DECO: Polishing Python Parallel Programming

Alex Sherman, Peter Den Hartog
University of Wisconsin - Madison
asherman@cs.wisc.edu, denhartog@wisc.edu

May 2016

Abstract

Modern computer hardware is becoming increasingly parallel, forcing programmers to develop parallel programs in order to achieve best performance. Parallel programming is difficult, especially for scientific programmers whose programs now suffer performance penalties for their lack of parallelism. We propose a simplification of traditionally parallel programming techniques that minimizes programmer interaction and does not require a knowledge of parallel programming. Our solution consists of two Python decorators, and typically requires only two lines of changes in order to parallelize existing serial programs.

1 Introduction

Utilizing parallel hardware has never been more important, and will only become more important. Unfortunately, in the case of many scientific programs, there is rampant under utilization of CPU resources due to a lack of parallel programming. This is not a result of the programs being serial in nature, but rather a lack of parallel programming models suited to scientific programmers [5]. Existing programming models do not address some of the important issues that should be considered in the case of scientific computing. Specifically, a parallel programming model for scientific programmers should be simple to use even for programmers with little knowledge in the area of parallel computing. So far few models meet this criteria, and the ones that do still have room for improvement. For this reason we propose our own solution, a minimal interface allowing a wider audience to better utilize the parallel CPU resources their programs could benefit from.

2 Related Work

Our work is modeled closely after an existing library called Pydron, and both libraries share some similarities with other existing models including OpenMP and Star-P. The main similarity between libraries in this class of parallel programming models is that they provide an API where the programmer is tasked only with identifying sections of code that can be run in parallel instead of synchronizing specific resources. This is a powerful abstraction, but depending on its implementation may come at the cost of some major restrictions.

OpenMP relies on compiler directives as a way for the programmer to denote parallel sections of code [3]. This limits implementations to compiled languages like C/C++ which we find to be an unacceptable restriction when interpreted languages like Python are becoming more common in the scientific community. Additionally OpenMP relies on shared memory, further limiting its scope to operating on a single machine rather than a distributed system unless major modifications to run on a distributed shared memory platform, such as EDSM [2], are made.

Pydron and Star-P both target problems which are embarrassingly parallel (as does DECO). Star-P is an interactive Matlab environment which automatically parallelizes the

```
@concurrent    #Identify the concurrent function
def do_work(key, data):
  data[key] = some_calculations(...)

data = {}
def run():
  for key in data:
    do_work(key, data)
  do_work.wait() # wait for workers to finish and synchronize mutations
```

Figure 1: Psuedo-code for example use of the concurrent decorator without automatic synchronization.

execution of side-effect free functions [1]. One of the goals for DECO was to avoid Star-P's requirement for explicit calls before and after a parallelized call to transfer data to and from its workers. Pydron operates in the same way but avoids extra function calls around the concurrent function by making use of Python decorators [4], which we have incorporated into DECO as well.

Pydron comes much closer to an appropriate solution for scientific computing applications. In Pydron, a programmer must mark functions that are free of side-effects and additionally mark a function whose body they wish to have parallelized. The parallelized function will be inspected for any calls to side-effect free functions and those will be executed in parallel. Pydron supports many forms of parallelization including multi-processing and cloud computing, removing many of the restrictions that would be encountered in OpenMP. However this model only works if the programmer has written side effect free functions, meaning functions which do not mutate any program state and only return a value. We find this to be overly restrictive as many scientific programs operate by iteratively modifying small sections of a larger data structure. Programs like these are difficult to parallelize in Pydron.

3 Decorated Concurrency

We propose an alternative solution to these existing concurrent programming techniques that we call decorated concurrency or DECO. This model very closely resembles Pydron's model of decorating functions we wish to execute in parallel and functions which call the parallel function, but it differs by altering some of the restrictions Pydron imposes. Rather than requiring the concurrent function to be purely functional, DECO allows the concurrent function to mutate arguments and global variables. DECO works by synchronizing these mutations from worker processes back to the main thread in a deterministic way. However, DECO does impose one important restriction on the program: all mutations may only by index based. This restriction allows DECO to require no explicit synchronization, instead inferring it and automatically inserting synchronization into the program.

4 Using DECO

In the general case, adapting a program for use with DECO would mean finding a worker function that is being called in the body of a loop and is working on independent sections of data in each call. For example, a function interpolating and transforming a given latitude range of satellite data with each call and storing it under a new key in a dictionary would be an ideal case for speedup. The function is then

```
@concurrent    #Identify the concurrent function
def do_work(key):
  return some_calculations(...)

data = {}
@synchronized
def run():
  for key in data:
    data[key] = do_work(key)
  print data # data will be automatically synchronized here
```

Figure 2: Example of concurrent result assignment using DECO. do_work will be run in parallel and the results assigned to data[key] at synchronization time. Also note the lack of a call to .wait(), which the @synchronized decorator handles automatically.

decorated with the @concurrent decorator, and a call to function.wait() is added after the loop body, as shown in Figure 1. As long as calls to the function do not require results from previous calls of the function, nothing else is needed to have work dispatched to multiple processes and run in parallel.

4.1 Argument Proxying

DECO operates by replacing the decorated concurrent function with a function which instead passes its arguments to work threads. When a mutable object is passed as an argument to the concurrent function DECO replaces it with a proxy object that appears, when read, the same as the original object but, when written to, records mutation operations. Mutations in this paper will refer only to square bracket operations or get and set operations. Typical mutable objects include lists and dictionaries but DECO also supports any object that implements __getitem__ and __setitem__. Any mutation operations in worker processes are synchronized back to the main process which then applies them in a deterministic order; operations resulting from a latter call to the concurrent function are the latest to be applied. This process is illustrated in Figure 3. This mutation synchronization happens only after a synchronization event which the programmer can insert manually but generally will be automatically inserted by DECO.

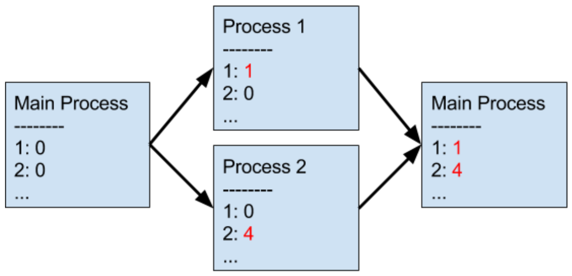

Figure 3: Illustration of mutation synchronization

4.2 Free Variable Proxying

In Python, module globals are created at module load time and mutations to them would not be reflected in the worker processes created by DECO. To enable use and mutation of global variables in worker processes, we parse and inspect the function body AST for free variables at time of decoration. Global values used in the function are then wrapped in proxy objects at call time, synchronizing their current values and propagating back any mutations that take place in the concurrent function. While this is an interesting feature, we hope programmers generally avoid mutating global variables.

```
#Original AST
Assign(Subscript(target, index), Call(concurrent, args))

#Modified AST
Call(concurrent.assign, (target, index) + args)
```

Figure 4: Illustration of modifications DECO makes to @synchronized functions, replacing assignments of the result of concurrent functions with calls to concurrent.assign which allows assignments to be performed at synchronization events

4.3 Automatic Synchronization

Avoiding explicit synchronization is an important goal of DECO because it otherwise requires programmers to have some experience with concurrent programming. In Figure 1 synchronization happens explicitly with a call to wait(), but in Figure 2 synchronization is inserted automatically before any references to data mutated by the concurrent function. These insertions are made by the @synchronized decorator, which traverses the synchronized function's AST recursively tracking potentially mutated data. When it encounters a later reference to one of the mutated data it first inserts calls to wait(). This modified function is then recompiled and replaces the original decorated synchronize function when it's called in a user program.

4.4 Assignment of Concurrent Results

Typically when writing serial programs, programmers are accustomed to directly accessing and assigning the return value of their functions. Migrating to multiprocessing in Python means refactoring these assignments to happen after the results have been produced. In DECO, if the @synchronized decorator is used, this rewrite will happen automatically in certain cases. If the assignments being made are to an indexed object, like lists or dicts, DECO will keep track of which results should go where and automatically perform the assignments when synchronization occurs, illustrated in Figure 2. This is made possible by rewriting the @synchronized function, replacing assignments of concurrent function results to a different call to the concurrent function allowing it to record the indexed object and the index as shown in Figure 4. This feature allows our approach to be applied to even more cases of serial programs without requiring a rewrite.

4.5 Index Based Mutation

We believe that the DECO programming model very closely mimics simpler serial programming and that this is due almost entirely to the limitations we impose for index based mutation only. Every operation that is synchronized is effectively a four-tuple of data containing a time stamp, a reference to the mutable object, the index being assigned to, and the value being assigned to it. This is important because it enforces thread safety, allowing us to remove most explicit synchronization. Because synchronization happens only at certain points, each worker process reads and writes to an effectively frozen version of otherwise mutable objects, eliminating race conditions. With that limitation in mind, a scientific programmer has a defined programming model to follow, giving them a clear goal when writing concurrent programs with DECO. Index based mutation limits programmers enough to provide safe parallel computing, but not too much to limit the utility of the resulting code.

5 Results

Because DECO uses Python's multiprocessing pool to parallelize operations, it performs al-

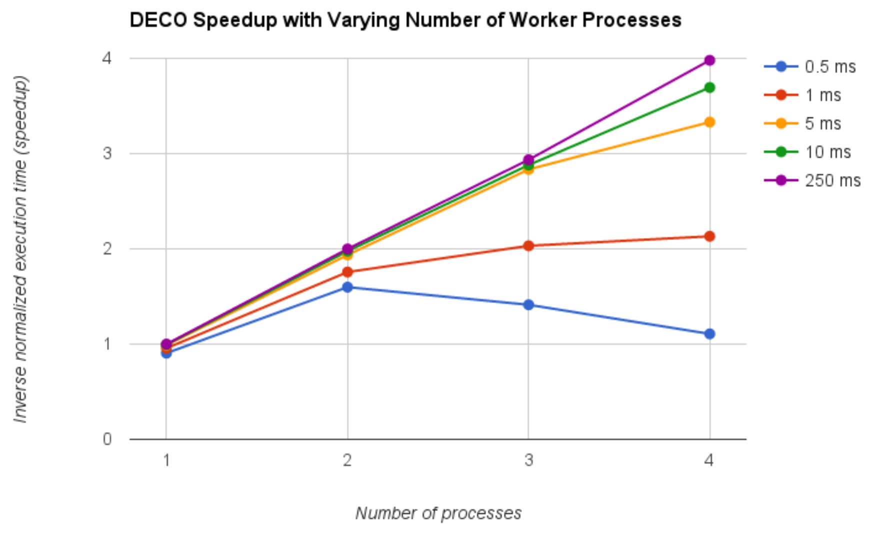

Figure 5: Benchmark timing results, lines represent separate concurrent function execution durations. The benchmark is a best case scenario, it simply calls time.sleep() so it produces no interference between cores and minimizes IPC overhead.

most identically to a program that was parallelized manually using a pool. The pool API is a little obtuse to begin with, as such we consider our simplification of the API a good result on its own. Adding to that the flexibility of mutating function arguments inside the worker processes and automatic assignment rewriting, we believe DECO is a great improvement over the built in Python multiprocessing library. The simplification of the API comes at no noticeable difference in performance which makes DECO a great alternative to an already great multiprocessing library.

5.1 IPC Overhead and Limitations

We find the inter process communication (IPC) overhead to be very low in multiprocessing.pool and consider it nearly ideal in terms of speedup when converting a serial program to parallel. However, pool is not a panacea so neither is DECO, there is a limit as to what functions will benefit from our approach. We benchmarked DECO with an example MD5 hashing workload and compared it to synchronous function calls running in standard single threaded Python. We varied the number of hashes computed in each call to the worker function to produce different lengths of execution for each call. As shown in Figure 6, our benchmarks suggest that DECO's multiprocessing model begins to overcome overhead limitations when the concurrent function takes longer than 1 millisecond to execute. Figure 5 also demonstrates that the effect of overhead becomes increasingly severe as the number of workers increases. This number will vary between machines and applications, but we consider 1 millisecond to be a reasonable lower bound on the execution length of targetable concurrent functions.

6 Future Work

Improvements could be made to the argument proxying and mutation synchronization technique as it currently is implemented in DECO. Presently mutations are only detected at top level arguments, meaning that mutations of elements of an argument will not be synchronized. We could improve our argument proxying in order to avoid this limitation. When an ele-

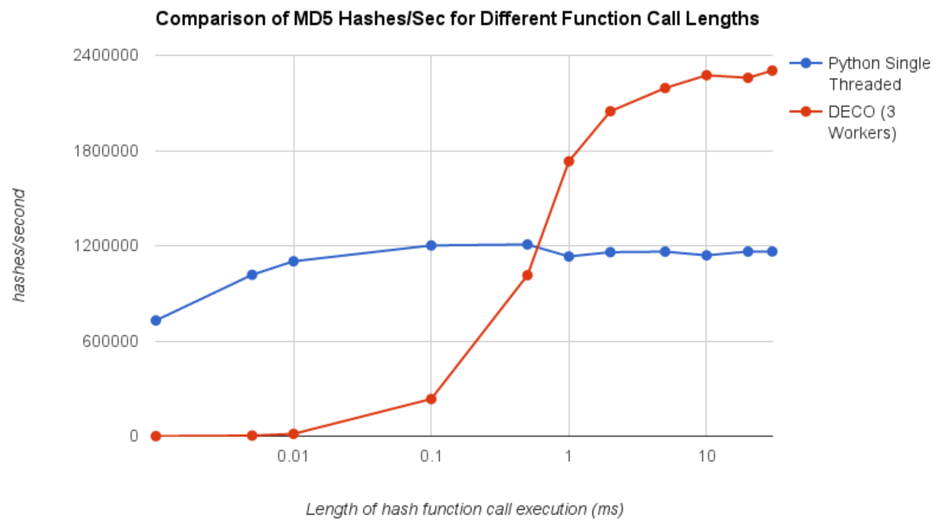

Figure 6: Benchmark results showing the crossover point at which DECO multiprocess overhead becomes negligible compared to the performance gains of parallelization.

ment is read from a proxy, rather than returning the raw value, DECO could return a proxy of the element. This proxying of argument elements could happen recursively, and in this way DECO could synchronize all forms of argument mutation.

6.1 Learning From Pydron

While investigating Pydron, we saw useful features we would like to include in DECO in the future. First Pydron uses data flow analysis to parallelize the @synchronized function. Incorporating data flow analysis into DECO could allow us to relax restrictions on index only based mutation, but up until now we have specifically avoided this in order to keep the library simple and concise. In the future we would like to consider methods of relaxing our restrictions, which may include implementing data flow analysis.

The concurrent programming model found in DECO and Pydron is easily extensible to forms of parallel computing other than threads/processes. In fact Pydron supports several varieties of cloud computing by using libcloud which allows it to run on Amazon's EC2, Digital Ocean and many other cloud providers. In our future work we would like to support libcloud as well to increase DECO's generality and make it more useful for programs that exhibit extreme parallelism.

7 Conclusion

We have proposed DECO, a simplification of concurrent programming techniques targeted at programmers with little understanding of concurrent programming. Taking inspiration from Pydron, we hope our technique can be useful to scientific programmers who have embarrassingly parallel programs that would otherwise execute in serial. Our technique maintains the performance of traditional Python multiprocessing but provides a simpler interface, making programming with DECO appear more like serial programming. All things considered we think DECO is close to the simplest abstraction for concurrent programming in the Python language while maintaining as much performance as possible.

References

[1] R. Choy, A. Edleman, J. R. Gilbert, V. Shah, and D. Cheng. Star-p: High productivity parallel computing. Technical report, DTIC Document, 2004.

[2] P. Den Hartog and A. Sherman. Edsm: An extensible system for distributed shared memory. Technical report, University of Wisconsin - Madison, 2015.

[3] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, 2012.

[4] S. C. Muller, G. Alonso, and A. Csillaghy. Scaling astroinformatics: Python + automatic parallelization. *Computer*, 47(9):41–47, 2014.

[5] C. M. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *Computer*, 23(12):13–23, 1990.