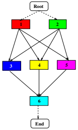
Simulating DAG Scheduling Algorithms with SimDAG

Frédéric Suter (CNRS, IN2P3 Computing Center, France)
Martin Quinson (Nancy University, France)
Arnaud Legrand (CNRS, Grenoble University, France)
Henri Casanova (Hawai'i University at Manoa, USA)

simgrid-dev@gforge.inria.fr

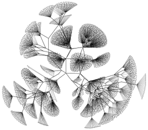
What is a DAG Scheduling Study?

X DAGs



x

Y platforms



x

Z Heuristics

```
For each task do
  Select resource
  Schedule task
end do
```

Agenda of this tutorial

1. How to describe DAGs?
2. How to describe resources?
3. How to write scheduling heuristics?

Objective

- Write a (simple) functional simulator step-by-step

Resources

- `http://simgrid.gforge.inria.fr/tutorials/simdag-101/exercises/`
- `http://simgrid.gforge.inria.fr/tutorials/simdag-101/solutions/`

Before Starting to Code Anything

- `#include "simdag/simdag.h"` is mandatory
- Start by initializing the SimDag stuff
- End by cleaning this stuff neatly

```
#include "simdag/simdag.h"

int main(int argc, char **argv){
  SD_init(&argc, argv);

  /* Insert your code here */

  SD_exit();

  return 0;
}
```

- Get this basic skeleton in `exercises/ex1-2_template.c`
- Open it with your favorite editor

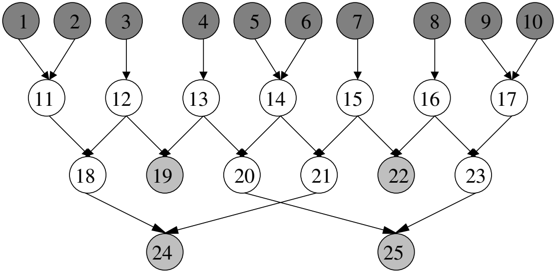
Agenda

- How to describe DAGs?
  - Definition
  - Manual Description
  - External Description and Automatic Load
- How to Describe Resources?
- How to Write Scheduling Heuristics?
- Conclusion

Definition of a DAG

Directed Acyclic Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

- $\mathcal{V} = \{v_i \mid i = 1, \ldots, V\}$
  - A set of vertices representing tasks
- $\mathcal{E} = \{e_{i,j} \mid (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$
  - A set of edges representing precedence constraints and/or data movements between tasks



Representing Vertices/Tasks

Sequential computation

- Use a `SD_task_t` of type `SD_TASK_COMP_SEQ`
- Constructor: `SD_task_create_comp_seq(name, data, amount)`
  - `name`: the name of the task, as given by the user
  - `data`: some user data attached to the task
    - Useful for scheduling attributes
  - `amount`: the number of flops computed by this task
- Destructor: `SD_task_destroy(task)`
- Can be used with any model compound handled by SURF
  - see `--help-models` for details

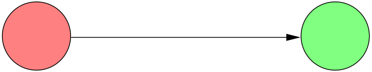
[Advanced] Parallel computation

- No type (`SD_TASK_NOT_TYPED`), default kind of `SD_task_t`
- Constructor: `SD_task_create(name, data, amount)`
  - `amount`: represents the sequential cost of the task
- Restricted to the `ptask_L07` model

Representing Edges/Dependencies

Control flow dependency

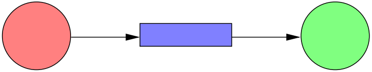
- a.k.a precedence constraint
- Goal: Force SimDAG to wait for the completion of ● to start ●
- Create a `SD_task_dependency`
  - `SD_task_dependency_add (name, data,●, ●)`



Representing Edges/Dependencies

Data flow dependency

- a.k.a passing data from a task to another
- Need to create a transfer ■ task between ● and ●
  - Add `SD_task_dependency` accordingly
  - `SD_task_dependency_add (name, data,●, ■)`
  - `SD_task_dependency_add (name, data,■, ●)`



Question

- How to declare a transfer task?

How to Represent Transfer Tasks

End-to-end communications

- If both source and destination are sequential tasks
- Use a task of type `SD_TASK_COMM_E2E`
- Constructor: `SD_task_create_comm_e2e(name, data, amount)`
  - `name`: the name of the task, as given by the user
  - `data`: some user data attached to the task
  - `amount`: the number of bytes transfered by this task

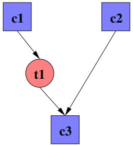
[Advanced] *MxN* data redistributions

- Same as for parallel computations
- No type (`SD_TASK_NOT_TYPED`)
- Constructor: `SD_task_create(name, data, amount)`
  - `amount`: represents the total number of bytes
  - Communication scheme defined at scheduling time
- Restricted to the `ptask_L07` model

Exercise 1

Describe the following graph

- Three sequential compute tasks
  - c1 computes 1e9 flops
  - c2 computes 5e9 flops
  - c3 computes 2e9 flops
- One end-to-end transfer task
  - t1 sends 5e8 bytes
- Don't forget the dependencies!
- Use template from `exercises/ex1-2_template.c`



Solution to Exercise 1

```
#include "simdag/simdag.h"

int main(int argc, char **argv) {
  SD_task_t c1, c2, c3, t1;
  SD_init(&argc, argv);

  c1 = SD_task_create_comp_seq("c1", NULL, 1E9);
  c2 = SD_task_create_comp_seq("c2", NULL, 5E9);
  c3 = SD_task_create_comp_seq("c3", NULL, 2E9);

  t1 = SD_task_create_comm_e2e("t1", NULL, 5e8);

  SD_task_dependency_add ("c1-t1", NULL, c1, t1);
  SD_task_dependency_add ("t1-c3", NULL, t1, c3);
  SD_task_dependency_add ("c2-c3", NULL, c2, c3);

  SD_exit();
  return 0;
}
```

- Get solution in `solutions/ex1-2.c`

Manage Your Set of Tasks

Use a Dynamic Array

- One out of the many useful data structures from the eXtended Bundle of Tools (XBT)
  - Requires `#include "xbt.h"`
- Dynars are dynamically sized vectors which may contain any type of variables
  - `xbt_dynar_t`
- Mandatory subset of functions
  - my_dynar = xbt_dynar_new (elm_size, free_method)
  - xbt_dynar_free_container (&my_dynar)
    - Free the dynar but not its content
  - xbt_dynar_push (my_dynar, &element)
  - xbt_dynar_pop(my_dynar, &element)
  - xbt_dynar_foreach(my_dynar, ctr, element)
    - Loop over elements in the array
  - xbt_dynar_length(my_dynar)
  - xbt_dynar_is_empty(my_dynar)
- For more information: `http://simgrid.gforge.inria.fr/simgrid/3.9/doc/group__XBT__dynar.html`

Retrieve Information on Tasks

Parameters of the constructor

- `SD_task_get_name(task)`
  - Can also be modified with `SD_task_set_name(task, "new_name")`
- `SD_task_get_data(task)`
  - Returns a `(void*)`, has to be casted by the user
  - Data can be attached at any time: `SD_task_set_data(task, (void*) data)`
- `SD_task_get_amount(task)` (non modifiable)
- `SD_task_get_kind(task)` (non modifiable)

Dependencies of task `T`

- Tasks on which `T` depends: `SD_task_get_parents(T)`
- Tasks depending on `T`: `SD_task_get_children(T)`
- Both functions return a `xbt_dynar_t`

Get everything

- `SD_task_dump(task)`

Exercise 2

- Get the solution of Exercise 1 in `exercises/ex1-2.c`
- Create a dynar of `SD_task_t`
- Push all the tasks in the dynar
- Browse the dynar
  - Dump information about each task
  - and destroy the task
- Destroy the dynar

Solution to Exercise 2

```
#include "simdag/simdag.h"
#include "xbt.h"

int main(int argc, char **argv) {
  SD_task_t c1, c2, c3, t1, tmp;
  unsigned int ctr;
  xbt_dynar_t tasks = xbt_dynar_new(sizeof(SD_task_t), &xbt_free);
  SD_init(&argc, argv);

  c1 = SD_task_create_comp_seq("c1", NULL, 1E9);
  c2 = SD_task_create_comp_seq("c2", NULL, 5E9);
  c3 = SD_task_create_comp_seq("c3", NULL, 2E9);
  t1 = SD_task_create_comm_e2e("t1", NULL, 5e8);
  SD_task_dependency_add ("c1-t1", NULL, c1, t1);
  SD_task_dependency_add ("t1-c3", NULL, t1, c3);
  SD_task_dependency_add ("c2-c3", NULL, c2, c3);

  xbt_dynar_push(tasks, &c1); xbt_dynar_push(tasks, &c2);
  xbt_dynar_push(tasks, &c3); xbt_dynar_push(tasks, &t1);

  xbt_dynar_foreach(tasks, ctr, tmp){
    SD_task_dump(tmp); SD_task_destroy(tmp);
  }
  xbt_dynar_free_container(&tasks);
  SD_exit();
  return 0;
}
```

- Get solution in solutions/ex1-2.c

SimDag Comes With Two Loaders

Common Features

- Creates all tasks and dependencies automatically
- Adds two special dummy tasks: `root` and `end`
- Returns a `xbt_dynar_t` of typed `SD_task_t`
  - `SD_TASK_COMP_SEQ` and `SD_TASK_COMM_E2E`

DAX format

- Format of workflow used by Pegasus (`http://pegasus.isi.edu/`)
- `SD_daxload(filename)`: loader for DAX files

DOT format

- Well-known format of the `graphviz` tool suite
- `SD_dotload(filename)`

DAX Format (1/2)

Header

- Name space and schema declaration (from Pegasus)
- Name of the DAX
- Number of jobs: `jobCount`
- Number of control dependencies: `childCount`

```
<?xml version="1.0" encoding="UTF-8"?>
<adag xmlns="http://pegasus.isi.edu/schema/DAX"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX
                          http://pegasus.isi.edu/schema/dax-2.1.xsd"
      version="2.1" count="1" index="0" name="smalldax"
      jobCount="3" fileCount="0" childCount="1">
...
```

DAX format (2/2)

Job description

- Described by: id, name, runtime, input and output files
  - Only computations are described ($\texttt{amount} = \texttt{runtime} \times 4.2e9$)
  - Output of task1 is a input of task2 ⇒ Transfer task + data flow dependency

```
<job id="1" namespace="SG" name="c1" version="1.0" runtime="10">
  <uses file="i1" link="input" register="true" transfer="true"
        optional="false" type="data" size="1000000"/>
  <uses file="o1" link="output" register="true" transfer="true"
        optional="false" type="data" size="1000000"/>
</job>
```

Control flow dependencies

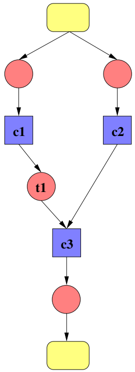
- task3 cannot start before the completion of task2
  - While there is no data flow dependency

```
<child ref="3">
  <parent ref="2"/>
</child>
```

Exercise 3

Describe the following graph in a DAX file

- Three sequential compute tasks
  - c1 runs for 10 seconds
    - Requires an input file of 2e8 bytes
    - Produces an output file of 5e8 bytes
  - c2 runs 50 seconds
    - Requires an input file of 1e8 bytes
  - c3 runs for 20 seconds
    - Requires an input file of 5e8 bytes
    - Produces an output file of 2e8 bytes
- Start from this skeleton:
  `exercises/ex3_template.xml`



Solution to Exercise 3

```
<?xml version="1.0" encoding="UTF-8"?>
<adag xmlns="http://pegasus.isi.edu/schema/DAX"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://pegasus.isi.edu/schema/DAX
                          http://pegasus.isi.edu/schema/dax-2.1.xsd"
      version="2.1" count="1" index="0" name="smalldax"
      jobCount="3" fileCount="0" childCount="1">
  <job id="1" namespace="SG" name="c1" version="1.0" runtime="10">
    <uses file="i1" link="input" register="true" transfer="true"
          optional="false" type="data" size="2e8"/>
    <uses file="o1" link="output" register="true" transfer="true"
          optional="false" type="data" size="5e8"/>
  </job>
  <job id="2" namespace="SG" name="c2" version="1.0" runtime="50">
    <uses file="i2" link="input" register="true" transfer="true"
          optional="false" type="data" size="1e8"/>
  </job>
  <job id="3" namespace="SG" name="c3" version="1.0" runtime="20">
    <uses file="o1" link="input" register="true" transfer="true"
          optional="false" type="data" size="5e8"/>
    <uses file="o3" link="output" register="true" transfer="true"
          optional="false" type="data" size="2e8"/>
  </job>
  <child ref="3">
    <parent ref="2"/>
  </child>
</adag>
```

- Get solution in `solutions/ex3.xml`

DOT Format

Task Description

- Described by an id and a size
  - The size correspond to the `amount` parameter of the task creator
  - Expressed in flops

Dependency Description

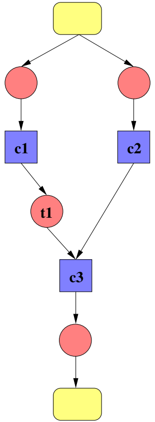
- Described by `src`→`dst` and a size
  - The size also corresponds to `amount`
  - A negative size indicates a control dependency
  - Expressed in bytes
- Dependencies from `root` and to `end` have to be explicit

```
digraph G {
   c1 [size="1e9"];
   root->c1 [size="1e8"];
   c1->end [[size="2e8"];
}
```

Exercise 4

Describe the following graph in a DOT file

- Three sequential compute tasks
  - c1 computes 1e9 flops
    - Requires an input file of 2e8 bytes
    - Produces an output file of 5e8 bytes
  - c2 computes 5e9 flops
    - Requires an input file of 1e8 bytes
  - c3 computes 2e9 flops
    - Requires an input file of 5e8 bytes
    - Produces an output file of 2e8 bytes
- Get Template in `exercises/ex4_template.dot`



Solution to Exercise 4

```
digraph G {
   c1 [size="1e9"];
   c2 [size="5e9"];
   c3 [size="2e9"];

   root->c1 [size="2e8"];
   root->c2 [size="1e8"];
   c1->c3   [size="5e8"];
   c2->c3   [size="-1."];
   c3->end  [size="2e8"];
}
```

- Get solution in `solutions/ex4.dot`

Exercise 5

Use the DAX (or DOT) loader (use a new source file)

- Call the loader
- Dump information of all tasks
- Destroy each task

Solution to Exercise 5

Exercise 5

Use the DAX (or DOT) loader (use a new source file)

- Call the loader
- Dump information of all tasks
- Destroy each task

Solution to Exercise 5

```
#include "simdag/simdag.h"
#include "xbt.h"

int main(int argc, char **argv) {
  unsigned int cpt;
  SD_task_t task;
  xbt_dynar_t dag;
  SD_init(&argc, argv);

  dag = SD_daxload(argv[1]);

  xbt_dynar_foreach(dag, cpt, task){
    SD_task_dump(task);
    SD_task_destroy(task);
  }
  SD_exit();
  return 0;
}
```

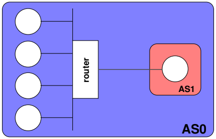
Agenda

- How to describe DAGs?
- How to Describe Resources?
  - Available Types of Resources
  - Creating an Experimental Environment
  - Attach User Data
- How to Write Scheduling Heuristics?
- Conclusion

Available Types of Resources

Types of resources

- Single Hosts: id and power
- Links: id, latency and bandwidth
- Clusters
  - id and name (prefix radical suffix)
  - power
  - private link latency (lat) and bandwidth (bw)
  - backbone latency (bb_lat) and bandwidth (bb_bw)
  - router

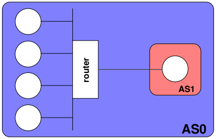


- routes: src and dst
- Resources grouped in Autonomous Systems (AS)
- Description in an XML platform file
- From a SimDag point of view:
  - A Host and (some) link(s) are grouped to form a workstation
  - SD_workstation_t data structure

Exercise 6

Describe the following platform

- One cluster
  - Named `my_cluster`
  - Four homogeneous hosts running at 4.2e9 flop/s
    - Named `c-x.me`, with x $\in [1-4]$
  - Four private links
    - Latency: 5e-5 seconds
    - Bandwidth: 1.25e8 bytes/s
  - One backbone
    - Latency: 5e-4 seconds
    - Bandwidth: 2.25e9 bytes/s
- One host in its own AS
  - Named `host1` and running at 4.2e9 flop/s
- One link connecting both ASes
  - Latency: 0.01 seconds
  - Bandwidth: 1e5 bytes/s
- Start from skeleton in `exercices/ex6_template.xml`



Solution to Exercise 6

cluster_and_one_host.xml

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
  <AS id="AS0" routing="Full">
    <cluster id="my_cluster" prefix="c-" suffix=".me" radical="1-4"
             power="4.2e9" bw="125000000" lat="5E-5"
             bb_bw="2250000000" bb_lat="5E-4"
             router_id="router1"/>

    <AS id="AS1" routing="Full">
      <host id="host1" power="1000000000"/>
    </AS>

    <link id="link1" bandwidth="100000" latency="0.01"/>

    <ASroute src="my_cluster" dst="AS1" gw_src="router1" gw_dst="host1">
      <link_ctn id="link1"/>
    </ASroute>
  </AS>
</platform>
```

- Get the platform file in solutions/ex6.xml

Creating the Environment

Loading the Platform File

- Use the `SD_create_environment` function
  - Takes a filename as input
  - Creates an array of `SD_workstation_t`
  - No need for deployment file in SimDag

Getting all the workstations

- Number of workstations: `SD_workstation_get_number()`
- Array of workstations: `SD_workstation_get_list()`

Workstation specific information

- `SD_workstation_get_name(workstation)` (non modifiable)
- `SD_workstation_get_power(workstation)` (non modifiable)
- `SD_workstation_get_data(workstation)`
  - Returns a `(void*)`, has to be casted by the user
  - Data can be attached at any time
    `SD_task_set_data(workstation, (void*) data)`
- Get everything: `SD_workstation_dump(workstation)`

Retrieving Information About Network

Getting all the links

- `SD_link_get_number()` returns the number of links
- `SD_link_get_list()` returns the list of links

Link specific information

- `SD_link_get_name(link)` (non modifiable)
- `SD_link_get_data(workstation)`
  - Returns a `(void*)`, has to be casted by the user
  - Data can be attached at any time: `SD_link_set_data(link, (void*) data)`
- `SD_link_get_sharing_policy (link)`
  - May this link cause contention or not

Route specific information

- `SD_route_get_size (src_workstation, dst_workstation)`
  - Returns the number of links on the route between two workstations
- `SD_route_get_list (src_workstation, dst_workstation)`
  - Returns the list of links on the route between two workstations

Exercise 7

- Use the source code of Exercise 5
- Load the platform file from Exercise 6
- Get the number of workstations
- Get the list of workstations
- Dump information about each workstation

Solution to Exercise 7

```
#include "simdag/simdag.h"
#include "xbt.h"

int main(int argc, char **argv) {
  unsigned int cpt;
  int nworkstations;
  const SD_workstation_t * workstations;
  SD_task_t task;
  xbt_dynar_t dag;
  SD_init(&argc, argv);

  dag = SD_daxload(argv[1]);

  xbt_dynar_foreach(dag, cpt, task)
    SD_task_dump(task);
  xbt_dynar_foreach(dag, cpt, task)
    SD_task_destroy(task);


  SD_create_environment(argv[2]);
  nworkstations = SD_workstation_get_number();
  workstations = SD_workstation_get_list();
  for (cpt = 0; cpt < nworkstations; cpt++)
    SD_workstation_dump(workstations[cpt]);

  SD_exit();
  return 0;
}
```

▶ Get solution in `solutions/ex5-7.c`

Attach User Data to Workstations

- This can be useful for scheduling
- Examples
  - When is a workstation available to execute a new task?
    - `double available_at /* a time */`
  - What is the last task scheduled on a workstation?
    - `SD_task_t last_scheduled_task`
- Principle:
  - Create a data structure comprising all needed information
    - Allocation and destruction functions
  - Write access functions (set/get)
    - Use the `data` field of a workstation (same is true for `SD_task_t`)

```
typedef struct _WorkstationAttribute {
  double available_at;
  SD_task_t last_scheduled_task;
} *WorkstationAttribute;

static double SD_workstation_get_available_at(SD_workstation_t ws) {
  WorkstationAttribute attr = (WorkstationAttribute) SD_workstation_get_data(ws);
  return attr->available_at;
}
static void SD_workstation_set_available_at(SD_workstation_t ws, double time) {
  WorkstationAttribute attr = (WorkstationAttribute) SD_workstation_get_data(ws);
  attr->available_at = time;
  SD_workstation_set_data(ws, attr);
}
```

Exercise 8

- Write a `SD_workstation_allocate_attribute` function
  - Allocate the data structure
  - Attach it the workstation given as input
- Write a `SD_workstation_free_attribute` function
  - Free the data structure
  - Reset the `data` field
- Write the access functions for the `last_scheduled_task` attribute
  - Use the functions for the `available_at` attribute

Solution to Exercise 8

```
static void SD_workstation_allocate_attribute(SD_workstation_t ws){
  void *data = calloc(1, sizeof(struct _WorkstationAttribute));
  SD_workstation_set_data(ws, data);
}

static void SD_workstation_free_attribute(SD_workstation_t ws) {
  free(SD_workstation_get_data(ws));
  SD_workstation_set_data(ws, NULL);
}

static SD_task_t SD_workstation_get_last_scheduled_task(SD_workstation_t ws){
  WorkstationAttribute attr = (WorkstationAttribute) SD_workstation_get_data(ws);
  return attr->last_scheduled_task;
}

static void SD_workstation_set_last_scheduled_task(SD_workstation_t ws, SD_task_t task){
  WorkstationAttribute attr = (WorkstationAttribute) SD_workstation_get_data(ws);
  attr->last_scheduled_task=task;
  SD_workstation_set_data(ws, attr);
}
```

- Get solution in `solutions/ex8.c`
- ...and copy the contents in `solutions/ex5-7.c`

Agenda

- How to describe DAGs?
- How to Describe Resources?
- How to Write Scheduling Heuristics?
  - General Information
  - A Simple Static Round-Robin Scheduler
  - The Min-Min List Scheduling Algorithm
- Conclusion

Definition of DAG Scheduling

Basic principle

- For each task
  - Assign a (set of) resource(s) for execution
  - Define an execution order
- Respect the precedence constraints
  - A task cannot start before all its predecessors have completed

Types of scheduling

- Offline
  - Take all decisions beforehand and then simulate
- Online
  - Take the decisions as the simulation goes

Running the Simulation

Static Schedules

- Build the complete schedule before running the simulation
  - Call a `SD_task_schedule*` function for each task
- Then call `SD_simulate(-1.)`
  - It will stop when all the work has been done
  - Or if no more tasks are reachable

Dynamic Schedules

- Build the schedule during the simulation
- Two options
  - Hold the simulation every X seconds to take more decisions: `SD_simulate(X)`
  - Add watchpoints on the state of tasks
    - `SD_task_watch (task, state)`
    - The simulation will be hold each time a watch point is reached
    - For in time when a task goes from `SD_TASK_RUNNING` to `SD_TASK_DONE`
- This requires to add an outer loop
- Dynamic rescheduling is possible with `SD_task_unschedule`

What You Can Get After the Simulation

- When the task did actually start
  - `SD_task_get_start_time (task)`
- When the task did actually finish
  - `SD_task_get_finish_time (task)`
- How many workstation were used to execute a task
  - `SD_task_get_workstation_count (task)`
- And which ones
  - `SD_task_get_workstation_list (task)`
- Plot a Gantt chart and analyze performance metrics
  - Using either Jedule or Pajé built-in instrumentation

A Simple Static Round-Robin Scheduler

- When scheduling DAGs
  - Compute tasks run on one host only
  - Data transfers are point-to-point communications
- Typed tasks ⇒ Get rid off all the complexity of parallel tasks
  -
- Creation
  - `compute_task = SD_task_create_comp_seq(name, data, amount)`
  - `transfer_task = SD_task_create_comm_e2e(name, data, amount)`
- Scheduling
  - `SD_task_schedulev(task, workstation_nb, workstation_list)`
  - `SD_task_schedulel(task, workstation_nb, ...)`
  - `amount` will be directly used
- Transfers are auto-scheduled

Exercise 9

- Start from the solution of exercise 8 in `solutions/ex8.c`
  - Important: Remove the line that destroy all the tasks
- Allocate attributes for each workstation
- Code the following (dummy) heuristic
  - For each compute task, i.e., whose kind is `SD_TASK_COMP_SEQ`
    - Schedule it on a workstation in a round robin fashion
    - Update the `last_scheduled_task` attribute
- Call the main simulation function
- Print value of the `last_scheduled_task` attribute, dump information and destroy attributes for each workstation
- Dump information for and destroy each task
- Print simulation time
  - Use the `SD_get_clock()` function

Solution to Exercise 9 (1/2)

```
int main(int argc, char **argv){
  unsigned int cpt, cpt2;
  int nworkstations;
  const SD_workstation_t * workstations;
  SD_task_t task;
  xbt_dynar_t dag;

  SD_init(&argc, argv);

  dag = SD_daxload(argv[1]);

  xbt_dynar_foreach(dag, cpt, task)
    SD_task_dump(task);

  SD_create_environment(argv[2]);

  nworkstations = SD_workstation_get_number();
  workstations = SD_workstation_get_list();

  for (cpt = 0; cpt < nworkstations; cpt++){
    SD_workstation_dump(workstations[cpt]);
    SD_workstation_allocate_attribute(workstations[cpt]);
  }

  ...
```

Solution to Exercise 9 (2/2)

```
  ...
  cpt=0;
  xbt_dynar_foreach(dag, cpt2, task)
    if (SD_task_get_kind(task) == SD_TASK_COMP_SEQ){
      SD_task_schedulel(task, 1, workstations[cpt]);
      SD_workstation_set_last_scheduled_task(workstations[cpt++], task);
    }
  SD_simulate(-1);

  for (cpt = 0; cpt < nworkstations; cpt++){
    printf("Last scheduled task on %s is %s\n",
           SD_workstation_get_name(workstations[cpt]),
           SD_task_get_name(SD_workstation_get_last_scheduled_task(workstations[cpt])));
    SD_workstation_dump(workstations[cpt]);
    SD_workstation_free_attribute(workstations[cpt]);
  }

  xbt_dynar_foreach(dag, cpt, task){
    SD_task_dump(task);
    SD_task_destroy(task);
  }
  printf("Simulation time: %f seconds\n", SD_get_clock());
  SD_exit();
  return 0;
}
```

- Get solution in `solutions/ex9.c`

A Complete Scheduling Simulator Example

The Min-Min List Scheduling Algorithm

- For each ready task
  - get the workstation that minimizes the completion time
- select the task that has the minimum completion time on its best workstation
  - And schedule it there
- Full code available at `$SIMGRID_HOME/examples/simdag/scheduling/minmin_test.c`

What is needed ?

- Functions to
  - Estimate the Earliest Finish Time of a task on a workstation
  - Find the workstation that minimizes this EFT
  - Get the list of ready tasks
- The main scheduling function
  - That dynamically take decisions each time a task completes
    - Thanks to watchpoints
    - Call `SD_simulate` several times

Some Useful Prediction Functions

Sequential Computation and End-to-End Communications

- `SD_workstation_get_computation_time (workstation, amount)`
- `SD_route_get_communication_time (src, dst, amount)`
- These functions do not take concurrent executions into account

Routes and workstations

- `SD_route_get_current_bandwidth (src, dest)`
- `SD_route_get_current_latency (src, dest)`
- `SD_workstation_get_available_power(workstation)`

[Advanced] Default Parallel Tasks

- `SD_task_get_execution_time`
  - Workstation list
  - Array of computation amounts
  - Communication matrix
- `SD_task_get_remaining_amount`
  - The simulation is hold, how much computation remains for this task?

Exercise 10

- Start again from the solution of exercise 8 in `solutions/ex8.c`
  - Important: Remove the line that destroy all the tasks
  - Allocate attributes for each workstation
- Write a function `double finish_on_at(SD_task_t task, SD_workstation_t workstation)` that
  - Estimate when the last incoming data (if any) may arrive on `workstation`
    - Got to know the *grand parent* of the task to know the transfer source
    - Got to know the *parent* of the task to know the transfer size
  - Estimate when `task` can actually start
    - Maximum of arrival of last data and availability time of `workstation`
  - Estimate the execution time of `task` on `workstation`
  - Add both values and return the result

Solution to Exercise 10 (1/2)

```
double finish_on_at(SD_task_t task, SD_workstation_t workstation){
  double result, data_available = 0., last_data_available, redist_time = 0;
  unsigned int i;
  SD_task_t parent, grand_parent;
  xbt_dynar_t parents, grand_parents;
  SD_workstation_t *grand_parent_workstation_list;

  parents = SD_task_get_parents(task);

  if (!xbt_dynar_is_empty(parents)) {
    last_data_available = -1.0;
    xbt_dynar_foreach(parents, i, parent) {
      if (SD_task_get_kind(parent) == SD_TASK_COMM_E2E) {      /* normal case */
        grand_parents = SD_task_get_parents(parent);

        xbt_dynar_get_cpy(grand_parents, 0, &grand_parent);
        grand_parent_workstation_list = SD_task_get_workstation_list(grand_parent);

        /* Estimate the redistribution time from this parent */
        redist_time = SD_route_get_communication_time(grand_parent_workstation_list[0],
                                                      workstation, SD_task_get_amount(parent));
        data_available = SD_task_get_finish_time(grand_parent) + redist_time;

        xbt_dynar_free_container(&grand_parents);
      }
  ...
```

Solution to Exercise 10 (2/2)

```
  ...

      if (SD_task_get_kind(parent) == SD_TASK_COMP_SEQ) { /* no transfer: control dep. */
        data_available = SD_task_get_finish_time(parent);
      }

      if (last_data_available < data_available)
        last_data_available = data_available;
    }

    xbt_dynar_free_container(&parents);

    result = MAX(SD_workstation_get_available_at(workstation), last_data_available) +
        SD_workstation_get_computation_time(workstation, SD_task_get_amount(task));
  } else {
    xbt_dynar_free_container(&parents);

    result = SD_workstation_get_available_at(workstation) +
             SD_workstation_get_computation_time(workstation, SD_task_get_amount(task));
  }
  return result;
}
```

- Get source of code of this function in `solutions/ex10.c`

Exercise 11

- Write a function `SD_workstation_t SD_task_get_best_workstation(SD_task_t task)` that
  - For each workstation, estimate the time at which `task` would finish
  - Keep the workstation that leads to the minimum value
  - Return this workstation
- Write a function `xbt_dynar_t get_ready_tasks(xbt_dynar_t dag)` that
  - Create a dynamic array of tasks
  - Browse the dag and push in the array all the tasks that are
    - A computation
    - And in the `SD_SCHEDULABLE` state (all compute ancestors are SD_DONE)
  - Return the built array

Solution to Exercise 11

```
SD_workstation_t SD_task_get_best_workstation(SD_task_t task) {
  int i, nworkstations = SD_workstation_get_number();
  double EFT, min_EFT = -1.0;
  const SD_workstation_t *workstations = SD_workstation_get_list();
  SD_workstation_t best_workstation;

  best_workstation = workstations[0];
  min_EFT = finish_on_at(task, workstations[0]);

  for (i = 1; i < nworkstations; i++) {
    EFT = finish_on_at(task, workstations[i]);
    if (EFT < min_EFT){
      min_EFT = EFT;  best_workstation = workstations[i];
    }
  }
  return best_workstation;
}
xbt_dynar_t get_ready_tasks(xbt_dynar_t dag) {
  unsigned int i;
  xbt_dynar_t ready_tasks = xbt_dynar_new(sizeof(SD_task_t), NULL);
  SD_task_t task;

  xbt_dynar_foreach(dag, i, task)
    if (SD_task_get_kind(task)==SD_TASK_COMP_SEQ && SD_task_get_state(task)==SD_SCHEDULABLE)
      xbt_dynar_push(ready_tasks, &task);

  return ready_tasks;
}
```

▶ Get source of code of these functions in `solutions/ex11.c`

Exercise 12

- Start from solutions of `ex8.c`, `ex10.c`, and `ex11.c`
- Write the `main` function that
  - Load the environment
    - Allocate attributes for all workstations
  - Load the DAG
    - Add watchpoints on the `SD_DONE` state for all tasks
  - Schedule the `root` on the first workstation
  - While `SD_simulate` return tasks whose state changed
    - Get the ready tasks, if none exists, just continue
    - Get their best workstation
    - Compute their EFT on that workstation
    - Select the one finishing the earliest
    - Schedule it
  - Manage resource dependencies
  - Do some cleaning

Solution to Exercise 12 (1/3)

```
int main(int argc, char **argv) {
  unsigned int cursor;
  double finish_time, min_finish_time = -1.0;
  SD_task_t task, selected_task = NULL, last_scheduled_task;
  xbt_dynar_t ready_tasks;
  SD_workstation_t workstation, selected_workstation = NULL;
  int total_nworkstations = 0;
  const SD_workstation_t *workstations = NULL;
  xbt_dynar_t dag, changed;

  SD_init(&argc, argv); /* initialization of SD */

  dag = SD_daxload(argv[1]);   /* load the DAX file */

  xbt_dynar_foreach(dag, cursor, task) /* add watchpoint on task completion */
    SD_task_watch(task, SD_DONE);

  SD_create_environment(argv[2]);  /* creation of the environment */

  /*  Allocating the workstation attribute */
  total_nworkstations = SD_workstation_get_number();
  workstations = SD_workstation_get_list();

  for (cursor = 0; cursor < total_nworkstations; cursor++)
    SD_workstation_allocate_attribute(workstations[cursor]);

  ...
```

Solution to Exercise 12 (2/3)

```
...
/* Schedule the DAX root first */
xbt_dynar_get_cpy(dag, 0, &task);
workstation = SD_task_get_best_workstation(task);
SD_task_schedulel(task, 1, workstation);

while (!xbt_dynar_is_empty((changed = SD_simulate(-1.0)))) {
  /* Get the set of ready tasks */
  ready_tasks = get_ready_tasks(dag);
  if (xbt_dynar_is_empty(ready_tasks)) {
    xbt_dynar_free_container(&ready_tasks);
    xbt_dynar_free_container(&changed);
    continue;  /* there is no ready task, let advance the simulation */
  }
  xbt_dynar_foreach(ready_tasks, cursor, task) {
    workstation = SD_task_get_best_workstation(task);
    finish_time = finish_on_at(task, workstation);
    if (min_finish_time == -1. || finish_time < min_finish_time) {
      min_finish_time = finish_time;
      selected_task = task;
      selected_workstation = workstation;
    }
  }

  SD_task_schedulel(selected_task, 1, selected_workstation);
  ...
```

Solution to Exercise 12 (3/3)

```
    ...
    /* Manage resource dependencies */
    last_scheduled_task =  SD_workstation_get_last_scheduled_task(selected_workstation);
    if (last_scheduled_task && (SD_task_get_state(last_scheduled_task) != SD_DONE) &&
        (SD_task_get_state(last_scheduled_task) != SD_FAILED) &&
        !SD_task_dependency_exists(SD_workstation_get_last_scheduled_task(
                                   selected_workstation), selected_task))
      SD_task_dependency_add("resource", NULL, last_scheduled_task, selected_task);

    SD_workstation_set_last_scheduled_task(selected_workstation, selected_task);
    SD_workstation_set_available_at(selected_workstation, min_finish_time);

    xbt_dynar_free_container(&ready_tasks);
    xbt_dynar_free_container(&changed);
    min_finish_time = -1.;     /* reset the min_finish_time for the next round */
  }
  xbt_dynar_foreach(dag, cursor, task)
    SD_task_destroy(task);
  xbt_dynar_free_container(&dag);
  xbt_dynar_free_container(&changed);

  for (cursor = 0; cursor < total_nworkstations; cursor++)
    SD_workstation_free_attribute(workstations[cursor]);

  SD_exit();
  return 0;
}
```

- Get source of code in `solutions/ex12.c`

Conclusion

- This tutorial gives examples of the basic usage of most SimDag function
  - You should be able to code your own simulator now!
- Where to find more information on SimDag
  - in `$SIMGRID_HOME/examples/simdag`
  - in the contrib section of SimGrid
    - A set of implementations of classical DAG scheduling algorithms
    - `svn co svn://scm.gforge.inria.fr/svn/simgrid/contrib/trunk/DAGSched`
- Feel free to contribute to SimDag and the contrib section
  - And to ask questions on `simgrid-user@lists.gforge.inria.fr`