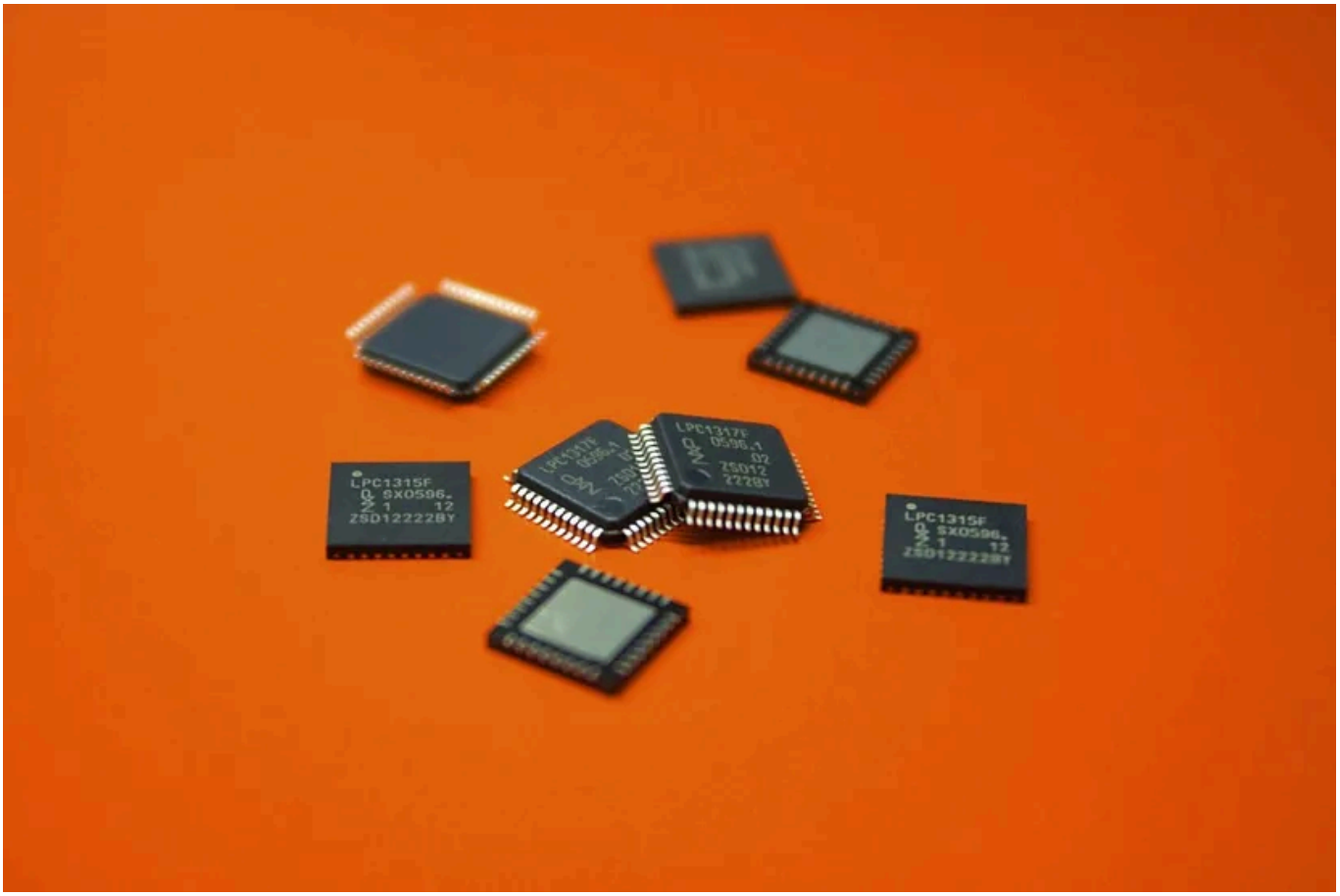


Think Like an Octopus (in Python)

An octopus can move its tentacles in parallel. Michael Ziegltrum shows different tactics to enable your Python programs to move similarly.

BlackRockEngineering · Follow
Published in BlackRock Engineering · 18 min read · 6 hours ago

*By: **Michael Ziegltrum**, CFA, Vice President in Core Portfolio Management Architecture team*

Pirates of the Caribbean 2: Dead Man's Chest had some memorable scenes. One that stood out featured the pirate captain Davy Jones playing his organ. Jones is part octopus-part man and in the scene his tentacles moved synchronously over the instrument playing chords. Each appendage worked together to deliver something bigger. What if our Python programs could do similarly?

In fixed income portfolio management, there are many interesting problems where we could apply these ideas. In exchange traded funds (ETFs) portfolio managers (PMs) often evaluate a list of orders and compare them to the portfolio. For example, a PM might look at a list of orders and see what percent of the bonds are from companies in the banking sector. She might compare this metric in the list of orders to the portfolio's allocation to banking. This is one metric, but the comparisons needed could be extensive. Hence the evaluation is a non-trivial piece of computation. As the scale increases via number of sectors, bonds, and portfolios, the time and memory usage to calculate metrics on an order list could be a problem. One idea is to evaluate multiple metrics at once like an octopus moving its tentacles in parallel. There are many different options towards this in Python including libraries like threading, multiprocessing, asyncio, ctypes, numpy, and more, each with varying trade-offs.

This post supplements an earlier post, "Citizen Developer Cookbook: Python Multiprocessing", from a colleague, Casey Clements, who wrote up some in-depth recipes focusing on aggregation, monitoring, and exception handling with multiprocessing. In this post, we'll ignore those important topics having been well-covered in Casey's excellent post.

Executive Summary

This blog post looks at various options for calculating the market value percent of a sector in an ETF across many funds and sectors. This includes: the Python threading API; the Python multiprocessing API; a second approach to using the Python multiprocessing API; writing the processing in C++ and calling from Python; and using numpy's vectorization. There are various trade-offs between these methods including simplicity, CPU and memory usage, runtime, and code safeness. Any production usage would need to weigh these trade-offs, but, in this toy problem, numpy is the winner.

Many of these trade-offs will not come as a surprise to savvy readers but in short: Python threading can only use one cpu; starting a new process using Python multiprocessing is heavy-weight and relatively slow vs starting a thread; we can use more than one cpu by writing C++ code and calling it from Python, with more complex and risky code; or we can leverage numpy which has its own C wrappers and vectorization that is probably better than what we would write.

Now, let's get into it!

Problem Setup

Let's say we want to write a program to take a list of orders and calculate how each sector's market value compares to a given target.

Let's model a portfolio and sector as per the below. This corresponds to a spreadsheet where each row is a bond and there are columns for market value percent and sector. If a bond is worth $5, in a portfolio valued at $100, that bond has a 5% market value percent. If that bond and another bond

with market value of 2% are the only two bonds from companies in the banking sector, the banking sector allocation in the portfolio is 7%. In a real portfolio we would probably need to store identifiers for the bonds as well as other pieces of info, but we'll ignore it in this toy example.

If you want a runnable script to follow along, a link is included at the bottom of this post.

```
@dataclass
class Portfolio:
    """
    Container class to keep track of market values weights and sectors for a por
    """
    market_value_pcts: "np.ndarray"
    sectors: "np.ndarray"


class Sector(Enum):
    """
    Enumeration to store sector information for bonds in a portfolio.
    """
    BANKING = 0
    BASIC_INDUSTRY = 1
```

```
    BROKERAGE_ASSET_MANAGERS = 2
    CAPITAL_GOODS = 3
    COMMUNICATIONS = 4
    CONSUMER_CYCLICAL = 5
    CONSUMER_NON_CYCLICAL = 6
    ELECTRIC = 7
    ENERGY = 8
    FINANCE_COMPANIES = 9
    FOREIGN_AGENCIES = 10
    INSURANCE = 11
    NATURAL_GAS = 12
    OTHER_INDUSTRY = 13
    OTHER_UTILITY = 14
    REIT = 15
    TECHNOLOGY = 16
    TRANSPORTATION = 17
```

We can instantiate some mock data with the below. Note that for each bond we store a 64-bit float and a 32-bit int. So, if we have 50,000 bonds, we store about 50,000 * (64 + 32) = 4,800,000 bits, or about 0.5MB. If we have 20 portfolios, this leads us to 10MB. On some Linux varieties, the limit to memory that we can use on the stack as local variables is 8MB, meaning for this data structure we would need to allocate memory from the heap. As such, this is a non-trivial chunk of data.

```
def prepare_mock_portfolio(num_bonds: int) -> Portfolio:
    """
    Helper function to prepare some fake portfolio data
    @param num_bonds: how many bonds to put in the fake portfolio
    @return:
    """
    market_value_pcts = np.random.uniform(size=(num_bonds,)).astype(np.float64)
    market_value_pcts /= np.sum(market_value_pcts)
    sectors = np.random.randint(low=0, high=len(Sector), size=(num_bonds,), dtyp
    portfolio = Portfolio(market_value_pcts, sectors)
    return portfolio
```

```
num_bonds = 50000
num_portfolios = 20
portfolios = []
for i in range(num_portfolios):
    portfolios.append(prepare_mock_portfolio(num_bonds))
```

Let's assume we want to calculate the sector weights vs targets for each portfolio, though in reality we may have more complex rules:

```
def log_tolerance(worker_num: int, sector: Sector, market_value_pct: float, targ
    print(
        f"worker: {worker_num} {sector.name}: {market_value_pct:.3f}, target_wei
        f"diff_from_tol: {(market_value_pct - target_weight):.3f}\n")


def calculate_sector_weights(worker_num: int, portfolio: Portfolio, target_weigh
    """
    Function to calculate how much market value is in each sector and log this f
    @param worker_num: which worker is responsible for the calculation
    @param portfolio: portfolio container
    @param target_weight: a target weight to compare against market value pct
    @return:
    """
    sector_weights = [0.0] * len(Sector)

    for i in range(portfolio.sectors.shape[0]):
        bond_sector = portfolio.sectors[i]
        sector_weights[bond_sector] += portfolio.market_value_pcts[i]

    for i in range(len(Sector)):
        sector = Sector(i)
        mv_pct = sector_weights[i]
        log_tolerance(worker_num, sector, mv_pct, target_weight)
    return
```

Examining Approaches

All of the below approaches were run on Linux (Debian bullseye) and Python 3.9.18. Example output looks like the below:

```
...
worker: 0 TECHNOLOGY: 0.054, target_weight: 0.056, diff_from_tol: -0.002
```

```
worker: 0 TRANSPORTATION: 0.055, target_weight: 0.056, diff_from_tol: -0.000
...
```

Iterative Approach

As we look at ways to solve this problem, the first thing that may come to mind is the iterative approach:

```
def iterative_calcs(portfolios: List[Portfolio], target_weight: float):
    """
    Iteratively calculate and log the market value weights for each sector in ea
    @param portfolios: list of Portfolio instances
    @param target_weight: target sector weight to compare actual weight to
    @return:
    """
    for pf in portfolios:
        calculate_sector_weights(0, pf, target_weight)
```

This runs in about 0.54 seconds and during the run the most virtual memory used was 474MB. The most CPU used was 100%. The relevant PID is 94 — I'm using a wrapper to run and profile these (PID 82 in the below). This wrapper starts up a subprocess to run the sector calculations according to a given methodology like threading or multiprocessing and it watches the resource usage as it does so.

```
top - 17:13:06 up  8:09,  0 users,  load average: 0.19, 0.08, 0.10
Tasks:   4 total,   2 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7.2 us,  2.8 sy,  0.0 ni, 90.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  25497.7 total,  23631.6 free,    698.1 used,   1167.9 buff/cache
MiB Swap:   7168.0 total,   7168.0 free,      0.0 used.  24521.0 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   94 root      20   0  474.0m  47.9m   5.9m R 100.0   0.2   0:00.47 python main.py
    1 root      20   0    5.9m   3.7m   3.2m S   0.0   0.0   0:00.03 /bin/bash
   82 root      20   0  459.6m  42.3m  14.8m S   0.0   0.2   0:01.12 python main.py
   97 root      20   0    8.5m   3.6m   3.2m R   0.0   0.0   0:00.00 top -c -b -n1 -E m -e m
```

Can we do better than this?

Threading Approach

In a lot of programming languages threads let us use do more than one thing at a time while sharing the same memory. Could we evaluate more than one rule at once using threading?

```
def threading_calcs(num_workers: int, portfolios: List[Portfolio], target_weight
    """
    Calculate and log the market value weights for each sector in each portfolio
    @param num_workers: how many threads to use at a given time. for example, 3.
    @param portfolios: list of Portfolio instances
    @param target_weight: target sector weight to compare actual weight to
    @return:
    """
    for pf_index in range(0, len(portfolios), num_workers):
        worker_list = []
        for thread_index in range(num_workers):
            pf_index = pf_index + thread_index
            if pf_index >= len(portfolios):
                continue
            args = (thread_index, portfolios[pf_index], target_weight)
            worker = threading.Thread(target=calculate_sector_weights, args=args
            worker_list.append(worker)
            worker.start()
        [worker.join() for worker in worker_list]
```

This runs in about 0.52 seconds and during the run the most virtual memory used was about 667MB. The most CPU used was 113%.

```
top - 17:13:06 up  8:09,  0 users,  load average: 0.19, 0.08, 0.10
Tasks:   4 total,   1 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7.7 us,  4.4 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.5 si,  0.0 st
MiB Mem :  25497.7 total,  23636.7 free,    693.0 used,   1168.0 buff/cache
MiB Swap:   7168.0 total,   7168.0 free,      0.0 used.  24526.0 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   99 root      20   0  666.7m  48.9m   6.3m S 113.3   0.2   0:00.34 python main.py
    1 root      20   0    5.9m   3.7m   3.2m S   0.0   0.0   0:00.03 /bin/bash
   82 root      20   0  459.6m  42.3m  14.8m S   0.0   0.2   0:01.13 python main.py
  107 root      20   0    8.5m   3.5m   3.1m R   0.0   0.0   0:00.00 top -c -b -n1 -E m -e m
```

Compared to the iterative approach we use 200MB more memory. We have improved run time by only 0.02 seconds, though. We may have expected to cut run time down by a third, given we use 3 workers. What gives?

The key is in our CPU usage which is also around 100%. If we were truly doing multiple things at a time, this would have been 300% (representing using 100% of 3 CPU cores). Python, as a language, though, has many built-in features to protect Python programmers from people like themselves.

One such mechanism is the global interpreter lock (GIL) which makes it so that only one thread can access the interpreter and do something at a given point in time. Many third-party libraries like numpy have C code that circumvents this, but if we are using pure Python we cannot get around this lock.

In future versions of Python, we may escape the GIL (see discussion here), but, for now, threading is not a good fit for CPU bound tasks like this one. It *may* have been a better fit if we had moved some of these calculations to a microservice that we could then call via an http or other call.

So, why was the CPU usage slightly over 100%? Probably the thread switching, which may use more than 1 CPU, but I leave this for discussion.

Multiprocessing Approach

There is another Python library that can do more than one thing at a time. The multiprocessing library starts up processes. Processes are heavier weight than threads and do not share memory space, so we will have to worry about passing data back and forth between processes. That said, the API was designed to be similar to the threading API so an initial implementation might look like the below:

```
def multiprocessing_calcs_bad(num_workers: int, portfolios: List[Portfolio], tar
    """

    @param num_workers: how many processes to use at a given time. for example,
    @param portfolios: list of Portfolio instances
    @param target_weight: target sector weight to compare actual weight to
    @return:
    """
    for pf_index in range(0, len(portfolios), num_workers):
        worker_list = []
        for thread_index in range(num_workers):
            pf_index = pf_index + thread_index
            if pf_index >= len(portfolios):
                continue
            args = (thread_index, portfolios[pf_index], target_weight)
            worker = multiprocessing.Process(target=calculate_sector_weights, ar
            worker_list.append(worker)
            worker.start()
        [worker.join() for worker in worker_list]
```

This runs in about 0.23 seconds, a bit better than half the run time of the iterative or threading approaches. Why is this? We can now do more than one thing at a time, like an octopus playing a church organ.

In the screenshot below we can see that we have spawned more Python processes, and each of those processes (PID 351, 352, 353) can use up to a full CPU. As an aside, in the screenshot we only see 6.7%. This is because I took the screenshot at the wrong time. In reality they probably used 100% CPU for a brief instant then exited.

```
top - 17:54:52 up  8:50,  0 users,  load average: 0.01, 0.02, 0.00
Tasks:   7 total,   4 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s): 19.9 us,  1.1 sy,  0.0 ni, 79.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  25497.7 total,  23616.9 free,    706.7 used,   1174.1 buff/cache
MiB Swap:   7168.0 total,   7168.0 free,      0.0 used.  24512.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  337 root      20   0  474.7m  53.6m   5.8m S  20.0   0.2   0:00.03 python main.py
  351 root      20   0  474.7m  50.2m   2.4m R   6.7   0.2   0:00.01 python main.py
  352 root      20   0  474.7m  50.2m   2.4m R   6.7   0.2   0:00.01 python main.py
  353 root      20   0  474.7m  50.2m   2.4m R   6.7   0.2   0:00.01 python main.py
    1 root      20   0    5.9m   3.7m   3.2m S   0.0   0.0   0:00.03 /bin/bash
  296 root      20   0  459.6m  47.1m  14.4m S   0.0   0.2   0:01.10 python main.py
  338 root      20   0    8.5m   3.5m   3.1m R   0.0   0.0   0:00.00 top -c -b -n1 -E m -e m
```

However, this is at the cost of using significantly more memory (perhaps 1.5 gigabytes more) than prior approaches because each process has to have a Python setup and that large mock data that we are working with.

Another drawback is that we again may have expected to cut down run time to a third of the iterative approach given we use three workers, but only cut it in half. What gives?

Well, starting up a process is an intensive thing, and in the above code we start up one process for each portfolio. Additionally, sending data to the new process like our portfolio list can take time.

Can we do better?

Multiprocessing Approach With Fixed Workers

With what I admit is a non-trivial amount of new code, we can bring up 3 workers and leave them up throughout the program, sending jobs to them. This should save process start up time. We also will copy the portfolio list just once to each of the three processes, rather than sending it once per portfolio. In this example I don't expect this latter bit to save a lot of time here but it could be more relevant for you depending on size and structure of your data.

```
@dataclass
class MPCalcSectorWeightArgs:
    """
    Container class for arguments to a multiprocessing worker to calculate portf
    """
    exit_flag: bool
    pf_index: int


def calc_sector_weights_wrapper(args_queue: Queue, worker_num: int, portfolios:
    """
    Helper function to bring up a worker and keep it up looking for new jobs to
    receives a job with an exit flag set in the queue.
    @param args_queue: a queue to poll for jobs or to exit
    @param worker_num: which worker this is
    @param portfolios: list of Portfolio instances
    @param target_weight: target sector weight to compare actual weight to
    @return:
    """
    exit_flag = False
    while not exit_flag:
        if args_queue.qsize() > 0:
            args = args_queue.get()
            pf_index = args.pf_index
            exit_flag = args.exit_flag
            if exit_flag:
                continue
            calculate_sector_weights(worker_num, portfolios[pf_index], target_we


def multiprocessing_calcs(num_workers: int, portfolios: List[Portfolio], target_
    """
    Calculate and log the market value weights for each sector in each portfolio
    keeping these processes up throughout the run to save on process start up ti
```

```
        @param num_workers: how many threads to use at a given time. for example, 3.
        @param portfolios: list of Portfolio instances
        @param target_weight: target sector weight to compare actual weight to
        """
        worker_list = []
        q_list = []
        for i in range(num_workers):
            q = multiprocessing.Queue()
            p = multiprocessing.Process(target=calc_sector_weights_wrapper, args=(q,
            worker_list.append(p)
            q_list.append(q)
        [p.start() for p in worker_list]
        for pf_index in range(0, len(portfolios)):
            args = MPCalcSectorWeightArgs(False, pf_index)
            worker_num = pf_index % num_workers
            q_list[worker_num].put(args)
        for worker_num in range(num_workers):
            args = MPCalcSectorWeightArgs(True, 0)
            q_list[worker_num].put(args)
        [worker.join() for worker in worker_list]
```

How did we do? Runtime is now around 0.20 seconds. We saved 13% versus the prior implementation. This is 0.03 seconds which may not be significant in the scheme of the 2–3 years of a typical blue-ringed octopus' life, but is a significant improvement in the computer world!

On memory, we did not improve nor did we expect to. With regards to CPU usage, as the processes below stay up longer than the processes in the prior example (given we only have 3 and we are sending jobs to them and keeping them alive), we see higher CPU usage. That said, this is only reflecting the fact that the processes are up longer and I have time to get the output from the top command.

```
top - 17:54:52 up  8:50,  0 users,  load average: 0.01, 0.02, 0.00
Tasks:   7 total,   4 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s): 22.1 us,  0.0 sy,  0.0 ni, 77.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  25497.7 total,  23618.4 free,    705.2 used,   1174.1 buff/cache
MiB Swap:   7168.0 total,   7168.0 free,      0.0 used.  24513.6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  362 root      20   0  474.7m  50.5m   2.7m R  80.0   0.2   0:00.12 python main.py
  363 root      20   0  474.7m  50.5m   2.7m R  80.0   0.2   0:00.12 python main.py
  364 root      20   0  474.7m  50.5m   2.7m R  73.3   0.2   0:00.11 python main.py
  360 root      20   0  667.0m  53.9m   6.1m S  20.0   0.2   0:00.03 python main.py
    1 root      20   0    5.9m   3.7m   3.2m S   0.0   0.0   0:00.03 /bin/bash
  296 root      20   0  459.6m  47.1m  14.4m S   0.0   0.2   0:01.11 python main.py
  361 root      20   0    8.5m   3.5m   3.1m R   0.0   0.0   0:00.00 top -c -b -n1 -E m -e m
```

If we wanted to push start up time more, the multiprocessing API supports other ways to pass data to processes and we might find that some are faster than others (like using shared memory).

C++ Wrappers and Recreating the Wheel

We are doing quite a lot of work here to get around Python's limitations on threading. Specifically, we are starting processes, passing data to them, and so on. But, Python has good integration with C and C++, and we could simply write our heavy-duty processing in C++ instead and use threading there, to avoid the Python global interpreter lock. We can create Python bindings for this using the library ctypes. This is a lot riskier, but let's look at an example.

In the below we set up one function in C++ to start threads and call the second function which calculates and writes the result to an existing output array.

```
/**
 * Looks at the input pf_num argument and writes the sector weights of that port
 * No size checking is done on the input pointers, so use this function with gre
 * for input pointer structure.
 *
 * @param  pf_num which portfolio to calculate, used to move to relevant section
 * @param  portfolio_bond_weights  pointer of size num_portfolios * num_bonds, f
```

```
 * @param  portfolio_bond_sectors pointer of size num_portfolios * num_bonds, fl
 * @param  num_bonds how many bonds per portfolio, assumed to be the same for ea
 * @param  portfolio_sector_weights the output array to write portfolio sector w
 * @param  num_sectors the number of sectors in the sector scheme. Used to move
 * @return      void
 * @see         calculate_sector_weights
 */
void worker_calculate_sector_weights(const uintmax_t pf_num, const float64_t *po
                                     const uintmax_t num_bonds,
                                     float64_t *portfolio_sector_weights, const
{
    const uintmax_t starting_out_index = pf_num * num_sectors;
    for (uintmax_t i = 0; i < num_sectors; i++)
    {
        portfolio_sector_weights[starting_out_index + i] = 0.0;
    }

    const uintmax_t starting_bond_index = pf_num * num_bonds;
    for (uintmax_t bond_index = starting_bond_index; bond_index < starting_bond_
    {
        const int32_t &sector = portfolio_bond_sectors[bond_index];
        const float64_t &market_value_pct = portfolio_bond_weights[bond_index];
        portfolio_sector_weights[starting_out_index + sector] += market_value_pc
    }
}

extern "C"
{
    /**
     * Batches out calculations of sector weights in portfolios to threads. Uses
     *  portfolio_sector_weights argument. No size checking is done on the input
     *  See argument descriptions for input pointer structure.
     *
     * @param  portfolio_bond_weights  pointer of size num_portfolios * num_bond
     * @param  pf_size how many portfolios in input pointers, used to move to re
     * @param  portfolio_bond_sectors pointer of size num_portfolios * num_bonds
     * @param  num_bonds how many bonds per portfolio, assumed to be the same fo
     * @param  portfolio_sector_weights the output array to write portfolio sect
     * @param  num_sectors the number of sectors in the sector scheme. Used to m
     * @return      void
     */
    void calculate_sector_weights(const float64_t *portfolios_bond_weights, cons
                                  const int32_t *portfolios_bond_sectors, const
                                  float64_t *portfolios_sector_weights, const ui
    {
        const uintmax_t num_workers = 3;
        std::thread thread_arr[num_workers];

        for (uintmax_t i = 0; i < pf_size; i++)
        {
```

```
            if (i % num_workers != 0)
            {
                continue;
            }
            for (uintmax_t j = 0; j < num_workers; j++)
            {
                uintmax_t pf_num = i + j;

                if (pf_num >= pf_size)
                {
                    continue;
                }
                thread_arr[j] = std::thread(worker_calculate_sector_weights, pf_
                                            num_bonds, portfolios_sector_weights
            }
            for (uintmax_t j = 0; j < num_workers; j++)
            {
                if (i + j >= pf_size)
                {
                    continue;
                }
                thread_arr[j].join();
            }
        }
    }
}
```

We can load this into Python by using the ctypes library and create a wrapper around it to do some argument validation. Then we can call this function as we've called previous functions.

```
def setup_cpp_lib():
  """
  Helper function to load in the shared library object for the C++ functions and
   on the functions in the library. Relies on the .so file being generated and i
   of the python file.
  @return:
  """
    cpp_lib = ctypes.cdll.LoadLibrary("./pf_sector_helpers.so")
    calculate_sector_weights = cpp_lib.calculate_sector_weights
    calculate_sector_weights.restype = None
```

```
    calculate_sector_weights.argtypes = [ndpointer(ctypes.c_double, flags="C_CON
                                         ndpointer(ctypes.c_int32, flags="C_CONT
                                         ndpointer(ctypes.c_double, flags="C_CON
    return cpp_lib


def cpp_calc_sector_weights(portfolios: List[Portfolio], target_weight: float, c
  """
  Python wrapper function for the C++ function to calculate sector weights. Calc
   weights for each sector in each portfolio using one thread per portfolio.
  @param portfolios: list of Portfolio instances
  @param target_weight: target sector weight to compare actual weight to
  @param cpp_lib: shared library loaded using setup_cpp_lib
  @return:
  """
    sector_array = np.concatenate([pf.sectors for pf in portfolios], axis=None,
    mv_arrays = np.concatenate([pf.market_value_pcts for pf in portfolios], axis

    if sector_array.shape != mv_arrays.shape:
        raise ValueError("sectors must have same number of entries as bonds")

    num_portfolios = len(portfolios)
    num_sectors = len(Sector)
    num_bonds = portfolios[0].market_value_pcts.shape[0]
    out_arr = np.zeros((num_portfolios * num_sectors,), dtype=np.float64)

    cpp_lib.calculate_sector_weights(mv_arrays, num_portfolios, sector_array, nu

    for pf_index in range(num_portfolios):
        row_index = pf_index * num_sectors
        for i in range(num_sectors):
            sector = Sector(i)
            log_tolerance(0, sector, out_arr[row_index + i], target_weight)
```

This runs in about 0.06 seconds, a *substantial* improvement from our prior methods. In fact, I had to use a new method to examine CPU and memory usage because it ran too quickly for my prior method. We see CPU Usage between 200% and 400% at various times in the script. I am running on a computer with 6 Cores and 12 logical processors. Using 3 threads in C++ I would expect up to 300% in top used. In reality, I am measuring at a point in

time and threads could be spinning up or down, so I don't see exactly these numbers. That said, usage over 100% indicates we are doing more than one thing at a time. C++, unlike Python, does not have a global interpreter lock (GIL), though production multithreaded code employs various locks and synchronization techniques to protect shared variables and such.

Memory usage is similar to the iterative method, given we are still using Python to call the C++ code. As an aside, at one point in the program CPU usage was 1200% (using each of my 12 logical processors) but this was numpy operations to make mock data and we will talk about numpy shortly.

```
top - 12:12:29 up 2 days,  3:08,  0 users,  load average: 0.13, 0.11, 0.09
Tasks:   4 total,   2 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  25497.7 total,  21481.0 free,    799.1 used,   3217.6 buff/cache
MiB Swap:   7168.0 total,   7168.0 free,      0.0 used.  24489.9 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  346 root      20   0  504.8m  34.5m   9.6m R 400.0   0.1   0:01.03 python3 main.py
    1 root      20   0    5.9m   3.6m   3.1m S   0.0   0.0   0:00.04 /bin/bash
   11 root      20   0    5.9m   3.7m   3.2m S   0.0   0.0   0:00.04 /bin/bash
  345 root      20   0    8.5m   3.4m   3.0m R   0.0   0.0   0:00.08 top -c -b -n1000 -E m -e m -d 0.001
```

The C++ example achieves performance gains, but at the cost of significantly more complex code. Note that we had to work with pointers opening the door to segmentation faults and I didn't do as much argument validation as I should have. I also played pretty fast and loose with shared resources in the threads. This is risky, there could be memory related bugs lurking just below the surface!

I wonder if someone already done an implementation of these calculations in C with some controls that also can take advantage of parallel computing?

numpy

The answer is yes, numpy has done a lot of work to enable fast vectorized calculations that take advantage of C and parallel processing. It also has a lot more controls and safeguards in place than the code I wrote above.

```
def numpy_calc_sector_weights(portfolios: List[Portfolio], target_weight: float)
  """
  Calculates and logs the market value weights for each sector in each portfolio
   processing.
  @param portfolios: list of Portfolio instances
  @param target_weight: target sector weight to compare actual weight to
  @return:
  """
```

```
        sector_array = np.array([pf.sectors for pf in portfolios], dtype=np.int32)
        mv_arrays = np.array([pf.market_value_pcts for pf in portfolios], dtype=np.f
        num_portfolios = len(portfolios)
        num_sectors = len(Sector)
        out_arr = np.zeros((num_portfolios, num_sectors,), dtype=np.float64)

        for sector_index in range(num_sectors):
            sector_mask = np.equal(sector_array, sector_index)
            sector_sums = np.sum(mv_arrays, axis=1, where=sector_mask)
            out_arr[:, sector_index] = sector_sums

        for pf_index in range(num_portfolios):
            for i in range(num_sectors):
                sector = Sector(i)
                log_tolerance(0, sector, out_arr[pf_index, i], target_weight)
```

This runs in about 0.08 seconds, about the same as our self-written method. Notably, CPU usage is higher than in the C++ code I wrote, probably because numpy has different default limits set on the number of cores it can use at once (I set number of threads to 3 in my C++ example). That said, it was tricky for me to identify the CPU usage at the point in the function that does the sector calculations and so how much higher I'm not sure about. I'm sure we could set limits on this if we wanted to, even via docker if needed.

```
top - 12:20:52 up 2 days,  3:16,  0 users,  load average: 0.01, 0.02, 0.05
Tasks:   4 total,   2 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  25497.7 total,  21470.0 free,    809.2 used,   3218.4 buff/cache
MiB Swap:   7168.0 total,   7168.0 free,      0.0 used.  24480.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  381 root      20   0  504.9m  33.0m  10.0m R  1200   0.1   0:00.70 python3 main.py
  380 root      20   0    8.5m   3.5m   3.1m R 100.0   0.0   0:00.08 top -c -b -n1000 -E m -e m -d 0.001
    1 root      20   0    5.9m   3.6m   3.1m S   0.0   0.0   0:00.04 /bin/bash
   11 root      20   0    5.9m   3.7m   3.2m S   0.0   0.0   0:00.05 /bin/bash
```

Other folks may be able to come up with a solution that leverages numpy to vectorize over sectors as well, rendering the for loop over sectors unnecessary and parallelized.

Regarding controls and safeguards, in numpy if I try to access an element outside an array, I get an error. In the C++ code I wrote above I do not check sizes and do validation. Instead, I parse memory addresses to a double. If I access an out-of-bounds element due to bad input or a bug, I will read memory I didn't intend to. Either this memory doesn't belong to me and I will cause a seg fault or I will silently read something I didn't intend to. Either are poor outcomes. Hence, the validation numpy does is useful.

```
Python 3.9.18 (main, Feb  1 2024, 06:03:49)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> arr = np.array([1, 2])
>>> arr[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 10 is out of bounds for axis 0 with size 2
```

There are other examples of controls and safeguards numpy does for us that I don't go into here.

Third Party Libraries

There are other libraries for parallel computing in Python. My colleague, Casey Clements, who wrote the blog post that I mentioned here briefly mentions Dask, Airflow, and Prefect.

Conclusion

Rather like an octopus trying to ambush a clown fish in a coral reef, there are many ways to approach parallel computing in Python. If I was reviewing a pull request implementing parallel computing (or ideally brainstorming with colleagues *before* a pull request ever came out) there are a few factors I would weigh:

- What is our current tech stack? If we are mainly a Python team, I will be biased to look at Python libraries like numpy that handle implementations.
- How many rules and how many bonds do we have? This will influence whether we prioritize approaches emphasizing memory, multiple CPUs, or other aspects. As the number of rules increases we get more benefit from the multiprocessing API, given we can amortize the process startup cost.
- How fundamental and how scalable does the process need to be? For a library that we rely on extensively, that needs to run fast many millions of

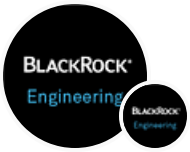
times a day, it may make sense to do a barebones C++ implementation.

There are many other topics in threading that we didn't discuss today like synchronization and deadlocks.

Looking at the approaches in this blog, for this problem, numpy is the winner for me. This is because it has similar performance to barebones C++ but with more controls, safeguards, and features in place.

I hope you all enjoyed this discussion. See full runnable source code examples in this repo.

Python | Parallel Computing | Financial Services | Numpy | Quantitative Finance

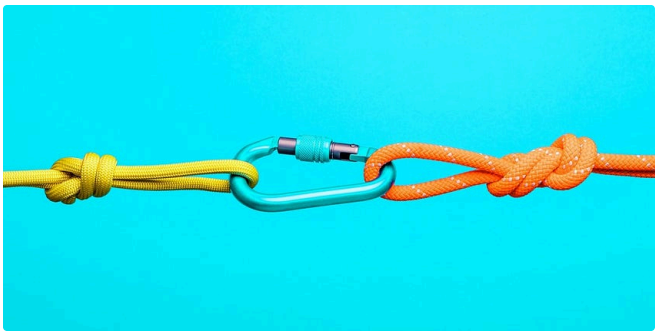


Written by BlackRockEngineering

1K Followers · Editor for BlackRock Engineering

Official BlackRock Engineering Blog. From the designers & developers of industry-leading platform Aladdin®. Important disclosures: http://bit.ly/17XHCyc

More from BlackRockEngineering and BlackRock Engineering



BlackRockEngineering in BlackRock Engineering

Introducing InGen

An Open Source ETT (Extract, Transform, Transfer) Python Tool

5 min read · Jan 31, 2024

21

BlackRockEngineering in BlackRock Engineering

Smart Pointers in C++

In this series of posts, Shreemoyee will dive into several recent additions to C++ which s...

9 min read · Jun 8, 2021

72

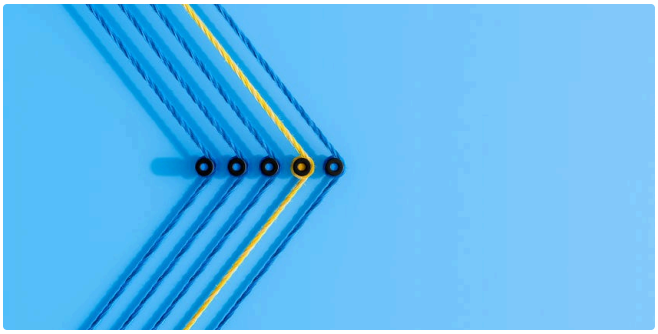
BlackRockEngineering in BlackRock Engineering

Domain-Driven Asset Management

Alan Moore describes the use of Domain-Driven Design for Asset Management at...

8 min read · Feb 28, 2023

65 2



BlackRockEngineering in BlackRock Engineering

Delivering Eventual Consistency with Kafka Streams

BlackRock Engineer, Kevin Sun shares an example of implementing eventual...

11 min read · Feb 23, 2021

184 2

See all from BlackRockEngineering

See all from BlackRock Engineering

Recommended from Medium

BlackRockEngineering in BlackRock Engineering

Similarity Learning, the art of identifying neighbors

BlackRock data scientists discuss how they think about similarity and advanced risk and...

3 min read · Nov 16, 2023

20

Quant Prep

Salaries for Investment Banks & Hedge Funds for London & New...

In this article, we'll delve into the average salaries and bonuses for various front-office...

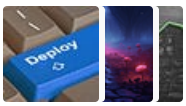
4 min read · Mar 11, 2024

92

Lists

Coding & Development

11 stories · 516 saves



Predictive Modeling w/ Python

20 stories · 1013 saves

Practical Guides to Machine Learning

10 stories · 1220 saves

ChatGPT

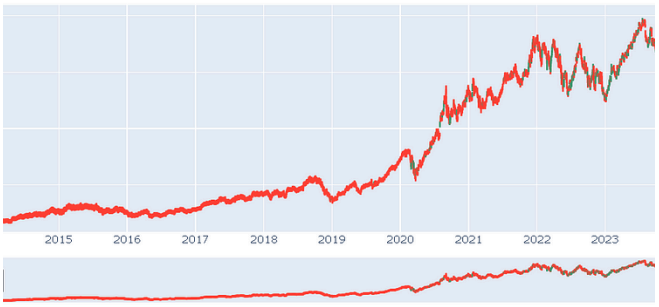
21 stories · 528 saves

The AI Quant

AI-powered Option Strategy Generation with Python

Welcome to the fascinating world of AI-powered option strategy generation with...

· 15 min read · 4 days ago



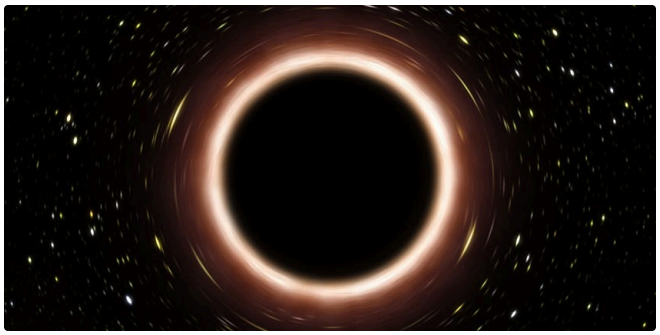
Maggie Hemings

Algorithmic Head and Shoulders Detection and Trading In Python

Unmasking the Head and Shoulders: A Trader's Secret Weapon

8 min read · Mar 3, 2024

47 4



Mirko Pet... in Mirko Peters—Data & Analytics ...

Introduction to Markov Chains

Welcome to our comprehensive guide on Markov Chains. This mathematical concept...

· 11 min read · Mar 2, 2024

111

Sofien Kaabar, CFA

Cracking The Dark Pool: Forecasting S&P 500 Using...

Creating a Random Forest Algorithm to Predict S&P 500 Returns

· 4 min read · Mar 7, 2024

61

See more recommendations