Implicit Parallelism in Compiled Python Programs

Nicolas Broeking
Department of Computer Science
University of Colorado Boulder
CSCI 5502
nicolas.broeking@colorado.edu

Joshua Rahm
Department of Computer Science
University of Colorado Boulder
CSCI 5502
joshua.rahm@colorado.edu

ABSTRACT

In the second half of the current decade, Moore's law has finally run out of steam as processor clock speeds have stagnated. However, while processor clock speeds remain the same, the number of CPUs and processors present on a device has drastically increased. This is especially true with the advent of general purpose GPUs (GPGPUs). How to effectively use these cores 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 parallelism, but we believe that the answer is to bake parallelism into the emitted assembly code so that the programmer never need know of its existence, but may still enjoy the many benefits of a multi-core machine.

Introduction

When programming, we do not always use the machine in the most time efficient manner. This is expressed mainly in programs that have parts which operate and execute independent of each other but are forced by the language or the programmer to execute in sequence rather than in parallel. We propose that instead of relying on the programmer to implement the threading, we can allow the compiler to do some threading implicitly to avoid common errors programmers make when trying to thread their applications.

We build this compiler such that it can detect if a function is pure; that is, a function whose return value depends solely on the input supplied to it. Using this, we can parallelize p3 in three stages.

First, we extend the runtime to have some more functions. These functions aid us in creating the environment needed to do necessary threading at runtime. Second, we change the compiler to emit the necessary calls to the extended runtime. And finally, we need to analyze the purity of each function; is a function pure, impure or something in-between.

Using the steps from above, we successfully created an implicitly parallelizing compiler that showed a significant increase in performance on many tests; all the while passing every one of our 157 tests, including all the instructor's tests.

This project has left us with some features which we would like to see implemented, but do to time constraints never got around to implementing, such as function scoring and implementing a "Green Thread" model.

Problem

There are many programs that can easily benefit from parallelism. Consider the following program:

```python
def calculatePi( place ):
        # calculate pi for ten minutes
        return pi.place

a = calculatePi(0)
b = calculatePi(1)
c = calculatePi(2)
d = calculatePi(3)
e = calculatePi(4)
f = calculatePi(5)

print a
print '.'
print b
print c
print d
print e
print f
```

In this program, we have several calls to `calculatePi`, and this function take 10 minutes to run, as a result this program takes over an hour to run. This is most unfortunate, as this program is very parallelizable, just the programmer decided not to implement any parallelism, and one cannot be blamed for this decision. When implementing parallelism manually, not only is it anemic in imperative languages, but is also full of potential pitfalls for the programmer dealing with synchronization and race conditions.

If there is a way to make the machine detect when functions may be parallelized and do the parallelism for the programmer, removing those potential caveats and tedious laboring, there is the potential to create a compiler that goes gangbusters.

For the Python programming language, implicit parallelism is not an easy problem to solve. Some languages, like Haskell, already have implicit parallelism, but this is under the structure of a very strong, static type system that makes it much easier to detect when blocks of code and functions may be able to run in parallel, all of which may be done at compile time. We have the task of parallelizing python code which is much more dynamic and as such, much of our logic for implementing the project must also be done dynamically.

Implementation

We extended the homework 6 compiler in 3 specific ways in order to achieve our goals. First was extending the runtime to include functions for the compiled programs to use. Second, was to emit the new assembly by extending the compiler code itself, and finally, detecting pure functions.

1. Runtime

We have extended the runtime to include two new public functions; `dispatch` and `join`. The headers of which are shown below:

```
u32_t dispatch(u32_t* out, pyobj func, int n, ...);
void join(u32_t);
```

The function dispatch takes an address to write the return value to, a function to call and a list of arguments passed as a `va_list`. The return value of dispatch is a thread id. This thread id is later used by the `join` function to know what thread to join on.

When dispatch is called, it does two things. First it checks to see if the function passed is pure. This is done via metadata attached to the function at compile time, a stage we will talk more about in the next section. If the function is pure, `dispatch` will then spawn a thread that runs a small assembly routine `__thread_start` that simply calls the function `func` with the arguments in the `va_list` and stores the result in the `*out`. This thread id is then returned from the function to be joined at a later time.

If the function is not pure, dispatch will just run the function sequentially store the result in `*out` and finally return 0, indicating the NULL thread id.

Finally, `join` is the analog to `dispatch`. For every `dispatch`, there should be at least one matching `join`. All join does is join on the thread id if it is not 0 otherwise it does nothing and just returns.

2. Emit Runtime Calls

Extending the compiler itself was the most difficult parts of this project. First we realized that now, our return values had to be on the stack since we needed to get the address to them. As a result we needed to add another rule to our graph coloring called `stack_only` meaning that the variable in question had to be on the stack so we can take the address of it.

Once that step was done, it was time to implement the famous `leal` instruction to get a pointer to the return value to pass as an argument to the `dispatch` function.

We ended up having to implement all the liveness and spill rules for this new instruction.

The liveness rules for `leal` are

$$L_{before}(leal(s,d)) = (L_{after} \cup \{s\}) \setminus \{d\}$$

After adding support for the leal instruction, we are able to modify our callfunc to emit assembly code that calls the runtime functions. For example, the program:

```
 def add(x, y):
        return x + y

x = add(1,2)
y = add(4, 6)
z = add(7,8)

print x
print z
print y
```

will be preprocessed to semantically be equivalent to the following:

```
def add(x, y):
        return x + y
x, y, z
tid1 = dispatch(&x, add, 1, 2)
tid2 = dispatch(&y, add, 4, 6)
tid3 = dispatch(&z, add, 7, 8)

join(tid1)
```

```
print x
join(tid3)
print z
join(tid2)
print y
```

In this new program, it is easy to see that instead of calling the functions directly, we instead call the dispatch function with not only the original arguments, but also a pointer to the return value (denoted by the C-style `&`) and a closure to call. Finally, before each value is used, the compiler is smart enough to inject a join to ensure the thread calculating that value has stopped before the value is used.

Thread Liveness

To implement the rule where each dispatch has at least one matching join, we had to implement a sense of thread liveness. That is, in all branches of the execution, we must make sure that the thread calculating a value is joined.

We do this in our flattening phase. As we iterate through each instruction, if we see an assign from a `CallFunc` then we add the left hand side of the assignment to a list called `joinable_vars`. At this point, if we see a variable being used that is in the `joinable_vars`, we inject a join statement on that thread id before the usage of that variable and proceed to remove that variable from the set of `joinable_vars`. At this point the Join node contains just the variable name, it is not until the next phase that the variable names get mapped to thread ids. If we come to an if statement, we implement a conservative policy that propagates a copy of `joinable_vars` as we decend into the if statement. That way, the if statement does not affect the `joinable_vars` as we must assume that no variables are joined in the if statement and still emit code to join those variables after the if statement even if it turns out to be redundant.

Similarly, loops must be assumed to not have been executed and as such, any threads spawned outside the loop must also be joined outside the loop even if they happened to be joined inside the loop.

For simplicity, we join all not joined threads at the end of each body of code.

For example, the following code snippet exemplifies this liveness

```
z = input()
x = f(a)

y = 0
if z == 3:
    y = g(x + 2)
y = y + x + 1
```

which is shown here

```
z = input()
        {z}
x = f(a)
        {z, x}
y = 0
        <- Join(z)
        {x}
if z == 3:
            <- Join(x)
            {}
    y = g(x + 2)
            {y}
            <- Join(y)

        {x}
        <- Join(x)
        {}
y = y + x + 1
```

Variable to Thread Mapping

As we move to the register selection phase of our compiler, our internal Core AST is filled with `Join(x)` nodes where x is the name of a variable. What we need to know is change the variable names to actual thread ids. The way we do this is, as we iterate through the Core AST, if we see an assign from a function call, we add the left hand side of the assignment to a dictionary and map it to a generated variable t_x that represents the thread executing x. Once we see a `Join(x)`, we swap it with a $CallFunc(Join, t_x)$ node, this then gets compiled into the correct instructions.

3. Purity Analysis

Purity analysis is where a large portion of the work for this project went. We detect the purity of a function during the uniquify and heapify phases of the compiler. We walk through a function under the context that it is already pure and look for evidence that the function is not pure. Rules for a pure function are as follows:

1. The pure function may not access its outer closure, as this is subject to change and may produce impure results. (This includes recursive functions)

2. A pure function may not access the subscripts of any arguments, or any local variable pointing to the arguments. As these are subject to change with race conditions and lead to impure results.

3. A pure function may only call a pure inner function.

4. A pure function may not call print as this may lead to different print ordering, leading to impure results.

So some examples of impure functions would be

```
def impure0(x):
    return x[0]

def impure1(x):
    def f():
        print "hi"
    f()
    return x

def impure2(x):
    return x + pure()
```

An example of a pure function is:

```
def pure0(x):
    return x + 5

def pure1(x):
    lst = [1,2,3]
    return lst[0] + lst[1] + lst[2] +- x
```

Conditional Purity

During this project, we have coined a concept known as *Conditional Purity*. A conditionally pure function is a function that may be considered pure depending on the inputs given to it as arguments. For example, consider the following code:

```
def conditionally_pure(f, x, y):
    return f(x) + f(y)
```

This function is cosidered impure by the rules stated above since it calls a function defined outside its scope. This function; however, is pure if the function f is pure. As such, this function is conditionally pure on f. We can represent this conditionally pure attribute and at runtime detect if the *call* to the function is pure or impure.

Purity Marking

In order to detect the purity of a function at runtime, we added an attribute to the function structure that consists of a 32-bit bitmask called *purity*. This bitmask is equal to 0 for impure functions, 1 for unconditionally pure functions, and, for conditionally pure function, $1 + \sum_{n=1}^{30} e(n)2^{n+1}$ where n is the n^{th} argument and $e(n) = 1$ if the function is conditionally pure on the n^{th} argument. $e(n) = 0$ otherwise. In other words, if the function is conditionally dependent on argument n, then the $n^{th} + 1$ bit is set to 1.

The way the compiler stores this bitmask persistently is with a `.long` tag that resides in the `.text` section before the definition of each function. For example, the functions above produce the assembly:

```
.long 0
impure:
    pushl %ebp
    movl %ebp, %esp
    ...

.long 1
pure:
    pushl %ebp
    movl %ebp, %esp
    ...

.long 3 # binary 0000 ... 0011
conditionally_pure:
    pushl %ebp
    movl %ebp, %esp
    ...
```

We modified create closure to take the value from the function pointer offset -4 to extract the tag put by the compiler. This way, the dispatch function can retrieve all the information it needs to determine if a function call can be parallelized at runtime.

Literature Survey

The meat of our project was based off of a research project completed with IBM and the University of Illinois. This project they were able to implement a Programming Model called Implicit Parallelism with Ordered Transactions or IPOT for short. IPOT is the idea that one can parallelize a sequential program using annotations. Each one of these annotations can be threaded at the start of the annotation and then joined again when another transaction needs the data from the transaction. This gave us inspiration with how we wanted to structure our compiler.

Our execution model almost identically matches the IPOT execution model. It has a spawn / squash stage which is the equivalent to our dispatch and join runtime functions. Instead of a purity analysis phase though it uses a conflict detection phase. The main difference is that we perform analysis at a function by function level. However if you are able to more fully optimize the code by parallelizing chunks then you need to perform conflict detection. For the purposes of our project we decided to go with purity analysis instead.

While reviewing their paper, the use of annotations brought up an interesting question among our team. Is having the user annotate the blocks of code that can be parallelized truly implicit parallelism? We decided that no it isn't, and that we could extend their work by having the compiler do all the heavy lifting and create a truly implicit compiler.

Results

Benchmarks

We ran the new compiler with implicit parallelism baked into it on all the previous tests we made for the compilers from the past and all passed. This shows that our compiler is still semantically correct. We also benchmarked the tests against the compiler submitted for homework 6. Overall the compiler produced code that ran 24.5% slower than the homework 6 counterpart. This is mostly due to the extra overhead of the dispatch function along with the overhead of spawning threads for simple functions, as our compiler indiscriminately spawns threads for *all* pure functions.

We did find very promising results for tailored examples run against the compiler. The following code was run against both the homework 6 compiler and the new compiler:

```
def map_sum(f, x, y):
    return f(x) + f(y)

def addOne(x):
    y = 0
    lst = [1,2,4]
    while y != 10000000:
        y = y + lst[2] +- (lst[1] + lst[0])
    y = y +- 9999999
    x = x + y
    return x

x = map_sum(addOne, 4, 5)
y = map_sum(addOne, 5, 6)
z = map_sum(addOne, 7, 8)
w = map_sum(addOne, 9, 10)
v = map_sum(addOne, 11, 12)

print x + y +- z + w +- v
```

With this test case, we saw a 4x speedup with the new compiler and a 8x speedup from interpreted Python on an AMD Phenom II processor. Notice that this is with the `map_sum` function being a conditionally pure function and `addOne` function being pure, so the compiler was able to detect purity and conditional purity and the runtime was able to detect pure calls.

Even though this is a fairly tailored example, it does show promise into the potential power of an implicitly parallelizing compilers and interpreters even if the language is a dynamically typed language.

Future Steps

With the limited implementation time of this project, we were not able to implement everything we wanted.

Function Scoring

The ability to score a function's complexity can eliminate the problem where simple functions are unnecessarily spawned in a separate thread. The way function scoring would have to be implemented is at runtime executing some form of static analysis to try to estimate how many instructions will be executed on average by this function. If the estimated number of instructions is more that the overhead for starting a joining a thread, then it should be marked as parallelizeable.

Better Purity Rules

Right now, our purity rules are Draconian. A future feature might be to liberalize the purity rules by performing more static analysis to determine if, for example, calls to outer functions are provably pure or if free variables used are guaranteed to be read only. This requires analysis not on the function itself, but all parent function to make sure the variables read from have no chance of changing by the functions that share the same closure.

Green Thread Model

Languages like Haskell have a "Green Thread" model. What this means is that instead of leveraging OS threads, these languages use user-space "green" threads. This makes spawning, joining and signaling green threads much, much faster than the same operations for an OS thread. The disadvantage is, of course, that green threads are not able to run on separate cores by themselves. This is why languages like Haskell spawn one OS thread for each CPU on the machine and then schedule the many user-space threads on these separate OS threads. This allows for more efficient usage of processor resources and reduces calls to the kernel to use the threads.

More Programmer Control

Finally, sometimes the compiler should not be in charge of making decisions without the programmers blessing. As such, it may be in the interests of the language designer no allow additional annotations (like C's `#pragma` or Java's `@interface`) that allow the programmer to specify what functions should be parallelize and what functions are better left to run sequentially. After all, the programmer knows best how the code will execute and whether it makes sense to have the compiler parallelize the code.

Conclusion

We started out with the challenge to build a compiler for a dynamically typed language and that we have done. We then challenged ourselves to make a compiler that not only compiled Python, but also was able to implicitly parallelize it. This is hard work seeing as even the Python interpreter does not even allow for explicit parallelism. Nonetheless, we were able to put together a working prototype that was able to parallelize simple programs.

We faced challenges with the dynamisms of Python, and realized there was no way around doing much of the logic at runtime. Nonetheless, we were still able to produce a compiler that generated code capable of spawning threads at appropriate time.

While our performance results at first may look discouraging as there was a general slowdown. However, we did see a very significant boost in speed for some tailored case which we believe implies that if the extensions mentioned above were to be implemented, we will see a significant improvement in the performance of nearly all cases.

Overall, we consider the project to be a resounding success given the initial uncertainty about whether or not it would even be possible to parallelize Python code, or even do on the fly thread spawning and joining.