Implicit Parallelism at Runtime in Compiled Python Programs (Update)

Nicolas Broeking Joshua Rahm

April 9, 2015

Abstract

As we make our way into the second half of the second decade of the third millennium, Moore's law has finally begun to wane as processor clock speed have started to stagnate. However, while processor clock speeds remain the same, the number of cores in a processor has drastically increased. This is especially true with the advent of general purpose GPUs (GPGPUs). How to use these cores effectively is still a hard problem in software engineering especially because parallel processing requires careful attention to protect against race conditions and other caveats. As a result, the approach many languages take is to outlaw true parallel processing, but we believe that the answer is to bake parallel processing into the emitted assembly code so that the programmer never need to know of its existence, but may still enjoy the many benefits of a multi-core machine.

1 Group

Backslash x90 is us; Nicolas Broeking and Joshua Rahm. We are graduate students at the University of Colorado Boulder with interests in Systems Development and Telecommunications.

2 Problem

In software engineering, parallelism is still a very under used feature. Many programming languauges require the programmer to implement parallelism explicitly, such as C/C++ and Java, others, such as Python and Ruby, must simply rely on the operating system's multiprocessing facilities to manage multiple instances of the program and use the IPC mechanisms of the operating system to share data. Each of these approaches have their benefits and their pitfalls.

In the latter case, it is well known that a context switch between two different processes is *much* slower than threads, not to mention that passing data with messages via the operating system's IPC interfaces in much slower than shared heap space. Yet still, multiprocessing requires significant boiler plate programming to implement. However, in spite of these shortcomings, it is easy to implement this system once the boiler is in place, as synchronization is not an issue.

In the former case, explicit threading, the problems are the inverse. Manual threading is fast and efficient, but require due diligence to prevent race conditions and deadlocks. In addition, manual threading, like its counterpart, still requires much boiler plate implementation to use it to its full advantage, and yet the shared nature of threads implies that if a single thread has a bug, then it may cause the entire system to crash.

3 Proposal

We propose to go around the manual, explicit implementation and instead let the compiler use a thread pool to parallelize problems implicitly.

The traditional computation model has been to have a single CPU execute each instruction sequentially, but what if we are able to separate blocks of execution in which they may be able to execute independently, and are therefore able to be parallelized.

For example, take the simple Python code below.

Parallel Matrix

```
def matrix():
    m = rand_matrix(1024, 1024)
    a = inv(m)

    m = rand_matrix_n(1024, 1024)
    b = inv(m)

    z = a * b

    return z
```

It is trivial to see that, in fact, a and b are actually independent of each other, and as a result may be implemented independently.

Even though m is assigned to, these can be semantically two different variables and treated as such.

Another colloquial example is the following:

Parallel List Comprehension

```
def f():
    x = 0
    while(x!=4):
        x+=1
    return x

def b():
    x = 0
    while(x!=400):
        x+=1
    return x


y = f()
z = b()

print y + z
```

In this example if f and b are pure functions, meaning they do not have any side effects, then we are able to parallelize this code for both aspects of this list comprehension. Our goal is to detect that f b can be run independently, imlicitly parallelize it while maintaining the semantics of the programs.

If we are able to detect pure functions and parallelize the code regardless of performance we will consider the project to be a success. However as apart of our reasearch we will determine the performance gain of our system and try to discover possible ways to increase it.

4 Approach

Python is a dynamically typed programming language that naturally does in many ways limit its ability to be effectively parallelized which will force us to work with a subset of the Python Programming Language, p_3, that will allow us to implement our improvements under the assumption that the user has not overridden certain operators and types.

With these fairly limited changes to the Python language, we are still left with the challenge of parallelizing a program which we have limited knowledge of at compile time. Since this is the case, we will need to, at compile time, add many clues to indicate what may be optimized and what may not be able to. This will add several phases to our compiler.

In order to accomplish our tasks we will extend the course compiler with two new phases, Purity Analysis and Interference Detection.

| Purity Analysis

We may no longer assume that all functions will modify the global state of the program, and discovering purity when it exists is going to be the main distinction between parallelizeable code and sequential code. A handful of languages, most notably Haskell, have implemented purity checking with their type checker. C++ has also to an extent with the *const* keyword.

Detecting purity will be fairly straight forward. The function f is considered pure if it never invokes any impure functions. A non pure function is a function that modifies the any variable that exists outside the scope of a function. So, for example, in this code sample

Purity

```
def pure(l):
    x = l.copy()
    x.append(5)
    return x

def impure(l):
    l.append(5)
    return l
```

The function *pure* is in fact pure while the function *impure* is impure. Even though, both functions call *append*, a mutator function, *pure* does it to a variable which was created in its own scope.

The question now becomes, what if the *append* function modifies some global state of the program? It is already considered to be an impure function since it mutates $self$. This is why, there needs to be some

further analysis on member functions to determine if they are globally pure (meaning they only mutate *self*) or if they are globally impure (they make modifications outside of the scope).

This is going to require metadata at runtime to discover the levels at which aspects may be parallelized. This data may be stored in a tag of the function object in the runtime to indicate the level of purity of a function.

This means a function such as the following

```
def map(f, l):
    return [f(x) for x in l]
```

may be compiled to an intermediate language like

```
def map(f, l) {
    if is_pure(f) {
        return parmap(f, l)
    } else {
        ret = []
        for(i in l) {
           ret.append(f(i))
        }
        return ret
    }
}
```

| Interference Detection

We must find all nodes where a variable is being assigned to a pure expression and spawn a thread at the begining of the program which calculates that value and join that thread before the variable is assingned. To do this, we will add a phase to our compiler that strips out those expressions and creates threads to compute them at the beginning of the function call and join those threads as the value are needed.

This will also require pure functions to be implemented such that all arguments are passed as a copy so that mutations to that variable in other threads are not seen by the current thread.

5 Literature Survey

Our approach will add the ability, at runtime, detect what aspects of the code may be able to be parallelized In a dynamically typed language. The sources we have found have provided insights on how to implicitly parallelize a traditional, statically typed language, but we are attempting to extend their work to parallelize a dynamically typed language. The work done by HAL achieved parallelism by using Partitioned Global Address Space Languages (PGAS). Another group, from IBM and The University of Illinois, achieved implicit parallelism using ordered transactions (IPOT). Both of these frameworks provide many insights into the implementation of implicit parallelization; we will use both of these insights in the design of our final compiler.

6 Evaluation

We will evaluate our success via a couple of metrics. First, we will consider our project successful if we are able to successfully compile and be semantically correct while using parallel processing.

Better performance of the tests relative to the original compiler is not necessarily a requirement for success, as we realize that of the tests are too small to gain much from multiple threads. However, the project will be a great success if we are able to see significant improvement in the runtime of our test suite.