Project 4
One Program, Three Ways

Derek Pockrandt and Benjamin Miller
Operating Systems
Kansas State University
Manhattan, Kansas 66506

Abstract

Multithreading is becoming more and more common in today's applications. From heavy industrial applications, to secure research, to complex and in-depth games, computations in parallel is becoming essential to computer science. As we reach the speed limits of core clock frequencies, parallel processing can also overcome these limitations. There are many libraries available to implement multithreading to applications today, and this project is tasked to compare and contrast three of these libraries to see which one operates best in which conditions: OpenMP, Pthreads, and MPI. Our tests have shown that each has advantages and disadvantages, and they all tackle the task of multithreading in their own way.

The problem we have solved

We designed a program that finds the longest common substring between adjacent lines. It then prints the common substrings alongside the line numbers of the articles. The file being examined contains 1.7Gb of first lines found in many Wikipedia articles.

The first iteration of the program is a serialized version that serves as a baseline for the logic of the program. This implementation then was expanded and adapted for various parallelized versions of the program, using the multithreading libraries OpenMP, MPI, and Pthreads. These multi-threaded versions were created to see how fast the program could run, and compare the running times on different core/node setups.

Why our solution is worth considering and why is it effective in some way that others are not

The tests that we have completed were consistent and were performed numerous times to ensure accuracy of information where available. Our test results proved promising suggestions that our code has effectively utilized and optimized multithreading tasks.
In addition to this, our code is robust, but uniform and easy to understand. The structure of our code suggest an easy to follow flow of information, and the method of implementation is easily adaptable to utilize other multithreading libraries that were not tested, as well as making use of special hybrid schemes, such as OMP and MPI combined.

How the rest of the paper is structured

First, we will go over the implementation in code, describing the differences between the different multithreaded libraries. There are significant portions of code that are shared between the libraries, and these are described in the next section.

We will then describe our methods of testing the algorithms, and look at an analysis of various test results compiled into graphs for each multithreaded library.

At the end of the document, we will provide our code and submission scripts as appendices for review and further discussion.

Implementation

Our Solution

For parallelizing our code, we created three separate, but functionally identical version of our code, running OpenMP, MPI, and Pthreads implementations for multi-threading.

The basic meaty code structure begins with the primary thread (often considered thread 0) reading into memory the 1.7Gb data file. Using the efficient 'getline' command available under Linux, we repeatedly dynamically allocate space for each next line. This read-reallocate can be considered slower than a prefixed allocate to prepare for more space, but considering the computation time required for this task, the read process is fairly quick, reliable, and uses memory as efficiently as possible.

Note that the file is only read if we need more lines. For example, when testing a 1k solution, only 1k lines are read, then the file is closed. All 1m lines are read only during the 1m lines tests.

A results array is prepared to store the resulting longest strings. It contains (lines-1) entries, since there are 99 comparisons with 100 lines of input.

Each thread now enters its 'thread work section,' where it will process its lines as defined by a quota generation. The work of n lines is broken down into quotas for each thread. This quota is fair and equal across all threads, but ensures end-cases are cared for during computation.

Due to the nature of this process, no synchronization is necessary, and there are therefore no critical sections in this code. A thread simply reads the lines it needs, searches for the longest substring (see below), and references the longest string in memory within the results array. Multiple threads can write to the results array at the same time safely, since it is simply an address marker whose contents are changed. It is also important to note that the contents that a thread is allowed to change depends directly on its assigned quota.

Once computation is complete, the main thread writes out the results to output (unless hushed), and all the other threads free up memory and exit. This implementation is general, and changes a bit when considering separated processes in MPI implementations.

Longest Common Substring

Once a thread has determined its quota, it simply calls 'findLongestSubstring' for each pair of strings within its quota. This implementation is equivalent across all threads and across all multi threaded libraries. This is the "busy work" section of the code that shouldn't be changed, regardless of how multithreaded an implementation gets.

Finding a common substring is broken down into two parts: equivalence search and diagonal search.

In equivalence search, an (n+1) x (m+1) table is created, where n and m are the lengths of the comparing strings a and b respectively. The table is zero initialized. The search occurs when each character in a is compared to each character in b, and if they are ever the same character, the entry in the table [i+1][j+1] is marked as a 1 as opposed to 0. The extra size in the table is part of a previous implementation that was not yet removed due to time constraints. The previous implementation summed on instant the [i][j]th item when a 1 value was found. Note in diagonal search, this isn't necessary. Upon a match being found, an additional linked list was added to containing the i and j indices of the positive match characters.

For diagonal search, after the equivalence search is complete, the linked list is searched through from start to end for each 1 value. Upon each value, the search occurs by checking consecutive lower right diagonal entries to find chains of 1’s. For example, a string “abcbcde” and “abcd” would have the following table and diagonal searches.

```
 abcbcde
a1
b 1 1
c  1 1
d     1
```

Note that 0 entries are omitted. We can see that there are two common strings of equal length, as denoted by diagonal 1 lines: “abc” “bcd”. Either one of these values could return, but “abc” will return since it was found first.

An optimization is added to the algorithm, where once an edge is searched by any diagonal search, it’s value is set to 0 to prevent future unnecessary searches. For example, if this was not the case, when we reach the first “b” in a and “b” in b, we would re-search the already searched diagonal for “abc”, looking at “bc”. Since this is a substring of “abc”, it will never be longer than “abc”, and it’s a redundant search.

The longest string index starting point (“a” in “abc”) is stored, as well as its length. The longest string is then stored in a new char*, and returned as the longest string in the two lines. Note that the linked list is used to prevent searching through an entire sparse matrix of many 0s and few 1s. The linked list allows us to jump directly to the next 1 value.

OpenMP Specifics
OpenMP is by far the simplest implementation, where a #pragma omp parallel surrounds the call to the thread work section. Each thread is given its threadId, which is used to calculate its quota. The number of threads variable is passed in by argument.

Pthreads Specifics
While still a bit more complex, pthreads is still a very simple implementation. The primary difference here is that we compile all of the required arguments to the thread work section function into one long argument variable. In a for loop, we assign a new threadId, merge it with the long arg, and start a new thread. When done, the main thread simply joins waits on all the other threads.

Due to this implementation, n+1 threads run instead of the requested n, but this is very temporary, as the main thread quickly attempts to join the other threads currently running, and we are left with n threads for the majority of the computation.

MPI Specifics
MPI was quite a bit more challenging that other implementations, simply because it required communication between completely different processes, instead of just memory sharing threads. Passing the file data and results array as pointer arguments wasn’t allowed anymore.

To overcome this limitation, the code is separated into main thread and non-main thread flows. The two flows are occasionally reconnected temporarily to share global information, synchronize execution times, and prevent other nasty bugs. The main thread, after reading in the file data, calculates each other thread’s quota, and sends them only the lines that they need.

Sending strings in MPI is tricky, let alone arrays of strings. Our implementation sends each character individually after telling the receiver how many characters to expect. The receiving, non-main thread knows its quota as well, so it knows how many sets of characters to expect. In this way, the non-main thread retrieves its distributed data required for completing its quota.

After every thread has received its data, a "start" broadcast is sent, and every thread, including the main thread, starts its thread work section at the same time. When completed, each thread waits at the "stop" broadcast until every thread is done.

Each thread in order then sends all of its resulting information back to the main thread in the same way the main thread distributed strings out. When completed, the strings are printed, a "clear" broadcast is sent, and all threads empty their memories and leave.

Obvious and required, MPI takes a bit more memory usage as compared to Pthreads and OpenMP. While the synchronizing broadcasts might not be necessary, they assist with debugging and prevent some other situational errors that are difficult to reproduce.

Evaluation

How we tested our solution

Our code was tested on K-State's Beocat Supercomputer, with the following configurations (where n stands for node and c stands for core, ie 2n4c would be 4 cores on 2 nodes each; 8 cores total): 1n1c, 1n2c, 1n4c, 1n8c, 1n16c, 1n32c, 2n4c, 4n2c, 4n4c, 16n2c, 2n16c.

We tested the implementations on different ranges of data, including 1k lines, 10k lines, 100k lines 500k lines and 1m lines. Each situation combination was tested multiples times (ranged 4 to 10 times each) to gather more accurate data. Some situations consistently failed, and are marked as such, however, most to all situations has promising and accurate results.

To schedule these tests on Beocat, we used a sequence of shell scripts, which will be provided in the appendix.

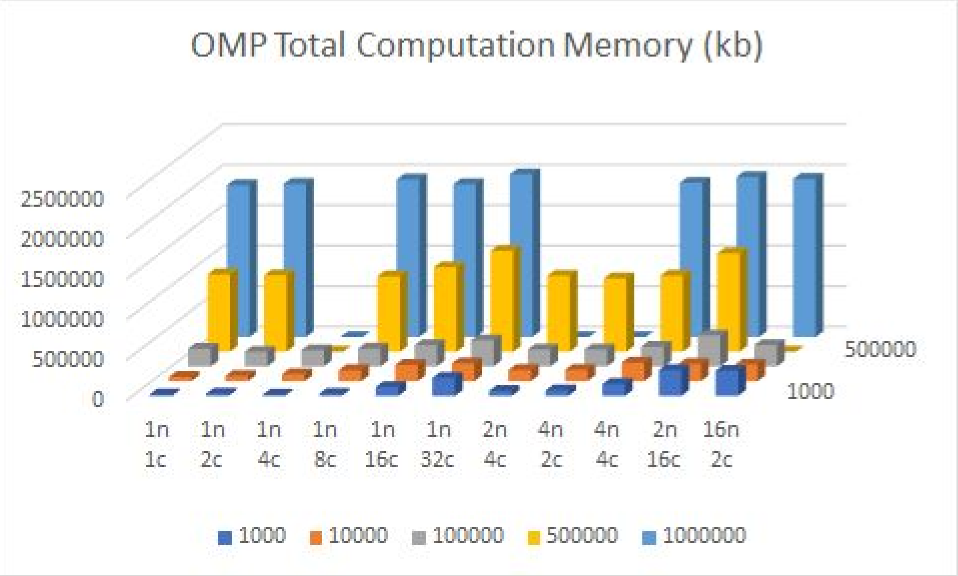
To compile multiple output results into averages to be used for 3D graph inputs, we created a simple Java application to parse the raw output from each multithreaded library. Various 3D graphs are shown below, along with an analysis of their results.

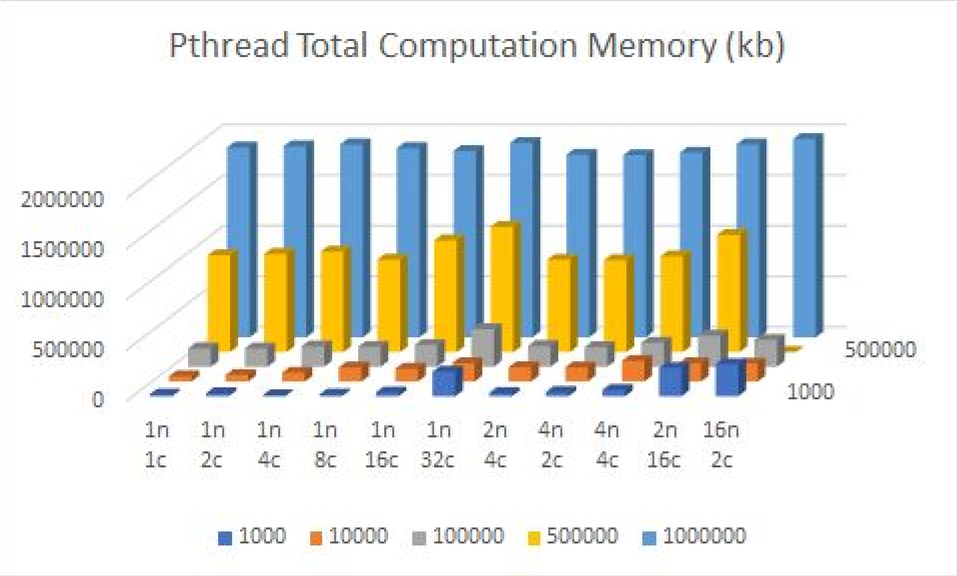
Memory Usage
Memory consumption was somewhat straightforward and predictable depending on the number of lines being computed. OMP seemed to be the most variable depending on the system, and also seems to have taken up the most memory compared to Pthreads and MPI. This makes sense considering that OMP is a very streamlined experience for the developer, there may be many checks going on in the background.

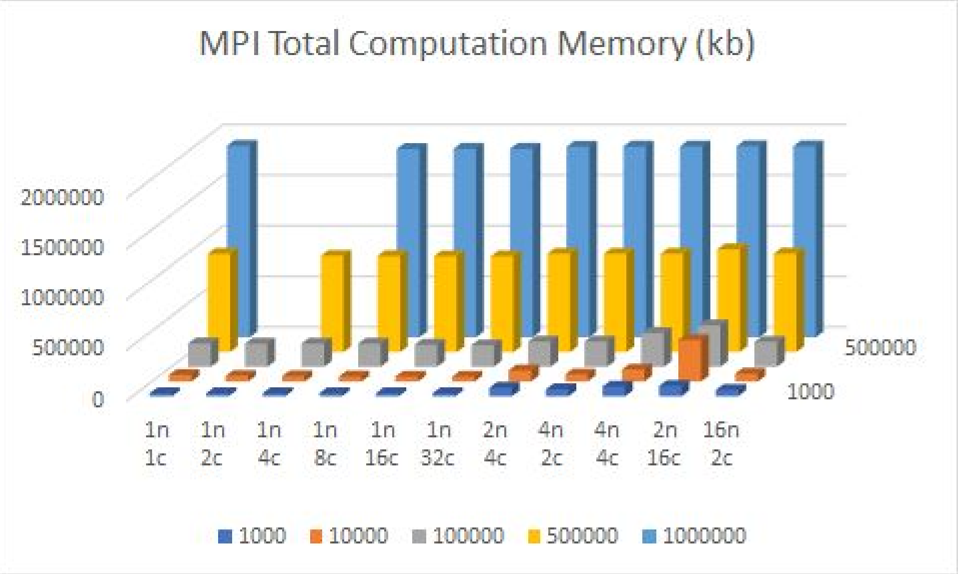
Pthreads were considerably more predictable considering the length of the line, but had a few odd outliers in the 1k lines tests. These outliers might be ignored as fs system glitches, but they could mean something else. Since pthreads is highly customizable and requires more input from the developer, background overhead is lower, thus a more even level across the board.

MPI was even more predictable and consistent across the various cores. The high case of 2n16c on 10k lines seems to be an outlier. The memory requirements didn't change nearly at all with 1 core compared to 32 cores. This is surprising, considering the higher cores require more copies of the data to be moved. I'm personally surprised that 1n1c isn't the smallest memory across the board.

Overall the memory scales linearly. That is 2x the lines means 2x the memory usage. This shows our implementation has a space usage on O(n) scale.







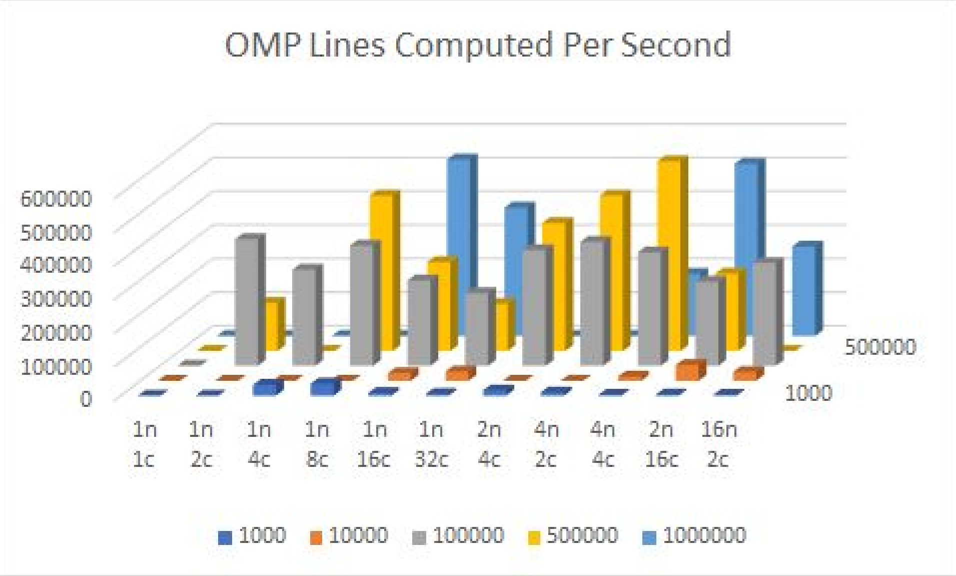
Computation Speed
Across the board, apart from MPI, computation times were significantly varied across core configurations and lines. One consistent trend is that prevalent in both is that computation over the lower rounds of 1k and 10k are much slower than other rounds. This make sense, considering more time is used to read in and communicate compared to the computation time.

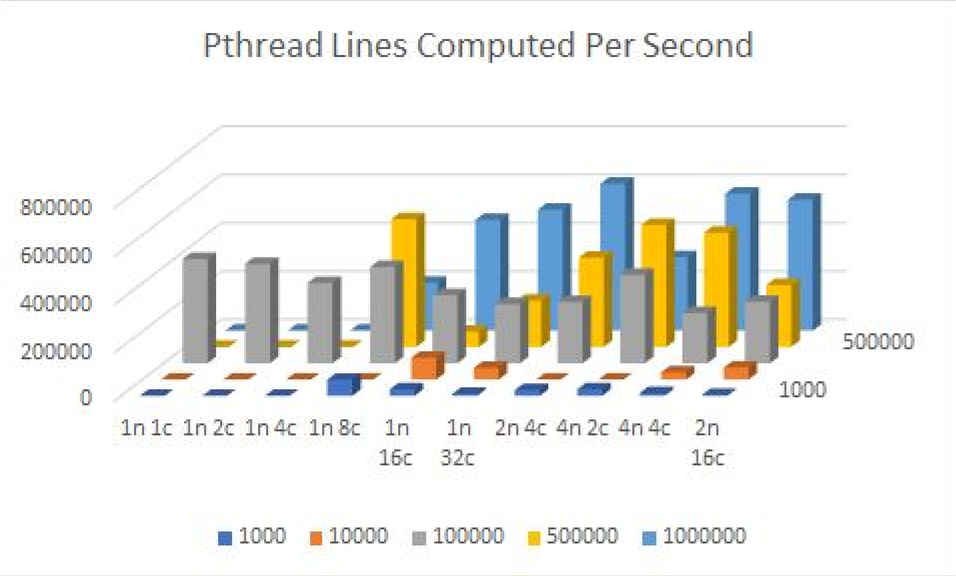
One reason the computations are so varied is the hidden overhead that is added when using OpenMP. Pthreads doesn't match with this theory, but there may still be some shared memory access observations going on in Pthreads and OpenMP which can reduce or increase computation time.

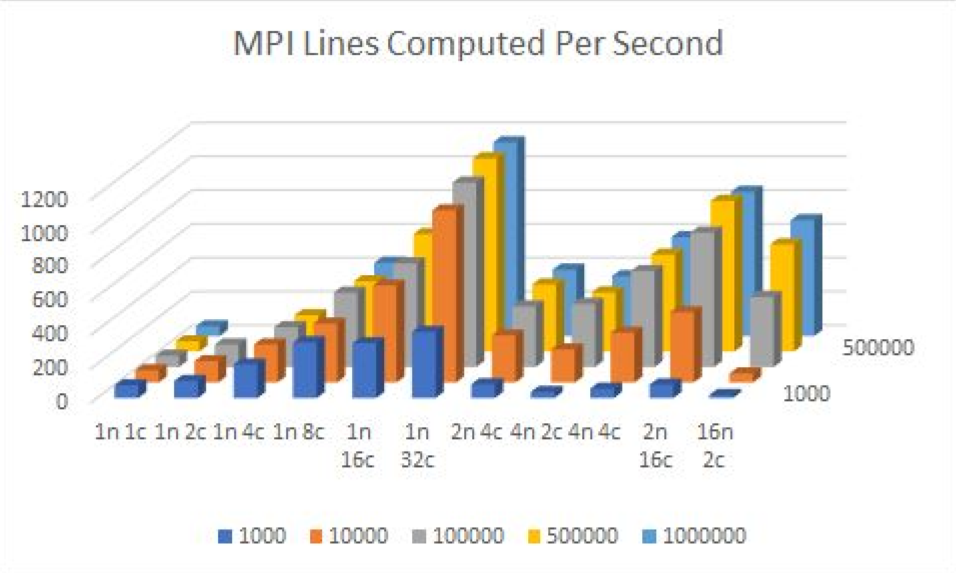
MPI is significantly slower than the other two, Pthread and OpenMP, but is considerably more predictable. With an increase in thread count, the increase in lines computed per second makes sense, as does the increase in performance as the number of lines increase.

The significant decrease in computation makes sense because there is a significant overhead of communicating between processes in MPI that simply does not exist at this level in Pthreads and OpenMP.

We also see that local communication within a node (2n4c vs 4n2c and 2n16c vs 16n2c) is much faster than communication between nodes.







CPU Efficiency

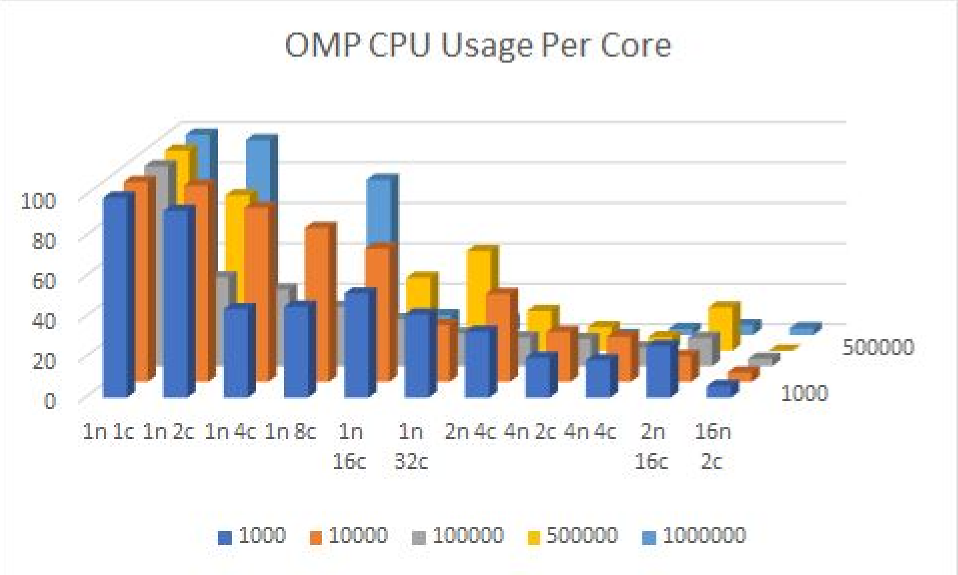
Over the board we see a typical trend of cpu usage per core going down as the number of cores we use increases. This is logical considering the more threads that we have, the more communication that is involved, which often results in waits on IO operations or other network communications.

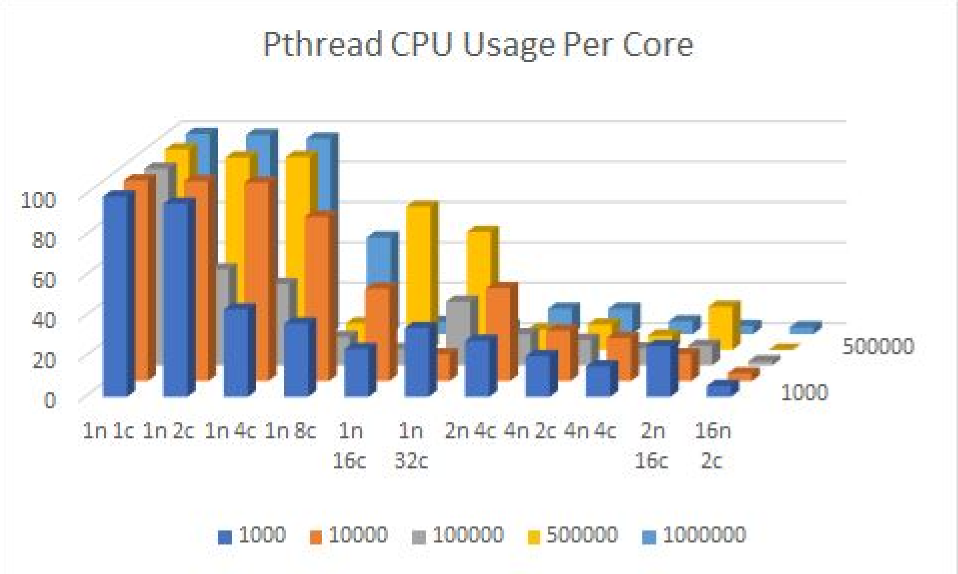
OpenMP and Pthreads are very similar in their graphs. As mentioned above, efficiency drops as our total thread count increases. As mentioned in the last section, communication within a node is faster than communication between nodes. This explains drops in efficiency between 2n4c and 4n2c, as well as 2n16c and 16n2c.

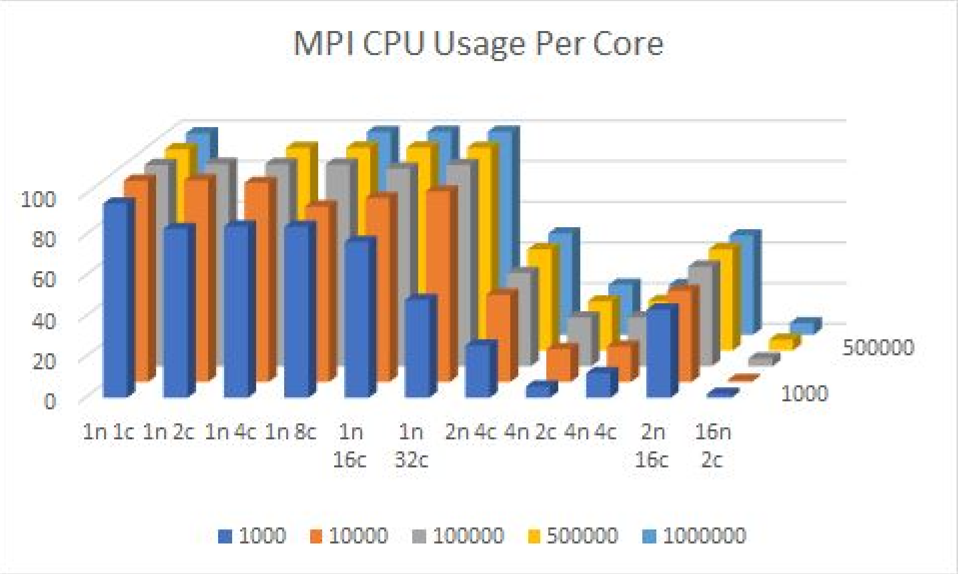
Also as mentioned before, OpenMP and Pthread both are varied in computation between different thread configurations. This is likely due to the minor but possibly frequent overhead checks on memory. More tests would also prove to solidify the variance in these graphs.

MPI seems to be considerably more efficient on core usage, especially when all information is stored in a single node (all 1n). Performance quickly drops when introducing multiple nodes, considering large amounts of communication take place between the main thread and all other threads.

Keep in mind that MPI proved less efficient per line than the other two, so these CPU usages might be better since the computations take longer in general.


OMP CPU Usage Per Core


Pthread CPU Usage Per Core


MPI CPU Usage Per Core

Conclusions and Future Work

The problem we have solved

From the information above, we've provided an interface for testing various multithreaded libraries, and compared and contrasted three popular methods for parallelizing code: OpenMP, Pthreads, MPI. Each has their own advantages and disadvantages, which depends on their

What to choose when

OpenMP, Pthreads, and MPI all work in different ways, with different levels of difficulty. There are many different ways of using them, but any one of them might not be the best for every case.

OpenMP is considerably fast to implement. Compared to the original single-threaded solution, we only had to add around 4 lines of code to implement OpenMP. Compared to Pthreads, OpenMP runs decently well, especially when you have minimal to no critical sections as in our tests.

Pthreads takes a little bit more time to implement. Some trickery might be necessary to pass multiple arguments to a thread calling function. Thread management, including creating and joining/waiting, is required to be implemented, but it comes at a benefit. Pthreads appears to run a bit faster than OpenMP, being the fastest implementation tested of the three.

MPI is a different beast entirely. MPI treats multiple threads as completely different processes running the same program, with communication channels between them. MPI is very case specific, but can be powerful if used correctly. In our case, MPI proved predictable and more stable, but significantly slower than the other two libraries. MPI is also very configurable, so there are many other options allowed for running MPI programs.

What next

One method that would be interesting to implement and test would be a hybrid scheme between MPI and either OpenMP or the more efficient Pthreads. Since MPI implements processes separately, one could combine that with multiple threads for each process with Pthreads or OpenMP. Test might show that MPI excels in inter-process communication, and Pthreads or OpenMP excels in intra-process communication. This may optimize various communication that must occur, and might make inter-node communication faster.

We could also implement one solution in Cuda using GPU's to process and communicate between threads. GPU's are designed for many-threaded applications, and communication and computation could be incredibly increased.

There are a number of possibilities for future work within the code creating in this project as well. Multiple runs as well as optimizing for many more cores and nodes working together could prove useful, and might lead to new insights concerning the differences among OpenMP, Pthreads, and MPI.

Sample Output:

```
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
101-102: </title><text>{{infobox
102-103: \n| image
103-104: \n| image
104-105: </title><text>{{refimprove|date=j
105-106: <page> <title>a_ch
106-107: <page> <title>a_ch
107-108: \n{{infobox album <!-- see wikipedia:wikiproject_albums -->\n| name         = a chorus of
108-109: \n\n==references==\n
109-110: united states]]\n| language        = [[english language|english]]\n|
110-111:  [[english language|english]]\n
111-112:  at the [[university of
112-113: </title><text>{{infobox
113-114: \n| title_orig     =\n| translator     =\n| image          =
114-115:        = english\n| series
115-116: }}\n\n\'\'\'\'\'a cr
116-117:  <!-- see wikipedia:wikiproject_albums -->\n
117-118: <page> <title>a_da
118-119: <page> <title>a_d
119-120: <page> <title>a_de
120-121: .<ref>{{cite web|url=http://www.
121-122: .<ref>{{cite web|url=http://www.
122-123: <page> <title>a_di
123-124: <page> <title>a_d
124-125: <page> <title>a_dr
125-126: <page> <title>a_dr
126-127: </title><text>{{infobox
127-128: <page> <title>a_
128-129: .\n\n==plot==\nthe film
129-130: {{infobox film\n| name            = a f
130-131: {{infobox film\n| name
131-132: <page> <title>a_foreign_
132-133: <page> <title>a_f
133-134: <page> <title>a_
134-135:  <!-- see wikipedia:wikiproject_
135-136: <page> <title>a_g
136-137: <page> <title>a_g
137-138: <page> <title>a_
138-139: </title><text>{{infobox
139-140: </title><text>{{infobox
140-141: <page> <title>a_h
141-142: <page> <title>a_h
142-143: <page> <title>a_
143-144: wikipedia:wikiproject
144-145: <page> <title>a_k
145-146: <page> <title>a_
146-147: </title><text>{{infobox
147-148: e</title><text>{{infobox s
148-149: <page> <title>a_little_
149-150: <page> <title>a_l
150-151: </title><text>{{infobox
151-152: \n| name
152-153: \n}}\n\'\'\'\'\'a mi
153-154: </title><text>{{infobox
154-155: </title><text>{{infobox album <!-- see wikipedia:wikiproject_albums -->\n| name
155-156: <page> <title>a_new_
156-157: </title><text>\'\'\'\'\'a
157-158: </title><text>\'\'\'
158-159: <page> <title>a_p
159-160:  <!-- see wikipedia:wikiproject_albums -->\n
160-161: </title><text>{{infobox
161-162: </title><text>{{infobox
162-163: -stub}}</text> </page>

163-164: <page> <title>a_p
164-165: \n}}\n\'\'\'\'\'a
165-166: <page> <title>a_ra
166-167:  = [[english language|english]]\n
167-168: <page> <title>a_r
168-169: </title><text>\'\'\'
169-170: \n\n==references==\n{{reflist}}\n\n
170-171: <page> <title>a_ro
171-172: <page> <title>a_
172-173: </title><text>{{infobox
173-174: </title><text>{{infobox book | <!-- see wikipedia:wikiproject_novels or wikipedia:wikiproject_books -->\n|
name
174-175:  <!-- see wikipedia:wikiproject_
175-176:  <!-- see wikipedia:wikiproject_
176-177: <page> <title>a_st
177-178: <page> <title>a_st
```

Appendix A - OpenMP Code:

```
#define _GNU_SOURCE
#include <omp.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>

char* findLongestSubstring(char* a, char* b);
void threadRun(int threadNumber, int numberOfThreads, unsigned long numberOfLines, char** fileData,
char** results);

// Simple linked list structure holding i and j index components.
struct ListItem {
  struct ListItem* nextItem;
  unsigned long i;
  unsigned long j;
};

int main(int argc, char *argv[]) {
  unsigned long numberOfLinesToProcess = 1000;
  int numberOfThreads = 1;
  char verbosity = 1;

  for(int i = 1; i < argc; i++) {
    if(strncmp(argv[i], "--lines=", 8) == 0) {
      numberOfLinesToProcess = strtoul(argv[i] + 8, NULL, 10);
    } else if (strcmp(argv[i], "-l") == 0 && i != (argc - 1)) {
      numberOfLinesToProcess = strtoul(argv[i + 1], NULL, 10);
      i++;
    } else if (strcmp(argv[i], "-q") == 0) {
      verbosity = 0;
    } else if (strcmp(argv[i], "-v") == 0) {
      verbosity = 2;
    } else if (strncmp(argv[i], "--threads=", 10) == 0) {
      numberOfThreads = (int)strtol(argv[i] + 10, NULL, 10);
      if(numberOfThreads < 1) {
        printf("Invalid Argument: number of threads must be at least 1.\n");
        return -1;
      }
    } else if (strcmp(argv[i], "-t") == 0 && i != (argc - 1)) {
      numberOfThreads = (int)strtol(argv[i + 1], NULL, 10);
      if(numberOfThreads < 1) {
        printf("Invalid Argument: number of threads must be at least 1.\n");
        return -1;
      }
      i++;
    } else if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
      printf("Arguments:\n"
          "--lines=# (-l #)     Set the number of lines to process from the input file.\n"
          "--threads=# (-t #)   Set the number of threads to utilize for processing.\n"
          "-v                   Enable verbose output.\n"
          "-q                   Silence all output.\n"
          "--help (-h)          Display this help.\n");
      return 0;
    } else {
      printf("Invalid Argument: unknown option: %s\n", argv[i]);
      return -1;
    }
  }

 // Prepare and open the file.
  char* filePath = "/homes/dan/625/wiki_dump.txt";
  if(verbosity == 2) {
    printf("Running with %d threads on %lu lines.\n", numberOfThreads, numberOfLinesToProcess);
```

```
      printf("File path: %s\n", filePath);
   }

   FILE *fp = fopen(filePath, "r");
   if(fp == NULL) {
      printf("Error! Unable to open file!");
      return -1;
   }

   // Read in the file line by line. Put data in fileData.
   unsigned long lineNumber = 0;
   long read;
   char* line = NULL;
   unsigned long lineLength;
   char** fileData = NULL;
   while (lineNumber < numberOfLinesToProcess && (read = getline(&line, &lineLength, fp)) != -1) {
      fileData = realloc(fileData, sizeof(char*) * (lineNumber + 1));
      if(fileData == NULL) {
         printf("Error! Unable to allocate memory for fileData: size %lu\n", (lineNumber + 1));
         return -1;
      }
      fileData[lineNumber++] = line;
      line = NULL;    // Dereference line to keep fileData unchanged while reading next line.
   }
   if(fileData == NULL) {
      printf("Error! Unable to read from file!\n");
      return -1;
   }
   fclose(fp);

   // Prepare the results.
   char** results = malloc(sizeof(char*) * (lineNumber - 1));     // With 100 lines, we have 99 comparisons.
   if(results == NULL) {
      printf("Error! Unable to allocate memory for results: size %lu\n", (lineNumber - 1));
      return -1;
   }

   // Compare all of the substrings. Begin thread section.
   omp_set_num_threads(numberOfThreads);
#pragma omp parallel
   {
      threadRun(omp_get_thread_num(), numberOfThreads, lineNumber, fileData, results);
   }

   if(verbosity != 0) {
      // End thread section. Print the results.
      for(unsigned long i = 0; i < (lineNumber - 1); i++) {
//       printf("%lu-%lu: '%s'\n", i, (i + 1), results[i]);
         printf("%lu-%lu: %s\n", i, (i + 1), results[i]);
      }
   }

   // Free all memory.
   for(unsigned long i = 0; i < (lineNumber - 1); i++) {
      free(results[i]);
      free(fileData[i]);
   }
   free(fileData[lineNumber]);
   free(results);
   free(fileData);
}

void threadRun(int threadNumber, int numberOfThreads, unsigned long numberOfLines, char** fileData,
char** results) {
   // Determine our quota.
   unsigned long quota = numberOfLines / numberOfThreads;
```

```
    if(quota * numberOfThreads < numberOfLines) {
        quota++;
    }

    // Determine our first and last lines.
    unsigned long firstLine = quota * threadNumber;          // First line we need to compare - inclusive.
    unsigned long lastLine = quota * (threadNumber + 1);     // Last line we need to compare - inclusive.
    if(threadNumber == (numberOfThreads - 1)) {
        lastLine = numberOfLines - 1;                   // If we are the last thread, ensure we get all of the lines.
    }

    // Correct quota if necessary.
    quota = lastLine - firstLine;
    if(quota < 0) {
        quota = 0;
    }

    // Complete quota to local results.
    char** localResults = malloc(sizeof(char*) * quota);
    if(localResults == NULL) {
        printf("Error! Unable to allocate memory for localResults: size %lu\n", quota * sizeof(char*));
        exit(-1);
    }
    for(int i = 0; i < quota; i++) {
        localResults[i] = findLongestSubstring(fileData[i + firstLine], fileData[i + firstLine + 1]);
    }

    // Copy local results to final results.
    for(int i = 0; i < quota; i++) {
        results[i + firstLine] = localResults[i];
    }
    free(localResults);
}

// Returns the longest common string between a and b. Be sure to free the returned char* when done.
char* findLongestSubstring(char* a, char* b) {
    // Get lengths of the strings.
    unsigned long aLength = strlen(a);
    unsigned long bLength = strlen(b);

    // Prepare linked list.
    struct ListItem *prefixList = malloc(sizeof(struct ListItem));
    if (prefixList == NULL) {
        printf("Error! Unable to allocate memory for prefixList: size %lu\n", sizeof(struct ListItem));
        exit(-1);
    }
    struct ListItem *recentList = prefixList;

    // Prepare array of (a+1) x (b+1) size.
    // The first row and column are 0's, every row after that is comparing a index to b index.
    char *set = calloc((aLength + 1) * (bLength + 1), sizeof(char));
    if (set == NULL) {
        printf("Error! Unable to allocate memory for localResults: size %lu\n",
                sizeof(char) * (aLength + 1) * (bLength + 1));
        exit(-1);
    }

    unsigned numOfElements = 0;

    // Go through each character. Compare a[i] to b[j] and mark it's corresponding location in set to 1 if
true.
    for (unsigned long i = 0; i < aLength; i++) {
        for (unsigned long j = 0; j < bLength; j++) {
            // Mark 0 row.
            if (i == 0 || j == 0) {
                set[((i) * bLength) + j] = 0;
            }
            // Compare a at i and b at j. Note the index access in set is different since buffer at 0 row/col.
            if (a[i] == b[j]) {
                unsigned long index = ((i + 1) * bLength) + j + 1;

                // Create new item in the linked-list
                recentList->nextItem = malloc(sizeof(struct ListItem));
                if (recentList->nextItem == NULL) {
                    printf("Error! Unable to allocate memory for localResults: size %lu\n", sizeof(struct ListItem));
                    exit(-1);
                }
                recentList = recentList->nextItem;
                recentList->i = i;
                recentList->j = j;
                numOfElements++;

                // Set the value of the array.
                set[index] = 1;
            }
        }
    }

    // Prepare for searching set.
    unsigned long longestIndex = 0;
    unsigned long longestValue = 0;
    struct ListItem *currentList = prefixList->nextItem;
    free(prefixList);
    // For each item in the linked list (a 1 initially), start a check diagonally for the longest common
substring.
    for (unsigned long k = 0; k < numOfElements; k++) {
        // Check diagonal.
        unsigned long i = currentList->i;
        unsigned long j = currentList->j;
        unsigned long it = i;
        unsigned long jt = j;
        while (set[((it + 1) * bLength) + jt + 1] == 1) {
            set[((it + 1) * bLength) + jt + 1] = 0;
            // Optimization: set a checked value to 0 to prevent sub-checks of sub-substrings.
            // EG - If we have "always true" as the longest string, without this optimization, we will end up
            // checking "lways true" as the longest string next row. Set to 0 to prevent sub-checks.
            it++;
            jt++;
        }
        // Compare with longest value found so far.
        if (it - i > longestValue) {
            longestValue = it - i;
            longestIndex = i;
        }
        // Go to the next item.
        struct ListItem *nextList = currentList->nextItem;
        free(currentList);
        currentList = nextList;
    }
    free(set);

    // Create the longest string found.
    char *longestString = malloc(sizeof(char) * (longestValue + 1));
    if (longestString == NULL) {
        printf("Error! Unable to allocate memory for longestString: size %lu\n", sizeof(longestValue + 1));
        exit(-1);
    }
    for (unsigned long i = 0; i < longestValue; i++) {
        longestString[i] = a[i + longestIndex];
    }
    longestString[longestValue] = '\0';
    return longestString;
```

```
}
```

Appendix B - Pthreads Code:

```
#define _GNU_SOURCE
#include <pthread.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>

char* findLongestSubstring(char* a, char* b);
void* threadRun(void* arg);

// Simple linked list structure holding i and j index components.
struct ListItem {
  struct ListItem* nextItem;
  unsigned long i;
  unsigned long j;
};

int main(int argc, char *argv[]) {
  unsigned long numberOfLinesToProcess = 1000;
  int numberOfThreads = 1;
  char verbosity = 1;

  for(int i = 1; i < argc; i++) {
    if(strncmp(argv[i], "--lines=", 8) == 0) {
      numberOfLinesToProcess = strtoul(argv[i] + 8, NULL, 10);
    } else if (strcmp(argv[i], "-l") == 0 && i != (argc - 1)) {
      numberOfLinesToProcess = strtoul(argv[i + 1], NULL, 10);
      i++;
    } else if (strcmp(argv[i], "-q") == 0) {
      verbosity = 0;
    } else if (strcmp(argv[i], "-v") == 0) {
      verbosity = 2;
    } else if (strncmp(argv[i], "--threads=", 10) == 0) {
      numberOfThreads = (int)strtol(argv[i] + 10, NULL, 10);
      if(numberOfThreads < 1) {
        printf("Invalid Argument: number of threads must be at least 1.\n");
        return -1;
      }
    } else if (strcmp(argv[i], "-t") == 0 && i != (argc - 1)) {
      numberOfThreads = (int)strtol(argv[i + 1], NULL, 10);
      if(numberOfThreads < 1) {
        printf("Invalid Argument: number of threads must be at least 1.\n");
        return -1;
      }
      i++;
    } else if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
      printf("Arguments:\n"
          "--lines=# (-l #)     Set the number of lines to process from the input file.\n"
          "--threads=# (-t #)   Set the number of threads to utilize for processing.\n"
          "-v                 Enable verbose output.\n"
          "-q                 Silence all output.\n"
          "--help (-h)          Display this help.\n");
      return 0;
    } else {
      printf("Invalid Argument: unknown option: %s\n", argv[i]);
      return -1;
    }
  }
  // Prepare and open the file.
  char* filePath = "/homes/dan/625/wiki_dump.txt";

  if(verbosity == 2) {
    printf("Running with %d threads on %lu lines.\n", numberOfThreads, numberOfLinesToProcess);
    printf("File path: %s\n", filePath);
    }

    FILE *fp = fopen(filePath, "r");
    if(fp == NULL) {
        printf("Error! Unable to open file!");
        return -1;
    }

    // Read in the file line by line. Put data in fileData.
    unsigned long lineNumber = 0;
    long read;
    char* line = NULL;
    unsigned long lineLength;
    char** fileData = NULL;
    while (lineNumber < numberOfLinesToProcess && (read = getline(&line, &lineLength, fp)) != -1) {
        fileData = realloc(fileData, sizeof(char*) * (lineNumber + 1));
        if(fileData == NULL) {
            printf("Error! Unable to allocate memory for fileData: size %lu\n", (lineNumber + 1));
            return -1;
        }
        fileData[lineNumber++] = line;
        line = NULL;    // Dereference line to keep fileData unchanged while reading next line.
    }
    if(fileData == NULL) {
        printf("Error! Unable to read from file!\n");
        return -1;
    }
    fclose(fp);

    // Prepare the results.
    char** results = malloc(sizeof(char*) * (lineNumber - 1));     // With 100 lines, we have 99 comparisons.
    if(results == NULL) {
        printf("Error! Unable to allocate memory for results: size %lu\n", (lineNumber - 1));
        return -1;
    }

    // Compare all of the substrings. Begin thread section.
    pthread_t threads[numberOfThreads];
    pthread_attr_t attr;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    //int threadNumber, int numberOfThreads, unsigned long numberOfLines, char** fileData, char**
   results
    void *arg = malloc(sizeof(int) + sizeof(int) + sizeof(unsigned long) + sizeof(char**) + sizeof(char**));
    unsigned offset = sizeof(int);
    memcpy(arg + offset, &numberOfThreads, sizeof(int));
    offset += sizeof(int);
    memcpy(arg + offset, &lineNumber, sizeof(unsigned long));
    offset += sizeof(unsigned long);
    memcpy(arg + offset, &fileData, sizeof(char**));
    offset += sizeof(char**);
    memcpy(arg + offset, &results, sizeof(char**));
    offset += sizeof(char**);

    void *threadArgs[numberOfThreads];
    for (int rc, i = 0; i < numberOfThreads; i++) {
        threadArgs[i] = malloc(offset);
        memcpy(threadArgs[i], arg, offset);
        memcpy(threadArgs[i], &i, sizeof(int));
        rc = pthread_create(&threads[i], &attr, threadRun, threadArgs[i]);
        if (rc) {
            printf("Error! Return code from pthread_create() is %d\n", rc);
            return -1;
```

```
      }
   }

   pthread_attr_destroy(&attr);
   void *status;
   for(int rc, i = 0; i < numberOfThreads; i++) {
      rc = pthread_join(threads[i], &status);
      free(threadArgs[i]);
      if (rc) {
         printf("Error! Return code from pthread_join() is %d\n", rc);
         return -1;
      }
   }

   if(verbosity != 0) {
      // End thread section. Print the results.
      for(unsigned long i = 0; i < (lineNumber - 1); i++) {
         printf("%lu-%lu: %s\n", i, (i + 1), results[i]);
      }
   }

   // Free all memory.
   for(unsigned long i = 0; i < (lineNumber - 1); i++) {
      free(results[i]);
      free(fileData[i]);
   }
   free(fileData[lineNumber]);
   free(results);
   free(fileData);

   pthread_exit(NULL);
}

//int threadNumber, int numberOfThreads, unsigned long numberOfLines, char** fileData, char** results
void* threadRun(void *arg) {
   int threadNumber;
   int numberOfThreads;
   unsigned long numberOfLines;
   char** fileData;
   char** results;

   unsigned offset = 0;
   memcpy(&threadNumber, arg + offset, sizeof(int));
   offset += sizeof(int);
   memcpy(&numberOfThreads, arg + offset, sizeof(int));
   offset += sizeof(int);
   memcpy(&numberOfLines, arg + offset, sizeof(unsigned long));
   offset += sizeof(unsigned long);
   memcpy(&fileData, arg + offset, sizeof(char**));
   offset += sizeof(char**);
   memcpy(&results, arg + offset, sizeof(char**));

   // Determine our quota.
   unsigned long quota = numberOfLines / numberOfThreads;
   if(quota * numberOfThreads < numberOfLines) {
      quota++;
   }

   // Determine our first and last lines.
   unsigned long firstLine = quota * threadNumber;           // First line we need to compare - inclusive.
   unsigned long lastLine = quota * (threadNumber + 1);      // Last line we need to compare - inclusive.
   if(threadNumber == (numberOfThreads - 1)) {
      lastLine = numberOfLines - 1;                   // If we are the last thread, ensure we get all of the lines.
   }

   // Correct quota if necessary.
```

```
    quota = lastLine - firstLine;
    if(quota < 0) {
        quota = 0;
    }

    // Complete quota to local results.
    char** localResults = malloc(sizeof(char*) * quota);
    if(localResults == NULL) {
        printf("Error! Unable to allocate memory for localResults: size %lu\n", quota * sizeof(char*));
        exit(-1);
    }
    for(int i = 0; i < quota; i++) {
        localResults[i] = findLongestSubstring(fileData[i + firstLine], fileData[i + firstLine + 1]);
    }

    // Copy local results to final results.
    for(int i = 0; i < quota; i++) {
        results[i + firstLine] = localResults[i];
    }
    free(localResults);
    pthread_exit(NULL);
}

// Returns the longest common string between a and b. Be sure to free the returned char* when done.
char* findLongestSubstring(char* a, char* b) {
    // Get lengths of the strings.
    unsigned long aLength = strlen(a);
    unsigned long bLength = strlen(b);

    // Prepare linked list.
    struct ListItem *prefixList = malloc(sizeof(struct ListItem));
    if (prefixList == NULL) {
        printf("Error! Unable to allocate memory for prefixList: size %lu\n", sizeof(struct ListItem));
        exit(-1);
    }
    struct ListItem *recentList = prefixList;

    // Prepare array of (a+1) x (b+1) size.
    // The first row and column are 0's, every row after that is comparing a index to b index.
    char *set = calloc((aLength + 1) * (bLength + 1), sizeof(char));
    if (set == NULL) {
        printf("Error! Unable to allocate memory for localResults: size %lu\n",
                sizeof(char) * (aLength + 1) * (bLength + 1));
        exit(-1);
    }

    unsigned numOfElements = 0;

    // Go through each character. Compare a[i] to b[j] and mark it's corresponding location in set to 1 if
true.
    for (unsigned long i = 0; i < aLength; i++) {
        for (unsigned long j = 0; j < bLength; j++) {
            // Mark 0 row.
            if (i == 0 || j == 0) {
                set[((i) * bLength) + j] = 0;
            }
            // Compare a at i and b at j. Note the index access in set is different since buffer at 0 row/col.
            if (a[i] == b[j]) {
                unsigned long index = ((i + 1) * bLength) + j + 1;

                // Create new item in the linked-list
                recentList->nextItem = malloc(sizeof(struct ListItem));
                if (recentList->nextItem == NULL) {
                    printf("Error! Unable to allocate memory for localResults: size %lu\n", sizeof(struct ListItem));
                    exit(-1);
                }
                recentList = recentList->nextItem;
                recentList->i = i;
                recentList->j = j;
                numOfElements++;

                // Set the value of the array.
                set[index] = 1;
            }
        }
    }

    // Prepare for searching set.
    unsigned long longestIndex = 0;
    unsigned long longestValue = 0;
    struct ListItem *currentList = prefixList->nextItem;
    free(prefixList);

    // For each item in the linked list (a 1 initially), start a check diagonally for the longest common
substring.
    for (unsigned long k = 0; k < numOfElements; k++) {
        // Check diagonal.
        unsigned long i = currentList->i;
        unsigned long j = currentList->j;
        unsigned long it = i;
        unsigned long jt = j;
        while (set[((it + 1) * bLength) + jt + 1] == 1) {
            set[((it + 1) * bLength) + jt + 1] = 0;
            // Optimization: set a checked value to 0 to prevent sub-checks of sub-substrings.
            // EG - If we have "always true" as the longest string, without this optimization, we will end up
            // checking "lways true" as the longest string next row. Set to 0 to prevent sub-checks.
            it++;
            jt++;
        }

        // Compare with longest value found so far.
        if (it - i > longestValue) {
            longestValue = it - i;
            longestIndex = i;
        }

        // Go to the next item.
        struct ListItem *nextList = currentList->nextItem;
        free(currentList);
        currentList = nextList;
    }
    free(set);

    // Create the longest string found.
    char *longestString = malloc(sizeof(char) * (longestValue + 1));
    if (longestString == NULL) {
        printf("Error! Unable to allocate memory for longestString: size %lu\n", sizeof(longestValue + 1));
        exit(-1);
    }
    for (unsigned long i = 0; i < longestValue; i++) {
        longestString[i] = a[i + longestIndex];
    }
    longestString[longestValue] = '\0';
    return longestString;
}
```

Appendix C - MPI Code:

```
#define _GNU_SOURCE
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include "mpi.h"

char* findLongestSubstring(char* a, char* b);
void threadRun(int threadNumber, int numberOfThreads, unsigned long numberOfLines, char** fileData,
char** results);
void getQuota(unsigned long numOfLines, int numOfThreads, int threadID, unsigned long *firstLine,
unsigned long *quota);

// Simple linked list structure holding i and j index components.
struct ListItem {
  struct ListItem* nextItem;
  unsigned long i;
  unsigned long j;
};

int main(int argc, char *argv[]) {
  int threadId;
  int numOfThreads;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &threadId);
  MPI_Comm_size(MPI_COMM_WORLD, &numOfThreads);

  unsigned long numberOfLinesToProcess = 1000;
  char verbosity = 1;
  unsigned long lineNumber = 0;
  char** fileData = NULL;
  char** results;
  unsigned long *line_sizes;

  if(threadId == 0) {
    for(int i = 1; i < argc; i++) {
      if(strncmp(argv[i], "--lines=", 8) == 0) {
        numberOfLinesToProcess = strtoul(argv[i] + 8, NULL, 10);
      } else if (strcmp(argv[i], "-l") == 0 && i != (argc - 1)) {
        numberOfLinesToProcess = strtoul(argv[i + 1], NULL, 10);
        i++;
      } else if (strcmp(argv[i], "-q") == 0) {
        verbosity = 0;
      } else if (strcmp(argv[i], "-v") == 0) {
        verbosity = 2;
      } else if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
        printf("Arguments:\n"
            "--lines=# (-l #)     Set the number of lines to process from the input file.\n"
            "--threads=# (-t #)   Set the number of threads to utilize for processing.\n"
            "-v                 Enable verbose output.\n"
            "-q                 Silence all output.\n"
            "--help (-h)        Display this help.\n");
        return 0;
      } else {
        printf("Invalid Argument: unknown option: %s\n", argv[i]);
        MPI_Abort(MPI_COMM_WORLD, -1);
      }
    }
    // Prepare and open the file.
    char* filePath = "/homes/dan/625/wiki_dump.txt";

    if(verbosity == 2) {
      printf("Running with %d threads on %lu lines.\n", numOfThreads, numberOfLinesToProcess);
      printf("File path: %s\n", filePath);
    }

    FILE *fp = fopen(filePath, "r");
    if(fp == NULL) {
      printf("Error! Unable to open file!");
      MPI_Abort(MPI_COMM_WORLD, -1);
    }

    // Read in the file line by line. Put data in fileData.
    long read;
    char* line = NULL;
    unsigned long lineLength;
    while (lineNumber < numberOfLinesToProcess && (read = getline(&line, &lineLength, fp)) != -1) {
      fileData = realloc(fileData, sizeof(char*) * (lineNumber + 1));
      if(fileData == NULL) {
        printf("Error! Unable to allocate memory for fileData: size %lu\n", (lineNumber + 1));
        MPI_Abort(MPI_COMM_WORLD, -1);
      }
      fileData[lineNumber++] = line;
      line = NULL;    // Dereference line to keep fileData unchanged while reading next line.
    }
    if(fileData == NULL) {
      printf("Error! Unable to read from file!\n");
      MPI_Abort(MPI_COMM_WORLD, -1);
    }
    fclose(fp);

    line_sizes = malloc(sizeof(unsigned long) * lineNumber);
    for (int i = 0; i < lineNumber; i++) {
      line_sizes[i] = strlen(fileData[i]) + 1;
    }

    // Prepare the results.
    results = malloc(sizeof(char*) * (lineNumber - 1));    // With 100 lines, we have 99 comparisons.
    if(results == NULL) {
      printf("Error! Unable to allocate memory for results: size %lu\n", (lineNumber - 1));
      return -1;
    }
  }
  // Broadcast number of lines
  MPI_Bcast(&lineNumber, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
  unsigned long myQuota;
  unsigned long myFirstLine;
  getQuota(lineNumber, numOfThreads, threadId, &myFirstLine, &myQuota);
  // Thread 0 start dispatching work.
  if(threadId == 0) {
    /// DISPATCH DATA TO THREADS
    for (int i = 1; i < numOfThreads; i++) {
      unsigned long otherQuota;
      unsigned long otherFirstLine;
      getQuota(lineNumber, numOfThreads, i, &otherFirstLine, &otherQuota);
      // Using j <= localQuota because thread x compares line n to n+1, and thread (x+1) compares line
n+1 to n+2...
      for (unsigned long j = 0; j <= otherQuota; j++) {
        // Send line (j + firstLine) to thread i
        MPI_Send(&line_sizes[j + otherFirstLine], 1, MPI_UNSIGNED_LONG, i, 0, MPI_COMM_WORLD);
        for (unsigned long k = 0; k < line_sizes[j + otherFirstLine]; k++) {
          MPI_Send(&(fileData[j + otherFirstLine][k]), 1, MPI_UNSIGNED_CHAR, i, j * 10000 + k,
MPI_COMM_WORLD);
        }
      }
    }
  } else {
    /// RECEIVE DISPATCHED DATA
    fileData = malloc(sizeof(char*) * (myQuota + 1));
    results = malloc(sizeof(char*) * myQuota);
    for(unsigned long j = 0; j <= myQuota; j++) {
      unsigned long lineLength = 0;
      MPI_Recv(&lineLength, 1, MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      fileData[j] = malloc(sizeof(char) * lineLength);
      for(unsigned long k = 0; k < lineLength; k++) {
        MPI_Recv(&(fileData[j][k]), 1, MPI_UNSIGNED_CHAR, 0, j*10000 + k, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      }
    }
  }

  int start = 0;
  if(threadId == 0) {
    start = 1;
  }
  MPI_Bcast(&start, 1, MPI_INT, 0, MPI_COMM_WORLD);

  /// RUN OPERATION
  threadRun(threadId, numOfThreads, myQuota, fileData, results);

  int finish = 0;
  if(threadId == 0) {
    finish = 1;
  }
  MPI_Bcast(&finish, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if(threadId == 0) {
    /// COLLECT DATA FROM THREADS
    for(int i = 1; i < numOfThreads; i++) {
      unsigned long otherQuota;
      unsigned long otherFirstLine;
      getQuota(lineNumber, numOfThreads, i, &otherFirstLine, &otherQuota);

      int ready = 1;
      MPI_Send(&ready, 1, MPI_INT, i, 9999999999 + i, MPI_COMM_WORLD);
      for(unsigned long j = 0; j < otherQuota; j++) {
        // Put line (j + firstLine) from thread i into results
        unsigned long lineLength = 0;
        MPI_Recv(&lineLength, 1, MPI_UNSIGNED_LONG, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        results[j + otherFirstLine] = malloc(sizeof(char) * lineLength);
        for(unsigned long k = 0; k < lineLength; k++) {
          MPI_Recv(&(results[j + otherFirstLine][k]), 1, MPI_UNSIGNED_CHAR, i, j*10000 + k,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
      }
    }
    if(verbosity != 0) {
      for(unsigned long i = 0; i < (lineNumber - 1); i++) {
        printf("%lu-%lu: %s\n", i, (i + 1), results[i]);
      }
    }
  } else {
    /// SEND DATA TO BE COLLECTED
    line_sizes = malloc(sizeof(unsigned long) * myQuota);
    int ready = 0;
    MPI_Recv(&ready, 1, MPI_INT, 0, 9999999999 + threadId, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    for(unsigned long j = 0; j < myQuota; j++) {
      line_sizes[j] = strlen(results[j]) + 1;
      MPI_Send(&line_sizes[j], 1, MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WORLD);
      for(unsigned long k = 0; k < line_sizes[j]; k++) {
        MPI_Send(&(results[j][k]), 1, MPI_UNSIGNED_CHAR, 0, j*10000 + k, MPI_COMM_WORLD);
```

```
         }
      }
   }

   int clearMem = 0;
   if(threadId == 0) {
      clearMem = 1;
   }
   MPI_Bcast(&clearMem, 1, MPI_INT, 0, MPI_COMM_WORLD);

   if(threadId == 0) {
      for(unsigned long i = 0; i < (lineNumber - 1); i++) {
         free(results[i]);
         free(fileData[i]);
      }
      free(fileData[lineNumber]);
   } else {
      for(unsigned long i = 0; i < myQuota; i++) {
         free(results[i]);
         free(fileData[i]);
      }
      free(fileData[myQuota]);
   }
   // Free all memory.
   free(results);
   free(fileData);
   free(line_sizes);
   MPI_Finalize();
}

void getQuota(unsigned long numOfLines, int numOfThreads, int threadID, unsigned long *firstLine,
unsigned long *quota) {
   // Determine global quota.
   unsigned long gQuota = numOfLines / numOfThreads;   //10 = 30 / 3
   if(gQuota * numOfThreads < numOfLines) {    // 10 * 3 < 30
      gQuota++;
   }
   (*firstLine) = gQuota * threadID;   // 10 = 10 * 1, 20 = 20 * 1
   unsigned long lastLine = gQuota * (threadID + 1);
   if((threadID) == (numOfThreads) - 1) {
      lastLine = numOfLines - 1;
   }

   // Correct quota if necessary.
   (*quota) = lastLine - (*firstLine);
   if((*quota) < 0) {
      (*quota) = 0;
   }
}

void threadRun(int threadNumber, int numberOfThreads, unsigned long numberOfLines, char** fileData,
char** results) {
   // Complete quota to local results.
   for(int i = 0; i < numberOfLines; i++) {
      results[i] = findLongestSubstring(fileData[i], fileData[i + 1]);
   }
}

// Returns the longest common string between a and b. Be sure to free the returned char* when done.
char* findLongestSubstring(char* a, char* b) {
   // Get lengths of the strings.
   unsigned long aLength = strlen(a);
   unsigned long bLength = strlen(b);

   // Prepare linked list.
   struct ListItem *prefixList = malloc(sizeof(struct ListItem));
```

```
  if (prefixList == NULL) {
     printf("Error! Unable to allocate memory for prefixList: size %lu\n", sizeof(struct ListItem));
     MPI_Abort(MPI_COMM_WORLD, -1);
  }
  struct ListItem *recentList = prefixList;

  // Prepare array of (a+1) x (b+1) size.
  // The first row and column are 0's, every row after that is comparing a index to b index.
  char *set = calloc((aLength + 1) * (bLength + 1), sizeof(char));
  if (set == NULL) {
     printf("Error! Unable to allocate memory for localResults: size %lu\n",
          sizeof(char) * (aLength + 1) * (bLength + 1));
     MPI_Abort(MPI_COMM_WORLD, -1);
  }

  unsigned numOfElements = 0;

  // Go through each character. Compare a[i] to b[j] and mark it's corresponding location in set to 1 if
true.
  for (unsigned long i = 0; i < aLength; i++) {
     for (unsigned long j = 0; j < bLength; j++) {
        // Mark 0 row.
        if (i == 0 || j == 0) {
           set[((i) * bLength) + j] = 0;
        }
        // Compare a at i and b at j. Note the index access in set is different since buffer at 0 row/col.
        if (a[i] == b[j]) {
           unsigned long index = ((i + 1) * bLength) + j + 1;

           // Create new item in the linked-list
           recentList->nextItem = malloc(sizeof(struct ListItem));
           if (recentList->nextItem == NULL) {
              printf("Error! Unable to allocate memory for localResults: size %lu\n", sizeof(struct ListItem));
              MPI_Abort(MPI_COMM_WORLD, -1);
           }
           recentList = recentList->nextItem;
           recentList->i = i;
           recentList->j = j;
           numOfElements++;

           // Set the value of the array.
           set[index] = 1;
        }
     }
  }

  // Prepare for searching set.
  unsigned long longestIndex = 0;
  unsigned long longestValue = 0;
  struct ListItem *currentList = prefixList->nextItem;
  free(prefixList);

  // For each item in the linked list (a 1 initially), start a check diagonally for the longest common
substring.
  for (unsigned long k = 0; k < numOfElements; k++) {
     // Check diagonal.
     unsigned long i = currentList->i;
     unsigned long j = currentList->j;
     unsigned long it = i;
     unsigned long jt = j;
     while (set[((it + 1) * bLength) + jt + 1] == 1) {
        set[((it + 1) * bLength) + jt + 1] = 0;
        it++;
        jt++;
     }
```

```
        // Compare with longest value found so far.
        if (it - i > longestValue) {
            longestValue = it - i;
            longestIndex = i;
        }

        // Go to the next item.
        struct ListItem *nextList = currentList->nextItem;
        free(currentList);
        currentList = nextList;
    }
    free(set);

    // Create the longest string found.
    char *longestString = malloc(sizeof(char) * (longestValue + 1));
    if (longestString == NULL) {
        printf("Error! Unable to allocate memory for longestString: size %lu\n", sizeof(longestValue + 1));
        MPI_Abort(MPI_COMM_WORLD, -1);
    }
    for (unsigned long i = 0; i < longestValue; i++) {
        longestString[i] = a[i + longestIndex];
    }
    longestString[longestValue] = '\0';
    return longestString;
}
```

Appendix D - run.sh for OpenMP:

```
#!/bin/bash
## Provide 3 arguments: NUM_OF_THREADS and MAX_LINES_TO_SAMPLE and NUM_OF_NODES

/usr/bin/time -f "%P\t%M Kb\t%e sec\t$3 nodes\t$1 cores\t$2 lines\tdwarves" -- ./openmp
--threads=$(( $1 * $3 )) --lines=$2 -v
```

Appendix E - superbatch.sh for OpenMP:

```
#!/bin/bash

for i in {1..10}
do
  sbatch --time=10:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=1 --mem=4GB --
            ./run.sh 1 1000000 1
  sbatch --time=5:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=1 --mem=4GB --
            ./run.sh 1 500000 1
  sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=1 --mem=4GB --
            ./run.sh 1 100000 1
  sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=1 --mem=4GB --
            ./run.sh 1 10000 1
  sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=1 --mem=4GB --
            ./run.sh 1 1000 1
  sbatch --time=5:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=2 --mem=4GB --
            ./run.sh 2 1000000 1
  sbatch --time=2:30:00 --constraint=dwarves --nodes=1 --ntasks-per-node=2 --mem=4GB --
            ./run.sh 2 500000 1
  sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=2 --mem=4GB --
            ./run.sh 2 100000 1
  sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=2 --mem=4GB --
            ./run.sh 2 10000 1
  sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=2 --mem=4GB --
            ./run.sh 2 1000 1
  for j in 1000 10000 100000 500000 1000000
  do
    for k in 4 8 16 32
    do
      sbatch --time=1:00:00 --constraint=dwarves --nodes=1 --ntasks-per-node=$k --mem=4GB --
            ./run.sh $k $j 1
    done
    sbatch --time=1:00:00 --constraint=dwarves --nodes=2 --ntasks-per-node=4 --mem=2GB --
            ./run.sh 4 $j 2
    sbatch --time=1:00:00 --constraint=dwarves --nodes=4 --ntasks-per-node=4 --mem=1GB --
            ./run.sh 4 $j 4
    sbatch --time=1:00:00 --constraint=dwarves --nodes=4 --ntasks-per-node=2 --mem=1GB --
            ./run.sh 2 $j 4
    sbatch --time=1:00:00 --constraint=dwarves --nodes=2 --ntasks-per-node=16 --mem=2GB --
            ./run.sh 16 $j 2
    sbatch --time=1:00:00 --constraint=dwarves --nodes=16 --ntasks-per-node=2 --mem=512M --
            ./run.sh 2 $j 16
  done
done
```

Appendix F - Java output parser:

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
      ArrayList<String[]> data = new ArrayList<>();
      try {
         BufferedReader reader =
            new BufferedReader(new InputStreamReader(new FileInputStream("rawOutput.txt")));
         while (reader.ready()) {
            data.add(reader.readLine().split("\t"));
         }
         double[][] doubleData = new double[data.size()][6];
         for (int i = 0; i < data.size(); i++) {
            for (int j = 0; j < 6; j++) {
               doubleData[i][j] = Double.parseDouble(data.get(i)[j]);
            }
         }
         int[][] count = new int[11][5];
         double[][] sum = new double[11][5];
         for (int i = 0; i < data.size(); i++) {
            int x = getNodeCoreIndex((int) (doubleData[i][3]), (int) (doubleData[i][4]));
            int y = getLinesIndex((int) (doubleData[i][5]));
            count[x][y]++;
            //Change following line to get different graphs. Below is lines per second.
            sum[x][y] += (doubleData[i][5]) / (doubleData[i][2]);
         }
         double[][] averages = new double[11][5];
         for (int i = 0; i < 11; i++) {
            for (int j = 0; j < 5; j++) {
               averages[i][j] = sum[i][j] / count[i][j];
            }
         }
         System.out.println("\t1000\t10000\t100000\t500000\t1000000");
         for (int i = 0; i < 11; i++) {
            System.out.print(getNodeCoreString(i));
            for (int j = 0; j < 5; j++) {
               System.out.print("\t" + averages[i][j]);
            }
            System.out.println();
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   private static int getLinesIndex(int lines) {
      switch (lines) {
         case 1000:
            return 0;
         case 10000:
            return 1;
         case 100000:
            return 2;
         case 500000:
            return 3;
         case 1000000:
            return 4;
      }
      return -1;
   }
```

```
    private static String getNodeCoreString(int index) {
        switch (index) {
            case 0:
                return "1n 1c";
            case 1:
                return "1n 2c";
            case 2:
                return "1n 4c";
            case 3:
                return "1n 8c";
            case 4:
                return "1n 16c";
            case 5:
                return "1n 32c";
            case 6:
                return "2n 4c";
            case 7:
                return "4n 2c";
            case 8:
                return "4n 4c";
            case 9:
                return "2n 16c";
            case 10:
                return "16n 2c";
        }
        return "";
    }
    private static int getNodeCoreIndex(int nodes, int cores) {
        switch (nodes) {
            case 1:
                switch (cores) {
                    case 1:
                        return 0;
                    case 2:
                        return 1;
                    case 4:
                        return 2;
                    case 8:
                        return 3;
                    case 16:
                        return 4;
                    case 32:
                        return 5;
                }
                break;
            case 2:
                switch (cores) {
                    case 4:
                        return 6;
                    case 16:
                        return 9;
                }
                break;
            case 4:
                switch (cores) {
                    case 2:
                        return 7;
                    case 4:
                        return 8;
                }
                break;
            case 16:
                return 10;
        }
        return -1;
}  }
```