CIS 520
Project 4: One Program, Three Ways
Design Document

Chase McCormick
Kevin Kellerman
Jordan Martin

Background and Motivation

Given a 1.7 GB text file, write a program to find the largest common substring between any sequentially ordered strings between new lines. The concern of optimization for this program is not in the searching algorithm for substrings, but rather in how the work can be parallelized using the following three parallelization implementations:

- Pthreads
- OpenMP
- MPI

Experimental Setup

Our experiments were run across the Elves compute nodes on K-State's Beocat system. The processors across these nodes were either 2x 8-Core Xeon E5-2690 or 2x 10-Core Xeon E5-2690 V2. RAM varied between 64GB, 96GB, and 384GB, though only up to 32GB was used on the processes. On the network side, the NICs used were 4x Intel I350, with 10GbE and QDR Inifiniband on Mellanox Technologies MT27500 Family ConnectX-3. The hard drive used was a 250GB 7200 RPM SATA drive on all of the nodes.

Analysis of Results

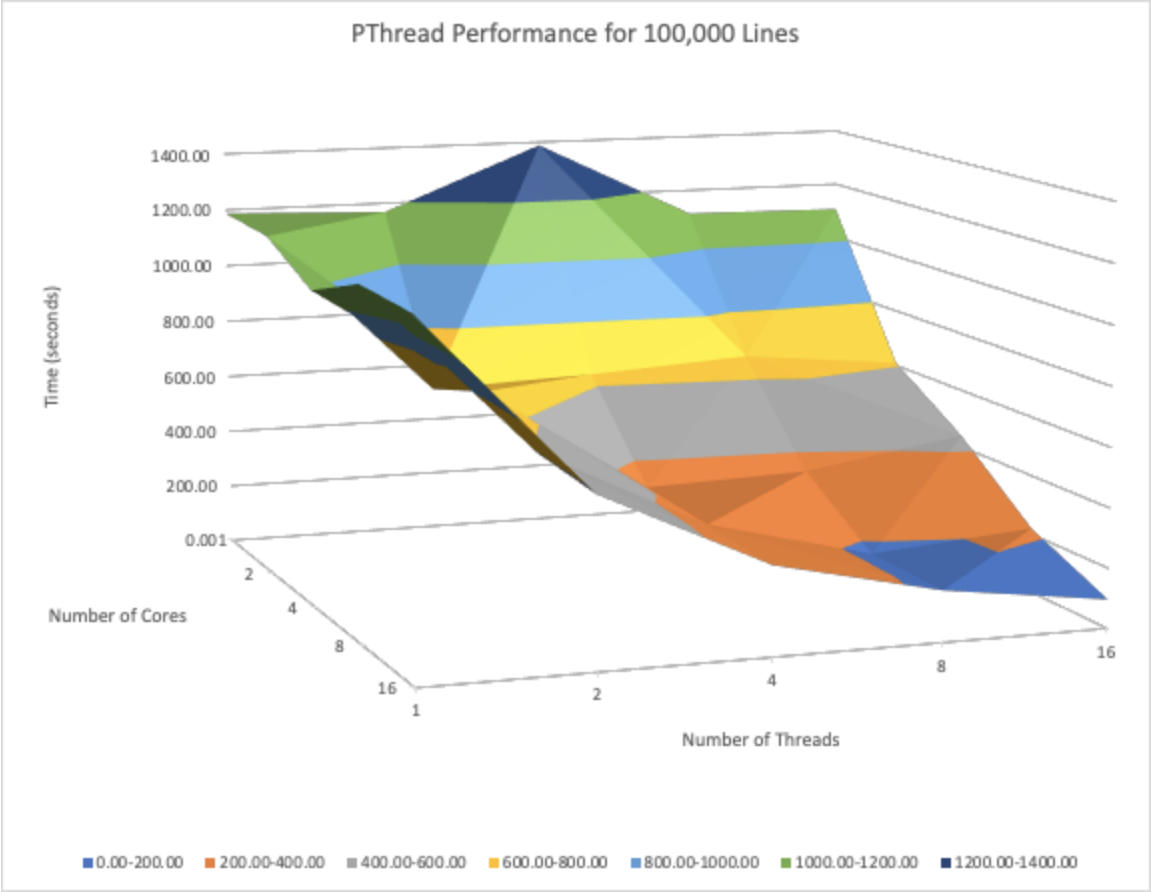
An analysis of various versions was performed with control subjects of different ranges of inputs and number of cores to visualize CPU efficiency and memory utilization on Beocat.

PThreads Implementation

The initial PThread implementation involved using the baseline algorithm to calculate the substring between a pair of lines and immediately print that results, as long as the thread running was next in line. Otherwise, it would yield the processor so the next thread in order could run. However, this proved to be inefficient, as processors were waiting to do meaningful work while they waited for the I/O to become available. So, this was changed to have each thread do all of their substring computations first, store them in a preallocated results array of strings, and then print all of their results, so long as they were the thread in order ready to print. If it was not that thread's turn to print, it would yield to allow threads to print in the proper order.

PThread differed from a single thread implementation in that each thread had to be created and joined to a list of threads. From there, each thread would run the lcs_threading() function, passing the thread's array, inside of which it would loop through calculating substrings, then loop through printing results. At the end of printing results, the thread increments the global value nextThread, which indicates the ID of the next thread that should print. Upon completion of this function, the thread exits. When all threads have exited, the PThreads list is destroyed, arrays freed, and the elapsed time is printed.

One note is that the printing of substrings on each calculation used significantly less memory than the preallocation. It allowed results to be freed, rather than stored across the duration of the thread. This meant that only 4GB, or perhaps even 2GB needed to be used, regardless of the number of lines being read. The downside was that this was much slower. Preallocating the array meant for faster loops and more efficient pipelining, with the downside that the results array was large to allocate and needed up to 16GB to run.

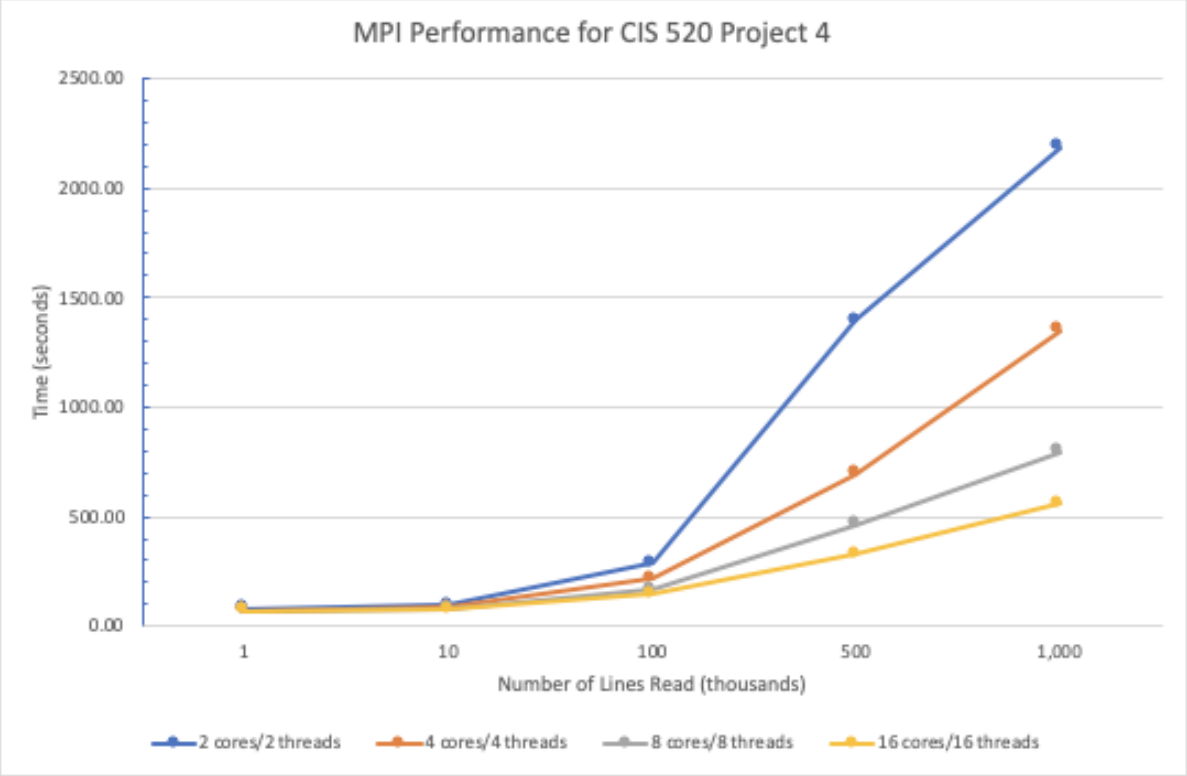


MPI Implementation

The basic implementation of MPI was similar to PThread, but MPI in concept is significantly different than PThread. In MPI, you have to consider that there are multiple processes running, not just threads, so it is more difficult to introduce shared variables. For MPI, each process would do all of its substring calculations, same as before. When the process is done, if it's not process 0, it loops waiting for a message from the process with an ID below its own. This is done to ensure that processes print their results in order, starting with process 0. When the final process prints, it sends a message to process 0, indicating that process 0 can continue on in the main loop and print the elapsed time. Note that when process 0 finishes with substrings, it returns to the main loop and waits for a message to be received from that final process.

In this implementation, MPI doesn't necessarily run multiple threads, but multiple processes across multiple cores. While this can take significantly more computational power, it results in drastic increases in performance efficiency for larger data sets. The downside to this is that, since processes are separate, there are no shared results

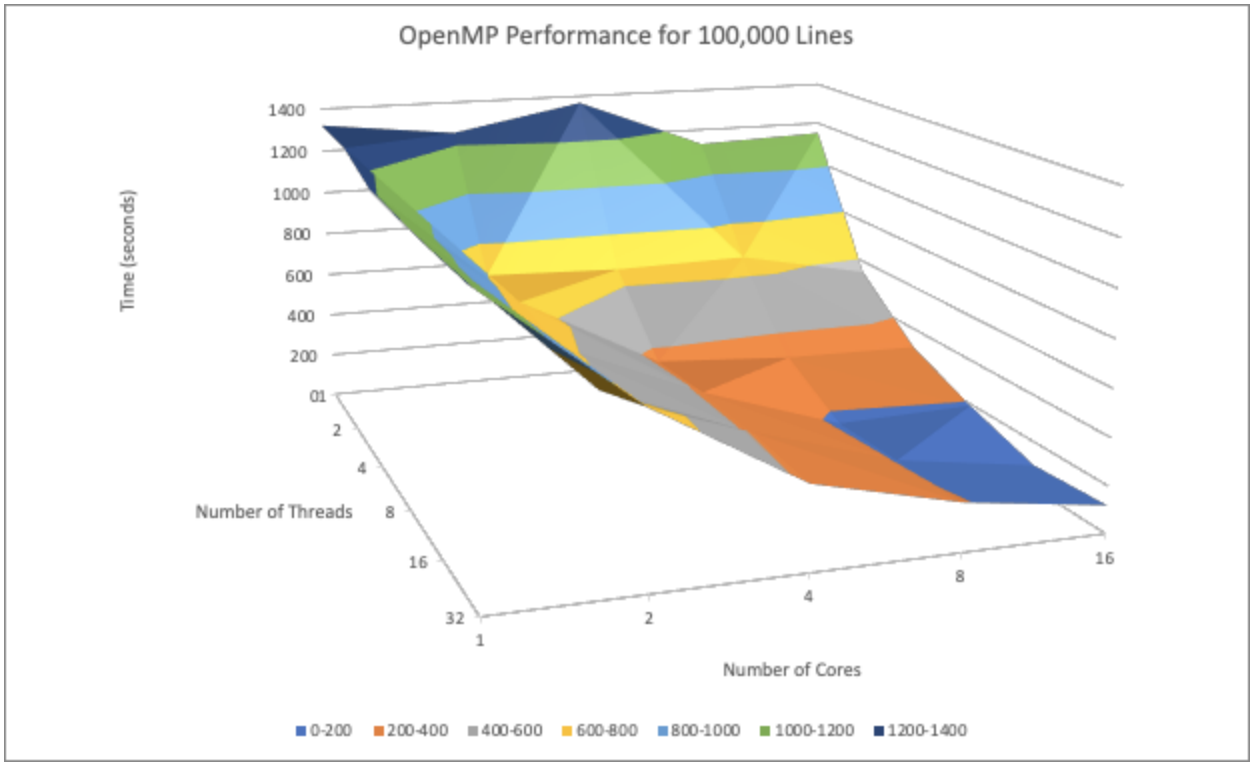
arrays, meaning each process has its own array of results. Because of this, more RAM is needed, in some cases up to 32GB for the 8-core, 1M line tests.



OpenMP Implementation

OpenMP is very, very similar to Pthreads, but it is higher level, so it is easier to implement. This, of course, comes with the trade-off of less granular control. Out of the three implementations, this was the easiest to implement and that is reflected in our code.

In general, like the other schemes, adding cores for the openMP implementation increased performance. The statistic with this implementation that jumped out the most was that peak performance was found when an equal number of threads and cores were running, with 16 cores and 16 threads being the most efficient. When OpenMP was tested, it was only given 4 hours to run through all of the tests. As a result, tests with problem sizes of 500k and 1 million reached a timeout when 2 or less cores were used. Nevertheless, a sufficient amount of data was still presented to see the strengths and weaknesses of the OpenMP library. In comparison, the results found for PThreads and OpenMP appear to be similar, whereas MPI offered different results.



Conclusion

As expected, each implementation was better when cores or threads were increased and by giving more processing power and parallelizing the code, performance was better. Overall, for the 100,000 lines that we used in our graphs and analysis, we were able to get the best performance out of pthreads with 16 cores and 16 threads at 96.7 seconds. Second place was OpenMP with 16 cores and 16 threads at 97.8. This was to be expected as they are functionally very similar. Finally, MPI at 16 cores and 16 threads took 143.6 seconds, so it was considerably worse for this dataset.

This project gave us a great look at each of the three implementations, so that in the future, we have a better understanding of which we prefer. In general, we felt that OpenMP was the easiest out of the two better implementations, so we would likely recommend it for future projects.

Appendix

First 100 Lines:

0-1: </title><text>\'\'\'aa
1-2: </text> </page>
2-3: }}</text> </page>
3-4: <page> <title>a
4-5: \n|foundation = 19
5-6: <page> <title>abc_
6-7: <page> <title>abc_
7-8: the [[australian broadcasting corporation]]
8-9: the [[australian broadcasting corporation]]
9-10: [[australian broadcasting corporation]]
10-11: </title><text>{{infobox
11-12: <page> <title>ab
12-13: </title><text>\'\'\'ab
13-14: </title><text>\'\'\'a
14-15: <page> <title>ac
15-16: <page> <title>acc
16-17: \n\n{{disambig}}</text> </page>
17-18: }}</text> </page>
18-19: \n\n{{disambiguation}}</text> </page>
19-20: </title><text>\'\'\'ac
20-21: <page> <title>ac
21-22: <page> <title>ac_
22-23: <page> <title>ac_
23-24: </text> </page>
24-25: \'\'\' may refer to:\n
25-26: \'\'\' may refer to:\n\n
26-27: <page> <title>ad
27-28: <page> <title>ad
28-29: <page> <title>a
29-30: \n\n{{disambig}}</text> </page>
30-31: <page> <title>afc
31-32: <page> <title>af
32-33: </text> </page>
33-34: <page> <title>a

34-35: </title><text>{{infobox
35-36: </title><text>{{
36-37: <page> <title>a
37-38: <page> <title>aid
38-39: <page> <title>ai
39-40: [[australian institute of
40-41: <page> <title>a
41-42: \n\n{{disambig}}</text> </page>
42-43:]]\n\n{{disambig}}</text> </page>
43-44: </text> </page>
44-45: n-stub}}</text> </page>
45-46: </text> </page>
46-47: </text> </page>
47-48: <page> <title>alar
48-49: <page> <title>al
49-50: <page> <title>alp
50-51: <page> <title>a
51-52: <page> <title>am
52-53: <page> <title>am
53-54: <page> <title>am
54-55: <page> <title>am
55-56: }}\n\n{{defaultsort:am
56-57: </title><text>{{unreferenced
57-58: class=\"wikitable\"
58-59: <page> <title>an
59-60: <page> <title>a
60-61: <page> <title>a
61-62: <page> <title>ap
62-63: <page> <title>ap
63-64: <page> <title>ap
64-65: <page> <title>ap
65-66: \n\n==external links==\n*
66-67: <page> <title>ap
67-68: </title><text>{{
68-69: {{cite web|url=http://www.
69-70: <page> <title>ar
70-71: <page> <title>a
71-72: ==\n{{reflist}}\n\n==
72-73: <page> <title>asa_

73-74: <page> <title>as
74-75: <page> <title>as
75-76: <page> <title>as
76-77: <page> <title>as
77-78: </text> </page>
78-79: <page> <title>as
79-80: american society for
80-81: [[association fo
81-82: \'\'\' is a [[france|french]] [[association football]]
82-83: <page> <title>a
83-84: <page> <title>at
84-85: <page> <title>at
85-86: <page> <title>at
86-87: <page> <title>at
87-88: <page> <title>a
88-89: {{cite web|url=http://www.
89-90: <page> <title>a
90-91: <page> <title>a
91-92:]]\n\n{{disambig}}</text> </page>
92-93: }}</text> </page>
93-94: </title><text>{{
94-95: {{unreferenced|date=
95-96: <page> <title>a_b
96-97: <page> <title>a_be
97-98: <page> <title>a_be
98-99: 2012}}\n{{infobox album
99-100: name   = a big
100-101: </title><text>{{infobox