ADDRESSING MULTITHREADING AND MULTIPROCESSING IN TRANSPARENT AND PYTHONIC METHODS

David Liu, Python Technical Consultant Engineer

Overview

- State of current concurrency and parallelism
- Nested parallelism and oversubscription
- A few composable methods of thread control
- How it works under the hood (*tbb*, *smp*)
- Pythonic style?
- Future of Pythonic style for parallelism
- Summary

Current State of Python concurrency and parallelism

- The Python ecosystem has had quite a few cool developments over the last few years:
  - *Threading library (2008)*
  - *Multiprocessing (2008)*
  - *Twisted (2008)*
  - *Concurrent futures (2009)*
  - *Cython (2009)*
  - *Tornado (2010)*
  - *Numba (2012)*
  - *Asyncio (2013)*
  - *Dask (2015)*
  - *Trio (2017)*

Current State of Python concurrency and parallelism

- The options in this space are *very* good compared to other ecosystems
- Majority do a good job of playing nicely with the *Global Interpreter Lock* (GIL) or walk around it with *distributed* or *vectorization techniques*
- In more domain specific areas, one can rely on high-end C libraries that have threading to harness parallelism (SciPy/NumPy)
- Recent trends have Python accessing increasing core count machines (from 2-4 to over 28 core) as commonplace
  - *Nested parallelism and oversubscription* now quite possible in kernels

The Safety of the GIL

- The GIL has been complained about by many in the Python space
- Many efforts have been made to try to remove the GIL
- As it stands, some of the main tenants of what guarantees the GIL provides are hard to ignore
  - *Read/write safety for Python Object access*
  - *Predictable behavior*
  - *Ensure reference counting doesn't get hosed*
  - *Makes extension module development easier (and removes the undue burden on developers)*

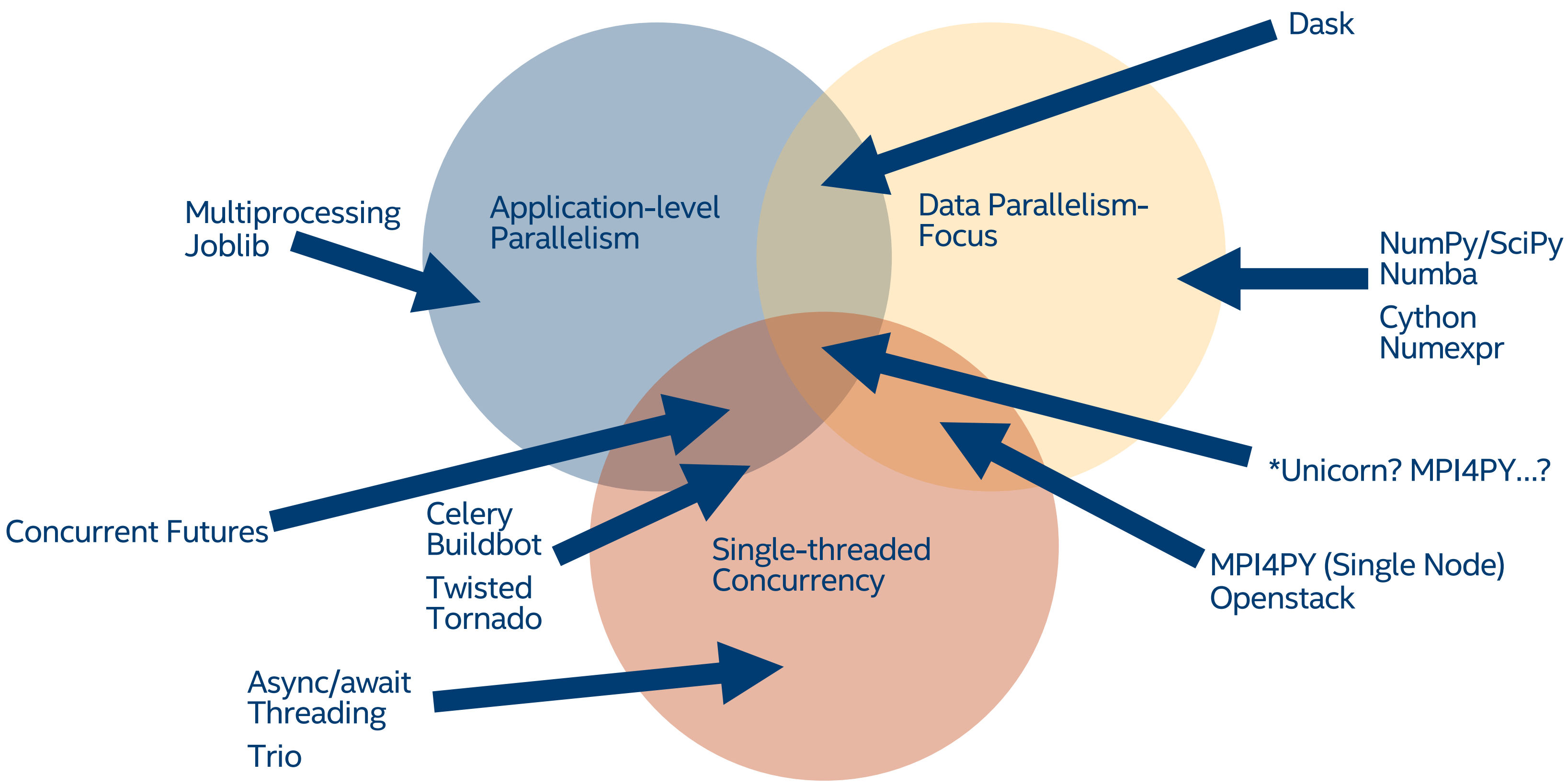
The Safety of the GIL (con't)

- In reality, **the GIL is a non-issue** as many have found *ways of stepping around the GIL*
- *SciPy and NumPy* are great examples—once a command is sent to SciPy, it gets dispatched where BLAS implementations like MKL and OpenBLAS are vectorized and parallelized
- Other frameworks directly access vectorization and exit the Python+GIL layer to utilize threads—*Numba, Numexpr, Cython* do this
- *Multiprocessing* frameworks can escape it via a separate process, which can also have separate threads

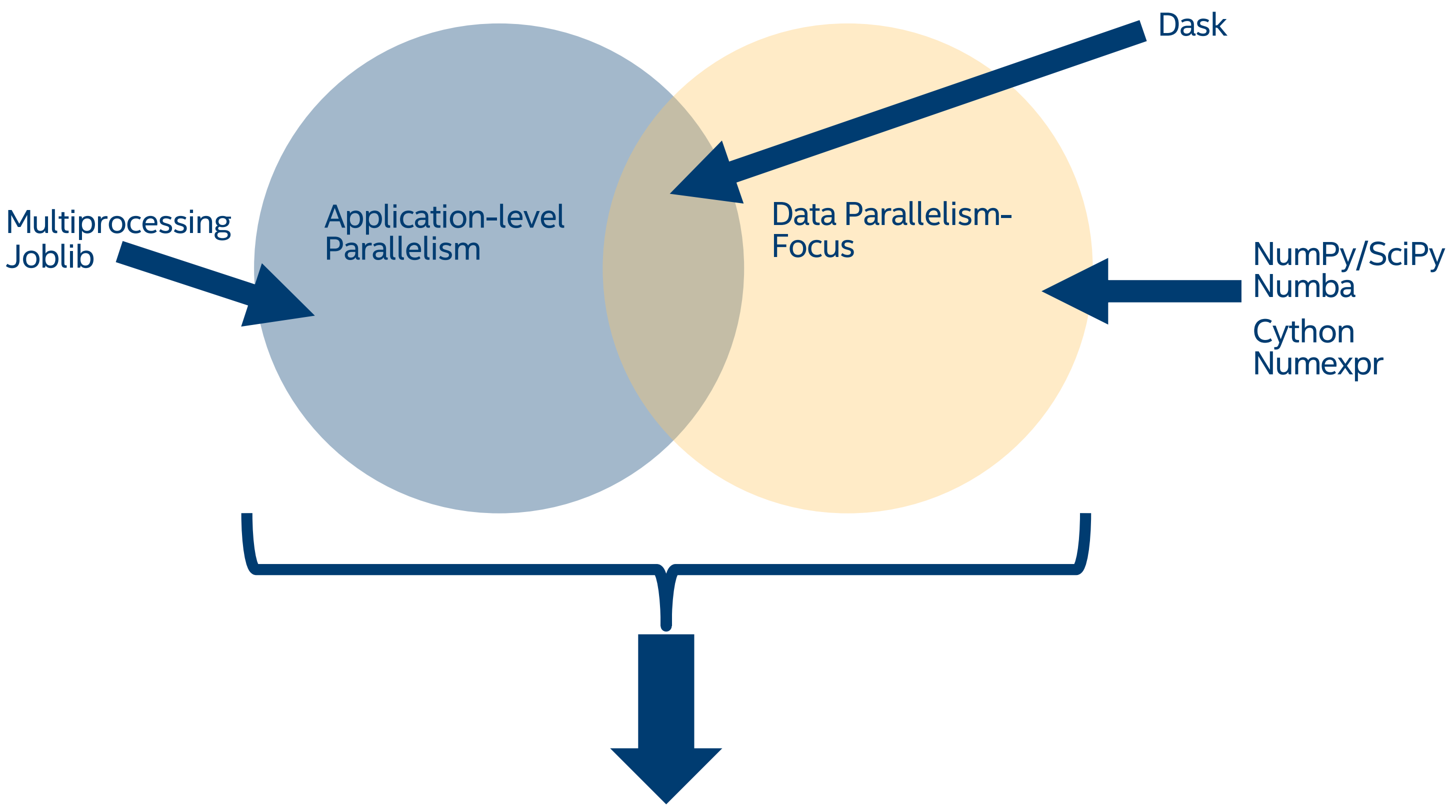
The Safety of the GIL (con't)

- Exiting the GIL with a C library is the generally the most *Pythonic-ish* way of doing things (as it encompasses the abstraction of a known computational flow)
- Composition of abstracted flows also works (splitting off into multiple processes)
- It is quite rare to absolutely necessitate a language to be completely thread safe; many of the advantages of Python would go away

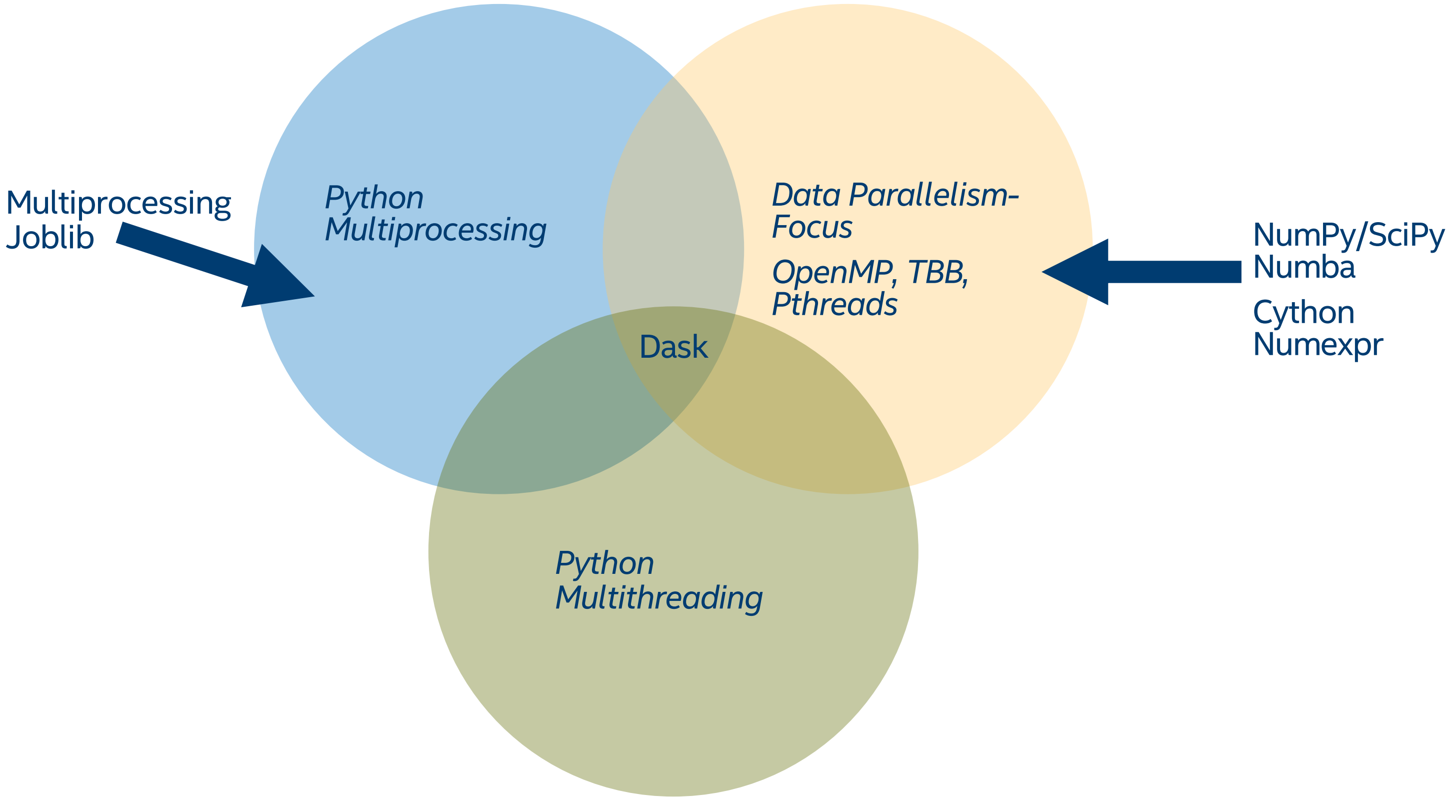
The spaces covered



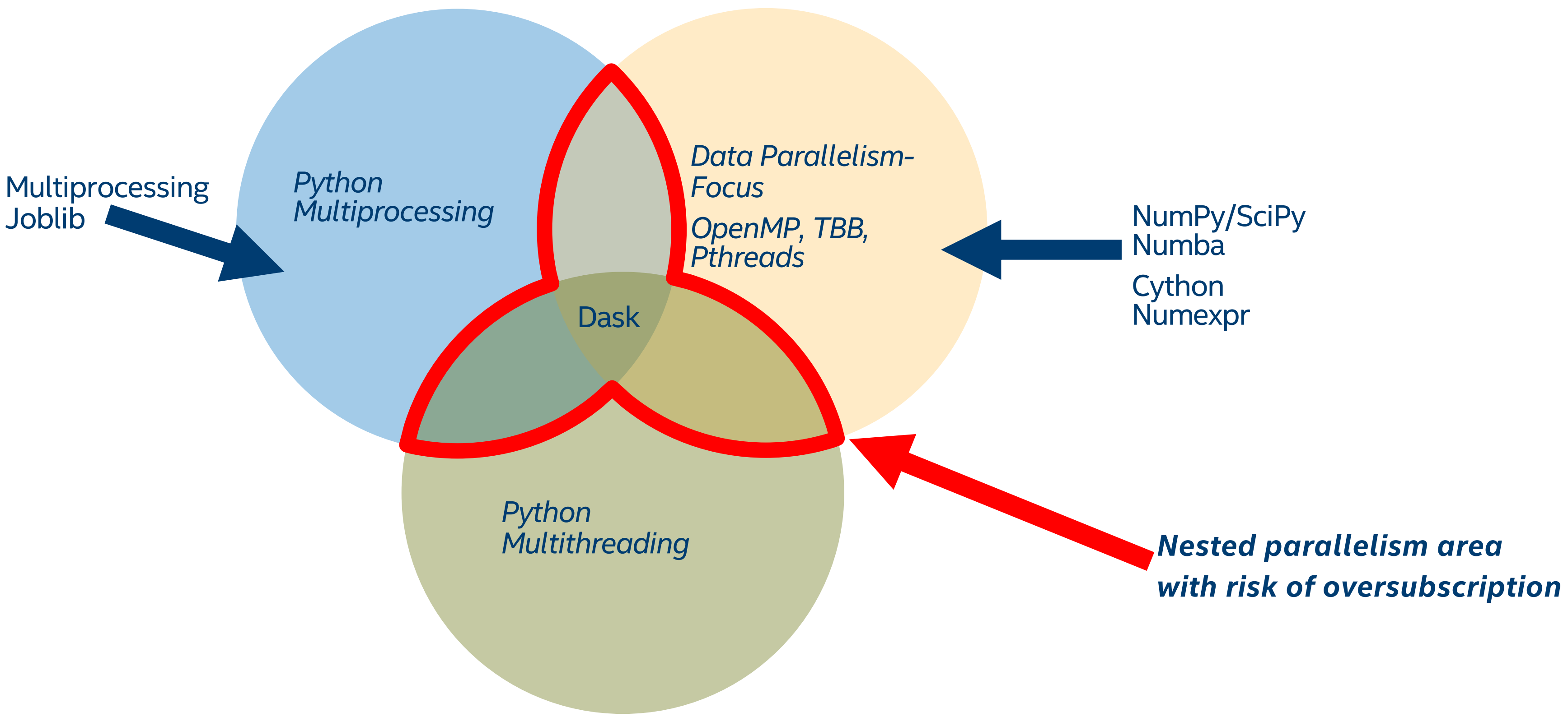
The spaces covered



The spaces covered



The spaces covered



Nested parallelism

```
data = numpy.random.random((256, 256))

pool = multiprocessing.pool.ThreadPool() # creates P threads

pool.map( np.linalg.eig, [data for i in range(1024)])
```

P Python threads ***** *P* NumPy→MKL→OpenMP threads = **P^2** threads total

Oversubscription

P software threads

P CPUs

*P*P* threads

P CPUs

Oversubscription overheads

- Types of impact
  - Direct OS overhead for switching out a thread
  - CPU cache becomes cold: invisible impact
  - Other threads are waiting until the preempted one returns
- *Tensorflow, Scikit-Learn, PyTorch* have a recurring battle with these
- How do they solve it?
  - Most use OMP_NUM_THREADS=1… KMP_BLOCKTIME=1…
  - SMP ironically addresses this (more on this later)

Introducing composability modules

- **tbb4py: Intel TBB for Python**
  - A Python C-extension package managing nested parallelism using dynamic task scheduler of Intel® Threading Building Blocks library
  - Instantiates via monkey patching of Python's pools and enabling TBB threading layer for Intel® MKL (no code changes required)
  - Dynamically maps tasks onto coordinated pool(s) to avoid excessive threads

Introducing composability modules

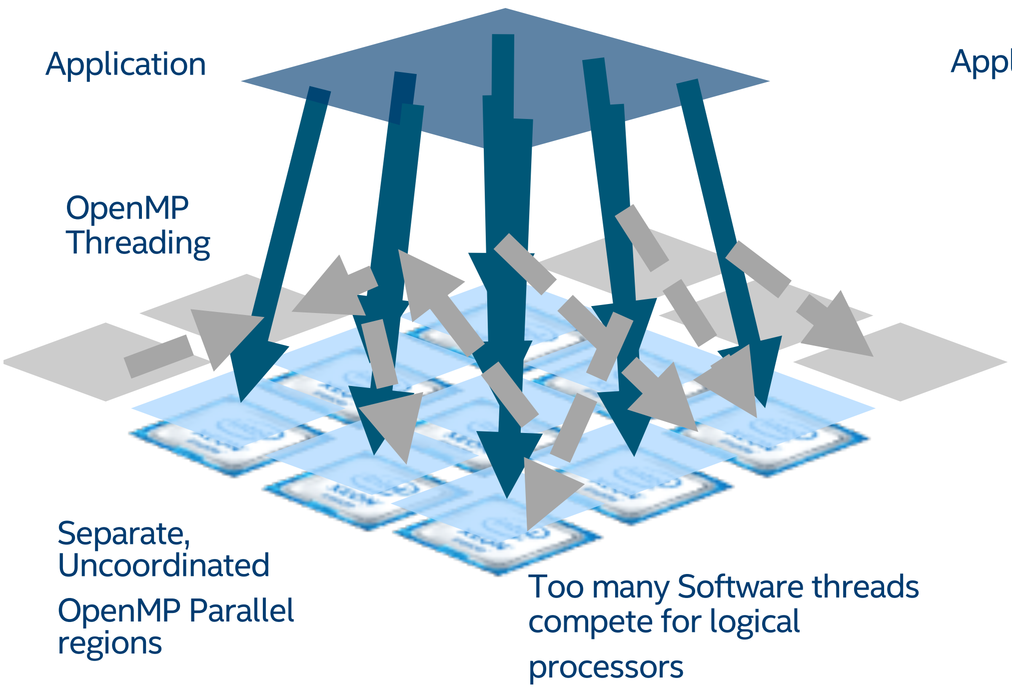
- **smp: Static Multi-Processing**
  - A Pure Python package managing nested parallelism through coarse-grain static settings
  - Instantiates via monkey patching of Python's pools (no code changes required)
  - Utilizes affinity mask + OpenMP settings to statically allocate resources and avoid excessive threads

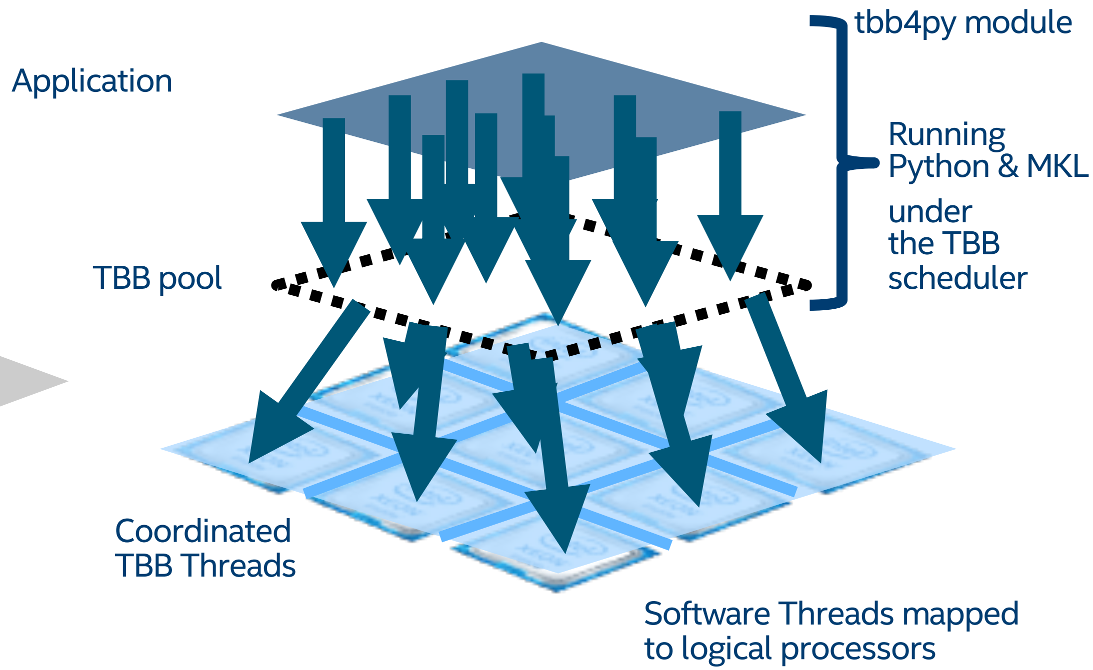
Nested parallelism (again)

```
data = numpy.random.random((256, 256))

pool = multiprocessing.pool.ThreadPool() # creates P threads

pool.map( np.linalg.eig, [data for i in range(1024)])
```

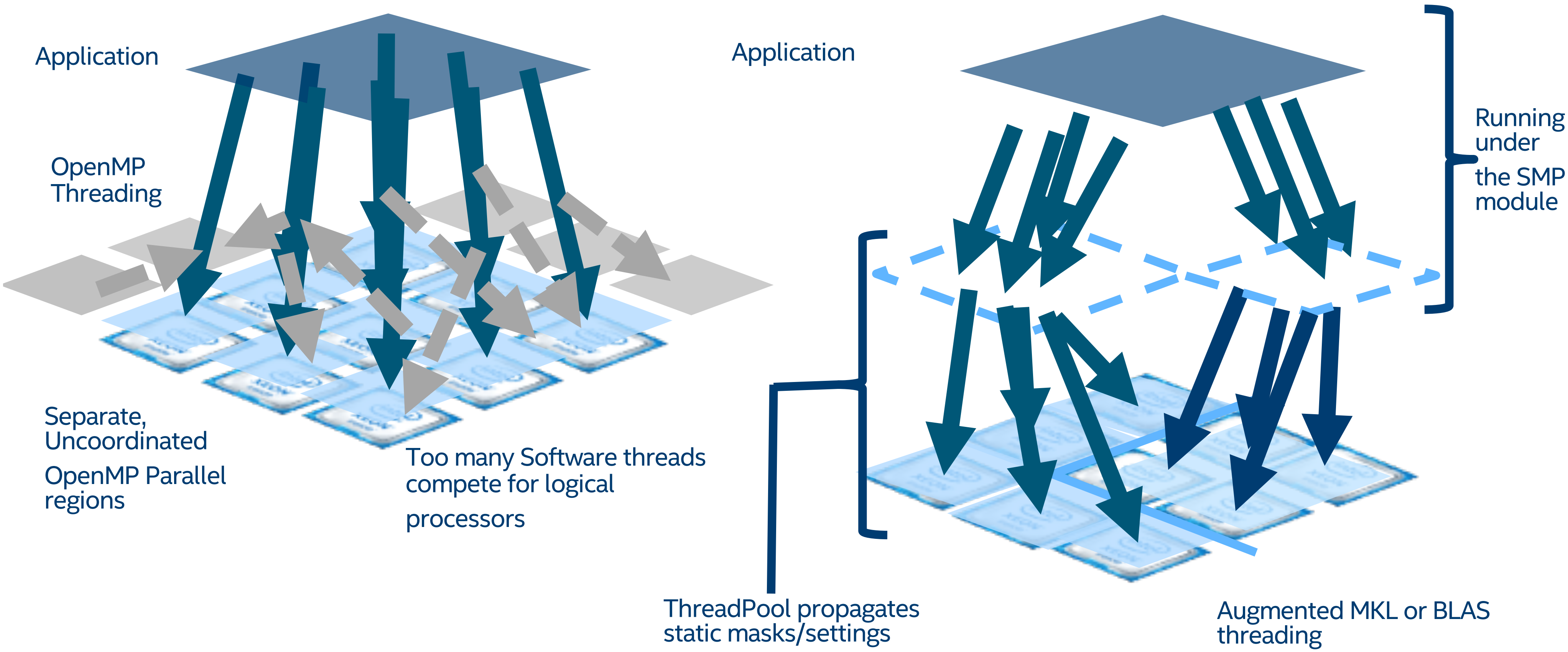
P Python threads ***** *P* NumPy→MKL→OpenMP threads = **P^2** threads total

TBB's Thread coordination system





SMP's total threading affinity system



DEMOS

Repository:

https://github.com/IntelPython/composability_bench/tree/master/scipy2018_demo

Current State of Python concurrency and parallelism (slight return)

- Much of the **concurrency and async areas** are rich with packages that help solve the needs of the majority of Python users
- True Parallelism is a small but strong area, so focus has generally been towards concurrency + async offerings
- Most ways of achieving parallelism in this area rely on *vectorization frameworks* or with multiprocessing or distributed
- How does one do so in a **semi-pythonic** way?

Pythonic-ish?

- Relatively few code changes
- Modify current behavior of a framework to fit one's needs (or prevent a massive rewrite)
- Directly in the Python std library
- Writable from the Python layer
- Easy interface to understand
- Keeps one in the Python layer (and does not drop to an IR)

How close can we get?

Pythonic-ish? (tbb4py)

- ✓ Relatively few code changes
- ✓ Modify current behavior of a framework to fit one's needs (or prevent a massive rewrite)
- ✗ Directly in the Python std library
- ✗ Writable from the Python layer
- ✓ Easy interface to understand
- ✓ Keeps one in the Python layer (and does not drop to an IR)

Pythonic-ish? (smp)

- ✓ Relatively few code changes
- ✓ Modify current behavior of a framework to fit one's needs (or prevent a massive rewrite)
- ✗ Directly in the Python std library
- ✓ Writable from the Python layer
- ✓ Easy interface to understand
- ✓ Keeps one in the Python layer (and does not drop to an IR)

Pythonic-ish style for parallelism?

- *How realistic is it to have a firm requirement for a Pure Python implementation?*
- *What is the best way to modify Python code? Monkey patching? New framework?*
- *At what level should the parallelism be controlled?*
- *Can an interface be agreed upon to operate on parallelism? (such as concurrency's concurrent futures)*

Pythonic-ish style for parallelism? (con't)

- How realistic is it to have a firm requirement for a Pure Python implementation?
  - Not required, but highly recommended
- What is the best way to modify Python code? Monkey patching? New framework?
  - Monkey patching is seeming to be the new normal

Pythonic-ish style for parallelism? (con't)

- At what level should the parallelism be controlled?
  - Python layer-sort of? It should have directives for how additional layers can compose it as the best case
- Can an interface be agreed upon to operate on parallelism? (such as concurrency's concurrent futures)
  - Jury is still out on this one, but with every iteration of attempts (like *smp*) we get a more clear picture

Summary

- **tbb4py** and **smp** attempt to address Pythonic-ish methods by attempting to *augment* the way we use multithreading and multiprocessing (attempting to not change underlying code)
- It is best to leave the two forms of multiprocessing and multithreading at their same levels—Python level and C level, respectively
  - Most multithreading is domain specific it needs to be in C, so it would need to be written or C or generated (like Numba, numexpr, Cython)
  - Perhaps leaving threading and multiprocessing directives as a file or comments might be better… but doesn't that just sound like #pragma omp?

Summary (con't)

- Having more "augmentable" threading behavior is more useful, but that means putting the bulk of the responsibility on the users themselves
- Threading for numerical has lots of known frameworks, proper threading from non-numerical may be possible but will require stricter typing than just "Python Object"
  - At that point… why are you using Python, right? Flexible vs. Strict
- The Python ecosystem has a critical mass of good frameworks looking to address multithreading and multiprocessing—so for those of you working in it, keep going!

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®

Software