CIS 520 Project 4 Performance Analysis

Authors: Christian Hughes, Matt Hixon, & Katie Kristiansen
Date: 04/29/17

STATEMENT OF PURPOSE

We've been tasked with creating a program that searches for list of strings within a set of Wikipedia entries. The program prints out each string that is present in the set of Wikipedia entires, alongside the line number(s) of each Wikipedia entry that the string is found in. The relevant data contains a set of **1,000,000 Wikipedia entires** and **50,000 strings**.

Our goal has been to make this process as fast as possible. We have leveraged the following multi-threading/parallelization libraries:

- Pthreads
- OpenMP
- MPI

We've created a serial version of the program, alongside multiple parallelized versions. This document analyzes the performance variations between different versions of the program, and addresses performance differentials between variations in thread/core/machine count.

EXPERIMENTAL CONFIGURATION

-HARDWARE-

All testing was completed on the **Beocat High Performance Computing Cluster** at Kansas State University. Execution was limited to the "Dwarf" compute nodes, which adhere to one of the following hardware specifications:

Hardware Specification 1:

Processors	2x 16-Core Xeon E5-2683 v4
Ram	128GB
Hard Drive	1x 1TB 7,200 RPM SATA
NICs	4x Broadcom BCM5719
40GbE	Mellanox Technologies MT27520 Family [ConnectX-3 Pro]

Hardware Specification 2:

Processors	2x 16-Core Xeon E5-2683 v4
Ram	128GB
Hard Drive	1x 1TB 7,200 RPM SATA
NICs	4x Broadcom BCM5719
100GbE	Mellanox Technologies MT27700 Family [ConnectX-4]

-SOFTWARE-

Beocat runs **x86_64 Linux** as its primary operating system. All program variations were written in **C** and compiled using **gcc (version 4.9.4)** in a **bash (version 4.3.48)** environment. In addition to the C standard libraries, the following multi-threading/parallelization libraries we used:

- **OpenMP:** Version 2.0.1
- **MPI:** Version 2.0-r4
- **Pthreads:** Version 2.0.7-r3

Beocat uses a job queueing system called **SGE** that manages/schedules program execution. All of our results were derived by submitting jobs, and capturing their results.

EXPERIMENTAL PROCEDURES

Each version of the program consists of the the following ordered steps, regardless of implementation:

1. Initialize an array (potentially very large) for storing the results of the search

function.
2. Read all data into memory. This includes the set of Wikipedia entries, and the list of strings to search for.
3. Search for each string within the set of Wikipedia entries.
4. Print out the results of the search.

Of these 4 steps, **only step 3 is parallelized**. Steps 1, 2, and 4 all run in a single thread. Additionally, step 4 cannot run until all threads have completed step 3.

All tests measure performance using **1,000,000 Wikipedia entries** and **50,000 strings**. We initially tested each implementation using a smaller data set (75 Wikipedia entries and 50 words) to ensure correctness. However, smaller data sets provide little insight into performance at scale; the compute time/memory requirements increase exponentially as the number of Wikipedia entires and search strings goes up.

File Access: The files containing the Wikipedia entires and search strings are hosted locally in the Beocat filesystem. None of the test data requires network utilization.

RAM Utilization: Every test is allocated **64GB of RAM** regardless of other parameters. Actual utilization hovers around **~48GB**. In tests that utilize multiple cores, 64GB of memory is allocated total (each core receives 64GB/Num of Cores). Because the results of the search function are stored in a single large array, allocating less RAM leads to a severe performance penalty; any amount of RAM lower than 48GB prevents the program from cacheing its entire working set.

The following parameters vary across test cases, and provide the basis for our performance analysis:

- **# of Machines**
- **# of Cores**
- **# of Threads**

Note: We ran the Linux *diff* command to ensure the correctness of our output files across tests. Beyond test-specific performance information, all tests produce identical (and correct) output.

-BASELINE IMPLEMENTATION-

We created a single-threaded serial version of the program to establish a performance baseline. This program (***/base/base_solution_prod.c***) was tested *exclusively* with the following parameters:

- # **of Machines:** 1

- **# of Cores:** 1
- **# of Threads:** 1

The program was run 10 times to establish an average performance metric. The results are as follows:

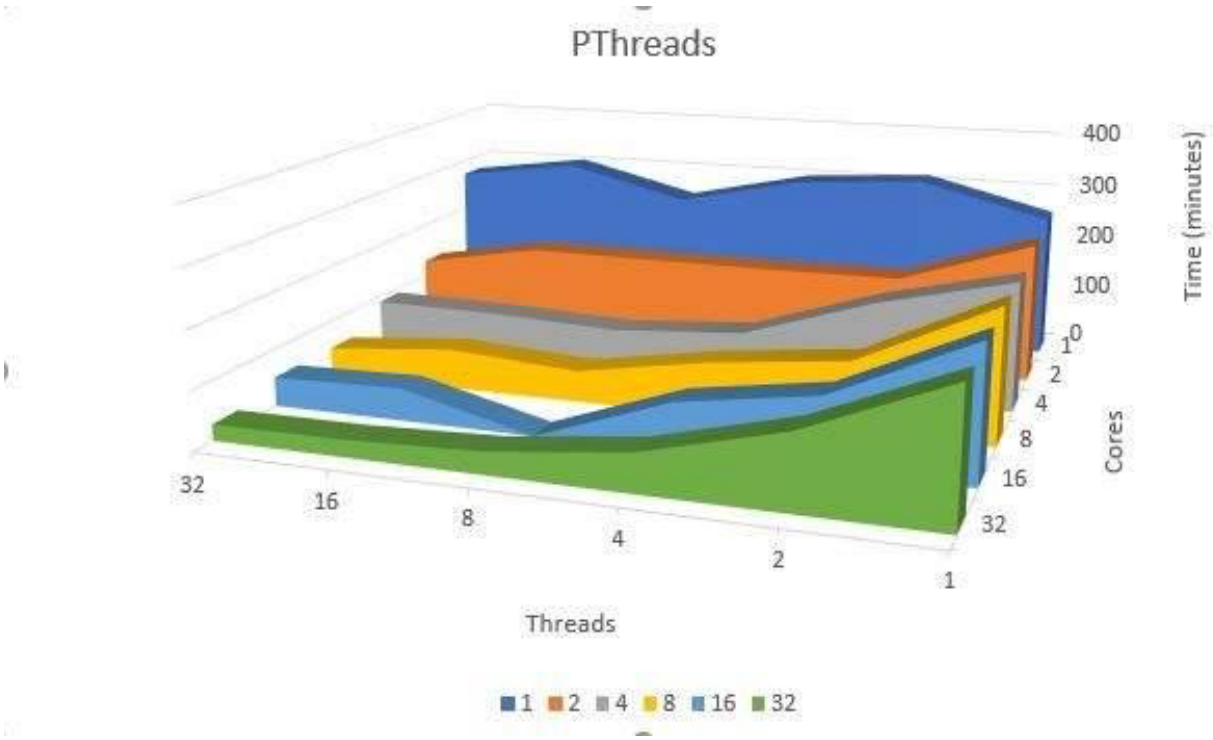
Trial Number	Time (In Milliseconds)	Time (In Minutes)
1	5867755.169000	97.795919483333
2	5859280.746000	97.6546791
3	6424992.005000	107.08320008333
4	6415012.288000	106.91687146667
5	6442890.049000	107.38150081667
6	6432940.929000	107.21568215
7	6086670.140000	101.4445023333
8	6121067.262000	102.0177877
9	7083612.984000	118.0602164
10	7077397.950000	117.9566325
AVERAGE:	6381161.95 Milliseconds	106.35 Minutes

On average, the baseline implementation took **~106 minutes** to reach completion. No parallelism is implemented. All other implementations are modifications of this one.

-PTHREADS IMPLEMENTATION-

This implementation parallelizes the program using the **Pthreads** library. All tests were run using a single machine, with many combinations of threads and cores. The results are as follows:

Pthreads Implementation Performance							
	Number of Threads						
Number of Cores		1	2	4	8	16	32
	1	257.15 min	314.75 min	299.34 min	248.57 min	304.47 min	272.26 min
	2	247.93 min	168.90 min	166.86 min	166.93 min	168.36 min	127.45 min
	4	227.04 min	171.11 min	94.47 min	78.13 min	87.30 min	90.26 min
	8	233.98 min	120.38 min	94.26 min	55.23 min	70.19 min	50.55 min
	16	230.41 min	**120.05 min**	84.05 min	Not Tested	56.29 min	50.98 min
	32	**223.96 min**	125.02 min	**64.82 min**	**37.99 min**	**29.37 min**	**27.99 min**



Some key insights are as follows:

- Single core performance is significantly slower than the base implementation, regardless of the number of threads.

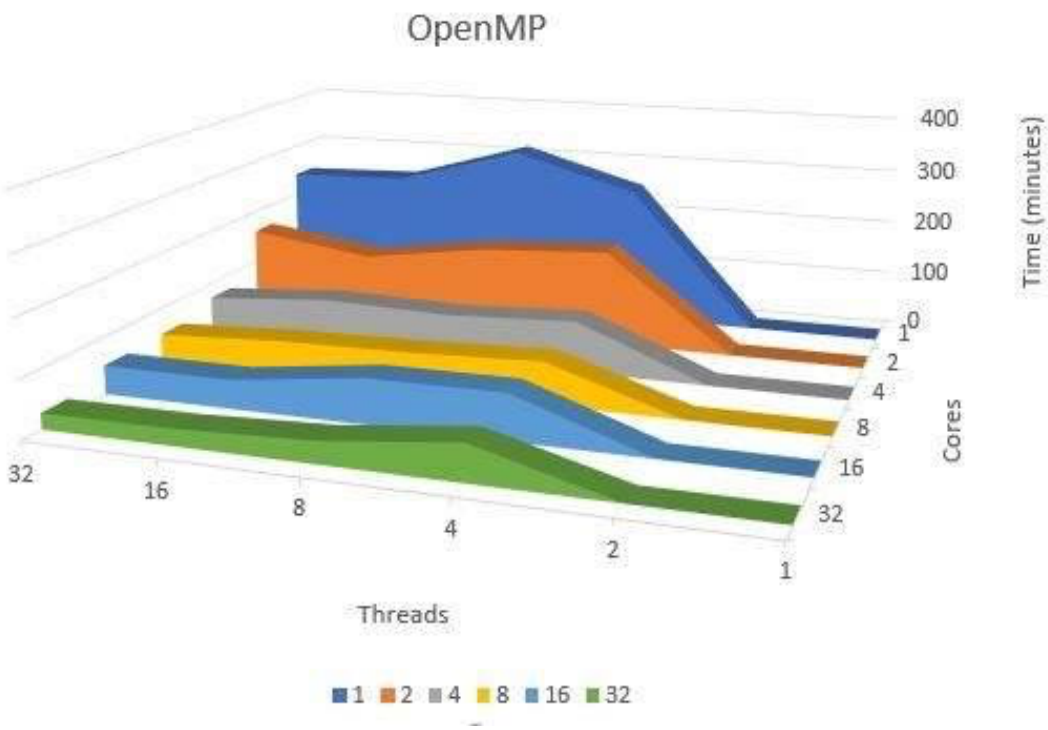
- Adding additional cores almost universally increases performance, regardless of the number of threads.
- The fastest times occur when the number of cores is equal to or exceeds the number of threads.
- Every test that utilizes **4 or more cores** alongside **4 or more threads** is faster than the base implementation.
- The fastest time (27.99 minutes) is about **3.8x faster** than the base implementation.

Note: Some tests were queued, but never actually executed (some jobs remained in the queue for well over 24 hours).

-OPENMP IMPLEMENTATION-

This implementation parallelizes the program using the **OpenMP** library. All tests were run using a single machine, with many combinations of threads and cores. The results are as follows:

OpenMP Implementation Performance							
	Number of Threads						
Number of Cores		1	2	4	8	16	32
	1	Not Tested	Not Tested	251.05 min	315.05 min	248.16 min	241.30 min
	2	Not Tested	Not Tested	174.02 min	161.11 min	128.57 min	160.99 min
	4	Not Tested	Not Tested	92.48 min	82.64 min	89.14 min	72.91 min
	8	Not Tested	Not Tested	74.75 min	66.60 min	63.43 min	52.87 min
	16	Not Tested	Not Tested	79.99 min	78.69 min	50.27 min	50.23 min
	32	Not Tested	Not Tested	**62.81 min**	**37.74 min**	**28.81 min**	**27.56 min**



Some key insights are as follows:

- Single core performance is significantly slower than the base implementation, regardless of the number of threads.
- Adding additional cores almost universally increases performance, regardless of the number of threads.
- The fastest times occur when the number of cores is equal to or exceeds the number of threads.
- Every test that utilizes **4 or more cores** alongside **4 or more threads** is faster than the base implementation.
- The fastest time (27.56 minutes) is about **3.8x faster** than the base implementation.

Note: Some tests were queued, but never actually executed (some jobs remained in the queue for well over 24 hours).

-MPI IMPLEMENTATION-

Despite our attempts to duplicate the MPI functionality found in the provided examples, we were unable to complete a working version. The results printed by our program were inconsistent, and attempts to scale the execution across multiple machines were not successful. Therefore, there is no relevant performance analysis for this implementation. We estimate that a working MPI implementation would have seen the fastest times as it would allow for distribution across multiple machines.

CONCLUSIONS

Our parallelized implementations provide a clear boost over our serial implementation in cases where cores/threads are abundant. When looking at the results for one thread in both the Pthread and OpenMP versions of our software, we noticed that performance was much worse than that of the serial solution. We believe that this is a result of the number of cache misses that occur when you utilize multiple cores with only one thread. As the number of cores was increased, however, we were able to overcome the cache misses and gain marginal improvements in the overall timing.

The performance characteristics of Pthreads and OpenMP are strikingly similar. The fastest test clocked in at **27.56 minutes** — we believe that some of our other tests may be faster. None of the tests eclipsed the 17 minute baseline that was established for this project.