P2 - Parallel Patterns

CS4204

word count: 1323

200007413

March 2024

Contents

List of Figures

1 Introduction

2 Design and Implementation

This implementation of the para-pat library is designed as a directional network of Node instances, with thread-safe input/output queues.

2.1 Node

The Node class is intended as an interface class.

The `run_and_wait()` function calls and waits for the asynchronous `run` function. This spawns a thread executing a subclass-defined `spawn` function. The `run()` function is a wrapper for the `spawn_fn`. It's job is to spawn a thread running this function. The `wait()` function just takes the `thread_id` attribute set by `pthread_create`, and makes the process/thread executing the wait function wait for the Node to finish execution. Waiting on a Node that hasn't been run is undefined behaviour. The `run_and_wait()` just runs these two functions.

An issue found during implementation was spawning a sub-class defined function from a parent class using `pthread_create`. This couldn't be done by directly overriding a declared `spawn` function due to c++ specification restrictions on parents fetching the pointer to an unspecified child's function. The solution was to create a **variable** `spawn_fn` holding a function pointer in the Node parent. when a child class is initialized, this variable is set to a pointer to that sub-classes `spawn` function.

The `spawn_fn` function then interacts with the `input_queue` and `output_queue` attribute. The `stop_on_empty` attribute (by default true) tells the node whether to stop when the input queue is empty. When false, the node continually attempts to fetch items from the input queue, allowing pipelines to be created.

The `set_stop_on_empty()` function sets the `stop_on_empty` attribute. This function can be overridden by sub-classes (such as Pipe) if required. The attribute should be set before execution to `true` if the input queue will not be added to during execution, and `false` otherwise. When an input queue is filled, this attribute can then be set to false, and the loop checking for input will cease.

2.2 Worker

A Worker is the most basic Node. It is initialized with a `worker_fn`, and optionally an input queue. If not, the input queue is initialized empty.

The spawn function repeatedly pops tasks from the input queue, running the worker function on each task, and then adds the result to the output queue.

Alone, these just allow a single level of parallelism, but are the building block for both Pipes and Farms.

2.3 Pipe

Pipes are three stage pipelines. Each stage is one Node object. When initialized, the master input queue is set as the input queue for stage 1, and the output queue of stage 1 is set as the input queue for stage 2, the output queue for stage 2 is set as the input queue for stage 3, and finally the master output queue is set as the output queue for stage 3.

By default, Nodes stop when their input queues are empty. Therefore, we set stage 1 to stop on empty by default, and 2 and stage 3 to not stop on empty. This is because when input is not added asynchronously, stage 1 is complete once the master queue is empty. However, stages 2 and 3 are spawned at the same time as stage 1, and may be faster than previous stages. Therefore, subsequent stages must not stop when their input queues empty until the previous stages are all complete.

The `set_stop_on_empty()` function had to be overridden for the Pipe class as all that is required to enable streaming input is to set the `stop_on_empty` attribute for stage 1. This is because stages 2 and 3 already stream input, and get told to stop by the `spawn` function once stage 1 is complete.

2.4 Farm

The Farm Node is initialized with a `worker_count` and `worker_fn`. The node function spawns the appropriate number of Worker nodes, each with the same worker function. Each Worker has it's own input queues filled from the Farm's master input queue asynchronously after the workers are spawned. This allows the workers to begin work on

input before the whole input has been distributed. After every worker is complete, the outputs are collected into the single master output queue.

The `set_stop_on_empty()` function did not require overloading, as the Workers are set to `false` to enable asynchronous input streaming. Instead, the attribute causes the farm to continue trying to distribute tasks until set to `false`.

A further improvement to the Farm would be to create an Emitter and Collector Node, which asynchronously fills input and collects output. This could be done by creating a Pipeline with a Worker for the Emitter and Collector. This would likely improve execution speed, as the output is only generated once all Workers are complete. However, I did not have enough time to implement and test this.

3 Testing

Unit testing was conducted using the Boost.Test library. Tests were repeated many times on different systems to mitigate inherent issues with parallel unit tests. However, their primary use was just to verify interactions between sequential functions, not parallelism itself. Results are found in Figure 1.

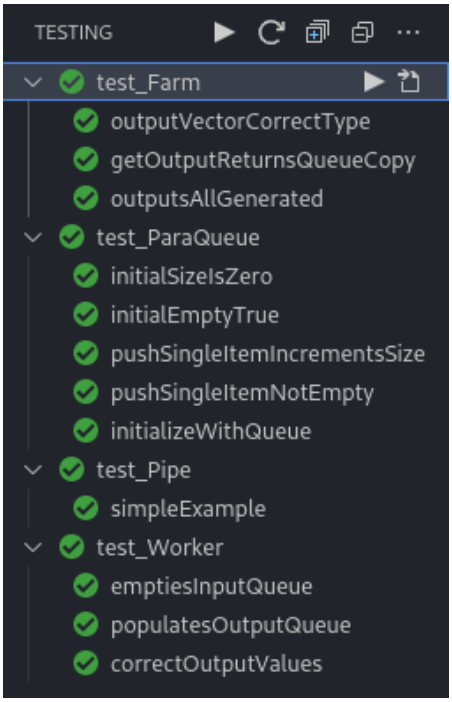

Figure 1: Boost.Test results showing each passed without error.

4 Evaluation

Note: Tests conducted on the same machine described in Figure 2.

```
pc9-008-l
--------------
OS: AlmaLinux 9.3 (Shamrock Pampas Cat) x86_64
Host: Micro-Star International Co., Ltd. PRO B660M-A DDR4 (MS-7D43)
Kernel: 5.14.0-362.24.1.el9_3.x86_64
Shell: bash 5.1.8
Resolution: 1920x1080, 1920x1080
CPU: 12th Gen Intel i5-12400 (12) @ 4.400GHz
GPU: Intel Alder Lake-S GT1 [UHD Graphics 730]
GPU: NVIDIA GeForce RTX 3060 Lite Hash Rate
Memory: 6367MiB / 31625MiB
```

Figure 2: Evaluation machine specs (generated using `neofetch` [2])

4.1 Convolution

A performance analysis and comparison was conducted using the `convolution.cpp` example with different implementations. The test processed 100 images created with the `create_inputs` command.

The sequential version took an average of 15.2017 seconds.

4.1.1 para-pat Farm

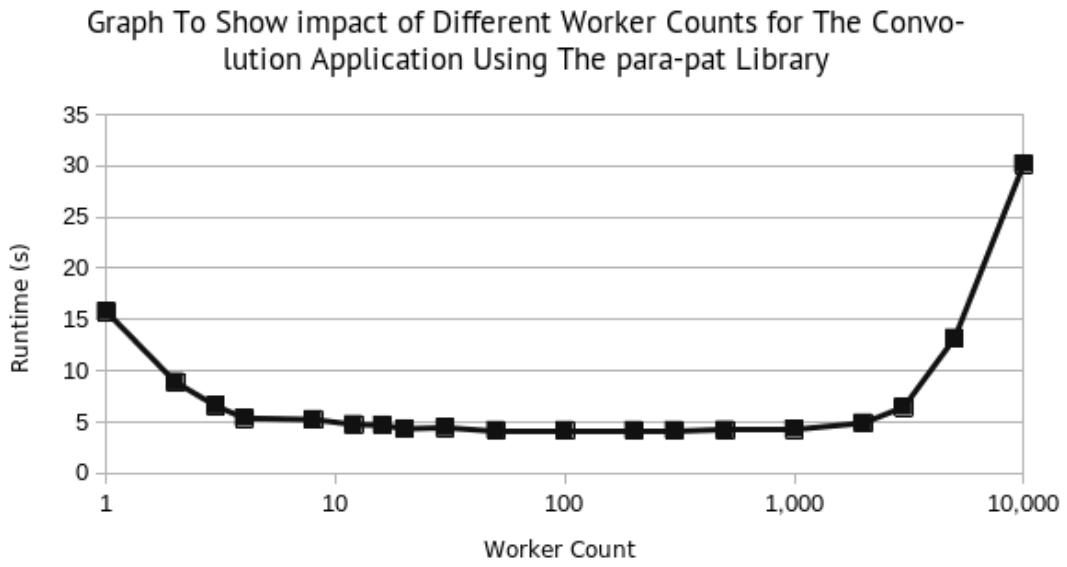

Figure 3: Graph showing the results of `convolution_parapat` with different worker counts. Worker count shown on a logarithmic scale, against runtime in seconds.

A simple Farm implementation was first conducted with a broad range of worker numbers. Here, the input queue is sequentially initialized, and the resource intensive task then ran in a Farm.Figure 3 shows a significant performance increase from the sequential version. Optimal performance was found just below the number of cores available on the machine (around 8 workers), enabling all cores to be used, whilst minimising overhead from redundant worker creation. Interestingly, no significant performance degradation was found until around 1000 workers. This is likely due to the redundant threads immediately ending when ran as they have no input so they only take cleanup resources.

The best case performance was 4.17176 seconds with 300 workers.

4.1.2 para-pat Farm-Streamed

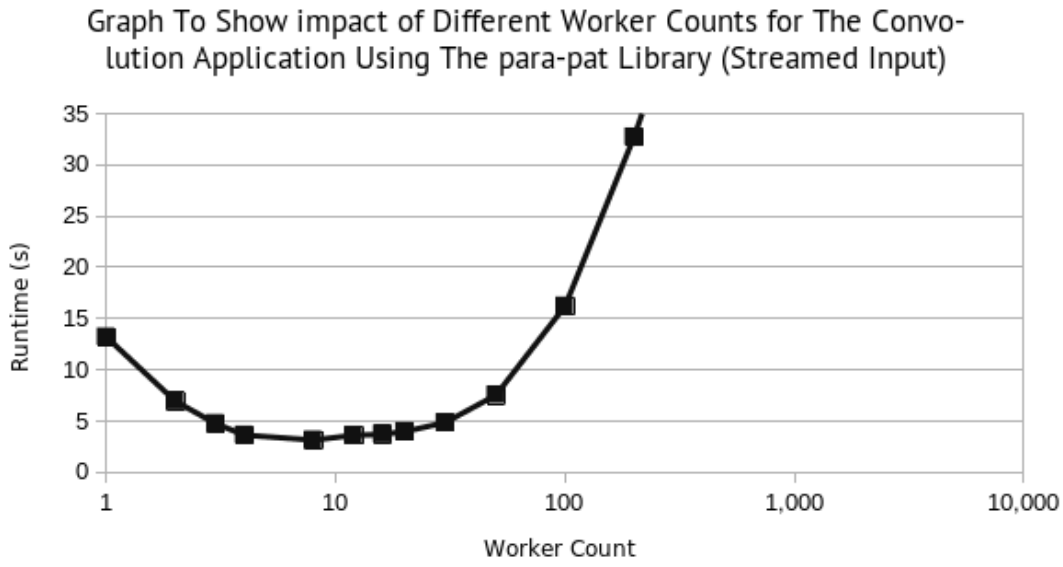

Figure 4: Graph showing the results of `convolution_parapat_stream` with different worker counts. Worker count shown on a logarithmic scale, against runtime in seconds.

An alternative implementation of the farm uses input streaming whilst the farm runs. This required calling `farm.set_stop_on_empty(false)`, and only setting true when all input has been passed in. The results in Figure 4 shows that using 6-8 workers outperformed the previous example. However, performane degrades quickly, due to the busy-waiting when input queues are empty. The best-case was the overall best performing at 3.14385 seconds with 8 workers. Over 4.8 times faster than the sequential example.

A possible optimisation could use thread freezing and signals to avoid busy waiting.

4.1.3 para-pat Pipe

Next, a Pipe implementation was used. The first Worker fetches the name of the image, the second reads and masks the image, and finally processes the images. This took an average of 13.34416 seconds (a slight improvement from the sequential version). This is most likely less performant as the main computation is in the final stage.

4.1.4 para-pat Pipe-Farm

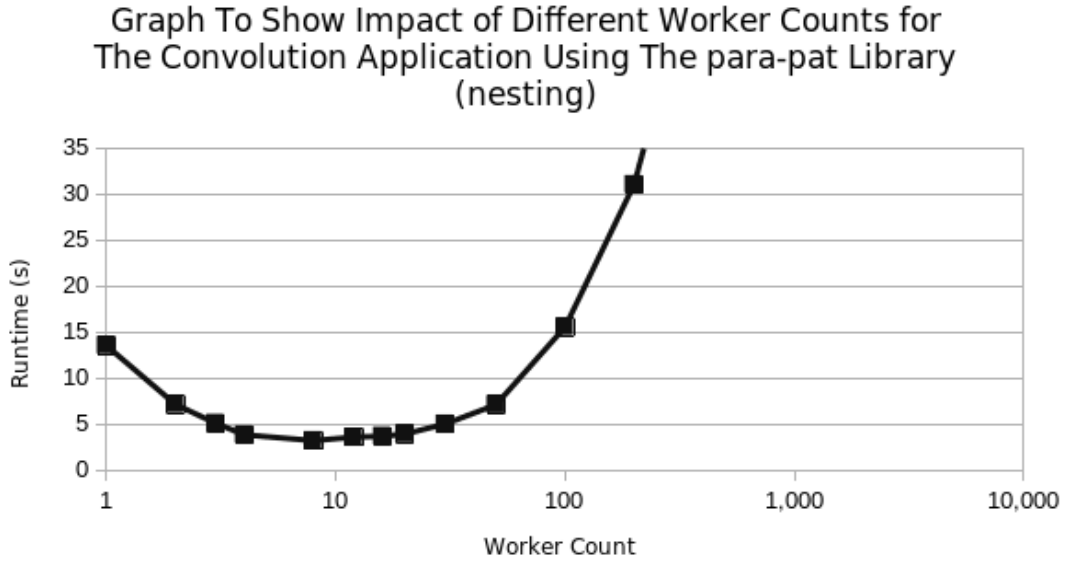

Figure 5: Graph showing the results of `convolution_parapat_nest` with different worker counts. Worker count shown on a logarithmic scale, against runtime in seconds.

The final para-pat implementation nests the final stage Farm into the Pipe. This had no significant performance benefit vs. the Farm alone, likely due to stages 1 and 2 being very small. The best-case performance was 3.22042 seconds with 8 workers.

4.1.5 FastFlow Farm

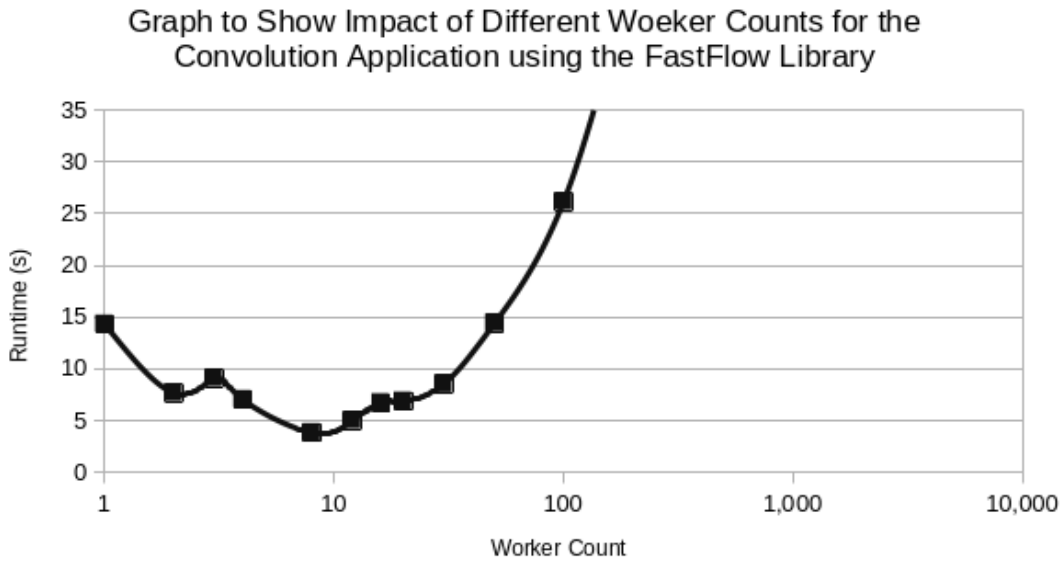

Figure 6: Graph showing the results of `convolution_fastFlow_farm` with different worker counts. Worker count shown on a logarithmic scale, against runtime in seconds.

Figure 6 shows the test results of the convolution program using the FastFlow[1] library's Farm implementation. The best-case performance was 3.84094 seconds with 8 workers.

This is significantly slower than the streamed Farm para-pat best case (see. Figure 4, and degrades with too many workers much faster than the queued para-pat Farm implementation (see. Figure 3).

5 Conclusion

Overall, all objectives were met for this practical. A library implementing Farms and Pipes was created. This was evaluated with different configurations and mixtures. It was compared to the Fast-Flow existing library and was found to be competitive.

References

[1] Marco Aldinucci et al. “Fastflow: High-Level and Efficient Streaming on Multi-core”. In: *Programming multicore and manycore computing systems*. John Wiley & Sons, Ltd, 2017. Chap. 13, pp. 261–280. DOI: https://doi.org/10.1002/9781119332015.ch13.

[2] *Neofetch 7.1.0*. URL: https://github.com/dylanaraps/neofetch/releases/tag/7.1.0. (accessed: 31-03-2024).