Andrew Downie
0786342

CIS*3090 – Assignment 1

Impressions of Pilot

My impressions of working with Pilot were very positive. Only once or twice do I remember thinking "huh, that's strange, why are things done like that?" Everything was straight forward as far as setting up the Pilot side of the program was concerned. I can't really think of a better way of making parallel programming intuitive while still giving the major benefits of parallelization. As far as I'm concerned the Pilot library does a good job of giving novice parallel programmers access and introduction to parallel programming.

Will Pilot serve me as a good platform of working knowledge to use while pushing further into the subject of parallel programming? Well I have no idea, I am after all, a novice parallel programmer. I suppose I'll likely be able to find an answer to this in future assignments.

As far as the two things that made me go "huh, that's strange…" They were both pretty trivial. The first was that there was no clear way to simply send a message without data, signalling to main that a worker is ready for more data. In my program I had the worker sending main a single char to signal that the worker was ready to do more. There may be a way to send a signal that does not also send data that I'm not aware of. Or maybe Pilot is setup to encourage the avoidance of designing programs like this.

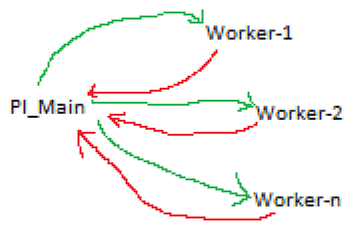
The second thing that made me go "huh…" was that PI_Select is not a queuing system. This become very obvious to me, because of how poorly the work gets distributed in my program, and in most cases only the first two workers will ever be given work to do. However, despite knowing it was not a queuing system from experience, I had still initially designed my program assuming that PI_Select was a queuing system, a simple addition to the PI_Select description saying "is not a queue underneath" or something could make this much clearer to many Pilot beginners.

Because of my initial assumption of PI_Select being a queue underneath, I encountered strange errors. I had setup a for-loop to go through the indexes of the worker-to-main channels. In that for-loop I placed a PI_Select, each time a channel was selected and then read from, the for-loop indexer would increment. However because PI_Select is not a queuing system like I originally thought, I was finding that I was getting messages about how my sends and reads did not match. This was because the PI_Select could select the same few channels over and over, until the for-loop indexer was exhausted and then move to new code prematurely as far as the untouched channels were concerned. This was fixed by forcing a sequential read through the channels.

My applications architecture and work distribution

My applications architecture and work distribution is pretty simple. In serial mode, the program just reads until it has a line from the input file, checks if the line is a palindrome, and then if the line is a palindrome, prints the lines number to the output file. This continues until there is nothing left to be read from the input file.

The parallel program has PI_Main reading in from the file, and sending data to workers as soon as it collects a number of lines. When PI_Main has collected a number of lines, it PI_Select's the next worker to send the data to. The selected worker is sent the line number of the first line it will receive. The worker is then sent the actual lines from the file right after. The worker will then parse the data into lines, and keep track of what line it is currently on, and what line numbers had palindromes on them. When PI_Main reaches the end of the file, PI_Main sends an end of data message to all the workers. When a worker receives end of data, it will send PI_Main all the lines it's been keeping track of then exit. When PI_Main receives the lines a worker has been keeping track of, it writes this info to the output file right away.



Channel Setup

Channels in red are bundled, so main can perform a PI_Select to find the next worker available to do more work

Pseudo Code

PI_Main:

```
while(inputFile != EOF)
  read line from inputFile
  linesRead++
  if linesRead >= LINES_BEFORE_SENDING
    PI_Select next worker
    send data to selected worker
    linesRead = 0

tell workers they are done

for each worker
  get workers data
  append workers data to outputFile
```

Worker:

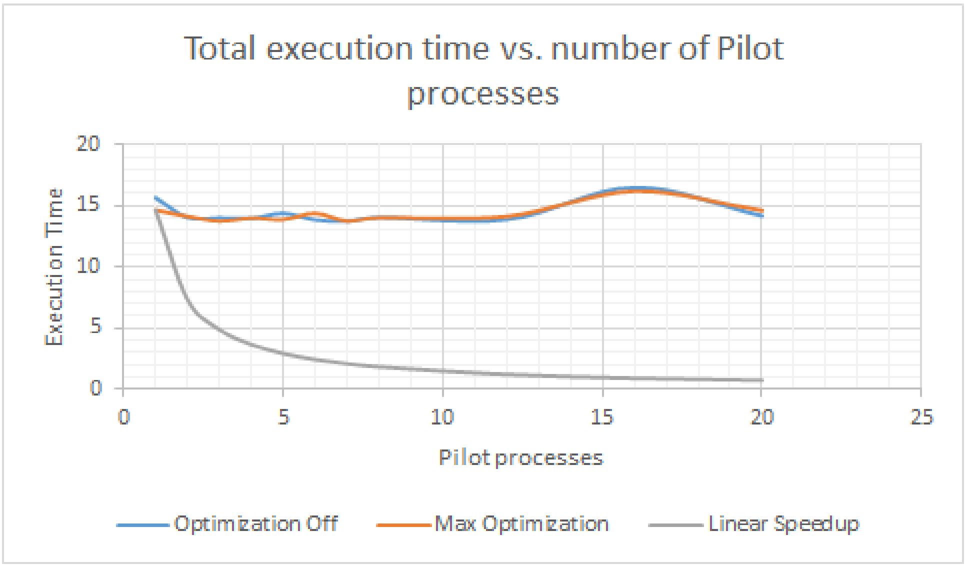
```
while(1)
  signal main I am ready to work
  read data from main
  if data == END_OF_DATA
    break
  for each line in data
    if line is pallindrome
      add line to FoundPallindromes

send PI_Main FoundPallindromes
```

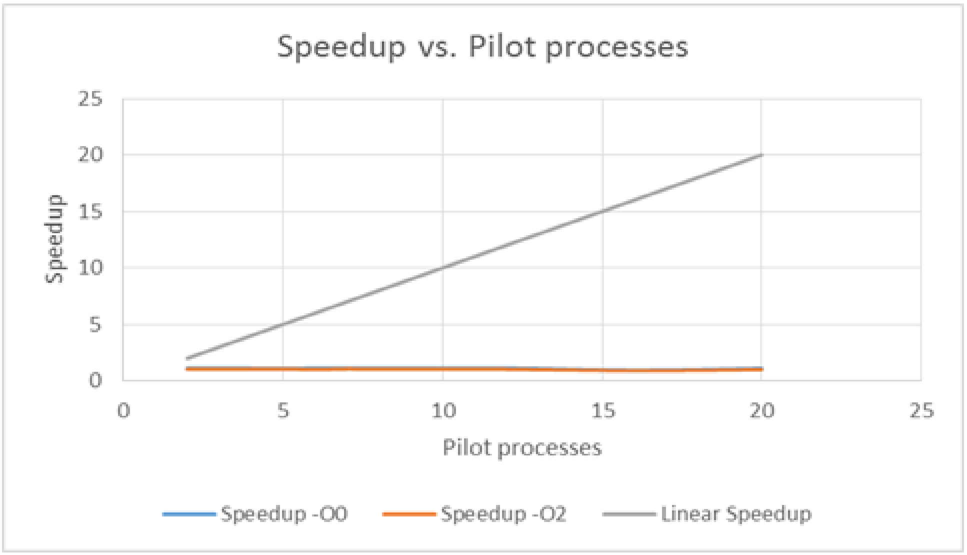
Graphs and program performance

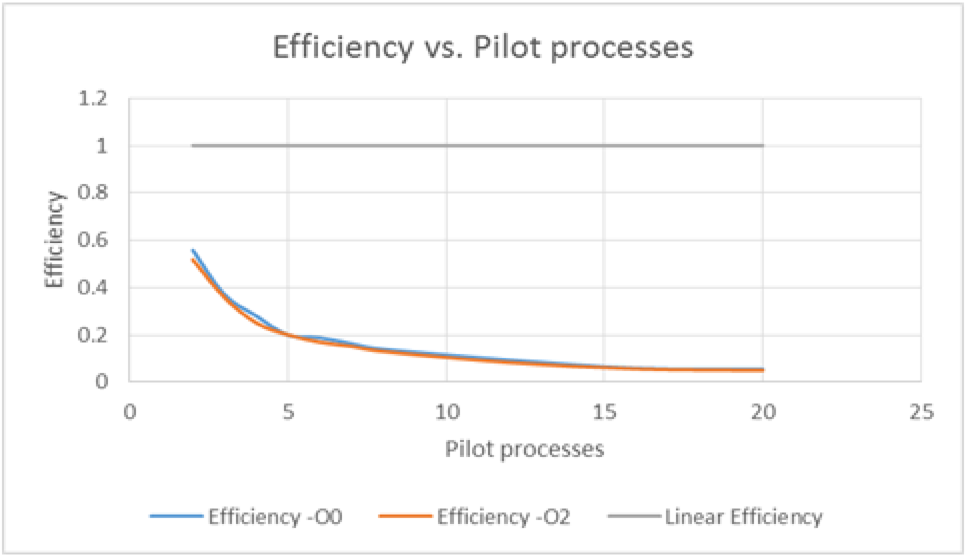
My program performed very poorly. In fact my program did not scale at all. The only reason I can think of that would cause the results below would because of the way my program reads the data in from a file as it processes, making the time spent reading in more data dwarf the time spent processing the data.

If I were to do this again I would have it read the entire file in, in main, and then send a char** representing all the lines of the file, to either the serial or parallel algorithm. This way the portion of the program that cannot be made parallel is removed from the portion of the program that can be made parallel. Then reading in the file can be measured separately from the actual algorithms, allowing a much more direct comparison between the serial and parallel algorithms. As it now the workers seem to be able to process the lines much faster than PI_Main can read them in and send them.

That said I don't think my design is bad, it's just bad for this case. I think my design would work much better for a case where the processing time is much more significant vs. the amount of time it takes to read data in from the file. However I find it very strange that my 2 process (main + 1 worker) does not perform noticeably worse than the serial. I chalk this up the file reading time dwarfing processing time but this is not at all what I expected.







Did optimization make a difference

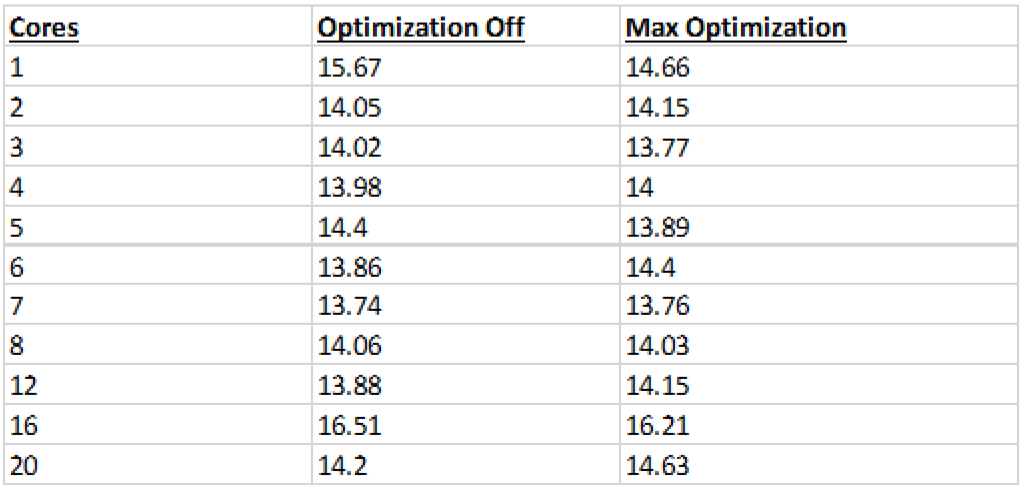
No, at least not as far as I can tell. Running with full optimization did *seem* to make the times between each equivalent run seem more consistent, but that's likely just luck in errors.

Optimization certainly did not noticeably improve program execution time, outside of reasonable margin of error between runs. Nor did optimization affect speedup or efficiency in any meaningful way. All versions of my program with the strings-10M file seem to run at ~14 seconds.

Raw Data

Cores	Optimization Off	Max Optimization
1	20.16, 11.24, 14.14, 16.48, 14.09, 16.09, 16.19, 15.24	14.17, 14.66, 14.65, 16.52, 14.30
2	13.81, 14.05, 14.12	13.92, 15.16, 14.15
3	13.93, 12.01, 14.24, 14.11	13.97, 13.71, 13.77
4	19.44, 14.06, 13.84, 13.89	13.77, 14.00, 14.02
5	14.4	>25, >25, >25, 13.95, 13.89, 13.82
6	16.07, 13.70, 17.27, 13.86, 13.72	14.4
7	13.74	13.76
8	15.01, 13.95, 13.89, 14.16	14.03
12	13.88	14.15
16	17.75, 18.04, 16.51, 14.12, 14.95, 14.12, 18.05	17.48, 16.68, 19.83, 15.68, 16.76, 13.96, 13.98, 16.21, 14.13
20	14.31, 14.20, 14.08	13.31, >25, >25, 16.03, 14.54, 14.63, 14.14

Median Data



Cores	Optimization Off	Max Optimization
1	15.67	14.66
2	14.05	14.15
3	14.02	13.77
4	13.98	14
5	14.4	13.89
6	13.86	14.4
7	13.74	13.76
8	14.06	14.03
12	13.88	14.15
16	16.51	16.21
20	14.2	14.63