PyPy Status Blog

THURSDAY, AUGUST 9, 2012

Multicore Programming in PyPy and CPython

Hi all,

This is a short "position paper" kind of post about my view (Armin Rigo's) on the future of multicore programming in high-level languages. It is a summary of the keynote presentation at EuroPython. As I learned by talking with people afterwards, I am not a good enough speaker to manage to convey a deeper message in a 20-minutes talk. I will try instead to convey it in a 250-lines post...

This is about three points:

1. We often hear about people wanting a version of Python running without the Global Interpreter Lock (GIL): a "GIL-less Python". But what we programmers really need is not just a GIL-less Python --- we need a higher-level way to write multithreaded programs than using directly threads and locks. One way is Automatic Mutual Exclusion (AME), which would give us an "AME Python".
2. A good enough Software Transactional Memory (STM) system can be used as an internal tool to do that. This is what we are building into an "AME PyPy".
3. The picture is darker for CPython, though there is a way too. The problem is that when we say STM, we think about either GCC 4.7's STM support, or Hardware Transactional Memory (HTM). However, both solutions are enough for a "GIL-less CPython", but not for "AME CPython", due to capacity limitations. For the latter, we need somehow to add some large-scale STM into the compiler.

Let me explain these points in more details.

GIL-less versus AME

The first point is in favor of the so-called Automatic Mutual Exclusion approach. The issue with using threads (in any language with or without a GIL) is that threads are fundamentally non-deterministic. In other words, the programs' behaviors are not reproductible at all, and worse, we cannot even reason about it --- it becomes quickly messy. We would have to consider all possible combinations of code paths and timings, and we cannot hope to write tests that cover all combinations. This fact is often documented as one of the main blockers towards writing successful multithreaded applications.

We need to solve this issue with a higher-level solution. Such solutions exist theoretically, and Automatic Mutual Exclusion (AME) is one of them. The idea of AME is that we divide the execution of each thread into a number of "atomic blocks". Each block is well-delimited and typically large. Each block runs atomically, as if it acquired a GIL for its whole duration. The trick is that internally we use Transactional Memory, which is a technique that lets the system run the atomic blocks from each thread in parallel, while giving the programmer the illusion that the blocks have been run in some global serialized order.

This doesn't magically solve all possible issues, but it helps a lot: it is far easier to reason in terms of a random ordering of large atomic blocks than in terms of a random ordering of lines of code --- not to mention the mess that multithreaded C is, where even a random ordering of instructions is not a sufficient model any more.

How do such atomic blocks look like? For example, a program might contain a loop over all keys of a dictionary, performing some "mostly-independent" work on each value. This is a typical example: each atomic block is one iteration through the loop. By using the technique described here, we can run the iterations in parallel (e.g. using a thread pool) but using AME to ensure that they appear to run serially.

In Python, we don't care about the order in which the loop iterations are done, because we are anyway iterating over the keys of a dictionary. So we get exactly the same effect as before: the iterations still run in some random order, but --- and that's the important point --- they appear to run in a global serialized order. In other words, we introduced parallelism, but only under the hood: from the programmer's point of view, his program still appears to run completely serially. Parallelisation as a theoretically invisible optimization... more about the "theoretically" in the next paragraph.

Note that randomness of order is not fundamental: they are techniques building on top of AME that can be used to force the order of the atomic blocks, if needed.

PyPy and STM/AME

Talking more precisely about PyPy: the current prototype `pypy-stm` is doing precisely this. In `pypy-stm`, the length of the atomic blocks is selected in one of two ways: either explicitly or automatically.

The automatic selection gives blocks corresponding to some small number of bytecodes, in which case we have merely a GIL-less Python: multiple threads will appear to run serially, with the execution randomly switching from one thread to another at bytecode boundaries, just like in CPython.

The explicit selection is closer to what was described in the previous section: someone --- the programmer or

the author of some library that the programmer uses --- will explicitly put `with thread.atomic:` in the source, which delimitates an atomic block. For example, we can use it to build a library that can be used to iterate over the keys of a dictionary: instead of iterating over the dictionary directly, we would use some custom utility which gives the elements "in parallel". It would give them by using internally a pool of threads, but enclosing every handling of an element into such a `with thread.atomic` block.

This gives the nice illusion of a global serialized order, and thus gives us a well-behaving model of the program's behavior.

Restating this differently, the *only* semantical difference between `pypy-stm` and a regular PyPy or CPython is that it has `thread.atomic`, which is a context manager that gives the illusion of forcing the GIL to not be released during the execution of the corresponding block of code. Apart from this addition, they are apparently identical.

Of course they are only semantically identical if we ignore performance: `pypy-stm` uses multiple threads and can potentially benefit from that on multicore machines. The drawback is: when does it benefit, and how much? The answer to this question is not immediate. The programmer will usually have to detect and locate places that cause too many "conflicts" in the Transactional Memory sense. A conflict occurs when two atomic blocks write to the same location, or when `A` reads it, `B` writes it, but `B` finishes first and commits. A conflict causes the execution of one atomic block to be aborted and restarted, due to another block committing. Although the process is transparent, if it occurs more than occasionally, then it has a negative impact on performance.

There is no out-of-the-box perfect solution for solving all conflicts. What we will need is more tools to detect them and deal with them, data structures that are made aware of the risks of "internal" conflicts when externally there shouldn't be one, and so on. There is some work ahead.

The point here is that from the point of view of the final programmer, we gets conflicts that we should resolve --- but at any point, our program is *correct*, even if it may not be yet as efficient as it could be. This is the opposite of regular multithreading, where programs are efficient but not as correct as they could be. In other words, as we all know, we only have resources to do the easy 80% of the work and not the remaining hard 20%. So in this model we get a program that has 80% of the theoretical maximum of performance and it's fine. In the regular multithreading model we would instead only manage to remove 80% of the bugs, and we are left with obscure rare crashes.

CPython and HTM

Couldn't we do the same for CPython? The problem here is that `pypy-stm` is implemented as a transformation step during translation, which is not directly possible in CPython. Here are our options:

- We could review and change the C code everywhere in CPython.
- We use GCC 4.7, which supports some form of STM.
- We wait until Intel's next generation of CPUs comes out ("Haswell") and use HTM.
- We write our own C code transformation within a compiler (e.g. LLVM).

I will personally file the first solution in the "thanks but no thanks" category. If anything, it will give us another fork of CPython that will painfully struggle to keep not more than 3-4 versions behind, and then eventually die. It is very unlikely to be ever merged into the CPython trunk, because it would need changes *everywhere*. Not to mention that these changes would be very experimental: tomorrow we might figure out that different changes would have been better, and have to start from scratch again.

Let us turn instead to the next two solutions. Both of these solutions are geared toward small-scale transactions, but not long-running ones. For example, I have no clue how to give GCC rules about performing I/O in a transaction --- this seems not supported at all; and moreover looking at the STM library that is available so far to be linked with the compiled program, it assumes short transactions only. By contrast, when I say "long transaction" I mean transactions that can run for 0.1 seconds or more. To give you an idea, in 0.1 seconds a PyPy program allocates and frees on the order of ~50MB of memory.

Intel's Hardware Transactional Memory solution is both more flexible and comes with a stricter limit. In one word, the transaction boundaries are given by a pair of special CPU instructions that make the CPU enter or leave "transactional" mode. If the transaction aborts, the CPU cancels any change, rolls back to the "enter" instruction and causes this instruction to return an error code instead of re-entering transactional mode (a bit like a `fork()`). The software then detects the error code. Typically, if transactions are rarely cancelled, it is fine to fall back to a GIL-like solution just to redo these cancelled transactions.

About the implementation: this is done by recording all the changes that a transaction wants to do to the main memory, and keeping them invisible to other CPUs. This is "easily" achieved by keeping them inside this CPU's local cache; rolling back is then just a matter of discarding a part of this cache without committing it to memory. From this point of view, there is a lot to bet that we are actually talking about the regular per-core Level 1 and Level 2 caches --- so any transaction that cannot fully store its read and written data in the 64+256KB of the L1+L2 caches will abort.

So what does it mean? A Python interpreter overflows the L1 cache of the CPU very quickly: just creating new Python function frames takes a lot of memory (on the order of magnitude of 1/100 of the whole L1 cache). Adding a 256KB L2 cache into the picture helps, particularly because it is highly associative and thus avoids a lot of fake conflicts. However, as long as the HTM support is limited to L1+L2 caches, it is not going to be enough to run an "AME Python" with any sort of medium-to-long transaction. It can run a "GIL-less Python", though: just running a few hundred or even thousand bytecodes at a time should fit in the L1+L2 caches, for most bytecodes.

I would vaguely guess that it will take on the order of 10 years until CPU cache sizes grow enough for a CPU in HTM mode to actually be able to run 0.1-second transactions. (Of course in 10 years' time a lot of other things may occur too, including the whole Transactional Memory model being displaced by something else.)

Write your own STM for C

Let's discuss now the last option: if neither GCC 4.7 nor HTM are sufficient for an "AME CPython", then we might want to write our own C compiler patch (as either extra work on GCC 4.7, or an extra pass to LLVM, for example).

We would have to deal with the fact that we get low-level information, and somehow need to preserve interesting high-level bits through the compiler up to the point at which our pass runs: for example, whether the field we read is immutable or not. (This is important because some common objects are immutable, e.g. PyIntObject. Immutable reads don't need to be recorded, whereas reads of mutable data must be protected against other threads modifying them.) We can also have custom code to handle the reference counters: e.g. not consider it a conflict if multiple transactions have changed the same reference counter, but just resolve it automatically at commit time. We are also free to handle I/O in the way we want.

More generally, the advantage of this approach over both the current GCC 4.7 and over HTM is that we control the whole process. While this still looks like a lot of work, it looks doable. It would be possible to come up with a minimal patch of CPython that can be accepted into core without too much troubles (e.g. to mark immutable fields and tweak the refcounting macros), and keep all the cleverness inside the compiler extension.

Conclusion

I would assume that a programming model specific to PyPy and not applicable to CPython has little chances to catch on, as long as PyPy is not the main Python interpreter (which looks unlikely to change anytime soon). Thus as long as only PyPy has AME, it looks like it will not become the main model of multicore usage in Python. However, I can conclude with a more positive note than during the EuroPython conference: it is a lot of work, but there is a more-or-less reasonable way forward to have an AME version of CPython too.

In the meantime, `pypy-stm` is around the corner, and together with tools developed on top of it, it might become really useful and used. I hope that in the next few years this work will trigger enough motivation for CPython to follow the ideas.

Posted by Armin Rigo at 10:27

11 comments:

Armin Rigo said...

This comment has been removed by the author.

August 9, 2012 10:41 AM

JohnLenton said...

A question: does a “donate towards STM/AME in pypy” also count as a donation towards the CPython work? Getting the hooks in CPython to allow exploration and implementation of this seems at least as important as the pypy work. In fact, I think it’s quite a bit more important.

August 9, 2012 1:29 PM

Armin Rigo said...

@John: I didn't foresee this development at the start of the year, so I don't know. It's a topic that would need to be discussed internally, likely with feedback from past donators.

Right now of course I'm finishing the basics of pypy-stm (working on the JIT now), and from there on there is a lot that can be done as pure Python, like libraries of better-suited data structures --- and generally gaining experience that would anyway be needed for CPython's work.

August 9, 2012 1:55 PM

Anonymous said...

With HTM you don't have to have a one-to-one mapping between your application transactions and the hardware interface. You can also have an STM, that is implemented using HTM. So you may do all the book-keeping yourself in software, but then at commit time use HTM.

August 9, 2012 4:53 PM

Nat Tuck said...

No. We really do want a GIL-free Python. Even if that means we sometimes need to deal with locks.

Right now a high end server can have 64 cores. That means that parallel python code could run faster than serial C code.

STM and other high level abstractions are neat, but they're no substitute for just killing the damn GIL.

August 9, 2012 5:37 PM

Anonymous said...

What does 'just killing the damn GIL' mean without something like STM? Do you consider it acceptable for Python primitives not to be threadsafe?

If you intend to run 64 cores, then what is the exact reason you need threading and can't use multiprocessing?

August 9, 2012 6:32 PM

Anonymous said...

Jesus Christ why don't we all just spend 5 min fiddling with the multiprocessing module and learn how to partition execution and queues like we partition sequences of statements into functions? So sick of GIL articles and the obsession with not learning how to divide up the work and communicate. In some ways the need to recognize narrow channels where relatively small amounts of data are being channeled through relatively intense blocks of execution and create readable, explicit structure around those blocks might actually improve the comprehensibility of some code I've seen. Getting a little tired of seeing so much effort by excellent, essential, dedicated Python devs getting sucked up by users who won't get it.

I think users are driving this speed-for-free obsession way to far. If anything bugs in a magical system are harder to find than understanding explicit structure and explicit structure that's elegant is neither crufty nor slow. Eventually, no interpreter will save a bad programmer. Are we next going to enable the novice "Pythonista" to forego any knowledge of algorithms?

We -need- JIT on production systems to get response times down for template processing without micro-caching out the wazoo. These types of services are already parallel by nature of the servers and usually I/O bound except for the few slow parts. Cython already serves such an excellent roll for both C/C++ API's AND speed AND optimizing existing python code with minimal changes. JIT PyPy playing well with Cython would make Python very generally uber. Users who actually get multiprocessing and can divide up the workflow won't want a slower implementation of any other kind. Getting a somewhat good solution for 'free' is not nearly as appealing as the additional headroom afforded by an incremental user cost (adding some strong typing or patching a function to work with pypy/py3k).

August 9, 2012 8:54 PM

jwp said...

template processing. lol.

August 9, 2012 8:59 PM

Maciej Fijalkowski said...

@Anonymous.

I welcome you to work out how to make pypy translation process parallel using any techniques you described.

August 9, 2012 10:27 PM

Benjamin said...

I get the overall goals and desires and I think they are fabulous. However, one notion that seems counterintuitive to me is the desire for large atomic operations.

Aside from the nomenclature (atomic generally means smallest possible), my intuition is that STM would generally operate more efficiently by having fewer roll-backs with small atomic operations and frequent commits. This leads me to assume there is some sort of significant overhead involved with the setup or teardown of the STM 'wrapper'.

From a broader perspective, I get that understanding interlacing is much easier with larger pieces, but larger pieces of code don't lend themselves to wide distribution across many cores like small pieces do.

It seems, to me, that you're focusing heavily on the idea of linearly written code magically functioning in parallel and neglecting the idea of simple, low-cost concurrency, which might have a much bigger short-term impact; and which, through use, may shed light on better frameworks for reducing the complexity inherent in concurrency.

August 10, 2012 8:27 AM

Armin Rigo said...

@Anonymous: "So you may do all the book-keeping yourself in software, but then at commit time use HTM.": I don't see how (or the point), can you be more explicit or post a link?

@Anonymous: I'm not saying that STM is the final solution to all problems. Some classes of problems have other solutions that work well so far and I'm not proposing to change them. Big servers can naturally handle big loads just by having enough processes. What I'm describing instead is a pure language feature that may or may not help in particular cases --- and there are other cases than the one you describe where the situation is very different and multiprocessing doesn't help at all. Also, you have to realise that any argument "we will never need feature X because we can work around it using hack Y" is bound to lose eventually: at least some people in some cases will need the clean feature X because the hack Y is too complicated to learn or use correctly.

@Benjamin: "atomic" actually means "not decomposable", not necessarily "as small as possible". This focus on smallness of transaction IMO is an artefact of last decade's research focus. In my posts I tend to focus on large transaction as a counterpoint: in the use cases I have in mind there is no guarantee that all transactions will be small. Some of them may be, but others not, and this is a restriction. In things like "one iteration through this loop = one transaction", some of these iterations go away and do a lot of stuff.

August 10, 2012 9:57 AM

Post a Comment

Links to this post

Create a Link

Home Older Post

Subscribe to: Post Comments (Atom)