# Title: Taking advantage of multiple CPUs for games - simply.


# Abstract:

Taking advantage of multiple CPUs for games --- simply, is the topic of this paper. Using a simple interface that many are already familiar with --- Pythons 'map' function. It talks about how to avoid the Global Interpreter Lock (GIL) limitation in Pythons threading libraries. As well as how to choose which parts of a game to thread. Finally it shows how easy it can be, by converting a full pygame to make use of multiple CPUs --- and benchmarking the game.


## *What is it about?*

Taking advantages of dual core, and multiple cpu systems.

This paper addresses CPython, and pygame written in C. Not the ironpython .NET version of python, or the Jython which is written in Java, or the Pypy which is written in python. Neither does it address pygamectypes, which uses ctypes rather than C code. This article concentrates on games using SDL software surfaces. SDL software surfaces are by far the most common type used from pygame. This article does not address OpenGL, or using SDL hardware surfaces. However you can apply some of this information to other types of games made with python.

## *What will you learn?*

You will learn how to take advantage of those extra cores/CPUs, so that your games will run quicker and smoother.

Goals of this article:

- Easy to use.
- Efficient to run.
- Slot into existing games easily.

Included code:

- threadmap.py.

There is a package included which makes using multiple cpus in your game easy.

If you know how to use the python 'map' function then you can use threads. tmap(func, data) is like map(func, data) except it uses a pool of worker threads to do its work. I also discuss how to convert some 'for loops' into using threads.

## *Different ways to take advantage of multiple CPUs.*

- Multiple threads.
- Multiple processes.

With python it can be more efficient to use multiple processes. Since python has the GIL lock. However luckily we can get around it, since many functions written in C do not need to worry about the GIL.

This is why we will use threading in this article - to take advantage of the multiple processors.

Threads allow you to access shared memory easily. Multiple processes can also access shared memory. However with these methods are not as easily portable as using threads. Using mmap or shared memory is also harder to use with external libraries like SDL. Even sharing normal python objects is hard with mmap and shared memory. So using threads are the simplest way to use multiple CPUs, sharing memory in python - in a portable manner.

## *The approach we will take.*

Make threads easier to use with a familiar api, and use libraries that people can reuse.

The blit, display, loading, and drawing functions are parts which are slow in most games - and can be sped up with multiple cores/CPUs. By threading these most common parts, most games will be able to take advantage of multiple CPUs/cores - as the work of threading these parts will have been done.

Time consuming parts in a game:

- loading resources. sounds, images.
- pre processing. Like drawing maps, generating visuals, or sounds.
- blitting. Drawing images to the screen.
- processing images. Scaling images, rotating images, tinting them etc.

Giving a programmer a simpler API for taking advantage of threads will allow people to use them more easily - without as much knowledge of threads. Using an existing api (the python map function) means that people will already know how to use threads if they know how to use map. I also show how to change an existing for loop so that it can use threads fairly easily.

Pygame already makes use of threads for some parts, which are hidden from the programmer. In the future, more parts of pygame will make use of threads so that the programmer will not need to know they exist. This way a programmer can use threads without even knowing it.

Making C modules which can be used from multiple threads is some times your only choice - if existing code isn't already set up to use threads. This is why we need to make more parts of pygame release the GIL in appropriate parts.

# Explanation of the threadmap module.

threadmap is the module which this paper talks about using.

threadmap is still a work in progress - with much further work to be done on it. It will be released at the cheese shop at some point, and probably included with pygame. It is licenced with the python licence - so if it proves useful more generally it could be put into python. At the moment it requires the python2.5 version of Queue - however future versions will also be able to work with python 2.3 and python 2.4. I guess this is version 0.2, an alpha version that is almost functionally complete.

threadmap comes in a few parts.

A WorkerQueue, a tmap function, and a benchmarking framework.

The WorkerQueue class allows you to manage different queues of tasks to do with a set number of threads. Unlike other thread pool classes it takes separate functions to work on, rather than working on data with one function. It supports optional call backs, and errbacks - but does not force their use.

The tmap function uses a worker queue to do it's work. The t in tmap stands for threaded - ie threaded map.

It still needs testing on a wider variety of machines, under different circumstances.

More information on using the API is below. However as a short demonstration at it's most simple you can do:
```
from threadmap import tmap
results = tmap(func, data)
```

# Is it even worth using multiple CPUs/Cores?

Many computers don't even have multiple cores, or CPUs yet. However they are becoming more common - and it has been on processor manufacturers(Intel, AMD etc) road map for quite a while. Even quite a lot of laptops come with dual core CPUs these days.

Even smaller portable devices like the GP2X, and PSP use multiple cores, as well as game machines like the new xbox.

But the thing is these new multiple core machines are so much faster than old machines. Even a single one of their cores is probably faster than many of the older machines around. When optimizing you often want to make things run faster on these old machines. Since the new machines can often run your game fast enough. So if you want to optimize things, there's probably other areas you can optimize first - so that more of your game players can take advantage of the optimizations.

However as every game programmer knows, you can always make use of extra CPU - usually for displaying more things on screen. As multiple Cores are going to get even more common, then if it's fairly easy to do - then you may as well take advantage of them.

Even considering that multiple cores are available to use - some times it's not more efficient to use them. Some things are limited by IO, or by memory bandwidth - so using threads might not help you at all in those situations.

## *Python and threading.*

Python uses a Global lock called the Global Interpreter Lock GIL in order to allow only one part of python to access the python api at a time. This is because portions of python are not thread safe.

The portions that are thread safe release this lock in order for threading to be done.

Unfortunately this means that a lot of code is limited to run at the one time.

However!!! Don't dismay, as pygame releases this GIL. Allowing processing to take place in the parts which don't care about the python API.

Here is the document on python threads.
http://docs.python.org/api/threads.html

With stackless python there are things called microthreads. Microthreads don't take advantage of multiple CPUs by themselves. Microthreads run within the same thread.

IronPython threading does not have a GIL. So threading with IronPython is more easily able to take advantage of multiple CPUs.

## Atomic ints, and lock free programming.

Another issue that python has with threading is that you can not use atomic ints from within python.

This page discusses the use of atomic ints as 'Interlocked operations'.
http://www.gamasutra.com/features/20060630/paquet_01.shtml

You can however use these from within C inside of python modules.

The python C API doesn't have a portable atomic int API. However SDL has been working on one - and it is available for many platforms already.

This paper doesn't discuss the use of atomic ints for threading any further.

## *Pygame, and SDL are already threaded.*

Parts of pygame, and SDL have always been threaded. Sound and music already run in a separate thread. This is for latency issues. You need to keep feeding sound to the sound card at regular intervals, otherwise it crackles or skips. Since sound requires a different frequency of updating than video or the game logic.

Other parts of pygame drop the GIL so that they can be done in separate threads. Like the image loading, and pygame.transform.scale functions.

Loading images in the background can be done with different threads. Some operating systems are better at combining read requests than others. Eg linux can optimize multiple read requests a lot better

than win98 can. So one thread reading per disk is probably a fairly good way to go. With other threads decoding and decompressing the images.

### SDL Threading, and locking

SDL has it's own portable threading, and locking mechanisms. SDL threads can be in use at the same time as python ones.

You can use Mutexes, and Semaphores from SDL.

SDL surfaces can be locked for direct access. This is to support hardware surfaces. For software surfaces there is a check to see if the surface requires locking at all: `SDL_MUSTLOCK(surface)`

Only when coding in C with SDL do we use the SDL threads. Since python already has their own portable threading implementation.

## Choosing which parts of a game to thread.

One of the hardest parts of using threading is coming up with techniques which work correctly without increasing the chance of errors, or weird bugs.

You usually can not render multiple frames ahead in advance with games, because things can change very quickly.

### *The slow parts - blit, flip, update.*

When optimizing anything, the first thing you do is profile your code. You find out which bits are slow.

Python comes with a profiling module which allows you to see which functions are the slowest. This way you can see which functions need profiling.

Another way is to measure the frames per second, or FPS. Measuring the time per frame is also useful, so is measuring individual parts of a frame. Eg, measure the time to display, the time handling events and updating logic.

If your game is running fast enough on your target hardware, then your job is done.

However with pygame games the slow parts are mostly the same in every game. That is the blit function calls, and anything which modifies surfaces - like the draw calls, and the transform calls(rotation and scaling images).

Blit functions are written in optimized C, and assembly code utilizing mmx or altivec they often take up a good portion of game time. Drawing to the screen can take a while.

So these are the parts we will concentrate on.

## *The dependant parts.*

For making a program parallel, one thing you need to do is to see which parts are dependant on which other parts.

If there are multiple parts which are not dependant on each other, then you can run those parts in separate threads really easily. You don't need to lock anything. Otherwise you might need to use some method to allow sharing the data - like using locking.

Something which might be cool - would be to use code coverage analysis to find dependant parts. If the code coverage finds that some parts are not dependant on each other, then maybe you can automatically make them parallel.

## *Data parallel parts.*

With games, you often have to do one function on many parts of data at once. Either you might need to draw many separate images to the screen, or you may need to load many different things at once.

You split the data up amongst the available threads, and let them work at it.

## *C modules, and threading.*

For a C module the simple way to avoid the GIL is like this:

```
Py_BEGIN_ALLOW_THREADS
stretch(surf, newsurf);
Py_END_ALLOW_THREADS
```

Since the stretch function shown here is entirely C code, and doesn't need to call any python code the two macros used are fine. However if you need to call back into python code then it gets a bit trickier.

This approach will be to used try and take advantage of threads for the more expensive pygame calls.

## *Checking to see if Python, or Pygame methods release the GIL.*

Unfortunately you need to check the source code for each function to determine this.

In pygame all the functions are in the src/ subdirectory.

Prior to pygame 1.9(unreleased as of writing) there were less functions which released the GIL. In the future the pygame documentation will note the thread safety, and if it releases the GIL or not.

Also with python functions you need to check the source.

# An example of API use for simpler threading.

There are many different ways to use threaded APIs in python. Below I present some ways which make

it simpler. It takes the approach of data parallelism being the easiest, and often most beneficial things to speed up.

Another approach to make things simpler is to copy the API of existing python idioms. The two which I present below are using map, and using a for loop.

You can find the code used at this address:
http://rene.f0o.com/~rene/stuff/europython2007/threads_article/

The threadmap.py code can be found here:
http://rene.f0o.com/~rene/stuff/europython2007/threads_article/threadmap.py

It will eventually be moved into the python package index (cheeseshop) at some point.

### *Map based api for threading.*

I have created a tmap function which uses a pool of worker threads in the background to complete the tasks.

This is how it can be used:

```
import threadmap
threadmap.init()

results = tmap(func, data)
```

### *For loop based api for threading.*

Since a lot of code uses a for loop rather than a map function call, here's an api for using a for loop with threads. So it's quite simple to translate an existing for loop into something that uses threads.

Translating a for loop to use worker threads:

```
wq = threadmap.WorkingQueue()

for d in data:
    wq.do(func, d)

wq.wait()
```

## Figuring out how many threads to use.

The number of threads you should use depends greatly on the type of task you are doing, and also on the platform you are using.

Some platforms have more expensive threads, and others have very lightweight threads. Old versions of linux had fairly bad threads. In current versions of linux you can have many threads running at once

with a smaller overhead.

To take advantage of multiple CPUs you'll need at least one thread for each CPU or CPU core. The main thread is counted in this count of threads.

## *Find out how many threads to use with a mini benchmark.*

Rather than deal with platform specific detection of the number of cores, it could be more robust to actually test if using multiple threads will be more efficient. This way figuring out the number of CPUs will most likely work on unknown platforms.

In threadmap.py there is a little benchmark which gets called to see if running multiple threads is more efficient. It also tries to figure out the optimal number of threads to use. This benchmark is only called once, at module threadmap.init() time. Also the module will work without running the benchmark.

The benchmark function in threadmap takes some arguments so that you can provide your own implementation - for testing some of your own games code. This way you could use separate amounts of workers for different tasks in your game.

## *Checking to see if multiple CPUs are being used*

The easiest way is to use platform specific tools whilst the program is running. On linux, *bsd, and Mac OSX the 'top' tool will show how much of each cpu is being used. On Windows there is the 'Windows Task Manager'. ctrl+alt+delete will bring it up, then click on the performance tab.

## *Platform differences.*

Threads are different on each platform. Even different versions of the same platform have different threading implementations. Threads in linux 2.4 are quite different to 2.6. Even later versions of linux 2.4 have different types of threads to earlier linux 2.4 implementations.

Some platforms can handle creating threads really quickly. Other platforms take a long time to create threads, and have limits on the number of threads you can create.

The different memory speeds, and bandwidth usage of your code can mean that on some platforms you get almost no speed up with more threads, and a 2x or more speed up on other platforms.

Python can even be complied with no thread support compiled in.

# Testing, and debugging threading problems.

Threads are notoriously hard to use. In the python world 'threads are evil' is often written, along with 'threads are considered harmful'.

The main reason for this is because threaded programs are harder to write correctly, and harder to debug.

So making threaded programs easier to write, and easier to debug is something that needs to be done in order for them to be used more easily.

So we need different techniques and for debugging and avoiding thread specific bugs.

The GIL in python helps to prevent some threading bugs, but not all. You can still get deadlocks and other threading problems.

## Encapsulating threaded code in modules.

One technique for avoiding thread bugs is to make use of threads via well tested library code. So library users do not need to understand how threads work at all. For example the sound code in pygame uses threads in the background, but the person using the library doesn't even have to thing about using threads.

By keeping the code in a separate module, it is also easier to test compared to having the threads mixed in all over the place in your program. You are able to use unittesting, and other testing techniques.

## Serialising your threaded program - a debugging technique.

If your program has a bug, then it can often be simpler to stop using threads whilst you debug. This simplifies the debugging, since debugging with threads is harder.

If your program does not have a bug in single threaded mode, then it is likely a thread specific bug.

In order to serialise your program your threading api needs to be able to go down to using just one thread.

The threadmap module allows you to use any number of worker threads. Which makes debugging easier.

## *More testing is required with threaded programs.*

Threaded programs require more testing because there are many different threading implementations, and many different platforms. They can also create more complex non determinant programs.

Creating unittests for separate sections of threaded code allows you to keep the amount of testing down. Since you need to test on multiple platforms anyway, creating unittests to make this testing simpler is a good idea.

To get the optimal performance you also need to use benchmarks to see if threading things on one platform is faster than another.

### Testing on Multiprocessor hardware is required.

You can test many parts of threaded programs on a single CPU system. However threaded behaviour on a multiprocessor system is different to threaded behaviour on a single processor system.

If your program works fine on a single processor system, but not on a multiprocessor system, then it is likely a thread bug.

### Processor affinity.

Even single threaded programs can have problems with multiple processor systems. These bugs happen when the program is moved between processors. Often these are timing related bugs, as getting the current time from each CPU may give a different result. Since games are so dependant on timing, these bugs show up more in games than other types of software. Setting a program to run on one processor can help solve these bugs.

As a debugging technique, it can sometimes help to set the application to stick to using just one CPU. If that solves the problem, then it is likely a process affinity type of bug.

## Converting a pygame game to take advantage of threads.

Below are some methods for converting your game to use threads. First there is a fairly simple example. Then I show how to convert a full game - the aliens example game that comes with pygame. I make two parts optionally use threads - the sprite drawing code, and the image loading code.

### *A simple example for using extra threads.*

Just say you needed to rotate a bunch of surfaces every frame.

```
import threadmap
threadmap.init()
tmap = threadmap.tmap

def scaleit(surf):
    return pygame.transform.scale(x, (44, 76))

scaled_surfs = tmap(scaleit, surfs)
```

Tada! that's it. It automatically finds out the optimum number of threads to use, starts up a pool of worker threads. Then when you call tmap(scaleit, surfs) it uses the number of threads detected. If you're on a single cpu system, then it doesn't use any extra threads at all.

### *Converting a full game to use threads - RenderUpdates.*

Pygame has a sprite module which tries to make it easier for people to manage their drawing of images.

Below I override the RenderUpdates.draw method so that it uses threads for processing.

The full example can be found at:
http://rene.f0o.com/~rene/stuff/europython2007/threads_article/aliens_multicpu.py

```
USE_THREADS = 1
if USE_THREADS:
    import threadmap
    threadmap.init()
    tmap = threadmap.tmap
else:
    tmap = map


class RenderUpdatesThreaded(pygame.sprite.RenderUpdates):

    def draw(self, surface):
        # recoded draw function to use threads.
        self.dirty = self.lostsprites
        self.lostsprites = []
        self.surface = surface
        tmap(self.draw_one, self.sprites())

        return self.dirty

    def draw_one(self, s):
        dirty_append = self.dirty.append
        surface = self.surface


        r = self.spritedict[s]
        newrect = surface.blit(s.image, s.rect)
        if r is 0:
            dirty_append(newrect)
        else:
            if newrect.colliderect(r):
                dirty_append(newrect.union(r))
            else:
                dirty_append(newrect)
                dirty_append(r)
        self.spritedict[s] = newrect
```

Then that's it!  For other games that use sprites and the RenderUpdates class you can just use this implementation instead.  Or it even might be in pygame one year.

## *Converting a full game to use threads - image loading.*

Here we use a

Note, when timing this, if the images are already in cache, then threads will only be useful if you have multiple CPUs/cores.

```
cached_images = {}

# here we use tmap to load them with the configured amount of threads.
```

```
def load_all_images(*files):
    surfs = tmap(load_image, files)
    r = {}
    for f, surf in zip(files, surfs):
        r[f] = surf

    return r

# these are the old image loading functions, except they also place surfaces in a cache.
def load_image(file, cached = 1):
    "loads an image, prepares it for play"
    if cached:
        if cached_images.has_key(file):
            return cached_images[file]

    file = os.path.join('data', file)
    try:
        surface = pygame.image.load(file)
    except pygame.error:
        raise SystemExit, 'Could not load image "%s" %s'%(file, pygame.get_error())
    return surface.convert()

def load_images(*files):
    imgs = []
    for file in files:
        imgs.append(load_image(file))
    return imgs



# now at the start of the image loading part
cached_images = load_all_images('player1.gif', 'explosion1.gif', 'alien1.gif', 'alien2.gif', 'alien3.gif',
'bomb.gif', 'shot.gif', 'background.gif')
```

Tada!  That uses separate threads to load each image.

# Future work on threadmap

The selection of the number of worker threads to use could be improved.  The benchmark to find out how many workers to use could try and make this scale better with slower/faster cpus.  It could first find some variables so that using 0 workers takes about 1.0 seconds.  Otherwise when computers get 10x faster again, this code will stop working.

tmap also needs to try and follow map semantics more closely with regards to exceptions.  As well as trying to pass as much of the map unittests as possible.

## *Different handling of data for IO bound, and CPU bound functions.*

The way the data, and functions are divided up into the tmap function could be improved for CPU bound functions.  Rather than pass one function/data pair into a thread at a time, we could divide the functions/data into equal amounts to be shared between the number of worker threads.  This would reduce the contention on the main Queue that the WorkerQueue uses(and therefore less locking) - as

well as meaning fewer function calls.

The current method is more optimized for operations which could take varying amounts of time - like IO operations.  Since if you divided the work up equally between the worker threads some times there would be spare worker threads hanging around doing nothing.  This is because one worker thread might get all of the quick to load things.  For example dividing url downloading - each url could take from 0.20 ms to 10 hours to download.  So if one thread got all the 0.20 ms downloads, then it'd be finished far quicker compared to the threads which got 10 hour downloads.