Proposal: Enable Optional Parallelism in PyGeoProcessing

Background

While pygeoprocessing does a really good job of being as efficient as it can reasonably be, we have long assumed that it's just not worth the overhead to parallelize operations. For InVEST applications on a researcher's laptop, this is typically sufficient. However, with the software directly supporting more of our science efforts, I have seen three parallel trends emerge:

1. A trend towards global-scale analyses, such as the Morgan Stanley project and Nature's Frontiers ("NCI").
2. Increased usage and adoption of finer-scale spatial data products at large scales, such as for an entire country, as with the case of Jesse's SDR run on a 30m DEM for the whole country of Colombia.
3. Increased usage of high-performance computing resources, such as Stanford's Sherlock cluster, which dramatically increases available compute resources, both in terms of CPU cores on a machine and the number of machines available.

Given these emergent trends, maintaining a strictly-serial approach to our most common algorithms, particularly those that can be trivially parallelized, is causing us to needlessly spend time waiting for jobs to finish when they can often be parallelized. This proposal, if adopted, could save us a nontrivial amount of time in our research scripts and open the door to HPC-grade parallelism of InVEST models (dependent on a future version of taskgraph).

Current State of Parallelism in PyGeoProcessing

Architecture

Most functions use `threading` paired with a `queue` to provide a separate thread of execution and pass information to/from the thread. The `pygeoprocessing.multiprocessing` subpackage contains a `multiprocessing`-based `raster_calculator` implementation that uses the same producer/consumer architecture to feed data to worker processes from the main thread.

There are 3 functions where threading is used in pygeoprocessing:

Pygeoprocessing.raster_calculator

`Raster_calculator` uses a separate thread to compute statistics.

Pygeoprocessing.convolve_2d

`Convolve_2d` uses two threads, one to load the work queue, and another to do the fft convolution. The main thread does all the GDAL-based reading/writing.

Pygeoprocessing.multiprocessing.raster_calculator

This is a `multiprocessing`-based parallel implementation of `raster_calculator` where the `local_op` is executed in separate processes. Rich implemented this separate version of `raster_calculator` due to a need for a project. While this sort of parallelism is unnecessary for most InVEST functions where disk I/O dominates the runtime, this function is highly useful for CPU-bound `local_op` functions, such as applying ML models as we do in NCI.

Proposed Implementation

I propose that we allow users to provide `concurrent.futures`-compliant `Executor` pools to `pygeoprocessing` functions that are likely to be CPU-bound. For use cases where function parameters need to be pickled, `str` aliases are supported instead.

Why `concurrent.futures`?

The package `concurrent.futures` is a part of the Python Standard Library (since 3.4) and provides a high-level interface for asynchronously executing callables. It allows for computational tasks to be submitted to `Executor` instances for execution when compute resources become available. This submission interface eliminates the need for us to create and manage a work queue. The job submission interface does not address task priority, only task execution.

Because concurrent.futures is a part of the Python Standard Library, other packages are looking to concurrent.futures for API design and creating API-compatible, drop-in replacements that swap out the computational backend. The package `parsl` (docs), for example, supports tasks to be executed on different machines, such as in an HPC environment, among other interesting things. `Dask` also supports passing executor tasks to clients.

API Changes

Functions that are likely to be CPU-bound will have a new parameter added, `executor=None`. For example:

```python
def raster_calculator(
    base_raster_path_band_const_list, local_op, target_raster_path,
    datatype_target, nodata_target,
    calc_raster_stats=True, use_shared_memory=False,
    largest_block=_LARGEST_ITERBLOCK, max_timeout=_MAX_TIMEOUT,
    raster_driver_creation_tuple=DEFAULT_GTIFF_CREATION_TUPLE_OPTIONS,
    executor=None):
```

The `executor` parameter is allowed to be one of the following:

- `None`, in which case the potentially-parallelized tasks will be executed synchronously.
- A subclass of `concurrent.futures.Executor`, such as `ProcessPoolExecutor` or `ThreadPoolExecutor`.
- A `str` indicating the pool method to use and the number of workers. This allows the parameter to be serialized and still have the function be parallelizable. If the number of workers is not provided, For example:
    - `Multiprocessing:3` would use `ProcessPoolExecutor` with `3` workers
    - `Threading:5` would use `ThreadPoolExecutor` with `5` workers
    - `Multiprocessing` or `threading` without the `:<n_workers>` would use `sys.cpu_count()` as the number of workers.

Within the pygeoprocessing function, CPU-bound tasks will be added to the executor and executed in an appropriate sequence. If there are any task dependencies involved in the function, dependencies will be handled through the `Future` object returned on job submission, and all `Future`s will be `join`ed before the pygeoprocessing function concludes.

The minimum set of functions I propose updating in this way are:

- `Align_and_resize_raster_stack`: calls to `warp_raster` are trivially easy to parallelize, as calls are executed in a simple for-loop. A prototype of the above API change in `align_and_resize_raster_stack` is available here.
- `Raster_calculator`: block computations would be the primary target. Statistics computation may also benefit from this parallelism.
    - `Raster_calculator` would also need to check for whether the `local_op` can be pickled if we are using a `ProcessPoolExecutor` and raise an appropriate

exception if it cannot be pickled. The `Executor` doesn't check for this automatically, it just fails silently.

Additionally, the following functions may benefit from parallelism but I personally have not observed enough of a bottleneck (and I am open to considering these if anyone has any frustrating experiences with them!):

- `Zonal_statistics`: calls to `reproject_vector`, `gdal.Warp` and `gdal.Rasterize` might benefit from parallelism.
- `Reproject_vector`: the reprojection of complicated features could be distributed across multiple CPU cores.
- `Convolve_2d`: the FFT calculations would benefit from parallelism.
- `Stitch_rasters`: there are a few possible places where rasters are warped that could be parallelized.

New Utilities

To implement this, I suspect we will benefit from implementing the following helper tools:

- A wrapper class that provides the same interface as `concurrent.futures.Executor` and `Promise` classes but for synchronous execution.
- A utility function to translate the string representations of executor type and number of workers to an `Executor` instance.

Benefits of This Approach

Parallelize when It's Useful

The proposed system will allow us to scale the computation of our pygeoprocessing operations when it's useful, but we don't have to worry about the overhead when it isn't needed or worth it, all without having to change the function signature except for the one relevant parameter.

Integration With Other Systems

Adhering to the standard implementation and API interface means that we expand our ability to integrate with other distributed computing systems while limiting the effort involved with integration.

Reduced Code

We could eliminate the `pygeoprocessing.multiprocessing` package and its associated tests, rolling any needed tests over to `raster_calculator`.

Costs Of This Approach

Implementation Time

To implement parallelism with the minimum set of functions, I estimate 1-3 full days of development time. Implementing parallelism in all of the possible functions would likely take 1-3 weeks of development time.

Additional Code Paths to Test

The ability to run on different executors means that we have additional code paths that would need to be tested for functionality and correctness.

Consequences to InVEST

Because `executor=None` will be the default parameter, this change will have no immediate effect on InVEST.

However, because the option is there, it gives us the ability to use parallelism in our research scripts and any task management frameworks that support multiple CPUs per task. If a future `taskgraph` iteration supports multiple tasks per job, then we could see the parallelism benefitting InVEST models.

Note that if ever we do decide to roll out support for multiprocessed versions of raster_calculator, any multiprocessed `local_op`s would need to be at the module level so that they can be pickled. An exception would be built in to `raster_calculator` to guard against this case.

Examples

Functional examples of this using a prototyped `align_and_resize_raster_stack` are available at this repo: https://github.com/phargogh/prototype-concurrency-in-pygeoprocessing