CIS520-project 4
Design 4
Group member:
JIE ZHENG, JARED WOLFE, FUBIN LUAN

MPI Report

- Why do performance analysis?
    - We're using MPI in different settings to see how it performs. The results will help us to determine whether we should use MPI in certain circumstances.
- So what we need?
    - Hardware configuration

Processors	2x 10-Core Xeon E5-2690 v2
Ram	96GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350
10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]

    - Software configuration
        - Operating System: Centos Linux Version 7
        - Compiler: icc version 18.0.3
- Specs
    - Data sizes - 1k, 10k, 100k, and 1M lines of data
    - 3 GB memory per CPU
    - 1, 2, 4, and 8 Threads

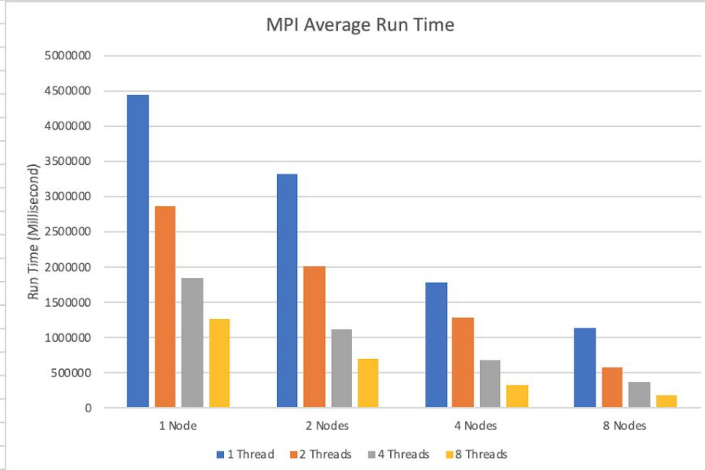
Performance of 1 millions lines of data

	Average Run Time			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	4441989.41	3310427.22	1777350.48	1145602.69
2 Threads	2865799.62	2005226.57	1280273.81	574354.816
4 Threads	1838206.28	1112282.59	680583.836	361986.441
8 Threads	1267389.51	699177.101	326370.963	175587.528

	1st Run - Run Times			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	1584613.13	1018715.23	523214.41	378374.541
2 Threads	1022331.05	515788.887	311356.973	256342.698
4 Threads	502223.957	291948.917	259032.664	146640.619
8 Threads	325069.574	229202.5	81942.7535	65446.613

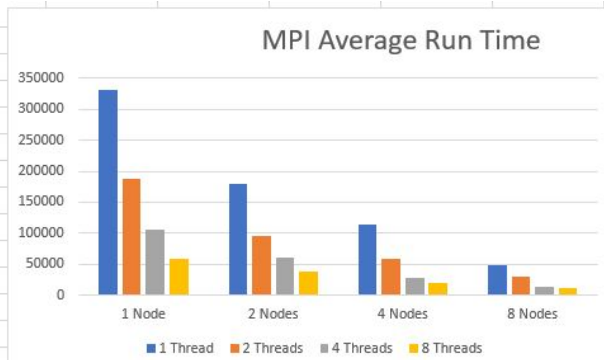
	2nd Run - Run Times			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	1403977.48	1177205.16	638983.176	383026.842
2 Threads	905791.924	665729.159	381923.587	173051.896
4 Threads	668631.246	409997.228	190280.573	107764.018
8 Threads	329038.509	279626.824	119313.885	50689.264

	3rd Run - Run Times			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	1453398.8	1114506.84	615152.896	384201.309
2 Threads	937676.648	823708.521	586993.247	144960.223
4 Threads	667351.081	410336.442	231270.6	107581.805
8 Threads	613281.429	190347.777	125114.324	59451.651



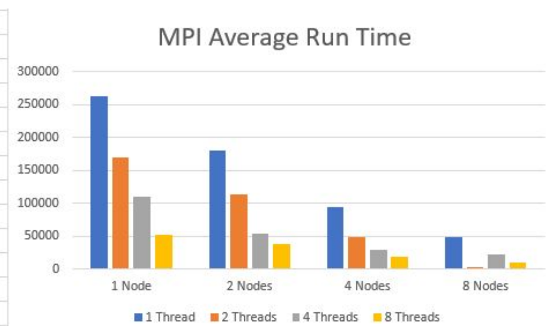
Performance of 100K lines of data

	100K Results			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	330792.3749	180399.138	113048.202	49184.751
2 Threads	187950.213	95454.154	57952.158	28974.033
4 Threads	106339.061	61635.943	28836.705	13638.069
8 Threads	59009.024	37678.037	19163.714	12042.159



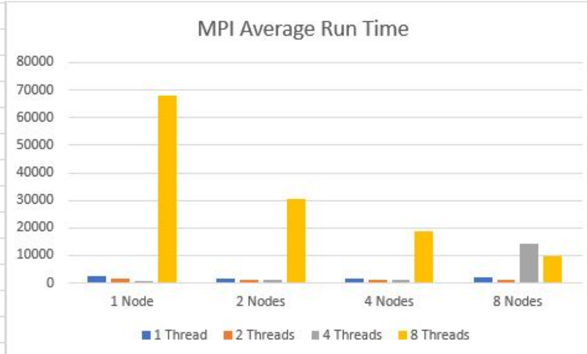
Performance of 10K lines of data

	10K Results			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	263569.162	180310.961	93792.759	47889.345
2 Threads	169760.098	113273.33	48638.795	3294.137
4 Threads	109339.388	53492.79	28821.549	22248.096
8 Threads	51922.112	37696.602	18395.899	9696.231



Performance of 1K lines of data

	1K Results			
	1 Node	2 Nodes	4 Nodes	8 Nodes
1 Thread	2316.9795	1814.958	1716.224	1900.042
2 Threads	1544.653	1202.977	1357.962	1380.79
4 Threads	844.665	935.543	1074.487	14423.639
8 Threads	68113.906	30684.187	18837.127	9826.835



In our MPI code, we reverse the order of one of the send/receive pairs to avoid race condition:

```
MPI_Barrier(MPI_COMM_WORLD);

if(rank == 0)
{
  printData(longestCommonSubstring, chunkSize, rank);
  tag = TAG_PRINT_SUBSTRINGS;
  for(i = 1; i < numTasks; i++)
  {
    MPI_Send(&tag, 1, MPI_LONG, i, TAG_PRINT_SUBSTRINGS, MPI_COMM_WORLD);
    MPI_Recv(&tag, 1, MPI_LONG, i, TAG_DONE_PRINTING, MPI_COMM_WORLD, &Status);
  }
}
else
{
  MPI_Recv(&tag, 1, MPI_LONG, 0, TAG_PRINT_SUBSTRINGS, MPI_COMM_WORLD, &Status);
  printData(longestCommonSubstring, chunkSize, rank);
  tag = TAG_DONE_PRINTING;
  MPI_Send(&tag, 1, MPI_LONG, 0, TAG_DONE_PRINTING, MPI_COMM_WORLD);
}
```

PThreads Report

- We want to compare different tools for multithreading to compare performance in run time and memory usage. This helps us find an optimal solution for the given problem.
- Configuration:

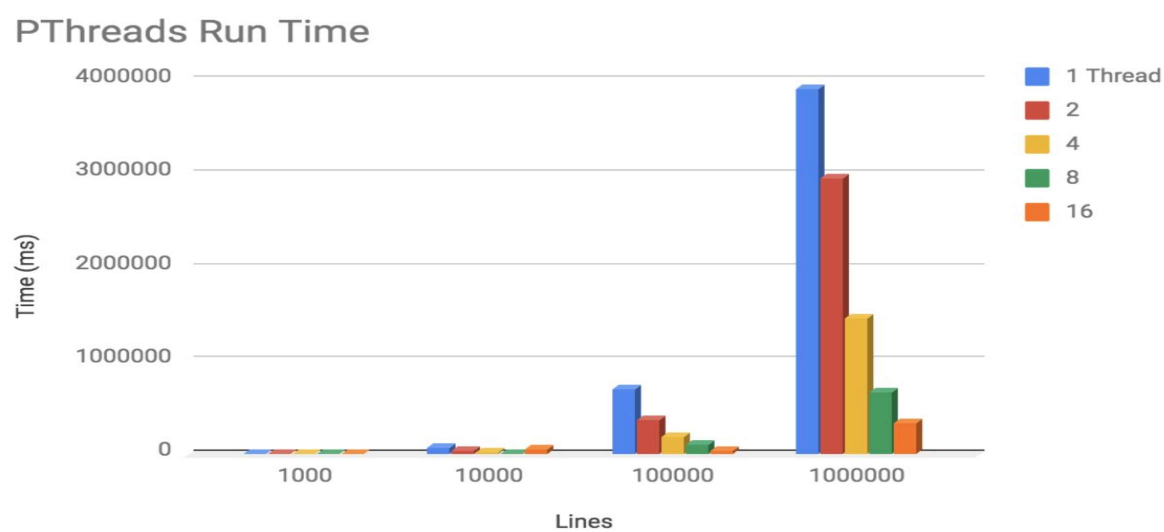
Processors	2x 10-Core Xeon E5-2690 v2
Ram	96GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350

10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]

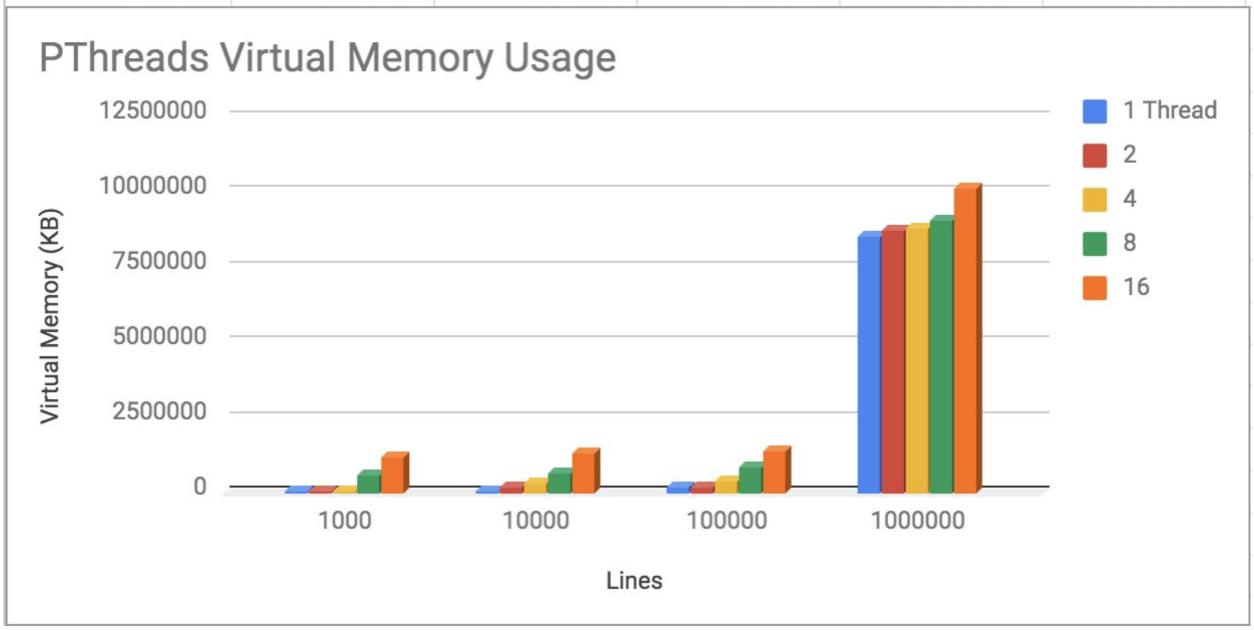
- Operating System: Centos Linux Version 7
- gcc version 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)

- Specs
  - Used 3GB per core
  - 1,2,8,16 threads/cores
  - Data sizes – 1000, 10000, 100000, 1M total lines of input
- Results

Lines/Threads	1 Thread	2	4	8	16
1000	6461.437	3414.916	2404.9735	972.581	727.303
10000	73152.8475	35045.7215	18394.9765	9442.9615	52959.0795
100000	698026.244	365546.0575	191485.336	103940.689	47859.5545
1000000	3915313.532	2955406.999	1463743.713	669788.12	344596.12



Lines/Threads	1 Thread	2	4	8	16
1000	16296	24492	40884	597956	1187812
10000	31500	170768	318232	613160	1290560
100000	184580	192776	405776	831776	1356096
1000000	8554793	8733196	8804224	9088112	10161372



PThreads allows us to indicate a critical section of code. A mutex is used to prevent race conditions.

```
pthread_mutex_lock(&mutexsum);
args->longestCommonSubstring[p] = (char *) malloc((max + 1) * sizeof(char));
memcpy(args->longestCommonSubstring[p], substr, sizeof(char) * max);
args->longestCommonSubstring[p][max] = 0;
pthread_mutex_unlock (&mutexsum);
```

OpenMP

1. We run this on beocat(the HPC cluster at K-State) and we use the approach that called OpenMP, which is an application program interface that used to explicitly direct multi-thread, shared memory parallelism. We use the multi-threaded parallelism to run a function that called find the longest common substring, to read in an huge file called wiki_dump.txt(almost 1.7GB), and by applying the OpenMP with all different sets of threads to increase the time that we need to deal the file.
2. For hardware, we used the Elve nodes

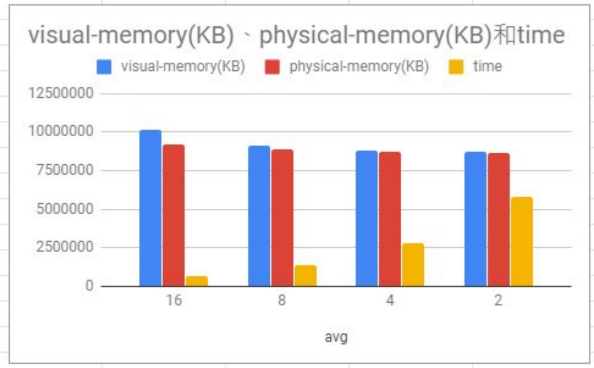
   [57-72,77]

Processors	2x 10-Core Xeon E5-2690 v2
Ram	96GB
Hard Drive	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350
10GbE and QDR Infiniband	Mellanox Technologies MT27500 Family [ConnectX-3]

   For software, we used:
   Linux eos 3.10.0-957.1.3.el7.x86_64 #1 SMP Thu Nov 29 14:49:43 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux, the version is
   NAME="CentOS Linux"
   VERSION="7 (Core)"
   ID="centos"
   ID_LIKE="rhel fedora"
   VERSION_ID="7"
   PRETTY_NAME="CentOS Linux 7 (Core)"
   ANSI_COLOR="0;31"
   CPE_NAME="cpe:/o:centos:centos:7"
   HOME_URL="https://www.centos.org/"
   BUG_REPORT_URL="https://bugs.centos.org/" And;
   gcc version 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)

   3. We run the code with 1000000 lines read in and keep track of the visual-memory and physical-memory and processing time for three times.

openMP	threads	visual-memory(K	physical-memory	time
	16	10161300	9179204	694512.366
	8	9088040	8906760	1353113.799
	4	8804152	8746900	2583967.588
	2	8733124	8655516	6177023.942
	16	10161300	9178968	678457.361
	8	9088040	8906784	1347179.906
	4	8804152	8747144	2907365.463
	2	8733124	8655532	5223089.864
	16	10161300	9178944	689192.24
	8	9088040	8906740	1339576.24
	4	8804152	8747248	2927487.426
	2	8733124	8655440	5910813.998

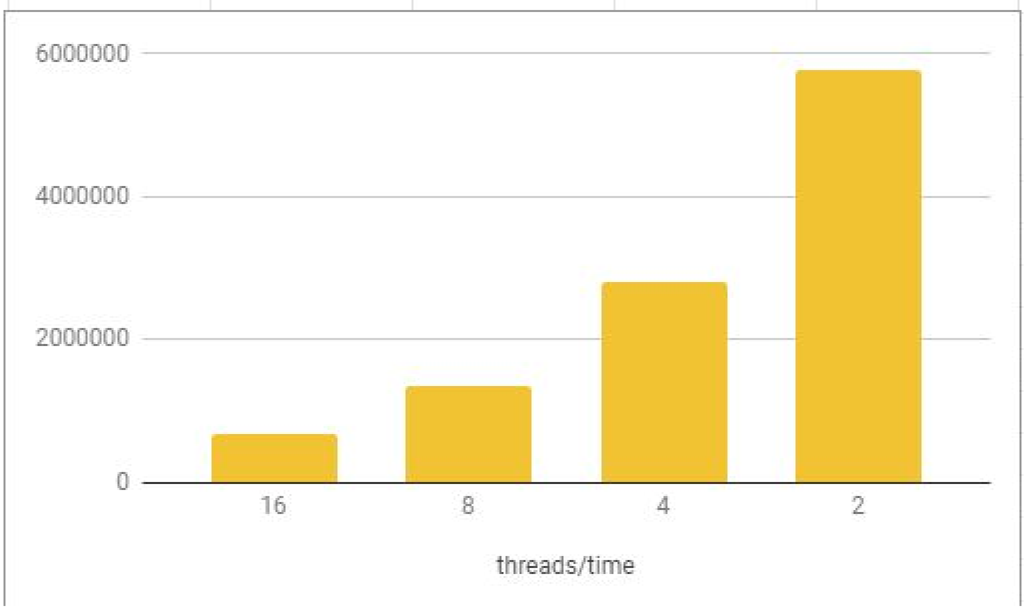


The average for these three times are:

avg	visual-memory(K	physical-memory	time
16	10161300	9179038.667	687387.3223
8	9088040	8906761.333	1346623.315
4	8804152	8747097.333	2806273.492
2	8733124	8655496	5770309.268

For the time comparison:

threads/time	16	8	4	2
	694512.366	1353113.799	2583967.588	6177023.942
	678457.361	1347179.906	2907365.463	5223089.864
	689192.24	1339576.24	2927487.426	5910813.998
	687387.3223	1346623.315	2806273.492	5770309.268



These graph showing that when we use OpenMp to run the program with multi-threads, the memory of the same size of data(1000000 lines) are slightly varys. But the time that processing them are dramatically dropping down when we apply for more threads works together.

One of the advantage of using OpenMP is that we don't require to check the race conditions. But we still can apply synchronization to avoid the race condition. But the synchronization is expensive, we apply it with critical to the crucial part of the paralleling in order to minimize the needs of synchronization.

```
//synchronization of critical
#pragma omg critical
{
  longestCommonSubstring[firstEntryIndex] = (char *) malloc((max + 1) * sizeof(char));
  memcpy(longestCommonSubstring[firstEntryIndex], substr, sizeof(char) * max);
  longestCommonSubstring[firstEntryIndex][max] = 0;
}
```

Only one thread at a time can enter a critical region.

And the region we call the OMP parallel is when we call the LCSalgorithm, which is the most important method we called for this program.

```
#pragma omp parallel for
  for (i = 0; i < NUM_WIKI_LINES - 1; i++)
  {
    algorithm(wiki_dump, longestCommonSubstring, i);
  }
```

References:

https://www.dartmouth.edu/~rc/classes/intro_mpi/mpi_race_conditions.html
https://github.com/levnikolaj/cis520_proj4