S6 needs no modification. Program definition is straightforward, since it is only necessary to add the **decomposer** and **recomposer** processes. What would need a more substantial modification would be the option definition code. This code can be seen in Figure S11.

```

1 decomposer_define_opts()
2 {
3     # Initialize variables
4     local cmdline=$1
5     local optlist=""
6
7     # Define option for decomposer FIFO
8     local fifoname="dc_fifo"
9     define_fifo_opt "-outf" "${fifoname}" optlist || return 1
10
11     # -f option
12     define_cmdline_opt "$cmdline" "-f" optlist || return 1
13
14     # Save option list
15     save_opt_list optlist
16 }
17
18 recomposer_define_opts()
19 {
20     # Initialize variables
21     local cmdline=$1
22     local process_spec=$2
23     local process_name=$3
24     local process_outdir=$4
25     local optlist=""
26
27     # Define name of output file
28     local outf="${process_outdir}/output.txt"
29     define_opt "-outf" "${outf}" optlist || return 1
30
31     # -c option
32     define_cmdline_opt "$cmdline" "-c" optlist || return 1
33
34     # Define option for decomposer FIFO
35     define_opt_from_proc_out "-inf" "decomposer" "-outf" optlist || return 1
36
37     # Save option list
38     save_opt_list optlist
39 }

```

Figure S11: Option definition for the Telegram problem with two processes.

4.3 Telegram Problem With Two Job Steps

The Telegram problem implementation with two processes given in the previous section can be modified so as to work with files instead of pipelines, resulting in a very similar process graph given in Figure S12, where the dashed line connecting the `decomposer` and `recomposer` processes is replaced by a continuous one.

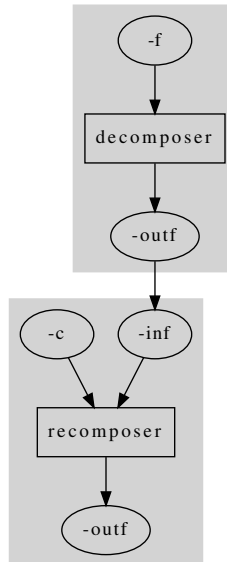


Figure S12: Process graph for the Telegram problem with two job steps.

When replacing the pipeline used to connect both processes by a regular file, the **recomposer** process can only start execution after the **decomposer** process has finished. This is graphically expressed in the dependency graph shown in Figure S13. The arc connecting the **decomposer** process to the **recomposer** process is labeled with **afterok** to specify that the latter will only start after the successful execution of the former.

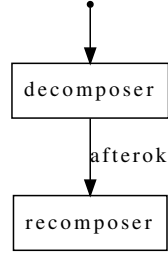


Figure S13: Dependency graph for the Telegram problem with two job steps.

The DeBasher implementation of the Telegram problem with two job steps can be made just by changing the code that defines the process options, as it is shown in Figure S14. This property of easily replacing processes by job steps is highlighted in [4] as one defining feature of FBP.

```

1 decomposer_define_opts()
2 {
3     # Initialize variables
4     local cmdline=$1
5     local process_spec=$2
6     local process_name=$3
7     local process_outdir=$4
8     local optlist=""
9
10    # -f option
11    define_cmdline_opt "$cmdline" "-f" optlist || return 1
12
13    # Define name of output file
14    local outf="{process_outdir}/words.txt"
15    define_opt "-outf" "${outf}" optlist || return 1
16
17    # Save option list
18    save_opt_list optlist
19 }
20
21 recomposer_define_opts()
22 {
23     # Initialize variables
24     local cmdline=$1
25     local process_spec=$2
26     local process_name=$3
27     local process_outdir=$4
28     local optlist=""
29
30    # -c option
31    define_cmdline_opt "$cmdline" "-c" optlist || return 1
32
33    # -inf option
34    define_opt_from_proc_out "-inf" "decomposer" "-outf" optlist || return 1
35
36    # Define name of output file
37    local outf="{process_outdir}/output.txt"
38    define_opt "-outf" "${outf}" optlist || return 1
39
40    # Save option list
41    save_opt_list optlist
42 }

```

Figure S14: Option definition for the Telegram problem with two job steps.

4.4 Array of Telegram Problems

The Telegram problem with two processes/job steps can be easily defined as an array, that is, we can execute an arbitrary number of **decomposer** and **recomposer** process pairs. Figure S15 shows the corresponding process graph, assuming that the array size is equal to 2.

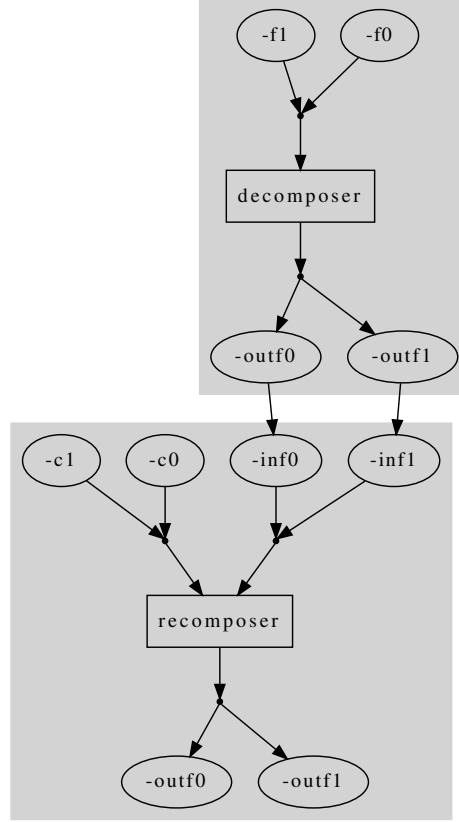


Figure S15: Process graph for an array of Telegram problems.

In order to implement an array of Telegram problems, it is only required to modify process option definition with respect to the initial implementation. Figure S16 shows how the `define_opts` method is redefined for the **decomposer** and **recomposer** processes. Specifically, the option lists for the different instances of the Telegram problem will be stored in an array. The code executes a `for` loop to populate these arrays.

```

1 decomposer_define_opts()
2 {
3     # Initialize variables
4     local cmdline=$1
5     local process_spec=$2
6     local process_name=$3
7     local process_outdir=$4
8     local optlist=""
9
10    # Obtain value of -f option
11    pref_of_files=$(get_cmdline_opt "${cmdline}" "-f")
12
13    # Process files
14    local i=0
15    for file in "${pref_of_files}"; do
16        local specific_optlist=${optlist}
17
18        # Define name of input file
19        define_opt "-f" "${file}" specific_optlist || return 1
20
21        # Define name of output file
22        local outf="${process_outdir}/words_${i}.txt"
23        define_opt "-outf" "${outf}" specific_optlist || return 1
24
25        # Save option list
26        save_opt_list specific_optlist
27
28        # Increase index
29        i=$((i + 1))
30    done
31 }
32
33 recomposer_define_opts()
34 {
35     # Initialize variables
36     local cmdline=$1
37     local process_spec=$2
38     local process_name=$3
39     local process_outdir=$4
40     local optlist=""
41
42    # Obtain value of -f option
43    pref_of_files=$(get_cmdline_opt "${cmdline}" "-f")
44
45    # -c option
46    define_cmdline_opt "$cmdline" "-c" optlist || return 1
47
48    # Process files
49    local i=0
50    for file in "${pref_of_files}"; do
51        local specific_optlist=${optlist}
52
53        # -inf option
54        define_opt_from_proc_task_out "-inf" "decomposer" "$i" "-outf" specific_optlist || return 1
55
56        # Define name of output file
57        local outf="${process_outdir}/output_${i}.txt"
58        define_opt "-outf" "${outf}" specific_optlist || return 1
59
60        # Save option list
61        save_opt_list specific_optlist
62
63        # Increase index
64        i=$((i + 1))
65    done
66 }

```

Figure S16: Option definition for an array of Telegram problems.

4.5 Imperative Telegram Problem

The Telegram problem can be implemented by means of DeBasher using only one process that internally calls the previously defined **decomposer** and **recomposer** processes in a purely imperative way. The corresponding process graph is shown in Figure S17, where only one process called **telegram** is defined.

The ability to execute processes in an imperative way from another one is exclusive to DeBasher and is not incorporated into other tools mentioned in this study.

The DeBasher implementation of the imperative Telegram problem requires defining the **telegram** process and its options, as it is shown in Figure S18. The **telegram** process calls the **decomposer** and **recomposer** processes imperatively by means of the **seq_execute** function (the name stands for *sequential execute*). The code of the called processes does not need to be redefined.

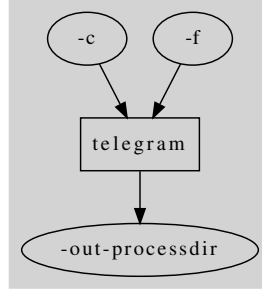


Figure S17: Process graph for the imperative Telegram problem.

```

1 telegram_define_opts()
2 {
3   # Initialize variables
4   local cmdline=$1
5   local process_spec=$2
6   local process_name=$3
7   local process_outdir=$4
8   local optlist=""
9
10  # Define the -out-processdir option, the output directory for the process
11  define_opt "-out-processdir" "${process_outdir}" optlist || return 1
12
13  # -f option
14  define_cmdline_opt "$cmdline" "-f" optlist || return 1
15
16  # -c option
17  define_cmdline_opt "$cmdline" "-c" optlist || return 1
18
19  # Save option list
20  save_opt_list optlist
21 }
22
23 telegram()
24 {
25   # Initialize variables
26   local outd=$(read_opt_value_from_func_args "-out-processdir" "$@")
27   local file=$(read_opt_value_from_func_args "-f" "$@")
28   local char_lim=$(read_opt_value_from_func_args "-c" "$@")
29
30   # Execute decomposer
31   seq_execute decomposer -f "${file}" -outf "${outd}"/words.txt || return 1
32
33   # Obtain number of lines of decomposer output
34   local nlines=$(wc -l "${outd}"/words.txt | awk '{print $1}')
35
36   if [ "${nlines}" -eq 0 ]; then
37     echo "Warning: Decomposer's output is empty" >&2
38     echo -n "${outd}"/output.txt || return 1
39   else
40     # Execute recomposer
41     seq_execute recomposer -c "${char_lim}" -inf "${outd}"/words.txt -outf "${outd}"/output.txt ||
42       return 1
43   fi
44 }

```

Figure S18: Option definition and process implementation for the imperative Telegram problem.

5 Telegram Problem Implementation with Other Tools

It is interesting to inspect how the Telegram problem can be implemented with other tools. All of the tools considered in this section are tools specialized in the execution of workflows. From all the Telegram problem versions described above, we will focus on the one using job steps described in Section 4.3, since the all of the tools used allow to implement it.

5.1 CWL

The Common Workflow Language³ (CWL), is not a particular tool but a language specification for which different implementations have been defined.

CWL distributes the code of the **decomposer** and **recomposer** processes in different files, as it is shown in Figure S19. CWL is very different with respect to other workflow engines because it only allows to execute one shell-script command per workflow step.

decomposer File

```
1 baseCommand: [awk, '{for(i=1;i<=NF;++i) print $i}']
2 stdout: words.txt
3
4 inputs:
5   input_file:
6     type: File
7   inputBinding:
8     position: 1
9
10 outputs:
11   decomposer_out:
12     type: stdout
```

recomposer File

```
1 baseCommand: [awk,-v]
2 stdout: output.txt
3
4 inputs:
5   char_lim:
6     type: int
7   inputBinding:
8     prefix: char_lim=
9     separate: false
10    position: 1
11
12 pattern:
13   type: string
14   default: 'BEGIN{len=0}
15     {
16       if(len + length($0) <= char_lim)
17       {
18         if(len > 0) printf " ";
19         printf "%s", $0;
20         len = len + length($0);
21       }
22       else
23       {
24         printf "\n%s", $0;
25         len = length($0);
26       }
27       if(len+1 <= char_lim)
28         len = len + 1;
29     }
30   inputBinding:
31     position: 2
32
33 words_file:
34   type: File
35   inputBinding:
36     position: 3
37
38 outputs:
39   recomposer_out:
40     type: stdout
```

Workflow File

```
1 inputs:
2   input_file: File
3   char_lim: int
4
5 steps:
6   decomposer:
7     run: telegram.decomposer.tool.cwl
8     in:
9       input_file: input_file
10      out: [decomposer_out]
11   recomposer:
12     run: telegram.recomposer.tool.cwl
13     in:
14       words_file: decomposer/decomposer_out
15       char_lim: char_lim
16      out: [recomposer_out]
17
18 outputs:
19   output:
20     type: File
21     outputSource: recomposer/recomposer_out
```

Figure S19: Implementation details of the Telegram problem with two processes using CWL.

³<https://www.commonwl.org/specification/>

5.2 WDL

The Workflow Description Language⁴ (WDL) is similar to CWL in the sense that is not a particular tool but a language specification.

WDL uses `command` blocks to incorporate shell-scripting code, as it can be seen in Figure S20.

```
1 task Decomposer {
2   File input_file
3
4   command <<<
5     awk '{for(i=1;i<=NF;++i) print $i}' ${input_file} > words.txt
6   >>>
7
8   output {
9     File words_file = "words.txt"
10  }
11 }
12
13 task Recomposer {
14   File input_file
15   Int char_lim
16
17   command <<<
18     awk -v maxlen=${char_lim} 'BEGIN{len=0}
19     {
20       if(len + length($0) <= char_lim)
21       {
22         if(len > 0) printf " ";
23         printf "%s", $0;
24         len = len + length($0);
25       }
26       else
27       {
28         printf "\n%s", $0;
29         len = length($0);
30       }
31       if(len+1 <= char_lim)
32         len = len + 1;
33     }' ${input_file} > output.txt
34   >>>
35
36   output {
37     File output_file = "output.txt"
38   }
39 }
40
41 workflow Telegram {
42   File input_file
43   Int char_lim
44
45   call Decomposer {
46     input:
47       input_file= input_file
48   }
49
50   call Recomposer {
51     input:
52       input_file= Decomposer.words_file,
53       char_lim= char_lim
54   }
55
56   output {
57     File output_file = Recomposer.output_file
58   }
59 }
```

Figure S20: Implementation details of the Telegram problem with two processes using WDL.

5.3 Nextflow

Nextflow [2] is a very successful and well known workflow manager using Groovy as a coordination language.

Figure S21 shows the implementation of both processes involved in the Telegram problem. Again there are specific blocks devoted to the execution of shell-script code.

⁴<https://openwdl.org/spec/>

```

1 process decomposer {
2   input:
3     path file
4
5   output:
6     path "decompose.out"
7
8   """
9   cat < ${file} | awk '{for(i=1;i<=NF;++i) print \${i}}' > decompose.out
10  """
11 }
12
13 process recomposer {
14   input:
15     path decompose_out
16
17   output:
18     path "recompose.out"
19
20   """
21   awk -v char_lim=${params.char_lim} 'BEGIN{len=0}
22   {
23     if(len + length(\$0) <= char_lim)
24     {
25       if(len > 0) printf " "
26       printf "%s", \$0
27       len = len + length(\$0)
28     }
29     else
30     {
31       printf "\\n%s", \$0
32       len = length(\$0)
33     }
34     if(len+1 <= char_lim)
35       len = len + 1
36   }' ${decompose_out} > recompose.out
37   """
38 }
39
40 workflow {
41   def ch = Channel.fromPath(params.file)
42   decompose(ch) | recompose
43 }

```

Figure S21: Implementation details of the Telegram problem with two processes using Nextflow.

5.4 Bpipe

Bpipe [5] is a platform to execute workflows which uses Groovy as the coordination language. In this regard, it is similar to Nextflow, and indeed, Nextflow was born as a branch of Bpipe according to the Bpipe creators.

Figure S22 shows the Bpipe code implementing both the `decomposer` and `recomposer` processes (or more accurately, job steps). In the code, `exec` blocks allows to execute shell-script code.

```

1 decomposer = {
2   exec """
3   cat < $input | awk '{for(i=1;i<=NF;++i) print \${i}}' > $output
4   """
5 }
6
7 recomposer = {
8   exec """
9   awk -v char_lim=${CHAR_LIM} 'BEGIN{len=0}
10  {
11    if(len + length(\$0) <= char_lim)
12    {
13      if(len > 0) printf " ";
14      printf "%s", \$0;
15      len = len + length(\$0);
16    }
17    else
18    {
19      printf "\\n%s", \$0;
20      len = length(\$0);
21    }
22    if(len+1 <= char_lim)
23      len = len + 1;
24  }' $input > $output
25  """
26 }
27
28 run {
29   decompose + recompose
30 }

```

Figure S22: Implementation details of the Telegram problem with two processes using Bpipe.

5.5 Snakemake

Snakemake [3] is a well established workflow execution tool. Snakemake follows a very different approach when compared to other tools such as Nextflow or Bpipe. In particular, it is inspired by the well known UNIX `make` tool. The `make` tool is typically used when compiling large C programs, resolving file dependencies and issuing commands to carry out the required recompilations. Snakemake follows a similar strategy to implement workflows composed of multiple processes.

Figure S23 shows the definition of the `decomposer` and `recomposer` processes using the Snakemake syntax. Again, the tool has a specific block to provide shell-script code.

```
1 rule decomposer:
2   input:
3     "data/data_1M.txt"
4   output:
5     "results/words.txt"
6   shell:
7     "cat {input} | awk '{{for(i=1;i<=NF;++i) \
8       {{print $i}} }}' > {output}"
9
10 rule recomposer:
11   input:
12     "results/words.txt"
13   output:
14     "results/output.txt"
15   shell:
16     "awk -v char_lim=20 '{{ \
17       current_line = current_line $0 \
18       if (length(current_line) < char_lim) \
19       current_line = current_line " " \
20       else {{ \
21         print current_line \
22         current_line = $0 \
23       }} \
24     }} \
25     END {{ \
26       if (length(current_line) > 0) \
27       print current_line \
28     }}' {input} > {output}"
```

Figure S23: Implementation details of the Telegram problem with two processes using Snakemake.

5.6 Toil

Toil [7] is a workflow engine that can operate in different ways, supporting CWL and WDL (see more details in Sections 5.1 and 5.2, respectively). However, Toil's native operation mode uses pure Python. This is reflected in Figure S24, where the code of the `decomposer` and `recomposer` processes is shown.

```
1 def decomposer(job, infid):
2     with job.fileStore.readGlobalFileStream(infid, encoding='utf-8') as fi:
3         with job.fileStore.writeGlobalFileStream(encoding='utf-8') as (fo, decfid):
4             for line in fi:
5                 line = line.strip()
6                 words = line.split()
7                 for word in words:
8                     print(word, file=fo)
9     return decfid
10
11 def recomposer(job, decfid, char_lim=20):
12     with job.fileStore.readGlobalFileStream(decfid, encoding='utf-8') as fi:
13         with job.fileStore.writeGlobalFileStream(encoding='utf-8') as (fo, recfid):
14             length = 0
15             for line in fi:
16                 word = line.strip()
17                 if (length + len(word) <= char_lim):
18                     if length > 0:
19                         print(" ", sep='', end='', file=fo)
20                         print(word, sep='', end='', file=fo)
21                         length = length + len(word)
22                 else:
23                     print("\n", word, sep='', end='', file=fo)
24                     length = len(word)
25                     if (length+1 <= char_lim):
26                         length = length + 1
27     return recfid
28
29 def telegram(job, infid, char_lim):
30     decfid = decomposer(job, infid)
31     j = job.addChildJobFn(recomposer, decfid, char_lim)
32     return j.rv()
```

Figure S24: Implementation details of the Telegram problem with two processes using Toil.

6 Programs with Arrays: a Code Reuse Example

DeBasher easily enables the definition of arrays of processes as we saw for the array of Telegram problems example in Section 4.4. In this section we show an alternative way to define a program incorporating an array of processes.

Let us consider a very simple program composed of only one process called `host1` that receives an identifier as input and executes the `hostname` UNIX command, printing the result to the standard output. We will refer to this program as `debasher_host_process`. Figure S25 shows the required DeBasher code.

```
1 host1_define_opts()
2 {
3     # Initialize variables
4     local optlist=""
5
6     # -id option
7     define_opt "-id" 0 optlist || return 1
8
9     # Save option list
10    save_opt_list optlist
11 }
12
13 host1()
14 {
15     # Initialize variables
16     local id=$(read_opt_value_from_func_args "-id" "$@")
17
18     # Show host name
19     local hname=$(hostname)
20     echo "${id}: ${hname}"
21 }
22
23 debasher_host_process_program()
24 {
25     add_debasher_process "host1" "cpus=1 mem=32 time=00:10:00 throttle=64"
26 }
```

Figure S25: Code of the `debasher_host_process` program.

In order to work with an array of `host1`, it is possible to modify the `define_opts` method defined above, so as to incorporate a `while` or `for` loop adding groups of options to the `optlist` variable (see Figure S16 for an example). Alternatively, we can also define the `generate_opts` method for the process, which returns the options corresponding to a given array index. When using `generate_opts`, it is also necessary to implement a method returning the array size called `generate_opts_size`.

Figure S26 shows the implementation of the `debasher_host_process` program, which executes an array of `host1` processes using the `generate_opts` method. Such method now defines the identifier to be passed to `host1` as the task index. The program uses the `load_debasher_module` function so as to load the `debasher_host_process` program that was described above. The program also executes the `add_debasher_program` function to execute `debasher_host_process` as a subprogram.

```

1 load_debasher_module "debasher_host_process"
2
3 host1_explain_cmdline_opts()
4 {
5     # -n option
6     local description="Number of array tasks"
7     explain_cmdline_req_opt "-n" "<int>" "$description"
8 }
9
10 host1_generate_opts_size()
11 {
12     # Initialize variables
13     local cmdline=$1
14
15     # -n option
16     local n_opt=$(get_cmdline_opt "$cmdline" "-n")
17
18     echo ${n_opt}
19 }
20
21 host1_generate_opts()
22 {
23     # Initialize variables
24     local cmdline=$1
25     local process_spec=$2
26     local process_name=$3
27     local process_outdir=$4
28     local task_idx=$5
29     local optlist=""
30
31     # -id option
32     define_opt "-id" "${task_idx} optlist || return 1
33
34     # Save option list
35     save_opt_list optlist
36 }
37
38 debasher_host_process_program()
39 {
40     add_debasher_program "debasher_host_process"
41 }

```

Figure S26: Code of the `debasher_host_process` program that executes a process array. Replacing a single process by an array does not require any modification in the `program` method.

7 Network Reconfiguration Example

This section shows how to reconfigure the connection pattern of a particular program. Let us define the `debasher_net_reconf_1` program composed of two processes: `process_a` and `process_b`. `process_a` takes a string as input and writes it to an output file. On the other hand, `process_b` takes the name of a file as input and prints its content to the standard input.

Figure S27 shows a graph for the `debasher_net_reconf_1` program, where the two processes are disconnected. The code related to the program is shown in Figure S28.

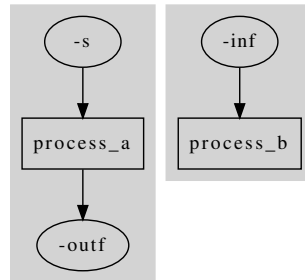


Figure S27: Process graph for the `debasher_net_reconf_1` program.

Now, let us suppose that we want to reconfigure the connection pattern of the `debasher_net_reconf_1` program. In particular, we need to connect the output of process `process_a` to the input of process `process_b`. We will refer to the new program as `debasher_net_reconf_2`, whose process graph is shown in Figure S29.

```

1 process_a_explain_cmdline_opts()
2 {
3     # -s option
4     local description="Input string"
5     explain_cmdline_req_opt "-s" "<string>" "$description"
6 }
7
8 process_a_define_opts()
9 {
10    # Initialize variables
11    local cmdline=$1
12    local process_spec=$2
13    local process_name=$3
14    local process_outdir=$4
15    local optlist=""
16
17    # -s option
18    define_cmdline_opt "$cmdline" "-s" optlist || return 1
19
20    # -outf option
21    define_opt "-outf" "${process_outdir}/outf" optlist || return 1
22
23    # Save option list
24    save_opt_list optlist
25 }
26
27 process_a()
28 {
29    # Initialize variables
30    local str=$(read_opt_value_from_func_args "-s" "$@")
31    local outf=$(read_opt_value_from_func_args "-outf" "$@")
32
33    # Print string to file
34    echo "${str}" > "${outf}"
35 }
36
37 process_b_explain_cmdline_opts()
38 {
39    # -inf option
40    local description="Input file"
41    explain_cmdline_req_opt "-inf" "<str>" "$description"
42 }
43
44 process_b_define_opts()
45 {
46    # Initialize variables
47    local cmdline=$1
48    local optlist=""
49
50    # -inf option
51    define_cmdline_opt "$cmdline" "-inf" optlist || return 1
52
53    # Save option list
54    save_opt_list optlist
55 }
56
57 process_b()
58 {
59    # Initialize variables
60    local inf=$(read_opt_value_from_func_args "-inf" "$@")
61
62    # Print content of input file to the standard output
63    cat "${inf}"
64 }
65
66 debasher_net_reconf_1_program()
67 {
68     add_debasher_process "process_a" "cpus=1 mem=32 time=00:10:00"
69     add_debasher_process "process_b" "cpus=1 mem=32 time=00:10:00"
70 }

```

Figure S28: DeBasher code for the `debasher_net_reconf_1` program.

Implementing the `debasher_net_reconf_2` program only requires to define a new module importing `debasher_net_reconf_1` and redefining the `define_opts` method for `process_b`. The resulting code is shown in Figure S30.

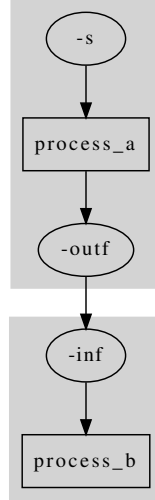


Figure S29: Process graph for the `debasher_net_reconf_2` program.

```

1 load_debasher_module "debasher_net_reconf_1"
2
3 process_b_explain_cmdline_opts()
4 {
5     :
6 }
7
8 process_b_define_opts()
9 {
10     # Initialize variables
11     local optlist=""
12
13     # -inf option
14     define_opt_from_proc_out "-inf" "process_a" "-outf" optlist || return 1
15
16     # Save option list
17     save_opt_list optlist
18 }
19
20 debasher_net_reconf_2_program()
21 {
22     add_debasher_program "debasher_net_reconf_1"
23 }

```

Figure S30: DeBasher code for the `debasher_net_reconf_2` program.

8 Programs with Cycles

DeBasher is the only tool among all those mentioned in this study that allows to implement programs with arbitrary cycles. This is due to the fact that FBP processes are in control of their life cycle, as opposed to the job steps implemented by other tools, which have a sequential nature and are executed reactively depending on the availability of the data they consume.

Figure S31 shows the graph of a program that contains a cycle. The program contains two processes, `master` and `worker`, where both are connected with pipelines.

Figure S32 shows the code of the processes. In particular, `master` uses a pipeline to send a numeric value to `worker`, and `worker` transforms the value, accumulates the result in an internal variable, and uses another pipeline to return the transformed value to `master`. More specifically, `worker` will add one to the value if the current cumulative value is below or equal to a threshold given by the user, or two otherwise. The cycle is repeated until `master` receives a value greater than another one given as input parameter. At the beginning of execution, `master` also receives the initial number for doing the calculations. Before terminating, `worker` prints the cumulative value to the standard output.

This simple example illustrates the ability of DeBasher to execute a program with cycles where one process maintains internal state information.

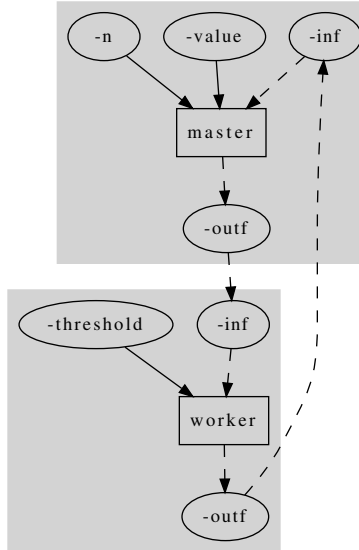


Figure S31: Process graph for a program with cycles.

```

1 master()
2 {
3     # Initialize variables
4     local n=$(read_opt_value_from_func_args "-n" "$@")
5     local value=$(read_opt_value_from_func_args "-value" "$@")
6     local inf=$(read_opt_value_from_func_args "-inf" "$@")
7     local outf=$(read_opt_value_from_func_args "-outf" "$@")
8
9     # Send value for transformation until is greater than n
10    while [ "${value}" -le "${n}" ]; do
11        echo "${value}" > "${outf}"
12        echo "Sent value ${value}"
13        value=$(cat "${inf}")
14        echo "Received value ${value}"
15        echo ""
16    done
17
18    # Send shutdown token
19    echo "${SHUTDOWN_TOKEN}" > "${outf}"
20 }
21
22 worker()
23 {
24     local threshold=$(read_opt_value_from_func_args "-threshold" "$@")
25     local inf=$(read_opt_value_from_func_args "-inf" "$@")
26     local outf=$(read_opt_value_from_func_args "-outf" "$@")
27
28     # Execute loop until the shutdown token is received
29     local sum=0
30     while true; do
31         value=$(cat "${inf}")
32         echo "Received value ${value}"
33         if [ "${value}" = "${SHUTDOWN_TOKEN}" ]; then
34             break
35         fi
36         # Decide value increment depending on threshold
37         if [ "${sum}" -le "${threshold}" ]; then
38             value=$((value + 1))
39         else
40             value=$((value + 2))
41         fi
42         # Update sum
43         sum=$((sum + value))
44
45         echo "Transformed value ${value}"
46         echo "Sum ${sum}"
47         echo ""
48
49         # Send current value
50         echo "${value}" > "${outf}"
51     done
52
53     echo "Final value of sum: ${sum}"
54 }

```

Figure S32: Process code of the debasher_cycle_state program.

9 Dynamic Scheduling Example

In the example of the previous section, we saw how to implement a program with cycles, where one process communicated with another to perform transformations of a number given as input. However, the calculations were done locally. DeBasher allows to schedule dynamic processes in HPC infrastructure so as to carry out computations.

To illustrate this ability, we use exactly the same program structure, but the functions that make the number transformations are executed in Slurm. The new program is called `debasher_cycle_dyn_sched`. Figure S33 shows the required code for the two processes. One of the functions used to make the calculations is now implemented in Python.

```
1 master()
2 {
3     # Initialize variables
4     local n=$(read_opt_value_from_func_args "-n" "$@")
5     local value=$(read_opt_value_from_func_args "-value" "$@")
6     local inf=$(read_opt_value_from_func_args "-inf" "$@")
7     local outf=$(read_opt_value_from_func_args "-outf" "$@")
8
9     # Send value for transformation until is greater than n
10    while [ "${value}" -le "${n}" ]; do
11        echo "${value}" > "${outf}"
12        echo "Sent value ${value}"
13        value=$(cat "${inf}")
14        echo "Received value ${value}"
15        echo ""
16    done
17
18    # Send shutdown token
19    echo "${SHUTDOWN_TOKEN}" > "${outf}"
20 }
21
22 transformation_a()
23 {
24     local value=$1
25     local outf=$2
26
27     value=$((value + 1))
28     echo ${value} > ${outf}
29 }
30
31 transformation_b_py=$(cat <<EOF
32 import sys
33 value = int(sys.argv[1])
34 fname = sys.argv[2]
35 with open(fname, 'w') as f:
36     f.write(str(value + 2))
37 EOF
38 )
39
40 worker()
41 {
42     local threshold=$(read_opt_value_from_func_args "-threshold" "$@")
43     local inf=$(read_opt_value_from_func_args "-inf" "$@")
44     local outf=$(read_opt_value_from_func_args "-outf" "$@")
45     local outd=$(read_opt_value_from_func_args "-outd" "$@")
46
47     # Execute loop until the shutdown token is received
48     local sum=0
49     while true; do
50         # Read value
51         local value=$(cat "${inf}")
52         echo "Received value ${value}"
53         if [ "${value}" = "${SHUTDOWN_TOKEN}" ]; then
54             break
55         fi
56
57         # Decide transformation function depending on threshold
58         if [ "${sum}" -le "${threshold}" ]; then
59             seq_execute_slurm transformation_a "${value}" "${outd}/transformation_result.txt"
60             value=$(cat "${outd}/transformation_result.txt")
61         else
62             seq_execute_slurm transformation_b_py "${value}" "${outd}/transformation_result.txt"
63             value=$(cat "${outd}/transformation_result.txt")
64         fi
65
66         # Update sum
67         sum=$((sum + value))
68
69         echo "Transformed value ${value}"
70         echo "Sum ${sum}"
71         echo ""
72
73         # Send current value
74         echo "${value}" > "${outf}"
75     done
76
77     echo "Final value of sum: ${sum}"
78 }
```

Figure S33: Code for the processes of the `debasher_cycle_dyn_sched` program.

10 User-Defined Triggers and Interactive Programs

The program in the previous section can be modified so as to receive the initial value for **master** and the threshold for **worker** by means of FIFOs. This opens the possibility of implementing user-defined triggers and program interactivity. We will refer to this program as the **debasher_cycle_trigger_interactive** program. Figure S35 shows a process graph for the program, and the new code for the processes is shown in Figure S34.

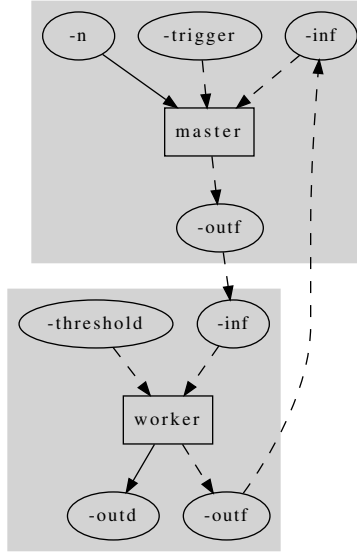


Figure S34: Process graph of a program with cycles with user-defined triggers and interactivity.

```

1 master()
2 {
3     # Initialize variables
4     local n=$(read_opt_value_from_func_args "-n" "$@")
5     local trigger_fifo=$(read_opt_value_from_func_args "--trigger" "$@")
6     local inf=$(read_opt_value_from_func_args "-inf" "$@")
7     local outf=$(read_opt_value_from_func_args "-outf" "$@")
8
9     # Set initial value
10    local initial_value=$(cat "${trigger_fifo}")
11
12    while [ "${initial_value}" != "${SHUTDOWN_TOKEN}" ]; do
13        # Send value for transformation until is equal or greater than n
14        local value="${initial_value}"
15        echo "* Initial value ${value}"
16        while [ "${value}" -le "${n}" ]; do
17            echo "${value}" > "${outf}"
18            echo "Sent value ${value}"
19            value=$(cat "${inf}")
20            echo "Received value ${value}"
21            echo ""
22        done
23
24        # Set initial value
25        local initial_value=$(cat "${trigger_fifo}")
26    done
27
28    # Send shutdown token
29    echo "${SHUTDOWN_TOKEN}" > "${outf}"
30 }
31
32 transformation_a()
33 {
34     local value=$1
35     local outf=$2
36
37     value=$((value + 1))
38     echo ${value} > ${outf}
39 }
40
41 transformation_b()
42 {
43     local value=$1
44     local outf=$2
45
46     value=$((value + 2))
47     echo ${value} > ${outf}
48 }
49
50 worker()
51 {
52     local threshold_fifo=$(read_opt_value_from_func_args "-threshold" "$@")
53     local inf=$(read_opt_value_from_func_args "-inf" "$@")
54     local outf=$(read_opt_value_from_func_args "-outf" "$@")
55     local outd=$(read_opt_value_from_func_args "-outd" "$@")
56
57     # Execute loop until the shutdown token is received
58     local sum=0
59     while true; do
60         # Read value
61         value=$(cat "${inf}")
62         echo "Received value ${value}"
63         if [ "${value}" = "${SHUTDOWN_TOKEN}" ]; then
64             break
65         fi
66
67         # Read threshold
68         local threshold=$(read_fifo_line "${threshold_fifo}")
69         echo "Threshold value ${threshold}"
70
71         # Decide transformation function depending on threshold
72         if [ "${sum}" -le "${threshold}" ]; then
73             seq_execute_slurm transformation_a "${value}" "${outd}/transformation_result.txt"
74             value=$(cat "${outd}/transformation_result.txt")
75         else
76             seq_execute_slurm transformation_b "${value}" "${outd}/transformation_result.txt"
77             value=$(cat "${outd}/transformation_result.txt")
78         fi
79
80         # Update sum
81         sum=$((sum + value))
82
83         echo "Transformed value ${value}"
84         echo "Sum ${sum}"
85         echo ""
86
87         # Send current value
88         echo "${value}" > "${outf}"
89     done
90
91     echo "Final value of sum: ${sum}"
92 }

```

Figure S35: Code for the processes of the debasher_cycle_trigger_interactive program.

11 Runtime Piping

Another interesting possibility is to connect two separate programs by using their FIFOs. The canonical example would be one program using a FIFO to produce its output, and another one using a FIFO to receive its input. Here we provide an example of such programs. For this purpose, we define the `debasher_counter`

program, which prints a count from 1 until another number given as input to an output FIFO. A process graph is given in Figure S36, and the process code can be seen in S37.

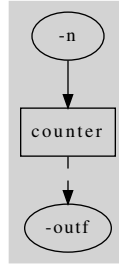


Figure S36: Process graph of a counter program.

```

1 counter_define_opts()
2 {
3   # Initialize variables
4   local cmdline=$1
5   local optlist=""
6
7   # -n option
8   define_cmdline_opt "$cmdline" "-n" optlist || return 1
9
10  # Define option for output FIFO
11  local fifoname="counter_out"
12  define_fifo_opt "-outf" "${fifoname}" optlist || return 1
13
14  # Save option list
15  save_opt_list optlist
16 }
17
18 counter()
19 {
20   # Initialize variables
21   local n=$(read_opt_value_from_func_args "-n" "$@")
22   local outf=$(read_opt_value_from_func_args "-outf" "$@")
23
24   # Increase value iteratively until it is greater than n
25   local value=1
26   while [ "${value}" -le "${n}" ]; do
27     echo "${value}" > "${outf}"
28     value=$((value + 1))
29   done
30
31   # Send shutdown token
32   echo "${SHUTDOWN_TOKEN}" > "${outf}"
33 }

```

Figure S37: Code related to the debasher_counter program.

On the other hand, we define the `debasher_echo` program, which receives a FIFO stream as input and prints it to the standard output. A process graph is shown in Figure S38 and the corresponding code is given in Figure S39.

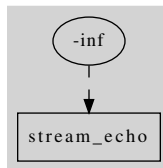


Figure S38: Process graph of an echo program.

Due to the fact that DeBasher exposes to the user all the FIFOs intervening in a program, it is possible to connect the two programs, for instance by using Bash commands. According to the process definitions, the names of the FIFOs to be connected would be `counter_out` for the `counter` process, and `stream_echo_in` for the `stream_echo` process.

```

1 stream_echo_define_opts()
2 {
3     # Initialize variables
4     local optlist=""
5
6     # Define option for input FIFO
7     local fifoname="stream_echo_in"
8     define_fifo_opt "-inf" "${fifoname}" optlist || return 1
9
10    # Save option list
11    save_opt_list optlist
12 }
13
14 stream_echo()
15 {
16     # Initialize variables
17     local inf=$(read_opt_value_from_func_args "-inf" "$@")
18
19     # Read input line by line
20     local line
21     while IFS= read -r line < "${inf}"; do
22         if [ "${line}" = "${SHUTDOWN_TOKEN}" ]; then
23             break
24         else
25             echo "${line}"
26         fi
27     done
28 }

```

Figure S39: Code related to the `debasher_echo` program.

12 Scalability Experiments

This section provides additional results to compare the ability to scale of the different tools tested in the study.

12.1 Experiments Configuration

The scalability experiments executed in this article are inspired in those reported in [1]. In that paper, a large set of instances of a one-step program (called `host_process`) or a two-step program (called `host_workflow`) is executed in an AWS Slurm ParallelCluster using different workflow managers.

The `host_process` program consists of a single process named `host1`, which internally runs the `hostname` UNIX command before terminating. Conversely, the `host_workflow` program executes both the `host1` and `host2` processes. Initially, it runs the `host1` process, identical to the one in the `host_process` program. Subsequently, it executes the `host2` process, which also runs the `hostname` command. For the rest of the experiments discussed in this section, we refer to a single execution of process `host1` or `host2` as a *workflow task*.

A DeBasher implementation of the `host_process` program has already been given in Section 6. However, our scalability experimentation is more focused on the two-step program. Figure S40 shows a process dependency diagram for the program, that we will call `debasher_host_workflow`. As it can be seen, the execution of `host2` depends on the previous execution of `host1`. During program execution, each workflow task (an instance of `host1` or `host2`) is assigned a numeric identifier, `id`. The arc connecting the two processes in Figure S40 is labeled with `aftercorr` to specify that the execution of the `host2` task with index `id` can start immediately after the `host1` task with the same index has successfully finished, that is, it has the same meaning as using the `aftercorr` dependency type in Slurm. The DeBasher implementation of both processes is shown in Figure S41. As it can be seen, both the `host1` and `host2` processes take an index as input and then execute the `hostname` UNIX command.

For our experiments, we implemented versions of the one- and two-step programs described above using CWL, WDL, Nextflow and DeBasher.

All the experiments were carried out in an AWS Slurm ParallelCluster with 1 head node and 32 computing nodes. Each node had 2 CPUs and 4GB of RAM. We used Toil [6] (v7.0.0) as execution engine for CWL, and Cromwell⁵ (v87) for WDL. Nextflow (v24.04) comes with its own engine.

⁵<https://github.com/broadinstitute/cromwell>

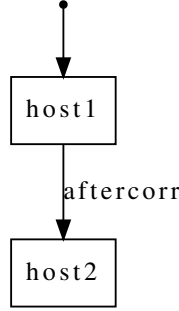


Figure S40: Process dependency graph for the `debasher_host_workflow` program.

```

1 host1()
2 {
3     # Initialize variables
4     local id=$(read_opt_value_from_func_args "-id" "$@")
5
6     # Show host name
7     local hname=$(hostname)
8     echo "${id}: ${hname}"
9 }
10
11 host2()
12 {
13     # Initialize variables
14     local id=$(read_opt_value_from_func_args "-id" "$@")
15
16     # Show host name
17     local hname=$(hostname)
18     echo "${id}: ${hname}"
19 }

```

Figure S41: Code for the `host1` and `host2` processes used in the `debasher_host_workflow` program.

12.2 Results

12.2.1 Overhead when Switching from Process to Workflow

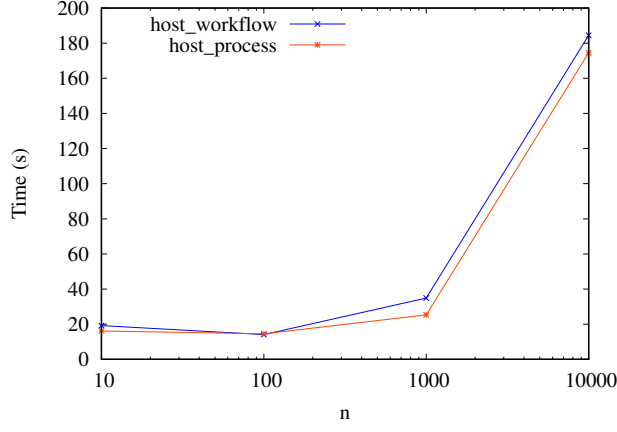
One of the experiments carried out in [1], consists in measuring the time cost of executing the same number of workflow tasks with both the one- and the two-step programs. This allows to determine the overhead introduced by the execution engine to handle process dependencies for the two-step program.

Figure S42 shows the time cost of Nextflow and DeBasher when executing an increasing number of tasks for the two programs being considered. As it can be seen, there is an almost perfect overlap between the time costs of both programs for DeBasher, meaning that it does not introduce any meaningful overhead due to process dependency handling. In contrast, Nextflow does introduce some overhead when transitioning from `host_process` to `host_workflow`.

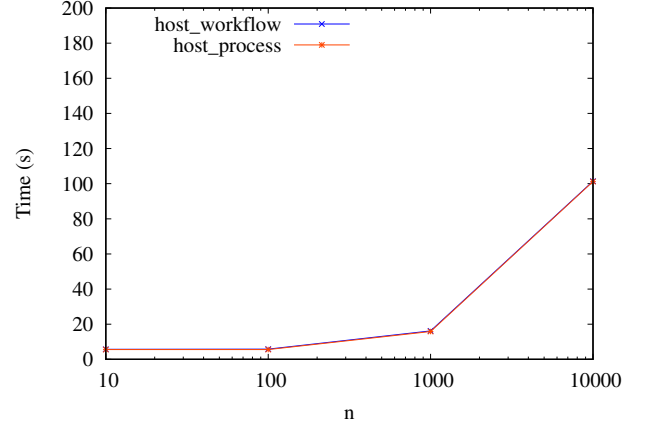
12.2.2 Task Distribution Across Nodes

The output of the `hostname` command for each task can be analyzed so as to get an idea of how well the tasks are distributed across the available cluster nodes. For this purpose, the output of all tasks is collected and the fraction of tasks executed at each node is computed. This fraction can be compared with the ideal one, where all tasks are equally distributed across nodes. Once the observed and ideal fractions of executed tasks have been calculated, we compute the mean absolute error (MAE) so as to get a quantitative measure of the deviation. A value of zero would mean that the ideal distribution was achieved.

Table S1 shows the MAE when executing `host_workflow` with 10K tasks by means of DeBasher and Nextflow. Nextflow was executed for increasing array sizes. As it can be seen in the table, DeBasher obtained the lowest MAE, indicating a better distribution of tasks across nodes with respect to Nextflow, no matter the size of the array chosen.



(a) Nextflow results.



(b) DeBasher results.

Figure S42: Time cost of Debasher and Nextflow when executing an increasing number of tasks using the one- and the two-step programs, called `host_process` and `host_workflow`, respectively,

	MAE
Nextflow (array size=1)	2.12e-02
Nextflow (array size=10)	1.18e-03
Nextflow (array size=100)	6.00e-04
Nextflow (array size=1000)	5.63e-04
Nextflow (array size=2000)	4.96e-04
Nextflow (array size=5000)	6.25e-04
DeBasher	4.59e-04

Table S1: Deviation from the ideal distribution of tasks across cluster nodes measured in terms of MAE for Nextflow (with various array sizes) and DeBasher when executing `host_workflow` with 10K tasks.

13 Sample Bioinformatics Workflow

DeBasher was used to implement a variant calling bioinformatics pipeline. The pipeline was run over two whole genome sequencing cancer datasets: the melanoma dataset MELA-AU (dataset ID: EGAD00001003388; 183 individuals) and the esophagus dataset ESAD-UK (dataset ID: EGAD00001003580; 303 individuals). Both datasets are available at <https://ega-archive.org/>.

Figures S43 and S44 show a process and a dependency graph, respectively, for the executed pipeline.

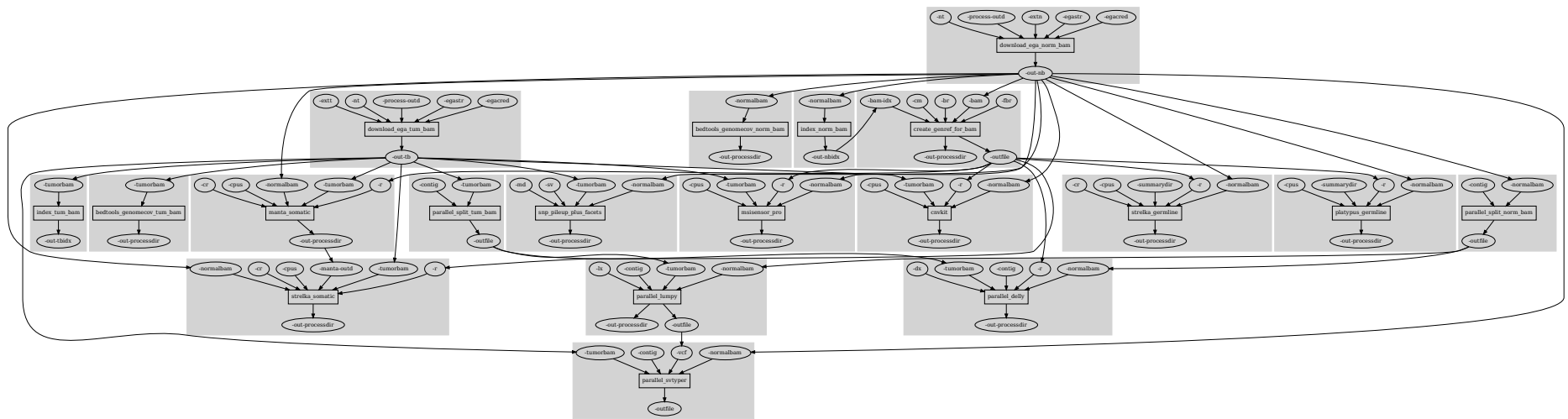


Figure S43: Process graph for a bioinformatics workflow implemented with DeBasher.

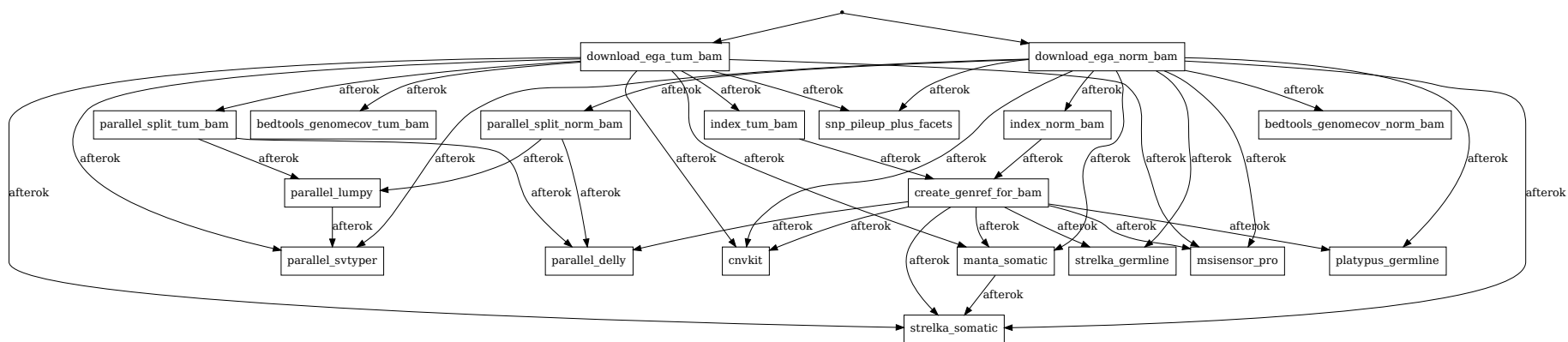


Figure S44: Dependency graph for a bioinformatics workflow implemented with DeBasher.

References

- [1] Azza E. Ahmed, Joshua M. Allen, Tajesvi Bhat, Prakruthi Burra, Christina E. Fliege, Steven N. Hart, Jacob R. Heldenbrand, Matthew E. Hudson, Dave Deandre Istanto, Michael T. Kalmbach, Gregory D. Kapraun, Katherine I. Kendig, Matthew Charles Kendzior, Eric W. Klee, Nate Mattson, Christian A. Ross, Sami M. Sharif, Ramshankar Venkatakrisnan, Faisal M. Fadlelmola, and Liudmila S. Mainzer. Design considerations for workflow management systems use in production genomics research and the clinic. *Scientific Reports*, 11(1):21680, Nov 2021.
- [2] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4):316–319, Apr 2017.
- [3] Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 08 2012.
- [4] J.P. Morrison. *Flow-based Programming: A New Approach to Application Development*. J.P. Morrison Enterprises, second edition, 2011.
- [5] Simon P. Sadedin, Bernard Pope, and Alicia Oshlack. Bpipe: a tool for running and managing bioinformatics pipelines. *Bioinformatics*, 28(11):1525–1526, 04 2012.
- [6] John Vivian et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, Apr 2017.
- [7] John Vivian, Arjun Arkal Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D. Deran, Audrey Musselman-Brown, Hannes Schmidt, Peter Amstutz, Brian Craft, Mary Goldman, Kate Rosenbloom, Melissa Cline, Brian O’Connor, Megan Hanna, Chet Birger, W. James Kent, David A. Patterson, Anthony D. Joseph, Jingchun Zhu, Sasha Zaranek, Gad Getz, David Haussler, and Benedict Paten. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology*, 35(4):314–316, Apr 2017.