CnC-Python: Multicore Programming with High Productivity

Shams Imam and Vivek Sarkar
Rice University
{*shams, vsarkar*}*@rice.edu*

Abstract

We introduce CnC-Python, an implementation of the Concurrent Collections (CnC) programming model for Python computations. Python has been gaining popularity in multiple domains because of its expressiveness and high productivity. However, exploiting multicore parallelism in Python is comparatively tedious since it requires the use of low-level threads or multiprocessing modules. CnC-Python, being implicitly parallel, avoids the use of these low-level constructs, thereby enabling Python programmers to achieve task, data and pipeline parallelism in a declarative fashion while only being required to describe the program as a coordination graph with serial Python code for individual steps. The CnC-Python runtime requires that Python objects communicated between steps be serializable (picklable), but imposes no restriction on the Python idioms used within the serial code. Programs written in CnC-Python are deterministic in that they produce the same outputs for the same inputs regardless of the execution schedules used by the runtime. Our implementation of CnC-Python uses the CnC Habanero-Java (HJ) runtime system, the Babel compiler to generate glue code while invoking Python from Java, and the multiprocessing module available in standard distributions of Python. However, the CnC-Python user need not be aware about Java, Babel, HJ, or any other component in our runtime to use our system. The HJ-based implementation allows us to reuse a mature CnC runtime for scheduling, and enables us to bypass the well publicized performance issues with the Python interpreter's global interpreter lock.

1 Introduction

In recent times, there is a higher emphasis being placed on programmer productivity, especially in the scientific community. Languages like Python and Matlab provide a high productivity development language for many domain experts due to their expressive yet simple syntax and semantics. Such programmers have limited experience with advanced parallel programming concepts such as threads and locks. With the advent of the multicore era, it is clear that improvements in application performance will primarily come from increased parallelism [19]. The domain experts, who are not trained to write parallel programs, are faced with the unappealing task of extracting parallelism from their applications. The challenge then is how to make parallel programming more accessible to such programmers.

In this paper, we introduce a Python-based implementation of Intel's Concurrent Collections (CnC) [10] model which we call CnC-Python. CnC-Python allows domain experts to express their application logic in simple terms using sequential Python code called *steps*. The domain experts also identify control and data dependences in a simple declarative manner. Given these declarative constraints, it is the responsibility of the compiler and a runtime to extract parallelism and performance from the application. CnC-Python programs are also provably deterministic making it easier for programmers to debug their applications. To the best of our knowledge, this is the first implementation of CnC in an imperative language that guarantees isolation of steps with respect to shared data access. In addition, the implementation of CnC-Python lays the foundations to realize the potential of CnC as a coordination language where steps are expressed in a combination of languages such as Python, Matlab, Java, C, C++, and Fortran.

2 Background and Related Work

Python [7] is an interpreted language with support for multiple programming paradigms. Its emphasis on code readability and a comprehensive standard library make it a highly productive language. In recent years, Python has gained in popularity and, as of January 2012, it is rated as the eighth most popular language [9]. Python also has support for extension modules that wrap external C libraries to deliver performance by running compiled native code without compromising productivity.

One popular technique for obtaining parallelism in multicore processors is the use of threads that execute in a shared memory. However, the default implementation of Python uses a global interpreter lock (GIL) that ensures only one bytecode is executed by the interpreter at a time, thereby serializing multithreaded computations [18]. Less popular Python interpreters such as Jython [6] and IronPython [5] do not have the GIL issues that limit multithreading performance, but they do not support many extension modules.

Some C extension modules, such as NumPy [17, 16] and SciPy [14], provide *some* functions that explicitly release the GIL and run multithreaded versions to gain parallel performance; however this is not the norm with most extension modules. Thus, there is no effective way to obtain multithreaded performance from Python by using extension modules. An alternate approach to gain multithreaded performance in Python is by using SEJITS [11] which provides a AST wrapper (called *specializer*) and compiles the application Python code into efficient native code, at runtime. However, this approach is applicable only to existing specializers and not to general Python code.

Another solution to obtaining parallel performance is to launch multiple Python processes and manage communication between them using the multiprocessing module from the standard Python distribution. Another open source module, Parallel Python [22], provides an abstraction over the multiprocessing module. However, both these modules are comparatively low-level; parallelizing simple operations requires explicitly launching processes/jobs and managing synchronization of processes/jobs and data.

Concurrent Collections (CnC) [10] is a declarative and implicitly parallel coordination model that builds on previous work, notably TStreams [15] and Linda [13]. CnC was developed with the intention of making parallel programming accessible to non-expert programmers who provide serial code for computation called steps and a declarative description of data and control dependences. CnC supports combinations of task, data, and pipeline parallelism while retaining determinism. CnC currently supports a wide variety of implementations in languages such as C++ [4], Java [3], Haskell [8] and Scala [1]. CnC-Python is an implementation of the CnC model which uses serial Python code for individual steps. The CnC-Python runtime requires that Python objects communicated between steps be serializable (picklable), but places no restriction on the Python idioms used within the serial step code. Importantly, the programmer is not exposed to *any* concurrency constructs, since they are all abstracted away in the runtime. As discussed in Section 4, our implementation bypasses multithreaded performance issues with the Python interpreter's GIL.

3 CnC-Python Programming Model

CnC relies on *Step Collections*, *Item Collections* and *Control Collections*, each of which are defined statically. At runtime, dynamic instances of members of these collections are generated. Each instance is identified by a *tag*. Step Collections correspond to the different computation steps. Each step instance in the collection accepts a tag as a required input. This input tag is then used to compute the tags for the data items and control tags consumed and produced by the step.

Item Collections correspond to different partitions of the data used by the user's application. Each data item in an item collection is indexed by tags, once an entry is placed in the collection corresponding to a tag it cannot be mutated, thereby ensuring dynamic single assignment. Data items are placed into an item collection using the `put(tag, data)` operation and are retrieved using the `get(tag)` operation. It is an error to attempt duplicate puts with the same tag on the same item collection. Control Collections are responsible for managing the execution of step instances and the control flow. The only operation allowed on a control collection is the `put(tag)` operation causing *prescription* of step instances with the given tag. Like item collections, it is an error to attempt a duplicate put of a tag on a control collection.

If a computational step might read/write data from/to an item collection, then a data dependence exists between the step and the item collection. Similarly all steps prescribed by a control collection for a given tag are static control dependent on the control collection tag. The execution order of steps is constrained only by their data and control dependences. A step is ready to execute as soon as it is prescribed by a control collection, but it may not be able to complete until all its input data dependences are satisfied. As such, synchronization and coordination in CnC applications come from data items and are handled by the runtime system. When two step instances do not share dependences, they can be executed in parallel.

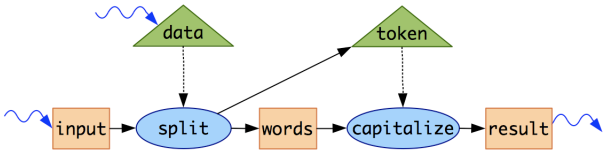

Figure 1: Block diagram representation of the capitalize odd-length words application. Step Collections, Item Collections and Control Collections are represented as ovals, rectangles, and triangles, respectively.

We now show an example of how a user writes an application in CnC-Python. The user needs to identify the computation steps, the data dependences between these steps, and control flow among the steps. Consider the example where we capitalize all odd length words from input strings (example adapted from [10]). The user identifies computation steps and item collections as shown in Figure 1. The two steps `split` the string into words and then `capitalize` the words. The control collection, called `token`, prescribes the `capitalize` steps.

The `split` step controls which words get capitalized by selectively writing tags to `token`. The squiggly arrows represent interactions with the *environment*, the environment produces and consumes data items and tags [10]. This allows the user to feed inputs to the graph and read results from the graph when the computation ends.

Next, the user needs to write a textual representation of the graph. The textual representation includes declaration of the item and control collections, the rules for prescribing step instances from control collections, and data dependence relations of steps and the environment with item and control collections. Figure 2 shows an example of such textual representation of the CnC graph of Figure 1.

```
// declarations, including data types
<string data>;
<string token>;
[string->string input];
[string->string words];
[string->string result];
// prescriptions
<data> :: (split);
<token> :: (capitalize);
// program inputs and outputs
env -> [input], <data>;
[result] -> env;
// producer/consumer relations
[input] -> (split) -> <token>, [words];
[words] -> (capitalize) -> [result];
```

Figure 2: Textual representation of the CnC graph of Figure 1. Declarations identify the Python types of the tags and data items in different collections. For example, the `input` item collections maps `string` tags to `string` data items. Arrows (`->`) are used to represent producer/consumer relations while double colons (`::`) are used to represent prescription relations.

Once the textual description of the graph is ready, it can be submitted to the CnC-Python translator. The translator uses the graph and the runtime API to generate code that initialize the data structures for the item and control collections. In addition, template code with the signatures for the step functions is also generated. Further details on the generated code will be explained in Section 4. When the translator completes, the user needs to fill in initialization logic (`Application`) and the step computation code (`SplitStep` and `CapitalizeStep`) as shown in Figure 3. The step code is simple sequential code in Python and there are no concurrency constructs exposed to the user.

4 CnC-Python Implementation

Our implementation of CnC-Python is currently targeted to shared-memory multiprocessors. The implementation includes a translator that generates supporting code from the textual description of a CnC graph and a runtime to

```
class Application:
  @staticmethod def onStart(args, input, data):
    # env produces inputs
    input.put("first", "i want odd length words")
    data.put("first")
  @staticmethod def onEnd(res):
    # env reads results
    res.printContents()

class SplitStep:
  @staticmethod
  def declareDeps(dependences, tag, inInput):
    dependences.add(inInput, tag)
  @staticmethod
  def compute(tag, inInput, ctrlToken, outWords):
    inStrVal = inInput.get(tag)
    wordList = inStrVal.split()
    for i in range(len(wordList)):
      loopTag = tag + "_" + str(i)
      outWords.put(loopTag, wordList[i])
      if len(wordList[i]) % 2 == 1:
        ctrlToken.put(loopTag)

# class CapitalizeStep not shown
```

Figure 3: CnC-Python program for the CnC graph from Figure 2. All the function signatures are generated by the translator, the user only fills in the function bodies using regular Python.

execute the computation steps in parallel when dependences allow. The motivation for the current design is to allow the CnC runtime to behave as a coordination layer, and to enable (in the future) the user to express computation steps in any combination of languages such as Python, Matlab, Java, C, C++, and Fortran. Figure 4 summarizes the design of our runtime and shows how the generated code is used by the runtime while executing the application. The main components of the implementation are a layer to handle concurrency in Habanero-Java (HJ) [12], a language interoperability layer using Babel [21], and a layer in Python to actually execute the computation steps in parallel. The CnC-Python user need not know anything about HJ or Babel to use our system.

Core CnC Runtime HJ is a parallel programming extension of Java which implements a variant of the Fork/Join Model called the Async/Finish Model to support lightweight dynamic task creation and termination. HJ also supports Data-Driven Tasks (DDTs) [20], an extension to futures inspired by dataflow programming models, to allow efficient expression of arbitrary task graph structures. The HJ task manager supervises the scheduling and execution of tasks in an internal thread pool. Implementing the core CnC runtime in HJ allows us to avoid serializing of concurrent operations by the Python interpreter (as explained in Section 2). Our CnC runtime implementation uses DDTs to wrap execution of prescribed step instances inside lightweight tasks.

In CnC, control collections determine which steps are executed. Control collections are application specific

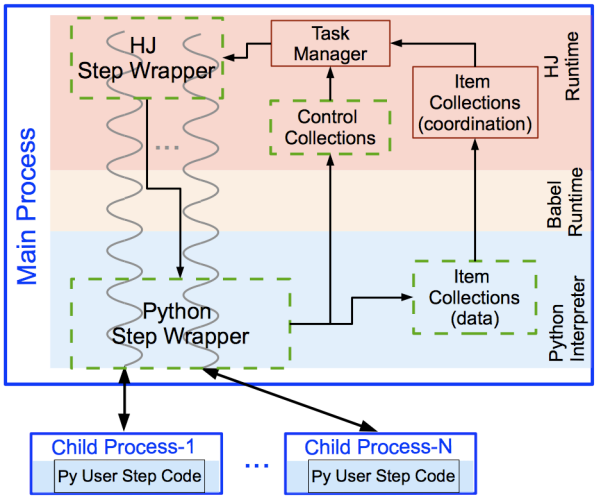

Figure 4: The CnC-Python Runtime implementation highlighting the component runtime boundaries. Fragments in dashed boxes are generated by the translator. It also displays the one-to-one mapping between threads and child processes.

and hence need to be generated by the translator. When a `put` operation is performed on a control collection, the runtime identifies the data dependences for each prescribed step. These dependences are used to create DDTs whose execution is delayed until **all** their data dependences are available. Thus, DDTs enforce control synchronization and coordination in CnC applications by delaying execution of the tasks. Since they are operated on concurrently, item collections are implemented using instances of `ConcurrentHashMap` [2]. When a data item is `put` into an item collection, relevant awaiting tasks are notified that the data item is available. Note that the item collections in the HJ layer only store information on whether a data item is available, the actual data item is stored in an accompanying data structure managed by the Python interpreter.

Language Interoperability We require a language interoperability layer to invoke steps written in Python from the core CnC runtime implemented in HJ. We decided to use Babel because it focuses on high-performance in-process language interoperability within a single address space and is already widely-used by the scientific community [21]. In addition, it supports full bidirectional interoperability among Java, C, C++, Fortran, and Python code. In our implementation of CnC-Python, Babel is used to generate glue code to enable invoking Python steps from Java (HJ) and allow item collections from Python to signal the HJ runtime when data items become available. Since crossing language boundaries involves argument and result conversion which can hurt performance, we restrict ourselves to use only string data as arguments and return types when making inter-language calls. This places the trivial restriction of CnC tags being only strings in our implementation (strings are general enough to represent tags in any CnC application). Similarly, we use a string representation to return the data dependences from the Python step while creating DDTs. Note that no such restrictions apply on the data items since they do not cross language boundaries.

Python Layer Babel manages the initialization of a single Python interpreter when the application starts and correctly handles parallel invocations into the interpreter. Our implementation includes a custom implementation of a concurrent map (in Python) to store the data items in item collections. As the standard implementation of Python prefers spawning processes over use of threads to achieve true parallelism (Section 2), the user steps execute in *child* processes spawned by our runtime. Both Java and Python threading models use native threads and we are guaranteed to be executing on the same thread even after inter-language calls. This allows us to maintain a one-to-one mapping between the main process threads and the spawned child processes. When the main Python interpreter needs to execute a user step, it prepares a meta object containing information on which step to run and all the data used by that step instance (available from the `declareDeps()` function). The interpreter then passes this meta object along to the child process via serialization. The child process executes the corresponding step code and serializes back the results, as shown in Figure 5. Executing the user steps in a separate address space prohibits the step computation from mutating its input data thus making our implementation of CnC capable of ensuring determinism with respect to shared data accesses.

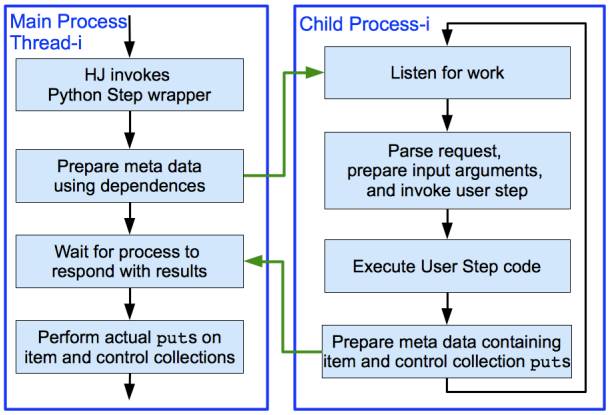

Figure 5: Steps involved in executing a user step computation (pun intended). Threads do not hold or compete for the GIL when waiting for a response from the child process.

5 Preliminary Experimental Results

We now study two main facets of our implementation: the runtime overhead introduced by the multi-language implementation and scalability while serializing data. We evaluate our runtime overhead by comparing with Parallel Python (PP) [22] (PP is implemented in pure Python). We examine scalability using an implementation of Cholesky [10] which involves (de)serialization of NumPy arrays and handling of interdependences between steps. Programmer productivity is achieved since the user writes unrestricted Python code for her step computations and a simple textual description of the CnC graph without being exposed to any concurrency constructs.

The applications were run on nodes which have two quad-core Intel Xeon processors running at 2.83 GHz. Each processor can access up to 16 GB of RAM. It also included a Sun Hotspot JDK 1.7, Habanero-Java 1.2.0, Python 2.7.2, and Babel version 2.0.0-rc7437M. Each configuration of each application was run five times and the minimum execution time of each configuration is reported.

# of Workers	2	4	6	8
Parallel Python	125.63	63.61	42.24	32.19
CnC-Python	33.86	11.54	6.88	5.41
Speedup Factor	3.71	5.51	6.14	5.95

Table 1: Comparing the CnC-Python and Parallel Python (v1.6.1) runtimes for the (embarrassingly parallel) SumPrimes application using inputs of 101000, 102000, ..., 299000. Execution times are reported in seconds.

In PP, the user has to explicitly manage the data dependences for the computations and handle coordination of executing computations. In CnC-Python such dependencies are handled transparently as they can be deduced from the CnC graph description. Both PP and CnC-Python spawn processes to act as workers and manage serializing/deserializing the data and results. Table 1 shows that the CnC-Python implementation is much faster than PP on an application called SumPrimes (from the PP distribution). PP is considerably slower since the data synchronization is implemented using Python locks and the runtime is effectively single threaded (due to the GIL). The CnC-Python runtime overcomes this limitation by handling concurrency issues at the HJ layer where concurrent multithreaded execution is possible.

Next, we analyze the cost of managing dependences and serializing data on a dense linear algebra application that uses NumPy arrays - Cholesky Decomposition. Table 2 shows the scalability achieved while running Cholesky for an array size of 1000×1000 with varying tile sizes. The tiled Cholesky algorithm consists of three steps: the conventional sequential Cholesky, triangular solve, and the symmetric rank-k update [10]. These steps can be overlapped with one another after initial factorization of a single block, resulting in both task and pipeline parallelism. Increasing the tile sizes increases the cost of serialization, but also increases the computation done per step and doesn't adversely affect the execution time (as the variance is low). In general, the application scales for the different tile sizes shown as there is enough parallelism and the amount of computation performed in the steps outweighs the cost of serializing data between processes. In Cholesky, further increasing the tile sizes would lead to loss in scaling due to the reduced parallelism (fewer independent step instances) and not due to serialization costs.

# of Workers	Tile Size 25	50	100	125	σ
2	230.76	244.71	238.44	237.17	5.72
4	88.36	82.66	82.08	83.28	2.88
6	54.74	50.47	51.61	54.02	2.00
8	39.97	36.51	39.53	42.88	2.60

Table 2: Standard deviation of execution times for Cholesky Decomposition of a 1000×1000 array with varying tile sizes. Executions times are reported in seconds and include the time for initializing the input array from disk serially.

6 Conclusion and Future Work

We introduced CnC-Python, an implementation of the Concurrent Collections (CnC) programming model for Python computations. CnC-Python allows Python programmers to achieve task, data and pipeline parallelism in a declarative fashion while only being required to describe the program as a coordination graph with serial Python code for individual steps. Our runtime separates the concurrency and coordination issues into separate layers implementing concurrency issues in a multithreaded runtime. We have shown such an implementation to be more efficient than a pure Python implementation. We plan to extend our implementation to add support for other productive languages such as Matlab allowing the user to write steps in any (supported) language of their choice. Another feature we wish to explore is minimizing the cost of data serialization by introducing extensions that allow the programmer to express data locality hints in CnC programs.

Acknowledgments We gratefully acknowledge support from an Intel award during 2011-2012. This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by NSF Expeditions in Computing Award CCF-0926127.

Availability Further documentation on CnC-Python and a distribution is available at http://cnc-python.rice.edu/.

References

[1] CnC-Scala. http://cnc-scala.rice.edu/.

[2] ConcurrentHashMap (Java Platform SE 7). http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html.

[3] Habanero Concurrent Collections Project. http://habanero.rice.edu/cnc.html.

[4] Intel Concurrent Collections for C++ 0.6 for Windows and Linux. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[5] IronPython - the Python programming language for the .NET Framework. http://www.ironpython.net/.

[6] Jython: Python for the Java Platform. http://www.jython.org/.

[7] Python Programming Language. http://python.org/.

[8] The haskell-cnc package. http://hackage.haskell.org/package/haskell-cnc.

[9] TIOBE Programming Community Index for January 2012. http://www.tiobe.com/index.php/content/paperinfo/tpci/.

[10] BUDIMLIĆ, Z., BURKE, M., CAVÉ, V., KNOBE, K., LOWNEY, G., NEWTON, R., PALSBERG, J., PEIXOTTO, D., SARKAR, V., SCHLIMBACH, F., AND TAŞIRLAR, S. Concurrent Collections. *Scientific Programming 18* (August 2010), 203–217.

[11] CATANZARO, BRYAN AND KAMIL, SHOAIB ASHRAF AND LEE, YUNSUP AND ASANOVI, KRSTE AND DEMMEL, JAMES AND KEUTZER, KURT AND SHALF, JOHN AND YELICK, KATHERINE A. AND FOX, ARMANDO. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Tech. Rep. UCB/EECS-2010-23, EECS Department, University of California, Berkeley, March 2010.

[12] CAVÈ, V., ZHAO, J., GUO, Y., AND SARKAR, V. Habanero-Java: the New Adventures of Old X10. *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)* (August 2011).

[13] GELERNTER, DAVID. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7* (January 1985), 80–112.

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

[15] KNOBE, KATHLEEN AND OFFNER, CARL D. Tstreams: A model of parallel computation (preliminary report). Tech. Rep. HPL-2004-78R1, HP Labs, July 2005.

[16] OLIPHANT, TRAVIS E. Guide to NumPy. http://numpy.scipy.org/numpybook.pdf.

[17] OLIPHANT, TRAVIS E. Python for Scientific Computing. *Computing in Science and Engineering 9*, 3 (2007), 10–20.

[18] PYTHONINFO WIKI. Global Interpreter Lock. http://wiki.python.org/moin/GlobalInterpreterLock.

[19] SUTTER, H., AND LARUS, J. Software and the Concurrency Revolution. *Queue 3*, 7 (September 2005), 54–62.

[20] TAŞIRLAR, S., AND SARKAR, V. Data-Driven Tasks and their Implementation. In *Proceedings of the International Conference on Parallel Processing (ICPP) 2011* (September 2011).

[21] THOMAS G. W. EPPERLY AND GARY KUMFERT AND TAMARA DAHLGREN AND DIETMAR EBNER AND JIM LEEK AND ADRIAN PRANTL AND SCOTT KOHN. High-performance language interoperability for scientific computing through Babel. *IJHPCA*, 1094342011414036 (2011).

[22] VITALII VANOVSCHI. Parallel Python. http://www.parallelpython.com/.