Share Memory Relaxation Parallel Program (hm474)

Approach

For this coursework I have two arrays. An array that contains the data to do relaxation on and a working array which threads can edit without interfering with each other. This doubles the memory consumption but allow threads to update values simultaneously. Each thread is assigned a certain number of elements in the array to average. For example if there were 3 threads and working on a 5 by 5 array then index 6 to 8 go to thread 1, index 11 to 13 go to thread 2 and index 16 to 18 where threads cannot edit the boundary values. The program uses barriers to synchronise threads. Firstly all the threads work out the average of all the 4 neighbours around each of there assigned indices in the array. The results are placed in the working array, then all the threads wait on the barrier once the thread have calculated the averages of all of it's assigned value. Once all the threads have reached this barrier they move on to update values in the data array. The barrier stops quick threads from updating value that slower threads are still using to calculate averages. Another barrier causes threads to repeat the process at the same time.

To terminate the program every element must not change by more than 0.1 after it's neighbours are averaged. When all of a thread's assigned values are no longer changing by 0.1 then the thread will incremented a locked shared variable. This variable needs to be locked as all the threads use it and race condition may occur. Once the threads have finished averaging they check to see if all the other threads have finished using the shared variable if not they go back around the loop again so that the barriers are unlocked.

The reasons I used this method was so that threads were only created and destroyed once. Both updating and averaging could be done in parallel. The only critical region was the single shared variable used to terminate the program. The critical region is only once updated for each thread meaning the lock is rarely locked hence less thread waiting time.

Correctness Testing

Firstly I repeatedly ran the program calculating the same size arrays using different numbers of threads to make sure that the same result appeared each time. Below is a sample of those tests. I ran each example here multiple times to make sure there were no race conditions causing incorrect values.

10 by 10 with 1 thread:
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.898438 0.818359 0.687500 0.734375 0.750000 0.750000 0.750000 0.859375 1.000000
1.000000 0.818359 0.687500 0.578125 0.500000 0.500000 0.601074 0.609375 0.800781 1.000000
1.000000 0.687500 0.578125 0.500000 0.500000 0.519043 0.596436 0.682587 0.806641 1.000000
1.000000 0.734375 0.500000 0.500000 0.529083 0.584238 0.659417 0.745520 0.857780 1.000000
1.000000 0.750000 0.500000 0.519043 0.584238 0.652289 0.724109 0.801050 0.891317 1.000000
1.000000 0.750000 0.601074 0.596436 0.659417 0.724109 0.787231 0.850353 0.919891 1.000000
1.000000 0.750000 0.609375 0.682587 0.745520 0.801050 0.850353 0.896791 0.946557 1.000000
1.000000 0.859375 0.800781 0.806641 0.857780 0.891317 0.919891 0.946557 0.976532 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000

10 by 10 with 5 threads:
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.898438 0.818359 0.687500 0.734375 0.750000 0.750000 0.750000 0.859375 1.000000
1.000000 0.818359 0.687500 0.578125 0.500000 0.500000 0.601074 0.609375 0.800781 1.000000

1.000000 0.687500 0.578125 0.500000 0.500000 0.519043 0.596436 0.682587 0.806641 1.000000
1.000000 0.734375 0.500000 0.500000 0.529083 0.584238 0.659417 0.745520 0.857780 1.000000
1.000000 0.750000 0.500000 0.519043 0.584238 0.652289 0.724109 0.801050 0.891317 1.000000
1.000000 0.750000 0.601074 0.596436 0.659417 0.724109 0.787231 0.850353 0.919891 1.000000
1.000000 0.750000 0.609375 0.682587 0.745520 0.801050 0.850353 0.896791 0.946557 1.000000
1.000000 0.859375 0.800781 0.806641 0.857780 0.891317 0.919891 0.946557 0.976532 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000

The centre of 100 by 100 array using 5 threads (centre 20x20):
0.571499 0.571503 0.571506 0.571511 0.571515 0.571520 0.571526 0.571531 0.571537 0.571543
0.571503 0.571505 0.571508 0.571511 0.571514 0.571518 0.571522 0.571526 0.571530 0.571534
0.571506 0.571508 0.571510 0.571512 0.571514 0.571516 0.571518 0.571521 0.571523 0.571525
0.571511 0.571511 0.571512 0.571512 0.571513 0.571514 0.571515 0.571516 0.571517 0.571518
0.571515 0.571514 0.571514 0.571513 0.571513 0.571512 0.571512 0.571511 0.571511 0.571511
0.571520 0.571518 0.571516 0.571514 0.571512 0.571511 0.571509 0.571508 0.571506 0.571505
0.571526 0.571522 0.571518 0.571515 0.571512 0.571509 0.571507 0.571504 0.571502 0.571500
0.571531 0.571526 0.571521 0.571516 0.571511 0.571508 0.571504 0.571501 0.571498 0.571495
0.571537 0.571530 0.571523 0.571517 0.571511 0.571506 0.571502 0.571498 0.571494 0.571492
0.571543 0.571534 0.571525 0.571518 0.571511 0.571505 0.571500 0.571495 0.571492 0.571489

The centre of 100 by 100 array using 20 threads (centre 20x20):
0.571499 0.571503 0.571506 0.571511 0.571515 0.571520 0.571526 0.571531 0.571537 0.571543
0.571503 0.571505 0.571508 0.571511 0.571514 0.571518 0.571522 0.571526 0.571530 0.571534
0.571506 0.571508 0.571510 0.571512 0.571514 0.571516 0.571518 0.571521 0.571523 0.571525
0.571511 0.571511 0.571512 0.571512 0.571513 0.571514 0.571515 0.571516 0.571517 0.571518
0.571515 0.571514 0.571514 0.571513 0.571513 0.571512 0.571512 0.571511 0.571511 0.571511
0.571520 0.571518 0.571516 0.571514 0.571512 0.571511 0.571509 0.571508 0.571506 0.571505
0.571526 0.571522 0.571518 0.571515 0.571512 0.571509 0.571507 0.571504 0.571502 0.571500
0.571531 0.571526 0.571521 0.571516 0.571511 0.571508 0.571504 0.571501 0.571498 0.571495
0.571537 0.571530 0.571523 0.571517 0.571511 0.571506 0.571502 0.571498 0.571494 0.571492
0.571543 0.571534 0.571525 0.571518 0.571511 0.571505 0.571500 0.571495 0.571492 0.571489

A quick eye test confirms that they are the same regardless of the number of threads being used.

Consistency of results

I ran the same configurations three times to see if the timing of executions on Balena was consistent.

Array Size = 1000 by 1000 Threads = 16 Trials = 4

Results:
15.374754
15.234983
15.356826
15.237290

Mean = 15.300963
Standard deviation = 0.0651410

Array Size = 500 by 500 Threads = 16 Trials = 4

Results:
1.962217
1.967810
1.964309
1.962991

Mean = 1.964331
Standard deviation = 0.00214465

Array Size = 200 by 200 Threads = 16 Trails = 4

Results:
0.375377
0.375450
0.375423
0.375365

Mean = 0.375404
Standard deviation = 0.000034

The standard deviation for all of these results are insignificant therefore it is accurate to say that the variation in Balena's executions time is insignificant.

Scalability

It is possible to scale a parallel program in two ways. By increases the number of processors (p) or increasing the amount of work (N). In the case of my program the work is the size of the square matrix. The speed up equation is:

Sp = time on a sequential processor / time on p parallel processors

This gives the number of time greater the speed of a program is on p processors than on a uni-processor. To calculate how much speed up increases as p increase (efficiency) I will use the equation :

Ep = Sp/p

Karp-Flatt equation (below) calculates the percentage of execution time spent running sequentially code.

E = ((1/sp) – (1/p)) / (1 – (1/p))

Where E is proportion of the program that is sequential. If the value of E is greater than 1 then the parallelisation of the program is slowing the program down.

Results

Below are the results of testing. Below that are graphs that show this data.

```
Speed Up = 0.980228, Efficiency = 0.490114, Threads = 2.000000, Work = 20, Karp-Flatt = 1.040341
```

Speed Up = 1.768825, Efficiency = 0.884412, Threads = 2.000000, Work = 50, Karp-Flatt = 0.130694
Speed Up = 1.941096, Efficiency = 0.970548, Threads = 2.000000, Work = 100, Karp-Flatt = 0.030346
Speed Up = 1.982569, Efficiency = 0.991285, Threads = 2.000000, Work = 200, Karp-Flatt = 0.008792
Speed Up = 1.996034, Efficiency = 0.998017, Threads = 2.000000, Work = 500, Karp-Flatt = 0.001987
Speed Up = 1.995319, Efficiency = 0.997659, Threads = 2.000000, Work = 750, Karp-Flatt = 0.002346
Speed Up = 1.997986, Efficiency = 0.998993, Threads = 2.000000, Work = 1000, Karp-Flatt = 0.001008

Speed Up = 1.042037, Efficiency = 0.347346, Threads = 3.000000, Work = 20, Karp-Flatt = 0.939488
Speed Up = 2.389891, Efficiency = 0.796630, Threads = 3.000000, Work = 50, Karp-Flatt = 0.127644
Speed Up = 2.807573, Efficiency = 0.935858, Threads = 3.000000, Work = 100, Karp-Flatt = 0.034269
Speed Up = 2.924907, Efficiency = 0.974969, Threads = 3.000000, Work = 200, Karp-Flatt = 0.012837
Speed Up = 2.949405, Efficiency = 0.983135, Threads = 3.000000, Work = 500, Karp-Flatt = 0.008577
Speed Up = 2.960562, Efficiency = 0.986854, Threads = 3.000000, Work = 750, Karp-Flatt = 0.006661
Speed Up = 2.965346, Efficiency = 0.988449, Threads = 3.000000, Work = 1000, Karp-Flatt = 0.005843

Speed Up = 0.940190, Efficiency = 0.188038, Threads = 5.000000, Work = 20, Karp-Flatt = 1.079519
Speed Up = 3.006113, Efficiency = 0.601223, Threads = 5.000000, Work = 50, Karp-Flatt = 0.165819
Speed Up = 4.359131, Efficiency = 0.871826, Threads = 5.000000, Work = 100, Karp-Flatt = 0.036754
Speed Up = 4.744782, Efficiency = 0.948956, Threads = 5.000000, Work = 200, Karp-Flatt = 0.013447
Speed Up = 4.878392, Efficiency = 0.975678, Threads = 5.000000, Work = 500, Karp-Flatt = 0.006232
Speed Up = 4.909778, Efficiency = 0.981956, Threads = 5.000000, Work = 750, Karp-Flatt = 0.004594
Speed Up = 4.925442, Efficiency = 0.985088, Threads = 5.000000, Work = 1000, Karp-Flatt = 0.003784

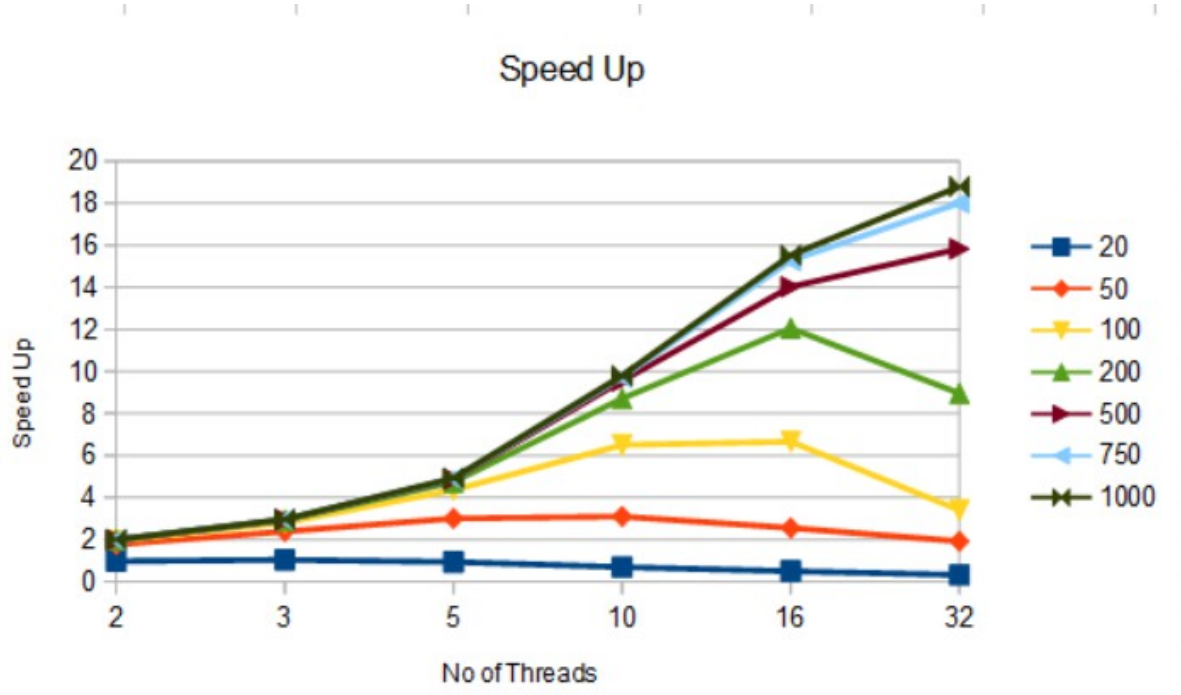
Speed Up = 0.694879, Efficiency = 0.069488, Threads = 10.000000, Work = 20, Karp-Flatt = 1.487889
Speed Up = 3.098292, Efficiency = 0.309829, Threads = 10.000000, Work = 50, Karp-Flatt = 0.247509
Speed Up = 6.501063, Efficiency = 0.650106, Threads = 10.000000, Work = 100, Karp-Flatt = 0.059801
Speed Up = 8.708908, Efficiency = 0.870891, Threads = 10.000000, Work = 200, Karp-Flatt = 0.016472
Speed Up = 9.553624, Efficiency = 0.955362, Threads = 10.000000, Work = 500, Karp-Flatt = 0.005191
Speed Up = 9.720951, Efficiency = 0.972095, Threads = 10.000000, Work = 750, Karp-Flatt = 0.003190
Speed Up = 9.793517, Efficiency = 0.979352, Threads = 10.000000, Work = 1000, Karp-Flatt = 0.002343

Speed Up = 0.501556, Efficiency = 0.031347, Threads = 16.000000, Work = 20, Karp-Flatt = 2.060047
Speed Up = 2.558150, Efficiency = 0.159884, Threads = 16.000000, Work = 50, Karp-Flatt = 0.350301
Speed Up = 6.661293, Efficiency = 0.416331, Threads = 16.000000, Work = 100, Karp-Flatt = 0.093462

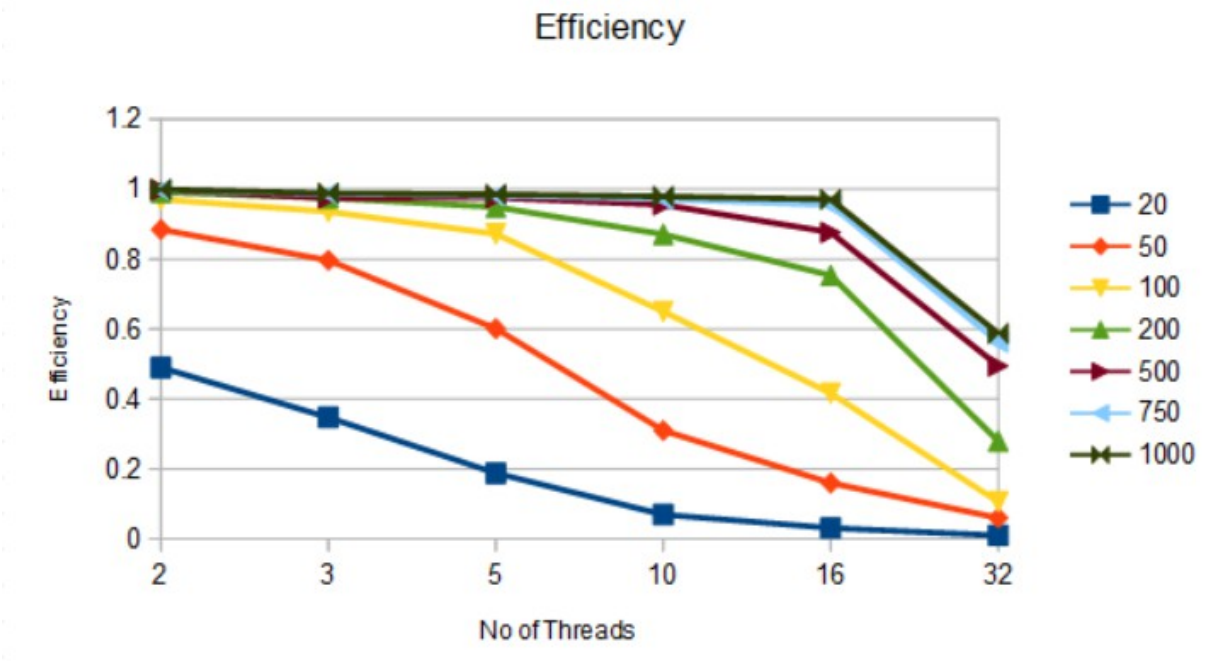
```
Speed Up = 12.055881, Efficiency = 0.753493, Threads = 16.000000, Work = 200,
Karp-Flatt = 0.021810
Speed Up = 14.025013, Efficiency = 0.876563, Threads = 16.000000, Work = 500,
Karp-Flatt = 0.009388
Speed Up = 15.274936, Efficiency = 0.954683, Threads = 16.000000, Work = 750,
Karp-Flatt = 0.003165
Speed Up = 15.532927, Efficiency = 0.970808, Threads = 16.000000, Work = 1000,
Karp-Flatt = 0.002005

Speed Up = 0.323625, Efficiency = 0.010113, Threads = 32.000000, Work = 20,
Karp-Flatt = 3.157411
Speed Up = 1.924007, Efficiency = 0.060125, Threads = 32.000000, Work = 50,
Karp-Flatt = 0.504257
Speed Up = 3.401954, Efficiency = 0.106311, Threads = 32.000000, Work = 100,
Karp-Flatt = 0.271173
Speed Up = 8.958942, Efficiency = 0.279967, Threads = 32.000000, Work = 200,
Karp-Flatt = 0.082963
Speed Up = 15.827345, Efficiency = 0.494605, Threads = 32.000000, Work = 500,
Karp-Flatt = 0.032962
Speed Up = 18.009448, Efficiency = 0.562795, Threads = 32.000000, Work = 750,
Karp-Flatt = 0.025060
Speed Up = 18.794723, Efficiency = 0.587335, Threads = 32.000000, Work = 1000,
Karp-Flatt = 0.022665
```

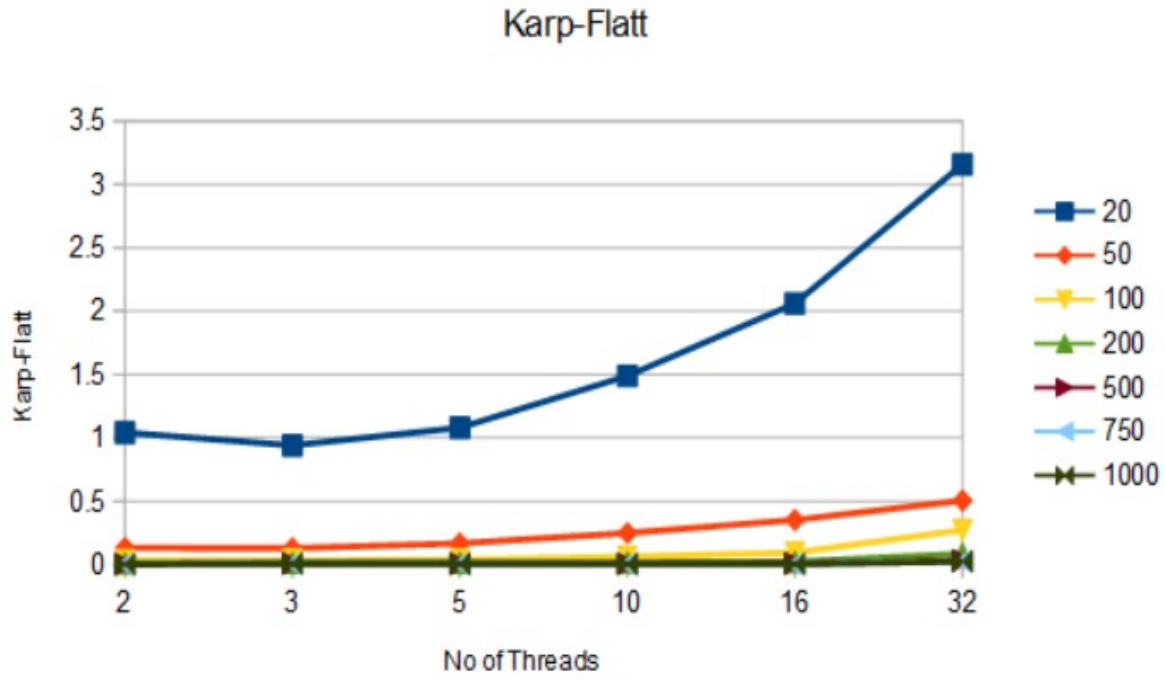
Graphs

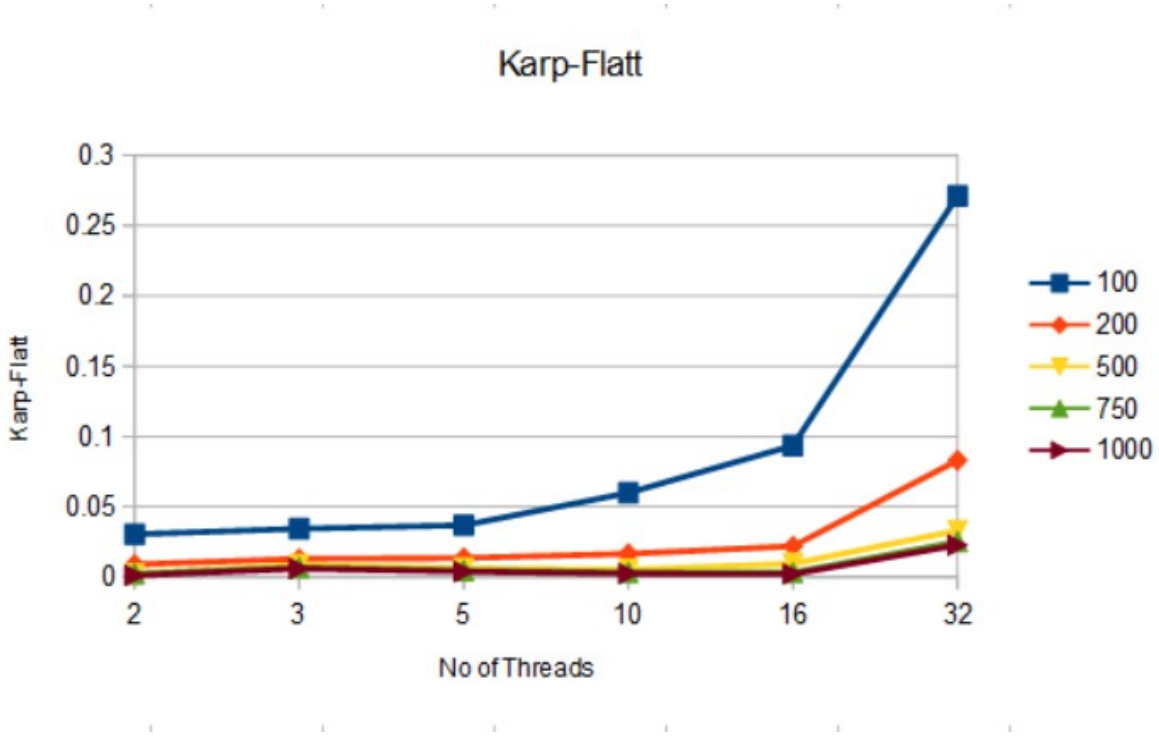


Above is the speed up plotted against the number of threads used for each computation. Each line represents an array of n by n at a certain number of threads t. Where the number in the key is n. It is clear from the number of threads that the speed up increases more when the number of elements in the array are greater. The decreases in the increases in speed up occurs at a lower number of threads the lower n is.



This graph shows efficiency at different amounts of n. The efficiency decreases at every n as the number of threads increase. However the greater n is the slower this decrease in efficiency is. The efficiency significantly decreases at 32 threads as CPUs start to share those threads as there are only 16 CPUs available.





The two graphs above down the Karp-Flatt value at threads t. The second graph shows the same data without 20 and 50 so that the higher values of n are more visible. Both these graphs show how the sequential parts of the program increases as the number of threads increase. But the higher values of n cause the program's sequential parts to decrease. This shows my program's parallel parts (e.g. averaging) scale faster at least initially than it's sequential parts (e.g. thread creation). The graphs also show the thread scheduling costs of running more than 1 thread on a CPU. If each thread is calculating a very small sub-section of the array then the cost of making those calculations parallel will be too large compared to the time saved calculating those values in parallel. When n is larger then it takes more threads to make the subsections of array small.