Application Skeleton Tool, v 1.1

Zhao Zhang, Daniel S. Katz
University of Chicago
{zhaozhang@uchicago.edu, dsk@uchicago.edu}

September 4, 2014

This document introduces the Application Skeleton tool, part of the AIMES project. The report includes an introduction of the tool, examples of how it can be used, a user manual, and some future activities.

Contents

1 Introduction

The Application Skeleton (just 'Skeleton' hereafter) tool [?, ?] lets users quickly and easily produce a synthetic distributed application that runs in a distributed environment, e.g. grids, clusters, clouds. The source code can be accessed at `https://github.com/applicationskeleton/Skeleton`. A skeleton application is a synthetic application that is intended to represent the computation, I/O, and networking behavior of a real application. In the Skeleton tool, an application is composed of one or more stages. The user needs to define each stage's task type, number of processes, task length, computation and I/O interleaving option, read/write buffer size, input source, input size, output size, and other properties.

The Skeleton tool reads an application description file as input, and produces three groups of outputs:

> **Preparation Scripts**: The preparation scripts are run to produce the input/output directories and input files for the synthetic applications.
>
> **Executables**: Executables are the tasks of each application stage. (We assume different stages use different executables.)
>
> **Application**: The overall application can be expressed in three different formats: shell commands, a Pegasus DAG, and a Swift script. The shell commands can be executed in sequential order on a single machine. The Pegasus DAG and the Swift script can be executed on a local machine or in a distributed environment. Executing the Pegasus DAG or Swift script requires Pegasus or Swift, respectively.

2 Examples

Four examples of how the Skeleton tool can be used to build application skeletons follow, for a bag-of-tasks application, a multi-stage application, a MapReduce application, and an iterative application.

2.1 Bag-of-tasks application

Listing 1 shows a sample input for a bag-of-tasks application with uniform task length, and uniform input and output file size, where each task reads one file and writes one file. Figure 1 shows the very simple DAG for this. The sample application description file can be accessed at `https://github.com/applicationskeleton/Skeleton/blob/master/src/sample-input/bag.input`.

Listing 1: Sample input for a bag-of-tasks application

```
Num_Stage = 1

Stage_Name = Bag
    Task_Type = serial
    Num_Tasks = 4
    Num_Processes = 1
    Task_Length = uniform 5
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 1
```

```
        Input_1.Source = filesystem
        Input_1.Size = uniform 1048576
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0
```

2.2 Multi-stage application

Listing 2 shows a three-stage application description file. The first stage has four tasks. Each reads two distinct files as input, runs for some time, and produces an output file. The runtime for each task has a normal distribution, described by a two-value tuple: [average, stdev]. The input and output file size have a similar distribution. (Supported distributions currently include: uniform, normal, gauss, triangular, lognorm, as further discussed in §3.3.1.) The second stage has six tasks. For each, the input file is the output of the first stage, as defined in Line 16, and the mapping between input files and tasks in the second stage is an all-pairs combination. (Mapping is discussed further in §3.3.2.) The mapping is determined by the parameter **Input_Task_Mapping**. The third stage has only one task. It reads the six input files that were the output files from the second stage, and produces a single output file.

Listing 2: Sample input for a three-stage application

```
1  Num_Stage = 3
2
3  Stage_Name = Stage_1
4      Task_Type = serial
5      Num_Tasks = 4
6      Task_Length = normal [10, 1]
7      Num_Processes = 1
8      Read_Buffer = 65536
9      Write_Buffer = 65536
10     Input_Files_Each_Task = 2
11         Input_1.Source = filesystem
12         Input_1.Size = normal [10485760, 1048576]
13         Input_2.Source = filesystem
14         Input_2.Size = normal [10485760, 1048576]
15     Output_Files_Each_Task = 1
16         Output_1.Size = uniform 1048576
17     Interleave_Option = 0
18
19 Stage_Name = Stage_2
20     Task_Type = serial
21     Num_Tasks = 6
22     Task_Length = uniform 32
23     Num_Processes = 1
24     Read_Buffer = 65536
25     Write_Buffer = 65536
26     Input_Files_Each_Task = 2
27     Input_Task_Mapping = combination Stage_1.Output_1 2
28     Output_Files_Each_Task = 1
29         Output_1.Size = uniform 1048576
30     Interleave_Option = 0
31
32 Stage_Name = Stage_3
33     Task_Type = serial
34     Num_Tasks = 1
35     Task_Length = uniform 32
36     Num_Processes = 1
```

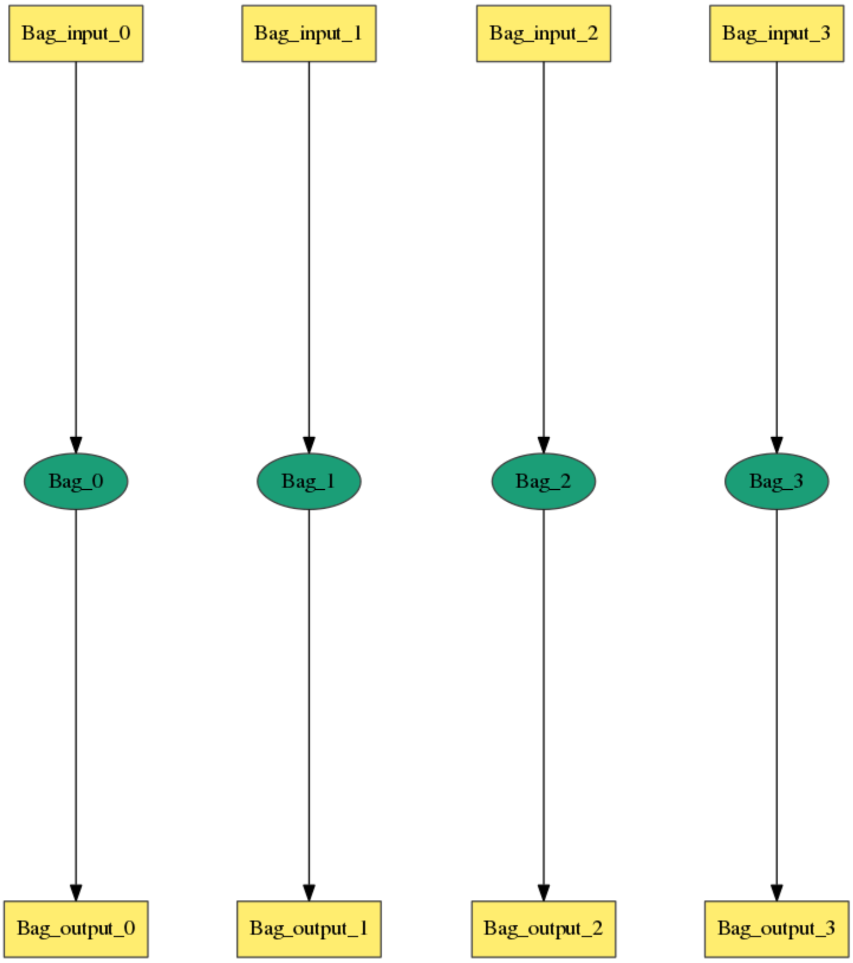

Figure 1: Task flow of an bag-of-task application

```
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 6
    Input_Task_Mapping = combination Stage_2.Output_1 6
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0
```

Figure 2 shows the task flow of the synthetic application that is produced by the Skeleton tool with Listing 2 as input file.

2.3 MapReduce application

Listing 3 shows an input file for a MapReduce application. Each map task reads one file and writes one file. Each Shuffle task task runs a fixed amount of time, representing reorganizing the file contents based on the hash value of some key in the file (the key can be a column in the text file). Each of these tasks reads all of the output files of the map tasks, and writes one output file (representing the records with hash values that fall in the task's range). The two Reduce tasks read the outputs of all shuffle tasks, run a fixed amount of time, and write two distinct output files of fixed size. The Gather stage has one task that reads the output files from stage 3 and writes one output file (representing a combination of its inputs). Figure 3 shows the DAG for this skeleton. The sample application description file can be accessed at `https://github.com/applicationskeleton/Skeleton/blob/master/src/sample-input/map-reduce.input`.

Listing 3: Sample input for a MapReduce application

```
Num_Stage = 4

Stage_Name = Stage_1
    Task_Type = serial
    Num_Tasks = 4
    Task_Length = uniform 10
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 1
        Input_1.Source = filesystem
        Input_1.Size = uniform 1048576
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0

Stage_Name = Stage_2
    Task_Type = serial
    Num_Tasks = 4
    Task_Length = uniform 32
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 1
    Input_Task_Mapping = combination Stage_1.Output_1 4
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0

Stage_Name = Stage_3
```

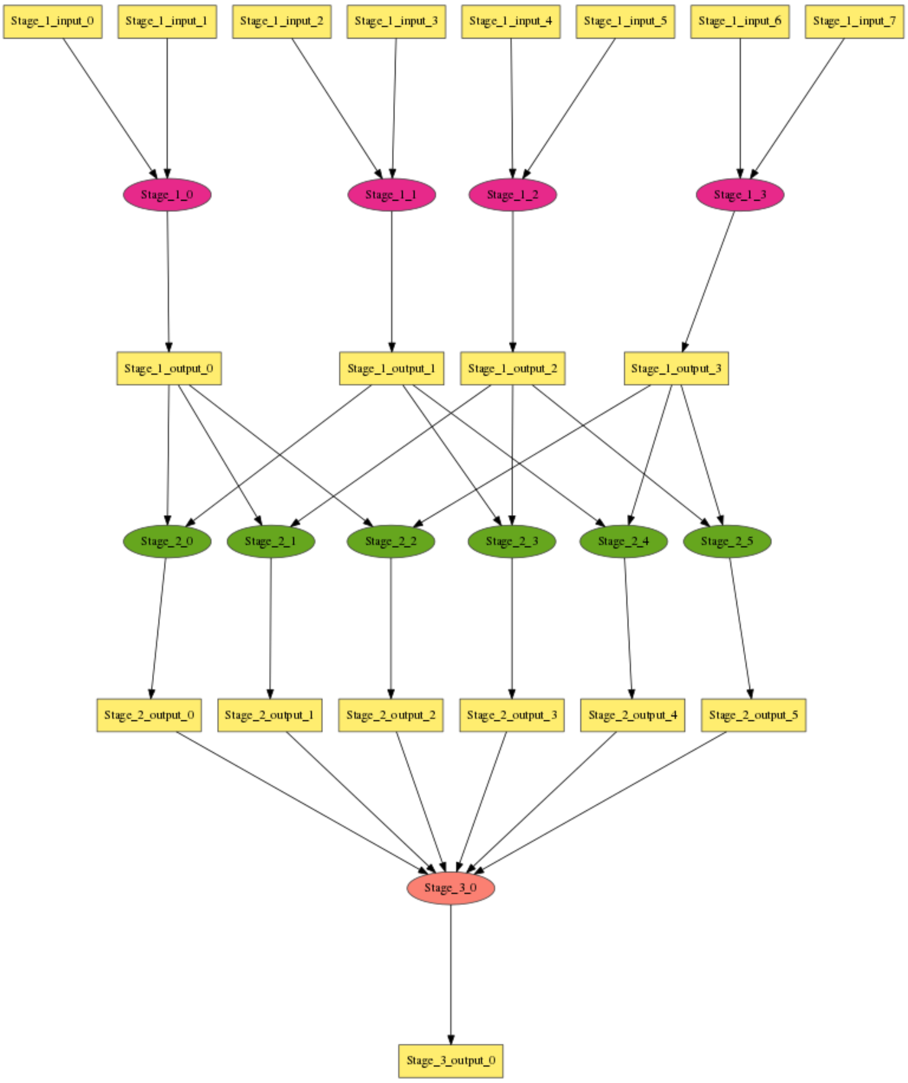

Figure 2: Task flow of the three-stage application

```
    Task_Type = serial
    Num_Tasks = 2
    Task_Length = uniform 32
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 2
        Input_1.Source = Stage_2.Output_1
        Input_2.Source = Stage_2.Output_1
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0

Stage_Name = Stage_4
    Task_Type = serial
    Num_Tasks = 1
    Task_Length = uniform 32
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 2
        Input_1.Source = Stage_3.Output_1
        Input_2.Source = Stage_3.Output_1
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0
```

2.4 Iterative application

The Skeleton tool can generate single-stage and multi-stage iterative applications, in both cases with a fixed number of iterations currently. To enable the iterative option, users need to specify three additional parameters: **Iteration_Num**, **Iteration_Stages**, and **Iteration_Substitute**. **Iteration_Num** specifies the number of iterations of the target stage.**Iteration_Stages** specifies the target stage(s). **Iteration_Substitute** specifies the substitution of input files of the target stage with the output files of latter stages.

Listing 4 shows a single-stage iteration example, in this case where the state is iterated three times, using the output from the previous iteration as the input for the second and third iterations. Figure 4 shows the visualized data flow.

Listing 4: Sample input for a single stage iterative application

```
Num_Stage = 1

Stage_Name = Stage_1
    Task_Type = serial
    Num_Tasks = 4
    Task_Length = uniform 10
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 1
        Input_1.Source = filesystem
        Input_1.Size = uniform 1048576
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576
    Interleave_Option = 0
    Iteration_Num = 3
    Iteration_Stages = Stage_1
    Iteration_Substitute = Stage_1.Input_1, Stage_1.Output_1
```

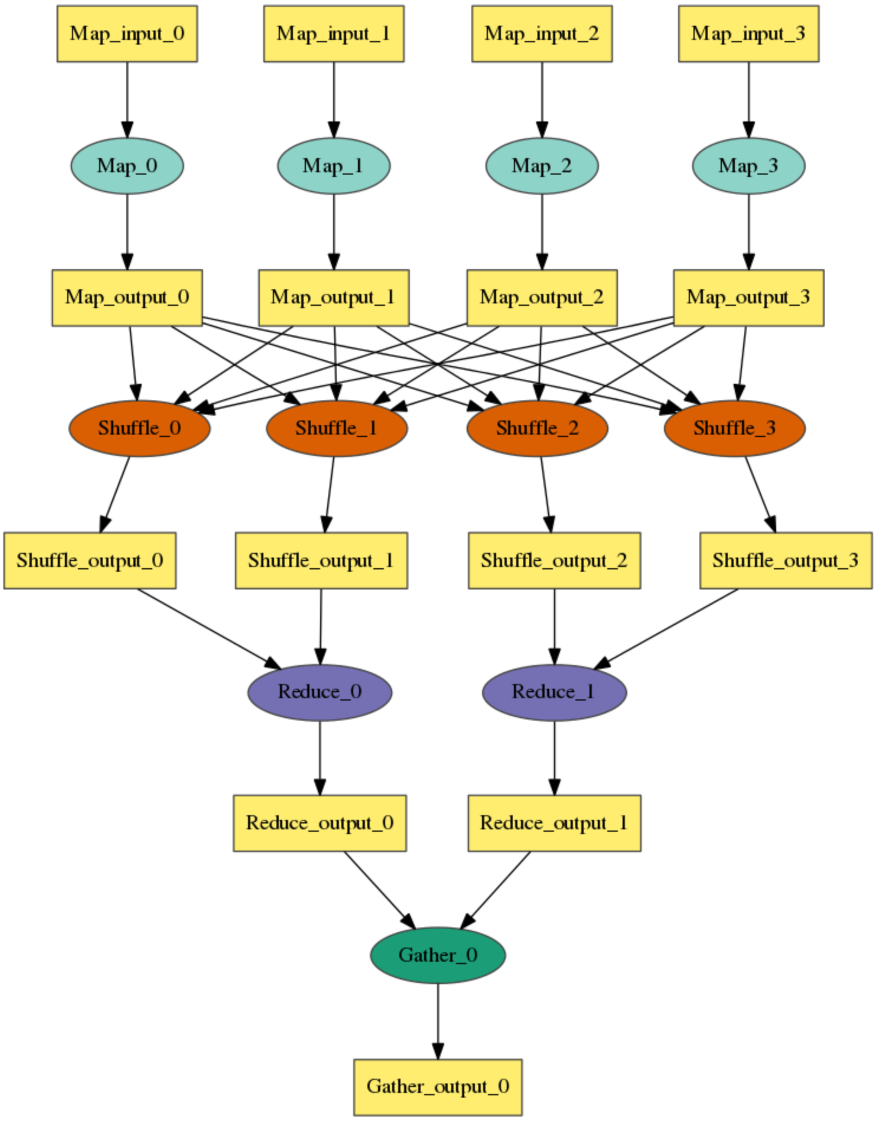

Figure 3: Task flow of an MapReduce application with Stage_1 referred as Map, Stage_2 referred as Shuffle, Stage_3 referred as Reduce, and Stage_4 referred as Gather

3 User manual

This section documents how a user can run the Skeleton tool and the available parameters to describe an application or an application stage.

3.1 Requirements

The Skeleton code is built to use Python3. To use Python2 instead, please follow the instructions in the skeleton source file (`https://github.com/applicationskeleton/Skeleton/blob/master/src/skeleton.py`).

3.2 Application-wide parameter

There is only one parameter that spans the whole description file:

- **Num_Stage**: defines the total number of stages on this application.

3.3 Stage-wide parameters

Parameters documented in this section is only valid within its stage declaration scope.

- **Stage_Name**: declares the stage name, e.g. Stage_1.
- **Task_Type**: declares the task type (serial or parallel).
- **Num_Processes**: declares the number of processes in each task. If **Task_Type** is serial, then **Num_Processes** has to be "1". If **Task_Type** is parallel, then **Num_Processes** can be an integer value that is greater or equal to 1.
- **Num_Tasks**: declares the number of tasks in the stage.
- **Task_Length**: declares the length of the tasks in the stage. The distribution of task lengths can be uniform, normal, gauss, triangular, or lognorm. The distribution of task lengths can be a function of the input file size if and only if there is one input file per task. Please refer to §3.3.1 for details.
- **Read_Buffer**: specifies the read buffer of each task, with default unit of bytes. e.g., 65536.
- **Write_Buffer**: specifies the write buffer of each task, with default unit of bytes. e.g., 65536.
- **Input_Files_Each_Task**: declares the ratio between number of input files and the number of tasks.
- **Input_$.Source**: declares the input source of the $th (i.e., 1 = first, 2 = second, 3=third, etc.) input file, either filesystem, or outputs of a previously defined stage (e.g., `Input_$1.Source = Stage_1.Output_1`).
- **Input_$.Size**: declares input file size of the $th input file. The distribution of task lengths can be uniform, normal, gauss, triangular, or lognorm. This parameter has to appear in pairs with **Input_$.Source**. (e.g., `Input_$1.Size = uniform 1048576`)

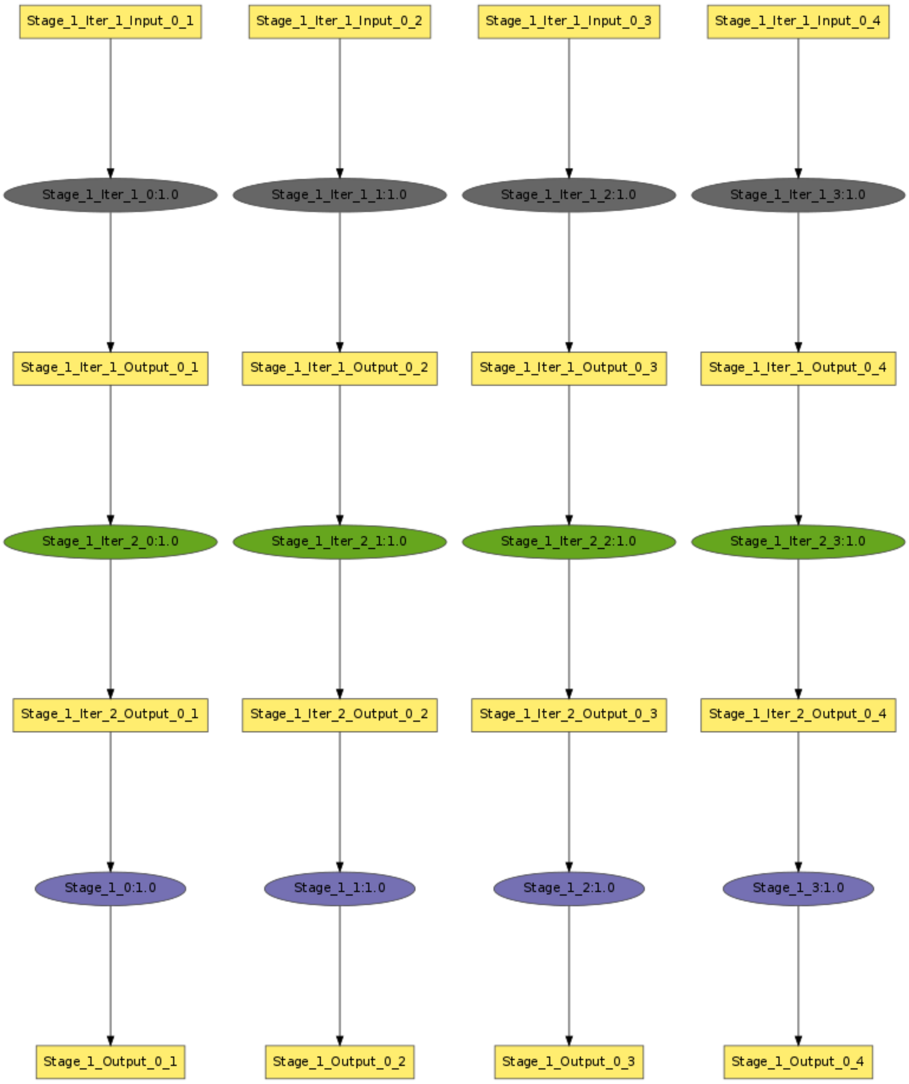

Figure 4: Task flow of a single stage iterative application

- **Input_Task_Mapping**: declares user specified input file and task mapping. The mapping option currently only supports combination mapping and external mapping. Combination mapping lets users specify the input files in combinations and map to tasks. Please refer to Line 27 in Listing 2 for an example. External mapping requires an executable that writes file grouping to standard output, with each group of file in a single line, delimited by spaces. Please refer to §3.3.2 for details and examples.
- **Output_Files_Each_Task**: declares the number of output files per task in the stage. (Multiple tasks writing to one file are not currently supported.)
- **Output_$.Size**: declares output file size for the stage. The distribution of task lengths can be uniform, normal, gauss, triangular, or lognorm.
- **Interleave_Option**: the interleaving option of the task. Please refer to §3.3.3 for details.
- **Iteration_Num**: optional parameter specifying the number of iterations of one or more stages.
- **Iteration_Stages**: the names of the stages involved in the iteration, including the current stage.
- **Iteration_Substitute**: the source and target files for substitution in iterations. (e.g., Line 18 in Listing 4).

3.3.1 Distributions

The skeleton tool implements two types of distributions.

The first type, statistical distributions, can be used for **Task_Length**, **Input_$.Size**, and **Output_$.Size**.

The second type, dependent distributions, can be used for **Task_Length** and **Output_$.Size**. **Task_Length** can be a function of **Input_$.Size**, and **Output_$.Size** can be a function of **Task_Length** or **Input_$.Size**.

Statistical distributions If **Task_Length**, **Input_$.Size**, and Output_$.Size are to be described as statistical distributions, one of the following distributions and formats should be used:

- **uniform**: [number], e.g. 5
- **normal**: [avg, stdev], e.g. [5, 1]
- **gauss**: [avg, stdev], e.g. [5, 1]
- **triangular**: [avg, stdev], e.g. [5, 1]
- **lognorm**: [avg, stdev], e.g. [5, 1]

For details, users may refer to `http://docs.python.org/3.3/library/random.html`

Dependent distributions

Task length as a function of input file size: The **Task_Length** can be a function of **Input_$.Size** if and only if there is one input file per task. The format to describe this option is:

- **input**: [coefficient, power], e.g. [4, 2]

where task length is computed as: $\text{coefficient} * \text{filesize}^{\text{power}}$. Listing 5 shows a sample code for this.

Listing 5: Use case of task length as a function of input file size

```
Num_Stage = 1

Stage_Name = Stage_1
    Task_Type = serial
    Num_Tasks = 4
    Task_Length = polynomial [4, 2] Input_1
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 1
        Input_1.Source = filesystem
        Input_1.Size = uniform 5
    Output_Files_Each_Task = 1
    Output_File_Size = uniform 1048576
```

Output file size as a function of input file size or task length: The **Output_File_Size** can be a function of either **Input_File_Size** or **Task_Length**. The formats to describe these two options are:

- **input**: [coefficient, power], e.g. [4, 2]
- **length**: [coefficient, power], e.g. [4, 2]

where output file size is computed as either $\text{coefficient} * \text{filesize}^{\text{power}}$ or $\text{coefficient} * \text{tasklength}^{\text{power}}$. Listing 6 and 7 show sample code for these two options.

Listing 6: Use case of output file size as a function of input file size

```
Num_Stage = 1

Stage_Name = Stage_1
Num_Stage = 1

Stage_Name = Stage_1
    Task_Type = serial
    Num_Tasks = 4
    Task_Length = uniform 10
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 1
        Input_1.Source = filesystem
        Input_1.Size = uniform 5
    Output_Files_Each_Task = 1
    Output_File_Size = polynomial [4, 2] Input_1
```

Listing 7: Use case of output file size as a function of task length

```
Num_Stage = 1
```

```
2
3 Stage_Name = Stage_1
4     Task_Type = serial
5     Num_Tasks = 4
6     Task_Length = uniform 10
7     Num_Processes = 1
8     Read_Buffer = 65536
9     Write_Buffer = 65536
10    Input_Files_Each_Task = 1
11        Input_1.Source = filesystem
12        Input_1.Size = uniform 5
13    Output_Files_Each_Task = 1
14        Output_1.Size = polynomial [10, 2] Length
15    Interleave_Option = 0
```

3.3.2 Mapping files between stages

Mapping files between stages can be done in two ways in this version of Skeleton. First, users can explicitly specify **Input_Files_Each_Task**, **Input_$.Source**, and **Input_$.Size**. Second, users can use **Input_Files_Each_Task** and **Input_Task_Mapping** options to specify the mapping if the mapping conforms a combinatorial or user defined function. With **Input_Task_Mapping** option, users can either use the **combination** option to specify the mapping or use the **external** routine to customize the mapping.

Direct specification mapping Lines 11-12 in Listing 7 show the usage of **Input_Files_Each_Task**, **Input_$.Source**, and **Input_$.Size**. Users can specify **Input_$.Source** as "filesystem" or as output files from previous stages: e.g., "Stage_1.Output_1".

Combinatorial mapping **Note that his mapping option requires Python3.2 or higher. External mapping is not compatible with Python2.**

Lines 27 and 40 in Listing 2 specify the input file mapping option for the second and third stage respectively. Instead of declaring every single input file's source and size, we describe the mapping with a combinatorial function. The following specification:

combination Stage_1.Output 2

can be interpreted as from the output files of Stage_1, choose two of them as input files for each task. Choosing two from N files can have $\binom{N}{2}$ different file combinations. Specifically in this case, choosing two files from {output_0, output_1, output_2, output_3} returns six pairs of files: {output_0, output_1}, {output_0, output_2}, {output_0, output_3}, {output_1, output_2}, {output_1, output_3}, {output_2, output_3}. These six file pairs will be assigned to six tasks, so each task gets a distinct file pair as its input files.

External Mapping Another **Input_Task_Mapping** option is *external*. Here, a user-specified shell script or a python function will be called by the Application Skeleton tool. The external script has to print the input files names of a task in a line, and the python function needs to return a nested list of input file names.

If user specifies the source of the input files of the second stage as the output of the first stage, but does not use the **Input_Task_Mapping** option, then the files are mapped to each task in a sequential manner: the first and second file are mapped to the first task, the third and forth file are mapped to the second task, and so on. If the mapping runs out of input files, it will go back to the beginning of the input file list and multiple tasks will consume some input files.

Listing 8: Use case for external mapper

```
Num_Stage = 2

Stage_Name = Stage_1
    Task_Type = serial
    Num_Tasks = 4
    Task_Length = uniform 10s
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 2
        Input_1.Source = filesystem
        Input_1.Size = uniform 1048576B
        Input_2.Source = filesystem
        Input_2.Size = uniform 1048576B
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576B
    Interleave_Option = 0

Stage_Name = Stage_2
    Task_Type = serial
    Num_Tasks = 6
    Task_Length = uniform 32s
    Num_Processes = 1
    Read_Buffer = 65536
    Write_Buffer = 65536
    Input_Files_Each_Task = 2
    Input_Task_Mapping = external sample-input/mapping.sh
    Output_Files_Each_Task = 1
        Output_1.Size = uniform 1048576B
    Interleave_Option = 0
```

Listing 9: Sample code for an external mapper

```
#!/bin/bash

echo Stage_1_Output_0_1 Stage_1_Output_0_2
echo Stage_1_Output_0_1 Stage_1_Output_0_3
echo Stage_1_Output_0_1 Stage_1_Output_0_4
echo Stage_1_Output_0_2 Stage_1_Output_0_3
echo Stage_1_Output_0_2 Stage_1_Output_0_4
echo Stage_1_Output_0_3 Stage_1_Output_0_4
```

3.3.3 Interleaving

There are four interleaving options for a skeleton task (the serial task, or the process with rank 0 in a parallel task), as visualized in Figure 5:

0. **interleave-nothing**: simply reads, computes, then writes

1. **interleave-read-compute**: interleaves reads and computations, then writes outputs

2. **interleave-compute-write**: reads all inputs, then interleaves writes and computations
3. **interleave-all**: interleave reads, computations, and writes

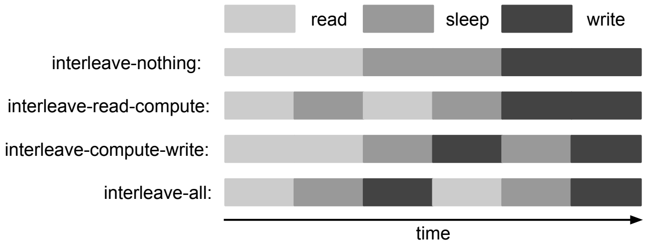

Figure 5: Visualization of the Interleaving Options

Line 17 in Listing 1 is a usage example of specifying the **Interleave_Option**.

3.4 Task Parameters

The Skeleton tool's task is implemented by a versatile synthetic C program, where users can specify parameters to define the computation and I/O behavior, if needed. (Note: this is optional; describing parameters for all tasks in a stage, as discussed in §3.3 will be sufficient for many application skeletons.)

The Skeleton task support two types: serial task and parallel task. A typical task specification looks like:

> Path_to_Task Task_Type Num_Processes Task_Length Read_Buffer Write_Buffer Num_Input Num_Output Interleave_Option [Input_File] [Output_File Output_Size]

which could be filled in as:

> task serial 1 16 65536 65536 1 1 0 input_file output_file 1048576

The parameters of a task can be specified as:

- **Task_Type**: A user needs to declare **Task_Type** to be either "serial" or "parallel"
- **Num_Processes**: If a task is declared to be "serial", **Num_Processes** has to be specified as "1". If a task is declared to be "parallel", then **Num_Processes** can be any integer value that is greater or equal to 1.
- **Task_Length**: **Task_Length** is specified in "seconds"; it can be a integer or floating point number
- **Read_Buffer**: **Read_Buffer** specifies how much data the task reads as a chunk from storage. This parameter might be affected by file system buffer or other system settings
- **Write_Buffer**: **Write_Buffer** specifies how much data the task writes as a chunk to storage.This parameter might be affected by file system buffer or other system settings

- **Num_Input**: Number of input files
- **Num_Output**: Number of output files
- **Interleave_Option**: Interleaving of input, output, and computation.
- [**Input_File**]: A user needs to list all the input files in the task description. The number of input file names has to be compliant with the parameter **Num_Input**
- [**Output_File, Output_Size**]: A user needs to list all output files with the file size strictly after each file name.

4 Examples in GitHub

This section explains the examples in the GitHub repository and their usage. The source code is available from `https://github.com/applicationskeleton/Skeleton/tree/master/src/sample-input`. To run these examples, go to the "src" directory, then execute "./skeleton.py sample-input/xxx.input Shell".

- **bag.input**: Generates a bag of independent tasks.
- **two-stage.input**: Generates a two-stage application with file dependencies between the stages.
- **combination.input**: Generates a two-stage application with file dependencies as a combinatorial functions.
- **single-stage-iterative.input**: Generates an iterative application over one stage.
- **multiple-stage-iterative.input**: Generates an iterative application over multiple stages.
- **map-reduce.input**: Generates an map-reduce application.
- **sample.input**: Generates a multi-stage workflow application.
- **external-mapper.input**: Generates an two-stage application using the external mapper to specify task file mapping.
- **mapping.sh**: A example external script that prints the file mapping to standard output.

5 Future work

- Site Specification: Enables automatic or semi-automatic remote compiling and task deployment on multiple remote sites.
- Error Reporting: Enables error reporting in the application description file.
- Task dependencies: Enables a task dependency purely on task sequence, not just through files.

- Concurrent tasks: Enables multiple concurrent tasks that exchange files or data while running.

6 Acknowledgments

This work was funded by DOE ASCR under award DE-SC0008617.