Using Application Skeletons to Improve eScience Infrastructure

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

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

***Abstract*—Computer scientists who work on tools and systems to support eScience (a variety of parallel and distributed) applications usually use actual applications to prove that their systems will benefit science and engineering (e.g., improve application performance). Accessing and building the applications and necessary data sets can be difficult because of policy or technical issues, and it can be difficult to modify the characteristics of the applications to understand corner cases in the system design. In this paper, we present the Application Skeleton, a simple yet powerful tool to build synthetic applications that represent real applications, with runtime and I/O 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. In addition, skeleton applications support simple reproducible system experiments since they are represented by a compact set of parameters.**

Our Application Skeleton tool (available as open source at https://github.com/applicationskeleton/Skeleton) currently can create easy-to-access, easy-to-build, and easy-to-run bag-of-task, (iterative) map-reduce, and (iterative) multistage workflow applications. The tasks can be serial or parallel or a mix of both. We select three representative applications (Montage, BLAST, CyberShake Postprocessing), then describe and generate skeleton applications for each. We show that the skeleton applications have identical (or close) performance to that of the real applications. We then show examples of using skeleton applications to verify system optimizations such as data caching, I/O tuning, and task scheduling, as well as the system resilience mechanism, in some cases modifying the skeleton applications to emphasize some characteristic, and thus show that using skeleton applications simplifies the process of designing, implementing, and testing these optimizations.

I. Introduction

Computer scientists who build tools and systems (programming languages, runtime systems, file systems, workflow systems, etc.) to enable eScience often have to work on real scientific applications to prove the effectiveness of the system. Accessing and building the real applications can be time consuming or sometimes infeasible for 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.

In addition, real applications may be difficult to scale or modify in order to demonstrate system trends and characteristics [8].

Our Application Skeletons idea was created in the AIMES project, whose goal is to explore the role of abstractions and integrated middleware to support eScience at extreme scales. AIMES is co-designing middleware from an application and infrastructure perspective. Thus, it requires applications with various characteristics for better application coverage. We have previously encountered many problems when accessing real applications and when trying to distribute applications and data as test cases to other researchers. Application Skeletons are intended to overcome these issues.

We previous presented the Skeleton idea of working around such issues by quickly and easily producing a synthetic distributed application that is executable in a distributed environment, for example, grids, clusters, and clouds [23]. The initial Application Skeleton tool took as input an application description file composed in a top-down approach: an application was described as a number of stages, and each stage had a number of tasks. Users specified the tasks at the stage level by articulating the number of tasks, task lengths, and input/output file sizes. Some of the parameters, such as task lengths and input file sizes, could be described as a statistical distribution. This Skeleton implementation could generate bag-of-task, map-reduce, and multistage workflow applications. The task implementation was based on UNIX/Linux sleep and dd commands. The skeleton applications were executable with common distributed computing middleware, Swift [27], [21], [22] and Pegasus [4], [3], as well as the ubiquitous UNIX shell on a single site (a local cluster with a shared file system). We measured the performance error between the skeleton application against real Montage [7], [9] and BLAST [15] applications on 64 processors of a BG/P supercomputer. Some stages had satisfying results while some had large performance gaps, as much as 13%.

In this paper, we show how the Application Skeleton tool has evolved, with improvements in multiple areas, so that the performance gap between skeleton applications and real applications has become much smaller, and we demonstrate the use of Application Skeletons in the design of a real middleware system: AMFORA [25]. At the application level, the latest Skeleton implementation inherits the original top-down approach of describing an application by stages. Applications now covered include bag-of-task, (iterative) map-reduce, and (iterative) multistage workflow applications. In terms of application task type, the applications can be composed of serial tasks, parallel tasks, or a mix of both. At the stage level, we improved input file and task mapping with a less ambiguous way of declaring distinct sources of multiple input files. At the task level, the new version of Skeleton has a completely new task design and implementation. The task is implemented as a versatile C program, and the compiled executable can be serial or parallel dependent on how it is compiled. Users can now specify a task's read/write buffer size, since such buffers are often used in real application code. The Skeleton task can mimic real application tasks' interleaving behavior for reads, writes, and computation. The tasks are C programs compiled to static executables, so they can run on supercomputers with an OS that doesn't have fork()/exec() support, such as the IBM Blue Gene/Q. Once the real application is represented by a Skeleton description file, simply specifying the skeleton parameters and selecting which tool or middleware to use means the skeleton application is easily distributed. And changing the parameters makes it easy to study how the system responds to different application characteristics.

Providing an easy-to-use way to specify a Skeleton application with acceptable performance difference between Skeleton applications and real applications was the major challenge of this work. In many cases, we had to relax the behavioral similarities between it and a real application to promote ease of programming. For example, while we want a skeleton task to mimic the exact behavior (operation sequence of computation and I/O) of a real application task, doing so would result in reimplementing the actual task and thus eliminating the ease of programming. We therefore provide only a limited number of operation interleaving options that users can choose from. A second challenge is the tradeoff between the ease of programming and expressiveness: we want Skeletons to cover iterative applications, but we don't want to fully implement a new control flow language. As a compromise, an iterative Skeleton application can have only a fixed number of iterations over stages; that is, we don't support dynamic convergence checking.

The contributions of this work include the following:

- An application abstraction that gives users good expressiveness and ease of programming to capture the key performance elements of distributed applications
- A versatile Skeleton task implementation that is configurable (serial or parallel, number of processes, read/write buffer size, input/output files, interleaving options)
- An interoperable Skeleton implementation that works with mainstream workflow frameworks and systems (Swift, Pegasus, and Shell)
- The usage of Skeleton applications to simplify system optimization implementation and highlight their impacts

The rest of the paper is organized as follows: Section II introduces the design of the Application Skeleton tool and the tradeoffs we made during the process. In Section III, we select three representative applications, and compare the Skeleton application performance against the real application performance. In Section IV, we show how application skeletons can help eScience infrastructure developers. Section V discusses related work and envisions future work. Conclusions are drawn in Section VI.

II. Application Skeleton Design

In addition to the design of application abstraction, stage characterization, and task configuration, the Application Skeleton's design involves other aspects, such as system tool compatibility and platform interoperability, to make it easy to build, easy to run, and easy to change. This section discusses the requirements in detail, the problems we encountered during the design process, and how we met those requirements and overcame the problems.

We are motivated by a wide variety of eScience application types for which we want to be able to build representative skeleton applications. The Application Skeleton implementation currently allows the user to express the following:

- Bag of Tasks: A set of independent tasks. Examples: MG-RAST [16], DOCK [18]
- MapReduce: A set of distributed application with key-value pairs as intermediate data. Examples: high energy physics histograms [5], object ordering [1]
- Multistage Workflow: A set of distributed applications with multiple stages using files for intermediate data. Examples: Montage [7], BLAST [15], CyberShake postprocessing [13]
- Iterative MapReduce: MapReduce with iterations. Example: graph mining [14]
- Campaign: An iterative application with a varying set of tasks that must be run to completion in each iteration. Example: Kalman filtering [20]

Application Skeletons of iterative applications are currently limited to those with a fixed number of iterations. Also, Application Skeletons can express multiple task types, serial, parallel, or a mix of both. The Application Skeleton concept ideally should also allow a concurrent application to be expressed, though this is not yet implemented. (A concurrent application comprises a set of tasks that have to be executed at the same time, e.g., coupled fusion simulation [10].) Examining several representative distributed applications, we observe that abstracting a distributed application stage by stage is expressive enough to cover the target applications.

A. Task Configuration

The core elements of tasks are computation and I/O. In the Application Skeleton design, time consumption of computation is represented by letting the task sleep for a user-specified period. A serial task only mimics the computation and I/O, since in many applications the communication between tasks is in the form of file production and consumption. Tasks with communication between processes (within a task) are referred to as parallel tasks.

An Application Skeleton task is implemented as a standalone C program. It requires only the standard math library and MPI library to preserve the portability. A serial task can be compiled with a standard C compiler, while a parallel task needs to be compiled with an MPI C compiler. Users specify the task type in the stage description, and the Application Skeleton tool will produce a compilation script that can compile the task code.

Other task properties include number of processes, task length, read buffer size, write buffer size, input files, output files, and operation interleaving option. The number of processes is one if the task is serial and is greater than one if the task is parallel. Task length is currently measured in seconds, which reflects the amount of computation in a task. An alternative would be flops. We select the runtime in seconds, because it is a direct indication of how long the task should run. Because the mapping of flops to time is, in general, not very predictable, we currently believe that the user can predict runtime more accurately than using flops. Additionally, we are currently investigating better ways to specify the task length that can reflect the difference of computing capacity on different machines.

A serial task reads input files, sleeps (to represent computation), and writes output files. In a parallel task, the rank 0 process reads input files, sleeps (in place of computation), then writes output files, while other processes simply sleep for the computation time length. All processes (including rank 0) wait at a barrier before the task exits.

We want to balance between mimicking the exact the operation sequence of the real application tasks' I/O and computation in the skeleton task, which requires reimplementing the actual task and results in poor programmability, and having very simple programming but very different performance. As a compromise, we define four interleaving options for a skeleton task (the serial task or the rank 0 process in a parallel task):

0) **interleave-nothing**: 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

B. Stage Characterization

We have previously shown basic examples of applications described using the Skeleton programming model [23]. In brief, we specify the task parameters at the stage level since it has a fairly global view of most of the task properties, which eases programming over specifying each task's parameters individually. The stage parameters are:

- **Stage_Name**: the stage name
- **Stage_Type**: the type of the tasks, serial or parallel
- **Num_Tasks**: the number of tasks
- **Num_Processes**: the number of processes per task
- **Task_Length**: the length of the tasks
- **Read_Buffer**: the read buffer size in the task
- **Write_Buffer**: the write buffer size in the task
- **Input_Files_Each_Task**: the ratio between number of input files and the number of tasks
- **Input_Source**: the source of the input files, either filesystem or outputs of a previously defined stage
- **Input_File_Size**: input file size for the stage, with distribution: uniform, normal, triangular, or lognorm
- **Input_Task_Mapping**: user-specified input file and task mapping, to support external mapping, letting the user override the mapping scheme implied by **Input_Files_Each_Task**
- **Output_Files_Each_Task**: 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
- **Interleave_Option**: the interleaving option of the task
- **Iteration_Num**: optional parameter specifying the number of iterations
- **Iteration_Stages**: the names of the other stages involved in the iteration
- **Iteration_Substitute**: the input file in tasks that will be replaced in the second and later iteration

Task_Length, Input_File_Size, and Output_File_Size can be stated as statistical distributions. The Application Skeleton tool currently supports uniform, normal, triangular, and lognorm distributions. Task_Length can also be specified as a polynomial function of Input_File_Size (of the first input file). Similarly, Output_File_Size can be specified as a polynomial function of Input_File_Size (of the first input file) or Task_Length.

C. Mapping Files and Tasks

Instead of declaring every input file's source and size, we describe the mapping with a combinatorial function, *combination Stage_1.Output 2*, interpreted as follows: choose two of the output files of Stage_1 as input files for each task. Choosing two from N files can have $\binom{N}{2}$ different file combinations. For example, 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}, and {output_2, output_3}. These six file pairs will be assigned to six tasks, so if there are six tasks, each will get a distinct file pair as its input files. (If there are more than six tasks, inputs will be repeated.)

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. An external script has to print the input files names of a task in a line, and a Python function needs to return a nested list of input file names.

If the 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 sequentially: the first and second file are mapped to the first task, the third and fourth 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.

D. Iteration Support

While many multistage workflows execute each stage once, some involve iteration, which we had not previously implemented [23]. Application Skeletons now supports one or more stages that are executed a fixed number of times. The optional Iteration_Num parameter declares the number of times a single stage should be executed. If Iteration_Num is used, Iteration_Stages optionally specifies which stages are included in this iteration in addition to the current stage, and Iteration_Substitute optionally specifies which input file of the tasks in the current stage will be replaced in the second and later iterations. Iteration support in Application Skeletons requires that the number of output files of the last stage in the iteration be the same as the the number of initial input files that are going to be replaced in the iterations.

One example is a single stage that iterates three times with the first input file for all tasks in this stage replaced by the output files from the previous iteration, after the first iteration. Application Skeletons uses the iteration number to differentiate the output files from each stage. In the first iteration, the stage consumes the input files declared in description (e.g., Stage_1_Input) and produces output files with names that include the iteration number (e.g., Stage_1_Output_Iter_1). Starting with the second iteration, the stage consumes the output files from the previous iteration and increases the iteration number, then produces the outputs. In the last iteration, the synthetic stage produces output files with the names that are declared in the application description file.

E. System Tool Compatibility and Multiple Sites

The Application Skeleton tool is implemented with Python, compatible with both Python2 and Python3. It reads an application configuration file, parses the text, and sets parameters for each stage. It then generates three types of files on the local site where it is launched:

> **Preparation scripts**, to produce the input/output directories and input files for the Skeleton application
> **Compilation scripts**, to compile the task source code into executables
> **Application**, the overall skeleton application, which can be implemented in one of three formats: a plain Bash shell script, with the command lines of each stage in a distinct script file; or a Pegasus DAG task description, with task, data and dependency declaration generated automatically; a functional Swift script that represents the complete application.

This design works well on a single site, such as a single computer or a local homogeneous architecture cluster with a shared file system. However, we also need to support distributed environments, such as running one application on multiple sites, where the sites can have heterogeneous architectures, resulting in difficult task compilation and deployment. We address this issue by asking the user for site-specific information; site-specific information includes site name, serial C compiler path, MPI C compiler path, compilation flags, and working directory. Then the Application Skeleton tool generates compilation scripts for each site. Those scripts are run on the remote machines to transfer the task source code and compile it.

III. Performance Evaluation

To examine and understand the differences between skeleton and real application performance, we select three applications (Montage, BLAST, and CyberShake PostProcessing), profile the application properties, produce the skeleton versions with the Application Skeleton tool, and compare the skeleton and real performance for each computation stage.

We use 64 BG/P processors with GPFS as the shared file system. Each of the processors has a ∼500MB RAM disk. Each application stage is executed with AMFORA [25], a parallel scripting framework on supercomputers. With AMFORA, we can simply list all tasks of a stage in a Bash script, and instruct AMFORA to execute them (in parallel, subject to data dependencies.)

To produce skeleton applications, we first find the stage parameters. Each task's length is measured as the time-to-solution with inputs and outputs cached in RAM disk. Although this method over counts the I/O time from/to RAM disk as task length, this is negligible compared with the time consumption of GPFS I/O due to the high-volume traffic with high concurrency. I/O traffic is profiled by collecting all system calls with the Linux strace command. As we have done previously [26], we align I/O-related system calls of all tasks in a stage using sequence order. Assuming all tasks get to the same system call at the same time, we can determine I/O concurrency and I/O buffer size.

To compare the performance of the skeleton and real applications, we run both with AMFORA. The tasks read/write files from/to GPFS. We run each stage five times and average the times-to-solution.

A. Montage

The Montage application in this work has eight stages. We build a 6x6 degree mosaic from 1,319 2MASS image files. Each file is ∼2 MB. The output of the last stage, mAdd, contains two files, each of ∼3.7 GB. Table I shows basic statistics of each stage. Measured Time Avg and Measured Time Stdev show the average and the standard deviation of the time-to-solution of all tasks in each stage, respectively.

For most stages, we place the input and output files on RAM disk and use the average time-to-solution as the task length, as previously stated. However, the input sizes for mImgtbl and mAdd exceed the maximum RAM disk size, so we cannot use this technique. We observe that mAdd task's time-to-solution is proportional to the number of input files when that 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. This method did not work well for mImgtbl in our previous study [23], so for this stage, we measure the time to copy the input data set from GPFS to RAM disk by directing the traffic to /dev/null and measure the time to copy output data from RAM disk to GPFS. We then subtract these two times from the time-to-solution of mImgtbl measured on GPFS and use the result as the estimated task length.

Profiling Montage tells us that the application uses a 64-KB buffer size for read and write and that the reads and writes of mBackground mostly don't overlap. Thus we set the interleaving option of mBackground to 0–interleave-nothing. The other Montage stages also have this interleaving behavior, and they are also set to 0. The input file sizes of each Montage stage are almost uniform, so we use a uniform set of skeleton parameters for each stage. One exception is for mDiffFit, where we previously found a performance gap between the skeleton and real mDiffFit of about 13% [23] when we set the Skeleton parameters to match the real number of files (two inputs and one output) and sizes. Examining the system call trace shows us that each mDiffFit task reads each input file

TABLE I. 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 (s)	Measured Time Stdev	Skeleton Task Length
mProject	1319	1319	2594	2800	10400	11.6	2.5	uniform 11.6
mImgtbl	1	1297	1	5200	0.8	N/A	0	uniform 30.1
mOverlaps	1	1	1	0.8	0.4	9.1	0	uniform 9.1
mDiffFit	3883	7766	7766	31000	487	1.8	0.6	uniform 1.8
mConcatFit	1	3883	1	1.1	4.3	2.1	0	uniform 2.1
mBgModel	1	2	1	4.5	0.07	288	0	uniform 288
mBackground	1297	1297	1297	5200	5200	0.4	0.08	uniform 0.4
mAdd	1	1297	2	5200	7400	N/A	0	uniform 519

TABLE II. TIME-TO-SOLUTION COMPARISON OF SKELETON MONTAGE AND REAL MONTAGE (SECONDS)

	mProject	mImgtbl	mOverlaps	mDiffFit	mConcatFit	mBgModel	mBackground	mAdd	Total
Montage	282.3	139.7	10.2	426.7	60.1	288.0	107.9	788.8	2103.7
Skeleton	281.8	136.8	10.0	412.5	59.2	288.1	106.2	781.8	2076.4
Error	-0.2%	-2.1%	-0.2%	-3.3%	-1.5%	0.03%	-1.6%	-0.9%	-1.3%

TABLE III. 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 (s)	Measured Time Stdev	Skeleton Task Length
split	1	1	64	3800	3800	0	N/A	0
formatdb	64	64	192	3800	4400	41.9	0.1	uniform 42
blastp	1024	4096	1024	70402	966	109.2	14.9	normal[109.2, 14.9]
merge	16	1024	16	966	867	4.4	4.1	normal[4.4, 4.1]

four times instead of one. So we set the Skeleton parameters to eight input files, repeating the two input file names four times. This procedure produces good results, as shown in Table II.

Comparing with previous measured error [23], we have now reduced the error of all eight stages by between 22.8% and 91.8%. The errors of all stages is now less than 4%, and four are under 1%. The error for the complete application is -1.3%.

B. BLAST

BLAST is a widely used sequence alignment tool. The version we use here is parallelBLAST [15], which has four stages. The first stage partitions a 3.8 GB data base into slices. The second stage formats each partition. The third-stage tasks query each formatted database partition with an identical set of query sequences. The fourth stage merges the partial results from all partitions into an output file for each query sequence file (since there could be multiple query sequences in one file). In our experiments, we run the first 1,024 queries of the NRxNR test case. Table III shows the basic statistics of each BLAST stage.

In general, we measure the time-to-solution of each task by executing them with the inputs and outputs on RAM disk. The split stage has a single task that reads the whole database and partitions it into several slices, so no significant computation is involved. We simply set the split task's length to zero. The I/O size of the formatdb, blastp, and merge stage tasks fit the BG/P compute node's RAM size, so we measure all times-to-solution directly. The times-to-solution of the formatdb stage is uniform, so we set all task lengths of this stage to that value. The time-to-solution distribution of the blastp stage varies, so we used a normal distribution described by the average and standard deviation, although we also attempted to use a uniform value, since the actual distribution is between the two. We also used a normal task length distribution for the merge stage.

All tasks of the four stages of BLAST read each input file just once. Each formatdb task reads one input file of ∼60 MB and writes three output files of size 56 MB, 16 MB, and 1 MB, respectively. We simplify the skeleton formatdb task with three output files with identical size of 21 MB, since this setting results in the same amount of metadata operation and I/O traffic.

TABLE IV. TIME-TO-SOLUTION COMPARISON OF SKELETON BLAST AND REAL BLAST (SECONDS)

	split	formatdb	blastp	merge	Total
BLAST	74.4	82.1	1996.3	35.9	2188.7
Skeleton	72.9	81.6	2028.9	36.3	2219.7
Error	-1.9%	-0.6%	1.6%	1.1%	1.4%

The split task uses 4096 Bytes as its read and write buffer size. Profiling of the formatdb stage showed us that each formatdb task reads the database with a 64-KB buffer. However, there are roughly 500,000 writes with a size of hundreds of Bytes. With our previous Skeleton implementation, we could not specify a buffer size and had to run the real and skeleton applications with all data in RAM to get close to real performance. With the new implementation, we are able to set the formatdb read buffer to 64 KB and the write buffer to 512 Bytes. In practice, 64 concurrent tasks each with ∼500,000 small writes can run for hours on GPFS. So when we compare the Skeleton formatdb and real formatdb, we run the real formatdb with data caching in RAM disk and run the skeleton formatdb in a similar way. For blastp and merge stage, we run the tasks with data in GPFS.

For all tasks in the four stages of BLAST, we see an interleaved computation and I/O pattern. In formatdb, each task reads a sequence, formats that sequence, and writes the result to output files. Thus, there are ∼500,000 writes with a size of hundreds of Bytes. In blastp, the task reads a sequence, computes the similarity score between this sequence and every sequence in one database partition, and writes the result. The merge task also interleaves its work. So we set the interleaving options as 3–interleave-all. The skeleton tasks read a piece of the input file, sleep for some time, and write a piece of the output, repeating these steps until all work is done.

Table IV compares the measured performance of the Skeleton BLAST and real BLAST. The error of each stage is -1.9%, -0.6%, 1.6%, and 1.1%, respectively. The measurements of the

TABLE V. NUMBER OF TASKS, INPUTS, AND OUTPUTS, AND INPUT AND OUTPUT SIZE, FOR EACH CYBERSHAKE POSTPROCESSING STAGE

Stage	# Tasks	# Inputs	# Outputs	In (MB)	Out (MB)	Measured Time Avg (s)	Measured Time Stdev	Skeleton Task Length
Extract	128	130	256	5400	11000	6.39	2.2	uniform 6.39
Seis	4096	4352	4096	11000	96	26.9	13.3	normal[26.9, 13.3]
PeakGM	4096	4096	4096	96	1.4	0.23	0.03	uniform 0.23

TABLE VI. TIME-TO-SOLUTION COMPARISON OF SKELETON CYBERSHAKE AND REAL CYBERSHAKE (SECONDS)

	Extract	Seis	PeakGM	Total
CyberShake	571.5	2386.5	81.5	3039.4
Skeleton	586.3	2443.3	83.3	3112.9
Error	2.6%	2.4%	2.3%	2.4%

skeleton blastp stage uses the normal task length distribution. (Our attempt with a uniform distribution had a larger error: -2.7%.) The overall application error is 1.4%.

C. CyberShake

CyberShake [13], [6] finds the probabilistic peak ground movement at a physical site caused by a set of potential earthquakes. The first step of the application generates strain Green tensors (SGTs), by running two MPI calculations that each produce one SGT file for the physical site. The second step is Postprocessing, which performs a parameter sweep over two dimensions: ruptures that could affect the site, and the variations of every rupture. For each rupture, Postprocessing extracts a subset of the SGT files, and then for each variation of that rupture, Postprocessing calculates a computational seismogram and finds the peak ground motion in the seismogram. In our experiments, we run the first 128 Extract tasks, and 4,096 Seis and PeakGM tasks. Table V shows the basic statistics of each CyberShake Postprocessing stage.

Our methodology to determine a skeleton task length is to use the time-to-solution of the real task with data cached in RAM disk. However, each Extract task needs to access the two SGT files and a rupture variation file, and the SGT file's size is 5.4 GB, which is too large for a compute node's RAM disk. So instead we ran the first 50 tasks manually on each compute node in sequence order, measured the times-to-solution and then computed the average and standard deviation. Although the tasks access data on GPFS, running one task at a time excludes the overhead incurred by highly concurrent I/O operations. The Seis and PeakGM tasks' I/O fits in RAM disk, so we directly measure the time-to-solution of each task. For task lengths of the Seis stage, we use a normal distribution specified by the average and standard deviation. The task lengths of the PeakGM tasks is uniform.

The two SGT input files for each Extract task are about 2.7 GB each. We observe from the strace profile that an Extract task does not read the whole input file; rather it reads only about 200 MB. So we set the skeleton parameters for two SGT input files with size of 200 MB, which preserves the I/O traffic that actually happens in an Extract task. We set the output files of the Extract, Seis, and PeakGM stage to a size of 34 MB, 36 KB, and 240 Bytes, respectively. The interleaving option for all three stages is set to 0–interleave nothing.

In the real CyberShake Postprocessing application, the file usage of the Extract output files (a subset of the SGT files) in the Seis stage is not uniform. Some files are used more than others. Computer scientists who want to implement an automatic file usage-based runtime replication system might find it infeasible to specify such unbalanced file usage with the current Application Skeletons implementation. One solution for this problem is the external file option, which can customize file mappings to tasks and can generate uneven file usage. Here however, we simply set the Seis tasks to read each pair of Extract output files evenly, which suffices to give performance matching that of the real application on our test system.

Table VI presents the comparison between the skeleton CyberShake Postprocessing applications and the real applications. All three stages have errors under 3%, with a total error of 2.4%.

IV. USING APPLICATION SKELETONS

We show four examples of using application skeletons for system improvements: data caching, task scheduling, I/O tuning, and resilience. Various system optimizations are implemented with the AMFORA [25] system. The experiments are carried out on the Google Compute Engine environment (referred to as GCE in the rest of the manuscript). Throughout all these experiments, we use the "n1-highmem-2" instances, which have two vCPU cores and 13 GB RAM.

A. Data Caching

Data caching is a common technique when application data fits in RAM. Here, we run real-mProjectPP from Montage and skeleton-mProjectPP. with files located on PVFS and AMFORA to show improved performance. PVFS and AMFORA use four GCE compute nodes. The real-mProjectPP workload finishes in 285.2 s with PVFS and 100.9 s on AMFORA. Skeleton-mProjectPP runs in 273.7 s on PVFS and 101.3 s on AMFORA. The data-caching optimization on skeleton-mProjectPP workload (63.0%) has identical improvement on the real-mProjectPP workload (64.6%). This example shows that skeleton applications can be used for designing, implementing, and testing the system optimizations, in place of the more complex real applications.

B. Task Scheduling

In this example, we seek to show the time-to-solution improvement of data-aware scheduling over the FIFO (first-in, first-out) scheduling algorithm. We implement both scheduling algorithms in the AMFORA task engine and run both real-mProjectPP and skeleton-mProjectPP on 16 compute nodes of GCE. The real-mProjectPP workload and skeleton-mProjectPP workload both have insignificant improvements of 0.7% and 1.6%, respectively, because of the nature of the mProjectPP tasks. The ratio of I/O to task length is the root cause of these insignificant improvements. When we modify the skeleton-mProjectPP with a 5x larger input file size, we can see a 16.4% time-to-solution improvement with data-aware scheduling over

FIFO. This shows an additional benefit of skeleton applications: we can use them in place of real applications and easily modify them to better understand when system optimizations will have a significant effect.

C. I/O Tuning

For applications that have highly concurrent and frequent metadata operations, using multiple metadata servers can improve the overall application performance [19], [2]. We built AMFORA with a configurable metadata server design, and we want to use the mProjectPP workload to verify that multiple metadata servers can actually improve the performance with a single metadata server. With 16 compute nodes of GCE, real-mProjectPP and skeleton-mProjectPP show only 1.1% and 1.2% improvements, respectively, with multiple metadata servers over a single metadata server, although the improvement is stable. The marginal improvement is again due to the nature of the mProjectPP tasks: the task execution is dominated by computation, which makes improvement in metadata access negligible. When we modify the skeleton-mProjectPP task with a 10x shorter task length, we see a 31.2% improvement with multiple metadata servers over a single metadata server. Similar to the last example, this shows a benefit of using skeleton applications over real applications.

D. Resilience Mechanism

In the AMFORA system resilience design, we proposed a dynamic replication approach that combines both task re-execution and file replication. The decision is made by calculating the expected recovery time using each replication strategy and choosing the one with lower overhead. Originally, we used the mBackground stage from Montage since it has a mix of tasks: some tasks' output files should be recovered by re-execution, while some should be recovered by file replication. We want to show that this dynamic replication approach has the best recovery performance over either pure re-execution or pure file replication when there is a node failure during execution. We ran the workload on various scales on GCE, but the improvements are marginal. The reason is that the recovery overhead difference of mBackground tasks is not significant, so even if we make the right decision for every task, the accumulated improvement is still not significant. To prove the benefit of this dynamic resilience strategy, we need to show a more significant improvement of the recovery performance. We first profiled the mBackground tasks' time-to-solution and I/O parameters, and generated a skeleton-mBackground workload with similar performance as real-mBackground. We then provisionally executed the skeleton-mBackground workload with AMFORA and recorded which files were replicated by re-execution and replications. We next modified the skeleton-mBackground workload to have 10x longer tasks for the tasks whose output files should be replicated, and 10x larger output files for the tasks whose output files should be recovered by re-execution. In other words, we magnified the penalty of the wrong replication decision. With this modified skeleton-mBackground work load, we see an increased improvement of 9.1%, 4.4%, and 4.9% on 4, 16, and 64 compute nodes of GCE, respectively.

V. Related Work

We believe that the idea of using application skeletons, particularly for overall system performance, is relatively novel. Skel [12] 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 [11]. WGL [17] abstracts an application from a dataflow point of view and lets users generate a Swift script for a workflow application by describing the dataflow patterns between stages.

VI. Conclusion and Future Work

Application Skeletons are motivated by the difficulties of real application access and modification that computer scientists who work on tools and systems have. We use a top-down approach to abstract a distributed application in Application Skeletons: applications are composed of a number of stages, while each stage is composed of a number of tasks, with task parameters specified at the stage level. The stage is built on a versatile task design that can handle different task types, various buffer sizes, multiple input/output files, and different interleaving scenarios. Overall, one can create easy-to-access, easy-to-build, easy-to-change, and easy-to-run bag-of-tasks, (iterative) map-reduce, and (iterative) multistage workflow applications. Our Application Skeleton tool is designed to work with mainstream workflow frameworks and systems: Bash Shell, Pegasus, and Swift.

Representing an application by a set of skeleton parameters means that the skeleton application can be easily shared, making middleware and tool experiments more reproducible. Changes in these parameters can be used to study particular aspects of system performance. Computer scientists can then focus on the system or tools they are building.

We have shown that the skeleton applications generated by our Application Skeleton tool have performance close to that of the real applications. We profiled three representative distributed applications—Montage, BLAST, CyberShake PostProcessing—to understand their computation and I/O behavior and then derived parameters required to specify the skeletons.

The Application Skeleton tool can produce skeleton applications that correctly capture important distributed properties of real applications but are much simpler to define and use. Comparing performance shows overall errors of -1.3%, 1.5%, and 2.4% for Montage, BLAST, and CyberShake PostProcessing, respectively. At the stage level, fourteen out of fifteen stages have errors of less than 3%, ten have errors less than 2%, and four have errors of less than 1%.

The four examples of system improvements (data caching, task scheduling, I/O tuning, and resilience mechanism) showed that using skeleton applications simplifies the process of system optimization design, implementation, and verification, and that by making small changes to the skeletons, we can highlight the effects of optimizations.

The Application Skeleton code is available as open source at https://github.com/applicationskeleton/Skeleton [24] under the MIT license. We invite the community to try it and to contribute to it.

We plan the following in the near future:

- Use application trace data to produce skeleton 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 in a task that when combined with a particular platform can give an accurate runtime for that task.
- Support concurrent tasks that need to run at the same time to exchange information.
- Test on distributed systems where latencies, particular file usage, and other issues may be more important than on the parallel systems and cloud environments.

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 and Google. 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.

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] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[3] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In M. D. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lect. Notes in Comp. Sci.*, pages 131–140. Springer, 2004.

[4] 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.

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

[6] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi. CyberShake: A physics-based seismic hazard model for southern California. *Pure and Applied Geophysics*, Online Fir:1–15, May 2010.

[7] 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.

[8] S. Jha, M. Cole, D. S. Katz, M. Parashar, O. Rana, and J. Weissman. Distributed computing practice for large-scale science and engineering applications. *Concurrency and Computation: Practice and Experience*, 25(11):1559–1585, 2013.

[9] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, and G. Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *Proc. 2005 Intl. Conf. on Par. Proc. Works.*, pages 85–94, 2005.

[10] 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.

[11] 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.

[12] 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.

[13] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghãn, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan, and E. Field. SCEC CyberShake workflows – automating probabilistic seismic hazard analysis calculations. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 143–163. Springer, 2007.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] S. Patil and G. Gibson. Scale and concurrency of GIGA+: file system directories with millions of files. In *Proc. of 9th USENIX Conference on File and Storage Technologies*, pages 13–13. USENIX Association, 2011.

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

[21] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009.

[22] 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.

[23] Z. Zhang and D. S. Katz. Application Skeletons: Encapsulating MTC application task computation and I/O. In *6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) 2013*, 2013.

[24] Z. Zhang and D. S. Katz. Application Skeleton v1.1. http://dx.doi.org/10.5281/zenodo.11096, Jul 2014.

[25] 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.

[26] Z. Zhang, D. S. Katz, M. Wilde, J. Wozniak, and I. Foster. MTC Envelope: Defining the capability of large scale computers in the context of parallel scripting applications. In *Proc. of 22nd ACM International Symposium on High Performance Distributed Computing*. ACM, 2013.

[27] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *IEEE Congress on Services*, pages 199–206. IEEE, 2007.