BottleMod: Modeling Data Flows and Tasks for Fast Bottleneck Analysis

Ansgar Lößer*, Joel Witzke†, Florian Schintke†, Björn Scheuermann*

*TU Darmstadt, Germany
†Zuse Institute Berlin, Germany

ansgar.loesser@kom.tu-darmstadt.de, witzke@zib.de, schintke@zib.de, scheuermann@kom.tu-darmstadt.de

Abstract—In the recent years, scientific workflows gained more and more popularity. In scientific workflows, tasks are typically treated as black boxes. Dealing with their complex interrelations to identify optimization potentials and bottlenecks is therefore inherently hard. The progress of a scientific workflow depends on several factors, including the available input data, the available computational power, and the I/O and network bandwidth. Here, we tackle the problem of predicting the workflow progress with very low overhead. To this end, we look at suitable formalizations for the key parameters and their interactions which are sufficiently flexible to describe the input data consumption, the computational effort and the output production of the workflow’s tasks. At the same time they allow for computationally simple and fast performance predictions, including a bottleneck analysis over the workflow runtime. A piecewise-defined bottleneck function is derived from the discrete intersections of the task models’ limiting functions. This allows to estimate potential performance gains from overcoming the bottlenecks and can be used as a basis for optimized resource allocation and workflow execution.

Index Terms—model, scientific workflows, data analysis workflows, bottleneck analysis

I. INTRODUCTION

Many scientific fields have a growing need for computationally intensive data analysis, from genome analysis to satellite image processing. Such complex processes are often referred to as scientific workflows. A scientific workflow consists of multiple processing steps, so-called tasks. Tasks can depend on the results of other tasks, creating a dependency graph between them.

Often, a single task is simply the execution of a program on one or more specific set(s) of input data, creating one or more outputs, which may then form input(s) of subsequent tasks. Such tasks can have very different temporal I/O behavior.

An intuitive example of this is video processing: Reversing a video file needs complete input before it can start computing the output. Other operations work on the data sequentially in order, like re-encoding a video: they can begin the computation while the input is still incomplete. Similar patterns exist on the output side: A video reversal task can only start when the input data is complete, but it can output continuously while the task is progressing; a task which, for instance, counts occurrences of specific patterns in a video and outputs the total number for each pattern will only be able to start outputting results after the entire video has been processed.

Other, almost arbitrary dependencies of input and output behavior over the course of execution of a task are easily conceivable, and a large variety of behaviors exists in virtually any field in which scientific workflows are applied.

Even if the behavior of individual tasks is usually reasonably simple, the interdependencies can quickly become complex. Depending on the data input and output behaviour of predecessor and successor tasks in the workflow, very different execution behaviour and very different resource utilization over time will result. How to understand and analyze these interdependencies given the structure of a specific workflow? How to model them appropriately? How to draw conclusions from them? These are the questions we are tackling here.

We propose a way to model a task’s I/O behavior over computation time in an abstract way. The tasks behavior and the available resources are modeled independently, to enable a separation of concerns. The model is described in Sect. II. In Sect. III we show how, based on such models, one can derive and predict the overall progress of the workflow, as well as the structure of the bottlenecks limiting the execution performance. The models are modular and allow the combination of task models to express chains of tasks or even complete scientific workflows. They are simple, yet flexible enough to describe a wide range of task behaviors. And they are constructed in a way that allows for quick and lightweight computations. This is a novel take on modeling the execution behavior of scientific workflows. In Sect. IV we show that this approach scales much more favorably than any existing one, and therefore also allows, e.g., for repeated evaluation during a workflow execution, adapting the predictions to live measurements. There are much more details to the functionality and implementation of BottleMod, which can be found in our paper preprint [1].

II. MODELING A PROCESS EXECUTION

This paper introduces BottleMod, a way of modeling the execution of a scientific workflow’s task as a so-called process in a generic mathematical fashion. A process can also be used to model other events. An example is a data transfer over the network, as demonstrated later in Sect. IV.
Fig. 1. Exemplary requirement functions. Part (a): common data requirement functions, where ‘stream’ is a data requirement function modeling a process that can progress with every new byte of the data dependency’s input i read (akin to stream processing). In contrast, ‘burst’ is a data input where all data must first be read entirely by the process before progress can be made. Part (b): common resource requirement functions. Here, ‘stream’ shows a resource needed continuously to let the process make progress p. ‘Burst’ shows a resource only required at the beginning, e.g., for a process that needs all its CPU time before it progresses at all.

Predicting the behavior of a task needs process-specific and execution-environment-specific knowledge. While a developer might know how a specific program works, the amount of resources available for the execution is typically unknown during development. On the other hand, the execution environment or the corresponding system administrator might have detailed knowledge about the available resources and how they will be allocated to the tasks. However, they do not know how the tasks work. BottleMod is designed to distinguish between the process-specific requirements and execution-specific resource allocation. This separation enables different parties to describe the requirements and resource allocation, resolving the issue. The information for the task model could either be statically annotated, measured, or learned from other executions of the same task. In the long run, BottleMod could be used for modeling, simulation, and analysis, providing valuable information for resource allocation and helping a dynamic scheduler with short- to midterm decisions. Also, the result visualization could be helpful in understanding how bottlenecks in an early stage can influence the execution later.

We model processes by their data and resource demands. Here, each process can have an arbitrary number of data and resource requirements.

1) Data is the input data directly available to the process. Input data can be stored for later usage. The model assumes that an unlimited amount of data can be stored, and that stored data does not expire.

2) Resources cannot be stored. Unused resources will not yield an advantage for the execution but are just gone. Therefore, resources help modeling the CPU time an application needs or the data rate of a link used for a data transfer.

These requirements are defined by requirement functions for each resource and input data per process. The functions describe how much of which input data (R_D) or resource (R_R) the process will need to progress until a certain point, measured by an abstract progress metric p. Accordingly, these functions are always monotonically increasing. Exemplary requirement functions are shown in Fig. 1. While requirement functions describe the requirements of a process, so-called input functions for a process define the amount of available data input (I_D) and available resource (I_R) at each point in time. Notice that contrary to the data input, the allocated amount of a resource can be lowered during execution.

These functions contain all the necessary information to derive a progress function P, describing the progress of the process over time—representing its execution. Output functions O map the process’s progress to the amount of data generated, enabling to calculate at which point in time how much usable output data is generated. The dimensions for all these functions can be chosen freely, given they are consistent for those interacting.

III. DEDUCING THE BEHAVIOR OF A PROCESS

Combining the process-specific requirement functions and execution-specific input functions allows for calculating the time a process needs to achieve progress. The amount of progress a process makes until a point in time t is represented by the progress function P(t). In this model, a process finishes when the maximum progress is reached. Analyzing a process’s execution corresponds to calculating the progress function of that process.

Calculating the progress function happens in two steps. First, the data limitations are combined to deduce the maximum progress possible regarding all process data inputs. Then, the effects of resource limits are taken into account to infer the process’s overall progress.

For every data input, a progress function P_{DK}(t) can be calculated with its requirement function R_{DK} and its respective data input function I_{DK}(t). This function defines the maximum progress limited by the specific data input k over time.

\[ P_{DK}(t) = R_{DK}(I_{DK}(t)), \quad \forall k \in \{1, \ldots, K\} \] (1)

The minimum of all those functions defines the maximum possible progress regarding all data inputs and imposes an upper limit on the overall progress.

\[ P_D(t) = \min(P_{D1}(t), \ldots, P_{DK}(t)) \] (2)

The overall progress P(t) is not only limited by the data progress but also by the resources.

Calculating the maximum possible progress considering resource demands is more complex than the deduction of data progress. Resources are not storable and therefore do not limit the maximum progress, but rather how fast more progress can be made at a certain point in time. Imposing the resource restrictions to get from P_D(t) to the actual progress function P(t), can be done with an iterative algorithm. The complexity of this algorithm depends on the amount and complexity of the used functions. Explaining the functionality...
and implementation of this algorithm is out of the scope of this document, but is described in more detail in our preprint [1].

Based on the final progress function $P(t)$ many features besides the execution time can be calculated. Using output functions $O_{Dm}$, the progress function implicitly indicates how much output data is generated. The result can be used as an input to another task. This enables chaining of multiple tasks and even modeling complete workflows. Also the limiting bottleneck at any point in time can be calculated, as well as the exact utilization of every resource. This can be a valuable tool in future work regarding scheduling and resource management.

IV. Evaluation

To evaluate the proposed model, we created a small workflow, shown in Fig. 2. Based on the nature of this workflow, different prioritizations of the shared resource (a network link) naturally result in different execution times for the whole workflow. The execution times of many different prioritizations were calculated by BottleMod and measured in experiments. The results are shown in Fig. 3 and show, that BottleMod works very accurate, given a model with enough precision.

We also found that the simulation time of BottleMod does not scale with the size of the input data. This is the case for other state of the art approaches, that use discrete event simulation (e.g. WRENCH [2]). The execution time of BottleMod only depends on the complexity and amount of the functions used to model the workflow.

V. Conclusion

We presented BottleMod, a comprehensive technique to model and analyze the behavior of (chains of) tasks mathematically. It can predict the runtime and resource usage of a process. Additionally, it can be used to find potential execution bottlenecks analytically and provide detailed information about them. This includes identifying the root cause and predicting the performance gain when the bottleneck is remedied.

The precision of BottleMod relies on the granularity of the provided mathematical functions. By differentiation of input and requirement functions, there is a separation of concerns between the execution and the process-specific details of the model. This can enable easier application of the model for task authors as well as execution environment administrators.

The evaluation (Sect. IV) showed that, given accurate information about the processes and the resource allocation, BottleMod already works very precise. In reality this can be a challenge since extracting accurate data for BottleMod can be difficult. This especially applies for tasks that are considered as black boxes. Their behavior can still be figured out and modeled through extensive logging during exemplary executions using, e.g., extended Berkeley packet filter [3] (based on the original Berkeley packet filter [4]).

The results of BottleMod contain the progress, bottlenecks and predicted resource usage of a process. All this information can be used by a resource manager to schedule resources proactively. The results could also enable more efficient placement of tasks, running as part of workflows in distributed environments, on nodes. In order to optimize resource allocation and mitigate the most disadvantageous bottlenecks, it is up to the resource manager to apply the insights found by the BottleMod analysis to the actual execution. The resource manager could change priorities or limit resources with methods such as Linux control groups [5] or SDN bandwidth reservations.

Many more details about the functionality, implementation and evaluation of BottleMod are described in our preprint [1]. Our current BottleMod implementation and the scripts used to execute our evaluation are publically available under: https://github.com/bottlemod/bottlemod.

REFERENCES

[1] A. Lößer, J. Witzke, F. Schintke, and B. Scheuermann, “Bottlemod: Modeling data flows and tasks for fast bottleneck analysis,” arXiv preprint arXiv:2209.05358, 2022.
[2] H. Casanova, R. F. da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, and F. Suter, “Developing accurate and scalable simulators of production workflow management systems with wrench,” Future Generation Computer Systems, vol. 112, pp. 162–175, 2020.
[3] eBPF Developers and S. Committee, “Extended berkeley packet filter (ebpf),” Jul. 2022. [Online]. Available: https://ebpf.io/
[4] S. McCanne and V. Jacobson, “The bsd packet filter: A new architecture for user-level packet capture,” in Proceedings of the USENIX Winter 1993 Conference, ser. USENIX’93, vol. 46. USA: USENIX Association, 1993.
[5] T. Heo, “Control group v2,” Oct. 2015. [Online]. Available: https://ebpf.io/