Design Document 4

Braden Schroeder

Nils Peterson

Caleb Holland

Software Architecture

OpenMP

OpenMP starts in main and almost immediately calls getLines() in order to get all of the lines out of the input file. These are then passed into doLongestSub(). In doLongestSub(), the subStrings data structure to hold the substrings is malloced and omp_set_num_threads() sets the number of threads that the program will be broken up into in the next line. This #pragma function does a lot of the multithreading logic for you, which is really convenient. We pull the thread num out and use that to choose the start and end lines that this thread will compute. These are iterated through and used to call longestSub() where we use dynamic computation to build a table of longest substrings. We pull out the longest substring, free up our table space, and return back to doLongestSub() where we store our substrings. Once all of these substrings are computed, we return to main and print all of our substrings.

PThreads

PThreads starts by entering main, creating an array of threads, and initializing their settings. The lines from the input file are also read in. We calculate the amount of lines that each thread should run, then we create these threads. These threads start in doLongestSub() where we iterate through their lines to calculate and call longestSub() on these. This dynamically calculates the longest substring using a table to keep track of the length of substrings. We free the table then use the index i and the length to pull out the substring and put it into our list of substrings. These threads meet back up at pthread_join and the list of substrings is iterated through and printed.

MPI

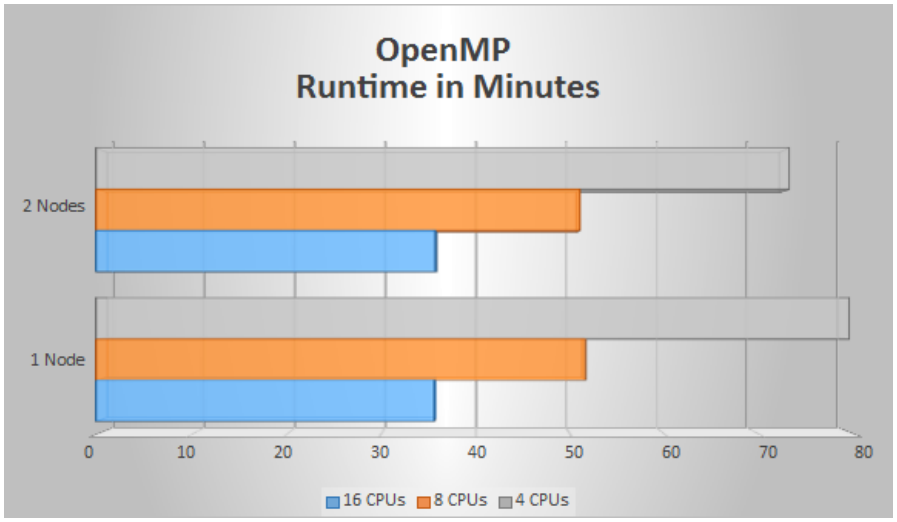
MPI starts in main(); however, unlike the other two algorithms, MPI brings in argc and argv to be used in computation. Our algorithm then goes into getLines() in order to read in all of the lines from the input file that will then be used to find the shortest substrings. We pass these

lines as well as argc and argv into doLongestSub() to start the multithreading process. We malloc the variable that will hold the substrings and initialize out multithreading using MPI_Init() and pass in argc and argv. We then pull the number of threads and the rank of the threads using MPI_Comm_size() and MPI_Comm_Rank(). We do this because this information goes straight from the script into argv and into the initialization call and is never hard-coded in like the rest of the multi-threading methods. We use this information to find the lines that each thread will run and calculate the substrings without longestSub() method. This, as mentioned earlier, dynamically builds a table to calculate and keeps track of the largest substring. After this we use MPI_Barrier() to halt the execution of all the threads until they all meet. Now that they have all met up, we allow thread to print. When it is done, it signals thread two to print. Thread two signals back to thread one who signals thread three and so on. This is done using the MPI_Send() and MPI_Recieve() functions. Finally, we finalize MPI and return to main().

Performance Analysis

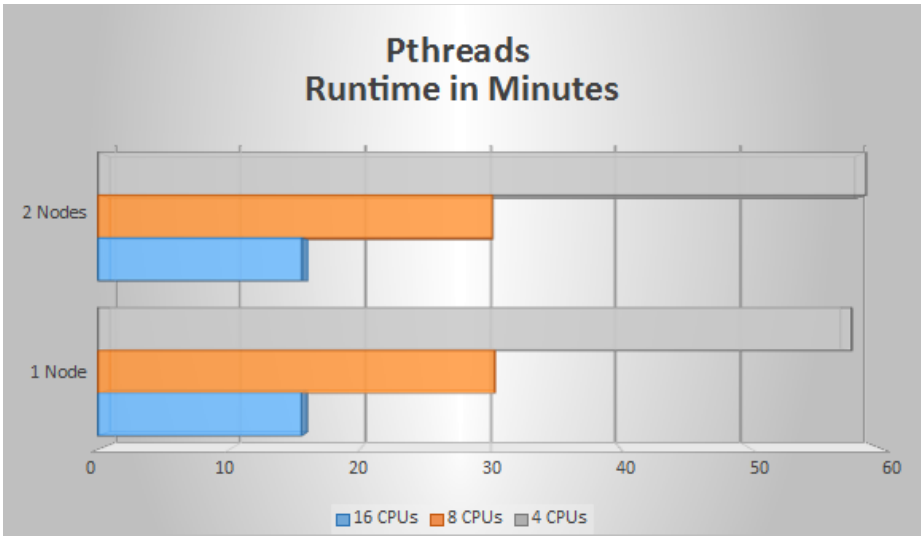
Performance Analysis by Core

During the course of our project we discovered that the more cores you use, the faster your program will run. You can see an example of this when viewing the run times of OpenMP run over one node. When run with 4 cores, this program took ~80 minutes to run. When run over 8 cores, this same code took ~52 minutes to run. Finally, the 4 core run took ~36 minutes to run. This is a significant improvement on the ~70 minutes that it took to run the 4 core.



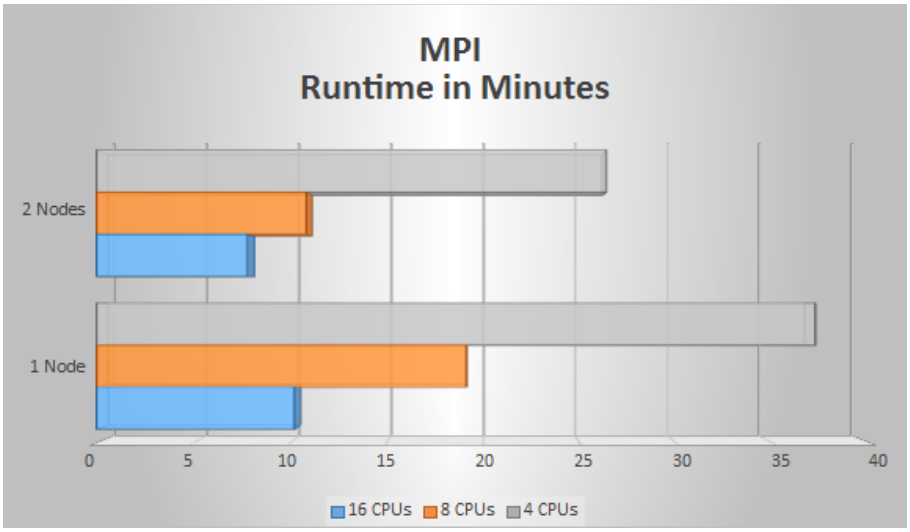
The same goes for any program run using two cores. We can see this in our running of PThreads where there was a very similar reaction to the number of cores running. When

Pthreads was run on 4 cores is takes ~59 minutes. We see an increase in 8 cores at ~30 minutes and then at 4 cores with ~15.5 minutes. In summary, the more cores that you choose to run your program, the faster it becomes. This stands for all three kinds of multithreading as well as all amounts of nodes. However, this comes with the trade off of scheduling issues. The more resources that your program demands, the longer it will take to get those resources. We experienced wait times while working with the software where we would have gotten our results sooner by utilizing less cores than we had requested.



Performance Analysis by Node

In regards to nodes, we did not get the performance results that we had expected to. When we ran OpenMP with 4 cores, 8 cores, and 16 cores all on one and two nodes. We got about the same results. We repeated this same process for pthreads and came to the same result again. However, when we repeated this process on MPI we get a completely different result. When we ran MPI with one node, we found 4 cores to run for ~37 minutes, 8 cores to run for ~19 minutes, and 16 cores to run for ~10



minutes. The second core improved these times greatly, with 16 cores running at ~8 minutes, 8 cores running at ~11 minutes, and 4 cores taking ~26 minutes. We believe that MPI worked so much better with two nodes than Pthreads and OpenMP because of the lack of abstraction. Because the libraries did not do a lot of the work for us this decreased overhead and allowed the program to run faster. Pthreads and OpenMP had no change in results from one node to two because overhead outweighed the increase in nodes.

Example Output (first 100 lines or so)

```
</title><text>\'\'\'aa
<page> <title>aa
}}</text> </page>
<page> <title>a
\n|foundation = 19
<page> <title>abc_
<page> <title>abc_
 the [[australian broadcasting corporation]]
 the [[australian broadcasting corporation]]
 [[australian broadcasting corporation]]
</title><text>{{infobox
<page> <title>ab
</title><text>\'\'\'ab
</title><text>\'\'\'a
<page> <title>ac
<page> <title>acc
\n\n{{disambig}}SOH</text> </page>
}}SOH</text> </page>
\n\n{{disambiguation}}SOH</text> </page>
</title><text>\'\'\'ac
<page> <title>ac
<page> <title>ac_
<page> <title>ac_
</text> </page>
\'\'\' may refer to:\n
\'\'\' may refer to:\n\n
<page> <title>ad
<page> <title>ad
<page> <title>a
\n\n{{disambig}}SOH</text> </page>
<page> <title>afc
<page> <title>af
<page> <title>af
<page> <title>a
</title><text>{{infobox
</title><text>{{
<page> <title>a
<page> <title>aid
<page> <title>ai
[[australian institute of
<page> <title>a
\n\n{{disambig}}SOH</text> </page>
]]\n\n{{disambig}}SOH</text> </page>
<page> <title>aj
n-stub}}</text> </page>
<page> <title>ak
</text> </page>
<page> <title>alar
<page> <title>al
<page> <title>alp
```

```
<page> <title>al
<page> <title>alp
<page> <title>a
<page> <title>am
<page> <title>am
<page> <title>am
<page> <title>am
}}\n\n{{defaultsort:am
</title><text>{{unreferenced
class=\"wikitable\"
<page> <title>an
<page> <title>a
<page> <title>a
<page> <title>ap
<page> <title>ap
<page> <title>ap
<page> <title>ap
\n\n==external links==\n*
<page> <title>ap
</title><text>{{
{{cite web|url=http://www.
<page> <title>ar
<page> <title>a
==\n{{reflist}}\n\n==
<page> <title>asa_
<page> <title>as
<page> <title>as
<page> <title>as
<page> <title>as
<page> <title>as
<page> <title>as
american society for
 [[association fo
\'\'\' is a [[france|french]] [[association football]]
<page> <title>a
<page> <title>at
<page> <title>at
<page> <title>at
<page> <title>at
<page> <title>a
{{cite web|url=http://www.
<page> <title>a
<page> <title>a
]]\n\n{{disambig}}SOH</text> </page>
}}SOH</text> </page>
</title><text>{{
{{unreferenced|date=
<page> <title>a_b
<page> <title>a_be
<page> <title>a_be
 2012}}\n{{infobox album
 name        = a big
</title><text>{{infobox
</title><text>{{infobox
\n| image
\n| image
</title><text>{{refimprove|date=i
```