An Easy and Safe Implementation of Truly Parallel Threads in CPython

Samuel Grayson
University of Illinois at Urbana-Champaign

Abstract

Mutually trusting processes can be safely run in the same address-space (effectively running **processes-as-threads**) without necessitating source-code changes. This has compelling performance improvements for programs that can be realized with very little work. Some programs such as Python do not support parallel threads, and this may be the only easy way to unlock high performance on multicore architecture.

In just one semester I have implemented a prototype of the process-as-threads abstraction for CPython, which allows it to run parallel threads safely. The performance characteristics are promising, and it seems to scale better than the state-of-the-art with a large number of cores.

1 Motivation

While distinct virtual-to-physical mappings (**address space**) for several reasons are extremely important in the general case, I argue that there is a specific case where we can relax them for the sake of performance. Multithreading and virtual memory-aliasing (`mmap`) already relax this constraint in *non*-memory-safe languages, which is more dangerous than my proposal.

Stability One malfunctioning process should not be able to cause other processes to malfunction. When a programmer wants to parallelize code, they implicitly trust themselves, so the security of processes is not necessary here.

Security Processes do not necessarily trust each other with their volatile data. In a memory-safe programming languages, direct memory access is not possible, eliminating this class of bugs.

On the other hand, running in the same address space has important performance benefits:

Reduced context-switching time When the OS switches from one process to another it has to flush the TLB, file-descriptor table, and other per-process resources; When switching between threads, this state is shared.

Reduced communication/synchronization time Inter-process communication (such as a Unix pipe) has to copy data into and out-from the kernel; Inter-thread communication can leverage the shared memory-space, copying the data once or zero times (sending a reference).

Reduced memory fragmentation Huge pages are intended to extend the reach of the TLB, however this faces implementation problems. If each process gets a huge page, physical memory runs out, even though each process might only be using a fraction of its page. Multiple threads however could share a huge page between themselves.

Parallel workloads are becoming increasingly important due to the limitations of single-core performance and the move to multi-core. While challenging, this is much more practical for many users than setting up a distributed system. That is why I am interested in making multi-core programming easier to do with existing paradigms.

Python is an obvious place to start. The Jupyter Notebook interface explicitly targets data-scientists and tools like Numpy, Scipy, and Pandas are extremely popular tools for data analytics. However Python fails to harness multicore parallelism with shared memory. If it could do this, then it would accelerate a lot of real-world workloads from big companies. Every Spark-node could be utilizing process-as-threads to accelerate intra-node data-transfers on many-core machines like the Xeon Phi.

Python cannot support parallel multithreading because it is not threadsafe. There is a **Global Interpreter Lock (GIL)** ensuring only one thread of Python bytecode is executing at a time. It has a `threading` module, but it is limited by the GIL. The state-of-the-art for Python is to use

the `multiprocessing` module. While this achieves true parallelism, it leaves a lot of performance on the table (slow context-switching, slow communcation, and poor memory fragmentation) because the process abstraction is so heavy-weight.

Many other interpreted-languages, such as Ruby, have a GIL as well. In these lanuages, neither normal processes nor normal threads can provide the benefits of running in the same address-space.

2 Process-as-Threads Programming Model

My vision is to spawn multiple processes in the same address-space such that they do not interfere with each other. This has important implications for each memory segment: text, static, initialized, heap, and stack.

text Function-calls are implemented by jumping to the virtual-address containing the function. As a consequence, when the compiler lays out the binary, it tells the loader to load each piece of executable code into a specific virtual-address. It will hardcode that address later at every function-call site. Imagine A wants to install `foo()` at 0xDEADBEEF and B wants to install `bar()` at the same address.

My solution is to use **Position-Independent Code (PIC)** and shared-libraries. When PIC is enabled, compilers add an extra layer of indirection: they first go to a table which is located by its address *relative* to the $PC. This **Global-offset Table (GOT)** contains the real address of the desired function. When linking as a shared-library, the code can be loaded anywhere, as long as the loader also updates the GOT.

static The same problem occurs for static and initialized global variables. The compiler normally hard-codes virtual-addresses to static variables. However, the same solution also works: The shared-library loader updates a GOT accessible by relative address.

heap I need to show that the programs don't interfere with each-other's heap space. I can prove this by observing that a memory-safe program never accesses (read or write) memory that never allocated first. As long as the memory-allocator is aware of multiple threads and does not return the same memory for multiple threads, then the programs will always allocate objects in non-conflicting places on the heap. The heap is logically partitioned (each object is only referenced by one thread) despite it not being literally partitioned in virtual-address space.

When I require that memory-safe accesses, I am not requiring any property of the *user's code*, just the *interpreter's code*. At this point, the Python interpreter is considered mature and usually user-code crashes, not its code.

stack With the other segments out of the way, it is safe to use threads. Threads give us independent stack-segments.

Despite having the *performance benefit of threads*, the process-as-thread programming model has the *strong semantics of processes*. With multithreading, each thread has to reason about another thread modifying all of its data, because other threads can take references to any thread's data. With processes-as-threads, data is not referencible from other process-as-threads by default. This is surprising considering that they run in the same address space, but if the code is written in a memory-safe language, it is impossible to take a reference to data from another process-as-thread without explicit permission. Where it is expedient, processes-as-threads can *explicitly* send a reference to another process-as-thread. This is a useful for the underlying runtime to do when it is safe (e.g. providing read-only access to a global constant). Because privacy is not default, it does not burden the programmer with complex memory semantics, race conditions, and deadlocks.

3 Python Implementation

In Python, a CPU-bound parallel work is often done using the `multiprocessing` library. It provides low-level primitives such actors, queues, and pipes, but also high-level abstractions such as a process pool. I could attempt to make process-as-threads emulate the low-level primitives, but instead I just attempted to provide the high-level abstractions with process-as-threads. Because my interface is the same, users only have to change one line of code to reference my library instead of the default `multiprocessing` library. Furthermore, accelerating a higher-order primitive means that I can transparently easily replace the interprocess-communication going on the background with fast interthread-communication.

Now I have the problem of non-atomic reference-counting. Two concurrent threads trying to change the reference-count of the same variable is a race-condition, which could render the whole program incorrect. To avoid this I deepcopy every object immediately before it gets sent to another thread. That way both threads have a copy of the object, but each thread only references its copy. Deepcopying is somewhat expensive, but is strictly cheaper than to serialization/deserialization, which is what multiprocessing usually has to do to communicate. Processes-as-thread's communication is cheaper in other ways: thye do not have to invoke a syscall to send a message. Note that immutable data-types such as strings, tuples, functions, and primitives *are not even copied at all*: when `deepcopy` encounters these, it just returns the original pointer. Since the destination is in the same address-space, this works without modification!

Now, my safety is guaranteed by the invariant that every object has references in at most one thread. The following three operations maintain the invariant:

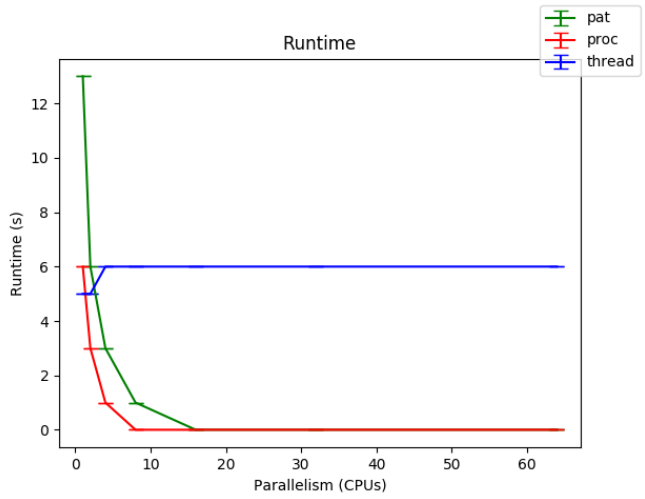

Figure 1:

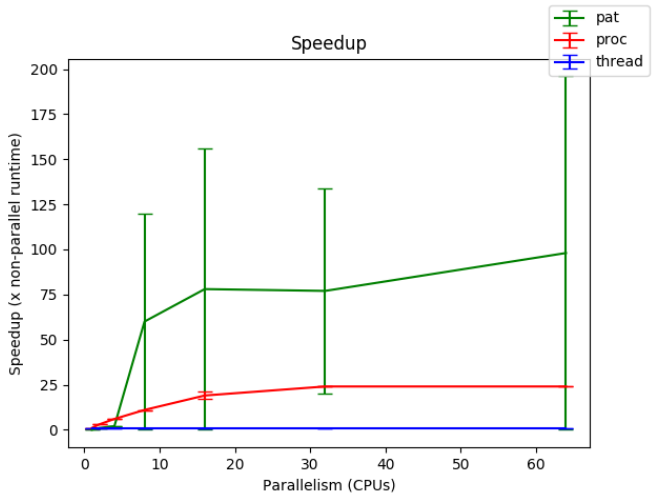

Figure 2:

- Allocating
- Deallocating
- Taking a reference within the same thread
- Copying within the same thread
- Deepcopy+send between threads

See Appendix 6.1 for implementation details.

4 Empirical Results

I tested this on a program that counts up to a large number, call it n. When this is parallelized, each of m actors has to count to $\lfloor \frac{n}{m} \rfloor$ The only communication is starting up the actors, telling them the number, and then waiting for them to finish. I ran it with process-parallelization (`proc`), a thread-parallelization (`thread`), and a process-as-thread parallelization (`pat`).

Figure 1 shows how the performance scales to many cores. Thread-parallelization actual gets worse when we add more cores. This demonstrates the GIL's limitation on Python's existing threading facilities. Process-as-threads and processes both scale well with these conditions, demonstrating truly parallel (not just concurrent) execution.

However, the lines get close together, so it is hard to tell how the performance scales at 64 cores. This is more apparent in the speedup-over-baseline in Figure 2. We see that process-as-threads totally outperforms processes with a very high number of CPU. The error-bars are very high because the runtimes are close to zero, so small absolute deviations blow up after taking the reciprocal.

This test does stress communication at all, which should only be more compelling for process-as-threads.

See Appendix 6.2 for my experimental conditions and Appendix 6.1 to reproduce.

5 Unresolved Issues

There is an unresolved issue that prevents Python imports from working properly. Usually when one loads shared-library, call it libA, libA's symbols are exported globally. This means that when a subsequent shared-library are loaded, call it LibB, their undefined symbols are resolved within libA. When an application loads Python, calls it to run a script, and that script loads a C-extension, the C-extension expects its undefined symbols to be resolved within Python.

However, this poses a problem for process-as-threads. Remember that each process-as-thread loaded its own copy of Python, so it could get its own copy of the text and static variables. Now when libB gets loaded, which library should its undefined symbols be resolved in? There is no POSIX compliant way of opening a shared-library and capturing its names in a namespace. One either has to export them globally or not at all through `dlopen`.

However Linux does offer such a facility: `dlmopen`. You can open a library *inside* a non-default namespace; its symbols get added to this namespace, and its undefined symbols are resolved in that namespace. Then I could make each thread have its own namespace object, and I would ensure that all libraries that would be loaded globally get loaded into that namespace object instead. Now when libB needs to be loaded, its unresolved symbols can be found in the calling thread's namespace object. But for some reason this is not working, and it is causing Python to crash when it attempts to import a C-module.

6 Conclusion

In just one semester I have implemented a prototype of the process-as-threads abstraction. I have been able to show promising performance benefits on a software project as mature as Python, in little time and with little experience. The process-as-threads abstraction can provide this benefit without modifying CPython nor the OS kernel at all. Compare this to other proposals which involve a massive overhaul of the CPython interpreter–a so-called 'GIL-ectomy'.

My vision is that with one more year I could develop a polished process-as-threads which could parallelize any programming language interpreter without any change to process-as-threads or the language. My work could even apply to run two different lanuage interpreters in the same addres-space, accelerating polyglot applications (i.e. a system utilizing code in multiple languages simultaneously).

Appendex

6.1 Availability

To reproduce:

1. Clone the code from here https://github.com/charmoniumQ/exec_sharing/
2. Switch to branch `dlib-load-no-libs`.
3. Run `./bootstrap_debian.sh` to prepare your machine. It should be running a recent Debian.
   (a) Be aware that the script will irrevocably uprade to Debian Testing. This is because my code needs a recent version of clang.
4. Run `./perf/count.sh` to reproduce the performance tests.
   (a) Run `./perf/plots.py` to produce the plots.
   (b) If you would like to try your own performance test with process-as-threads, mimic `./perf/count_pat.py`.
5. Run `make tests` to run functional tests.
   (a) If you would like to write your own functional tests, mimic those in `./tests/test_*.sh`.

6.2 Experimental Conditions

I used an AWS c5.18xlarge (72-core) with Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz.

A smaller machine would probably be more appropriate, because I did not max out the cores.

I used `taskset` to ensure the processes did not migrate threads.

I used `perf` to capture total-time. I decided not to present user-time and system-time separately because they were unstable.