Algorithm Description and parallelization approach

stencil_1D.c:

The stencil_1D algorithm incrementally walks an array taking the averages between [i-2,i+2] and places that value into the i'th element. After each pass, the algorithm runs again increasingly smoothing the values. My parallelization approach simply parallelizes the inner loop since the outer loop has a dependency on it's previous iteration. Since calculating each element i in our array takes relatively the same amount of time, we can use a static allocation strategy.

stencil_2D.c:

The stencil_2D algorithm works similarly to the stencil_1D algorithm but extends the calculation into a grid. When calculating each value of i, rather than averaging [i-2, i+2], this algorithm averages all 8 neighbors to the point as well as the point itself. When approaching parallelizing this code, I chose to set up a parallel for on the outer loop much like the approach to stencil_1D. Statically allocating threads also seemed optimal since the thread load can easily be split up fairly.

mat_vec.c:

The mat_vec.c algorithm calculates the value c[i] such that c[i] equals the sum of all values in A[i]. Since the value of c[i] incrementally increases over the domain of j, each run of the inner loop must be done thread locally. Naturally, this forces us to place our parallelization on the outer loop. Much like the last two problems we can statically allocate threads to increase performance because the load should be naturally balanced.

Experimental Setup

Machine Specifications

Name: loveland.cs.colostate.edu
Cores: 12
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 15360K

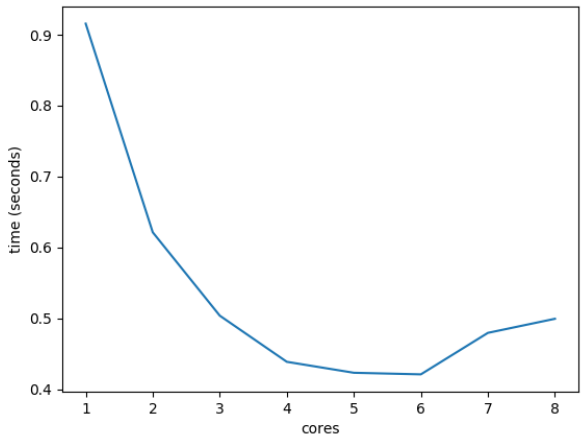
Testing each algorithm using 1-8 cores. Each algorithm is loaded into the python test harness which does 7 runs on each core count. The highest and lowest value for each test are dropped and the remaining 5 are averaged to produce our result. Afterwards, each run time is compared to the sequential baseline. The commands for each experiment are as follows:

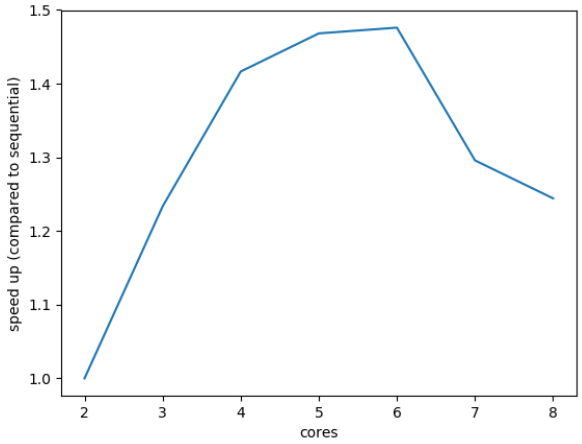
$stencil_1D 4000 200000
$stencil_2D 800 2000
$mat_vec 25000 10000

Experimental Results

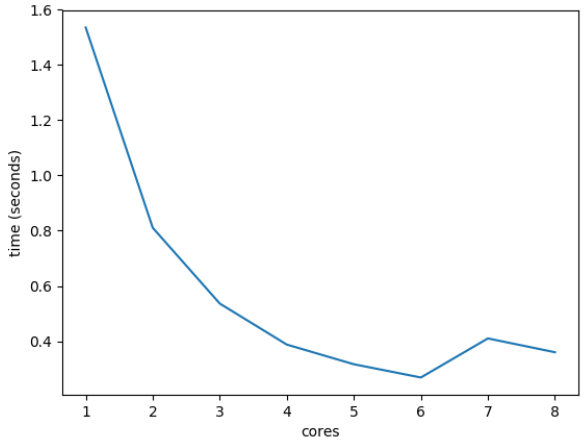
As a general trend, the sequential time was the slowest in comparison to any of the threaded times. Nearly each experiment showed a linear speedup as the number of cores assigned to the program increased which eventually tapers off around 7 cores. I assume this occurs when the number of fork/join operations in each iteration begin to dominate the run time.

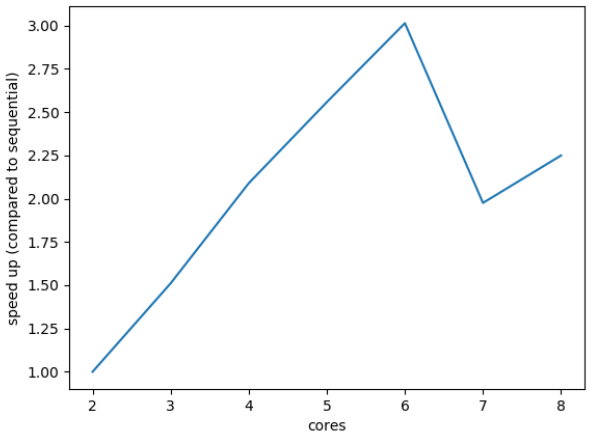
stencil_1D





stencil_2D





mat_vec