Justin Bahr
NUID: 001592707

High Performance Computing
Homework 1 - Question 1

a. I decided to write three of my own single-threaded benchmark programs in C++: Q1_floatbench.cpp, Q1_matrixbench.cpp, and Q1_intbench. Q1_floatbench measures how long it takes for the machine to perform 100,000 operations of multiplying two random randomly generated floats from two different arrays and storing the result in a third array. Q1_matrixbench measures how long it takes the machine to multiply two 100 by 100 matrices, one of which is sparse with all zeros in every other row and column. Q1_intbench measure it how long takes the machine to perform 100,000 sets of the operation a[i] = a[i]*b[i]+c[i]-d[i], where a, b, c, and d are all arrays with randomly generated integers 0 to 99. I ran my benchmarks on the COE Vector Linux platform (Intel ® Xeon ® CPU E5-2698 v4 @ 2.20GHz, 20 cores per socket, 2 sockets, with 791957492 KB of memory, and Linux 4.18.0). I also ran my benchmarks on the Explorer Cluster (Intel ® Xeon ® CPU E5-2680 v4 @2.40GHz, 14 cores per socket, 2 sockets, with 263358372 KB of memory, and Linux 5.14.0). Here are my results in nanoseconds:

COE Vector	floatbench	matrixbench	intbench		Explorer CPU	floatbench	matrixbench	intbench
1	342129	2868836	244280		1	463050	3417824	298990
2	365538	2865207	244851		2	462095	3425246	304071
3	347190	2868035	260284		3	460860	3415969	294133
4	338590	2866489	236267		4	464151	3420006	294709
5	335440	2866122	243035		5	456016	3418779	294299
6	336632	2861709	244266		6	463309	3415046	308050
7	342211	2865857	243033		7	464256	3417915	295011
8	339197	2865484	242979		8	456511	3415699	293271
9	335431	2911364	236522		9	471946	3416369	315798
10	336302	3011810	241484		10	463160	3415192	293780
Average (ns)	341866	2885091.3	243700.1		Average (ns)	462535.4	3417804.5	299211.2

While each benchmark type consistently had a fairly narrow band of runtimes, there was some variation. Overall, the main thing I noticed was that matrixbench ran on the COE Vector system had the widest range of runtimes, with most runtimes sitting around 2.866 milliseconds and two outliers at 2.911 and 3.012 milliseconds. Since these benchmarks use randomly generated numbers, I attribute most of the runtime variation to differences in input variables' sizes. I believe that matrixbench had more variation since it was composed of a sparse matrix multiplication, where fewer input variables are multiplied by a non-zero number, but these variables are operated on multiple times. In summary, a smaller deviation to the input variables could have a larger effect on runtime. Another possible reason for variation could be whether any background processes were running on the machine, taking up resources. Some other notably slower runtimes were 260284 nanoseconds for intbench and 365538 nanoseconds for floatbench.

b. For all three benchmarks, the COE Vector system was about 1.25x faster than the Explorer CPU system. These systems have different versions of Linux and different processor models. I believe the runtime difference is due to the different processors. Since all of these benchmarks are single threaded, the number of cores on each CPU should not have an effect on runtime. I was initially surprised to see that the COE Vector system was faster because the Explorer CPU uses a faster clock speed (2.4GHz vs 2.2GHz). However, the COE Vector system CPU is a model E5-2698, and the Explorer Cluster CPU is a model E5-2680. The E5-2698 has more cache storage (40 MB vs 20 MB), which could contribute to a faster runtime if there is a lower percentage of cache misses. Another factor could be subtle improvements in the microarchitecture, especially any improvements targeting single-threaded performance.

c. I applied the -O2 optimization flag, which is a common optimization flag in the g++ compiler used to gain better performance without a significant increase in compilation time. I found this quite effective to reduce run time. Here are the runtimes and speedups of the optimized executables in the COE Vector and Explorer CPU systems:

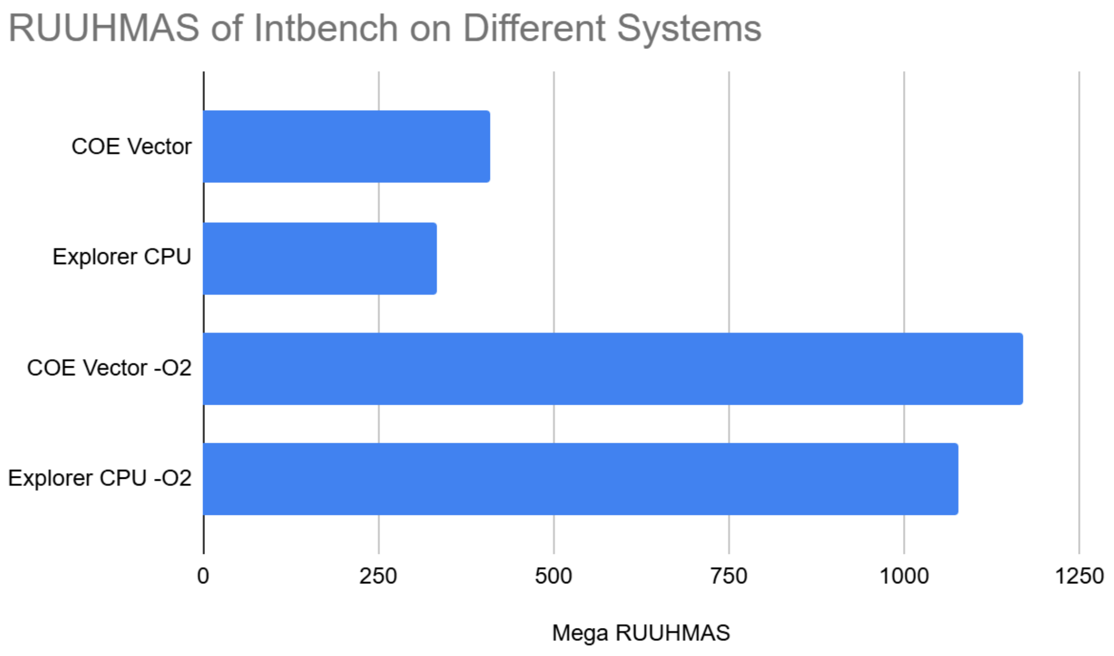
COE Vector	floatbench	matrixbench	intbench
Average (ns)	178055.1	600486.1	85521.2
Speedup	1.92	4.8	2.85

Explorer CPU	floatbench	matrixbench	intbench
Average (ns)	246816.5	722878.1	92768.1
Speedup	1.87	4.73	3.23

The -O2 flag can make a number of optimizations including function inlining, improving cache utilization, removing redundant calculations, simplifying loops, and rearranging instructions. Matrixbench has the most effective speedup of 4.73-4.8, followed by intbench with a speedup of 2.85-3.23, and finally floatbench with a speedup of 1.87-1.92.

I expected matrixbench to have a large speedup, since the zero-valued rows and columns leave many operations that can be avoided if this condition (multiply by zero) is checked first. The -O2 optimization likely removed many redundant computations for matrixbench, and may have performed loop blocking to reduce cache misses. Intbench likely benefited from the instruction rearrangement done by -O2. This benchmark performs a variety of integer operations, some of which are much faster than others. It may have also benefited from effective cache prefetching. I expected floatbench to have the lowest speedup, which matches with my results. Floating point multiplication does not have as many obvious ways to improve performance. However, a speedup of almost 2 could have been caused by prefetching optimizations and instruction reordering.

d. Based on my intbench workload, which multiplies, adds, and subtracts integers with values 0 through 99, I have created the RUUHMAS (**R**andom **U**nsigned ints **U**nder a **H**undred **M**ultiply **A**dd and **S**ubtracts per second) metric. Specifically, this is a measure of how many instances of Y = A*B+C-D operations can be performed in one second when A, B, C, and D are all integers from 0 to 99. This is basically an overall measure of how fast a machine can handle very basic integer math. I could see this or a similar metric being useful for graphics, cryptography, and data processing applications which use many integer computations.



e. I could obtain a large additional speedup from rewriting a multi-threaded floatbench, which is naturally parallelizable. For both floatbench and intbench, I would partition my input and output arrays into as many sub arrays as I had available Pthreads. Then I would pass each Pthread a sub array and the relevant computations(s) and storage (written as a separate function) would be the assigned task. Next, I would join the Pthreads back together after each completes its section. Similarly for matrixbench, I would partition the first matrix operand by rows, one section for each Pthread. Then I would pass each Pthread the first sub matrix, the full second matrix, and a separate matrix multiplication function would be the assigned task. Finally, I would join the Pthreads and stack the outputs on tops of each other in order to form the full product matrix. This works because each index is computed independently and matrix multiplication works as follows [R1xC1]*[R2xC2] = [R1xC2], for resultign dimensions where R and C are the number or rows and columns in a matrix respectively.