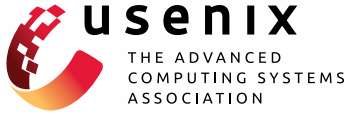

Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud

Stefan C. Müller, ***ETH Zürich and University of Applied Sciences Northwestern Switzerland;***
Gustavo Alonso and Adam Amara, ***ETH Zürich;***
André Csillaghy, ***University of Applied Sciences Northwestern Switzerland***

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muller

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Pydron: semi-automatic parallelization for multi-core and the cloud

Stefan C. Müller[1,3], Gustavo Alonso[1], Adam Amara[2], and André Csillaghy[3]

[1]Department of Computer Science, ETH Zürich
[2]Department of Physics, ETH Zürich
[3]University of Applied Sciences Northwestern Switzerland

Abstract

The cloud, rack-scale computing, and multi-core are the basis for today's computing platforms. Their intrinsic parallelism is a challenge for programmers, specially in areas lacking the necessary economies of scale in application/code reuse because of the small number of potential users and frequently changing code and data. In this paper, based on an on-going collaboration with several projects in astrophysics, we present Pydron, a system to parallelize and execute sequential Python code on a cloud, cluster, or multi-core infrastructure. While focused on scientific applications, the solution we propose is general and provides a competitive alternative to moving the development effort to application specific platforms. Pydron uses semi-automatic parallelization and can parallelize with an API of only two decorators. Pydron also supports the scheduling and run-time management of the parallel code, regardless of the target platform. First experiences with real astrophysics data pipelines indicate Pydron significantly simplifies development without sacrificing the performance gains of parallelism at the machine or cluster level.

1 Introduction

In astronomy and other big-data sciences, the data generated by experiments and simulations is growing by leaps and bounds. Scientists have to use sophisticated computing infrastructures to be able to analyze and process all their observations.

Scientific data is often of a different nature than business data. Data from instruments and simulations has to be heavily processed before conclusions can be drawn from it. This process is repeated many times to calibrate and clean the data and to tune parameters and algorithms. Often this process is exploratory, using non-standard tools and ad-hoc developed code.

For example, in [29] Refregier et al. describe a procedure where repeated executions of a wide-field astronomy image simulator [3] are used to develop and calibrate the simulator, match the simulations with data from real observations, and perform a robustness analysis on the parameter space. With long lasting missions, such as the RHESSI spacecraft, launched in 2002 and still producing data today, changes to processing algorithms, data, and infrastructure happen continuously and will continue throughout the life time of the spacecraft and beyond [31]. This implies a constant correction of the data and the algorithms that adds significant overhead.

One can argue that today there are enough platforms – hardware and software – to support such application scenarios. However, this is far from being the case. The way to achieve performance today is through large scale parallelism: multi-core, rack-scale, or cloud computing. Current approaches to make parallelism available to developers typically provide either low level interfaces to parallel hardware (such as pthreads [7] or MPI [14] which are non trivial to use) or they require a complete integration into frameworks such as Spark [38] or Hadoop [13]. With fast changing code, legacy applications as well as legacy data formats, it is often impractical to apply such frameworks because of their rigid requirements in terms of data formats and algorithmic structure. This is not a question of the adequacy of these systems to the task at hand. It is a question of the total cost of adapting such a framework for the entire life cycle of a scientific mission. Code is often specific to an instrument and to the research of a single group. As a result, the economies of scale that would justify larger development efforts, as required to adopt existing frameworks, are just not there.

In this paper we argue for an alternative approach to separate application specific code from the parallelization framework (language and run-time). Our approach tries to provide maximum flexibility and ease of use for the application developer, with a system that takes care of parallelization, deployment, and scheduling on a variety of target infrastructures.

Our system, Pydron, can semi-automatically parallelize sequential Python code and run the result on multi-core, cluster, or cloud systems. The API consists only of two Python decorators: one to mark functions that should be parallelized, and one to mark functions that are free of side-effects. We use Python because of its wide spread use in the astronomy community. Pydron might even open the door for scientists to start using cloud based systems such as Hadoop and Spark by not requiring them to change their programming habits and not having to deal with the parallel infrastructure used to run the code.

Existing approaches, such as SEJITS [9], have automatically parallelized Python code when the code is restricted in form and data types, or introduced systems, such as CIEL [21], that can scale to multi-machine infrastructures when the application is written in a domain specific language. Pydron combines the advantages of both approaches.

With Pydron, the paper makes the following contributions:

- Pydron allows scientists to work in the language and with the tools they are familiar with, while giving them access to multi-core, cluster and cloud infrastructures.
- We show that the barrier for the application developer to benefit from modern infrastructures can be significantly lowered by using semi-automatic parallelization.
- We demonstrate how a dynamic data-flow graph can be used to counter the limitations of statical analysis when applied to a dynamic language.
- We present a system with three interchangeable elements that can be used to apply the ideas both to other languages and to other execution platforms – both hardware and software.

2 Related Work

Big data in scientific applications has lead to a large variety of systems to make the use of high performance infrastructures simpler for the developer.

Science Data Archives A significant effort has been made to simplify the analysis of scientific data once it has been collected and processed into science-ready products. For data that can be represented in tabular form, e.g. the Sloan Digital Sky Survey [33], databases are typically used. The large data volumes and the sophisticated queries can made specialized extensions to the database system necessary [32]. When data does not easily fit into a relational data model, other approaches are required, such as SciDB [5], a database system that generalizes the relational concept to multidimensional arrays to better support data types such as images or spectra. Many of these extensions are application specific and are rarely used in other contexts.

Delayed Execution Before data can be analyzed, it needs to be processed. Many of the languages currently in widespread use were not designed with parallelization in mind, which leads to a demand for easy-to-use interfaces between the language and the parallel infrastructure. One approach, used by Spark [38], Weaver [6], or FlumeJava [10] is to collect expressions during the execution of the program. Instead of directly executing an operation, an object is returned that represents the not-yet-calculated result. Those objects can then participate in other operations, resulting in an expression graph. The expression graph is then evaluated in parallel.

Such systems typically introduce a set of data types together with operations that can be applied to them. The close control of the system over both data and operators allows for efficient parallelization and sophisticated data management. However, it also forces the developer to formulate the code using only the data formats and operators provided. For the scientific applications we are interested in, this is a significant limitation as those applications often use legacy code and data formats.

Source-to-Source Translation The approach used by SEJITS [9] or Parakeet [30] is to translate the source code into another programming language, such as CUDA [22] or C++, more suited for the targeted infrastructure. The translation and subsequent compilation typically happens just-in-time during execution. The performance improvement comes from a more efficient target language, and / or from parallel execution on hardware such as graphics processor units. These systems place constraints on the code they can translate: Not all data types and operations may be available in the target language, and since Python is dynamically typed, the systems typically require complete type inference. For example, both SEJITS and Parakeet operate only

on NumPy [17] data types and cannot handle other objects. This makes such systems attractive to speed up inner loops, where the amount of translated code is relatively small, and it it easier to comply with the constraints. Thus, parallelization is typically fine grained. Pydron targets infrastructures on which Python is available, and can therefore avoid the constraints that result from a translation into a different language.

Domain Specific Languages When parallelizing at a coarse granular level, near the outer most loops, the distributed tasks will use a significant amount of unmodified application's code. Applying strong constraints is less practical. SWIFT [36], CIEL [21], or PigLatin [24], use custom 'orchestration' languages in which the outermost loops are parallelized.

Parallelization of the actual computations in the inner loops are typically not addressed in those languages. Instead, such systems provide convenient ways to call code written in other languages. The orchestration languages are often functional or have other means to avoid side-effects which would hinder automatic parallelization. Those constraints are less restrictive than in the source-to-source translation approach since they only apply to the outer loops and not to the code called from within. Systems such as Taverna [23] also belong to this category. They use a graphical programming language, in the form of a work-flow graph, to specify the orchestration of the computation. A separate language enforces a strict separation between orchestration and computation. Pydron blurs the barrier between orchestration and computation and avoids the learning curve of an additional language.

Streaming Data streaming systems such as Spark Streaming [39] and Naiad [20] use a data-flow graph representation. Records are passed along the edges, an the vertices represent operations on them. Several of the non-streaming systems also use graph representations internally. In those systems vertices are executed once, and data that flows along the edges typically represent larger units of data (for example sets of records). Data streaming can achieve finer parallelization, on the granularity of individual records, while keeping the graph at a manageable size since there is no need to have a vertex per record.

For use-cases that can be formulated as record streams, such systems can scale well to many nodes. The developer has to provide the implementation of the vertices and, unlike Pydron, also has to provide the structure of the graph, for example with a SQL or LINQ [37] expression.

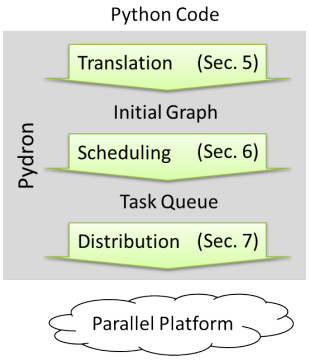

Figure 1: Pydron Overview

Static Compiler Optimization Modern processors support parallelism in various ways, with multiple cores, multiple threads, and vectorized instructions. Compile-time optimizations to make use of those features have been studied extensively [12]. Those approaches are limited by the large search space of such optimizations. As a result such optimizations are typically only applied to relatively simple innermost loops and not at higher levels.

In contrast to these systems, Pydron parallelizes regular Python code, similar to compiler optimizations or source-to-source translation, but it uses a coarser granularity and scales beyond shared memory systems. Existing systems that can scale to such an infrastructure use either domain specific languages or force the developer to formulate the problem in a form dictated by the system. In Pydron, however, the parallelization and execution are separated. As a result, Pydron can easily target either multi-core, clusters, or cloud platforms (or a combination there of). Our approach supports the complete Python language, without the constraints of existing source-to-source translation systems. Since we do not translate the distributed code into a different language, we are not limited to support only those data types and functions from Python libraries that have equivalents in the target language.

3 System Overview

Pydron operates by translating Python into an intermediate data-flow graph representation. The graph is then evaluated by a scheduler sub-system which uses a dis-

tribution sub-system to execute individual tasks to the worker-nodes.

Dynamic-typing, late-binding, and side-effects makes static analysis of Python code hard. We use a dynamic data-flow graph to account for the dynamic nature of Python. This is is a similar approach as used by CIEL [21], and we too use dynamic changes to the graph to handle data dependent control flow, such as loops and branches. We take this idea a step further and continuously refine the data-flow graph to incorporate information about object types and the data itself as it becomes known during execution.

Pydron consists of three components (Figure 1):

The *translator* transforms Python code of those functions decorated with *@schedule* into their initial graph representation. All language constructs can be translated. The translation happens at run-time, the first time the function is invoked. The translation process is described in Section 5.

When a translated function is invoked, the *scheduler* component analyzes the graph to decide in which order the tasks have to be executed and which of them may run in parallel. It fills a queue with tasks that are ready for execution. When the results become known after execution, the scheduler is informed. The scheduler is responsible for making the dynamic graph changes and to add those tasks that have now become ready for execution to the queue. Scheduling is described in Section 6.

The tasks in the queue are distributed to worker nodes for execution. The *distribution system* is responsible to acquire the resources and to start the Python interpreters (typically one per core) which will execute the tasks, as well as to release the resources at the end. It also deploys the the user's application on the worker nodes. We have several back-ends implemented to support cloud, cluster and multi-core infrastructure. The distribution system is described in Section 7.

Pydron has been designed to make these components interchangeable so as to allow extensions to target other languages and execution platforms. The components described in this paper focus on achieving full support for the Python language with greatest flexibility for the developer.

4 Language API: Decorators

To make the system as easy to use as possible, the API of Pydron consists only of two decorators. *@schedule* marks the functions which should be considered for automatic parallelization. This allows the developer to control which parts of the application the system will parallelize. Since the developer marks complete methods with *@schedule*, and not individual loops or statements, this is typically a simple task as most applications will have only a few central functions that orchestrate the process.

The *@functional* decorator informs Pydron that the marked function is free of side-effects and may be run on a different machine. The function has to meet the following criteria:

- No modification of objects passed as arguments.
- Arguments and return values need to support serialization with Python's pickle API [25].
- No assignments to global variables.
- No environment interactions.

The criteria apply only to the observed behavior of the function, not to every operation within its implementation. Especially, the last criterion can be interpreted rather freely as it isn't a technical constraint of Pydron.

Environment interactions are not tracked by our system and could lead to non-deterministic behaviour if executed in parallel. Sometimes this can be acceptable. If, for example, log messages are generated inside a function marked with *@functional*, a non-deterministic order of the log messages should be acceptable.

Another common situation is file IO. Open file handles cannot be passed to marked functions since Pydron has currently not support for remote file operations. Often it is sufficient to track the files by their filenames. If the function only reads files for which it has received the corresponding filename as an argument and returns the name of the files it has written, then the function can typically be safely marked as *@functional*. Pydron will track the file dependencies between the functions by tracking their names, enforcing the correct order of execution. This is especially handy when operating on clusters with a shared file system. In astronomy, many codes already use files to store intermediate data products, making this workaround particularly simple.

We don't currently automatically check if the conditions for the *@functional* decorator hold, even though some automatic checks could be implemented to support the developer in this decision.

5 Language Translation

The intermediate data-flow graph used by Pydron is directional, acyclic, and bipartite. There are two types of nodes: Value-nodes which represent immutable data and tasks which represent operations on data.

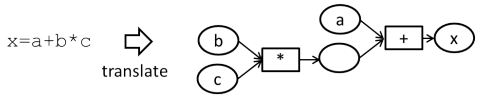

Figure 2: Operator Translation

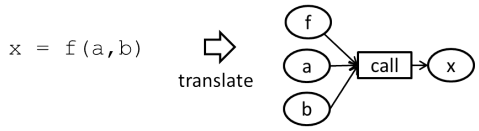

Figure 3: Function Call Translation

Figure 2 shows the data-flow graph for a simple expression. In general, variables become value-nodes. Expressions and statements become tasks. Intermediate values in expressions are also represented by value-nodes. Those have no direct correspondence with a Python variable, but behave no differently otherwise.

Functions with the *@schedule* decorator are translated the first time they are invoked. Pydron uses Python's built-in parser to create an abstract syntax tree of the function's code. This tree is then traversed twice. A first pass identifies the scope of the variables. The actual translation happens in the second pass.

In work-flow systems such as Taverna [23] there are also other type of edges. In Pydron dependencies between tasks can only result from data dependencies.

5.1 Function Calls

Functions are first-class objects in Python. The invoked function is represented by a value-node since the function may itself be the result of an operation. This value-node is an input to a *call*-task, together with the arguments passed to the function. The return value is again a value-node. Figure 3 shows a simple example.

Pydron supports all of Python's language features for function calls, such as keyword arguments and argument lists.

In general, we cannot know at translation time which function is invoked. We have to assume that it may have side-effects or modify arguments passed to it in-place. This leads to additional edges connected to the task (as described in Section 5.3).

To improve the readability of the data-flow graphs in this paper, we show a slightly compacted form. Instead of showing the function calls as in Figure 3, we hide the input for the function object and name the *call* task-node by the function. We will also hide intermediate value-nodes, as shown in Figure 2, from expressions. Instead we directly connect the two tasks.

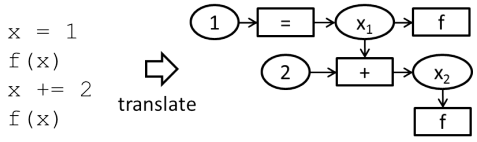

Figure 4: Static Single Assignment Form

5.2 Static Single Assignment Form

Python variables can be reassigned. This conflicts with the property that value-nodes represent immutable data. Therefore a one-to-one relationship between variables and value-nodes is not possible. We translate the Python code into a static single assignment form [11]. A Python variable is represented by a series of value-nodes, each representing the content the variable would hold for a period of the time in a sequential execution of the code.

Figure 4 shows an example. The variable x is assigned twice. The value node x_1 represents the content before the += operator is executed, x_2 represents the content after. Once this operator has been executed, both x_1 and x_2 are known and the scheduler (see Section 6) will be able to schedule both calls to f for parallel execution. We don't show the subscripts explicitly in the other figures as the order can be derived easily from the graph structure.

5.3 In-place modifications

Python objects can change after their creation. This poses a problem since value-nodes represent immutable data. Creating a copy before a modification is impractical since the data of the value-node may not have value semantics.

Pydron uses another solution based on the following observation: The value represented by a value-node becomes known once the producing task has been executed. The value becomes permanently unknown after an in-place modification on the value-node. Conceptually, the value-node still represents the unchanged value. If the task which performs the in-place modification is executed after all other tasks that use that value-node have completed, then the modification will not have an observable effect as the value-node will no longer be needed.

An operation with a potential in-place modification is translated differently. The input edge that connects the task with the affected value-node is flagged as *last-read*. A new value-node to represent the modified value is added as an output, in accordance with the static sin-

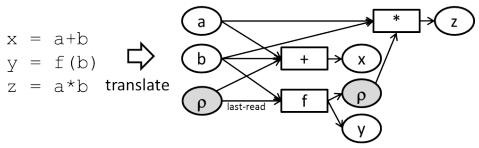

Figure 5: Synchronization point

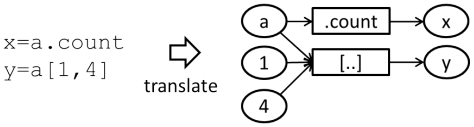

Figure 6: Attributes and Subscripts

gle assignment form (see Section 5.2). The scheduler is aware of the *last-read* flag and ensures correct ordering of task-executions.

In the general case, this is still insufficient to guarantee the same results since the objects changed in-place may be referenced from other value-nodes as well. For each affected value-node, the above special translation would have to be applied. We cannot identify them in general. Therefore, whenever there is a risk of having an in-place modification that may affect other value-nodes, we make the task into a synchronization point.

A pseudo variable is introduced to model synchronization points. This variable is implicitly read by all tasks. A synchronization point is translated as an in-place modification on this variable. The *last-read* edge created by that translation ensures that all previous tasks have to be executed before the synchronization point. All tasks after the synchronization point will use the new value-node, representing the changed pseudo variable, as an input, and will therefore be forced to wait til after the synchronization point.

Synchronization points are also used to model operations with potential side-effects. Such as a call to a function which is not marked with *@functional*. Since the invoked function is not known at translation time, all calls are initially translated as a synchronization point. We will use the adaptive graph refinement to remove those synchronization points for @functional functions.

Figure 5 shows an example. The *last-read* flag forces the addition to complete before f is invoked. The assignment of the pseudo-variable ρ, forces the multiplication to execute after f.

There would be more straight-forward ways to model synchronization points, for example by having ρ as an output of the addition, but this method allows us to reuse the technique of in-place modifications, reducing the overall complexity of the system.

5.4 Attribute and Subscript

When used as a right-hand-side expression, both attributes and subscripts are translated to a task which receives the object as an input. For attribute access, the name of the attribute is stored in the task. In case of a subscript, the indices are also passed as inputs. Pydron supports all indexing constructs, including slicing. Figure 6 shows a simple example with both attribute and subscript used as a right-hand-side expression.

Both attributes and subscripts can be used as a left-hand-side expression as well. Those tasks have the assigned value as an additional input. By its nature, such an assignment is an in-place modification on the object, for which additional edges have to be added to the graph, as described in Section 5.3.

5.5 *@functional* Decorator

Functions decorated with *@functional* are not changed at all. The only effect is that Pydron internally keeps track of those functions.

When the function object on a *call*-task (see Section 5.1) becomes known during the evaluation of the graph, the scheduler checks if the invoked function is *@functional*. If so, the graph is changed to remove the synchronization point.

This usually happens quite early during the evaluation of a graph as most invoked functions will be stored in global variables (Section 5.9) and are not the result of operations.

5.6 Conditional Statements

The translation of the *if* statement makes use of the dynamic data-flow graph. The complete *if* statement is initially translated into a single task. The condition is an input to this task. Both the *body* and the *else* section are translated individually into sub-graphs.

During translation of the *body* and *else* sections, the variables read and assigned are kept track of. They too become inputs and outputs of the *if*-task.

A variable in Python can have an undefined content if it is assigned in only one of the sections. In the data-flow graph each value-node must be the output of a task. For such situations a special task is added to the graph which produces an undefined value as a result. The scheduler is aware of value-nodes with an undefined content and will produce the same exceptions on an attempt to use the

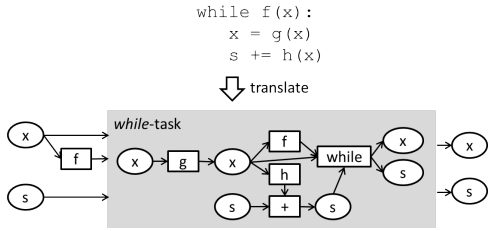

Figure 7: *while* Loop Translation

```
for i in items:
   f(i)
   if i == end:
     break
   g(i)
```

reformulation

```
for i in items until break_flag:
   f(i)
   if i == end:
     break_flag = True
   if not break_flag:
       g(i)
```

Figure 8: Reformulate *break* Statement

value-node as Python would when using an undefined variable.

5.7 Loops

The translation of *for* and *while* loops use the same techniques as for conditional statements. The loop *body* and the optional *else* section are translated into sub-graphs. The complete loop construct is translated initially into a single task. The condition, in case of a *while* loop, or the iterator, in case of a *for* loop, is an input to this task. The expression of a *for*-loop evaluates to an iterable. We insert another task which uses the built-in function *iter()* to get the iterator.

At the end of the body sub-graph the loop task itself is added to form a tail-recursive pattern. This may seem to enforce sequential execution, and indeed it will do so, unless the scheduler finds the requirements met at runtime that allow parallel execution the iterations.

Figure 7 shows an example. The *while* loop is first translated into a single *while*-task, internally storing the sub-graph of its *body*. For every variable read in the body there is a corresponding input, and for every variable assigned there is an output. The sub-graph also contains the inner *while*-task which forms the tail recursion.

5.8 *return*, *break*, and *continue*

The three statements *return*, *break*, and *continue* interrupt the regular control flow. Pydron translates those statements by reformulating them with conditional expressions and flag variables. Figure 8 shows an example. The interrupting statement sets a flag variable. Once such a flag has been set, all code afterwards is put into a condition checking the flag. Since the interrupting statement might be inside nested *if* statements, multiple conditions might be introduced. The task of the loop is aware of the flags and uses them to decide if to replace the task by the *body* sub-graph or if to end the loop, with or without a final replacement with the *else* sub-graph. In case of the return statement, the *return* value is stored together with the flag. This reformulation is performed on the fly during the translation into the data-flow graph.

5.9 Non-local variables

Besides being local to the function, variables can also belong to the module in which the function is defined (global variables). In addition Python allows functions to be defined within functions. Those nested functions may access the variables of the enclosing function (closure variables). Pydron supports both global variables, closures, and nested functions.

Python stores the value of a closure variable in a *cell*-object that lives on the heap. Any access to the variable is transparently transformed into an access to this *cell*-object. We can translate functions containing closure variables by using the same approach as Python: A *read-cell* or *write-cell* task is added to the graph whenever a closure variable is accessed. The task is reading or writing the *cell*-object when executed. This has to be done in both the enclosed and the enclosing function. Pydron identifies the variables that are accessed from enclosed functions in a first pass over the abstract syntax tree.

Any access to global variables can be translated similarly with *read-global* and *write-global* tasks. Assignment to a global variable is considered a side-effect and leads to a synchronization point. Reading a global variable does not.

5.10 Exception Handing

Exception handling statements such as *try-except* or the *with* statement are translated as well.

At first, exceptions seem to forbid any parallelism as every operation could potentially throw an exception. The decision if an operation is to be executed can only be made once the previous operation has finished. We

can still achieve parallelism by using speculative execution [27]. An operation is executed even if it is not clear if an exception in a previous operation may occur. This is possible because when we execute a task without side-effects or in-place modifications and discard its outputs then this has the same observable behavior as if we would not have executed the task at all.

If a task does have side effects, this translates into a synchronization point which forces all previous operations to complete before it. In this situation we know if any of the previous operations raised an exception.

The cost of this method is that we potentially waste significant resources on speculatively executed tasks should an exception occur. If we assume that exceptions are used in rare scenarios and not for regular control flow, then exception handing has little impact on the potential parallelism.

5.11 *yield* Statement

The *yield* statement is special since it transforms the function in which it appears into a generator. When the function is invoked, the execution of the function pauses and an iterator is returned. When elements are consumed from the iterator, the execution proceeds from *yield* to *yield* statement.

The *yield* statement can be translated into a data-flow equivalent which is treated specially by the scheduler. Between reaching a *yield* statement and the next consumption of an element on the iterator, only tasks can be executed which are free of side-effects and perform no in-place modifications. This is similar to exception handing as we cannot say for sure if another element will be consumed by the iterator, making the execution of any operation after a *yield* statement speculative.

6 Scheduling

The scheduler component of Pydron takes as input the graph produced in the translation step and produces as output a continuously updated list of tasks to be executed.

The scheduler becomes active when a function marked with *@schedule* is invoked. It keeps track of the execution progress and enforces the correct order of execution and decides which tasks may run in parallel.

A task is ready for execution if the following conditions are met:

- All its inputs are known.
- The task does not require further changes of the graph.
- For any input with the *last-read* flag (see Section 5.3) all other tasks that share this input have already completed.

This guarantees the correct order of execution. If multiple tasks fulfill those criteria, they may execute in parallel.

All tasks that are identified as ready for execution are placed in a queue. This queue is read by the distribution system (Section 7). The distribution system informs the scheduler once the execution of a task has completed. The outputs of the finished tasks become known, potentially leading to more tasks becoming ready for execution.

The scheduler also uses the information that becomes available during execution for refinement of the data-flow graph. The availability of run-time information allows for various optimizations, of which a small number has already been implemented in Pydron.

6.1 Adaptive Graph Refinement

Some tasks will require changes to the graph. Every time the value of a value-node becomes known the scheduler informs the tasks which have this value-node as an input and allows them to change the graph. There are two kinds of changes made to the graph:

- Removal of a synchronization point (Section 6.2).
- Replacement of the task-node by a sub-graph (Section 6.3).

6.2 Removal of synchronization points

The dynamic nature of Python often doesn't allow to make strong guarantees from the code alone. This forces us to translate the code into a graph with many synchronization points. The most common cause are *call* expressions, since the invoked function is not known at translation time (see section 5.3). Once the called function becomes known, the scheduler can check if it is marked as *@functional*. If so, the synchronization point is removed. The two value-nodes for ρ are merged into one and the *last-read* flag is removed.

In most codes, the functions themselves are not the result of expressions, but are either globally defined or object attributes, therefore most synchronization points can often be removed early in the execution.

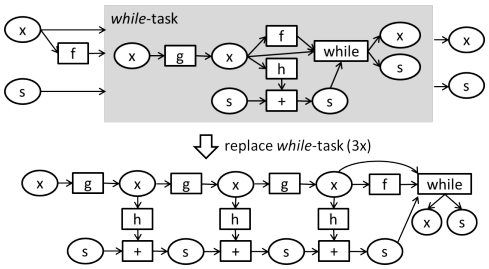

Figure 9: *while* loop parallelization

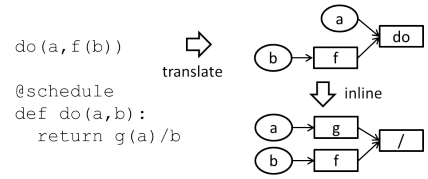

Figure 10: Inline Substitution

6.3 Loops

After the translation, there is a single node in the graph for a loop (section 5.7). Loop tasks are replaced by their sub-graphs during scheduling. According to the loop condition, the scheduler replaces the looop task with either its *body* or *else* sub-graph. Since the *body* sub-graph contains the loop node again, a tail-recurison is formed. At all stages the graph is acyclic which is a different approach as taken by Naiad [20] were loops are modeled with feed-back edges and the control flow is implemented with timestamps on the records passed through the graph.

If the decision for the execution of the next iteration depends on the complete execution of the body, or if the body contains a synchronization point, then the replacement of the tail-recursive tasks must happen after executing all previous tasks, resulting in a sequential execution. But if the condition is known early, as it is often the case with *for*-loops, then the complete loop can be unrolled in a short time.

Figure 9 shows the graph from the *while* loop of Figure 7 after three replacements. This method can still allow for parallelism even if the loop iterations are not completely independent of each other. If *g* executes faster than *h*, the *while* loop will unroll faster than a single *h* executes, allowing for several parallel executions of *h*. Even if *g* is slow, *g* can run in parallel with the *h* call of the previous iteration.

The summation is still executed sequentially, without making any associativity assumptions on the potentially overloaded plus operator.

6.4 Scheduler Relocation

The scheduler can run on the workstation of the user, but it can also be relocated to a remote Python process managed by the distribution system. If the latency between the user's workstation and the remote machines is substantial (such as when executing on a cloud), this will greatly reduce the communication overhead. The data transferred from and to the workstation is reduced to the arguments passed to the *@schedule* function, the data-flow graph of the function, and the return value.

6.5 Inline Substitution

If, during the evaluation of the graph, an invoked function is found to be decorated with *@schedule*, then this function can be translated to a data-flow graph as well.

Instead of invoking the original Python method, the *call*-task is replaced by the function's graph. This corresponds to the inline substitution optimization performed by compilers [11]. Inline substitution can expose additional parallelism as shown in Figure 10. The call to *do* is inline substituted, allowing for parallel execution of *f* and *g*, even though *f* is required to calculate an argument of the call. This works since the substitution can be performed as soon as the invoked method is known, even before the arguments are calculated.

Inline substitution is optional and the scheduler may decide not to inline a call and instead run the original, untranslated, function to control the granularity of the parallelization, depending on the target execution platform.

6.6 Scheduler-local Execution

Some tasks, notably those with side-effects, cannot be distributed safely. Such tasks are executed directly within the thread of the scheduler. Since they enforce a synchronization point, such tasks cannot be executed in parallel anyway.

For some tasks, it might not be worth the effort of sending them to a worker node for parallel execution even though it would be formally possible. For example, multiplying two integers has no side-effects, but the overhead of distributing this task is in no relation to the cost of the operation itself. The scheduler can decide to run such tasks locally.

Pydron currently applies a simple heuristic based on the type of the operation. Since the decision has to be

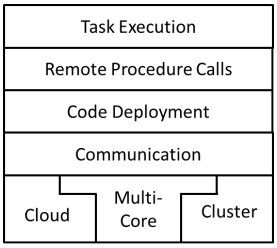

Figure 11: Layers of the distribution system

made only once all the input values to the task are known, quite elaborate techniques could be used, based on operation and data size information, to balance communication and execution cost.

7 Distribution

The distribution system takes tasks that are ready for execution and distributes them to Python run-times, potentially on another machine. It is able to automatically allocate and free resources, so that the user only has to run the application on a local workstation, as one would for regular Python. The system is highly configurable to support different infrastructures. Several configurations can be prepared for the user to choose from.

During operation, the system typically maintains several running Python processes on which it can execute tasks in parallel. The task, and all the inputs, are transferred to the Python process where the task is executed. The outputs are transferred back to the scheduler.

The main effort of the distribution system is to start or acquire the worker nodes, start the Python processes, and establish the means required for remote method invocation. The distribution system uses a layered architecture (Figure 11).

7.1 Worker-Node Acquisition

Before the Python processes can be started, the machines need to be acquired. There are several implementations of this layer. Each provides the means to acquire machines, run a command on them, and release the machines. This API is then used to start a Python process with a small boot-strap script passed to it as an argument.

Multi-core Back-end Python's global interpreter lock makes threads unusable for exploiting multi-core machines, we therefore use multiprocessing instead. This back-end starts processes on the local machine using the *subprocess* module provided by Python.

Cluster Back-end Instead of starting Python locally, a job is submitted to the cluster's job queue, asking for a number of nodes on which the process is run. The job submission is done with a configurable bash script. Pydron can also execute this bash script remotely via an SSH connection as it is often required for clusters with a login-node.

Cloud Back-end The cloud back-end first starts worker nodes. Pydron is using Apache libcloud [1] which supports various commercial cloud providers. Once the instances have started, Pydron opens an SSH connection to each to execute the command. The disk image which is booted can be configured, as can the type of the nodes. The image must contain a Python installation and allow SSH access. Neither Pydron nor the user's application need to be installed on it.

Combining Back-ends The multi-core back-end is often combined with the cluster or cloud back-end to make use of multiple cores on multiple machines.

7.2 Establishing Communication

The boot-strap script establishes communication. Pydron currently supports communication via TCP connections. To mitigate problems caused by firewalls and network address translation, connection attempts are made in both directions. Other methods, such as MPI [14] could also be implemented.

Each node has one communication channel to the workstation from which Pydron was first started. Additional channels for direct communication with other participating nodes are opened on demand, as is needed when the scheduler is relocated (see Section 6.4) to a worker node.

7.3 Code deployment

To execute tasks on remote nodes, the application's code has to be available on the nodes. Manual deployment of the code can be tedious, especially if the nodes do not have access to a shared file system.

Pydron automates this process by using a Python import hook [34] on the worker nodes. When a Python module is imported which is not available on the worker node, the import hook loads the source code from the user's workstation over the established communication channel. Python's internal caching of loaded modules ensures that this has to be done only once per module and Python process.

The code of Pydron itself is also transferred to the worker nodes. This simplifies the deployment of Pydron, as it does not need to be installed on every node. It also avoids potential version compatibility issues as all participants use exactly the same version of Pydron.

7.4 Third-party libraries

The code deployment system also works for most third-party libraries. This further simplifies the deployment of the code on the worker nodes, and makes Pydron more transparent to the user. The exception are libraries that contain native code. Pydron currently does not attempt to transmit native code libraries. In some cases, particularly when the worker-nodes are binary compatible, or when the libraries can be installed via Python package repository, automatic deployment of such libaries could be made possible, but this currently not implemented.

An example of a library which has to be manually deployed is SciPy [17]. This does not prevent us from using SciPy. The primary data type, NumPy arrays, are serializable with pickle. The methods of SciPy which do not change the data in-place can be whitelisted as *@functional*.

7.5 Remote Procedure Call

A simple remote procedure call (RPC) protocol is established on top of the communication channel. It uses Python's *pickle* API to serialize the invoked function, the arguments passed to it, as well as the return value or the raised exception.

7.6 Executing Task-Nodes

RPC connections are established from the node on which the scheduler is running to all other available nodes. The distribution system keeps track of idle and busy workers. Tasks added by the scheduler to the queue of ready tasks are assigned to idle workers. The task is then executed on the node using an RPC call. The result is passed back to the scheduler.

7.7 Fault Tolerance

With increased number of nodes, the probably of a single node failing is greatly increased. The distribution system is in charge of monitoring the Python processes. If a process fails to react, it is taken out of the set of available workers. If it was executing a task, this task is put back to the queue of tasks ready for execution. Since only tasks without side-effects are executed on remote nodes there are no conflicts arising from executing a task twice.

Currently, we follow a simple policy of rescheduling failed tasks. In the future we will explore more complex policies that could take user input into consideration.

8 Discussion

It is the simplicity of use and design that makes Pydron attractive for domains such as astronomy that lack the economy of scale to justify porting efforts to a different language or to other frameworks. The system works without sophisticated language analysis, scheduling, or resource management. Using more advanced implementations for those components will certainly improve the performance further. CIEL [21], in particular, is a system that contains many components from which Pydron could profit.

Pydron shares the architecture with systems such as CIEL and Dryad [16]: An orchestration language is translated into a data-flow graph. The individual tasks, represented by nodes, are typically written in a language such as Java. They are sent to worker nodes for execution. Such systems have the advantage over approaches such as MapReduce [13] in handling iterative computations [21]. In Section 2 we discussed how a separate orchestration language can be a barrier to adopt a solution. In addition, there are also technical consequences. Two separate languages implies two spaces in which objects can reside:

Data Space for the data which is a processed by the individual tasks.

Coordination Space for data that is required to determine the control flow.

CIEL allows data to pass from the data space to the coordination space, by use of a special operator which makes assumptions on the format of the data. This feature allows for data dependent control flow. Pydron goes a step further. By avoiding a separate language, there is only one space where objects reside. Processed data and coordination data can be treated equally, as one would in a regular single-threaded program, reducing the complexity the developer has to handle to profit from parallel infrastructures.

The separation between orchestration code and computation code still exists in Pydron. Functions annotated with *@functional* contain the code executed within a task, functions annotated with *@schedule* mark orchestration code. Only orchestration code is translated into

the data-flow graph. *@functional* code might be sent to a worker node for execution, but there it is executed as regular Python code. The line between the two is less obvious in Pydron, since there is no need to use a different language for orchestration code. In addition, Pydron has the option to execute orchestration code regularly, instead of translating it into a data-flow graph (Section 6.5), further blurring the line.

Both CIEL and Pydron change the data-flow graph based on the data. CIEL allows a task to spawn new tasks during execution. To account for the dynamic nature of Python, Pydron requires additional changes, most notably it those required to remove the synchronization points (Section 5.3), We also allow tasks to trigger changes to the graph before all its inputs are available.

There are a number of features in CIEL that Pydron is currently missing, such as the multiple-queue-based scheduler, fault tolerance for the master node, and streaming. Such features will be integrated into future versions of Pydron.

Fully automated parallelization of sequential languages has been studied in depth which has lead to systems such as Helix [8]. Such systems perform a static analysis on the code to identify loops that can be parallelized. The search space for the inference of the data dependencies includes all code potentially executed within the loop. This effectively limits such approaches to the inner-most loops. Dynamic languages such as Python are particularly difficult in this respect. In consequence, parallelization is fine granular, and small orchestration overheads quickly become the bottleneck. This reflects in the way such systems operate. For example, the parallelization constructs may be directly inserted into the compiled code, instead of evaluating a data-flow graph at run-time.

Pydron parallelizes on a coarse granularity. To keep the search space reasonable, the user has to help out with the *@functional* annotation. The code analysis of Pydron is similar to the data-flow analysis performed by automatic parallelization systems, yet the design is primarliy driven by the need of coping with the dynamic nature of Python. Since Pydron can make decisions at run-time, it can avoid some of the complex problems such as pointer analysis [15]. Such analysis could still be integrated into Pydron in the future as it would allow certain decisions to be made before the actual data is computed.

We don't see Pydron as a replacement for systems such as Helix. In fact, it would be possible to combine both. Combining Pydron with another parallelization system works very well in practice. In section 7.4 we describe how Pydron can be used with SciPy [17]. If SciPy is compiled with multi-threaded ATLAS [35], then the numerical functions would exploit multiple-cores, while Pydron can parallelize the outer loops across several machines.

```
@schedule
def train_forest(data, labels, count):
  forest = []
  for i in range(count):
    tree = train_tree(i, data, labels)
    forest += [tree]

  def predict(sample):
    predictions = [tree.predict(sample)[0]
                   for tree in forest]
    return Counter(predictions).most_common(1)
  return predict
```

Figure 12: Random Forest Implementation

9 Scalability

In this section, we demonstrate the scalability of Pydron for multi-core, cluster and cloud infrastructures. We also provide insights through several experiments on how Pydron operates. All measurements were taken with CPython 2.7.6.

9.1 Multi-core

We use a machine-learning example for the multi-core and cluster measurements. The random forest method [4] trains several decision trees on a random sub-set of the training samples. Predictions are made by majority vote among the predictions made by the individual decision trees. We used 50% of the samples in the MNIST handwritten digits data-set for training [19] (approx. 27 MB).

The code is shown in Figure 12. The *train_forest* function is annotated with *@schedule*. The *for*-loop can be unrolled completely in the beginning of the execution since *train_tree* is annotated with *@functional*. *train_forest* returns a nested function to make predictions, using a closure variable to access the forest. If predictions were expensive, then annotating the nested function with *@schedule* would parallelize the list-comprehension as well. The implementation of train_tree is using scikit-learn from SciPy [17] internally. Pydron handles calls to thrid-party libraries as any other function call (see section 7.4).

Figure 13 shows the learning time on a single machine with 64 cores (AMD Opteron 6276) when running the code using regular Python and when using Pydron

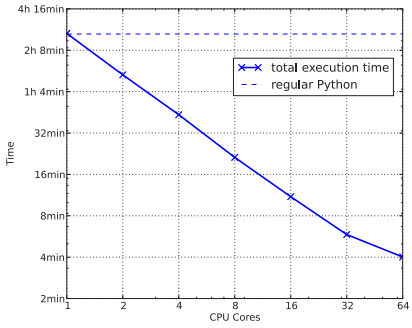

Figure 13: Random Forest Training on Multi-core

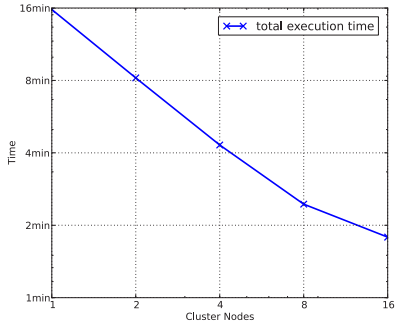

Figure 14: Random Forest Training on Cluster

with an increasing number of cores. It scales nearly linearly and only flattens once the communication overhead becomes noticeable. This is to be expected as Pydron currently makes no use of shared memory for communication. The horizontal line marks the execution time with regular, single threaded Python, which is about 90s (~1%) faster than Pydron with a single core.

9.2 Cluster

Figure 14 shows the result of the same machine-learning code when Pydron is instructed to execute it on a cluster (Intel Xeon L5520). We use the combination of the cluster and the multi-core back-end to utilize the 8 cores of each node. We run the experiment with up to 16 cluster nodes with a total of 128 cores, at which point the scalability starts to degrade due to communication overheads. The execution time of regular, single threaded, Python is not shown in the figure as it would be about eight times slower than a single node. When running Pydron with only one of the node's cores, the difference is comparable to the one shown for the 64-core machine.

```
@schedule
def parameter_sweep(in_file):
  images = []
  basis = []
  for center in np.linspace(0.01, 0.1, 6):
    for edge in np.linspace(0.7, 1.0, 6):
      images+=[create_images(in_file, center, edge)]
      basis+=[create_basis(in_file, center, edge)]
```

Figure 15: Parameter Sweep Code

9.3 Cloud

Running the machine-learning code on the cloud produces results comparable to those on the cluster, we therefore use cloud computing to demonstrate Pydron on a larger astronomy use-case.

PynPoint [2] is a method for detection of planets outside the solar system. The challenge of exo-planet detection lies in the extreme contrast between the bright host star and the faint planet. Optical effects and atmospheric distortions spread the light of the star over an area larger than the orbit of the planet. PynPoint models the point-spread function of the star with a principal component analysis (PCA) to remove the spread-out light from the star, leaving the planet visible in the residue.

We use a real high-contrast imaging data-set of β Pictoris [18] and the massive exo-planet orbiting it. The data set was taken with the Very Large Telescope. The raw data is publicly available from the European Southern Observatory (ESO) archive (Program ID: 084.C-0739(A)). Some data reduction steps [26] have already been applied to the data. The data set consists of 24000 individual exposures, totaling to 3.8 GB.

PynPoint operates in two main phases. In the first phase, the images are prepared and the basis of the PCA are calculated. In the second phase, the modeled point-spread-function of the star is removed from the exposures. The exposures are then rotated to compensate for earth's rotation and aggregated into the final result. The second phase is fast enough to be used interactively by the scientists to study the effect of the method's parameters. However, some parameters affect the first phase which takes about 15 minutes to execute. We have used Pydron to scale the parameter sweep over the two main parameters used in the first phase.

Six values are used for each parameter, resulting in a total of 36 executions. The code is shown in Figure 15. *in_file* contains the path to the input data file in HDF5 format. The two functions *create_images* and *create_basis* are both decorated with *@functional*. Since they are independent of each other, all 72 calls could be run in parallel. The implementation of those methods

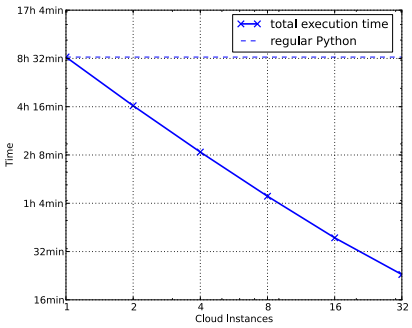

Figure 16: Scalability on Amazon EC2

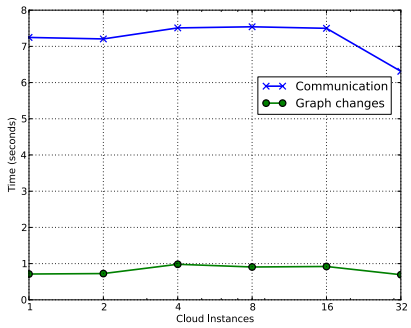

Figure 17: Pydron overhead for communication and dynamic graph changes on Amazon EC2

use numerical routines from SciPy [17] which use multi-threading internally to utilize multiple cores. Pydron can be used together with such libraries. We use Pydron's cloud back-end to parallelize across multiple cloud instances. To get a clearer performance analysis we do not combine it with the multi-core back-end. Thus we use one Python process per instance.

We use Amazon EC2 with *m2.large* instances, with two CPU cores each. Adding cores would not scale well with this workload, as the routines can only profit from the parallel SciPy library for a part of their execution. The cloud instances are connected to a shared file system used as a scratch space. This file system is provided by two separate EC2 instances (*c2.2xlarge*) which provide the storage from a total of four solid state drives. The file system is clustered with glusterFS [28]. The file system initially contains the input data.

Figure 16 shows the execution time for up to 32 instances (64 cores). The execution time includes the time required to start the instances, which takes about one minute.

Other than in the machine-learning use-case, the actual data is transmitted over a shared file system, while Pydron only handles the paths, as described in Section 4. The Pydron induced overhead is therefore very small, about 7s. With a large number of instances the throughput of the scratch file system becomes a bottleneck, as each parameter combination produces approximately 4 GB of data. This bottleneck could be easily addressed by increasing the number of nodes of the clustered file system.

The overhead introduced by Pydron is neglectable in this use-case. The translation of the Python code into the initial data-flow takes five milliseconds. Figure 17 shows that less than a second is spent for all dynamic changes in the data-flow graph and that less than eight seconds are required for communication, including serialization with pickle. Both can partially run in parallel, reducing the effective impact. With 32 instances the workers are limited by the shared file system, the lower CPU utilization speeds up pickle.

10 Conclusions

Semi-automatic parallelization provides easy-to-use access to high performance computing infrastructures for many problems that can be parallelized at a sufficiently coarse granularity.

By putting the focus on non-intrusiveness and a low learning curve, instead of on optimal usage of infrastructure, Pydron can lower the barrier for scientists to access high performance computing infrastructures.

We plan to release Pydron under an open source licence. Please check *www.pydron.org* for availability.

References

[1] apache libcloud, a unified interface to the cloud https://libcloud.apache.org, 2014.

[2] AMARA, A., AND QUANZ, S. P. Pynpoint: an image processing package for finding exoplanets. *Monthly Notices of the Royal Astronomical Society 427*, 2 (2012), 948–955.

[3] BERGÉ, J., GAMPER, L., ET AL. An ultra fast image generator (ufig) for wide-field astronomy. *Astronomy and Computing 1*, 0 (2013), 23 – 32.

[4] BREIMAN, L. Random forests. *Machine Learning 45*, 1 (2001), 5–32.

[5] BROWN, P. G. Overview of sciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 963–968.

[6] BUI, P., YU, L., ET AL. Scripting distributed scientific workflows using weaver. *Concurrency and Computation: Practice and Experience 24*, 15 (2012), 1685–1707.

[7] BUTENHOF, D. R. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[8] CAMPANONI, S., JONES, T., ET AL. Helix: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (2012), CGO '12, ACM, pp. 84–93.

[9] CATANZARO, B., KAMIL, S., ET AL. Sejits: Getting productivity and performance with selective embedded jit specialization. *Programming Models for Emerging Architectures* (2009).

[10] CHAMBERS, C., RANIWALA, A., ET AL. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 363–375.

[11] COOPER, K. D., AND TORCZON, L. *Engineering a compiler*. Morgan Kaufmann [Oxford], San Francisco (Calif.), 2012.

[12] DARTE, A., ROBERT, Y., ET AL. *Scheduling and automatic Parallelization*. Springer, 2000.

[13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004* (2004), pp. 137–150.

[14] FORUM, M. P. I. MPI: A Message-Passing Interface Standard. Version 2.2, Sept. 2009.

[15] HIND, M. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (New York, NY, USA, 2001), PASTE '01, ACM, pp. 54–61.

[16] ISARD, M., AND YU, Y. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 987–994.

[17] JONES, E., OLIPHANT, T., ET AL. SciPy: Open source scientific tools for Python http://www.scipy.org, 2001.

[18] LAGRANGE, A.-M., BONNEFOY, M., ET AL. A giant planet imaged in the disk of the young star beta pictoris. *Science 329*, 5987 (2010), 57–59.

[19] LECUN, Y., AND CORTES, C. The MNIST database of handwritten digits http://yann.lecun.com/exdb/mnist.

[20] MURRAY, D. G., MCSHERRY, F., ET AL. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[21] MURRAY, D. G., SCHWARZKOPF, M., ET AL. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 9–9.

[22] NVIDIA. Nvidia CUDA http://nvidia.com/cuda, 2007.

[23] OINN, T., GREENWOOD, M., ET AL. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience 18*, 10 (Aug. 2006), 1067–1100.

[24] OLSTON, C., REED, B., ET AL. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008), SIGMOD '08, ACM, pp. 1099–1110.

[25] PYTHON SOFTWARE FOUNDATION. pickle – python object serialization http://docs.python.org, 2014.

[26] QUANZ, S. P., KENWORTHY, M. A., ET AL. Searching for gas giant planets on solar system scales: Vlt naco/app observations of the debris disk host stars hd172555 and hd115892. *The Astrophysical Journal Letters 736*, 2 (2011), L32.

[27] RAGHAVAN, P., SHACHNAI, H., AND YANIV, M. Dynamic schemes for speculative execution of code. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on* (Jul 1998), pp. 309–314.

[28] RED HAT INC. GlusterFS http://www.gluster.org, 2013.

[29] REFREGIER, A., AND AMARA, A. A way forward for cosmic shear: Monte-carlo control loops. *Dark Universe Journal* (in press, http://arxiv.org/abs/1303.4739).

[30] RUBINSTEYN, A., HIELSCHER, E., ET AL. Parakeet: A just-in-time parallel accelerator for python. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism* (2012).

[31] STOLTE, E., VON PRAUN, C., ET AL. Scientific data repositories: Designing for a moving target. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 349–360.

[32] SZALAY, A. S., GRAY, J., ET AL. Indexing the sphere with the hierarchical triangular mesh. *CoRR abs/cs/0701164* (2007).

[33] THAKAR, A. R., SZALAY, A. S., ET AL. The catalog archive server database management system. *Computing in Science and Engineering 10*, 1 (2008), 30–37.

[34] VAN ROSSUM, J., AND MOORE, P. New Import Hooks http://legacy.python.org/dev/peps/pep-0302, 2000.

[35] WHALEY, R. C., AND PETITET, A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35*, 2 (February 2005), 101–121.

[36] WILDE, M., HATEGAN, M., ET AL. Swift: A language for distributed parallel scripting. *Parallel Computing 37*, 9 (2011), 633–652.

[37] YU, Y., ISARD, M., , ET AL. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008), vol. 8, pp. 1–14.

[38] ZAHARIA, M., CHOWDHURY, M., ET AL. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.

[39] ZAHARIA, M., DAS, T., ET AL. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 423–438.