EL307:
Introduction to performance programming with applications in Bioinformatics

Project 3:
Optimize and Parallelize LD computations

To speed up the computation of LD two methods were performed. The first optimization, is the fairly simple, use of the register keyword. By using this we request from the compiler to store the variable in a processor register rather than in regular memory. The CPU generally moves data to and from the register in order to use it for the specified operations. A variable that is very often used in a program such as a for iterator, is a good candidate to be stored on register. In our case, the iterators that parse through the region_a and region_b array are moved on to the register. We also request that the compressed_size be moved to the register, since it evaluates the iterators at every for loop, as well as the counters that are incremented.

The second optimization, built on top of the first one, is the, not so elegant, method of loop unrolling. Every time you run a for loop the compiler is building a checking mechanism on top. By unrolling the loop, we reduce the total number of checks that have to be performed. In our case, we tell the program: “what you would do in 5 iterations I want you to do it in 1” and so we reduce the number of checks that will need to be performed.

Below the runtimes of both optimizations are displayed as well as their speed gain compared to our baseline:

RUNTIME (seconds)	1st run	2nd run	3rd run	Average
OPTIMIZED (baseline)	5.37801	6.05638	5.80546	5.7466166667
OPTIMIZATION_REGISTER	4.7581	4.80504	4.81153	4.7915566667
OPTIMIZATION_REGISTER_LOOP_UNROLLING	4.71782	4.94312	5.01267	4.8912033333

SPEEDUP	
OPTIMIZATION_REGISTER	1.1993214453
OPTIMIZATION_REGISTER_LOOP_UNROLLING	1.1748881155

There is a 20% gain of performance once we use the register, however we don’t see any gain from the loop unrolling method. By moving variables to the register we already made the for loop evaluation fast, so reducing the number of for loop evaluations by loop unrolling doesn’t give us any noticeable difference.

Finally, we move on to a more advance optimization technique, parallelization with OpenMP. Using pragma we distribute the iterations of the for loop to different threads. By incrementing by the number of threads in each loop we basically distribute the iterations in a cyclic manner, similarly to how one distributes a deck of cards. Our sequential program was overwriting upon the same array it was using, but since we are doing that process in parallel now we need to be sure the different iterations are concurrent. To make sure that no element is overwritten before usage, we simply use new arrays to store the compressed form. That way different iterations won’t conflict with one another.

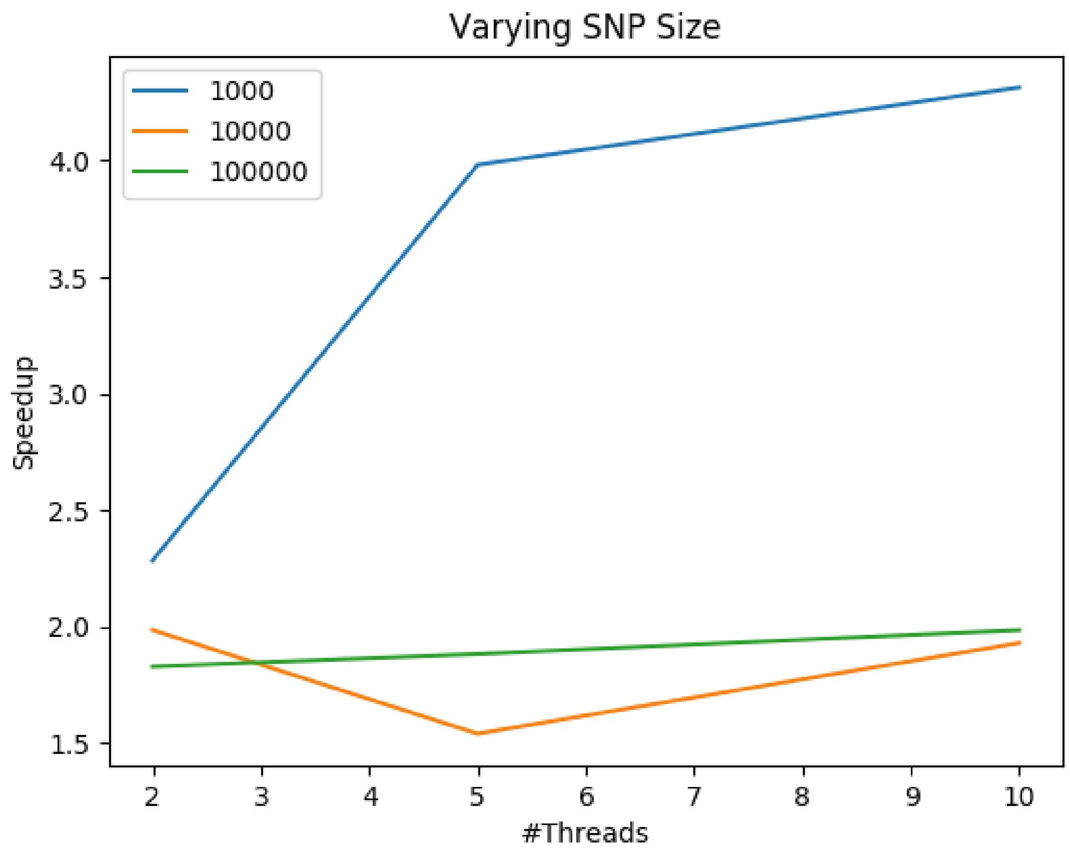
Below are the runtime results of this implementation for number of threads 2, 5 and 10 with varying range of region b size and snp size. The runtime for our previous optimization (sequential register optimization) is also displayed.

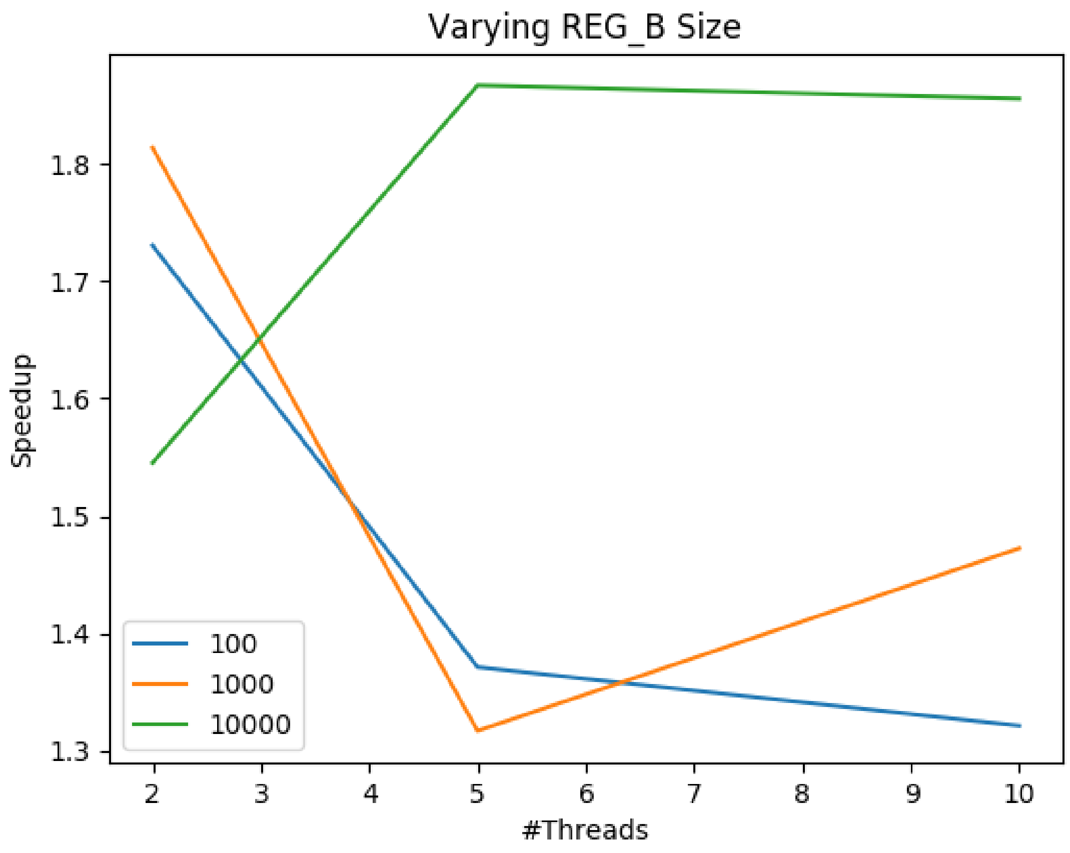
Varying REG_B_SIZE	RegisterOpt	#THREADS					
		2		5		10	
	Runtime	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup
100	0.50141	0.28975	1.7304918033	0.36565	1.3712840148	0.37946	1.3213777473
1000	4.5734	2.52159	1.8136969135	3.47248	1.3170414228	3.10594	1.4724688822
10000	47.99647	31.05931	1.545316686	25.70965	1.8668659433	25.86621	1.8555663934

Varying SNP_SIZE	RegisterOpt	#THREADS					
		2		5		10	
	Runtime	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup
1000	1.21946	0.53385	2.284274609	0.30624	3.9820402299	0.28268	4.3139238715
10000	5.64801	2.84474	1.9854222178	3.66529	1.5409449184	2.92676	1.9297824215
100000	55.63322	30.41864	1.8289187156	29.55292	1.8824948601	28.02728	1.9849667895

The speed up gains of multi-threading compared to our sequential optimization are also demonstrated in the 2 plots below:

For small snp size the scaling is good with increasing number of threads going up as high as 4 times faster, but when it comes to large size of snp we hardly see any difference as we increase the number of threads. With SNP constant to 10.000 and varying the size of REG_B we never gain more than 2x performance.. More threads seem to perform better when increasing REG_B size.


Varying SNP Size
1000
10000
100000
Speedup
4.0
3.5
3.0
2.5
2.0
1.5
2
3
4
5
6
7
8
9
10
#Threads


Varying REG_B Size
100
1000
10000
Speedup
1.8
1.7
1.6
1.5
1.4
1.3
2
3
4
5
6
7
8
9
10
#Threads