To run program run the script: ./run-program $1 $2 $3

- if you enter 3 arguments then the program will run sequential version (a, b) bounds

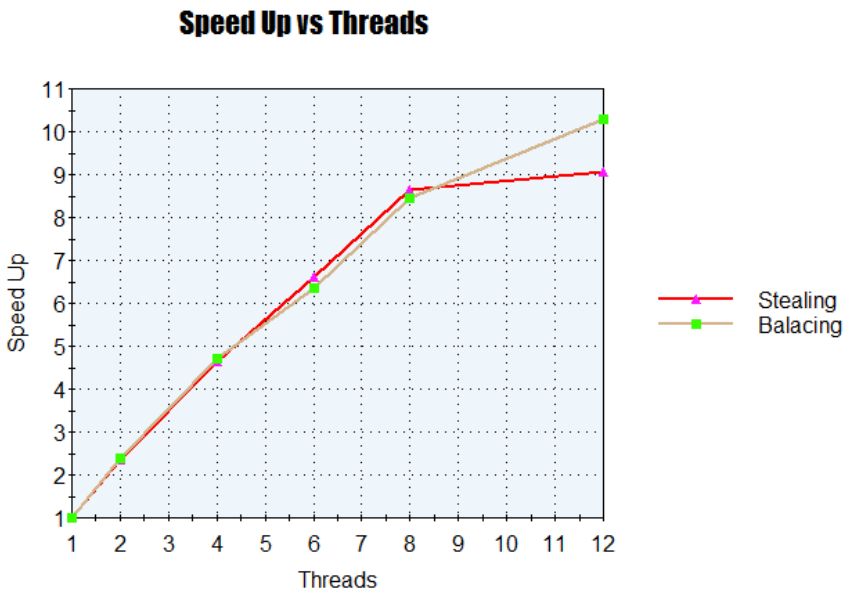
- if you enter 3 arguments then the program will run parallel version with $1 = threadCount, $2= A, $3 = B (A and B are bounds are integral)

This program (sequentially and in parallel) is used to approximate a integral value of the line y = x^3 over a set of bounds that the user enters. It uses the monte-carlo method would basically randomly selected many points in a closed square and determines if the point lie within the integral and takes the fraction of points that line within the integral and multiples by the area of that is enclosed to get an approximation.

To implement the parallel solution I have to figure out a way to divide the problem into very small subproblems. I decided to do that by dividing the bounds into very small piece and calling a "task" a certain number of points being randomly calculated (as described above) in a very small interval. The smaller intervals could be done in parallel and then summed and returned. This would be way faster than a sequential version for many points to be calculated.

The difficulty of this specific parallel implementation was trying to figure out how to creating a stealing queue if the queue reached a certain threshold or became empty. I solved this by creating an infinite for loop and when the queue reached that threshold it would try to steal from another queue randomly. If that queue was already empty or didn't create many elements then that particular thread would finish what it had in its queue and be marked as done. Once all threads marked themselves as being done then

the Shutdown method would stop waiting and we could return a sum of all the intervals approximations.



As we can see for the program it is very parallelizeable. This is expected since almost all aspected of a monte-carlo approximation of an integral would be parallelizeable since everything can be approximated separately then added up. Since we don't have to worry about the creation and tear down of threads every time they finish it also improves speedup.

The speedup would not be as noticeable if I used a smaller overall interval. I decided to use a -10 to 10 (x-axis bounds) for the speedup for both parallel and sequential versions. I

did this to amplify speedup when lots of data points are expected to be calculated. If I used a smaller interval (such as 0 to 2) then we would not see such a great speed up.

I believe that there is a slight difference in the speed up between the stealing and balancing algorithms. I believe that the balancing algorithm is a better implementation since we never let one of the queues go to empty so it keeps the queues way more balanced. I think that is difference between the two parallel algorithms would be more noticeable if I picked a larger queue threshold. For this speedup graph for balancing I only picked 20 tasks and since these tasks can be done so quickly then the difference is almost negligible.