Pipelined edge detection with Drake

Nicolas Melot

November 23, 2015

Abstract

In this document, we describe the structure of an edge detection application. We use this example to detail, step-by-step, the design of the edge detection application using the Drake framework. The algorithm is simplistic and there exist many better edge detection techniques. However, the example is particularly well adapted to streaming computation and the simplicity allows us to focus on pipelining with Drake instead of actually detecting edges. In this document, we often refer to streaming and pipeline applications to denote the same concept. Similarly, application and taskgraph description are synonymous in the document and refer to a detailed description of the pipeline used to build an executable binary. We finish by giving more hints to solve the second part of TDDD56 Lab3 on sorting with Drake.

1 Example application

We propose our edge detection application to take a color picture as input and output another black and white picture showing edges. Figure 1 gives an example of the application. The design our application is voluntarily simplified in order to make this example as simple as possible. In particular, we simplify the problem to pictures composed of a single continuous line of pixels in order to avoid issues related to false edge detection at pixels in the edge of the global picture frame. The algorithm we consider consists in 4 steps:

1. Convert the picture into levels of gray (*mono*)
2. Compute the derivative of the sequence of grayscale pixels (Δ)
3. Magnify luminosity changes detected by derivation (x^2)
4. Convert final number into grayscale color (*output*)

The algorithm takes as input a sequence of colored pixels shown in Fig. 2(a). The picture exhibits a sharp edge easy to detect, a very smooth edge that should not be detected as an edge and a third mid-size edge that cannot be detected by the algorithm proposed. Step 1 decomposes a pixel into red, green and blue colors and average them to obtain a grayscale equivalent. Figure 3(a) demonstrate the value of each grayscale pixel produced by this step; we can see that converting colors into monochrome with

(a) A random still life picture

(b) Edges detected by the "Difference of Gaussian" gimp filter.

Figure 1: An edge detection example. Picture from `http://www.public-domain-image.com/still-life/slides/still-life-picture-of-a-pumpkin-and-other-various-fruit-on-black-background.html`.

(a) A simple sequence of colors with sharp and soft edges.

(b) Sequence of colors after grayscale conversion.

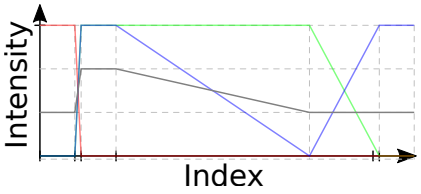
(c) Edges detected by our algorithm.

Figure 2: Input, intermediate and output "picture" of our edge detection algorithm.

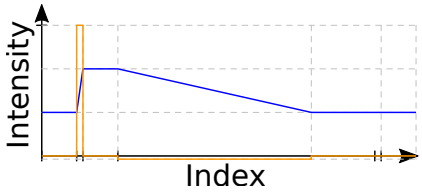
our simple averaging method makes the third gradient to disappear. Step 2 computes the derivative of the grayscale curve computed in Step 3(a). Figure 3(b) shows the resulting curve; we can already see where are the regions that correspond to a more or less sharp edge. The first edge is indeed sharp an the second one is very weak. Finally, we can apply Step 3 to increase sharp edges and decrease the weak ones (see Fig. 3(c)). Finally, Step 4 converts numbers computed in Step 3 into grayscale colors to obtain a picture as the one shown in Fig. 2(c).

2 Implementation and parallelization

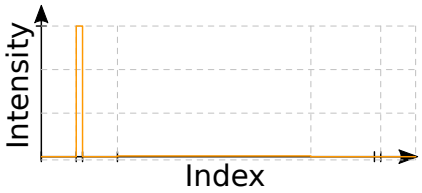
We can implement our algorithm sequentially with a function for each step. Listing 1 gives a sequential implementation in C of our simple edge detection algorithm. The



(a) Red, green and blue components of colors of Fig. 2(a) and corresponding average grayscale values.



(b) Derivative of the grayscale curve for Fig. 2(a).



(c) Corresponding edge values for Fig. 2(a).

Figure 3: Successive transformation of Steps 1 through 3 of our edge detection algorithm.

sequential loop runs can be unrolled as (time step from top to bottom):

mono(red[0], green[0], blue[0])
delta(gray[0])
$x^2(deriv[0])$
output(magn[0])
mono(red[1], green[1], blue[1])
delta(gray[1])
$x^2(deriv[1])$
output(magn[1])
mono(red[2], green[2], blue[2])
delta(gray[2])
$x^2(deriv[2])$
output(magn[2])
mono(red[3], green[3], blue[3])
delta(gray[3])
$x^2(deriv[3])$
output(magn[3])
mono(red[4], green[4], blue[4])
delta(gray[4])
$x^2(deriv[4])$
output(magn[4]).

Because of the loop-carried dependency (static variable in function Δ), it is hard to parallelize it by simply dividing the picture into smaller pieces and process them in parallel.

Listing 1: Sequential implementation of our edge detection algorithm.

```
1  float mono(char red, char, green, char, blue) {
2    return (red + green + blue) / 3;
3  }
4  float delta(char gray) {
5    // static var values persist accross calls
6    static char first_run = 1;
7    static char last = 0;
8    if(first_run == 0)
9      return (gray - last) / 255;
10   first_run = 0;
11   last = gray;
12   return 0;
13 }
14 float xsquare(float derivative) {
15   return derivative * derivative;
16 }
17 float output(float edge) {
18   return edge * 255;
19 }
20 int main() {
21   // Load pixel with red, green and blue channels
         separately
22   char *red = ...
23   char *green = ...
24   char *blue = ...
25   size_t size = ... // number of pixels
26   for(int i = 0; i < size; i++) {
27     gray[i] = mono(red[i], green[i], blue[i]);
28     deriv[i] = delta(gray[i]);
29     magn[i] = xsquare(deriv[i]);
30     out[i] = output(magn[i]);
31   }
32 }
```

We can take profit of pipeline parallelism: we can have a core to run *output* function to output the first pixel, while another processes *xsquare* for the second pixel, a third core runs Δ to compute the derivative at the third pixel and a fourth core converts the fourth pixel into grayscale. If we have 4 cores and make each of them run one step and forward intermediate data to the next core, then we can have the pipelined execution

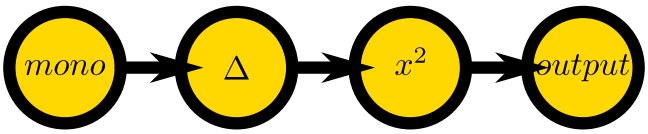


Figure 4: Four tasks and communication channels of the streaming edge detection application.

unrolled as (time step from top to bottom, cores from left to right):

mono(red[0], green[0], blue[0])				Warm-up
mono(red[1], green[1], blue[1])	*delta(gray[0])*			
mono(red[2], green[2], blue[2])	*delta(gray[1])*	$x^2(deriv[0])$		
mono(red[3], green[3], blue[3])	*delta(gray[2])*	$x^2(deriv[1])$	*output(magn[0])*	Steady-state
mono(red[4], green[4], blue[4])	*delta(gray[3])*	$x^2(deriv[2])$	*output(magn[1])*	
	delta(gray[4])	$x^2(deriv[3])$	*output(magn[2])*	Flush
		$x^2(deriv[4])$	*output(magn[3])*	
			output(magn[4]),	

therefore obtaining speedup. The warm-up phase progressively fills the software pipeline with data. Until this phase is done, the processor's activity pattern is not stable, i.e., the pattern doesn't repeat itself across cores or through time. During the steady-state period, the pattern repeats itelf. Here, every core runs a task (none is ever inactive) and the pattern is repeated at each time step. Finally, in the flushing phase, the activity pattern does no long repeat and as the last data to be processed progresses toward the exit of the pipeline, more and more cores become inactive.

3 Edge detection with Drake

In this section, we give details on the steps to follow to implement our edge detection application with Drake. We explain the concept of each step and we give examples of the steps' translation it into Drake's framework.

3.1 Define the application structure

In Sec. 1, we give a high-level description of an edge detection algorithm. We propose 4 steps: *mono*, Δ, x^2 and *output*; each step forwards intermediate data from one another, in order of step number. Each step can be translated into a *task*, the unit of execution of a streaming application and in Drake. Figure 4 gives a graphic representation of the pipelined application with its tasks and communication channels between them. We need to give a detailed description of all tasks in the application. This includes the amount of work the task performs in relation with other tasks, its degree of parallelism (or "maximum width", the maximum number of threads the task can take profit of),

Name	Module	Workload	Max. width	Efficiency
mono	mono	3	1	$p = 1?1 : 0$
Δ	delta	2	1	$p = 1?1 : 0$
x^2	xsquare	1	1	$p = 1?1 : 0$
output	output	1	1	$p = 1?1 : 0$

Table 1: Detailed description of the fours tasks of the pipelined edge detection application.

	mono	Δ	x^2	*output*
mono	0	1	0	0
Δ	0	0	1	0
x^2	0	0	0	1
output	0	0	0	0

Table 2: Detailed description of the fours tasks of the pipelined edge detection application.

its parallel efficiency that models the workload penalty due to parallelization, such as the necessary synchronization between threads, as a function of number of threads. Finally, the description include the task's *module*, i.e., the source file containing the task's code. A task's workload can be estimated in various way, including microbenchmarking or complexity analysis, obtaining a value for n input elements. We replace n by the number of elements the task processes, in relation with the number of elements processed by other tasks. Here, since all tasks process the same number of elements (each pixel of the input picture), $n = 1$ for all tasks. Additionally, task *mono* performs 2 additions and 1 division; its total workload is therefore 3. Although additions can be run in parallel, we assume here that the task is sequential only[1]; therefore its maximal width is 1 and its parallel efficiency is 1 (no penalty) when using one core and 0 (infinite penalty) when using any other number of cores. Finally, we implement the code of task *mono* in module "mono", that Drake finds in the C source file *mono.c*. Table 1 gives detailed information about all tasks in our edge detection application. We also need to define communication between tasks. *mono* sends intermediate data to Δ, Δ processes and forwards it to x^2 and x^2 sends data to *output* for the final color conversion and storage of final values. The communication matrix is summed up in Table 2.

3.1.1 Describe application with Drake

This section describes how one can describe a streaming application with Drake. We take the example of the pipelined edge detection algorithm described in Sec. 1 and whose detailed task description is given in Sec. 3.1. Listing 2 shows a portion of information given in Sec. 3.1, in the XML-based graph description language GraphML; the complete description file is available in appendix A. The graph key *name* (line 1) holds the name of the application. *deadline* denotes the time in millisecond within

[1] Another reason is that Drake doesn't support parallel tasks in its current implementation.

which all tasks of the application must run; here, it is set to 0 to disble any real-time constraint. In line 3 begins the description of the *mono* task. Drake models a task as a node in a graph. GraphML restricts node to have for identifier nX where X is a number starting at 0. We can find there the same information as in Table 1, i.e. the task's name, module, workload, parallel degree and parallel efficiency. here, parallel efficiency formulas are parsed with exprtk. Listing 2 also shows the modeling of Δ task and lines 17 and 18 describe the directed edges between *mono* and Δ to model the intermediate data forwarded between both tasks and also described in table 2.

Listing 2: Detailed information on the edge detection application in GraphML.

```
1   <data key="g_name">edge_detection</data>
2   <data key="g_deadline">0</data>
3   <node id="n0">
4     <data key="v_name">mono</data>
5     <data key="v_module">mono</data>
6     <data key="v_workload">3</data>
7     <data key="v_max_width">1</data>
8     <data key="v_efficiency">exprtk: p == 1 ? 1 : 1e-06
          </data>
9   </node>
10  <node id="n1">
11    <data key="v_name">delta</data>
12    <data key="v_module">delta</data>
13    <data key="v_workload">2</data>
14    <data key="v_max_width">1</data>
15    <data key="v_efficiency">exprtk: p == 1 ? 1 : 1e-06
          </data>
16  </node>
17  <edge source="n0" target="n1"/>
18  <edge source="n1" target="n2"/>
```

3.1.2 Source code of a Drake module

We show the C implementation of a Drake module in Listing 3. We take as example the task *mono* introduced in Sec. 1 and 3.1. A Drake module cannot have global variables unless they are delcared as *static*. A module must implement 5 functions: *drake_init*, *drake_start*, *drake_run*, *drake_kill* and *drake_destroy*. *drake_run* is the most important function as it holds the code the task runs to process its input and produce output data. *drake_init* and *drake_destroy* allow for task initialization and destruction before and after Drake builds or destroys the whole pipelined application; it is the best place to allocate memory, read input data or free resources. Argument *aux* in line 1 of Listing 3 is a pointer to some data forwarded from the main program to the task by Drake for initialization. *drake_start* and *drake_destroy* facilitate task bootstrapping and shutdown while the pipeline and other tasks may be running. In the case of *mono*, they do not provide any useful functionality. All functions must return 0 if an error occured, with the exceptions of *drake_start* and *drake_run*, that return 0 if they need to run again,

either to attempt bootstrapping again or because more computation is needed. In any other case, tasks must return a non-zero value. Drake doesn't automatically decide that a task needs to stop or not. Instead, the programmer decides, for example based on the availability of input data and on the current activity of predecessor tasks. Such information is available through the task argument of each function.

Listing 3: C implementation of task *mono* for Drake.

```
1  int drake_init(task_t *task, void* aux) {
2    struct channels *chan = (struct channels*)data;
3    red = chan->red;
4    green = chan->green;
5    blue = chan->blue;
6    length = data->length;
7    return 1;
8  }
9  int drake_start(task_t *task) {
10   return 1;
11 }
12 int drake_run(task_t *task) {
13   static int i = 0;
14   size_t size, j;
15   link_t *out = pelib_array_read(link_tp)(task->succ, 0);
16   int* addr = pelib_cfifo_writeaddr(int)(out->buffer, &
         size, NULL);
17   for(j = 0; i < length && j < size; i++, j++)
18     addr[j] = (red[i] + green[i] + blue[i]) / 3;
19   }
20   pelib_cfifo_fill(int)(out->buffer, j - 1);
21   return drake_task_depleted(task);
22 }
23 int drake_kill(task_t *task) {
24   return 1;
25 }
26 int drake_destroy(task_t *task) {
27   return 1;
28 }
```

A Drake task is a C structure that holds information about the task, such as its name, its internal Drake identifier or accesses to the task's input and output links from and toward other tasks. Listing 4 gives the members of Drake task and link structures that useful to the programmer. Other embers are for Drake to use only and are not shown in this document. Input and output links are accessible through *pred* and *succ* arrays of link, respectively. Consult the pelib doxygen documentation on generic arrays, or see more code examples for more information on how to read a link from a generic array of links. A link contains references to task structures that produce and consume the data the link conveys and the data itself is stored in a generic FIFO.

Finally, a task is always in on state within the *task_status* enumeration. If at task is in state *TASK_KILLED*, *TASK_ZOMBIE* or *TASK_DESTROY*, then it will not produce or consume any data anymore. Note that tasks' state is guaranteed to be propagated from producers to consumers only, but not in the other direction. Links and tasks structures make possible to determine if more data is available now for a task and if more data can be expected in input channels. This is facilitated by the Drake function *drake_task_depleted*, that returns 0 if more data is available from any of the task's input links or if any its predecessors did not reach any of the states *TASK_KILLED*, *TASK_ZOMBIE* or *TASK_DESTROY* yet.

Listing 4: Members of a Drake task structure.

```
1  enum task_status {TASK_INVALID, TASK_INIT, TASK_START,
       TASK_RUN, TASK_KILLED, TASK_ZOMBIE, TASK_DESTROY};
2  struct task {
3   char *name;                          Human-readable name of the task
4   array_t(link_tp) *succ;              Links toward successors of the task
5   array_t(link_tp) *pred;              Links toward predecessors of the task
6   task_status_t status;                Current state of the task
7  };
8  struct link {
9   struct task *prod;                   Task producing data into the link
10  struct task *cons;                   Task consuming data fromt the link
11  cfifo_t(int)* buffer;                FIFO structure holding the data conveyed
12 };
```

3.2 Describe the underlying architecture for Drake

In order to build a pipelining application binary, we need to specify details on the platform we intend to execute the program; in particular, we need to specify the number of cores available or the operating frequencies offered to the programmer. Drake uses information on the intended architecture to build the final executable binary. In particular, it makes it spawn as many threads as there are cores available. Listing 3.2 shows a the platform description file for Drake; the platform exhibits 4 cores, each of them able to run at 3.5GHz[2].

```
1  # Number of cores
2  param p := 4;
3
4  # Frequency levels
5  set F := 3500;
```

[2]In its current state, Drake for Intel processors doesn't support frequency switching, therefore any frequency information is actually ignored.

	Core 1	Core 2	Core 3	Core 4
Task	*mono*	Δ	x^2	*output*
Start time (ms)	0	0	0	0
Frequency	3.5 GHz	3.5GHz	3.5GHz	3.5GHz
Width	1	1	1	1
Workload	3	2	1	1

Table 3: Detailed description of the fours tasks of the pipelined edge detection application.

3.3 Define execution schedule

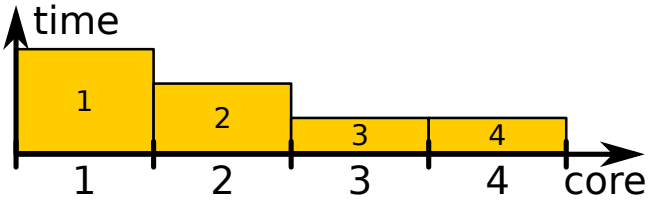
With the detailed streaming application (Sec. 3.1) and platform (Sec. 3.2) descriptions available, we can define the application static schedule. The static schedule defines, for each core, the sequence of tasks it executes in the steady-state of the pipeline. In the warming-up and flushing phases, tasks can be invoked the same way as in the steady state and exit immediately due to the lack of data to process. Assuming a platform of 4 cores and a unique frequency level described in Listing 3.2, the schedule for our edge detection application consists in 1 task per core, all tasks running at 3.5GHz. Table 3 summarizes the schedule. The schedule specifies the width and the workload of tasks in the schedule. This allows the work of tasks to be split in two smaller task instances with different parameters, such as frequency or number of cores. The width allows Drake to distinguish between a task running once on 2 cores and a task running twice on one core each, if it appears twice in the schedule.

It is crucial for performance to provide good schedule. A common way to optimize a schedule consists in spreading the workload as much as possible through the cores available (load-balancing), but other techniques also try to reduce communications between cores. The schedule of Table 3 can be visually represented as in Fig. 5(a).

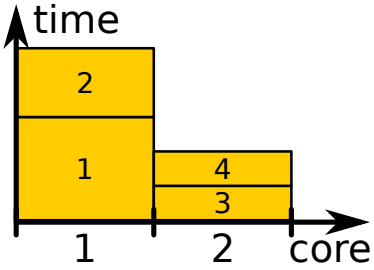
Figures 5(b) and 5(c) represent bad and good schedules, respectively. In Fig. 5(b), we can see that core 1 has much more work assigned that core 2. As the application runs for as long time as the core running for the longest time, and since the most loaded core in Fig. 5(c) is less loaded than the most loaded core in Fig 5(b), the schedule shown in Fig. 5(c) produces a faster final executable for 2 cores. Tables 4 and 5 show the detailed tables for the bad and good schedules of Fig. 5(b) and 5(c), respectively.

3.3.1 Automatic scheduling

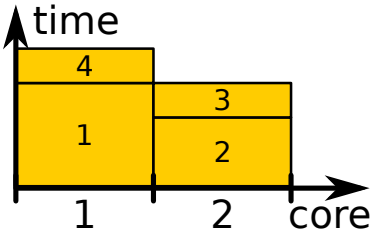
Scheduling sequential or parallel tasks in various settings is a widely research area. The classic algorithm *Longest Processing Time* (LPT) schedules sequential tasks into multiple cores, optimizing load-balancing. It produces solutions of quality $O(4/3 \cdot S^*)$ (i.e. at worst 4/3 times worse than the optimal solution). It works by sorting tasks in decreasing order of workload and distribute them one by one to the least loaded processor. Figure 6 shows the overall process.



(a) Visual representation of schedule in Table 3.



(b) A bad schedule for 2 cores.



(c) A better schedule for 2 cores.

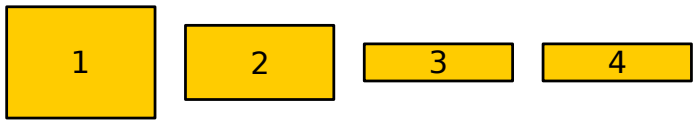
Figure 5: Visual representation of possible schedules for the edge detection application.

	Core 1	Core 2
Task	*mono*	x^2
Start time (ms)	0	0
Frequency	3.5 GHz	3.5GHz
Width	1	1
Workload	3	1
Task	Δ	*output*
Start time (ms)	$85.714 \cdot 10^{-6}$	$28.571 \cdot 10^{-6}$
Frequency	3.5GHz	3.5GHz
Width	1	1
Workload	2	1

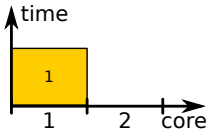
Table 4: Bad schedule of edge detection for 2 cores (assuming an instruction takes 1 cycle to execute).

	Core 1	Core 2
Task	*mono*	Δ
Start time (ms)	0	0
Frequency	3.5 GHz	3.5GHz
Width	1	1
Workload	3	2
Task	*output*	x^2
Start time (ms)	$85.714 \cdot 10^{-6}$	$57.143 \cdot 10^{-6}$
Frequency	3.5 GHz	3.5GHz
Width	1	1
Workload	3	1

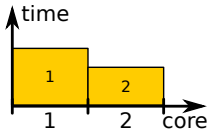
Table 5: Good schedule of edge detection for 2 cores (assuming an instruction takes 1 cycle to execute).



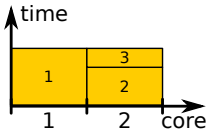
(a) Starting point of LPT: sorting task in drecreasing order of workload.



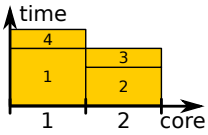
(b) Place task 1 to core 1 or 2 (here core 1).



(c) Place task 2 to less loaded core 2.



(d) Place task 3 to still less loaded core 2.



(e) Place task 4 to core 1 or 2 (here core 1).

Figure 6: Unrolling LPT for the edge detection application.

3.3.2 Schedules for Drake

Drakes uses XML to read streaming application's static schedule. Listing 3.3.2 shows the Drake XML schedule equivalent to the one shown in Table 3. We can see the sequence of tasks each core, delimited by the xml *core* markup and identified by an identification number starting at 1. We can see all task instance and details shown in Table 3.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schedule name="manual" appname="edge_detection">
3   <core coreid="1">
4    <task name="mono" start="0" frequency="3500" width="1"
         workload="3"/>
5   </core>
6   <core coreid="2">
7    <task name="delta" start="0" frequency="3500" width="1"
          workload="2"/>
8   </core>
9   <core coreid="3">
10   <task name="xsquare" start="0" frequency="3500" width="
        1" workload="1"/>
11  </core>
12  <core coreid="4">
13   <task name="output" start="0" frequency="3500" width="1
        " workload="1"/>
14  </core>
15 </schedule>
```

As LPT can compute a schedule automatically, we provide an implementation of LPT; Listing 5 shows how to generate an XML schedule from an application (taskraph) description in GraphML and a platform description and AMPL.

Listing 5: Command line to use our implementation of LPT.

```
:~\$ lpt --taskgraph /path/to/description.graphml --
    platform /path/to/platform.dat --schedule-output /path
    /to/output/schedule.xml
```

3.4 Build binary from a application and platform description, modules' source code and schedule with Drake

Sections 3.1 through 3.3 cover how to design and write various source code and description for Drake. This section gives how to combine them together and use Drake to build an executable that runs the pipeline. The easiest way to build a executable binary consists in filling fields in a suitable build script. Fig. 7 summarizes the process to build a streaming application with Drake. Jigsaw puzzle pieces shows components drake uses to build an application. They include a taskgraph description, a platform description, a schedule, modules implementations, a main program running the pipeline,

the drake framework and a drake platform backend. A typical programmer using Drake writes the taskgraph description, the main program and the task implementation (light blue areas in Fig. 7). The Drake platform backend implements platform specific operations, such as the allocation of communication memories and the transfers of data from a core to another. The Drake framework uses the platform backend and internal data structure to provide transparent task to task communication, regardless of the schedule.

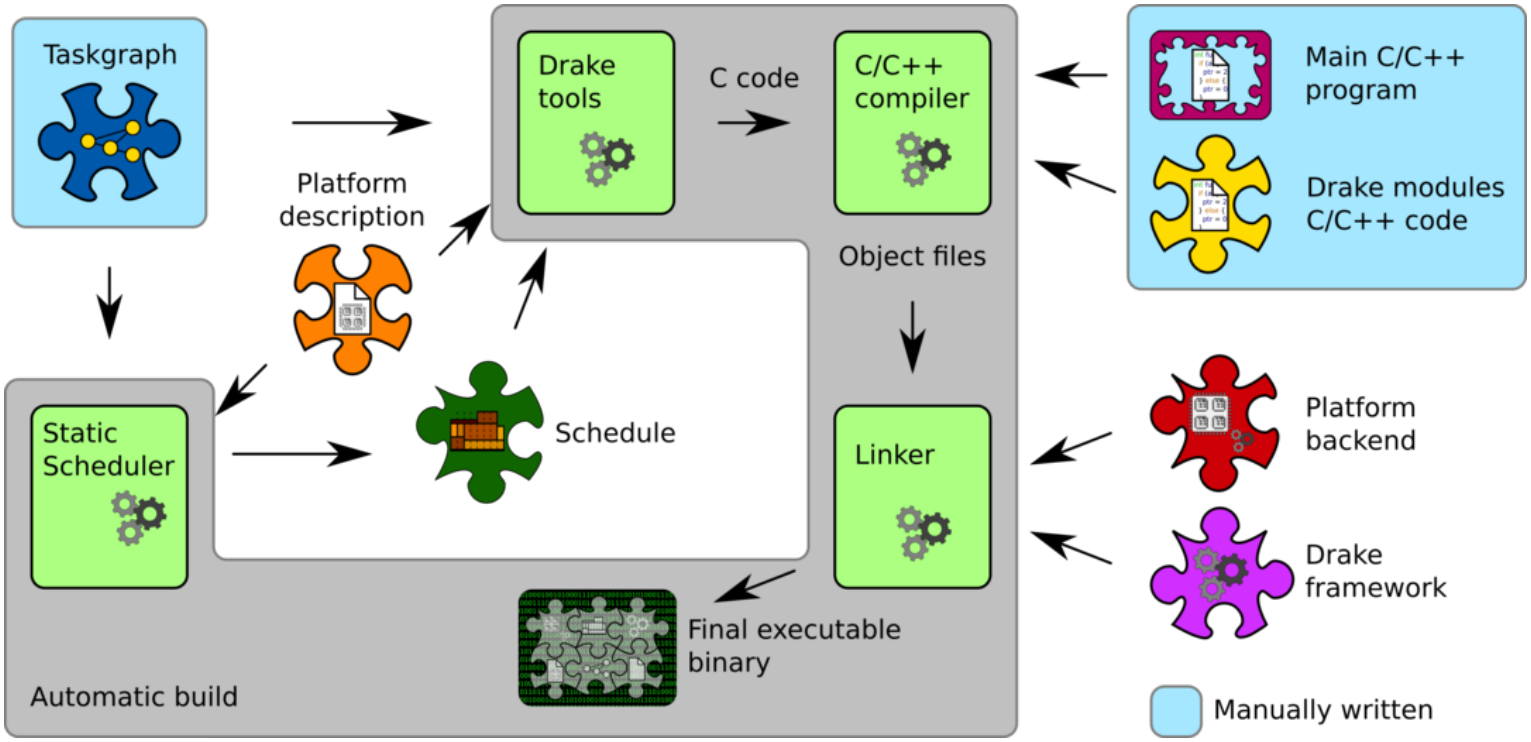


Figure 7: Overview of the application building process with Drake.

A Drake application stub includes a GNU Makefile that automatizes most of the building process. Listing 6 gives the portion of *src/Makefile.in*, that holds all information the building scripts need to build the final application. Lines 1 to 6 manage the final binary and lines 8 to 12 build a static library containing all the code generated by Drake. Line 1 gives the name of the final executable to be generated. The prefix *bincxx* indicates that the linker adds all C++ libraries to the final binary. Line 2 gives the list of the source files for the executable *edge*, compiled with a C compiler. All C source files names must end with the ".c" suffix. Line 3 expresses a dependency of program *edge* to the library descriptor *edge.pc*, that also depends on the library itself; this guarantees the final binary to be rebuilt upon any modification of any part of that library. The *edge* library contains all Drake modules and the code generated by Drake from the taskgraph description, the platform description and the schedule. Line 4 gives the system packages required to build the final application. We can see a reference to the Drake framework and the drake backend for Intel IA architectures. Line 6 and 7 are linking flags required by the linker to include the *edge* library (containing the Drake modules' code).

Line 8 gives the name of the library the script generates; here, it generates the static library *libedge.a*. The prefix *drake* indicates that the edge library must be built using the Drake process. Line 9 gives the file containing the taskgraph description, appendix A gives the complete file for the edge detection application described in Sec. 3.1. The prefix *drakecc* indicates that Drake looks for all files with a name given in the *module* field of a task description in the taskgraph description file (with a ".c" suffix), and compile them with a C compiler. Line 10 indicate the scheduler Drake uses to generate a schedule. This line is neutralized by line 11, that gives the file containing a schedule already computed. Finally, line 12 gives the file containing the platform description, for which Listing 3.2 gives an example.

Listing 6: Portion of GNU Makefile used to automatize the building of the Edge detection Drake application.

```
1  bincxx_package = edge
2  bincc_edge_src = edge.c
3  bin_edge_deps = edge.pc
4  bin_edge_pkg = drake drake-intel-ia
5  bin_edge_libs = -ledge
6  bin_edge_ldflags = -L.
7
8  drake_package = edge
9  drakecc_edge_graph = edge.graphml
10 drake_edge_scheduler = lpt
11 edge_schedule = edge.xml
12 drake_edge_platform = platform.dat
```

With all files set up as explained in Sec. 3.1 through 3.4, we can build the edge detection Drake application with the mere command:

```
:~\$ make
```

4 Pipelined mergesort with Drake

Section 3 gives technical information on how to use Drake to build a streaming application. The lab skeleton provides taskgraph and platform descriptions and the instructions describe the description files that match each phase of the experiment Freja manages. Section 3 of the lab instructions discusses how the initial merge tree is structured, how each function of the merge module are used and its implications on scheduling. All the building chain is provided fully functional. Platform descriptions initially define uniprocessor platforms; they are expected to be updated to reflect the number of cores to fit the experiment.

All taskgraph descriptions give the structure 4-levels merge tree for each number of cores Freja runs in the experiment, you need to define the taskgraph structure as it is shown in Sec. 3.1 of this document, and implement the *drake_run* function of the merge module. Think about how mergesort works, identify the tasks and how they communicate; the initial Drake taskgraph description give string suggestions. See the taskgraph descriptions to find the source file that contains the C source code of the module. The source file also contains extensive comments on the framework that provide information to implement the reading of input and the writing of output data for each task. It also gives details on how the module can obtain all the necessary information on the state of the pipeline and its tasks.

You are also required to create efficient schedules for each setting in the Freja experiment, either manually or by giving accurate data in the taskgraph description and use the LPT scheduler. Freja uses the building tools to generate executable binary versions intended for 1 to 6 cores and run them to produce the usual plots. You are required to show speedup for 1 to 4 cores, preferably to 6 cores if the platform allows it and particularly for 3 cores.

Appendix

A mono.c

This section gives a description of the edge application in GraphML format.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-
               instance"
4          xsi:schemaLocation="http://graphml.graphdrawing.
               org/xmlns
5 ..........http://graphml.graphdrawing.org/xmlns/1.0/
      graphml.xsd">
6 <!-- Created by igraph -->
7   <key id="g_name" for="graph" attr.name="name" attr.type
        ="string"/>
```

```
 8   <key id="g_deadline" for="graph" attr.name="deadline"
       attr.type="string"/>
 9   <key id="v_name" for="node" attr.name="name" attr.type=
       "string"/>
10   <key id="v_module" for="node" attr.name="module" attr.
       type="string"/>
11   <key id="v_workload" for="node" attr.name="workload"
       attr.type="double"/>
12   <key id="v_max_width" for="node" attr.name="max_width"
       attr.type="string"/>
13   <key id="v_efficiency" for="node" attr.name="efficiency
       " attr.type="string"/>
14   <graph id="G" edgedefault="directed">
15     <data key="g_name">edge_detection</data>
16     <data key="g_deadline">0</data>
17     <node id="n0">
18       <data key="v_name">mono</data>
19       <data key="v_module">mono</data>
20       <data key="v_workload">3</data>
21       <data key="v_max_width">1</data>
22       <data key="v_efficiency">exprtk: p == 1 ? 1 : 1e-06
           </data>
23     </node>
24     <node id="n1">
25       <data key="v_name">delta</data>
26       <data key="v_module">delta</data>
27       <data key="v_workload">2</data>
28       <data key="v_max_width">1</data>
29       <data key="v_efficiency">exprtk: p == 1 ? 1 : 1e-06
           </data>
30     </node>
31     <node id="n2">
32       <data key="v_name">x2</data>
33       <data key="v_module">xsquare</data>
34       <data key="v_workload">1</data>
35       <data key="v_max_width">1</data>
36       <data key="v_efficiency">exprtk: p == 1 ? 1 : 1e-06
           </data>
37     </node>
38     <node id="n3">
39       <data key="v_name">output</data>
40       <data key="v_module">output</data>
41       <data key="v_workload">1</data>
42       <data key="v_max_width">1</data>
43       <data key="v_efficiency">exprtk: p == 1 ? 1 : 1e-06
           </data>
```

```
44      </node>
45
46      <edge source="n0" target="n1"/>
47      <edge source="n1" target="n2"/>
48      <edge source="n2" target="n3"/>
49    </graph>
50  </graphml>
```

B mono.c

This section gives the C implementation of the Drake module "mono" in the edge detection application.

```
1  #include <drake.h>
2  #include <drake/link.h>
3
4  static char* red;
5  static char* green;
6  static char* blue;
7  static size_t length;
8
9  struct {
10   char *red, *green, *blue;
11   size_t length;
12 } channels;
13
14 int
15 drake_init(task_t *task, void* data)
16 {
17   struct channels *chan = (struct channels*)data;
18   red = chan->red;
19   green = chan->green;
20   blue = chan->blue;
21   length = data->length;
22
23   return 1;
24 }
25
26 int
27 drake_start(task_t *task)
28 {
29   return 1;
30 }
31
32 int
```

```
33 drake_run(task_t *task)
34 {
35   static int i = 0;
36   size_t size, j;
37   link_t *out = pelib_array_read(link_tp)(task->succ, 0);
38   int* addr = pelib_cfifo_writeaddr(int)(out->buffer, &
       size, NULL);
39   for(j = 0; i < length && j < size; i++, j++)
40   {
41     addr[j] = (red[i] + green[i] + blue[i]) / 3;
42   }
43   pelib_cfifo_fill(int)(out->buffer, j - 1);
44   return drake_task_depleted(task);
45 }
46
47 int
48 drake_kill(task_t *task)
49 {
50   return 1;
51 }
52
53 int
54 drake_destroy(task_t *task)
55 {
56   return 1;
57 }
```