Application Skeletons: Encapsulating MTC Application Task Computation and I/O

Zhao Zhang
Department of Computer Science
University of Chicago
zhaozhang@uchicago.edu

Daniel S. Katz
Computation Institute
University of Chicago & Argonne National Laboratory
d.katz@ieee.org

ABSTRACT

Computer scientists who work on tools and systems meant to support or enable a variety of distributed computing applications want to prove that the systems they design actually help those applications. However, doing this by using the actual applications can be difficult due to policy or technical issues when accessing and building the application and necessary data sets. These issues led us to the idea of an Application Skeleton – a simple yet powerful tool to build synthetic applications that represent real applications, with runtime, I/O, and intertask communication close to those of the real applications. This allows computer scientists to focus on the system they are building; they can work with the simpler Skeleton applications and be sure that their work will also be applicable to the real applications. Skeletons currently can create easy-to-access, easy-to-build, and easy-to-run bag-of-task, map-reduce, and multi-stage workflow applications. In this initial work, we show that a Skeleton version of the Montage application has a runtime difference of 2.6% in total on 64 processors on a BG/P supercomputer. And six of eight stages have an error within 5%.

1. INTRODUCTION

Computer scientists who build tools and systems (programming languages, runtime systems, file systems, workflow systems, etc.) to support distributed applications often have to work on real scientific applications to prove the effectiveness of the system. However, accessing and building the real applications is time consuming or sometimes infeasible due to one or more of the following reasons:

- Some applications (source) are privately accessible
- Some data is difficult to access
- Some applications use legacy code and are dependent on out-of-date libraries
- Some applications are hard to understand because of the knowledge gap between the computer scientists and domain scientists

To address these issues, our goal is to build a tool that let users quickly and easily produce a synthetic distributed application that is executable in a distributed environment, e.g. grids, clusters, and clouds. We want the synthetic applications to have close to identical runtime, I/O, and intertask communication to the real applications. Also, we want the synthetic applications to be executable with some well deployed distributed computing middleware, e.g. Swift [14] and Pegasus [2], as well as the ubiquitous Unix shell.

The challenge of this research is to provide an easy-to-use programming (specification) model to express a Skeleton application with an acceptable performance difference between it and the real application it represents. Our Skeleton concept uses a top-down approach to abstract the application: an application is composed by a number of stages, and each stage has a number of tasks. Users describe an application by specifying the number of stages and the number of tasks, input and output file and task mapping, task length, and file size inside each stage. Our Skeleton tool lets users to specify task lengths and file sizes as statistical distributions or a polynomial functions of other parameters. For example, input file size can be a normal distribution, task length can be a linear function of input file size, and output size can be a binomial function of task runtime.

The contributions of this work include:

- An application abstraction that gives users good expressiveness to capture the key performance elements of applications.
- A versatile Skeleton implementation that is interoperable with mainstream workflow frameworks and systems (e.g., Shell, Pegasus and Swift).

The rest of the paper is organized as following: Section 2 presents a number of distributed applications. Section 3 introduces the Skeleton design and its programming (specification) model. In Section 4, we evaluate the runtime of all stages of the Montage application against the Skeleton version of the application. Section 5 discusses related work, we conclude and discuss future work in Section 6.

2. DISTRIBUTED APPLICATIONS

Application Skeleton is motivated by a wide variety of application types, so the Skeleton aims to be expressive for those applications in return. The initial Skeleton implementation allows the user to express:

- Bag of Tasks: A set of independent tasks. Examples: MG-RAST [10], DOCK [12]

- MapReduce: A set of distributed application with key-value pairs as intermediate data format. Examples: high energy physics histograms [3], object ordering [1]
- Multi-stage Workflow: A set of distributed applications with multiple stages and use POSIX files as intermediate data format. Examples: Montage [4], BLAST [9]

The Skeleton concept will also allow expressing the following types of applications, though this is not yet implemented:

- Iterative MapReduce: MapReduce application with iteration requirement. Example: graph mining [8]
- Campaign: An iterative application with a varying set of tasks that must be run to completion in each iteration. Examples: Kalman filtering [13]
- Concurrent Tasks: A set of tasks that have to be executed at the same time. Examples: Coupled fusion simulation [5]

3. SKELETON DESIGN AND PROGRAMMING MODEL

A Skeleton represents an application using a top-down approach: the whole application is composed of stages, each of which is composed of tasks. A synthetic application is specified initially by the number of stages in the application. Each stage is defined by the input/output files, tasks (number and length), and file-task mapping inside each stage.

To understand the Skeleton tool, one should consider its overall design (§3.1) as well as how a particular application Skeleton is specified (§3.2).

3.1 Skeleton Tool Design and Usage

The Skeleton tool is implemented as a parser. It reads in a configuration file that specifies a Skeleton application, and produces three groups of outputs:

Preparation Scripts: The preparation scripts are run to produce the input/output directories and input files for the Skeleton application.

Executables: Executables are the actual tasks of each application stage. (We assume different stages use different executables.)

Application: The overall Skeleton application can be implemented in one of three formats: shell commands, a Pegasus DAG, or 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, to be installed.)

To execute a Skeleton application, the preparation scripts must first be run to create the initial input date files. Then the Skeleton application itself can be run (the DAG can be run with Pegasus, the Swift script with Swift, or the Shell script with Bash). The executables produced by the Skeleton tool as the tasks for each stage copy the input files from the shared file system to RAM, sleep for some amount of time (specified as the runtime), and copy the output files from RAM to the shared file system.

Limitations

An application Skeleton should include application-specific information but not platform-specific information. In I/O for example, the amount of I/O is application-specific, and how long the I/O takes is determined when the Skeleton is run on a particular resource. However, we currently model the computational work in a task as a runtime. This is reasonable at this point since most current CPU cores are roughly the same speed due to power issues, but it doesn't match the Skeleton goal. We are investigating better solutions currently, including specifying operation counts and more complex performance models. Unfortunately, the more complex this model gets, the harder it will become for a user to to specify a skeleton.

3.2 Skeleton Specification

Specifying a Skeleton application starts with declaring the number of stages, as shown in the configuration file fragment in Listing 1.

Listing 1: Declaring Number of Stages

```
Num_Stage = 3

Stage_Name = Stage_1
    ...
Stage_Name = Stage_2
    ...
Stage_Name = Stage_3
    ...
```

Each stage is described by the following parameters:

- **Stage_Name**: the stage name. The tasks of this stage are named: ${Stage_Name}_${Task_id}. (This is particularly useful when when external mappers are used)
- **Num_Tasks**: the number of tasks in the stage.
- **Task_Length**: the length of the tasks in the stage. The distribution of task length can be uniform, normal, triangular, or lognorm. The distribution of task length can be a function of the input file size if and only if there is one input file per task.
- **Input_Source**: the source of the input files. This can be either filesystem or outputs of a previously defined stage (e.g. Stage_1.output).
- **Input_Files_Each_Task**: the ratio between number of input files and the number of tasks.
- **Tasks_Each_Input_File**: the ratio between number of tasks and the number of input files. Along with **Input_Files_Each_Task**, these two parameters define the mapping between input files and tasks in this stage, unless an external mapper is used.
- **Input_File_Size**: input file size for the stage. The distribution of task lengths can be uniform, normal, triangular, or lognorm.
- **Input_Task_Mapping**: user specified input file and task mapping. The mapping option currently only supports external mapping. It requires an executable that writes file grouping to standard output, with each group of file in a single line, delimited by spaces. This option lets the user override the mapping scheme implied by **Input_Files_Each_Task** and **Tasks_Each_Input_File** when the application uses a more complex mapping than can be specified with just these two parameters.

Table 1: Skeleton Parameter Format

Parameter	Format	Example	notes
Num_Tasks	Integer	16	
Task_Length	dist [parameter][unit]	uniform 32s	other dists: normal, triangular, lognorm
Input_source	filesystem|Stage_$.Output	Stage_1.Output	
Input_Files_Each_Task	Integer	2	
Tasks_Each_Input_File	Integer	2	
Input_File_Size	dist [parameter][unit]	uniform 1048576	the default unit is byte, other dists available
Input_Task_Mapping	external /path/to/exec	external map.sh	each line of map.sh contains the input files of a task
Output_Files_Each_File	Integer	2	
Output_File_Size	dist [parameter][unit]	uniform 1048576	the default unit is byte, other dists available

- **Output_Files_Each_File**: the number of output files per task in the stage. (Multiple tasks writing to one file are not currently supported.)
- **Output_File_Size**: the output file size in the stage. The distribution of task length can be a statistical distribution such as uniform, normal, triangular, or lognorm. It can also be a polynomial function of input file size or task length.

Listing 2: Sample input for a three-stage application

```
1  Num_Stage = 3
2
3  Stage_Name = Stage_1
4      Num_Tasks = 4
5      Task_Length = normal [10, 1]s
6      Input_Source = filesystem
7      Input_Files_Each_Task = 2
8      Tasks_Each_Input_File = 1
9      Input_File_Size = normal [1048576, 1]
10     Output_Files_Each_Task = 1
11     Output_File_Size = normal [1048576, 1]
12
13 Stage_Name = Stage_2
14     Num_Tasks = 6
15     Task_Length = uniform 32s
16     Input_Source = Stage_1.Output
17     Input_Files_Each_Task = 2
18     Tasks_Each_Input_File = 3
19     Output_Files_Each_Task = 1
20     Output_File_Size = uniform 1048576
21
22 Stage_Name = Stage_3
23     Num_Tasks = 1
24     Task_Length = uniform 32s
25     Input_Source = Stage_2.Output
26     Input_Files_Each_Task = 6
27     Tasks_Each_Input_File = 1
28     Output_Files_Each_Task = 1
29     Output_File_Size = uniform 1048576
```

Listing 2 shows a complete three-stage application description file, and Table 1 explains the required data type and format of the parameters. 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]Unit. The input and output file size have a similar distribution. (Supported distributions currently include: uniform, normal, triangular, lognorm, as further discussed in §3.2.1.) The second stage has six tasks. For each, the input file is the output of the first stage, denoted by Line 16, and the mapping between input files and tasks in the second stage is an all-pair combination. (Mapping is discussed further in §3.2.2.) The mapping is determined by two parameters: Input_Files_Each_task and Tasks_Each_Input_File. The third stage has only one task, which reads as input the six output files from the second stage, and produces a single output file.

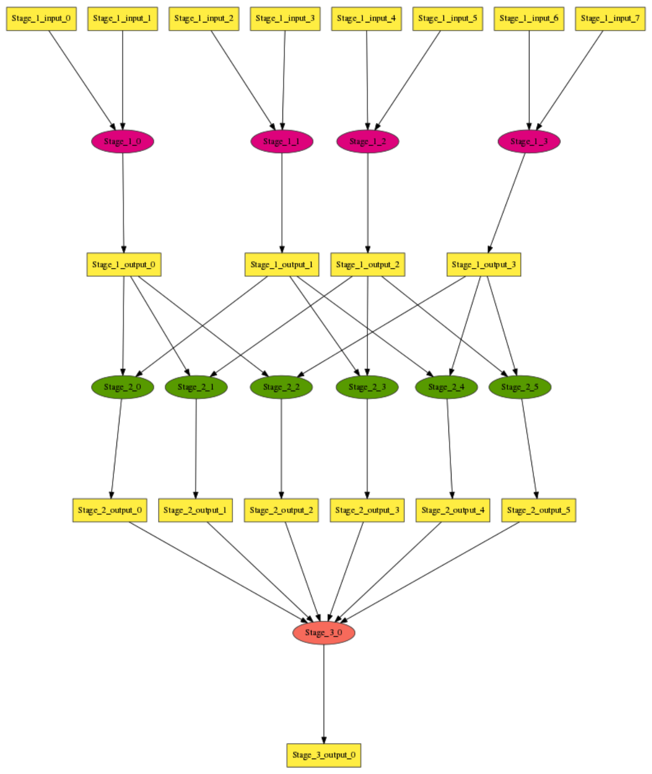

Figure 1: Task flow of the three-stage application

Figure 1 shows the task flow of the synthetic application that is produced by the Skeleton tool with Listing 2 as input.

3.2.1 Parameter Distribution

The skeleton tool implements two categories of distributions.

The first category, statistical distributions, can be used for **Task_Length**, **Input_File_Size**, and **Output_File_Size**.

If **Task_Length**, **Input_File_Size**, **Output_File_Size** are to be described as statistical distributions, one of the distributions and formats in Table 2 should be used.

Table 2: Statistical Distributions

Name	Format	Example
uniform	[number][unit]	5s, 1048576B
normal	[avg, stdev][unit]	[5, 1]s, [1048576, 1000]B
triangular	[avg, stdev][unit]	[5, 1]s, [1048576, 1000]B
lognorm	[avg, stdev][unit]	[5, 1]s, [1048576, 1000]B

The second category, dependent distributions, can be used for **Task_Length** and **Output_File_Size**. **Task_Length** can be a polynomial function of **Input_File_Size**, and **Output_File_Size** can be a polynomial function of **Task_Length** or **Input_File_Size**.

The **Task_Length** can be a polynomial function of **Input_File_Size** if and only if there is one input file per task. This is described as:

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

where task length is computed as: coefficient $*$ filesize$^{\text{power}}$. Listing 3 shows an example of this.

Listing 3: Task length as a function of input file size

```
Num_Stage = 1

Stage_Name = Stage_1
    Num_Tasks = 4
    Task_Length = input [4, 2]x
    Input_Source = filesystem
    Input_Files_Each_Task = 1
    Tasks_Each_Input_File = 1
    Input_File_Size = normal [1048576, 1000]
    Output_Files_Each_Task = 1
    Output_File_Size = normal [1048576, 1000]
```

3.2.2 *File-Task Mapping*

As previously mentioned, mapping files between stages can be done either based on a default mapping implied by **Input_Files_Each_Task** and **Tasks_Each_Input_File**, or by an external mapping routine.

The default file-task mapping is calculated by the value of **Input_Files_Each_Task** and **Tasks_Each_Input_File**. If **Tasks_Each_Input_File** is 1, then **N** tasks shall have **N*Input_Files_Each_Task** distinct input files, with each task mapped to **Input_Files_Each_Task** input files, such as in lines 7 and 8 in Listing 2. If **Input_Files_Each_Task** is set to the number of distinct input files (the number of input files is implicitly set to $\frac{\textbf{Input\_Files\_Each\_Task*Num\_Tasks}}{\textbf{Tasks\_Each\_Input\_File}}$, or inherited from previous stage), and **Tasks_Each_Input_File** has the same value as **Num_tasks**, then each task in this stage maps to all input files, such as in lines 26 and 27 in Listing 2.

External mapping option can be used to describe more complex mappings between tasks and files. An optional parameter, **Input_Task_Mapping**, overrides the default file-task mapping. Currently, this can only be set to **External**. External mapping requires an shell executable that outputs the file grouping to standard output, with each line in a row, and with files delimited by white space. Listings 4 and 5 show an use case and a sample implementation of the external mapper, respectively. The naming rule for the files is: Stage_Name_fileusage_fileid, e.g., Stage_1_input_0. The Skeleton tool only checks the naming correctness of the file names produced by the external mapper: the file names have to start with the stage name, and output files of other stages have to exist before they are mapped to tasks.

Listing 4: Use case of external mapper

```
Num_Stage = 1

Stage_Name = Stage_1
    Num_Tasks = 4
    Task_Length = normal [10, 1]s
    Input_Source = filesystem
    Input_Files_Each_Task = 2
    Tasks_Each_Input_File = 1
    Input_File_Size = normal [1048576, 1000]M
    Input_Task_Mapping = External external.sh
    Output_Files_Each_Task = 1
    Output_File_Size = normal [1048576, 1000]M
```

Listing 5: Sample code of external mapper

```
#!/bin/bash

echo Stage_1_input_0 Stage_1_input_4
echo Stage_1_input_3 Stage_1_input_5
echo Stage_1_input_1 Stage_1_input_6
echo Stage_1_input_2 Stage_1_input_7
```

4. EVALUATION

To compare the performance of the Skeleton application and the real application, we use a 6x6 degree image mosaic example from Montage [4] and the first 256 queries of the NRxNR test of BLAST [9] on 64 IBM BG/P processors. The tasks are launched with AMFS [15] and the input and output files are read and written from/to the GPFS shared filesystem. Figure 2 shows the data flow patterns between Montage stages.

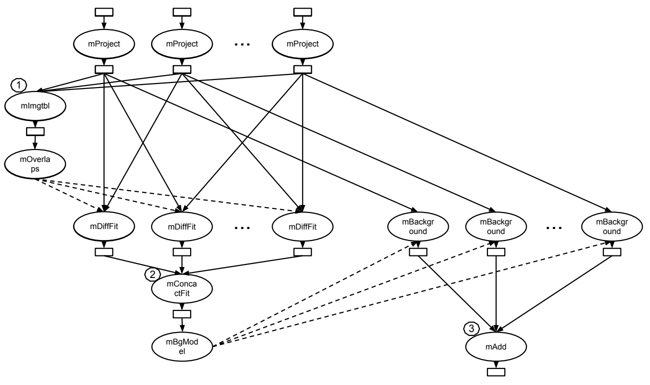

Figure 2: Montage dataflow. Ovals represent tasks and boxes files. Solid lines show file transfers, dashed lines show additional control flow dependencies.

Table 3 shows basic statistics for each Montage stage. Measured Time Avg shows the average time-to-solution of all tasks in each stage. The Skeleton Task Length column shows the exact task length we set in the Skeleton, determined as follows: For the mProjectPP, mOverlaps, mDiffFit, mConcatFit, mBgModel, and mBackground stages, we place the input/output files on RAM disk then round the average time-to-solution up to the nearest integer as the task length. For stages of mImgtbl and mAdd, the input size exceeds the RAM disk size on a compute node, so we cannot execute the tasks with the data on RAM disk. Based on observation and validation, we see that the task's time-to-solution is proportional to the number of input files when the file number is small (10-30), so we project the time-to-solution with the full input data set based on the measured time-to-solution on a smaller data set.

In the Skeleton configuration for the stages where there are a large number of input/output files of the same size, (mProject, mImgtbl, mDiffFit, mBackground and mAdd), we specify that all files for a stage are that size. For mConcatFit, the input file sizes vary from 157 bytes to 292 bytes. To simplify the the Skeleton specification, we set all input

Table 3: Number of tasks, inputs, and outputs, and input and output size, for each Montage stage

Stage	# Tasks	# Inputs	# Outputs	In (MB)	Out (MB)	Measured Time Avg (sec)	Measured Time Stdev	Skeleton Task Length
mProject	1319	1319	2594	2800	10400	11.1	2.5	12
mImgtbl	1	1297	1	5200	0.8	N/A	0	16
mOverlaps	1	1	1	0.8	0.4	9	0	9
mDiffFit	3883	7766	7766	31000	487	1.7	0.6	2
mConcatFit	1	3883	1	1.1	4.3	14	0	14
mBgModel	1	2	1	4.5	0.07	283.1	0	284
mBackground	1297	1297	1297	5200	5200	0.4	0.08	1
mAdd	1	1297	1	5200	7400	N/A	0	519

Table 4: Time-To-Solution Comparison of Skeleton Montage and Real Montage (seconds)

	mProject	mImgtbl	mOverlaps	mDiffFit	mConcatFit	mBgModel	mBackground	mAdd	Total
Montage	290.4	139.7	10.2	359.2	64.6	283.3	102.6	793.4	2040.6
Skeleton	283.4	124.3	10.5	313.5	67.0	283.2	98.2	807.6	1987.6
Error	-2.4%	-11.1%	2.9%	-12.7%	3.9%	-0.04%	-4.3%	1.8%	-2.6%

files of mConcatFit (the output of mDiffFit) to the average file size of 200 bytes. Note that for both the real Montage application and the Skeleton, we use a staging approach to execute the task: we first copy the input files from GPFS to RAM disk, execute the task, write the output files to RAM disk, then copy the output files back to GPFS.

Table 4 compares performance of the Montage Skeleton with the real Montage application on 64 IBM BG/P quad-core processors with GPFS as the shared filesystem. For each data point, we measured the performance three times and show the average value. In total, the Skeleton Montage runs in 2.6% less time the real Montage application.

Among the eight stages, mProject, mDiffFit, and mBackground are those with parallel execution. The measured error is -2.4%, -12.7% and 1.8% respectively. The significant error of mDiffFit is due to the time-to-solution distribution of all tasks: a distribution with an average of 1.7 seconds and a standard deviation of 0.6. The ratio between standard deviation and average is 35.4%, while the ratio for mProject and mBackground is 22.5% and 20.0% respectively. This higher ratio implies a higher variability of mDiffFIt tasks, thus using average time-to-solution for all tasks should result worse precision than the other two stages. We believe this gap can be improve by using Skeleton's statistical distribution functionality.

mOverlaps, mConcatFit, and mBgModel each have a single task, and the input/output fit in a compute node's RAM disk. The Skeleton versions have errors of 2.9%, 3.9% and -0.04%. mImgtbl and mAdd are each a single task with 1297 input files, whose size exceeds a single node's RAM disk size. With the projected task length, the errors of the two stages are -11.1% and 1.8%. We can tell that mImgtbl's time-to-solution grows more than linearly with the number of input files as the variable, while mAdd's time-to-solution grows close to linearly.

For BLAST, we run the first 256 queries from the NRxNR test case on 64 BG/P compute nodes. We set up the task length as uniform for the formatdb and blastp stages. The task lengths of the merge stage vary from one second to 14 seconds, so we use the actual task length for these 16 skeleton tasks. The input file sizes of formatdb tasks are uniform, but the outputs are not. Each formatdb task has three output files, their sizes are 56 MB, 16MB, and 1MB respectively. Due to the limitation of the present Skeleton implementation, we define the synthetic output files as 3, with uniform size of 21 MB. Table 5 shows some basic statistics of BLAST stages and Table 6 shows the measured performance and the comparison between the Skeleton BLAST and real BLAST.

The formatdb stage's error is 7.2%. One possible reason is that the real formatdb task has ~500,000 small writes, each with hundreds of bytes, while our Skeleton synthetic application's writes has a much larger buffer. The 8.0% error of blastp stage could be due to the uniform task length distribution, as the real distribution ranges from 80s to 160s. The predicted merge time-to-solution is low because of the per sequence reads of the input files, which are each hundreds of bytes, while the Skeleton merge has a single large read.

5. RELATED WORK

Skel [7] uses a similar idea to understand the I/O performance of parallel applications on supercomputers. Users can extract the I/O behavior from an application, then produce a skeletal application that mimics the I/O operations and pattern by specifying a Skel configuration file. The produced skeletal application can run on ADIOS [6]. WGL [11] lets users generate a Swift script for a workflow application by describing the data flow patterns between stages.

6. CONCLUSIONS AND FUTURE WORK

We have shown the Skeleton tool can produce synthetic distributed applications that correctly capture important distributed properties of real applications but are much simpler to define and use. The Skeleton tool currently can generate applications that represent bag-of-tasks, MapReduce, and multi-stage workflows. Skeleton applications can be run with mainstream workflow frameworks and systems: Shell, Pegasus, and Swift. The execution comparison between the the initial Skeleton Montage and BLAST and the real Montage and BLAST on 64 BG/P processors shows an acceptable runtime difference of 2.6% and 8.0%, and for each individual stage, the difference ranges from 0.04% to 12.7%, with six out of eleven stages within 5%. These can be improved by more carefully configuring the Skeleton.

In the near future, we will open source the Skeleton code, and invite users and contributors from a wider community to try it and expand it. Our longer term plan includes:

- Use application trace data to produce synthetic applications ideally purely from the trace data but initially from a combination of trace data and user guidance.
- Determine a way to represent the computational work

Table 5: Number of tasks, inputs, and outputs, and input and output size, for each BLAST stage

Stage	# Tasks	# Inputs	# Outputs	In (MB)	Out (MB)	Measured Time Avg (sec)	Measured Time Stdev	Skeleton Task Length
formatdb	64	64	192	3800	4400	41.9	0.1	42
blastp	1024	4096	1024	70402	966	109.2	14.9	110
merge	16	1024	16	966	867	4.4	4.1	real length

Table 6: Time-To-Solution Comparison of Skeleton BLAST and Real BLAST (seconds)

	formatdb	blastp	merge	Total
BLAST	82.1	1996.3	35.9	2114.3
Skeleton	76.2	1835.9	34.0	1946.1
Error	7.2%	8.0%	2.9%	8.0%

in a task that when combined with a particular platform can give an accurate runtime for that task.

- Support tasks with interleaved computation and I/O, rather than just read-compute-write.
- Support tasks that are not generic single core tasks, such as those that internally include OpenMP or MPI, or those that should be bound to specific hardware, e.g., GPUs.
- Support concurrent tasks that need to run at the same time to exchange information.
- Support iterations in Skeleton configuration model.
- Investigate an accurate task-file mapping specification that supports cases other than 1-1 and n-1 that requires less user programming.

Acknowledgments

This work was supported in part by the U.S. Department of Energy under the ASCR award DE-SC0008617 (the AIMES project). It has benefited from discussions with Shantenu Jha, Andre Merzky, Matteo Turilli, Jon Weissman, and Lavanya Ramakrishnan. Computing resources were provided by the Argonne Leadership Computing Facility. Work by Katz was supported by the National Science Foundation while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[2] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahl, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[3] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *4th IEEE International Conf. on eScience*, pages 277–284, 2008.

[4] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. J. of Comp. Sci. and Eng.*, 4(2):73–87, 2009.

[5] S. Klasky, M. Beck, V. Bhat, E. Feibush, B. Ludäscher, M. Parashar, A. Shoshani, D. Silver, and M. Vouk. Data management on the fusion computational pipeline. *Journal of Physics: Conference Series*, 16:510–520, 2005.

[6] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system ADIOS. In *Proceedings of 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24. ACM, 2008.

[7] J. Logan, S. Klasky, H. Abbasi, Q. Liu, G. Ostrouchov, M. Parashar, N. Podhorszki, Y. Tian, and M. Wolf. Understanding I/O Performance Using I/O Skeletal Applications. In C. Kaklamanis, T. Papatheodorou, and P. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2012.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD International Conf. on Management of Data*, pages 135–146, 2010.

[9] D. R. Mathog. Parallel BLAST on split databases. *Bioinformatics*, 19(14):1865–1866, 2003.

[10] F. Meyer et al. The metagenomics RAST server–a public resource for the automatic phylogenetic and functional analysis of metagenomes. *BMC Bioinformatics*, 9(1):386, 2008.

[11] L. Meyer, M. Mattoso, M. Wilde, and I. Foster. WGL - a workflow generator language and utility. `http://dx.doi.org/10.6084/m9.figshare.793815`.

[12] D. Moustakas, P. Lang, S. Pegg, E. Pettersen, I. Kuntz, N. Brooijmans, and R. Rizzo. Development and validation of a modular, extensible docking program: DOCK 5. *J. of Computer-Aided Molecular Design*, 20:601–619, 2006.

[13] H. W. Sorenson. *Kalman filtering: theory and application*, volume 38. IEEE Press, 1985.

[14] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, pages 633–652, September 2011.

[15] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, and I. Foster. Parallelizing the execution of sequential scripts. In *Proc. of the International Conf. on High Performance Computing, Networking, Storage and Analysis (SC13)*, 2013.