THE UNIVERSITY OF CHICAGO

INTEGRATING TASK PARALLELISM INTO THE PYTHON PROGRAMMING LANGUAGE

BY
TIMOTHY G. ARMSTRONG

PAPER SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CHICAGO, ILLINOIS
MAY 2011

ABSTRACT

One of the major current challenges in computer science is providing programming models and abstractions that allow efficient and correct parallel programs to be expressed. The challenge occurs in a number of domains, with different scales of paralellism required. In the consumer space, eight or ten cores will soon be common on desktop computers, and the performance of software which cannot take full advantage of many cores will lag. In the scientific space, scientific investigations increasingly require computationally intensive simulation and data crunching on computer clusters.

Coordination languages are one way in which the problem of writing parallel programs can be made more manageable. A coordination language provides a specialized way to specify the communication and synchronization between the different components of a parallel or distributed application. In many cases, the parallelism in a program is naturally expressed as task parallelism, where independent tasks which produce and consume data can run in parallel.

This paper proposes and demonstrates the feasibility of embedding a task-parallel coordination language, PyDFlow, into the general-purpose scripting language Python. PyDFlow currently supports loosely coupled workflows, where tasks communicate through input and output files, but the language is designed to be extensible beyond this.

PyDFlow represents parallel computations as graphs of tasks linked by data dependencies and provides a programming model somewhat akin to functional languages, where computations are specified by constructing expressions by composing functions. This gives natural syntax and semantics for expressing a parallel or distributed computation that integrates well with the Python language, and allows execution techniques from the functional language literature to be applied.

Table of Contents

List of Figures 5

List of Tables 7

1 Introduction 8

2 Coordinating Task-Parallel Computations 11
2.1 Coordination Languages 11
2.2 Task Parallelism 12
2.3 Task Dependencies 13
2.4 Grid Workflow Systems 14
2.5 Skeletons 15
2.6 Recent Trends 16
2.6.1 The rise of cloud computing 17
2.6.2 The Rise of Dynamic Languages 18
2.7 Scripting and Coordination 19
2.8 Domain Specific Languages 21
2.9 Formal Models of Coordination 22

3 Functional Languages as Coordination Languages 24
3.1 Strict and Non-Strict Evaluation Strategies in Functional Languages 24
3.2 Task Graphs and Thunks 25
3.3 Parallel Evaluation 26
3.3.1 The Eager Beaver Strategy for Strict Languages 28
3.3.2 The Eager Beaver Strategy for Lazy Languages 29
3.3.3 Similarities between Lazy and Strict Strategies 29
3.3.4 Reducing Overhead 30
3.4 Futures and Write-Once Variables 31

4 Motivation 34
4.1 Extending a Scripting Language with Coordination Constructs . . 34
4.2 Major Design Considerations 35

5 Basic Design of PyDFlow 39
5.1 Applying Functional Programming and Task Graphs 39
5.2 Primitives 40
5.3 Details of Abstractions 40
5.3.1 I-vars 40
5.3.2 Binding I-vars 42
5.4 System Architecture 43
5.5 Extensions 44
5.6 PyDFlow Functions 45
5.7 The Redirection Operator 48
5.8 Task Graph Representation 49
5.9 Laziness 49
5.10 Compound Functions 51
5.11 Type System 52
5.12 Similarity to Swift 54

6 Execution of Task Graph 58
6.1 Executing Synchronous and Asynchronous Tasks 58
6.2 Depth First Traversal of Task Graph 59
6.3 Parallel Evaluation 63
6.3.1 Limitation on Parallelism Imposed by Python 63
6.3.2 Parallel Graph Execution 64
6.3.3 Work-stealing 65
6.4 Promise Pipelining 66

7 Applications 70
7.1 Montage Astronomical Imaging Workflow 70
7.2 URL Fetcher: Integration with other Python Code 70

8 Extending PyDFlow with Python 76
8.1 Mappers 76
8.2 Generators 78

9 Conclusions and Future Work 82
9.1 Conclusion 82
9.2 Future Work 82

List of Figures

3.1 A comparison of the same code with both strict and lazy semantics. The whole expression shown in both cases is an example of a pure expression. 25
3.2 A Montage astronomical imaging workflow expressed in Haskell and as a task graph. The data dependency graph is obtained by expanding the Haskell data-dependency graph, treating the functions used (mArchiveInfo, mAdd, etc) as atomic operations. . . . 27
3.3 A comparison of futures and write-once variables in pseudocode. . 32

5.1 State transition diagram for PyDFlow I-vars. Not shown is the state transition that can occur from any state when the I-var is garbage collected. 41
5.2 PyDFlow Software Architecture illustrating the relationship between the core of the PyDFlow software and the components of an extension. 43
5.3 A PyDFlow program using concurrent Python tasks. *py_ivar* is an I-var type that can contain any Python value 44
5.4 A PyDFlow program that executes external applications. *localfile* is a specialization of the *file_ivar* I-var type that represents a file on the local file system. 45
5.5 This figure illustrates how a PyDFlow decorator wraps a Python function. 47
5.6 This figure illustrate a task graph being constructed as I-vars are created and PyDFlow functions are called to construct tasks and I-vars. 47
5.7 This diagram illustrates the evolution of the in-memory task graph for a trivial example. The propagation of data through the task graph and enabling of tasks can be seen. The interaction of task graph execution with the garbage collector is shown. In the CPython implementation, with reference counting garbage collection, tasks and I-vars will be collected immediately once eligible. 50
5.8 Example code performing a parallel merge sort of a number of files on disk using the Unix *sort* utility. The files must be in text format and consist of numbers separated by line breaks. The function *merge_sort* implements divide-and-conquer in order to merge sort the files, with the task graph forming a reduction tree with the unsorted files at its leaves, intermediate sorted files at its internal nodes, and the sorted file at its root. The code in the *merge_sort* function is invoked lazily when the graph executor explores a particular branch of the tree, unfolded it gradually. 53

5.9 MODIS example in Python that demonstrates the look and feel of the language. This script shows a variety of different ways that I-vars can be manipulated and stored using Python data structures. For example, the *recoloredImages* list is constructed by appending items one at a time, while the *land* list is constructed using a "mapper" function. 55
5.10 MODIS example in Swift 56

6.1 A high level view of task graph execution showing the flow of tasks, I-vars and messages between different components of PyDFlow. 59
6.2 Pseudocode showing how a new asynchronous task type and corresponding executor can be implemented for PyDFlow. 60
6.3 State Transition Diagram for PyDFlow tasks 61
6.4 A detailed look at the PyDFlow graph executor module, showing how data passes between the worker thread stack, the input and resume queues, and the asynchronous executors. 62
6.5 A simple example of promise pipelining 66

7.1 A scaled down astronomical image montage of the Pleaides star cluster. 71
7.2 Shared Montage definitions in PyDFlow 72
7.3 Montage script in PyDFlow that creates an image of the Pleaides star cluster. 73
7.4 PyDFlow code for a parallel URL fetcher. 74

8.1 Four different ways of using mappers to bind Swift variables to files in Swift: *array_mapper*, *structured_regexp_mapper* *simple_mapper* with array indexing and *simple_mapper* with named structure members. Some examples were adapted from the Swift [79] documentation. 77
8.2 Equivalents to Swift mappers in Python. Python's *map* primitive can be used to emulate Swift *array_mapper*. *SimpleMapper* is similar to Swift's *simple_mapper*, except it is dynamically typed, so can support both indexed and named members at the same time. *SubMapper* is similar to *structured_regexp_mapper*, as it transforms a list of inputs with a regular expression. 78
8.3 Python code to iterate through the cartesian product of two lists using a generator. 79
8.4 Python code using asynchronous callbacks to handle task completion. 80
8.5 Python code using a resultset to handle task completion without callbacks. 80

List of Tables

7.1 Performance of a Montage workflow creating a 3×3 degree image of the Pleaides start cluster, comparing serial execution time with Swift and PyDFlow on a four processor desktop PC. The job throttle was set to five concurrent jobs for PyDFlow, ten concurrent jobs for Swift, and 15 concurrent jobs for Swift+Coasters . These settings were obtained by testing a range of parameters from four to 20 for each system. The time measured does not include the time taken to download image tiles. 71

SECTION 1
Introduction

Parallel programming has traditionally been a rather advanced speciality, with the vast majority of working programmers confined to writing serial code, and parallel programming touched on at most briefly in undergraduate computer science curriculum.

Recent trends in hardware design and applications will require more and more programs to expose a significant degree of parallelism if they are to make use of more than a fraction of the capability of new hardware. Desktop processors with four cores are now common, it seems inevitable that processor with eight or more cores will be standard in desktop computers.

In the scientific space, improved data collection technologies have resulted in huge volumes of data requiring analysis, and scientists are increasingly making use of simulation and automated analysis. In many cases, scientists with only basic programming skills are in the position where they need to be able to exploit the parallelism of multi-core machines, computer clusters or grids in order to advance their research.

The parallel programming abstractions that have traditionally been in the widest use in production software systems - multi-threading, shared memory and message passing - are poorly suited to this new world, where a larger fraction of programmers will be under pressure to parallelize programs in order to improve performance. It is difficult to write correct programs with these abstractions: there are no safeguards against synchronization problems such as race conditions and deadlocks. It is easy to express synchronization bugs in a program that relies on shared memory or message passing as its coordination mechanism. Even worse, it is difficult to find them once they are in a codebase: testing and debugging techniques that are effective on serial code often are ineffective at finding synchronization bugs, particularly when they are triggered by race conditions. It is not unusual for latent synchronization bugs to remain undetected until a system is put under heavy load, or until a seemingly harmless change in the environment changes the ordering of events.

Programmers who have a deep understanding of synchronization issues, and

who scrupulously engineer parallel programs can navigate these pitfalls, at the cost of expending a large amount of time and mental effort. But the majority of working programmers have neither the expertise nor the time to do this.

Working in a distributed environment, with a computation run across many computers, adds an additional amount of complexity to the problem: aside from synchronization issues, now programs must have a degree of fault-tolerance built in, as it is typically problematic if failures of network links or individual hosts causes an entire program to crash.

The work described in this paper, which is realized in the PyDFlow library, is motivated by these challenges: it aims to make it significantly easier to express task parallelism. Programmers without experience with parallel programming but who understand their application well can often readily identify independent computations within their serial programs. This task parallelism represents low-hanging fruit that could easily be exploited by non-specialist programmers given a clean abstraction for expressing and managing the execution of parallel tasks. Much previous work has been done on providing coordination languages and abstrations for task parallelism. Section 2 surveys the existing literature on the topic.

The main thesis of this paper is that appropriate abstractions for task parallelism embedded in a flexible general-purpose language like Python can express a significant amount of task parallelism without being onerous to learn or to integrate with existing code. PyDFlow has been designed with a strong view toward supporting the integration of disparate systems. There are a number of different usage scenarios that PyDFlow is designed to accommodate. A common scenario is where an number of existing serial subprocedures or executables must be composed to run in parallel. Another is where an existing function is to be parallelized without any alteration to the external interface. A further scenario is where we want to specify a new kind of parallel task that perhaps runs on a particular device such as a GPU or uses a particular grid or cloud software stack to support execution of remote tasks. PyDFlow aims to be flexible enough so as not to require any radical changes to the library to support unanticipated uses.

Using a dynamically typed "scripting" language like Python is natural, as these languages and their software ecosystems are already well suited for quick compo-

sition and integration of software components. PyDFlow aims to leverage this existing strength, integrating multiple components to run in parallel. Adding support for task parallelism to a scripting language is likely to make it easy to incrementally parallelize a program, provided that the new additions work with existing language constructs.

Furthermore, PyDFlow is designed to allow coordination of applications comprised of different types of parallel task. The architecture of the library separates the logic for managing inter-task dependencies and execution from the logic for actually executing the task. PyDFlow supports tasks that are simply function invocations on a local machine, but the library design can coordinate execution of tasks on alternative devices such as GPUs, or on remote machines, with fault tolerance and dependency management provided by the PyDFlow library. Adding a new type of task to PyDFlow does not require any changes to the core library: it merely requires a small set of functions to be written for the new task type that allow tasks to be started and data to be managed.

SECTION 2
Coordinating Task-Parallel Computations

2.1 Coordination Languages

The concept of a coordination language was introduced to describe a methodology for developing distributed systems that attempts to separate the concerns of computation and coordination. The term coordination language was first introduced to describe the Linda tuple space system [37]. The Linda model provides a shared space containing tuples of data, along with a small set of primitive operations to read, write and query this tuple space. The Linda model could be implemented as libraries for a range of languages, permitting interoperation between different systems.

At the time there were many existing languages, such as C or $C++$ that were mainly suited to efficiently expressing computations running in a single shared memory space. These languages, however, did not provide primitives that were well-suited to expressing the coordination of distributed softare components. By providing a *coordination language* - a standard set of primitives for coordination between distributed processes, this weakness could be addressed. A system like Linda can realized as a library rather than as a programming language proper, meaning the coordination language can be orthogonal to the computation language, as it is possible for programs written using different coordination languages to use Linda libraries for coordination.

Coordination languages are an appealing way to reduce the difficulty of developing correct distributed/parallel systems. They have a number of strengths. One advantage comes from using a well-defined communication paradigm. Once a developer understands the coordination language, then they can more easily understand the coordination logic of a new application or software module. If a lower level communication mechanism such as TCP sockets is being used directly, then the number of different ways that communication can be implemented is much larger. Another advantage comes if the communication primitives provided by the coordination language can be easily reasoned about, for example if they are compositional [34] or deterministic [14].

2.2 Task Parallelism

Parallelism can exist in a program in a large number of forms and granularities.

The granularity of parallelism exploited can vary widely. At the lowest level, a modern computer processor will exploit instruction-level parallelism (ILP) [39] to overlap execution of different machine instructions such as arithmetic operations and memory accesses. The mechanisms involved in exploiting the different levels of parallelism are quite different. Typically it makes little sense for a programmer to try and exploit instruction-level parallelism , as it is entirely dependent on the processor architecture and requires an intimate understanding of the working of the particular processor. Instruction-level parallelism is usually captured by a combination of the processor, which can dynamically identify the dependencies between instructions, and the compiler, which can compile code in such a way as to expose maximum instruction-level parallelism through techniques such as loop unrolling. While impressive performance gains have been had by automatically exploiting more and more instruction-level parallelism, the move to multi-core chips and the need of many applications to scale up to multiple computers dictate that gains in performance will have to come from exploiting coarser grained parallelism [39], which cannot be captured easily by hardware techniques and which is extremely challenging to discover using program analysis techniques.

Two types of coarse-grained parallelism are data parallelism and task parallelism. In data parallelism, parallelism is obtained by applying the same operation to each data item in a set in parallel. Different granularities of data parallelism can exist. Vector machines, for example, can apply individual arithmetic operations in parallel, but other data-parallel systems will apply more complex and computationally intensive functions to the data items. Data parallel programming models typically support parallelism for regular data structures such as multi-dimensional arrays, but can be extended to support more complex nested data structures [9][12]. In either case the constraint remains that parallelism in the program must conform closely to the data structures used in the program.

Task-level parallelism, in contrast, does not impose the same kind of regularity as data-parallelism: heterogenous parallel tasks can be created and composed in irregular ways with communication and synchronization under programmer con-

trol. Different granularities of task-level parallelism also exist, from long-running batch jobs [50] to individual functions in a low level language like C [10].

This can make it significantly easier to express the parallelism available in many applications, which in many cases is naturally conceived of as heterogenous tasks with control dependencies. It can also allow improved performance by dividing work between tasks in ways that improve data locality or by allowing dynamic scheduling based on data availability [35].

2.3 Task Dependencies

There are various programming models that are used to express task parallelism.

The most basic method is effectively just to provide low-level primitives for creating new threads and for synchronizing between them. Communication between tasks can simply be through shared memory, relying on the programmer to implement correct and safe communication. Standard threading libraries provide the basic primitives such as thread creation and locks to allow this, and additional means of communication such as queues or concurrent data structures can augment this.

Another option is a synchronous *fork/join* model, where a program consists of sequential sections, which break into parallel sections by forking parallel tasks, for example for each iteration of a for loop. The parallel sections have a well-defined end point where all of the forked tasks synchronize and sequential execution resumes. This model is supported by systems in wide use such as OpenMP [19]. This model does limit the kinds of parallelism that can easily be expressed because of its inherently synchronous nature: in many cases it is difficult to exploit heterogenous parallelism with this model.

A more general model then is the task graph model, where control-flow dependencies between tasks are made explicit - a first class concept within the language. In this task graph model, the program is modelled as a directed acyclic graph (DAG) of tasks, with the control-flow dependencies between then as edges in the graph. A task can only be run once all of the tasks it depends on (i.e. that it has in-edges from) have completed. This graph is necessarily acyclic, as oth-

erwise tasks would be dependent on themselves[1]. The task graph model is more flexible than the the fork/join model, and less error-prone than code that directly uses threads, locks and shared memory for coordination.

A task graph can also be implied if tasks synchronize on shared variables or "futures" (§3.4). In Compositional C++, for example, tasks synchronize on shared write-once variables, effectively inducing a data-flow graph based on these dependencies that then determines the order in which tasks can be run.

In some cases the task dependencies in a program will exactly match the data dependenciee, in which case the task graph and the data-flow graph of the program are the same. Structuring a program in this way makes the parallelism quite easy to identify and reason about. If in addition we avoid side-effects and manipulation of shared state, the program can have desirable properties such as compositionality and determinism. In task-parallel languages with side-effects, such as Cilk [10] and Compositional C++ [13] do not impose any particular restrictions on what the parallel tasks may do and indeed it is possible for the tasks to communicate in arbitrary ways by modifying global state. However, by convention, programmers would avoid using shared state as much as possible.

Some task-parallel programming systems can provide stronger guarantees about tasks not interfering with each other. Safety can be enforced at the language level if the language is non-imperative (§3) or at the systems level if the task is executed on a separate machine or in a separate memory space, such as in the workflow systems described in the next section.

2.4 Grid Workflow Systems

A particular type of coordination language is a workflow system, which are often for scientific applications and in the context of grid computing [26][80]. These systems are designed to allow the composition of varied data sources, and tasks in order to generate scientific data and perform analyses. These tasks could executable programs run on remote resources, web services, data transfers, or any

[1]There are closely related forms of graphs which can be cyclic, for example if recursion or iteration is represented in the graph, but we restrict the discussion to the case where every vertex in the task graph is a task that executes exactly once

similar task. Workflow systems are typically designed to support large-scale computations, managing data dependencies between components of the workflow and ensuring that the final result is produced even in an unpredictable and sometimes unreliable distributed environment.

In order to provide a uniform abstraction over the components of a system, workflows typically view a task as a “black box” with a number of inputs and outputs of potentially different types. These inputs and output could be a mix of files, database entries and invocation parameters. A typical workflow has have a number of data sources, a set of desired output data items, and intermediate steps that represent the transformations required to derive the outputs from the inputs. In essence, these workflow systems are similar to a build tool like make [30] and indeed there are workflow systems such GridAnt [2] and gxp-make[66] that are extensions of the build tools ant and make respectively.

The dependencies between tasks are often conceptualized as a task graph of data dependencies. In some workflow systems, for example in DAGman [69] or Pegasus [25], the workflow is expressed at the outset simply as a explicit graph in a special XML file format. Some other workflow systems, such as Swift [81] or GridAnt [2] add iteration constructs, which means that the topology of the data DAG depends on values calculated during the execution of the workflow.

The representation of a computation as a task graph, aside from providing a convenient abstraction for the workflow, permits various optimizations and features to be implemented. Many workflow engines use the DAG representation in order to schedule and plan task execution, procuring computation resources and moving required data in advance. This requires that the system be able to infer inter-task dependencies ahead of time, which is supported directly if an explicit DAG representation is used, and can also be supported if dependencies can be discovered at runtime, or be statically inferred from the code.

2.5 Skeletons

Skeletons were proposed as a solution to the challenge of programming parallel systems, where the complexity of implementing parallel programs is reduced through the used of fixed patterns of parallel coordination [16]. Common ex-

amples of skeletons include “master-worker”, where a large number of tasks are run in parallel, with load-balancing between workers, “reduction”, where a large number of values are reduced down to a single value using a reduction function These patterns are common in popular parallel programming tools. MapReduce, for example, is a popular programming model for clusters [24] that uses a single communication pattern as a skeleton.

Many common problems can be solved using a relatively small number of communication patterns, so the burden of reimplementing them from scratch can be removed from the programmer. Typically also this approach permits efficient and optimized implementations with well-understood performance properties to be used [21]. Skeletons have been proposed as a coordination language paired with a sequential computation language such as FORTRAN [22]. A range of different skeleton based coordination languages have been proposed, each providing different sets of primitives and working in different contexts [20, 29, 47, 52, 75].

Skeletons can be viewed as special cases of the more general data flow graph approach, as most if not all skeletons are simply special cases of data flow graphs. The advantage of standardising the pattern is that it can permit specific optimisations for those particular graph topologies, or can simply make implementating the system easier. For example, some supercomputing systems provide hardware-level support for MPI reduction operations [1], and the fixed structure of a MapReduce job makes it easier to reason about and improve the performance of techniques such as replication [24].

2.6 Recent Trends

There are a number of trends occuring in the software that can inform design of coordination languages. For many programmers, factors such as speed of development, availability of libraries, fault tolerance and ease of scalability can trump factors like type safety and efficiency, at least when it comes to building real systems using existing components. As a result, it is definitely arguable that systems intended for use by a wide range of programmers should prioritize these aspects.

In particular, we look at the popularity of massively parallel programming models for cluster hardware like MapReduce or Dryad and also the success of

of dynamic programming languages for a wide variety of applications, such as web development is also a relevant trend, as anecdotally many programmers experience greatly improved productivity using dynamic languages and frameworks written in these languages. Looking at these solutions provides some insight into the needs and problems faced by users of distributed systems.

2.6.1 The rise of cloud computing

There are many examples of parallel programs that don't require intricate coordination to occur between processes running on separate machines. Rather, the coordination in the application can be expressed as files or data streams being produced and consumed by tasks.

MapReduce [24] is an important example, where a fixed pattern of communication and a system design that emphasizes robustness over efficiency and low latencies has proven sufficient to support a wide range of applications. The capabilities that MapReduce implementations such as Hadoop [78] provide have turned out to be rather useful to a large range of companies, who use it for the types of information-retrieval and data-processing applications it was originally intended, as well as more intricate tasks such as machine-learning algorithms [67] and scientific simulations [63].

These systems tend to eschew low latencies and support for unconstrained communication between cluster elements in favour of a more restricted model of computation. The success serves to illustrate that, for many problems, what is needed is a clear way to express parallelism, a model that scales well and a system that is fault-tolerant. Improving performance is of course always beneficial, and supporting more intricate coordination would broaden the range of potential applications, but these are not necessarily key to the usefulness of cloud computing systems. Hadoop, the most popular implementation of MapReduce, is a good example: it has attracted a large base of users despite having quite high performance overheads and making it difficult to express such fundamental operations as joins in an efficient way [65].

Furthermore, Pig Latin, a data-processing language implemented on top of Hadoop, allows specification of data-processing pipelines at a higher level, at the

high cost of a 50% increase in runtime, yet was still widely adopted within Yahoo, a major search engine [36]. This demonstrates that expressivity and reliability is more important than raw performance for many parallel applications.

The MapReduce model can be generalized to support more general communication patterns. There are a number of different systems that also provide a distributed file-system and the ability to run massively parallel jobs. Iterative MapReduce is a conservative extension to MapReduce that allows multiple iterations of the MapReduce pattern to be overlapped [27]. Dryad is a further generalization supports a range of different operators that can be used to construct more sophisticated communication patterns [41]. CIEL similarly supports arbitrary patterns of communication [57].

2.6.2 The Rise of Dynamic Languages

Over the last decades, dynamic programming languages such as Python [64], PHP[70], Perl [77], Ruby [31], TCL[60], Groovy [46] have arisen from various sources. These languages are characterized as dynamic, as opposed to static, because many things, particularly types and variable bindings, are implemented at runtime (ie. dynamically) rather than at compile time. This has various benefits: it is relatively straightforward to implement interpreters for these languages, and implementation of dynamic behaviors such as introspection, which are challenging in a static language, become relatively straightforward.

Many of these languages have managed to attract a strong community of users and developers and a large number of tools and libraries available. These languages are well-suited to rapid prototyping due to the range of libraries and the multiple programming paradigms embodied in these languages, and the conciseness of the syntax.

A classification of languages into *scripting* and *systems* languages has been proposed by Ousterhout [59], with the above-mentioned dynamic languages classified as scripting languages. Ousterhout argues that two distinct categories of languages have become used widely in practice by programmers. Systems languages such as C or Java provide low level access to the machine and can allow highly efficient code using complex data structures to be written, but tend to be

more verbose, inflexible and require more code to be written to achieve a task compared with scripting languages. Scripting languages are characterized as concise and flexible but due to their dynamic features, they typically are challenging or impossible to implement efficiently. Ousterhout argues that scripting languages allow much higher programmer productivity, and can be used to build software far quicker without sacrificing much performance by building high performance modules using systems languages, but then specifying the high level logic of an application by use scripting languages to glue together the modules of a system.

It is entirely possible that the dichotomy proposed by Ousterhout is a false one that can be bridged: it is not obvious that the desirable qualities for expressing "scripts" can only be provided by low-performance dynamic languages.

However, regardless of whether the dichotomy is fundamental it appears that the split between scripting languages and systems languages is entrenched, at least with the the programming languages currently in widespread use amongst working software developers. For example, Ruby on Rails [6] and Django for Python [33] are frameworks designed to enable rapid development of websites with content generated and stored in a backing database. These frameworks use features of dynamic languages and are widely popular amongst web developers, because the frameworks are designed so that common behavior for database-driven websites , such as filling a html template with values or selecting a number of records from a database can be expressed extremely concisely and in a way that is straightforward to change. The performance penalty for using these languages is not terrible because most of the heavy lifting is performed by web servers and databases written in more performant systems programming languages.

2.7 Scripting and Coordination

It is natural to ask, given the similarities between the roles played by "coordination" language and "scripting" languages, whether the concepts are closely related. After all, scripting languages are often used to orchestrate components written in other languages.

The virtues attributed to scripting languages such as conciseness and speed of development would seem also be be virtues in a coordination language, particu-

larly in scenarios where we want to be able to easily reconfigure software components that execute in a parallel fashion. One of the greatest challenges in developing parallel software is managing the complexity of communication and state. If the coordination logic of a parallel application could be condensed to a single readable script, then developing, debugging and understanding the communication logic would be greatly simplified, especially if the coordination language's primitives are easy to reason about.

Swift [79, 81] and Skywriting[56, 57] are two scripting languages designed expressly for the purpose of coordinating distributed applications.

Both languages are designed to coordinate computations on a cluster of machines. Swift in particular is quite flexible about the environments it supports: through the coasters mechanism it can run on high-performance supercomputers, on networks of workstations and on wide-area grids. In both systems, the coordinating scripting language runs on a single machine, but spawns off independent tasks running across many machines on the cluster. Communication between tasks in both instances occurs through files, which means that data can be passed between tasks without having to pass through the machine running the scripting language.

Both languages rely on a sharp division between the "coordination space" and the "data space" of the application. The scripting language runs in the coordination space, and all of the variables and data in the coordination space reside in the same shared memory space. All coordination decisions, such as which tasks to run, how tasks interact and when to terminate are made within the coordination space. The data space is the space in which tasks are run and the bulk data processed by the application must reside. Data can be moved between the two spaces, but typically moving large amounts of data from the data space to the control space would be avoided for performance reasons.

Centralizing control of the parallel computation within the coordination space makes it straightforward to reason about the behavior of the system, because the behavior is analagous to serials programs and does not require reasoning about the interactions of many distributed processes. Keeping large datasets and intensive computation outside of the coordination space means that the performance of the

application is not too dependent on the performance of the scripting language or bottlenecked by the limited resources that exist in the coordination space.

Both of these languages, however, assume a particular relationship between the script and the code being orchestrated: they presume that the script is responsible for controlling the parallel execution from the top-down, with the top-level coordination logic contained in a master script file. In neither case are there facilities provided to integrate the scripting language with other code in any other way. In contrast, one of the strengths of scripting languages is that they are more flexible than this: for example, TCL is designed so that the interpreter can be embedded within an application, and so that it can "glue" together components in arbritrary ways. In many cases it would be desirable to have the flexibility to add parallelism into existing software from the bottom up by replacing sequential modules with parallel versions. They also are rather special-purpose languages, without the more general purpose programming constructs and wide range of libraries that more mature scripting languages provide, and which is one of the main contributors to the rapid code development that is possible with them.

2.8 Domain Specific Languages

Domain-specific languages (DSLs) are special purpose languages that are constructed for the purpose of expressing programs within a specific domain. They trade off generality in order to allow a more concise, expressive or exotic syntax that best suits the problem being solved [53]. Typically they are used instead of more traditional libraries when the constructs that best allow the program to be expressed do not map naturally to the constructs in a general purpose programming language. DSLs can either be entirely new languages, with a customized syntax, parser and compiler or intepreter, or they can be embedded DSLs, where the program in the DSL is specified with the constructs of the host language (for example as an abstract syntax tree) and then interpreted within the host language. withing

Because of the way Swift and Skywriting are specifically targeted to supporting the top down coordination of components of particular types, they are perhaps better thought of as domain-specific languages for coordinating tasks rather than

fully-fledged scripting languages. It is possible to achieve similar goals with a domain specific language using a programming model distinctly different from a scripting language or a functional language.

One effort that aims to produce task-parallel domain specific languages (DSLs) is Delite [11]. Delite is different from Swift and Skywriting in that it is not designed to run in a distributed setting, rather focusing on managing multiple, potentially heterogenous compute resources on a single machine. Delite, rather than being a domain-specific language on its own, provides tools to construct customized domain-specific languages.

Delite is implemented in Scala, and provides abstractions that allow programmers to implement custom data types and operations on those data types that can be managed by the Delite runtime. Delite also allows construction of custom control-flow constructs. Delite operators when applied to Delite objects build a deferred task graph and return a future for the data that will be generated. Delite takes advantage of the pure functional subset of Scala, to allow static analysis of the DSL expressions

Another domain-specific language, Bloom, takes more inspiration from logic languages such as Datalog, where every data item is a tuple representing a logical predicate and program logic is expressed by inference rules from one predicate to another. Data is partitioned in a distributed environment by key, and communication is induced when logical inferences are made that produce a tuple that resides in a different machine's partition of the tuple-space. The Bloom language itself is entirely data-parallel rather than task parallel, but it can be used to facilitate task-parallel programming because arbitrary code can be triggered by events.

2.9 Formal Models of Coordination

Orc [45] is another coordination language designed specifically for writing programs that orchestrate distributed components. Orc presents an interesting contrast to systems like Swift and Skywriting because it attempts to build a coordination language on a minimal set of abstractions.

Orc models all local and remote functions with a uniform abstraction, the "site". A site is in many ways similar to a asynchronous remote function: a call

to a site can happen concurrently with other site calls, and can also be cancelled. Site calls which never terminate or halt can be used to model persistent processes.

Coordination in Orc is based on a minimal concurrency calculus, which allows sophisticated coordination patterns that utilize a distributed set of sites to be specified with a small set of combinators. Only four combinators: the parallel combinator, the sequential combinator, the pruning combinator and the otherwise combinator are sufficient to express a wide range of applications. A small functional programming language called Cor is also a part of Orc for convenience, but the concurrency calculus is in fact sufficiently expressive that Cor is able to be compiled to Orc.

SECTION 3

Functional Languages as Coordination Languages

The idea of using non-imperative languages to specify parallel computations has been a subject of research over several decades. The primary attraction is that these language have large "pure" subsets, to which powerful but relatively straightforward parallelization techniques can be applied without affecting the semantics of the language. In this section we discuss functional programming languages, as they are most relevant to the rest of the paper, but other forms of non-imperative languages, such as logic programming languages also have this property. There are a number of parallel logic programming languages [23], and at least one, Strand [34], has been proposed as a coordination language.

In functional programming languages such as Scheme, ML or Haskell, large subsets of the programming language are purely applicative: that is, formally they can be viewed as a single expression in the language, rather than as a sequence of statements to be executed in a particular order. These functional languages also tend to favour a programming style where evaluation of expressions has no side effects, such as input, output or modifications of program variables. Expressions that are both applicative and side effect free are commonly referred to as *pure*. It is feasible to apply parallel evaluation strategies to the pure subsets of these languages languages without any modification to semantics, because each subexpression will reduce to the same value regardless of whether it is evaluated before, after or in parallel with another subexpression.

3.1 Strict and Non-Strict Evaluation Strategies in Functional Languages

Given a program in a pure language, there are various strategies that can be used to evaluate the expression and obtain the final value.

Strategies vary in the order in which subexpressions are evaluated. A distinction is commonly made between *strict* and and *non-strict* (or *lazy*) strategies. Figure 3.1 demonstrates the radically different order in which expressions are evaluated. Note that in the lazy version, x is never evaluated.

```
1  let x = sqrt(123) # x evaluated
2      y = sqrt(456) # y evaluated
3      z = sqrt(4)   # z evaluated
4  in
5    let a = if z == 2:# z compared
6          then y    # a set to value of sqrt(456)
7          else x
8    in
9        # a now contains a value
10       ... rest of expression ...
```

(a) Strict evaluation

```
1  let x = sqrt(123)  # thunk created for x
2      y = sqrt(456)  # thunk created for y
3      z = sqrt(4)    # thunk created for z
4  in
5    let a = if z == 2  # z evaluated and compared
6          then y     # a and y same thunk
7          else x
8    in
9      # a/y is evaluated later on demand
10       ... rest of expression ...
```

(b) Lazy evaluation

Figure 3.1: A comparison of the same code with both strict and lazy semantics. The whole expression shown in both cases is an example of a pure expression.

If an evaluation strategy is strict then variables, including local variables and function arguments can only contain final values. This means that when an expression is assigned to a variable, for example by the statement: `x = fibonacci(20) - 1`, then first the expression `fibonacci(20) - 1` must be evaluated (by calling the fibonacci function and then subtracting one) and only then is the resulting value assigned to the variable `x`.

In contrast, in a lazy language, a variable can be bound to a unevaluated expression, so the above statement would immediately create an in-memory representation of the expression `fibonacci(20) - 1` and bind it to the variable `x` without evaluating it. The expression is then only evaluated if the value of the variable is needed. These suspended expressions are commonly referred to as *thunks*. Thunks can be manipulated in many ways, such as being passed as function arguments or stored in data structures without triggering evaluation.

3.2 Task Graphs and Thunks

Lazy functional languages are closely related to the task-graph model of parallel computation. While the programming model is typically not described in terms of graphs of tasks, the in-memory representation of an unevaluated expression forms a directed graph, which is typically acyclic[1].

This is because thunks will often contain references to other thunks: one thunk

[1]some lazy functional languages such as Haskell in fact can have cyclic data structures, but this feature is not particularly relevant to this discussion

may be a suspended application of an operation that takes a number of different thunks as arguments [44]. The recursive nature of thunks means that a recursive structure of thunks bound to a variable actually forms a graph. This is very similar to a task graph, in which the vertices of the graph represent activities that may or may not have executed: they represent suspended computations in the same way that thunks do.

One difference between the two models is granularity: in the task graph abstract, tasks are effectively opaque and can be of a large granularity. In contrast, thunks can represent even the application of a single primitive operation, such as an arithmetic operation or a conditional statement, or they can also represent a function invocation that will expand to a much more complex expression.

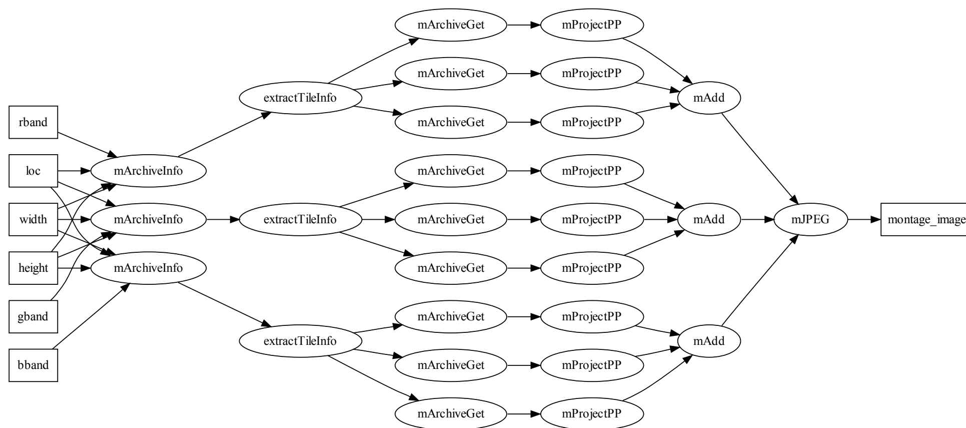
Functional language programmers do not typically think of expressions as a graph, but current implementation techniques for lazy functional languages such as Haskell have their roots in the graph reduction model of computation [44], where a program is represented as a graph and computation proceeds through incremental reduction steps until the graph is reduced to some canonical form, ie. a value. Examples of reduction steps are: evaluation of a primitive operator, expansion of a function application, replacement of a variable with its value.

To illustrate the commonalities between the two models, Figure 3.2 shows a simple astronomical imaging workflow is expressed as both a task graph and in Haskell syntax. The similarity is not total: if Figure 3.2 was a real Haskell program with the nodes representing Haskell functions, each node would be further expanded into primitive operators, rather than treated as an opaque task.

Given the similarity in the functional and task graph models of computation, semantics and the well-developed techiniques for expressing and executing functional programs, there are ideas that can be borrowed from the functional programming world to apply to build task-parallel systems in other languages.

3.3 Parallel Evaluation

Typically programming languages have been designed with sequential evaluation semantics in mind. This has some advantages: sequential semantics are typically easier to understand and reason about, and it avoids the non-determinism that




```
1  -- simpleColorMontage is a function with 6 input arguments
2  simpleColorMontage loc width height rband gband bband =
3      mColorJPEG rimg gimg bimg
4      -- the where clause contains local definitions for simpleColorMontage
5      where [rimg, gimg, bimg] = map bandImg [rband, gband, bband]
6
7            -- bandImg creates a greyscale image for this band of light
8            bandImg band = mAdd projected
9                  where raw = fetchTiles band
10                       projected = map mProjectPP raw
11
12           {- fetchTiles retrieves image tiles for band and part of sky
13              from an online archive -}
14           fetchTiles band = map mArchiveGet tileInfo
15                 where tilesFile = mArchiveInfo loc width height band
16                       -- parse the tile list file into a Haskell list
17                       tileInfo = extractTileInfo tilesFile
```

Figure 3.2: A Montage astronomical imaging workflow expressed in Haskell and as a task graph. The data dependency graph is obtained by expanding the Haskell data-dependency graph, treating the functions used (mArchiveInfo, mAdd, etc) as atomic operations.

could easily be introduced by parallel evaluation. For example, if different unrecoverable errors are caused by the evaluation of two different subexpressions, then a sequential order in which the subexpressions are to be evaluated ensures that the same error is raised whenever the program is run. Even if a language is defined in terms of its sequential behavior, however, it is possible to develop *correct* parallel evaluation strategies that preserve this.

In the rest of the section I discuss a simple parallel evaluation strategy, the "eager beaver" [7] strategy, and show how the strategy must be modified to work practically with strict and lazy languages. This strategy simply involves recursively starting parallel evaluation of pure subexpressions whenever they are encountered, creating parallel work whenever an opportunity for parallel evaluation is encountered.

3.3.1 The Eager Beaver Strategy for Strict Languages

If the language has entirely *strict* semantics, where function arguments are evaluated before the body of a function, then subexpressions can correctly be evaluated regardless of whether the final value is needed, except for special cases such as expressions that are the branches of a conditional expression.

However, strict semantics impose an ordering on the evaluation of expressions that is stricter than data dependencies dictate.

For example, the evaluation of any part of the body of a function cannot proceed until the function arguments have been fully evaluated, so opportunities for parallelism that would arise from evaluating arguments and the function body are lost. Inter-procedural optimization techniques can in some cases identify these opportunities. More typically, however, this problem can be solved by introducing a explicit *future* construct into the programming language, which is similar to a thunk in that it represents a value that is yet to be computed, but distinct in that the evaluation will occur in parallel in a different thread. Multilisp, for example, introduces an explicit *future* annotation that can be applied to a function invocation, which alters the function to return a handle that can be used to access the future value of the expression being computed [38]. Manticore, a parallel version of Standard ML, offers a similar *pval* construct [32]. Futures and related con-

structs have been used in many different languages, including both imperative and functional languages. We discuss futures further in Section 3.4.

3.3.2 The Eager Beaver Strategy for Lazy Languages

If the language is *lazy* then eager beaver evaluation is more problematic, as it can lead to expressions being unnecessarily evaluated, and more importantly, can lead to incorrect behavior.

The eager beaver strategy can lead to results differing from the sequential semantics because of a particular property of lazy languages. Under lazy semantics, the fact that an expression is valid and evaluates to a value does not guarantee that all subexpressions are also valid. This is because evaluation of the enclosing expression might not actually demand the value of the bad subexpression. If an eager evaluation strategy starts evaluation of the bad subexpression before it is needed, then sophisticated runtime strategies are required to avoid difficulties: any errors must be quarantined until they would have occured under sequential semantics, if they occurred at all.

Furthermore, the eager beaver evaluation strategy is problematic from a performance point of view, as a potentially unbounded amount of time and space can be expended on unnecessary evaluation of unneeded subexpressions.

We can modify the evaluation strategy to be more conservative, and correct, so that it only creates parallel work when it can be inferred that the result will be needed through static strictness analysis [44]. Unfortunately this is in general a particularly difficult problem in languages like Haskell, where laziness is pervasive down to the level of primitive operators such as list construction. As a result, parallel Haskell implementations [73] tends to rely on additions to the language that introduce an element of strictness, such as parallel annotations for expressions [72], or strict data structures[12].

3.3.3 Similarities between Lazy and Strict Strategies

These additions to strict and lazy functional languages above result in somewhat of a convergence in semantics: starting from a lazy language ended up resulting

in a language that was lazy with optional strictness, while starting with a strict language lead to a language that was strict with optional laziness.

3.3.4 Reducing Overhead

Another serious difficulty with parallel evaluation is that, in some cases, too much rather than too little parallelism will be identified. The number of parallel tasks generated can be vast for many applications, with the overhead required for managing them overwhelming any benefits from parallelism.

If subexpressions are to be calculated on different CPUs, then there are a number of steps that must be taken for each chunk of work to be shared, which can add up to at least several CPU clock cycles. A description of the work must be created, and stored in shared memory. This typically involves some sort of locking and bookkeeping. Then if the expression is to be evaluated on a different CPU, all of the required data must be moved from one CPU cache to another. Clearly if the subexpression was small: for example a simple arithmetic expression, then the costs of parallelism exceed any gains.

Techniques have been proposed and implemented that reduce this overhead. One approach is to avoiding splitting parallel work into tasks of too small a granularity. Cilk, a task-parallel version of C, tackles the problem by using a variety of tricks to minimise the overhead of parallel task creation and scheduling [10].

A more radical approach, was to tackle the problem at the hardware level with data flow computer architectures, where the burden of managing fine-grained data dependencies was handled by specialized hardware [76]. Despite a large amount of research on the topic, dataflow machines did not become practical alternatives to conventional computer architectures.

The problem of too fine grained tasks is typically solved in practice, for example, in parallel implementations of Haskell [73], by requiring the programmer to explicitly annotate which expressions should be evaluated in parallel. The laziness and strictness modifiers described above partially achieve this, as the programmer provides some information to guide what should be evaluated in parallel. Other language constructs such as parallel tuples [32] provide another way to annotate which parts of the program are worth evaluating in parallel.

3.4 Futures and Write-Once Variables

Futures [7] (also called promises[49], or a variety of other less common names) are a commonly utilized construct used when either implementing non-strictness in a strict language, or when coordinating asynchronous parallel computations. A future is in essence a proxy for a value that may or may not yet be computed.

There are a large number of different variations on futures. At a minimum a future has two states and supports two operations. The states are *filled*, when the value has been computed and is directly accessible through the future, and *unfilled*, when the value is in the process of being computed. The operations are *creation*, where the future is created and associated with a computation that will eventually yield a value, and *resolution*, where the current thread of computation synchronizes on the future and is suspended until the result is available.

Futures can be either *implicit* or *explicit*. Implicit futures, for example those in Multilisp [38], require some support at the language level and can be manipulated in the same way as any other variable: they are transparent proxies. Creation of the future can occur by, for example, adding a `future` keyword to a function call. Resolution of the future occurs when the value of the variable is needed. Explicit futures do not require integration at the language level: they are realized as a distinct data type that supports the required operations explicitly.

The basic two-state future is a *eager* future: as soon as the future is created, evaluation can commence. We can add an additional *unforced* to create a *lazy* future. The evaluation of a lazy future only starts when the value is *forced*, either synchronously when the future is resolved, or asynchronously if an additional operation to start asynchronous resolution of the future is supplied. Lazy futures give some additional control to the programmer, who can now decide when the computation should commence.

Write-once variables are a related construct that can be used to provide similar behavior in an imperative context. For example, Compositional C++ [13], a fully imperative language, provides a type of write-once variable called a *sync* variable. Id, a functional language with some imperative features, provides a version called the I-var [58]. Write-once variables differ in that the variable can be declared and have storage allocated prior to it being written.

```
// start calculation in parallel
int x = do_calculation()

if (cond) {
    // semantically invalid to assign to a
        future
    x = another_calculation() // invalid!
}

int y = x + 2 // synchronize on x
int z = f(x, y) // synchronize on x and y

// must define list at declaration time
int A[] = map(g, [1..100])
```

(a) Futures

```
// using like a future
int x
x = do_calculation() // start in parallel

// cannot do this with a future
int y
if (cond) {
    y = calcone()
}
else {
    y = calctwo()
}

z = f(x, y) // synchronize on x and y

if (othercond) {
    // runtime error if this branch taken
    y = calcthree()
}

int A[100]
A[0] = 0
for i = 1 to 99 {
    // write in strange order
    // difficult to express this with futures
    A[(7*i)%100] = calc(i, A[(7*(i-1))%100]);
}
```

(b) Write-once variables

Figure 3.3: A comparison of futures and write-once variables in pseudocode.

We illustrate the difference between these two constructs in Figure 3.3. Write-once variables can be used in the same way as futures by declaring them and then immediately assigning to them, but there are situations in an imperative language where it is useful to create a write-once variable and assign to it later. Like a future, attempts to read a write-once variable will block the reading thread until the data is available. This concept is generalized to arrays, for example I-structures in Id [4], by enforcing that each array element can only be written once This enables some imperative algorithms that rely on indexing into an array.

A disadvantage of write-once variables is that it becomes possible to express invalid programs with them: in an imperative program, it is possible to write twice to the variable, causing an error. A future differs in that, at creation time, it is already bound to contain the result of a particular computation, thus any attempts to express assignment to a future are semantically nonsensical. However, this behavior means that any program that uses only blocking reads and does not cause such a write-twice error cannot have any data races on write-once variables, elim-

inating a major cause of race conditions[2]. It is also possible to express deadlocks if the variable is never written to.

Write-once variables have proven to be a powerful construct in task-parallel coordination languages, as they allow imperative languages to enjoy some of the more powerful properties of functional languages. For example, in Compositional C++, if two components of a program are themselves deterministic, then when composed in parallel [13] using only write-once variables for communication, the property of determinism is preserved. Swift [79] also uses write-once variables to achieve similar determinism properties.

[2]If there are other sources of nondeterminism in the program, such as I/O or synchronization using nondeterministic mechanisms, then it is possible that the number of writes to a variable will be nondeterministic

SECTION 4
Motivation

The rest of this paper describes a Python library and programming model called PyDFlow that is designed to provide similar functionality to parallel scripting languages like Skywriting [57] and Swift [79]. PyDFlow currently supports a large subset of the features of Swift embedded within Python, and demonstrates the feasibility and value of embedding parallel data-flow coordination within a general purpose serial language.

In this section the specific considerations that played a role in its design are explained.

4.1 Extending a Scripting Language with Coordination Constructs

We chose Python as a host language because of its popularity among programmers and because it provides powerful features to extend the language such as introspection and decorators allowed us to achieve the desired semantics without introducing a large amount of boilerplate code into user code. At the same time, Python has disadvantages. For one, it is an interpreted language and as a result, significantly slower than many alternatives. However, as discussed in Section 2.6, for most intended uses, this is not a problem, as the advantages of the language for rapid development can be significant. We try to leverage this advantage by designing the library to allow programmers to more easily leverage parallel system, by readily parallelizing those parts of a serial program that could be split into tasks with data dependencies between them.

There are a number of libraries that allow task-parallel computation in Python. The built-in *multiprocessing* module provides a means to execute tasks in parallel on multiple CPUs, while IPython [61], PiCloud[28] and Celery [68] provide tools to execute Python functions on a pool of workers, some or all of which may be on remote machines. Celery and PiCloud support dependencies between tasks: Celery tasks can spawn subtasks, while PiCloud provides a form of deferred result equivalent to a future and also permits explicit job dependencies to be specified.

PaPy provides a form of data-flow parallel programming to Python [15] that uses a pipeline metaphor where components are explicitly connected together with data pipes, either by Python function calls or with a graphical user interface.

There are several important additional features that PyDFlow provides, that are important for its goal to be a general purpose task-parallel coordination language that fits naturally in Python:

- Standard mechanisms to support the definition of task types that are not simple Python functions and may execute on remote systems.
- A consistent abstraction for handling arbitrary types of remote data.
- Ability to construct and manipulate data structures containing futures.
- Ability to functionally compose parallel scripts.
- An implementation of futures that integrate naturally into the language so that tasks can be chained together with simple function invocations.
- Ability to handle large task graphs.
- Data type description and checking to catch type errors before execution.

4.2 Major Design Considerations

- **Integration with host language**: One of the pitfalls observed in past attempts at creating coordination languages is that, while designing a novel language and runtime from the ground up allows a lot of flexibility in language design and implementation, the burden it places on applications programmers of learning a new language, and the difficulties of integrating a new language with existing codebases, limit the applicability of the coordination language [34]. A key goal of PyDFlow is to express task parallelism without requiring unnecessary rewriting of existing code to fit into a new framework. Ideally a practical coordination language should allow task-parallelism to be added to a software module without exposing this implementation detail to clients of the module. Further, it should also allow the

integration of existing code in different languages or on different systems into parallel tasks.

This means that new constructs introduced should largely be orthogonal to the existing control flow constructs and it should be possible to manipulate the functions and futures provided by the library in a way that is natural to a programmer who already knows Python.

In particular, it should be possible to use Python control flow constructs such as for loops, list comprehensions and generators; task definitions should behave in the same way as standard Python functions; it should be possible to copy references to data objects in the same way as standard Python objects, and data items should be safely garbage collected as it goes out of scope.

- **Minimal syntax:** The library constructs should allow a parallel computation to be expressed with the minimal amount of extra code required to specify the parallelism and dataflow in the computation. The mantra DRY ("Don't Repeat Yourself") [40] is popular in the dynamic languages community and is a generally good principle of software construction. We want to minimize the amount of extra code required to enable parallel execution of tasks.

- **Extensibility**: It should be possible to extend the library with new types of tasks that permit execution of computations on a variety of different execution platforms, including: local multi-core CPUs, attached GPUs, remote nodes in clouds and clusters, and on local or distant supercomputer resources. This requires the ability to extend the library with new I-var types to represent data of different types, formats, and in different locations, which can be consumed and produced by tasks.

  Such extensibility not only has the practical benefit of allowing parallel scripting to be applied to new domains, but also serves to force the development of a clean model of task coordination and execution that is divorced from the peculiarities and implementation details of the particular system being coordinated.

- **Efficient support of remote tasks and data**: PyDFlow aims to coordinate the execution of tasks on remote machine, which will potentially be operating on large amounts of data.

  It is thus essential that as much intensive computation and I/O as possible remains local to the data and the execution of tasks. We need to provide abstractions that allow intuitive and powerful specification and manipulation of remote tasks and data, but which can implement as many possible operations as calls to procedures running on remote machines, near to the data.

  We want the bulk of work to occur in the data space, with only a small amount of computation occuring in the coordination space[1].

  For example, if a task running on a machine in a cluster requires data on another machine on the cluster, the only action that should be required in the coordination space is triggering a remote function that performs the work to set up the data transfer. Even better is if data dependencies can be resolved entirely in the data space: for example if the cluster machine is passed a reference to the data, and then is able to retrieve the data for itself.

- **Moderate overhead**: As much as possible, the overhead of the task graph execution per task should be constant and delay in starting execution of tasks should be minimal. Functional programming implementation techniques adhere to these constraints, so can provide some guidance.

- **Support for computations of unbounded size**:

  1) techniques for throttling unbounded paralalism (eg the Swift foreach.throttle property) and

  2) techniques that enable actual tasks to be lighter weight than an OS or language (ala Swift) A system that requires the entire task graph to be constructed in memory is limited in its applicability, as it cannot support computations that are either large scale, long running or have many small tasks.

[1] see Section 2.7 for a definitions of data and coordination spaces

It would be possible to build and run a task graph in pieces using Python's sequential constructs. But this is clumsy and asks too much of application programmers. It also has the disadvantage that it can easily result in the insertion of unnecessary sequential barriers and bottlenecks as an artefact of the programming model.

Functional languages do not have this problem, as it is possible for the tasks to create dependent tasks recursively. This suggests one way to avoid this problem: have primitives that allow for construction of the task graph on the fly, for example by recursively unfolding a tree of tasks implementing a parallel reduction. We also need garbage collection to allow cleanup of finished tasks and data, as explicit resource management is unwieldy.

- **Support for non-fixed task graphs**: There are many scenarios where the application cannot be naturally expressed as a fixed task graph. For example, it would be common to have parallel optimization procedures where the results of each phase must be inspected to check to see whether the procedure has converged and can be terminated.

  By working within Python, we already have this: we can use Python's sequential control flow constructs. For the optimization procedure, we could have a loop that built and ran the task graph for one iteration, checked the condition, and then built and ran the next iteration's task graph, using the prior results as output.

- **Feasibility of parallel execution:** The model of computation for the task graph should not have any fundamental constraints that prevent distributed execution of the task graph. In particular, if the semantics of the task graph were defined so that all of the control flow must be managed centrally, then we would be stuck with a master-worker system, which has fundamental limits on scalability [51].

SECTION 5
Basic Design of PyDFlow

This chapter discusses the basic design decisions made in implementing PyDFlow and provide a rationale for the basic architecture of the system. The current design and implementation is intended as a proof of concept to demonstrate that:

- The coordination primitives are both concise and expressive.
- Different types of task and I-var can be accommodated within the same framework

5.1 Applying Functional Programming and Task Graphs

As discussed in Section 3.2, there is a close similarity with this representation of the task graph and the way suspended computations are represented in lazy functional languages. By recognizing this similarity, we can take advantage of previous work on parallel and lazy functional languages to gain straightforward semantics and clean syntax for dealing with concurrent and asynchronous tasks, by introducing write-once variables and lazy evaluation into the Python language.

Since the target is to embed the language within Python, an imperative language, we unfortunately cannot build a side-effect free language from the ground up, which means that we rely on convention rather than hard guarantees to avoid side-effect-related mischief.

We want tasks in to be able to represent a variety of different asynchronous parallel computations. This is similar to workflow systems using the task graph model, where typically the interface for a task in in workflow is general enough that extensions can be written to support new task types. PyDFlow tries to remain as agnostic as possible about the actual nature of the tasks: they could be arbitrary computations occuring in arbitrary places operating on arbitrary data, being coordinated through appropriately-defined proxy objects for the task and input and output data.

5.2 Primitives

For PyDFlow we draw on ideas from both the task graph model and the lazy functional model. PyDFlow is a task-parallel coordination language in which the basic unit of execution is a *task*, and where data dependencies between asynchronous tasks are represented by write-once *I-vars* (§5.3.1). We use Python's *decorator* metaprogramming feature to implement a new type of *PyDFlow function* (§5.6) that enables varying task types to be expressed concisely in Python.

These tasks and I-vars, discussed in more detail in Section 5.3, can be linked together into a task graph, portions of which are executed on demand when an I-var is forced. As the task graph executes, tasks fill in heir output I-vars, and thus enable their dependent tasks to execute. I-vars can be explicitly created and bound to data (§5.3.2), or can be the deferred result of a PyDFlow task, in which case we call that I-var the output I-var of the task. An output I-var holds a reference to the task to be executed, which is started when the value of the I-var is demanded. Thus output I-vars behave like explicit, lazy futures.

We can use the idea of separate coordination and data spaces to explicate the abstraction further. The task graph exists in coordination space, with the PyDFlow tasks objects as proxies for computations, and I-vars as proxies for data in the data space. Thus, the task graph is merely a convenient and compact abstraction of a much larger computation that is being coordinated.

5.3 Details of Abstractions

5.3.1 I-vars

A PyDFlow I-var takes its name from the related variable type in the Id programming language [58], but is different in three ways. First, it is a proxy for data that is stored externally to the current process, rather than for an in-memory value or object. Secondly, it can have lazy semantics: if the PyDFlow I-var is the output of a task, then the task is started only on demand. Thirdly, the PyDFlow I-var is not integrated at the language level: explicit operations are used to read and write to an I-var.

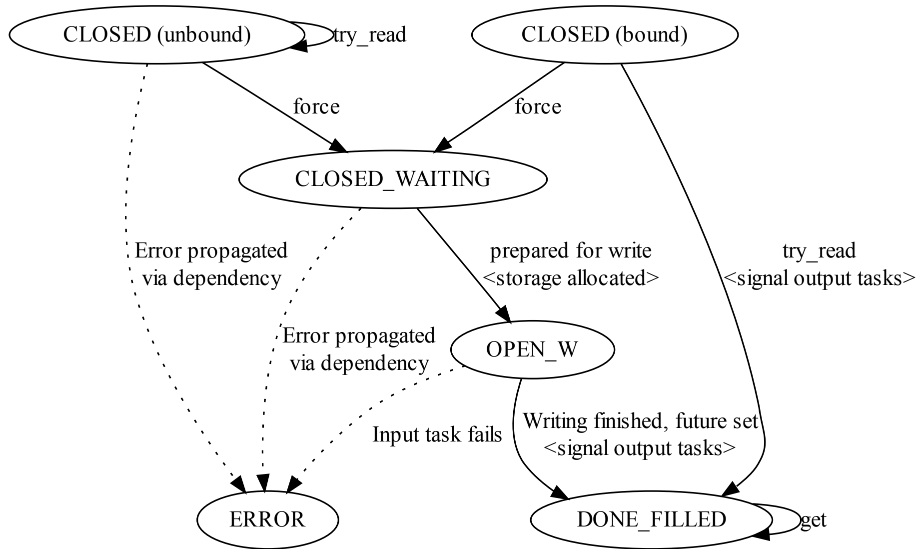


Figure 5.1: State transition diagram for PyDFlow I-vars. Not shown is the state transition that can occur from any state when the I-var is garbage collected.

All I-vars are implemented as a subclass of `Ivar`, which supports several standard operations. The << operator directs the output of a PyDFlow task to an I-var. Invoking the *get* method on the I-var blocks the calling thread until the I-var's data is available, starting the associated computation. An additional asynchronous *spark* method is also provide that starts the computation running but does not block.

We decided to give I-vars this lazy, rather than eager, behaviour because it gave more control to the programmer, by allowing the times of graph construction and execution to be distinct. The use of write-once variables was necessary to implement data flow semantics in Python, a sequential language. It allows the results of parallel computations to naturally be manipulated with Python's built-in control-flow constructs without requiring any complex synchronization logic to handle the asynchronous tasks.

After an I-var is created, the sequential Python program is free to pass the I-var as an argument to any Python function (but most likely to other PyDFlow functions); insert them into data structures (lists, hash tables, etc); or use them as input

for future computations. A more restrictive fork-join version of task parallelism would not give the same freedom in how the program can be structured.

Write-once variables like these I-vars seem to provide a natural bridge between the sequential and parallel world: data driven computations can be specified and then initiated by the sequential part of the program, with synchronization occuring when the sequential part of the program requires a value to proceed.

5.3.2 Binding I-vars

It is often useful for an I-var to be associated with a location in memory, in a file system, or in some other means of storage, particularly when I-vars are merely proxies for external data such as files. It is common that a programmer not only wants to have a programmatic handle to the future result of a computation, but they also want the result of the computation to end up in a particular place in memory or in a file system. It is also often necessary that an I-var should be associated with already existing data, whether it be a variable in memory or a file on disk.

Beyond the features of a traditional I-var, PyDFlow I-vars have a concept of "binding" an I-var in common with Swift [79]. If an I-var is bound to a Python value, for example a string in memory, then reading from the I-var will always return that value. If it is bound to a location, then the idea of binding is overloaded: the I-var might either be read or written. If it is read from before being written to, any preexisting data in the bound location will become the contents of the I-var. If an I-var is written to and not bound to a location - typically if the output of a PyDFlow task is directed to the I-var, then data is written to the bound location of the I-var.

A I-var can also be *unbound*. The implementation of an I-var in PyDFlow is responsible for allocation of temporary storage, such as a temporary file, for unbound I-vars. We take advantage of Python's garbage collection to clean up temporary data for unbound I-vars: when an I-var object is garbage collected, it is responsible for cleaning up any temporary resources it uses.

When an I-var is bound to a location or some data, the PyDFlow core system treats the contents of a location as being an opaque object. More sophisticated notions of binding can be supported, but this is left to the I-var implementation. This

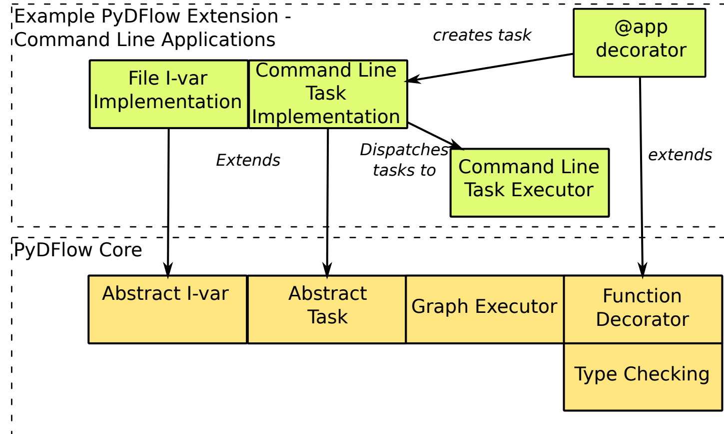


Figure 5.2: PyDFlow Software Architecture illustrating the relationship between the core of the PyDFlow software and the components of an extension.

is in contrast to Swift, where a more complex notion of binding, called *mapping*, exists. The data bound to a varaible might consist of multiple files, directories and records within a file that are be mapped to a logical data type in Swift comprising arrays and structures. There has been research on how to define such mapping in a general enough way to capture a large range of different storage schemes and data formats with specification languages like DFDL [62] and XDTM[55].

The state transitions for any PyDFlow I-var type are illustrated in Figure 5.1. Abstracting an I-var's behavior in this way allows PyDFlow to manipulate task graphs in an abstract way that is independent on the implementation of a particular I-var type.

5.4 System Architecture

The relationship between the different components of PyDFlow is illustrated in Figure 5.2. We view PyDFlow as a core set of modules which manage the creation and execution of a task graph, along with extension modules which support execution and management of different types of tasks and data. An extension is implemented by coding a number of classes and modules to standard interfaces, building on top of the core PyDFlow functionality.

5.5 Extensions

Two different extensions have been implemented in the current version of PyDFlow.

```
1  @func((py_ivar), (py_ivar))
2  def hello(name):
3      return 'hello ' + name + '!'
4
5  # we can create an unbound ivar
6  x = py_ivar()
7
8  # or we can bind the var to some data
9  y = py_ivar('there')
10 # this will print the string 'there'
11 print y.get()
12
13 x << hello(y)
14
15 # This will print the string 'hello there!'
16 print x.get()
```

Figure 5.3: A PyDFlow program using concurrent Python tasks. *py_ivar* is an I-var type that can contain any Python value

The first facilitates concurrent execution of Python functions. A new type of task is defined by applying the decorator `@func` to a Python function. The basic I-var type that is consumed and returned from these tasks is a `py_ivar`, containing a Python value. The computation performed by these tasks can be pure Python code, but could also invoke functions coded in other languages using a language interoperability tool such as SWIG [8] or Babel [48] or even use inline C code using a tool like Weave [43]. Tasks can also use any Python library, including high-performance libraries such as NumPy [43], which can enable performance much greater than that of interpreted Python code. A simple example is shown in Figure 5.3.

The second extension facilitates execution of external application programs that produce and consume files. These files are represented with the `file_ivar` I-var type, which is a proxy to data stored in a file system. A task is defined by applying the decorator `@app` to a Python function. The decorated Python function must return an `App` specification with the name of an application and command line arguments. This approach gives a great deal of flexibility in how the application command line can be constructed. The code within this function

```
1  @app((localfile), (localfile, str)) # str is the standard Python string type
2  def line_append(infile, suffix):
3      ''' Appends suffix to end of each line '''
4      regex = 's/$/' + suffix + '/'
5      return App('sed', regex, infile, stdout=outfiles[0])
6
7  # oldfile.txt should already exist
8  in_ivar = localfile('oldfile.txt')
9
10 #the output of a function is redirected to newfile.txt
11 out_ivar = localfile('newfile.txt')
12 out_ivar << line_append(in_ivar, "!")
13
14 out_ivar.get() # run my_function and wait until result ready
15
16 # file ivars provide an open() method to read from the file
17 # we print the numbered lines of the file
18 for i, line in enumerate(out_ivar.open().readlines()):
19     print i, line
```

Figure 5.4: A PyDFlow program that executes external applications. *localfile* is a specialization of the *file_ivar* I-var type that represents a file on the local file system.

has access to arguments that will contain the path of the input `file_ivar`, for arguments of that type, or can contain a simple Python variable, which allows Python values to be used in the command line. A simple example is shown in Figure 5.4.

5.6 PyDFlow Functions

The previous section showed that new types of PyDFlow tasks could be defined using decorated functions *hello* and *line_append*. We refer to these decorated functions as PyDFlow functions.

The first example of a PyDFlow function, *hello* is simply a lazy version of the plain Python function that was decorated: the `@func` decorator simply transforms a function into being a lazy PyDFlow version with a `py_ivar` as its output. We can assign the task's result to an I-var using the << operator. This also works if the PyDFlow function has multiple output I-vars, for example by writing `x, y << f(a, b)`.

PyDFlow functions also allow the output I-vars to be used in exactly the same way as futures (§3.4): if the << operator is not used to direct the output of the task

to a particular I-var, and instead the usual Python assignment operator = is used, then the return value of the function will be a new unbound output I-var for the future value of the function. For example, if we only wanted to allocate a temporary file for the output of `line_append`, then instead of the the code in Figure 5.4, we could instead write `out_ivar = line_append(in_ivar, "!")`. This behavior allows functions to be easily composed, with temporary storage for the unbound I-vars being allocated and freed as needed. For example, we can chain together invocations of the `hello` function from Figure 5.3 by simply writing `hello(hello(hello(py_ivar('there'))))`.

As well as making the function lazy, the `@app` decorator applies additional transformations to the original *line_append* function in order for it to launch the extern *sed* program. The illustrates the flexibility of this decorator mechanism to implement new task types. The original function did not actually start the command line application: it merely constructed an *App* object that describes the task to be executed. The first argument to *App* is the executable to be run, and subsequent arguments are the executable's arguments. The paths of the task's input files are directly available as function arguments, while the paths of the task's output files are represented with placeholders obtained by indexing the special *outfile* object. Input and output redirection can be specified using keyword arguments to *App*. The `@app` decorator can insert code to take this *App* specification and actually run the external application.

These decorators are a special feature of Python that can be applied to functions in order to modify the behavior of functions in quite powerful ways. A decorator can insert arbitrary code to run in place of the original function, although typically the decorator will simply insert code that runs before or after the original function, maybe performing some transformations on the function arguments or return values. Decorators, such as `@app`, are implemented as a Python class with a constructor, which takes some number of parameters, and a single method, which takes a single function as an argument and returns a modified version of the function. This process, as it is used in PyDFlow, is illustrated in Figure 5.5.

The PyDFlow decorators insert code in the transformed function to:

- check the types of the input arguments against the input types specified in

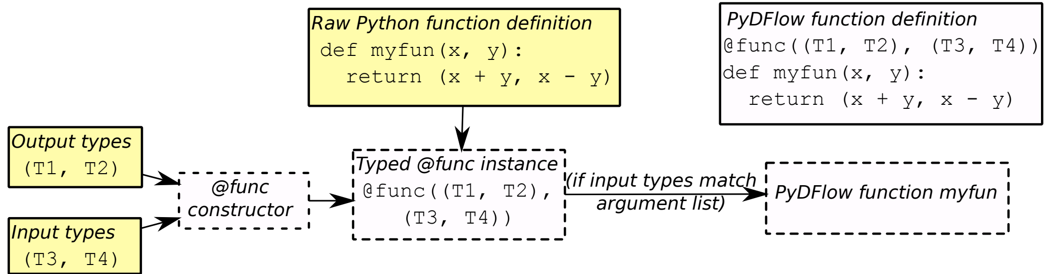


Figure 5.5: This figure illustrates how a PyDFlow decorator wraps a Python function.

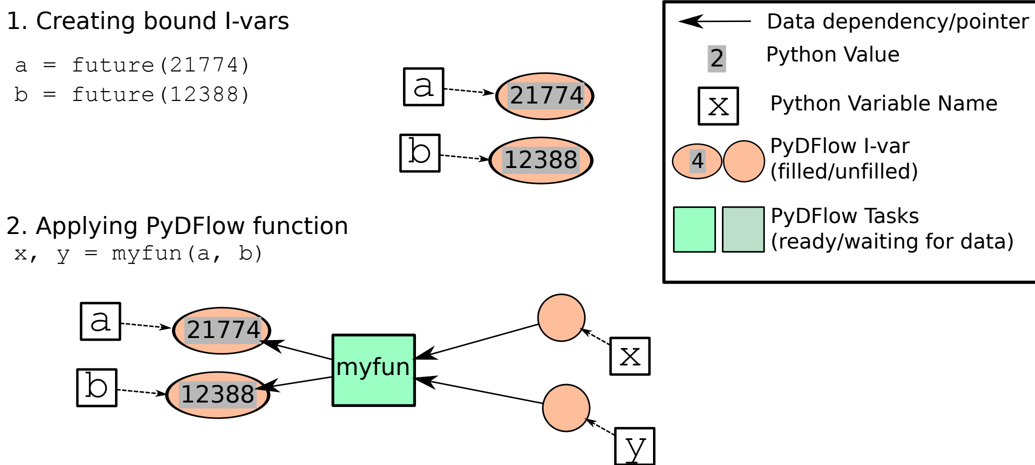


Figure 5.6: This figure illustrate a task graph being constructed as I-vars are created and PyDFlow functions are called to construct tasks and I-vars.

the second argument of the decorator (§5.11).

- construct a PyDFlow task with an output I-var or output I-vars with the appropriate output types (the first argument of the decorator), as shown in Figure 5.6.

The I-vars returned will be unbound, but the task's output can be redirected to another bound I-var if needed.

The PyDFlow task constructed represents a suspended computation that can be run at some later time. When the task is later run, several actions take place:

- Data, or references to data are unpacked from the input I-vars and transformed it into the form required by the real function. The exact transformation will depend on the task type: for example for file I-vars the file path is provided to the function as a string.

- The original function is invoked with the unpacked data.

- If the task is executed asynchronously, the task is dispatched to the appropriate task executor.

- Once the task is finished, the task and I-var states are updated and the PyDFlow graph executor is notified so it can schedule any newly enabled tasks.

5.7 The Redirection Operator

The operator <<, which we call the redirection operator, has already been introduced as a way to direct the output of a task to an I-var. In this section we briefly elaborate on its implementation.

The return value of a PyDFlow task is in fact always a new unbound I-var. The redirection operator is a binary operator that takes two I-vars, for example with the statement `x << y`. Here the I-var `y` is redirected to `x`, which means that the task that previously would have written its output to `y` will now write its output to `x`.

A PyDFlow function returns an I-var, so this permits us to write `x << f()` if we want to direct the output of `f()` to the I-var `x`.

In contrast, using the assignment operator (ie. `x = f()`) would assign the newly created I-var to the variable `x`, overwriting the reference to the original `x`. So, = assigns a Python value to a Python variable, while << assigns the present or future contents of one I-var to another I-var.

It is also possible to use an idiom that permits binding and assigning to a Python variable in a single statement. The statement `x = localfile("/path/to/file") << f()` expresses both the binding of an I-var and the direction of a task's output to the I-var.

5.8 Task Graph Representation

The suspended computation that is created by successive PyDFlow function calls can be thought of as an in-memory task graph of linked Python objects. It is a graph of Tasks and I-vars: each Task is linked to its input and output I-vars.

Whenever a PyDFlow function is applied to some arguments, a new task is created, representing the suspended computation. No execution of tasks occurs until the value of one of the I-vars is demanded using the *get* or *spark* methods.

Execution of tasks is performed by an execution model that traverses the task graph, executing the tasks needed to fill the target I-var. As execution proceeds, links between tasks are severed when no longered necessary, allowing the garbage collector in Python to clean up unneeded data objects. A simple program, and the execution of the corresponding task graph is shown in Figure 5.7. The execution algorithms and approach are described in detail in Chapter 6.

The current state of a task or I-var is described by a state tag that is attached to the corresponding in-memory Python object. Figure 5.1 illustrate the states that an I-var can be in, and the events that trigger state transitions. The state transitions for a task are more complex, as they are closely tied to the graph execution strategy. They are described in detail in Section 6.2.

5.9 Laziness

As discussed in Section 3, the choice of strict versus lazy evaluation poses a significant challenge to exploiting parallelism with a lazy evaluation strategy. For-

```
 1 # Define a new I-var type to hold integers
 2 IntIvar = py_ivar.subtype()
 3
 4 # Deferred function of type (IntIvar, IntIvar) -> IntIvar
 5 @func((IntIvar), (IntIvar, IntIvar))
 6 def add(a, b):
 7     return a + b
 8
 9 @func((IntIvar), (IntIvar, IntIvar))
10 def gcd(a, b):
11     # Sequential gcd calc with Euclid's algorithm
12     while b != 0:
13         oldb = b
14         b = a % b
15         a = oldb
16     return a
17
18 # Construct task graph for simple computation,
19 # first inserting values into IntIvar containers
20 x = add(gcd(IntIvar(21774), IntIvar(12388)), IntIvar(4))
21
22
23 # execute it, block on I-var, then print value
24 print(x.get())
```

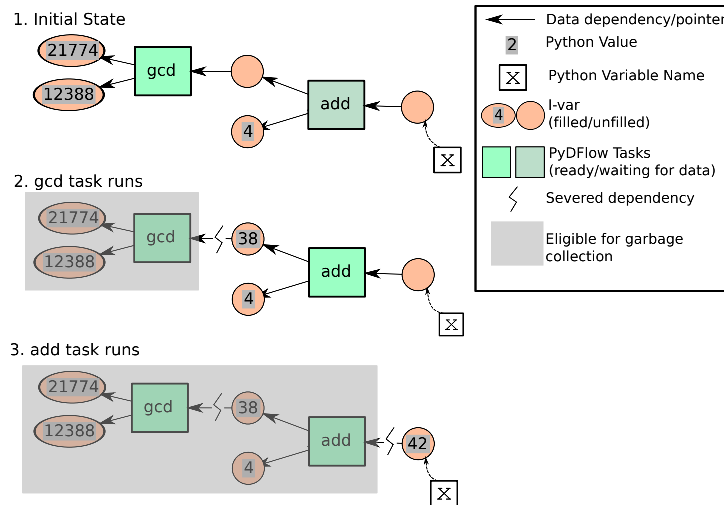


Figure 5.7: This diagram illustrates the evolution of the in-memory task graph for a trivial example. The propagation of data through the task graph and enabling of tasks can be seen. The interaction of task graph execution with the garbage collector is shown. In the CPython implementation, with reference counting garbage collection, tasks and I-vars will be collected immediately once eligible.

tunately in PyDFlow, many of the issues to do with laziness are not present: all inputs to tasks are strict by necessity: if the task is an external binary, for example, then it cannot be run unless all of it's input files are available (ie. its input I-vars are filled). This makes it easy to identify parallelism: if a task has more than one argument, the executor knows immediately that can safely evaluate them all in parallel.

However, wherever possible PyDFlow is lazy: when a PyDFlow task function is invoked, evaluation does not commence immediately. Execution of a task graph only occurs on demand, when the contents of an I-var that is part of the graph is forced. The lazy evaluation strategy is useful mainly because it allows I-vars to be constructed, manipulated and passed around without triggering execution. Once the I-var is forced, race conditions are possible as the graph executor may be executing tasks and updating I-vars in the graph. Laziness allows the I-var and task graph to be modified after creation without risk of this type of race condition.

This was particularly useful in the implementation of the redirection operator (§5.7), which redirects the output of a task to a different I-var. If the task started running as soon as the task was created, then then the task may start running and writing its output to the wrong I-var before the redirection operator is applied, meaning that the data may need to be copied from one place to another unnecessarily.

5.10 Compound Functions

Given the features described up to now, there is still a serious limitation in PyDFlow: only fixed task graphs can be specified. Furthermore, if we have a task graph consisting entirely of tasks that are strict in all their argument, and that cannot recursively create new tasks, then the entire task graph to be executed has to be instantiated in memory before it is executed. This is contrary to one of our design goals. Our solution to this problem is to have a new class of task that can recursively create new tasks, which we call *compound* tasks, matching the name in Swift. In contrast, the previously defined tasks are called *atomic* tasks, as they are indivisible. A compound function has the following properties:

- It manipulates the I-vars themselves rather than contents of I-vars.
- It is non-strict in its arguments: the arguments do not have to be reduced to a filled I-var before the expansion occurs.

An example using a compound function to perform a parallel merge sort is shown in Figure 5.8.

5.11 Type System

We have included a basic type system for PyDFlow functions. One of the stronger rationales for a type system is that it can catch errors at compile time. While Python, by design, cannot support static type checking before execution, with a system of deferred execution, type-checking at task graph construction provides some similar benefits to type-checking at compile time, as it can catch type errors before computation commences.

Python's type system is a class-based system with multiple inheritance. Every PyDFlow I-var already has a Python type t by virtue of being a subclass of the base PyDFlow `Ivar` class. For example, all in-memory Python variables are subclasses of *py_ivar*, any file manipulated by *app* functions is a subclass of *file_ivar*, and files on the local filesystem are subclasses of *localfile*, which is itself a subclass of *file_ivar*.

Every PyDFlow function has a function type mapping from tuples to tuples of the form $(t_1, ..., t_n) \rightarrow (s_1, ..., s_m)$, where t_i is an input argument, and s_i is a return value. The type of a PyDFlow function is specified as arguments to a decorator. For example, `@func((A, B), (C, D, F))` specifies that the function should have type $(C, D, F) \rightarrow (A, B)$. The output types of the function must all be I-var types (because all outputs are wrapped in I-vars), but the input types can be any valid Python type, or `None` if type checking should be disabled for that input argument. In the current implementation, the types of input arguments to a function are checked at runtime when a function is applied. If the I-var types don't match the type signature of the function, a type error will occur. This means that type errors are caught before the task begins to execute. The type checking handles optional and keyword arguments correctly.

```
# define new file types to better represent different classes of files
numfile = localfile.subtype()
sorted_numfile = numfile.subtype()

@app((numfile), ())
def make_empty():
    return App("touch", outfiles[0])

@app((sorted_numfile), (numfile))
def sort(f):
    return App("sort", "-n", "-o", outfiles[0], f)

@app((sorted_numfile), (sorted_numfile, sorted_numfile))
def merge(f1, f2 ):
    # use unix sort utility to merge two sorted files
    return App("sort", "-n", "-o", outfiles[0], "-m", f1, f2)

@compound((sorted_numfile), (Multiple(numfile)))
def merge_sort(*unsorted):
    # unsorted is a list of numfile ivars
    if len(unsorted) == 0: # if no inputs provided
        return make_empty()
    elif len(unsorted) == 1:
        return sort(unsorted[0])
    else:
        # split the list in two and handle recursively
        split = len(unsorted)/2
        f1 = unsorted[0:split]
        f2 = unsorted[split:]
        return merge(merge_sort(*f1), merge_sort(*f2))

# 100 unsorted files from unsorted_0.txt to unsorted_99.txt
unsorted = [numfile("unsorted_%d.txt" % i) for i in range(100)]

result = sorted_numfile("sorted.txt")
# * operator expands so that each list element is an input argument
result << merge_sort(*unsorted)
result.get()
```

Figure 5.8: Example code performing a parallel merge sort of a number of files on disk using the Unix *sort* utility. The files must be in text format and consist of numbers separated by line breaks. The function *merge_sort* implements divide-and-conquer in order to merge sort the files, with the task graph forming a reduction tree with the unsorted files at its leaves, intermediate sorted files at its internal nodes, and the sorted file at its root. The code in the *merge_sort* function is invoked lazily when the graph executor explores a particular branch of the tree, unfolded it gradually.

If an input argument's type is not a subclass of `Ivar`, then the value of the argument is passed directly to the function body. This is useful for parameterizing I-vars: it is fairly common that users would want to provide parameters to tasks that are numbers, strings, or some other primitive data type. Special processing of `Ivar` arguments can be implemented, depending on the type of task and I-var. For example, if a `file_ivar` I-var representing a file is passed to an `@app`-decorated function, then the file's path is substituted for the I-var as the input argument.

When it comes to the outputs of a function, PyDFlow's type system is limited in the checking it can perform on the output types, because the framework regards the contents of I-vars as opaque. It simply assumes that the output of a task is a valid instance of that type, and tags it as such.

This typing scheme for executables and files is derived from Swift, but has also been explored before by the Saguaro operating system [3], where executables can be typed in a similar way. Saguaro can deal with the output type problem because it integrates a Universal Type System, which specifies the internal structure of files, but this assumes a particular machine-independent file structure and is inappropriate for a system that must

5.12 Similarity to Swift

As a result of all of the design decisions described, we end up with a scripting interface that bears a close resemblance to Swift in functionality and style, but is implemented as a library and can interact with standard Python code. To illustrate, Figure 5.9 and Figure 5.10 show an example script that "processes" and analyzes NASA MODIS satellite data [18] written in PyDFlow and Swift respectively.

Both scripts start with a set of satellite image tiles in which each pixel has been classified into one of a small number of land usage categories, such as forest, farmland, desert or ocean. The scripts perform the following steps:

1. Extract the number of time each land use appears in the pixels of each tile.
2. Analyze these statistics to determine the 10 tiles with the largest number of urban pixels.

```
# Define new localfile types
imagefile = localfile.subtype()
landuse = localfile.subtype()

@app((landuse), (imagefile, int))
def getLandUse(input, sortfield):
    return App('getlanduse.sh', input, sortfield, stdout=outfiles[0])

@app((localfile, localfile), (int, int, Multiple(landuse)))
def analyzeLandUse(usetype, maxnum, *inputs):
    return App('analyzelanduse.sh', outfiles[0], outfiles[1], usetype, maxnum,
        modisdir, *inputs)

@app((imagefile), (imagefile))
def colormodis(input):
    return App('colormodis.sh', input, outfiles[0])

# Mappers return read-only array.
geos = GlobMapper(imagefile, os.path.join(modisdir, '/*.tif'))
land = SubMapper(landuse, geos, '(h..v..).*$', '\\1.landuse.byfreq',
    directory=outputdir)

# Find the land use of each modis tile
# zip allows us to iterate over two lists in tandem
# PyDFlow << operator directs output to mapped file
for l, g in zip(land, geos):
    l << getLandUse(g,1);

# Find the top 10 most urban tiles (by area)
urbanUsageType=13;
bigurban = localfile(os.path.join(outputdir, 'topurban.txt'))
urbantiles = localfile(os.path.join(outputdir, 'urbantiles.txt'))
(bigurban, urbantiles) << analyzeLandUse(urbanUsageType, 10, *land)

# Map the files to an array.
# script blocks here on open() command until urbantiles ready
urbanfilenames = [line.strip() for line in urbantiles.open().readlines()]

# Use built-in map to apply imagefile constructor to filenames
urbanfiles = map(imagefile, urbanfilenames)

# Create a set of recolored images for just the urban tiles
recoloredImages = []
for uf in urbanfiles:
    recoloredPath = os.path.join(outputdir,
        os.path.basename(uf.path()).replace('.tif', '.recolored.tif'))
    recolored = imagefile(recoloredPath) << colormodis(uf)
    recoloredImages.append(recolored)

# Start everything running and wait until completion
waitall(recoloredImages)
```

Figure 5.9: MODIS example in Python that demonstrates the look and feel of the language. This script shows a variety of different ways that I-vars can be manipulated and stored using Python data structures. For example, the *recoloredImages* list is constructed by appending items one at a time, while the *land* list is constructed using a "mapper" function.

```
1  type file;
2  type imagefile;
3  type landuse;
4
5  app (landuse output) getLandUse (imagefile input, int sortfield) {
6    getlanduse @input sortfield stdout=@output ;
7  }
8
9  app (file output, file tilelist) analyzeLandUse (landuse input[], int usetype,
      int maxnum, string input_loc) {
10   analyzelanduse @output @tilelist usetype maxnum input_loc @filenames(input);
11 }
12
13 app (imagefile output) colormodis (imagefile input) {
14   colormodis @input @output;
15 }
16
17 imagefile geos[]<filesys_mapper; location=input_loc, suffix=".tif">;
18 landuse   land[]<structured_regexp_mapper; source=geos,match="(.*)\\.tif",
19             transform="\\1.landuse.byfreq">;
20
21 # Find the land use of each modis tile
22 foreach g,i in geos {
23   land[i] = getLandUse(g,1);
24 }
25
26 # Find the top 10 most urban tiles (by area)
27 int UsageTypeURBAN=13;
28 file bigurban<"topurban.txt">;
29 file urbantiles<"urbantiles.txt">;
30 (bigurban, urbantiles) = analyzeLandUse(land, UsageTypeURBAN, 10, input_loc);
31
32 # Map the files to an array
33 string urbanfilenames[] = readData(urbantiles);
34 trace(urbanfilenames);
35 imagefile urbanfiles[] <array_mapper;files=urbanfilenames>;
36
37 # Create a set of recolored images for just the urban tiles
38 foreach uf, i in urbanfiles {
39   imagefile recoloredImage <single_file_mapper;
40         file=@strcat(@strcut(urbanfilenames[i],"(h..v..)"),".recolored.tif")>;
41   recoloredImage = colormodis(uf);
42 }
```

Figure 5.10: MODIS example in Swift

3. Recolor these 10 tiles to be human-viewable.

SECTION 6
Execution of Task Graph

In this section we describe how the PyDFlow runtime executes a task graph within the graph executor module. Viewed as a black box, the interface to the graph executor model is simple: to initiate execution, an I-var is passed to the graph executor module. The graph executor model then traverses the task graph and executes all necessary tasks to fill the I-var. As the graph is executed, the state of the task graph is updated and I-vars are filled. User code can receive notification of when data is ready by waiting on an I-var, or by registering to receive callbacks from an I-var. The overall flow of data is illustrated in Figure 6.1, and the process of updating a task graph was shown previously in Figure 5.7.

The graph executor module has a configurable number of worker threads that collaboratively execute a task graph.

6.1 Executing Synchronous and Asynchronous Tasks

The execution of a PyDFlow task is broken up into two phases:

- *Synchronous phase:* in which some amount of computation occurs in a graph executor worker thread.
- *Asynchronous phase:* in which the task is executed outside of the graph executor module, for example on a remote host. This asynchronous execution would be supported by a separate asynchronous task executor component.

The implementation of each task type determines what computation occurs in each stage. The asynchronous phase is optional. For example, local Python function tasks only need to execute synchronously in a worker thread, but a task that invokes an external application does so asynchronously.

Tasks that have only a synchronous phase are called *synchronous* tasks, while tasks that also have an asynchronous phase are called *asynchronous* tasks. The distinction is important for the executor, as in order to support an asynchronous

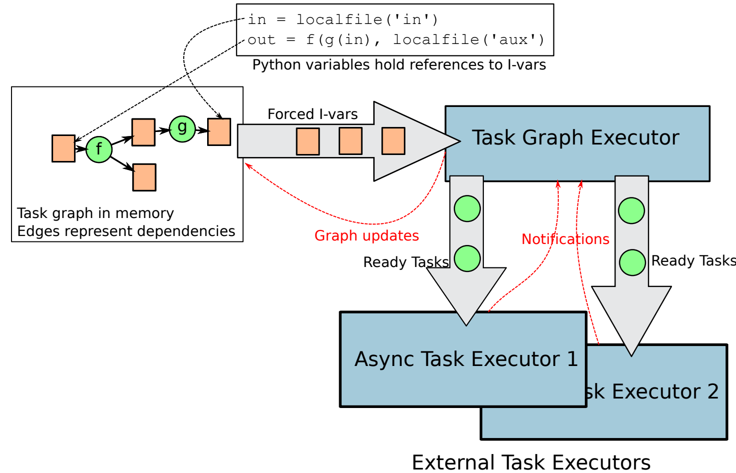


Figure 6.1: A high level view of task graph execution showing the flow of tasks, I-vars and messages between different components of PyDFlow.

task type, the graph executor, which is responsible for managing inter-task dependencies, needs to be notified by the asynchronous task executor when tasks complete.

This can be implemented by inheriting from the PyDFlow `Task` class (or a subclass thereof) and overriding the exec method, as showing in Figure 6.2.

6.2 Depth First Traversal of Task Graph

When an I-var is forced through the *get()* or *spark()* methods, then the I-var is added to a graph executor queue and it becomes a *target* that the graph executor will fill by resolving all of its dependencies.

It does this with a depth first search (DFS) of the directed acyclic task graph, maintaining a stack of unresolved I-vars. At any point in time a graph executor thread is focusing on a *target* I-var. At first it checks the task for which the I-var is an output: running that task is required to fill the I-var. If it can immediately resolve the target by executing the task, it will execute that task. If the task has unresolved dependencies, it will continue the search to resolve the dependencies. The depth first exploration can end in two ways. A I-var may be successfully resolved, and the new target obtained from the stack. Otherwise a target I-var

```
class CustomTask(AtomicTask):
    def _exec(self, continuation, failure_continuation,  contstack):
        << run user-provided task function to get task specification >>
        << add task, continuation, failure_continuation, contstack
                                                    to execution queue >>

    def isSynchronous(self):
        return False


def monitor_loop():
    """
    This function is an event loop run by a separate handler thread
    to manage asynchronous execution of tasks.
    """
    while 1:
        if << execution slot avail >>:
            << try to get new task from execution queue >>

            << if new task available, prepare input and output I-vars
                and start asynchronous execution >>

        << check if tasks have finished >>
        for all finished tasks:
            if task failed:
                failure_continuation(task, exception)
            else:
                continuation(task, contstack)
```

Figure 6.2: Pseudocode showing how a new asynchronous task type and corresponding executor can be implemented for PyDFlow.

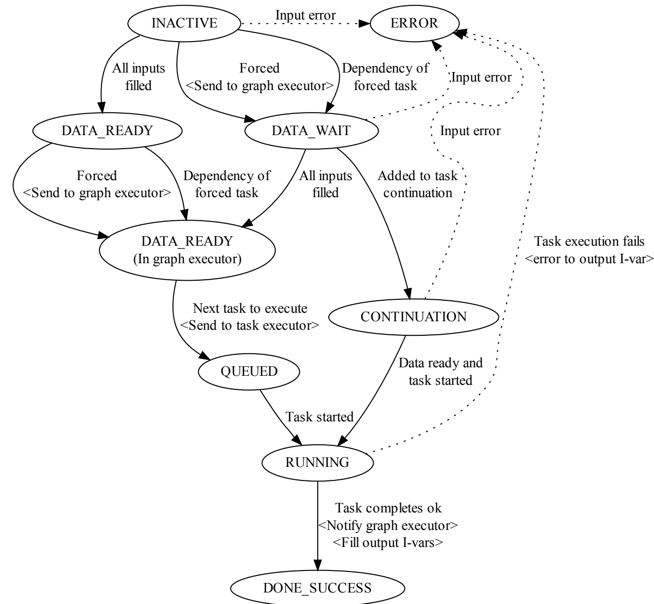


Figure 6.3: State Transition Diagram for PyDFlow tasks

might be encountered with pending dependencies: for example if all the tasks it depends on are being executed. In this case the graph executor must suspend the I-var, to be resumed (through the resume queue) when all of its dependencies are resolved. This process is illustrated in Figure 6.4.

All communication between the graph executor thread and client code of the library occurs through the task graph. The graph executor thread updates the state of all the I-vars and tasks in the task graph as execution proceeds Client code can query the state of I-vars, block on I-vars, and also register functions to be called when I-vars are filled.

The strategy has some similarity to the graph-reduction approach to evaluating applicative languages, where a graph represents the program, and it is executed by incrementally rewriting and reducing the graph [74]. The Glasgow Haskell compiler, the standard implementation of Haskell, uses a highly optimized version of graph reduction [44]. The optimizations used cannot be applied easily to PyDFlow, as they depend on the specific low-level details of the programming language and involve various low-level compiler tricks. In particular, knowing

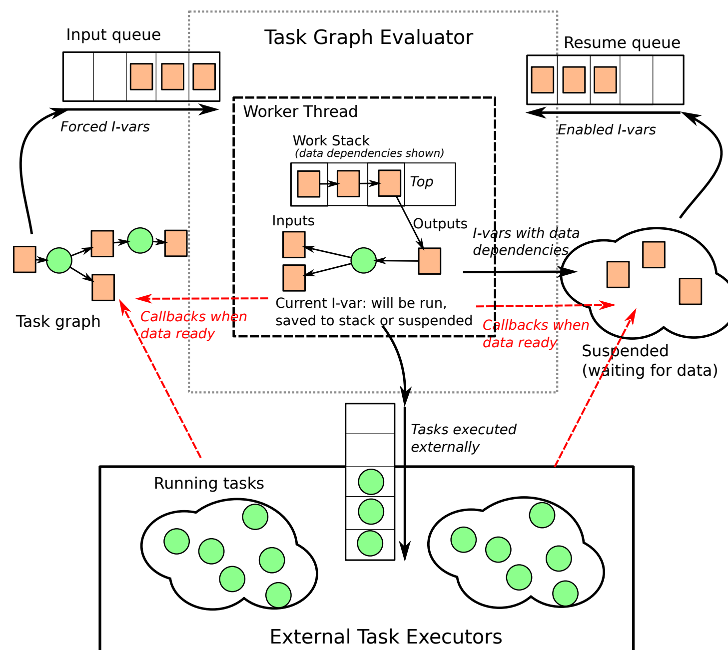


Figure 6.4: A detailed look at the PyDFlow graph executor module, showing how data passes between the worker thread stack, the input and resume queues, and the asynchronous executors.

the exact type and binary representation of all program data allows many sophisticated optimizations to be performed based on program analysis. As a result, the graph execution method used in PyDFlow resembles an unoptimized naïve graph reduction approach, where a PyDFlow task graph is executed by incrementally updating an in-memory graphical structure. However, the task graph execution in PyDFlow includes a number of tricks that can improve the performance of distributed programs.

The states and transitions for PyDFlow I-vars were shown earlier in Figure 5.1. The state diagram for PyDFlow tasks, shown in Figure 6.3, is significantly more complicated. The complexity comes for a number of reasons. In some cases we simply want to distinguish between two states simply so that the status of execution can be determined from the task graph: separate QUEUED and RUNNING states are not needed for correctness, but it will be useful in the future if a user wants a snapshot of the state of a longer-running computation. The rest of the complexity comes because of the modular nature of the implementation: the task object is accessed by a variety of different modules. In order to support asynchronous tasks that did not run in the graph executor thread, to avoid repeatedly dispatching the same task, for example if it was reached multiple times on the DFS of the task graph, it was necessary to distinguish task that were ready to run (DATA_READY) and tasks that had been dispatched to be executed asynchronously (QUEUED). Implementing multiple execution threads with workstealing (§6.3.3) also required the QUEUED state in order to avoid race conditions where multiple threads executed the same task. Promise pipelining (§6.4) required the addition of the CONTINUATION state, to describe a task that was waiting for data that would be propagated through a pipeline.

6.3 Parallel Evaluation

6.3.1 Limitation on Parallelism Imposed by Python

The amount of parallelism possible when running native python code is unfortunately limited by Python's Global Interpreter Lock (GIL), an artifact of the CPython implementation that uses a single shared lock to control access to in-

terpreter data structures.

Python's implementation uses OS threads to implement its *threading* module. However, only one thread in each Python instance can be executing interpreted Python code at a given time.

In spite of this limitation, there are a number of reasons why providing a multithreaded evaluator implementation is useful:

- If the synchronous parts of tasks perform a significance amount of I/O, then we can get speedups. The GIL is released when a Python process is blocked for I/O, so having multiple threads of execution permits.
- If the computationally intense parts of synchronous tasks are performed outside of Python, for example in a C extension module. It is possible for Python extension written in a language such as C or Fortran to release the GIL if the extension procedure does not require access to the Python interpreter.

  Typically applications written in Python for which performance is important will have significant portions of the program or supporting libraries that are not interpreted Python code. For example, the popular *numpy* library for Python provides efficient subroutines for linear algebra and other mathematical operations that release the GIL while they execute.
- it is useful as a initial step towards an implementation of a distributed memory evaluator
- it provides insight into parallel evaluation strategies for implementations of Python without the GIL limitation, and for different languages.

6.3.2 Parallel Graph Execution

Treating the evaluation of the task graph as an exercise in graph traversal and updates translates in a relatively straightforward way to a parallel evaluation strategy in shared memory: we just need multiple threads of evaluation, a way to create and share work between threads and logic to ensure safety.

Parallel versions of the g-machine have been proposed, such as the (v, g) machine [5]. The parallel evaluation technique used for PyDFlow has similarities. The (v, g) machine relies on a explicit "spark" annotation to specify that a graph node is a candidate for parallel evaluation. When a node is sparked, it is entered into a shared work pool, from which evaluation threads draw work. Flags on a node are marked to indicate whether a thread has started work on evaluating a node.

The current parallel evaluation strategy implemented for PyDFlow is a bit more complex. I-vars can be made targets (i.e. sparked) when the *get()* or *spark()* methods are called by client code. These I-vars are entered into the input queue, shown in Figure 6.4. A simple strategy for parallel execution would be for each execution thread to add unexplored branches during the DFS back into the input queue, so that they can be executed by other threads. Such a *work-sharing* strategy with a central queue, however, has downsides: adding and removing items from the queue constantly requires significant synchronization overhead, and the strategy does not have good data locality properties. Instead we use a *work-stealing* approach, where each thread retains ownership over all of the target I-vars in encounters in its search until an idle thread "steals" the work. Different work-stealing approaches can produce good load balance, increase the probability of data dependencies being in a processor's cach and reduce the amount of synchronization by avoiding querying a central queue [10].

6.3.3 Work-stealing

The PyDFlow graph executor, showing in Figure 6.4, can be augmented easily to support work-stealing. Multiple worker threads are added, all of which share the same input queues and resume queues. Worker threads will attempt to get work from either of these queues, but if none is available, they will attempt to steal target I-vars from other worker threads.

Their work stacks can then be stored in a deque (double-ended queue) data structure, which is designed to support popping data items from either end. The thread performs a DFS by exploring the graph, adding visited nodes to the right side of the deque and backtracking by popping them off the right. Other idle

```
1 # a and b are intermediate values, c is the result
2 a = f(x)
3 b = g(a)
4 c = h(b)
5 c.get()
```

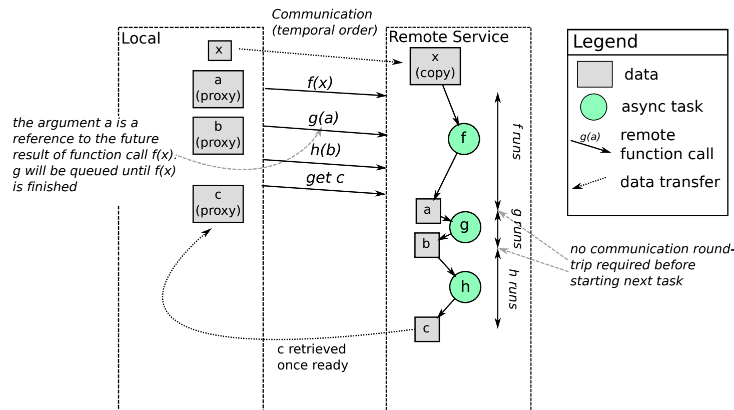


Figure 6.5: A simple example of promise pipelining

threads will attempt to find work by popping data items off the left side of the deque. One can see that stealing from the left side is likely to result in stealing more work, which is desirable for efficiency and load balancing [54]. The stealing strategy is randomized: a thread looking for work probes the other worker threads in a random order.

In order to avoid a situation where all threads have no work to do, and are unnecessarily polling each others' deques for work, a mechanism is provided for the threads to go idle if all threads have no work.

6.4 Promise Pipelining

The use of futures allows promise pipelining [49]. Promise pipelining takes advantage of the dependency information provided by deferred execution and futures to allow data dependencies to be resolved and computations to immediately proceed without the intervention of a centralized task manager. A simple example of promise pipelining is shown in Figure 6.5. In a distributed setting, the boundary between local and remote would be a network. In PyDFlow, the remote executor would not necessarily have to be on a separate machine: promise pipelining can

be used between PyDFlow's graph execution code and a custom asynchronous task executor.

The advantages of promise pipelining are:

1. reducing the latency between task invocations if the executor can support resolving dependencies itself, as there is no network round-trip required to notify completion of one task and start the execution of another.

   This is even more useful if the system is implemented in a centralized way with a central master dispatching tasks to the workers fashion, as it reduces the load on the master and also reduces the impact of master latency on the overall performance of the system.

2. potentially reducing the number of required data copies (this depends on implementation - if the data is stored on the remote machine and a reference returned, this doesn't matter so much).

3. reducing scheduling and task dispatch overhead if a string of dependent tasks can be dispatched as a unit.

Having a system that understands promise pipelining and can delegate some scheduling functions to a particular executor is also very useful in being able to implement executor-specific optimizations. For example, the CUDA toolkit for GPU programming supports a construct for asynchronous task execution and data transfer called *streams*, where a sequence of dependent data transfers and tasks can be chained together and executed in sequence by the GPU, without requiring any intervention from the driver program to proceed [17]. This can help to improve utilization of the GPU by making it possible to overlap CPU execution, data transfers from main memory to GPU memory and GPU computations. In order to support such optimizations, the use of promise pipelining means that a CUDA executor module for PyDFlow could be take advantage of the *stream* mechanism to execute multiple GPU tasks in an overlapped manner and with greater efficiency.

Another example where promise pipelining could be used to implement an executor-specific optimization is the case where output data of a task can begin to be read before the process writing to it finishes: i.e. when the intermediate data

can be streamed (a completely different type of stream to the above-mentioned). In this case dependent tasks can be scheduled before the task finishes. By using promise pipelining to provide dependency information to a task executor, the executor can then start dependent tasks executing earlier. It is difficult to anticipate all possible optimizations for different executors, so by having an understanding of promise pipelining built into the task graph executor, it is possible to provide enough dependency information to an executor so that it can implement the optimizations itself.

CIEL implements promise pipelining in order to achieve the above goals, including streaming [57]. There are a number of different levels of pipelining that could be implemented:

1. *no pipelining*: the PyDFlow task graph executor is entirely responsible for managing data dependencies. When a task finishes, PyDFlow is notified, updates task graph state, then will dispatch any tasks that are now disable to the executor.

2. *sequential pipelining*: when the graph executor is performing a depth first search on the task graph, it keeps track of a sequence of tasks that have no other unresolved dependencies aside from the preceding task in the sequence. When a runnable task is found, the runnable task and this sequence of tasks (which can be thought of as the continuation of the runnable task) are sent to the executor. This means that the executor can safely execute these tasks in sequence, notwithstanding a task failure. For a given backend executor, we will generally want to execute these dependent tasks all on the same resource. Notifications of task completion can be send asynchronously back to the master. With this scheme, promise pipelining can only occur over non-branching sequences of tasks.

3. *branching pipelining*: when the task graph executor still identifies sequences of tasks and dispatches these to the executors, but the executor also provides a way that allows dependencies to be sent to the executor. These dependencies specify a number of I-vars for which the data is needed to proceed, and a task that should be executed when the data is ready. This way an external

executor can take responsibility for managing dependencies, perhaps in a distributed manner.

PyDFlow currently only implements *sequential pipelining*. This is done while a worker thread (§6.2) is performing a depth-first traversal of the task graph. As the traversal proceeds, for each I-var to be resolve, a *continuation list* is built that contains a sequence of tasks. In this sequence, each task only needs data generated by the previous task to become runnable. If a task only needs one input to become runnable, then the depth first traversal can immediately move to that input task and the dependent task and its continuation list can be added to the continuation list of the input task.

The above strategies are complicated by scenarios where the task graph consists of tasks that are executed by two different executors. In these situations we need to fall back on the first strategy and have dependencies resolved by the general PyDFlow infrastructure. For *sequential pipelining*, this is implemented by having the asynchronous executor pass back any tasks in the continuation list that it did not run.

SECTION 7
Applications

7.1 Montage Astronomical Imaging Workflow

One example application we have implemented in Python with PyDFlow is a simple workflow constructing an astronomical image using the Montage [42] engine. This workflow uses an online service that provides access to tiled images of different parts of the sky. The workflow constructs a false color composite image of a small 3×3 degree section of the sky using three different bands of light for the red, blue and green I-vars of the image. The image is shown in Figure 7.1. It first contacts an online service, using the `mArchiveList` application to find the image tiles are available that fall within that region for a particular color band. It then parses the text file produced by `mArchiveList` and downloads each image tile separately with `mArchiveGet`. Once the images are downloaded, the tiles are reprojected to the desired projects for the final image with `mProjectPP`, and then combined together with `mAdd` to create a single monochrome image for each band that covers the 3×3 degree section of the sky.

The workflow must perform all these steps for the different light bands that will be used in the final image (in this example red, blue and infrared bands are used). Then, finally, the monochrome images for each of the I-vars are combined into a single result image using `mJPEG`.

Some `@app` definitions for the Montage engine, which can be composed to create different Montage workflows, are shown in Figure 7.2. A relatively simple example script that generates an image of the Pleaides star cluster is shown in Figure 7.3. This script was benchmarked against serial and Swift implementations to demonstrate a speedup over a serial implementation, and its competitiveness with Swift. The results are shown in Table 7.1.

7.2 URL Fetcher: Integration with other Python Code

PyDFlow has been designed with the consideration that it should fit in with the rest of the Python language, permitting it to be composed with ordinary sequential

Figure 7.1: A scaled down astronomical image montage of the Pleaides star cluster.

System	Runtime (s) (5 trials)
bash (serial)	822 ± 14
Swift (parallel)	370 ± 12.5
Swift+Coasters (parallel)	358 ± 40
PyDFlow (parallel)	300 ± 15

Table 7.1: Performance of a Montage workflow creating a 3×3 degree image of the Pleaides start cluster, comparing serial execution time with Swift and PyDFlow on a four processor desktop PC. The job throttle was set to five concurrent jobs for PyDFlow, ten concurrent jobs for Swift, and 15 concurrent jobs for Swift+Coasters . These settings were obtained by testing a range of parameters from four to 20 for each system. The time measured does not include the time taken to download image tiles.

```
1  # PyDFlow file types for Montage.
2  #===============================================================
3  MosaicData = localfile.subtype() # Text file with mosaic metadata
4  MTable = localfile.subtype()
5  MImage = localfile.subtype() # Image tile in FITS format with image metadata
6  MStatus = localfile.subtype() # mFitplane status file
7  JPEG = localfile.subtype() # Image in JPEG format
8
9  class RemoteMTable(MTable):
10     """ Text table describing set of images that can be downloaded """
11     def read_urls(self):
12         """ Returns a list of (url, image name) pairs from table """
13         for line in self.open().readlines()[3:]: # ignore 3 line header
14             toks = line.split()
15             yield (toks[-2], toks[-1])
16
17 # PyDFlow apps for Montage.
18 #===============================================================
19 app_paths.add_path(path.join('/var/tmp/code/Montage_v3.3', "bin"))
20
21 @app((RemoteMTable), (str, str, str, None, None))
22 def mArchiveList(survey, band, obj_or_loc, width, height):
23     """ Download info from IRSA about image tiles for location in sky """
24     return App("mArchiveList", survey, band, obj_or_loc, width, height,
           outfiles[0])
25
26 @app((MImage), (str))
27 def mArchiveGet(url):
28     """ Download an image from the archive """
29     return App("mArchiveGet", url, outfiles[0])
30
31 @app((MImage), (MImage, MosaicData))
32 def mProjectPP(raw_img, hdr):
33     """ reproject using plane-to-plane algorithm """
34     proj_img = outfiles[0]
35     return App("mProjectPP", "-X", raw_img, proj_img, hdr)
36
37 @app((MImage) ,(MTable, MosaicData, Multiple(MImage)))
38 def mAdd(img_tbl, hdr, *imgs):
39     """
40     adds together images.
41     img_tbl specifies images to be added, all images must be in same directory
42     """
43     return App("mAdd", "-p", path.dirname(imgs[0]), "-n", img_tbl, hdr,
           outfiles[0])
44
45 @app((MTable), (Multiple(MImage)))
46 def mImgtbl(*imgs):
47     """ Generate text table with image metadata """
48     return App("mImgtbl", path.dirname(imgs[0]), outfiles[0])
49
50 @app((JPEG), (MImage, MImage, MImage))
51 def mJPEGrgb(rimg, gimg, bimg):
52     """
53     Create rgb jpeg image with three FITS images for R, G, B channels
54     """
55     return App("mJPEG", "-out", outfiles[0],
56         "-red", rimg, "-1s", "99.999%", "gaussian-log",
57         "-green", gimg, "-1s", "99.999%", "gaussian-log",
58         "-blue", bimg, "-1s", "99.999%", "gaussian-log")
```

Figure 7.2: Shared Montage definitions in PyDFlow

```
1
2  def strip_ext(path, ext):
3      """ Strip given file extension if present """
4      n = len(ext)
5      if len(path) >= n:
6          if path[-1*n:] == ext:
7              return path[:-1*n]
8          else:
9              return path
10
11 dims = (3,3) # size of montage in degrees
12 srcdir = os.path.dirname(__file__) # File source directory
13 header = MosaicData(path.join(srcdir, "pleiades.hdr")) # mosaic info
14
15
16 def archive_fetch(bands):
17     """ Find which image files are needed for this band and sky region """
18     table = RemoteMTable(path.join(bands, 'raw', 'remote.tbl'))
19     return table << mArchiveList("dss", bands,"56.5 23.75", dims[0], dims[1])
20
21
22 def process_one_band(bands, tbl):
23     """ Create merged image for one band """
24     # Download images from server separately
25     raw_images = []
26     for url, fname in tbl.read_urls():
27         # raw images go in the raw subdirectory
28         raw_path = path.join(bands, 'raw', fname)
29         raw_image = MImage(raw_path) << mArchiveGet(url)
30         raw_images.append(raw_image)
31
32     # projected images go in the proj subdirectory
33     projected = [MImage(path.join(bands, 'proj',
34                     # remove .gz suffix for new file
35                     strip_ext(path.basename(r.path()), ".gz")))
36                 for r in raw_images]
37     # Now reproject the images
38     for proj, raw_img in zip(projected, raw_images):
39         proj << mProjectPP(raw_img, header)
40
41     # Generate a temporary table with info about images
42     proj_table = mImgtbl(*projected)
43     # Now combine the projected images into a montage
44     band_img = MImage(path.join(bands, bands + ".fits"))
45     return band_img << mAdd(proj_table, header, *projected)
46
47 # Get info for the three bands we are interested in
48 allbands = ['DSS2B', 'DSS2R', 'DSS2IR']
49 img_tables = [archive_fetch(bands) for bands in allbands]
50
51 # Stitch together tiles in each band
52 band_imgs = [process_one_band(bands, tbl)
53             for bands, tbl in resultset(img_tables, allbands)]
54
55 # Make a false-color JPEG image out of the three bands
56 res = JPEG("DSS2_BRIR.jpeg") << mJPEGrgb(band_imgs[2],
57                             band_imgs[1], band_imgs[0])
58 res.get() # Calling get triggers execution
```

Figure 7.3: Montage script in PyDFlow that creates an image of the Pleiades star cluster.

```
THRESHOLD = 5

Urllist = py_ivar.subtype()
Pagelist = py_ivar.subtype()

@func((Pagelist), (Urllist))
def fetch(urls):
    """
    Fetch a list of urls and store data in a list
    """
    data = []
    for url in urls:
        data = urllib.urlopen(url).read()
        data.append(read)
    return data

@compound((Pagelist), (Urllist))
def fetch_split(list):
    """
    Recursively split the list until it is in chunks of size
    <= THRESHOLD, then fetch data
    """
    n = len(list)
    if n <= THRESHOLD:
        return fetch(list)
    else:
        return join(fetch_split(list[:n/2]),
                    fetch_split(list[n/2:]))


@func((Pagelist), (Pagelist, Pagelist))
def join(list1, list2):
    """
    Rejoin result lists
    """
    return list1 + list2

def fetchurls(urls):
    """
    Fetch a number of web pages in parallel
    The use of PyDFlow is not visible to callers of this function
    """
   return fetch_split(urls).get()
```

Figure 7.4: PyDFlow code for a parallel URL fetcher.

Python programs without any unnecessary limitations.

This example demonstrate one way in which PyDFlow can be integrated as a component of a larger Python program, allowing incremental parallelisation of a program. This example shows how Python can use an existing Python library- `urlib`, and also how PyDFlow can be used transparently to parallelize a single function. The code for a program that fetches web URLs in parallel is shown in Figure 7.4. A large list of URLs is broken down into chunks of at most size 5 using the `fetch_split` compound procedure, and then multiple PyDFlow worker threads download the URLs in parallel.

This example also illustrates a potential problem with PyDFlow functions recursively calling each other: if a PyDFlow worker thread calls the function `fetch_split` and blocks on the return value, we create the potential for a deadlock, given that there is a fixed number of worker threads.

In the initial implementation of PyDFlow deadlocks did occur if all the worker threads were executing functions that then blocked on PyDFlow futures, as no worker threads were able to make progress.

This was fixed by detecting when a worker thread attempts to block on a PyDFlow variable, and assigning that thread to evaluate the task's dependencies. The only downside to this approach is that if this happens recursively, the usage of the Python stack continues to increase.

SECTION 8
Extending PyDFlow with Python

Python offers additional language features that allow powerful and natural extensions to facilitate parallel programming in PyDFlow. I present some examples of the synergy between Python's language features and the PyDFlow library to illustrate the utility of PyDFlow.

8.1 Mappers

Swift [79] has a concept of a mapper as a primitive notion in a language. Section 5.3.2 discusses how one important aspect of mapping - directly binding I-var an I-var to an (opaque) storage location - is implemented in PyDFlow. Swift provides a range of different mappers that bind one or more Swift variables in sophisticated ways [71]. For example, the `structured_regexp_mapper` (see Figure 8.1) is essentially a function that takes an array of I-vars and performs a textual transformation on the file paths using a regular expression to obtain a new array of I-vars of the specified type bound to the transformed paths. The simple mapper behaves somewhat differently: the mapper is instantiated with a Swift variable type, and a file path prefix and a suffix such as `foo` and `.txt`. The mapper can then be indexed with integers if the mapped variable is an array, or named members if the mapped variable is a structure. File names corresponding to the array index or structure member with matching prefixes and suffixes are generated, as shown in Figure 8.1.

Many mappers can be implemented in PyDFlow as a function that takes a number of arguments and returns a mapped I-var or array of I-vars. An example of this is the *SubMapper* in Figure 8.2. This can be implemented in a straightforward manner as a Python function because its semantics mean that the complete set of mapped files is known based on the arguments to the mapper. A simple implementation of *SubMapper* could just return a standard Python list full of bound I-vars, but we implemented a read only list primitive to ensure that the array is write-once.

If mappings are performed after the mapping in instantiated, for example if

```
type type_a;
type type_b;

string s[] = [ "a.txt", "b.txt", "c.txt" ];
// array of files bound to above paths
type_a f[] <array_mapper;files=s>;

// array of files bound to to a.processed.txt, b.processed.txt, etc
type_b g[] <structured_regexp_mapper; source=f, match="(.*)\.txt",
                              transform="\1.processed.txt">;
```

```
type messagefile;
type mystruct {
  messagefile left;
  messagefile right;
};

app (messagefile t) greeting(string m) {
    echo m stdout=@t;
}
messagefile outfile[] <simple_mapper;prefix="baz",suffix=".txt">;

outfile[0] = greeting("hello"); // bound to baz0000.txt
outfile[1] = greeting("goodbye"); // bound to baz0001.txt

mystruct out <simple_mapper;prefix="qux",suffix=".txt">;

out.left = greeting("hello"); // bound to quxleft.txt
out.right = greeting("goodbye"); // bound to quxright.txt
```

Figure 8.1: Four different ways of using mappers to bind Swift variables to files in Swift: *array_mapper*, *structured_regexp_mapper simple_mapper* with array indexing and *simple_mapper* with named structure members. Some examples were adapted from the Swift [79] documentation.

```
1 # use the localfile constructor with map built-in function to emulate
      array_mapper
2 first = map(localfile, ['a.txt','b.txt','c.txt'])
3
4 # use regular expression transformation to bind to to a.processed.txt, etc
5 second = SubMapper(localfile, source=first, match='(.*).txt',
      transform='\\1.processed.txt')
```

```
1  messagefile = localfile.subtype()
2
3  @app(messagefile, str)
4  def greeting(m):
5      return App("echo", m, stdout=outfiles[0])
6
7  # instantiate a mapper object
8  mp = SimpleMapper(messagefile, prefix="baz", suffix=".txt")
9  mp[0] << greeting("hello") # goes to baz0.txt
10 mp[1] << greeting("goodbye") # goes to baz1.txt
11 mp.left << greeting("hello left") # goes to bazleft.txt
12 mp.right << greeting("goodbye right") # goes to bazright.txt
```

Figure 8.2: Equivalents to Swift mappers in Python. Python's *map* primitive can be used to emulate Swift *array_mapper*. *SimpleMapper* is similar to Swift's *simple_mapper*, except it is dynamically typed, so can support both indexed and named members at the same time. *SubMapper* is similar to *structured_regexp_mapper*, as it transforms a list of inputs with a regular expression.

indexing a mapped array causes a new file to be mapped, then the mapper can be implemented for PyDFlow as a class using Python's flexible object model that allows you to implement custom behavior when an object's attributes are accessed, when an object is indexed, or when an object is iterated over. An example of this is the *SimpleMapper* shown in Figure 8.2, where bound variables are created in response to accessing named members such as `mp.left`, or indices such as `mp[0]`.

8.2 Generators

Python has a language feature termed generators, which is a special variant of a coroutine. It is commonly used to implement iterators: an abstraction where elements of a collection can be requested one-by-one. It is particularly useful when the data is stored externally and is large enough that it is undesirable to store the entire data set in memory, or when dealing with large collections with elements that can be computed on-the-fly. The first capability is commonly used in Python

```
def cartesian(set1, set2):
    for x in set1:
        for y in set2:
            yield (x, y)

for x, y in cartesian(xrange(1000), xrange(1000)):
    print x, y
```

Figure 8.3: Python code to iterate through the cartesian product of two lists using a generator.

database interfaces, where one wishes to iterate over the results of a database query without realising the entire result set in memory at once. An example of the second scenario is where one wishes to iterate over cartesian product of two sets. Generating a list of all of the products in memory is clearly an inefficient use of space, rather one wants to generate permutations incrementally. A generator can be used to do this without exposing the details of managing the state, as shown in Figure 8.3 below.

Generators provide a powerful tool that can be combined with PyDFlow to allow clean specification of

One abstraction that we have implemented using generators is the *resultset*. A common scenario that will occur is where we want to run a large number of tasks in parallel, and process the results of these tasks. If the data from the task execution is forwarded to another PyDFlow task, then that task will immediately be able to start execution once the data is available. This is because the semantics of the PyDFlow task graph are inherently parallel and data-driven. However, consider the scenario where we wish to execute a list of independent tasks and then perform some processing (maybe just printing the results) within the Python interpreter. Suppose we just want to print a message to the screen to notify the user when each data item is available.

The most straightforward solution is to spark all of the I-vars, and then simply to call get on each I-var in sequence. However, there is no reason to assume that data will become available in this order, so many notifications will be delayed. The `spark` method on an I-var provides a way to register a callback function that will be called when data appears in the I-var. We can take advantage of this feature to achieve our aim, as illustrated in Figure 8.4 but dealing with asynchronous

```
1  intfile = localfile.subtype()
2
3  @app((intfile), (intfile))
4  def sort(unsorted):
5      return App('sort', '-n', unsorted, '-o', outfiles[0])
6
7  # Map all txt files in current directory
8  unsorted = GlobMapper('*.txt')
9
10 # Create outputs
11 sorted_files = [intfile('sorted_' + file.get()) << sort(file)
12                 for file in unsorted]
13
14 # A callback function which prints a message
15 def callback(done_file):
16     print done_file.get(), 'is ready!'
17
18 # Print a message to the screen when each completes
19 for s in sorted_files:
20     s.spark(done_callback=callback)
21
22 # wait until everything completes, callbacks will occur asynchronously
23 waitall(sorted_files)
```

Figure 8.4: Python code using asynchronous callbacks to handle task completion.

```
1  intfile = localfile.subtype()
2
3  @app((intfile), (intfile))
4  def sort(unsorted):
5      return App('sort', '-n', unsorted, '-o', outfiles[0])
6
7  # Map all txt files in current directory
8  unsorted = GlobMapper('*.txt')
9
10 # Create outputs
11 sorted_files = [intfile('sorted_' + file.get()) << sort(file)
12                 for file in unsorted]
13
14 # Print a message to the screen when each completes
15 for done_file in resultset(sorted_files):
16     print done_file.get(), 'is ready!'
```

Figure 8.5: Python code using a resultset to handle task completion without callbacks.

callbacks adds an unnecessary level of complexity to the solution of a fairly simple task.

The *resultset* abstraction takes advantage of generators to provide an iterator interface that, given a set of I-vars (that may or may not have been forced), iterators over the I-vars in the order in which they complete execution. Figure 8.5 illustrates how the resultset abstraction can be used to solve the same problem without having to reason about asynchronous callbacks.

Note that we can also mix app and function types of tasks in order to achieve the same effect, illustrating the power of the abstractions.

SECTION 9
Conclusions and Future Work

9.1 Conclusion

I have surveyed some of the existing material on task parallelism and described a system, PyDFlow, for expressing task parallel programs in the Python programming language.

The utility of the library is evident from the examples provided: it can specify in a concise and clear way task-parallel computations that are either parallel procedure calls or external command line applications. Moreover, the PyDFlow model permits the definition of new task types without modifying the core framework.

Embedding the system within an interpreted programming language is a severe performance disadvantage for coordination code, but for many applications the speed of execution of the coordination code is less relevant than the ability for the coordination language to allow programmers to express parallel tasks that run on a range of devices.

9.2 Future Work

- **Distributed task graph execution**: In the current system, the task graph evaluation and tracking of dependencies is centralized in a single instance of Python, which will eventually become a bottleneck as we try to scale up the system to large number of tasks, or stress the system by running many short tasks. We want to be able to partition the DAG across different processes (on same computer or across a network), with processes on different machines assuming responsibility for fragments of the task graph by stealing work. We are partway there with the multi-threaded workstealing executor, but the current implementation assumes that the task graph is stored in memory shared between all worker threads. As well as methods for partitioning the task graph, we would need a distributed protocol for tracking and resolving data dependencies

The ability to partition the task graph across multiple machines would also permit the parallel evaluation of Python functions on a single multiprocessor machine, getting around the GIL interpreter and allowing parallel execution of pure Python code.

- **Fault-tolerance**: The larger and longer running a parallel computation, the higher the chance that a component of the system will fail. Future work will involve implementing methods to recover from system component failure. Failure of other components, such as the master Python interpreter or data storage now leads to a failure of the entire execution. We would like to implement recovery mechanisms that support the resumption of crashed jobs and the recovery of lost data.

  Fault tolerance combined with distributed task graph execution will present further challenges.

- **More control flow functions**: We want to support general control flow within the task graph. For example, conditional statements must currently be handled by serial Python code, by partially evaluating a PyDFlow graph, inspecting the result in Python and then further extending the graph. It would be more desirable if conditional statements could be handled within the task graph, with branching decided based on data availability. This would require that the task graph executor support both lazy and eager evaluation of the arguments of a control flow construct, as the condition of a conditional statement must be evaluated before either branch.

- **Streaming data**: It would be desirable to extend the task graph model to support tasks can read and write from stream I-vars, enabling producer and consumer tasks to be run in parallel to implement data processing pipelines.

  These streams could be data streams passed between external tasks, like in Dryad [41] or CIEL[57]. They could also be in-memory streams implemented with Python generators.

- **Additional execution engines**: Adding support for additional executors and data formats would be desirable. For example, we could support a

distributed execution engine like CIEL [57] or Coasters [79], or perhaps distributed file systems like HDFS[78] or CIEL [57].

We could also support more exotic task types, such as GPU computation kernels in CUDA using available Python bindings.

REFERENCES

[1] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 253–262, New York, NY, USA, 2005. ACM.

[2] Kaizar Amin, Gregor von Laszewski, Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, and Albert Rossi. GridAnt: A client-controllable grid workflow system. *Hawaii International Conference on System Sciences*, 7:70210c, 2004.

[3] Gregory R. Andrews, Richard D. Schlichting, Roger Hayes, and Titus D. M. Purdin. The design of the saguaro distributed operating system. *IEEE Trans. Softw. Eng.*, 13:104–118, January 1987.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11:598–632, October 1989.

[5] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the (v , g)-machine. In *FPCA '89: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, New York, NY, USA, 1989. ACM.

[6] Michael Bachle and Paul Kirchberg. Ruby on rails. *IEEE Software*, 24:105–108, 2007.

[7] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGART Bulletin*, pages 55–59, August 1977.

[8] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19:599–609, July 2003.

[9] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30:207–216, August 1995.

[11] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP '11: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 35–46, New York, NY, USA, 2011. ACM.

[12] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM.

[13] K. Chandy and C. Kesselman. Compositional c++: Compositional parallel programming. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1993.

[14] K. M Chandy and Ian Foster. A deterministic notation for cooperating processes. Technical report, California Institute of Technology, Pasadena, CA, USA, 1993.

[15] Marcin Cies̀lik and Cameron Mura. PaPy: Parallel and distributed data-processing pipelines in python. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *SciPy '09: Proceedings of the 8th Python in Science Conference*, pages 41–47, Pasadena, CA, USA, 2009.

[16] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. Research monographs in parallel and distributed computing. Pitman, London, 1989.

[17] NVIDIA Corporation. CUDA Programming Guide 3.2, Oct 2010.

[18] Oak Ridge National Laboratory Distributed Active Archive Center (ORNL DAAC). Modis subsetted land products, collection 5, 2010.

[19] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5:46–55, January 1998.

[20] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3l experiment. In *Proceedings of the 1998 ACM Sigplan Workshop on ML*, pages 31–39, 1998.

[21] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer Berlin / Heidelberg, 1993.

[22] John Darlington, Yike Guo, Hing Wing To, and Jin Yang. Functional skeletons for parallel coordination. In *Euro-Par '95: Proceedings of the First International Euro-Par Conference on Parallel Processing*, pages 55–66, London, UK, 1995. Springer-Verlag.

[23] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Comput. Surv.*, 26:295–336, September 1994.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[25] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In Marios D. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 131–140. Springer Berlin / Heidelberg, 2004.

[26] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[27] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, New York, NY, USA, 2010. ACM.

[28] K. Elkabany, A. Staley, and K. Park. PiCloud - cloud computing for science. simplified. In *SciPy 2010: Proceedings of the 9th Python in Science Conference*, Austin, TX USA, 2010.

[29] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. QUAFF: efficient c++ design for parallel skeletons. *Parallel Computing*, 32:604–615, September 2006.

[30] Stuart I. Feldman. Make: a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[31] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 1 edition, January 2008.

[32] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *DAMP '07: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 37–44, New York, NY, USA, 2007. ACM.

[33] Jeff Forcier, Paul Bissex, and Wesley Chun. *Python Web Development with Django*. Addison-Wesley Professional, 1 edition, 2008.

[34] Ian Foster. Compositional parallel programming languages. *ACM Transactions on Programming Language and Systems*, 18:454–476, July 1996.

[35] Ian Foster. Task parallelism and high-performance languages. In Guy-Ren Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, vol-

ume 1132 of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin / Heidelberg, 1996.

[36] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *PVLDB: Proceedings of the VLDB Endowment*, 2:1414–1425, August 2009.

[37] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, February 1992.

[38] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions of Programming Languages and Systems*, 7:501–538, October 1985.

[39] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[40] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[41] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[42] Joseph C. Jacob, Daniel S. Katz, Thomas Prince, G. Bruce Berriman, John C. Good, and Anastasia C. Laity. The montage architecture for gridenabled science processing of large, distributed datasets. In *Proceedings of the Earth Science Technology Conference*, 2004.

[43] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[44] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(02):127–202, 1992.

[45] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, FMOODS '09/FORTE '09, pages 1–25, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[47] Herbert Kuchen and Murray Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, Mar 2002.

[48] G Kumfert, T Dahlgren, T Epperly, and J) Leek. Babel 1.0 release criteria: A working document. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, USA, 2004.

[49] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[50] Miron Livny, M. Livny, Rajesh Raman, Todd Tannenbaum, Jim Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997.

[51] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain. *SciDAC Review*, 17:30–37, 2010.

[52] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming.

In *InfoScale '06: Proceedings of the 1st International Conference on Scalable Information Systems*, New York, NY, USA, 2006. ACM.

[53] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, December 2005.

[54] E. Mohr, D.A. Kranz, and Jr. R.H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2:264–280, 1991.

[55] L. Moreau, Y. Zhao, I. Foster, J. Voeckler, and M. Wilde. XDTM: The XML Dataset Typing and Mapping for specifying datasets. 2005.

[56] Derek G. Murray and Steven Hand. Scripting the cloud with skywriting. In *HotCloud '10: Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[57] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI 2011: Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation*, page 00, 2011.

[58] Rishiyur S. Nikhil. An overview of the parallel language id (a foundation for ph, a parallel dialect of haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

[59] J.K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23 –30, March 1998.

[60] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[61] Fernando Pèrez and Brian E. Granger. IPython: A system for interactive scientific computing. *Computing in Science and Engineering*, 9:21–29, 2007.

[62] Alan W Powell, Michael J Beckerle, and Stephen M Hanson. Data Format Description Language (DFDL) v1.0 specification. Technical Report DFD-P-R.174, Open Grid Forum, 2011.

[63] Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Choi, Seung-Hee Bae, Hui Li, Bingjing Zhang, Tak-Lon Wu, Yang Ruan, Saliya Ekanayake, Adam Hughes, and Geoffrey Fox. Hybrid cloud and cluster computing paradigms for life science applications. *BMC Bioinformatics*, 11(Suppl 12):S3, 2010.

[64] Guido Rossum. Python reference manual. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1995.

[65] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53:64–71, January 2010.

[66] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun'ichi Tsujii. Design and implementation of GXP Make – a workflow system based on Make. *eScience 2010: Proceedings of the 6th Annual IEEE International Conference on eScience*, 0:214–221, 2010.

[67] Apache Mahout Team. Apache Mahout: Scalable machine-learning and data-mining library, 2011. `http://mahout.apache.org/`.

[68] Celery Team. Celery - the distributed task queue, 2011. `http://celeryproject.org/`.

[69] Condor Team. The directed acyclic graph manager, 2011. `http://www.cs.wisc.edu/condor/dagman`.

[70] PHP Team. Php, 2011. `http://www.php.net`.

[71] Swift Team. Swift user guide, 2011. `http://www.ci.uchicago.edu/swift/guides/userguide.php`.

[72] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8:23–60, January 1998.

[73] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed haskells. *J. Funct. Program.*, 12:469–510, July 2002.

[74] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[75] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28:1709–1732, December 2002.

[76] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18:365–396, December 1986.

[77] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, Third edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2008.

[78] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[79] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *to appear in Parallel Computing*, 2011.

[80] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34:44–49, September 2005.

[81] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, reliable, loosely coupled parallel computation. *Services, IEEE Congress on*, 0:199–206, 2007.