UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

Part III: Parallelization and GUI

Just-in-time Compiled Python for Bioinformatics Research

Johanna Elena Schmitz, Jens Zentgraf and Sven Rahmann

ISMB 2024, Montréal, Canada (July 12, 2024)

Parallelization

Idea

- Given a big data set on which we want to compute something.
- Each data point can be processed independent of each other.
- Process multiple data points at the same time.

External: Processes

- Split the input into multiple files.
- Run the tool for each input file.
- Merge results of all runs.

Internal: Threads

- Run the tool once.
- Split the input internally.
- Output results for complete file.

Process vs. Thread

Processes

- Independent programs
- Separate memory space
- Isolated from other processes
  - To share data, it has to be written to disk
  - Communication is complex
- Run a application multiple times
- Big overhead

Threads

- Lightweight version of processes
- Share memory with the parent thread
- Not isolated
- Easy communication between threads and the process
- Used to parallelize steps inside an application
- Small overhead

Problem

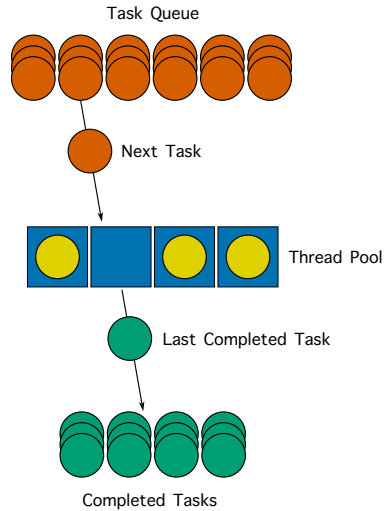
- Python uses a **global interpreter lock** (GIL).
- Only one interpreter per Python process.
- Threads cannot work in parallel since they share the same interpreter.
- Processes have independent interpreters.

Threadpools

- Create a thread pool with a number of threads
- Each thread works independent
- Split the input into multiple tasks
- Submit each task to a task queue
- Each thread picks a task and computes it
- If a thread is finished, it can pick a new task
- The results can be directly processed or the program waits until all threads are done

Python ThreadPoolExecutor

- `max_workers` defines the number of threads
- Using `submit(func, *params)` we can submit a task to the thread pool
- `wait(tasks)` waits until all tasks are done



Using Threads with Numba

- Numba compiles the code to CPUDispatcher objects.
- If no fallback to Python is necessary, Numba does not use the interpreter.
- Numba can release the GIL when execution a function compiled with `nopython` (`@njit` or `@jit(nopython=True)`).
- This enables the use of threads.

Using Threads in Python

- Python provides an experimental parameter to disable the GIL.
- New features are available in Python version 3.13.
- Still experimental!
- Can still be buggy!

Example Running Mean

```
@njit(locals=dict(asum=int64))
def move_mean(array, window_size, out):
    asum = sum(array[:window_size])
    out[0] = asum // window_size
    for i in range(len(array) - window_size):
        asum += array[i + window_size] - array[i]
        out[i + 1] = asum // window_size

def main(args):
    # Read file containing numbers
    with open(f"array_n{args.n}.csv") as arr_file:
        arr = np.array(list(map(int, arr_file.read().split())),
                       dtype=np.int64)
    out = np.empty(len(arr) - args.window_size + 1, dtype=np.int64)
    move_mean(arr, args.window_size, out)
```

Example Running Mean

Release the GIL

```
@njit(nogil=True, locals=dict(asum=int64))
def move_mean(array, window_size, out):
    asum = sum(array[:window_size])
    out[0] = asum // window_size
    for i in range(len(array) - window_size):
        asum += array[i + window_size] - array[i]
        out[i + 1] = asum // window_size
```

Example Running Mean

Split the input

```
def main(args):
    ...
    threads = 4
    arr = np.array(arr, dtype=np.int64)
    out = np.empty(len(arr) - args.window_size + 1, dtype=np.int64)
    borders = [int(i * len(arr) / threads) for i in range(threads)]
    borders.append(len(arr))
    for b in blocks:
        move_mean(arr[borders[t]:borders[t+1] + window_size - 1],
            window_size, out[borders[t]:borders[t+1]])
```

Example Running Mean

Create a Thread Pool

```
def main(args):
    ...
    threads = 4
    borders = [int(i * len(arr) / threads) for i in range(threads)]
    borders.append(len(arr))
    with ThreadPoolExecutor(max_workers=threads) as executor:
        futures = [executor.submit(move_mean_njit_parallel,
            arr[borders[t]:borders[t+1] + window_size - 1],
            window_size, out[borders[t]:borders[t+1]])
            for t in range(threads)]
        wait(futures)
```

Parallelize Motif Matcher

- Split the sequence into chromosomes.
- Submit each chromosome as an independent task to the threadpool.
- Run the pattern search in each chromosome independently.

Problem

Reading the FASTA file (not easy to parallelize) takes longer than very efficient pattern search.
$\rightarrow$ Parallelization does not provide a large speed-up. $\rightarrow$ Parallelization only advisable if you can parallelize the computationally intensive parts.

Streamlit

Goals

- Turning python scripts to **shareable** web apps.
- **Easy** to implement without needing front-end experience.
- Compatible with most python libraries, such as **matplotlib, seaborn, plotly, pandas, Pytorch, SymPy,** etc.

Supported Browsers

- Google Chrome
- Firefox
- Microsoft Edge
- Safari

How to Use Streamlit?

1. Write a python script and import `streamlit`.
2. Add widgets, buttons, etc. to your code.
3. You can also add elements to a **sidebar**.
4. Based on the status of an input widget you can run different code and show the results. Some widgets, like buttons, allow to directly add a **callback function** that is executed when the button is clicked.
5. You can create **multiple pages**, by having multiple `.py` files in a pages folder.

```python
import streamlit as st

st.title('Your title.')

# creates a radio button at the sidebar
c = st.sidebar.radio('Pick a color:',
    ('red', 'green'))

# run different code for selected c
def find_red_fruits():
    pass

if c == 'red':
    fruits = find_red_fruits()
    st.write('\t'.join(fruits))
```

Important Streamlit Functions

Text, Data, Chart Elements

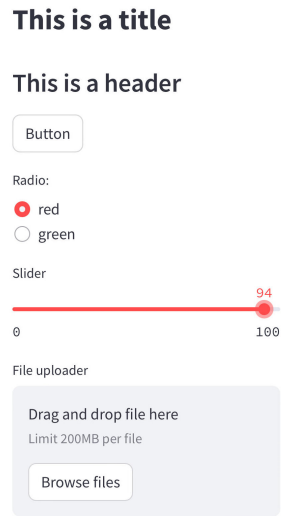
- `st.title`, `st.header`, `st.caption`, `st.text`, etc.
- `st.table`, `st.dataframe`, etc.
- `st.bar_chart`, `st.line_chart`, etc.
- `st.pyplot`, `st.altair`, etc.

Input Widgets

- `st.button` , `st.radio`, `st.numeric_input`, `st.slider`, etc.
- `st.file_uploader` (uploads file to RAM)
- etc.

Media Elements

- `st.image`, `st.video`, etc.



How to Run your Streamlit Web app?

Run `streamlit run filename.py`

- Opens the `streamlit` app in a new browser window and runs the script from top to bottom.
- Changing an input widget triggers the script to rerun from top to bottom.
- Streamlit detects if there are changes in your source code and integrates the changes in the next rerun.
- Caching allows to store the results of compute-intensive function calls and return the cached result when the function is called again with the same inputs.
- For caching, use function decorator `@st.cache_data`.

Take Home Messages

- The `nopython` mode allows to release the GIL.
- Releasing the GIL enables the use of threads.
- Threadpools are an easy way to assign tasks to threads.
- Speed-up for parallelization depends on problem.

- Streamlit is an easy way to create shareable web apps in python.
- Streamlit apps are Python scripts that run from top to bottom.
- Streamlit support many ways to interact with the user, display text, media or charts and connect to data sources.
- Every time the app is opened in a browser, a new `streamlit` session starts.
- Every time the user interacts with the a widget, the script reruns.
- Session states and caching allows you to save information between reruns.

Hands on Session

1. Motif Search App

We will now take a quick look at the implementation of our motif search streamlit app.

2. Write Your Own Web App

- Write your own streamlit web application. The app should at least contain
  1. input widgets
     - for the version (pure python or numba),
     - for the motif,
     - to upload the reference file
  2. a start button to run the motif search
  3. a results section with
     - the elapsed time,
     - a visualization, e.g., the number of matches per chromosome

You can find the API reference at
https://docs.streamlit.io/develop/api-reference.