POSH: Python Object Sharing

Steffen Viken Valvåg <steffenv@fast.no> Åge Kvalnes <aage@cs.uit.no>
Kjetil Jacobsen <kjetilja@cs.uit.no>
University of Tromsø, N-9037 Tromsø, Norway

Abstract

Python uses a single global lock known as the global interpreter lock (or GIL) to serialize execution of byte codes. The GIL becomes a major bottleneck when executing multi-threaded Python applications, in particular on multi-processor architectures. This paper presents POSH, which is an extension module to Python that attempts to address the problems associated with the GIL by enabling placement of Python objects in shared memory. In particular, POSH allows multiple processes to share objects in much the same way that threads do with standard Python objects. We have found that the use of POSH allows some applications to be structured as if they used threads, but without the GIL bottleneck.

1 Motivation

Python language support for threads was introduced at a late stage in the Python development process. At this point, the majority of Python code was not thread-safe and an approach using coarse grained locking was adopted. More specifically, the approach was to use a single global lock known as the global interpreter lock (or GIL) to serialize execution of byte codes. The GIL has remained a major bottleneck when executing multi-threaded Python applications, in particular on multi-processor architectures. To illustrate the performance degradation imposed by Python's threading model, consider the simple application of multiplying two matrices. A master thread spawns a set of worker threads which calculate rows in parallel. When all rows have been calculated, the application terminates. The results from a Python version of parallel matrix multiplication are shown in Table 1. The application is run on an 8-way SMP machine with 2GB of RAM hosting 200MHz Pentium Pro processors, Linux 2.4.18 and Python2.2. Each Python thread runs 100 byte codes between each context switch[1].

The execution times are weighted, assigning a weight of 100 to the time used by a single-threaded implementation. The results in Table 1 show that when more threads are added, performance does not increase with the number of workers (and hence the number of CPUs in the system). Rather, performance decreases because of contention on the GIL. The degradation is most noticeable when a second thread is introduced, as there is no lock contention in a single-threaded application. Performance issues aside, threads in Python present an attractive programming model, since multiple threads can communicate e.g. by synchronized access to standard container types.

A common workaround to achieve parallelism on multi-processor architectures is to use multiple processes instead of threads. In such multi-process applications, inter-process communication is usually performed with shared memory or an ad-hoc messaging API using e.g. pipes. One drawback with using multiple processes in Python is that programming communication among multiple

[1] In Python2.3 this is the default number of byte codes executed per context switch.

Table 1: Multi-threaded matrix multiplication

Number of workers	1	2	3	4	5	6	7	8
Time used (weighted)	100.0	100.7	108.1	108.7	109.1	109.2	109.2	109.4

processes can be more complex than communication between threads using standard container types.

This paper presents POSH, which is an extension module to Python that attempts to address the above problems by enabling placement of Python objects in shared memory. POSH allows multiple processes to share objects in much the same way that threads do with standard Python objects. This following sections present the design, usage and performance evaluation of POSH.

2 Design Overview

The objective of POSH is to allow regular Python objects to reside in shared memory, where they can be made accessible to multiple concurrent processes. Objects allocated in shared memory are referred to as *shared objects*. For convenience, the types of shared objects are referred to as *shared types*, even though the actual type objects are not allocated in shared memory. Since type objects are essentially read-only data structures, the shared types can easily be made accessible to all processes by creating them prior to any `fork()` calls.

The process of creating a shared object that is a copy of a given non-shared object, is referred to as *sharing* the object. Objects that are eligible for sharing are called *shareable objects*, and their types are referred to as *shareable types*. POSH maintains a mapping that specifies the correspondence between shareable and shared types.

When an object is shared, its type is mapped to the corresponding shared type, and an instance of the shared type is created, which copies its initial value from the object being shared. The basic strategy for implementing a shared type is to subtype its equivalent shareable type, overriding its allocation methods. This allows the shared type to inherit the existing implementation details of the shareable type, while overriding the crucial aspect of object allocation.

An exception to this rule are the shared container types, which are reimplemented from scratch by POSH. Recursive data structures generally cannot be stored in shared memory using regular pointers, since shared memory regions can be attached at different virtual addresses in different processes. POSH uses an alternative abstraction, *memory handles*, to refer to locations in shared memory in a manner that is unambiguous to all processes. Shareable versions of the standard container types are implemented using this abstraction. Section 2.1 goes into more detail about memory handles.

For objects that are accessible to multiple concurrent processes, the regular garbage collection algorithm used by Python is inadequate. POSH implements a separate garbage collection algorithm for shared objects. In the event of abnormal process terminations, the algorithm allows the reference counts of shared objects to be corrected accordingly. Since the Python interpreter is unaware of the garbage collection algorithm used for shared objects, POSH wraps all shared objects in special *proxy objects* to shield them from direct reference. Section 2.2 describes proxy objects and the garbage collection algorithm implemented by POSH.

Shared objects are never made directly accessible to Python code; they are always accessed by means of their proxy objects. This provides a single access point to shared objects, which allows POSH to enforce synchronization policies for shared objects in a transparent manner. For instance,

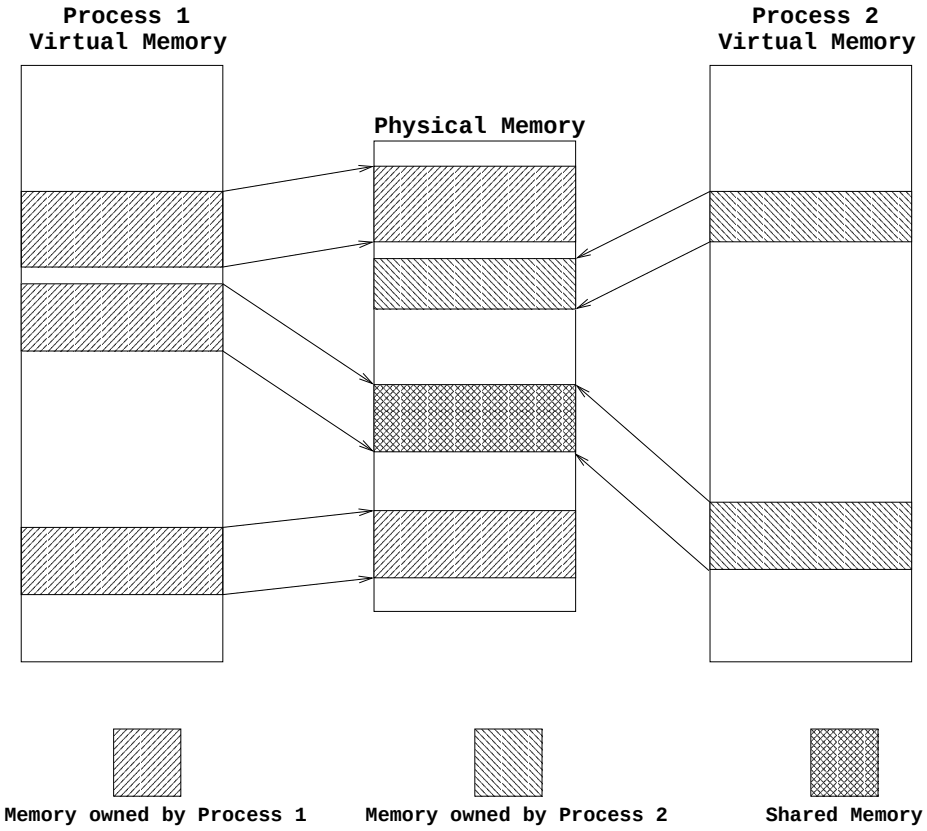

Figure 1: Typical implementation of shared memory.

monitor access semantics can be enforced implicitly whenever a shared object is accessed. Section 2.3 describes how POSH handles synchronization.

2.1 Memory Handles

Figure 1 shows a conceptual view of how most operating systems implement shared memory. Each process has its own virtual address space, which is typically larger than the physical address space. Certain regions of the virtual address space are mapped to regions of physical memory, while other regions remain unused.[2] The operating system, in conjunction with the hardware architecture, typically implements such mappings by means of a separate *page table* per process. Most regions of physical memory are owned by a single process, and only mapped into the virtual address space of that process. However, a region of *shared memory* may be created if the operating systems maps a region of physical memory into the virtual address spaces of several different processes. This enables all the processes to access the same region of physical memory.

The fact that each process has its own virtual address space, with a mapping defined by a separate page table, has a consequence that is not immediately obvious. It implies that the virtual address at which a given shared memory region starts may in fact vary from process to process. Therefore, a location in shared memory cannot be uniquely identified by means of a virtual address, since the virtual address may refer to different physical addresses in different processes. From a

[2] Virtual memory may also be backed by secondary storage, a fact which is ignored here for simplicity of exposition.

programmer's point of view, this means that a pointer (which is simply a virtual address), cannot uniquely identify a location in shared memory to a set of multiple processes. Consequently, recursive data structures, such as trees or linked lists, cannot be represented in shared memory using pointers.

POSH deals with this limitation by introducing the concept of *memory handles*. POSH assigns a unique identifier to each shared memory region upon creation. A memory handle consists of the identifier for a shared memory region, and a byte offset in that region. This information uniquely identifies the memory location to all processes, regardless of the actual virtual address at which the region starts. Each process maintains a separate table that stores the starting virtual address (in that process) of every shared memory region. When a process needs to access the memory location to which a memory handle refers, the actual address is found by looking up the starting address in the table, and adding the given byte offset. Since the table is implemented as a simple array, indexed by the shared memory region's identifier, the lookup is a simple constant-time operation.

Python's standard container objects (dictionaries, lists and tuples) are examples of recursive data structures. Their object structures contain pointers that refer to other objects. For instance, a tuple object contains an array of pointers, where each pointer refers to an element of the tuple. As explained above, the use of pointers makes the standard container objects unsuited for placement in shared memory. By using memory handles in place of pointers, POSH implements separate shareable versions of the standard container types. This allows dictionaries, lists and tuples to be shared. The ability to create shared dictionaries also paves the road for supporting shared objects with attributes, as this is implemented by storing the attribute values in a shared dictionary.

2.2 Garbage Collection

POSH implements a separate algorithm for multi-process garbage collection of shared objects.

Python implements garbage collection through reference counting, maintaining a simple reference count for each object. Reference counts are updated using the `Py_INCREF` and `Py_DECREF` macros, which increment and decrement the count without any form of synchronization. Consequently, these macros will not work correctly for shared objects, which can be accessed concurrently by multiple processes. Furthermore, using a single reference count per object makes it impossible to avoid memory leaks when processes terminate abnormally, since no knowledge is maintained about the number of references any particular process holds to an object. A garbage collection algorithm for shared objects must be able to account for references from all live processes, and should be able to correct the reference counts of shared objects when processes terminate abnormally. POSH implements an alternative garbage collection algorithm for shared objects that meets these requirements. It builds on the observation that references to shared objects fall within one of two categories.

(A) References stored in a process' address space. This may be viewed as a reference leading from a process to the shared object.

(B) References stored in another shared object. This may be viewed as a reference leading from one shared object to another.

POSH ensures that every reference from a process to a shared object is wrapped in a special *proxy object*. A proxy object holds a single reference to a shared object, and allows transparent access to the shared object by forwarding all attribute accesses. A process may have any number of references to a given proxy object, but there is never more than one proxy object in a process for a given shared object. The problem of tracking references of type (A) is thus reduced to tracking all proxy objects in existence. POSH sets an upper limit to the number of processes that can be

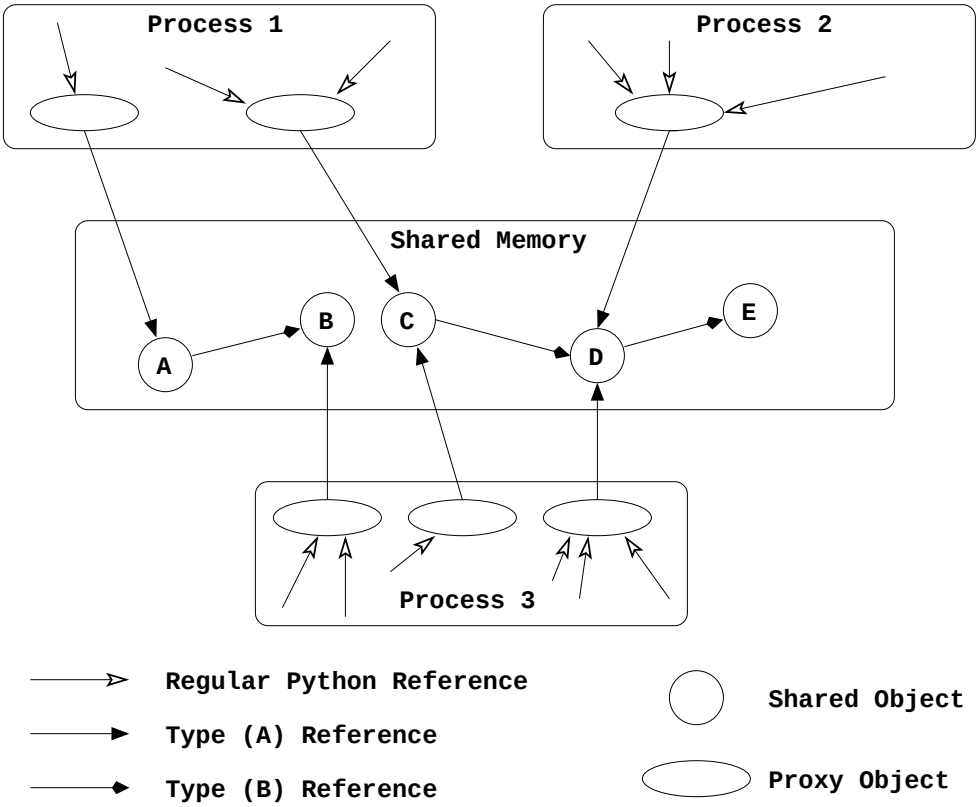

Figure 2: References between regular and shared objects.

created, and associates a bitmap with each shared object. The bitmap uses a single bit per process to indicate whether or not that process has a proxy object for the shared object. From another viewpoint, the bitmap represents a binary reference count per process.

Maintaining such a bitmap consequently accounts for all references of type (A) as defined above. To account for references of type (B), leading from one shared object to another, a separate counter is maintained. Together, the bitmap and the counter constitute the state that must be maintained for each shared object. When all the bits in the bitmap are cleared (indicating that no process has a proxy object for it), and the counter is 0, the shared object can be deleted.

Figure 2 shows an example in which three processes are sharing a total of five objects. In the figure, none of the processes have a proxy object for E, meaning that all the bits in E's bitmap are cleared. However, there is a reference from D to E, which is reflected in E's counter and prevents the object from being deleted. Conversely, there are two proxy objects for C, which means that two of the bits in the bitmap are set. There are no references to C from other shared objects, however, so once the two proxy objects are deleted, C will be deleted as well.

The normal reference count, `ob_refcnt`, has no meaning for shared objects, and is initialized to 2^{30}. This value is chosen so that C code that views shared objects as regular `PyObject`s can use the `Py_INCREF` and `Py_DECREF` macros without harm, as long as the operations performed are neutral with regard to the reference count. The unsynchronized access to the object's reference count leads to the theoretical possibility of *lost updates*, which could be a problem if the reference count reached 0 or overflowed. However, the probability of a lost update incorrectly modifying the reference count *in the same direction* a total of 2^{30} times is assumed to be so extremely low that the possibility can be disregarded.

2.3 Synchronization

POSH allows different synchronization policies to be applied to different shareable types, according to their synchronization requirements.

Since POSH allows multiple processes to access shared objects concurrently, synchronization may be required to preserve consistency. However, the synchronization requirements may vary from type to type. Immutable objects, such as strings and tuples, are essentially read-only data structures, and may safely be accessed concurrently. Mutable types, such as lists or dictionaries, require access synchronization to ensure that their internal data structures remain in a consistent state. The same goes for all user-defined types that support attribute assignment.

In POSH, the synchronization of access to a shared object is controlled by a special *policy object* which is associated with its type. When a shareable type is declared using `posh.allow_sharing()`, the policy object may be specified as an optional argument, and it is later accessible through the special `__synch__` attribute of the shared type. Whenever an operation is performed that requires access to a shared object, a call is made to the shared object's policy object, which may or may not decide to acquire a lock before allowing the operation to be performed. When the operation is complete, a similar call is made to the policy object, and it may release any locks that have been acquired. If the value `None` is specified for the policy object, unsynchronized access to the shared object is allowed. In addition, POSH defines a policy object that implements monitor access semantics, restricting access to the shared object to one process at a time. Further policy objects may be defined by the user, to provide customized behavior. For instance, a custom policy object could log all method calls that are made to a given shared object, for later reference.

2.4 Restrictions on Shared Objects

In general, almost any kind of Python object can be shared between processes using POSH. However, there are a few restrictions that shareable types should obey.

- Shareable types should not override the low-level memory allocation methods `tp_alloc` and `tp_free`. These are low-level methods that can only be overridden by extension types implemented in C. Their task is to manage the allocation and deallocation of the memory required for the type's instances. POSH relies on overriding these methods in order to allocate the objects in shared memory.

- Shareable types may not override the attribute lookup methods `__getattribute__`, `__setattr__` or `__delattr__`. POSH implements attributes for shared objects by associating a shared dictionary with each shared object that supports attributes. The dictionary provides storage for the object's attributes and is analogous to the `__dict__` attribute of regular objects. POSH overrides the attribute lookup methods of shareable types with versions that employ the shared dictionary. Shareable types may still provide a custom `__getattr__` method, which is a fall-back method that gets invoked whenever an `AttributeError` would normally be raised. In addition, shareable types retain the capability to customize their attributes with special attribute descriptors.

- A related restriction, which will probably be lifted in later versions of POSH, is that shareable types cannot have a nonempty `__slots__` attribute. Future versions of POSH can allow this by means of special attribute descriptors, but the feature has not yet been implemented.

- Methods of shareable types should never store the `self` argument in any way that makes it persist beyond the lifetime of the method call. For instance, it should never be stored in a global variable. The reason for this is that the `self` argument refers directly to the shared object, and not to a proxy object, which is normally the case when shared objects are accessed. Without the protection of a proxy object, the lifetime of the shared object cannot be guaranteed beyond the duration of the method call. This is because proxy objects play a vital part in the garbage collection algorithm implemented by POSH, in keeping track of the existing references to shared objects. If the `self` argument were to be stored in a global variable, the garbage collection algorithm would lose track of the shared object and might delete it prematurely.

  If the `self` argument needs to be stored for later reference, the value stored should be `posh.share(self)`, which wraps the object in a proxy object. Given this small inconvenience, one might wonder why POSH does not automatically wrap the `self` argument in a proxy object. The answer is simple — the `self` argument should always be an instance of the class to which a method belongs. The proxy objects created by POSH are all instances of a common type, and passing a proxy object in place of `self` would thus violate the semantics of the language, in addition to producing a number of odd side effects.

Most of these restrictions are pretty clear, and do not pose significant limitations in practice. However, the last point is rather subtle, and requires a certain vigilance on the programmer's part to avoid. POSH is able to issue warnings if a type is violating any of the other restrictions, but enforcing the last one remains up to the programmer.

```
import posh

l = posh.share(range(3))
if posh.fork():
    # Parent process
    l.append(3)
    posh.waitall()
else:
    # Child process
    l.append(4)                                                    10
    posh.exit(0)
print l

-- Output --
[0, 1, 2, 3, 4]
  -- OR --
[0, 1, 2, 4, 3]
```

Figure 3: Two processes accessing a shared list.

3 Example Usage

POSH provides functions for declaring shareable types and sharing objects, in addition to some utility functions for process management. Of particular importance is the version of `fork()` provided by POSH, which should always be used for process creation when POSH is used. The function has the exact same semantics as the standard `os.fork()`, but performs some initialization tasks that are required by POSH. In addition, POSH defines the convenience functions `posh.forkcall()`, which executes a function call in a child process, and `posh.waitall()`, which waits for the completion of all child processes.

3.1 Basic Usage

The idiom for sharing an object `x` with POSH is simply `x = posh.share(x)`. This creates a shared object whose initial value is copied from the regular object. The shared object will compare equal to the original object. In other words, `x == posh.share(x)` will always be true (provided `x` is shareable).

Besides this form of explicit sharing, objects can also be shared implicitly by assigning them to a shared container object. If a non-shareable object is assigned to a shared container object, an exception is raised.

Since shared types generally inherit their implementations from their shareable counterparts, the result of an operation involving a shared object will generally be a non-shared object. For instance, the result of multiplying two shared integers will be a regular `int` object. However, since the value is implicitly shared if it is reassigned to a shared container object, this normally behaves as expected. For example,

```
d = posh.share({'order': 'spam '})
d['order'] *= 4
```

```
import posh, random

class Parrot(object):
    def __init__(self):
        self.vocabulary = ["spam", "eggs"]

    def learn_word(self, word):
        self.vocabulary.append(word)

    def speak(self):
        print random.choice(self.vocabulary)

posh.allow_sharing(Parrot, posh.generic_init)
```

Figure 4: Definition of a shareable `Parrot` type.

```
>>> import posh
>>> from Parrot import Parrot
>>> p = posh.share(Parrot())
>>> p.learn_word("bacon")
>>> p.speak()
bacon
>>> p.speak()
eggs
>>> p.speak()
bacon
>>> p.speak()
spam
>>> type(p)
<class 'posh._proxy.SharedParrotProxy'>
```

Figure 5: Interpreter session using a shared `Parrot` object.

will behave as expected. The value `'spam spam spam spam '` is computed as a regular string, which is subsequently shared when reassigned to the shared dictionary.

When new processes are created using `posh.fork()`, objects that are shared will not be duplicated. Rather, they remain accessible to both the parent and child processes. The processes can perform inter-process communication simply by modifying the contents of shared container objects.

All of the standard types for which it makes sense, are shareable. This includes the standard int, long, float, complex, str, unicode, list, tuple and dict types. Figure 3 shows a small example that creates a shared list, forks, and appends one item to the list from both the parent and the child process. The output may vary depending on the way the processes are scheduled, but the consistency of the shared list is always maintained by the implicit synchronization enforced by POSH.

Table 2: Matrix multiplication performance.

Number of workers	Threads	POSH	Weighted
1	433.4	439.2	101.3
2	466.5	225.3	52.0
3	468.3	155.9	36.0
4	471.0	116.6	26.9
5	472.8	94.1	21.7
6	473.2	80.4	18.6
7	473.4	69.2	16.0
8	474.2	61.7	14.2

3.2 Using POSH with User-Defined Types

In addition to the standard Python types, types defined by the user (using the class statement) can also be made shareable. These types have to be registered with POSH using the `posh.allow_sharing` function, which will create an appropriate shared type through sub-typing. User-defined shareable types have to be registered prior to any `posh.fork()` calls, in order for the shared types to remain accessible to all processes.

For instance, Figure 4 shows the definition of a new `Parrot` type whose instances will be shareable. There is nothing special about the type definition in itself — the only point of interest is the call to `allow_sharing` following the class statement. The arguments to `allow_sharing` are the shareable type and an optional *initializer function*. The latter is a function that specifies how to create a new instance of the shared type given an instance of the shareable type. In this particular example, the function specifies how to create a shared `Parrot` object from an existing non-shared `Parrot` object. The function used in the example, `posh.generic_init`, will be appropriate for many user-defined types. It assigns all the attributes of the non-shared object to a new shared object. The user may freely define new initializer functions as needed.

An interpreter session that illustrates the use of a shared `Parrot` object is listed in Figure 5. The final line reveals that the object, while behaving exactly like a `Parrot` object, is in fact a proxy object for a shared object. The attributes that are assigned to shared objects are implicitly shared, just like the items assigned to shared lists and dictionaries.

4 Performance

Table 2 shows the execution times in seconds for running a matrix multiplication application with two input matrices of size 200x200. The application is run on an 8-way SMP machine with 2GB of RAM hosting 200MHz Pentium Pro processors, Linux 2.4.18 and Python2.2. Each Python thread runs 100 byte codes between each context switch.

The last column shows the results for POSH as they appear when a weight of 100 is assigned to the execution time of a single-threaded calculation.

When using only 1 worker POSH performs slightly worse than threads. This is caused by the general overhead associated with operations on shared objects. Each computed value in the result matrix has to be copied once in order to share it, which involves the allocation of a new object, and may require the creation of a new shared memory region. The application represents the matrices

as lists of rows, where each row is itself a list. As noted, complications inherent with shared memory forces POSH to reimplement shared lists using the abstraction of memory handles, which inevitably leads to a less efficient implementation. In addition, shared objects are accessed via proxy objects, which adds a level of indirection to the operations and thus degrades performance.

However, POSH achieves the expected scalability on multiple processors, since the computations are performed by separate processes. In this particular application, POSH clearly outperforms threads when more than 1 worker is employed.

The full source code of the matrix multiplication application is available in the POSH source distribution, as noted in Section 6. A point of interest is that the code executed by a single worker is identical, regardless of whether the worker is a thread or a process. This ensures a fair comparison of execution times, and also demonstrates the great degree of transparency offered by POSH.

5 Conclusions

Threads provide a useful abstraction for programming CPU-intensive parallel algorithms and master-worker computations. In Python, threads can easily communicate using synchronized access to standard container objects. However, threads in Python lack scalability for CPU-intensive applications because of the global interpreter lock. This paper has presented an approach called POSH which retains much of the programming model associated with threads but by means of processes and objects in shared memory. The ability to use standard Python objects in shared memory simplifies communication among multiple processes. However, accessing shared objects with POSH incurs some overhead compared to accessing native Python objects, but this overhead is usually subsumed by the increased parallelism when running on more than one processor.

6 Availability

POSH is an open source project hosted at Sourceforge.net[3]. The matrix multiplication application is located in the examples folder of the POSH CVS tree[4].

7 Acknowledgments

The authors would like to thank Fast Search & Transfer ASA[5] for their support. The project is partly supported by NSF (Norway) grant No. 112578/431 (DITS program) and the Norwegian Research Council (Norges Forskningsråd) grant No. 152956/431 (IKT-2010 program).

[3] http://poshmodule.sourceforge.net/
[4] http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/poshmodule
[5] http://www.fast.no/