ARTICLE IN PRESS

Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Application skeletons: Construction and use in eScience

Daniel S. Katz [a,*], Andre Merzky [b], Zhao Zhang [c], Shantenu Jha [b]

[a] Computation Institute, University of Chicago & Argonne National Laboratory, Chicago, IL, USA

[b] RADICAL Laboratory, Rutgers University, New Brunswick, NJ, USA

[c] AMPLab, University of California, Berkeley, CA, USA

HIGHLIGHTS

- Skeleton applications represent the key parameters of parallel and distributed eScience applications.
- Skeleton applications are easy-to-program, easy-to-build, and easy-to-use, and easy-to-share.
- Skeleton applications have similar performance to the real applications on which they are based.
- Skeletons application are built from an open source: https://github.com/applicationskeleton/Skeleton.
- Skeleton applications can be used to demonstrate system optimizations.

ARTICLE INFO

Article history:
Received 24 March 2015
Received in revised form
18 September 2015
Accepted 3 October 2015
Available online xxxx

Keywords:
Computational science
Data science
Application modeling
System modeling
Performance modeling
Parallel and distributed systems

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, parallel, or a mix of both. The parameters to represent the tasks can either be discovered through a manual profiling of the applications or through an automated method. 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.

Published by Elsevier B.V.

1. 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 are difficult to access.
- Some applications use legacy code and are dependent on out-of-date libraries.

* Corresponding author.
E-mail addresses: d.katz@ieee.org (D.S. Katz), andre@merzky.net (A. Merzky), zhaozhang@eecs.berkeley.edu (Z. Zhang), shantenu.jha@rutgers.edu (S. Jha).

http://dx.doi.org/10.1016/j.future.2015.10.001
0167-739X/Published by Elsevier B.V.

- Some applications are hard to understand because they implicitly assume domain knowledge.

In addition, real applications may be difficult to scale or modify in order to demonstrate system trends and characteristics. Our work is partially motivated by the state of distributed applications. We previously highlighted the challenges of developing distributed applications, showing that the lack of development abstractions and the complexity of deployment were two important barriers [1,2].

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 previously presented [3] 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, and then showed [4] improvements in reducing the gap between skeleton and real application performance, as well as showing Skeleton applications could be used to simplify understanding and demonstrating system optimizations, such as AMFORA [5].

The Application Skeleton tool takes 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 specify tasks at the stage level by articulating the number of tasks, task lengths, and input/output file sizes. Applications can be composed of serial tasks, parallel tasks, or a mix of both. 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 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 does not have fork()/exec() support, such as the IBM Blue Gene/Q. Some of the task parameters, such as task lengths and input file sizes, can be described as a statistical distribution. The task implementation is based on the UNIX/Linux sleep and dd programs, controlling the CPU time and I/O, respectively.

This Skeleton implementation can generate bag-of-task, (iterative) map-reduce, and (iterative) multistage workflow applications. The skeleton applications are executable with common distributed computing middleware, Swift [6–8] and Pegasus [9,10], as well as the ubiquitous UNIX shell on a single site (a local cluster with a shared file system), and the skeleton set of tasks can also be output as a generic JSON object that can be used by other systems, such as in our case, by our AIMES middleware.

Skeleton parameters can be discovered by either by manually or automatically profiling the application. Once the real application is represented by a Skeleton description file, it can easily be distributed and reused. We measured the performance error between the skeleton application against three real applications (Montage [11,12], BLAST [13], and CyberShake PostProcessing [14]) on 64 processors of a BG/P supercomputer with per stage and total errors of between 1 and 3%. And changing the parameters makes it easy to study how the system responds to different application characteristics, specifically data caching, task scheduling, I/O tuning, and resilience mechanism.

This paper summarizes our previous work [3,4] and adds further work toward automated, system independent application profiling. Specifically, almost all of Section 4 is new, and Section 2 (related work) has been expanded to provide better context for our work; as such, it also has been moved to earlier in the paper. Sections 3 and 5 were initially shortened to focus on key points and then modified to include automated profiling, then further modified as part of the review process. The introduction (Section 1) and conclusions (Section 7) have been reworked for this paper but used the introductions and conclusions of the previous papers as starting points. Section 6 has just been condensed from [4].

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).
- A comparison of two methods for the estimation of Skeleton parameters: manual, system specific application profiling versus automated, system independent profiling.
- An interoperable Skeleton implementation that works with mainstream workflow frameworks and systems (Swift, Pegasus, and Shell), and allow general output for other systems.
- The usage of Skeleton applications to simplify system optimization implementation and highlight their impacts.

The rest of the paper is organized as follows: Section 2 discusses the idea of application modeling, including related work. Section 3 introduces the design of the Application Skeleton tool and the tradeoffs we made during the process. In Section 4, we discuss how Skeleton parameters can be determined. In Section 5, we select three representative applications, and compare the Skeleton application performance against the real application performance. In Section 6, we show how application skeletons can help eScience infrastructure developers. Conclusions are drawn in Section 7.

2. Modeling background and related work

Researchers have been using application replacements of various types (e.g., kernels, benchmarks, reduced applications, miniapps, traces) for experiments for a long time, and some of those replacements have been and are certainly tunable in one way or another. These application replacements are used for a variety of purposes, including:

- They are easier to build than the actual application.
- They can be reused across different system to compare those systems.
- They run faster than the actual application.
- They can be shared with collaborators who do not have access to the application or the data.

However, we believe that the idea of using application skeletons, particularly for overall system performance, is relatively novel. The distinction in our work is that the Skeletons provide a systematic application replacement capability, which both preserves the significant part of the application's behavior and is tunable across a range of diverse applications. Because of the problem space is pervasive but attempts to solve it systematically are rare, particularly for distributed applications, the related work we discuss here are examples of some of the different types of application replacements that have been used.

Examples of benchmarking/analysis work in a parallel (not distributed) context include the NAS Parallel Benchmarks [15], and Berkeley Dwarfs [16] (also called 'motifs') and the related OpenDwarfs [17]. The NAS benchmarks include both kernels and pseudo-applications (an example of reduced applications), intended for use on parallel (not distributed) computers. These

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

benchmarks can be scaled by choosing different predeveloped classes. However, they cannot be arbitrarily tuned to expose particular system features, as skeletons can. The dwarf work has at its core the idea that there is a key computational or data movement kernel that is responsible for most of the application's performance, and that that kernel can be extracted and used to represent the full application, at least in terms of performance. The CORAL set of benchmark codes [18], intended for HPC vendors, include a set of "Skeleton Benchmarks", but this term is used to refer to benchmarks that each focus on a specific platform characteristic, unlike our application skeletons.

Additional examples are from Kerbyson et al. [19], who used simplified version of parallel MPI applications to study Blue Gene systems; and Worley et al. [20,21], who studied a parallel spectral transform shallow water model by implementing the real spectral transform in what is otherwise a synthetic code that replicates a range of different communication structures as found in different parallelizations of climate models. Similarly, Prophesy [22] is an infrastructure that helps in performance modeling of applications on parallel and distributed systems through a relational database that allows for the recording of performance data, system features and application details. The applications data gathered has some overlap with the processes by which we can obtain our skeleton parameters.

Miniapps are another tool for improving the performance characterization of high-performance applications. Two specific projects are Mantevo [23], which provides dominant numerical kernels contained in an actual "stand-alone" application, predicated upon the fact that the performance critical sections of applications are fairly localized and MADbench [24], a comprehensive miniapp that approximates the behavior of a cosmic background radiation application, including performing compute and I/O operations. The more general DOE Miniapps [25] project focuses on the programming models and communication patterns of a range of applications from material sciences to combustion.

An alternative to reducing the application over the space of the application is reducing it over time. For example, Sodhi et al. [26] developed a tool called performance skeletons. These are short running parallel MPI programs with performance meant to fractionally represent the performance of the full applications. Unlike the dwarf work, this assumes that the work done by applications is spread through out the application (not in a single kernel). This idea of fractional versions of parallel applications is also unlike our work, which produces simplified and scalable versions of full distributed applications. Our work overlaps with Sodhi's in its approach to automatically capturing the execution behavior of an application and automatically generating a synthetic skeleton program that reflects that execution behavior; however, Sodhi focuses on single system MPI applications that do not have disk I/O.

Skel [27] uses a similar idea, though it aims at understanding 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 [28]. There are a number of other I/O benchmarking projects related to applications, but they are generally focused just on the I/O aspects of the applications [29]. Another similar idea is found in the Tigres project [30], which has a concept of templates that could potentially be used as an alternate method to define skeletons, though the skeletons described by these templates would more limited than the skeletons described using our method.

A project focusing on single set of applications, for the ATLAS high energy physics experiment [31], included workload generator software that has some similarity with our application skeletons, though it is much less general.

The work that is probably closest to ours is WGL [32], which abstracts an application from a dataflow point of view and lets users generate a Swift script for a potentially distributed workflow application by describing the dataflow patterns between stages. Since there is some institutional overlap between the WGL developers and Skeleton developers and because both overlap the Swift team, it is likely that informal discussions carried some of this concept from WGL to Skeletons, though there is a nine-year gap between the projects.

Another application replacement is system traces, used when the analyzers do not have access to the applications or the data. Some examples of system traces can be found in: Chen et al. [33], who analyzed a collection of industrial MapReduce workload traces and built a big data processing benchmark for MapReduce systems; Harter et al. [34], who studied the underlying HDFS traces to understand the performance of the Facebook message service, then proposed potential improvements; and Ousterhout et al. [35], who analyzed the Spark traces of industrial applications to study potential improvements of potential hardware upgrades.

Loosely related work includes Holl et al. [36], who have studied how to store and share parameters values for shared workflows, an idea that could be applied to skeleton values; and algorithmic motifs (formerly called skeletons) [37], which abstracts parallel application with an algorithmic approach, and lets users build parallel programs based on the algorithmic motifs. The term skeleton has also been used in SKOPE, which produces a descriptive model about the semantic behavior of a workload, which can infer potential transformations and help users understand how workloads may interact with and adapt to emerging hardware. SKOPE models can be shared, annotated, and studied by a community of performance engineers and system designers.

3. 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 [38], DOCK [39].
- MapReduce: A set of distributed application with key–value pairs as intermediate data. Examples: high energy physics histograms [40], object ordering [41].
- Multistage Workflow: A set of distributed applications with multiple stages using files for intermediate data. Examples: Montage [11], BLAST [13], CyberShake postprocessing [14].
- Iterative MapReduce: MapReduce with iterations. Example: graph mining [42].
- Campaign: An iterative application with a varying set of tasks that must be run to completion in each iteration. Example: Kalman filtering [43].

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. Examining several representative distributed applications, we observe that abstracting a distributed application stage by stage is expressive enough to cover the target applications. The Application Skeleton concept ideally should also allow a concurrent application (a set of tasks that have to be executed at the same time, e.g., coupled fusion simulation [44]) to be expressed; this is future work.

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

3.1. 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. Section 4.3 discusses the tradeoffs between this approach and an alternative representation of compute load, via FLOPs.

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 and having very simple programming. The former requires reimplementing the actual task and results in poor programmability, while the latter gives 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.

3.2. Stage characterization

We have previously shown basic examples of applications described using the Skeleton programming model [3]. 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.

3.3. 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.

3.4. Iteration support

While many multistage workflows execute each stage once, some involve iteration, which we had not previously implemented [3]. 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

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

last stage in the iteration be the same as 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.

3.5. 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.

4. Determining skeleton parameters

In this section, we describe how Skeleton parameters are determined, both manually and automatically. We also discuss how automatic generation of Skeleton parameters can lead to Skeletons that are system independent.

4.1. Manual application profiling

Skeleton parameters can be determined by manually running and analyzing an application. Specifically, to measure task length, we place both input and output files on RAM disk and use `'time'` to measure the time-to-solution of the tasks. Although this method includes the I/O time from/to RAM disk as part of the task length, this extra time is generally negligible compared with the I/O time using shared disk, which involves high-volume traffic and high concurrency. After gathering this data, it can be examined to determine the average task length and variance, and this can then be used to generate the skeleton parameters that give the needed task length distribution. `'strace'` is used to profile the I/O traffic, determining the number of reads and writes and the amount of data that is read and written by a task. As we have shown previously [45], I/O-related system calls of all tasks in a stage can be aligned using sequence order. By assuming all tasks get to the same system call at the same time, I/O concurrency and I/O buffer size can be determined. This also can be used to determine the interleaving pattern between computation and I/O. While this process is manual and a bit time-consuming, it has also proven to generally be successful [4], though with some modifications needed, as described in Section 5.

4.2. Automated application profiling via SYNAPSE

The process of manual application profiling as described in Section 4.1 is time consuming. As an alternative approach, we investigated the use of *SYNAPSE* [46,47], an automated application profiler and emulator. While SYNAPSE requires specific system preparations for its profiling step (specifically Linux kernel support for low level hardware counters), it fully automates the application profiling at a comparable level of granularity as the manual approach. In many ways, it is complementary to the work described in [45]: the latter introduces the notion of a *system's envelope* to describe system performance in an application independent way, whereas SYNAPSE describes application performance independent of the system's envelope, by focusing on system independent performance metrics.

4.3. System independent profiling

Most, but not all, of the application characteristics listed in Section 3.2 are system independent, i.e., they have values that allow faithful reproduction of application behavior on target systems. For example, the data input size (in Bytes) is constant, whether the application is executed on a local desktop, on a remote cluster, or on a supercomputer. Some metrics are, however, not system-independent. Specifically, the application runtime depends on the specific resource performance, such as the CPU type and clock speed.

The underlying reason is that some metrics do not really denote metrics of *resource consumption*, but rather describe the *effect* of that consumption. This is most evident for a metric which is specified as time duration or frequency, but it can similarly apply to the concurrency of disk I/O, memory I/O, network I/O and computation, whose realization and performance can depend on the underlying system architecture. For example, the manual application profiling described in Section 4.1 outlines how RAM disks are used to separate the effects of I/O on concurrent computation—exactly because the computation is measured as runtime, which is influenced by the overall system performance and load.

SYNAPSE addresses this by estimating the number of FLoating-point OPerations (FLOPs.[1]) performed by the application, which represents its compute resource consumption, and which is thus independent of the specific target system's architecture, configuration, and load. While it would be interesting to apply similar system independent metrics to other resources types (for example, as mentioned, the I/O-Compute concurrency), we found it difficult to extrapolate the respective parameters without a complete model of the hardware and OS layer. Without resorting to code inspection, it is difficult to distinguish if I/O concurrency on the physical layer is caused by explicit instructions from the application, or by caching and latency hiding in the operating system layers.

[1] We use *'FLOPs'* as the plural of *'1 FLOP'*, versus *'FLOPS'* as *'FLoating-point OPerations per Second'*

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

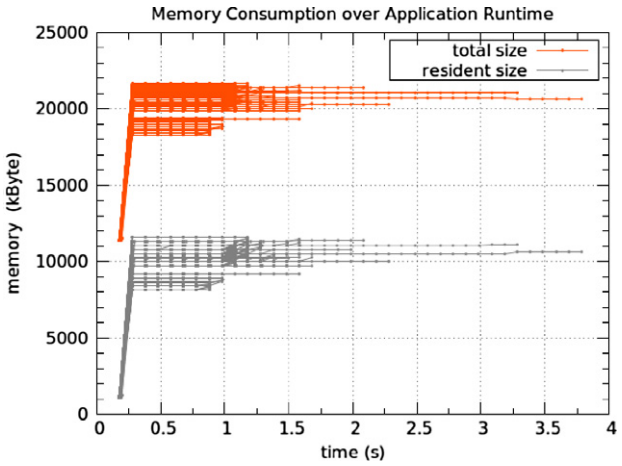

Fig. 1. Memory consumption profiles for Montage `mProjectPP` tasks: Two clusters of memory loads can be identified, where the cluster of smaller memory consumption generally correlates to smaller runtimes. For the majority of tasks (the larger cluster), the memory consumption is constant, and runtime is not influenced by other factors.

To quantify the hardware resources consumed by an application, SYNAPSE inspects the Linux kernel and hardware counters, via the `/proc/` filesystem and the Linux system tool `'perf'`. SYNAPSE provides a command line tool (`synapse-profile`) that runs the application in question under non-intrusive observation. During the run, SYNAPSE reads (with fixed frequency) different system hardware and kernel counters that report on the process' resource consumption. Those data are stored as a time series in a database, so that resource consumption is later available as a series of sampled data points.

Sampling the application's resource consumption at frequent intervals results in application profiles that describe the application behavior over its runtime. However, sampling is not appropriate for metrics that are very sensitive to system load (e.g., CPU utilization), or that are only meaningful as an integrated total (e.g., peak memory size, number of spawned threads). Fig. 1 displays the time variation of memory consumption over the set of Montage `mProjectPP` tasks, which shows a clustering into tasks that allocate two distinctly different amounts of memory, and the relation of that to the overall runtime (presumably due to less work being performed).

The consumed FLOPs are derived from the total number of consumed CPU cycles, average CPU utilization (what percentage of the CPU has been used) and average CPU efficiency (what percentage of the used cycles has been spent on computation, versus, for example, on cache misses and failed branch predictions.) Our underlying assumption is that the same application code will show similar utilization and efficiency on comparable architectures. Fig. 2 shows the strong correlation between runtime and consumption rates of CPU resources.

In summary, focusing on metrics that represent resource consumption instead of metrics that represent the *effects* of resource consumption (such as runtime) allows a more accurate representation of the application behavior on arbitrary target resources, and thus provides adequate input to the application emulation via skeletons. In Section 5.1, we will discuss the extent to which the profiles provided by SYNAPSE support the results of the manual profiling, which have been used in the skeleton emulation experiments.

5. 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 chose these three applications for a variety of reasons. First, they are applications we have familiarity with from previous research projects, either in the applications themselves or computer science problems in how to best run them on a variety of parallel and distributed systems. Second, because fifteen stages well represent the variety of dataflow patterns that can occur in MTC applications, including gather, scatter, reduce, pipeline, and multicast [48]. Finally, these applications and their stages are scalable and include a variety of computation/data movement ratios [45].

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 [5], 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, mostly using the methods in Section 4.1. Repeated runs of the applications show very little variance in runtime, and thus the skeleton parameters have no relevant variance either. The assumed normal distribution for the skeleton parameters turned out to not always reflect reality (see for example the skewed distribution shown in Fig. 2 [left]), but does reflect the correct range of measured application characteristics. 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, though there was no significant variation in the results.

5.1. Montage

The Montage application used 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 1 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 measure skeleton parameters as previously described (see Section 4.1). However, the input sizes for mImgtbl and mAdd exceed the maximum RAM disk size, so we cannot use our standard technique of running the tasks with data in RAM disk. 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 [3], 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.

Additionally, for mDiffFit, we previously found a performance gap between the skeleton and real mDiffFit of about 13% [3] 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 four times instead of one. So we set the Skeleton parameters to eight input files, repeating the two input file names four times.

Using the manually-generated parameters, we measure the error of all eight stages as less than 4%, with four under 1%, as shown in Table 2. The error for the complete application is -1.3%.

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

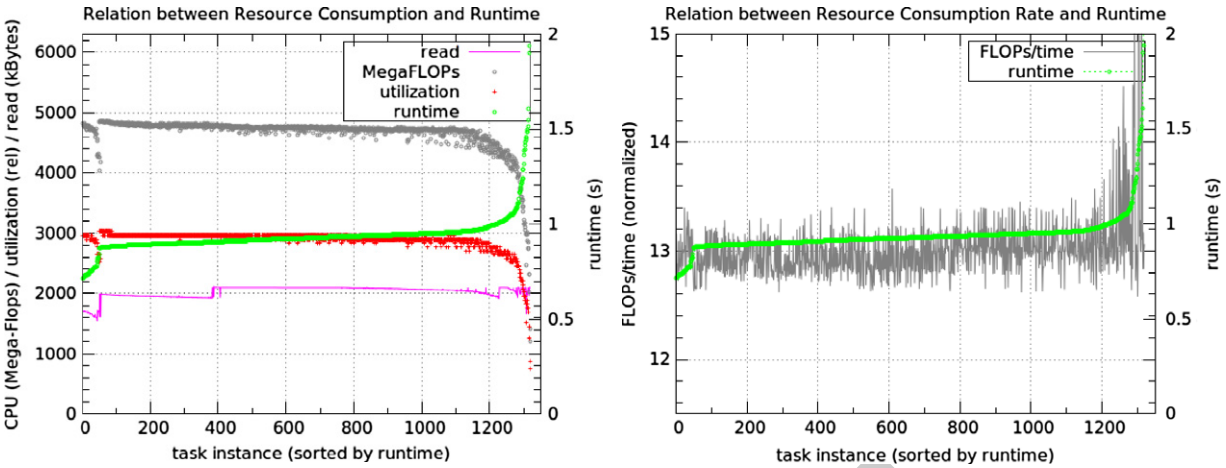

Fig. 2. Variation of task runtime depending on resource consumption: (left) a fraction of tasks with smaller reads consume less FLOPs and thus have shorter runtimes. Overall, the CPU utilization has a dominant influence on runtime, and negates the effects of lower FLOP consumption; (right) the resulting CPU consumption rate shows a strong correlation to observed application runtime, independent of I/O performance.

Table 1
Number of tasks, inputs, and outputs, and input and output size, for each Montage stage.

Stage	# Tasks	# Inputs	# Outputs	In (MB)	Out (MB)	Runtime Avg (s)	Runtime Stdev	Skeleton task length	
mProject	1319	1319	2594	2800.0	10400.0	11.6	2.5	uniform	11.6
mImgtbl	1	1297	1	5200.0	0.8	N/A	0.0	uniform	30.1
mOverlaps	1	1	1	0.8	0.4	9.1	0.0	uniform	9.1
mDiffFit	3883	7766	7766	31000.0	487.0	1.8	0.6	uniform	1.8
mConcatFit	1	3883	1	1.1	4.3	2.1	0.0	uniform	2.1
mBgModel	1	2	1	4.5	0.1	288.0	0.0	uniform	288.0
mBackground	1297	1297	1297	5200.0	5200.0	0.4	0.1	uniform	0.4
mAdd	1	1297	2	5200.0	7400.0	N/A	0.0	uniform	519

Table 2
Time-to-solution comparison of skeleton montage and real montage (s).

	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
Stdev	9.7	1.8	0.1	8.4	3.4	0.1	2.8	5.9	–
Skeleton	281.8	136.8	10.0	412.5	59.2	288.1	106.2	781.8	2076.4
Stdev	7.7	4.1	0.1	7.8	2.3	0.1	9.9	6.0	–
Error	−0.2%	−2.1%	−0.2%	−3.3%	−1.5%	0.1%	−1.6%	−0.9%	−1.3%

We also compare the measured skeleton parameters as listed in Table 1 to the values derived from the system independent profiling, as discussed in Section 4.3. The disk I/O parameters are found in exact agreement with the manual profiling—the size of the input data files is fixed after all. The SYNAPSE profiling though did automatically pick up the repeated file reads of the mDiffFit tasks. For the task runtimes, we calculated the expected runtimes t_{task} as follows:

$$t_{task} = f_{task}/e_{task}/u_{task}/p^{scaled}_{resource}$$

where f_{task} is the number of FLOPs the task consumes, e_{task} is the code efficiency, u_{task} is the CPU utilization, and p^{scaled}_{peak} is a scalability factor which is derived from the target CPU's peak FLOPS performance, scaled by the actually observed performance in performance benchmarks. The latter value, for the specific BG/P used for the experiments, is set to $\frac{191GFlops/sec}{557GFlops/sec} = 0.34$, based on a 2008 HPC challenge entry for that target machine.[2] The code efficiency and CPU utilization are measured on an architecturally very different x86 CPU.

Table 3 shows that while most runtimes predicted by system independent profiling have the correct order of magnitude, they deviate from the values observed via manual profiling by about 25%, either positively or negatively. This is not surprising, given the effort that is usually spent on optimizing the core computational libraries and application codes toward a specific target architecture, which makes the code efficiency and CPU utilization not easily portable from one architecture and operating system to another. The manual profiling on any target architecture thus gives, naturally, more precise runtime values for that architecture, but the automated profiling via system independent variables provides reasonable estimates for resources consumptions on resources for which manual profiling is difficult or impossible to perform.

Table 3
Runtime observation versus runtime prediction.

Stage	Measurement	Prediction	Deviation
mProject	11.6	8.54	1.35
mImgtbl	N/A	27.41	N/A
mOverlaps	9.1	7.36	1.23
mDiffFit	1.8	N/A	N/A
mConcatFit	2.1	N/A	N/A
mBgModel	288.0	386.69	0.74
mBackground	0.4	0.62	0.64
mAdd	N/A	1477.54	N/A

[2] http://www.hpcchallenge.org/custom/index.html?lid=103&slid=229

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

Table 4
Number of tasks, inputs, and outputs, and input and output size, for each BLAST stage.

Stage	# Tasks	# Inputs	# Outputs	In (MB)	Out (MB)	Runtime Avg (s)	Runtime Stdev	Skeleton task length
split	1	1	64	3800	3800	0.0	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]

Table 5
Time-to-solution comparison of skeleton BLAST and real BLAST (s).

	split	formatdb	blastp	merge	Total
BLAST	74.4	82.1	1996.3	35.9	2188.7
Stdev	0.6	1.7	6.0	1.1	–
Skeleton	72.9	81.6	2028.9	36.3	2219.7
Stdev	2.7	0.3	1.2	0.5	–
Error	−1.9%	−0.6%	1.6%	1.1%	1.4%

5.2. BLAST

BLAST is a widely used sequence alignment tool. The version we use here is parallelBLAST [13], 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 4 shows the basic statistics of each BLAST stage.

In general, we measure the task parameters as described in Section 4.1. The split stage has a single task does no significant computation, so we simply set the split task's length to zero.

Also, 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 results in the same amount of metadata operations and I/O traffic.

Finally, 64 concurrent formatdb tasks each with ∼500,000 small writes can run for hours on GPFS. So when we compare the Skeleton and real formatdb, we run the both with data caching in RAM disk. For blastp and merge stage, we run the tasks with data in GPFS.

Table 5 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 overall application error is 1.4%.

5.3. CyberShake

CyberShake [14,49] 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 6 shows the basic statistics of each CyberShake Postprocessing stage.

We generally use the previously discussed methods to determine skeleton parameters. 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.

In addition, 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.

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. 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 7 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%.

6. Using application skeletons

Here, we show examples of using application skeletons (with manually generated parameters) to study four system improvements: data caching, task scheduling, I/O tuning, and resilience. Various system optimizations were implemented with the AMFORA [5] system. The experiments were carried out on the Google Compute Engine (GCE) environment. Throughout all these experiments, we used "n1-highmem-2" instances, which have two vCPU cores and 13 GB RAM.

6.1. 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.

6.2. 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

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

Table 6
Number of tasks, inputs, and outputs, and input and output size, for each CyberShake Postprocessing stage.

S Stage	# Tasks	# Inputs	# Outputs	In (MB)	Out (MB)	Runtime Avg (s)	Runtime Stdev	Skeleton task length
Extract	128	130	256	5400	11000.0	6.4	2.2	uniform 6.39
Seis	4096	4352	4096	11000	96.0	26.9	13.3	normal [26.9, 13.3]
PeakGM	4096	4096	4096	96	1.4	0.2	0.1	uniform 0.23

Table 7
Time-to-solution comparison of skeleton CyberShake and real CyberShake (s).

	Extract	Seis	PeakGM	Total
CyberShake	571.5	2386.5	81.5	3039.4
Stdev	15.9	40.8	1.5	–
Skeleton	586.3	2443.3	83.3	3112.9
Stdev	1.5	4.9	2.7	–
Error	2.6%	2.4%	2.3%	2.4%

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.

6.3. I/O tuning

For applications that have highly concurrent and frequent metadata operations, using multiple metadata servers can improve the overall application performance [50,51]. 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.

6.4. 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.

7. 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 repeated and easily shared, making middleware and tool experiments more reproducible. This leads in two directions: computer science education and computer science research.

In computer science education, skeleton applications can be used by students to allow them to understand important properties of parallel and distributed applications. Simple skeletons can be used as instantiations of application abstractions, and more complex skeletons can be used in place of real applications.

In computer science research, including but not limited to the educational process, skeleton application and changes to their 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.

We also investigated the tradeoffs between manually-measured, system dependent skeleton parameters and automated, system independent skeleton parameters. Manual, system dependent parameters are more precise than system independent metrics, but

are also harder to obtain. The system independent parameters, however, represent reasonable estimates for resource consumption on systems where manual profiling is difficult or impossible to perform.

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 (with manually-measured skeleton parameters) 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 standard deviations for all experiments is consistently well below 10%, (mostly below 5%), and is also consistent between application and skeleton measurements.

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 [52] 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.
- Investigate how well Skeletons represent applications when configured with system independent parameters, as derived by SYNAPSE.
- 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.
- Allow more varieties of I/O behavior for parallel tasks than just process 0 performing all I/O.

Acknowledgments

This work is part of the AIMES project, supported by the U.S. Department of Energy under the ASCR awards DE-FG02-12ER26115, DE-SC0008617, and DE-SC0008651. It has benefited from discussions with Matteo Turilli, Jon Weissman, Michael Wilde, Justin Wozniak, Lavanya Ramakrishnan, and Simon Caton. 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. Jha, M. Cole, D.S. Katz, M. Parashar, O. Rana, J. Weissman, Distributed computing practice for large-scale science and engineering applications, Concurrency Comput. Pract. Exp. 25 (11) (2013) 1559–1585. http://dx.doi.org/10.1002/cpe.2897.

[2] S. Jha, D.S. Katz, M. Parashar, O. Rana, J.B. Weissman, Critical perspectives on large-scale distributed applications and production grids (Best paper award), in: The 10th IEEE/ACM Conference on Grid Computing 2009, 2009, pp. 1–8. http://dx.doi.org/10.1109/GRID.2009.5353064.

[3] Z. Zhang, 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.

[4] Z. Zhang, D.S. Katz, Using application skeletons to improve eScience infrastructure, in: 10th IEEE International Conference on eScience, 2014, http://dx.doi.org/10.1109/eScience.2014.9.

[5] Z. Zhang, D.S. Katz, T.G. Armstrong, J.M. Wozniak, I. Foster, Parallelizing the execution of sequential scripts, in: Proc. of the International Conf. on High Performance Computing, Networking, Storage and Analysis, SC13, 2013, http://dx.doi.org/10.1145/2503210.2503222.

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

[7] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, I. Raicu, Parallel scripting for applications at the petascale and beyond, Computer 42 (2009) 50–60. http://dx.doi.org/10.1109/MC.2009.365.

[8] M. Wilde, M. Hategan, J.M. Wozniak, B. Clifford, D.S. Katz, I. Foster, Swift: A language for distributed parallel scripting, Par. Comp. (2011) 633–652. http://dx.doi.org/10.1016/j.parco.2011.05.005.

[9] 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, D.S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Sci. Program. J. 13 (3) (2005) 219–237.

[10] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, M. Livny, Pegasus: Mapping scientific workflows onto the grid, in: M.D. Dikaiakos (Ed.), in: Lect. Notes in Comp. Sci., vol. 3165, Springer, 2004, pp. 131–140.

[11] 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, R. Williams, Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking, Int. J. Comput. Sci. Eng. 4 (2) (2009) 73–87. http://dx.doi.org/10.1504/IJCSE.2009.026999.

[12] D.S. Katz, J.C. Jacob, G.B. Berriman, J. Good, A.C. Laity, E. Deelman, C. Kesselman, G. Singh, A comparison of two methods for building astronomical image mosaics on a grid, in: Proc. 2005 Intl. Conf. on Par. Proc. Works, 2005, pp. 85–94. http://dx.doi.org/10.1109/ICPPW.2005.6.

[13] D.R. Mathog, Parallel BLAST on split databases, Bioinformatics 19 (14) (2003) 1865–1866. http://dx.doi.org/10.1093/bioinformatics/btg250.

[14] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan, E. Field, SCEC CyberShake workflows—automating probabilistic seismic hazard analysis calculations, in: I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, 2007, pp. 143–163. http://dx.doi.org/10.1007/978-1-84628-757-2.

[15] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, The NAS parallel benchmarks 2.0, Tech. Rep. NAS-95-020, NASA Advanced Supercomputing (NAS) Division (Dec 1995). https://www.nas.nasa.gov/assets/pdf/techreports/1995/nas-95-020.pdf.

[16] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[17] K. Krommydas, W. chun Feng, M. Owaida, C.D. Antonopoulos, N. Bellas, On the characterization of OpenCL dwarfs on fixed and reconfigurable platforms, in: IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, 2014, pp. 153–160. http://dx.doi.org/10.1109/ASAP.2014.6868650.

[18] Department of Energy Oak Ridge, Argonne, and Livermore Laboratories, CORAL collaboration benchmark codes. https://asc.llnl.gov/CORAL-benchmarks/.

[19] D.J. Kerbyson, K.J. Barker, D.S. Gallo, D. Chen, J.R. Brunheroto, K. Ryu, G.L. Chiu, A. Hoisie, Tracking the performance evolution of blue gene systems, in: Proc. IEEE/ACM Supercomputing 2012, 2013, pp. 317–329.

[20] P. Worley, I. Foster, Parallel spectral transform shallow water model: A testbed for parallel spectral transform algorithms, in: Scalable High Performance Computing Conference, 1994, pp. 207–214.

[21] I. Foster, P. Worley, Parallel algorithms for the spectral transform method, SIAM J. Sci. Stat. Comput. 18 (3) (1997) 806–837.

[22] V.E. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, I.R. Judson, Prophesy: an infrastructure for analyzing and modeling the performance of parallel and distributed applications, in: The Ninth International Symposium on High-Performance Distributed Computing, 2000, pp. 302–303. http://dx.doi.org/10.1109/HPDC.2000.868668.

[23] The Mantevo project. https://mantevo.org.

[24] J. Borrill, J. Carter, L. Oliker, D. Skinner, R. Biswas, Integrated performance monitoring of a cosmology application on leading HEC platforms, in: International Conference on Parallel Processing, ICPP 2005, 2005, pp. 119–128. http://dx.doi.org/10.1109/ICPP.2005.47.

[25] The characterization of the DOE mini-apps. http://portal.nersc.gov/project/CAL/overview.htm.

[26] S. Sodhi, J. Subhlok, Q. Xu, Performance prediction with skeletons, Cluster Comput. 11 (2) (2008) 151–165. http://dx.doi.org/10.1007/s10586-007-0039-2.

[27] J. Logan, S. Klasky, H. Abbasi, Q. Liu, G. Ostrouchov, M. Parashar, N. Podhorszki, Y. Tian, M. Wolf, Understanding I/O performance using I/O skeletal applications, in: C. Kaklamanis, T. Papatheodorou, P. Spirakis (Eds.), Euro-Par 2012 Parallel Processing, in: Lecture Notes in Computer Science, vol. 7484, Springer, 2012, pp. 77–88. http://dx.doi.org/10.1007/978-3-642-32820-6_10.

Please cite this article in press as: D.S. Katz, et al., Application skeletons: Construction and use in eScience, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.10.001

[28] J.F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, 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, ACM, 2008, pp. 15–24. http://dx.doi.org/10.1145/1383529.1383533.
[29] R. Thakur, Parallel I/O benchmarks, applications, traces. http://www.mcs.anl.gov/~thakur/pio-benchmarks.html.
[30] L. Ramakrishnan, S. Poon, V. Hendrix, D. Gunter, G. Pastorello, D. Agarwal, Experiences with user-centered design for the Tigres workflow API, in: 10th IEEE International Conference on e-Science, e-Science 2014, vol. 1, 2014, pp. 290–297. http://dx.doi.org/10.1109/eScience.2014.56.
[31] D. Karpenko, R. Vitenberg, A.L. Read, Atlas grid workload on ndgf resources: Analysis, modeling, and workload generation, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 79:1–79:11. http://dl.acm.org/citation.cfm?id=2388996.2389104.
[32] L. Meyer, M. Mattoso, M. Wilde, I. Foster, WGL—a workflow generator language and utility. http://dx.doi.org/10.6084/m9.figshare.793815.
[33] Y. Chen, S. Alspaugh, R. Katz, Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads, in: Proceedings of the VLDB Endowment, vol. 5, 2012, pp. 1802–1813.
[34] T. Harter, D. Borthakur, S. Dong, A.S. Aiyer, L. Tang, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Analysis of hdfs under hbase: a facebook messages case study, in: FAST, vol. 14, 2014, p. 12.
[35] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, Making sense of performance in data analytics frameworks, in: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, USENIX Association, Oakland, CA, 2015, pp. 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.
[36] S. Holl, D. Garijo, K. Belhajjame, O. Zimmermann, R. De Giovanni, M. Obst, C. Goble, On specifying and sharing scientific workflow optimization results using research objects, in: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science, WORKS '13, ACM, New York, NY, USA, 2013, pp. 28–37. http://dx.doi.org/10.1145/2534248.2534251.
[37] I.T. Foster, R.L. Stevens, Parallel programming with algorithmic motifs, in: ICPP, 1990, pp. 26–34.
[38] F. Meyer, D. Paarmann, M. D'Souza, R. Olson, E.M. Glass, M. Kubal, T. Paczian, A. Rodriguez, R. Stevens, A. Wilke, J. Wilkening, R.A. Edwards, The metagenomics RAST server—a public resource for the automatic phylogenetic and functional analysis of metagenomes, BMC Bioinformatics 9 (1) (2008) 386. http://dx.doi.org/10.1186/1471-2105-9-386.
[39] D. Moustakas, P. Lang, S. Pegg, E. Pettersen, I. Kuntz, N. Brooijmans, R. Rizzo, Development and validation of a modular, extensible docking program: DOCK 5, J. Comput. -Aided Mol. Des. 20 (2006) 601–619. http://dx.doi.org/10.1007/s10822-006-9060-4.
[40] J. Ekanayake, S. Pallickara, G. Fox, MapReduce for data intensive scientific analyses, in: 4th IEEE International Conf. on eScience, 2008, pp. 277–284.
[41] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Comput. Netw. ISDN Syst. 30 (1–7) (1998) 107–117. http://dx.doi.org/10.1016/S0169-7552(98)00110-X.
[42] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proc. of the 2010 ACM SIGMOD International Conf. on Management of Data, 2010, pp. 135–146.
[43] H.W. Sorenson, Kalman Filtering: Theory and Application, vol. 38, IEEE Press, 1985.
[44] S. Klasky, M. Beck, V. Bhat, E. Feibush, B. Ludäscher, M. Parashar, A. Shoshani, D. Silver, M. Vouk, Data management on the fusion computational pipeline, J. Phys. Conf. Ser. 16 (2005) 510–520.
[45] Z. Zhang, D.S. Katz, M. Wilde, J. Wozniak, 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, http://dx.doi.org/10.1145/2503210.2503222.
[46] A. Merzky, S. Jha, Synapse: Bridging the gap towards predictable workload placementUnder review, http://arxiv.org/abs/1506.00272.
[47] A. Merzky, SYNAPSE v0.9 (Mar 2015). http://dx.doi.org/10.5281/zenodo.16024. http://github.com/radical-cybertools/radical.synapse/.
[48] Z. Zhang, D.S. Katz, J.M. Wozniak, A. Espinosa, I. Foster, Design and analysis of data management in scalable parallel scripting, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC12), IEEE Computer Society Press, 2012, p. 85.
[49] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, K. Vahi, CyberShake: A physics-based seismic hazard model for southern California, Pure Appl. Geophys. (2010) 1–15. http://dx.doi.org/10.1007/s00024-010-0161-6. Online Fir.
[50] S. Patil, 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, USENIX Association, 2011, pp. 13–13.
[51] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, T. Ludwig, Small-file access in parallel file systems, in: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE, 2009, pp. 1–11.
[52] D.S. Katz, A. Merzky, M. Turilli, M. Wilde, Z. Zhang, Application Skeleton v1.2 (Jan 2015). http://dx.doi.org/10.5281/zenodo.13750.

Daniel S. Katz is a Senior Fellow in the Computation Institute at the University of Chicago and Argonne National Laboratory and is currently a Program Director in the Division of Advanced Cyberinfrastructure at the National Science Foundation. He is also an adjunct faculty member at the Center for Computation & Technology (CCT), Louisiana State University LSU. Dan's interest is in the development and use of advanced cyberinfrastructure to solve challenging problems at multiple scales. His technical research interests are in applications, algorithms, fault tolerance, and programming in parallel and distributed computing. He is also interested in policy issues, including citation and credit mechanisms and practices associated with software and data, organization and community practices for collaboration, and career paths for computing researchers.

Andre Merzky is a Computer Scientist at Rutgers University. He studied Physics in Germany and Scotland. He has worked on topics in distributed computing concerning visualization, data management, and high level APIs, and is actively involved in the Open Grid Forum (OGF).

Zhao Zhang is a joint postdoc researcher in AMPLab and BIDS (Berkeley Institute for Data Science) at University of California, Berkeley. His research is to enable data intensive scientific applications on distributed and parallel systems. He is also interested in data management in scientific computing for reproducible research. Zhao receives his Ph.D. from the Department of Computer Science, University of Chicago in June 2014.

Shantenu Jha is an Associate Professor of Computer Engineering at Rutgers University. His research interests lie at the triple point of Cyberinfrastructure R&D, Applied Computing and Computational Science. Shantenu leads the RADICAL-Cybertools (http://radical-cybertools.github.com) project which are a suite of standards-driven and abstractions-based tools used to support large-scale science and engineering applications. Away from work, Jha tries middle-distance running and biking, tends to indulge in random musings as an economics-junky, and tries to use his copious amounts of free time with a conscience.