# CITS3402: Project 1: Efficiency Optimisation with OpenMP

**Zachary Newman**
**21149116**

**Aim:** To parallelize the fem1d.c code using OpenMP to improve performance time and also provided analysis of this improved performance from the old code.

**Notes:**
The original code has a bugs in it that produce a segmentation faults when set to certain parameters. For the sake of analysis these have marked (X) and ignored.

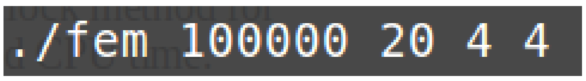
**Process:**
The first modifications to the c code where a general rearrangement of the structure, moving method descriptions above their method headers and making appropriate variables in main into global variables so that each function does not have to pass these every time. Basically this process involved removing redundancies in the code.

Next global array declarations where change to dynamic memory allocations to allow larger array sizes. Not only will this allow the code to produce a better approximated finite element answer but it also allows us to see the the benefits of OpenMp parallelism when running a large process. Most printf statements where then altered to fprintf statement which read to selected files for correctness testing later. The omp_get_wtime method was used over the clock method for evaluating time as this measured actual passing time while clock measured CPU time.

It was decided to alter the code to run both the old unparallelized and the new parallelized code sequentially when the program was executed. This was done to conduct better analysis on the improved performance of the program. The code was also altered to provide arguments; The first specifying the subdivision size, the second the degree of basis functions used, third the number of threads used for the new version of the program and fourth the number of trails or executions for both versions. The Average function then computes the average process time for the trails of each version. This is done to eliminate possible outliers caused by runtime conditions in our analysis.

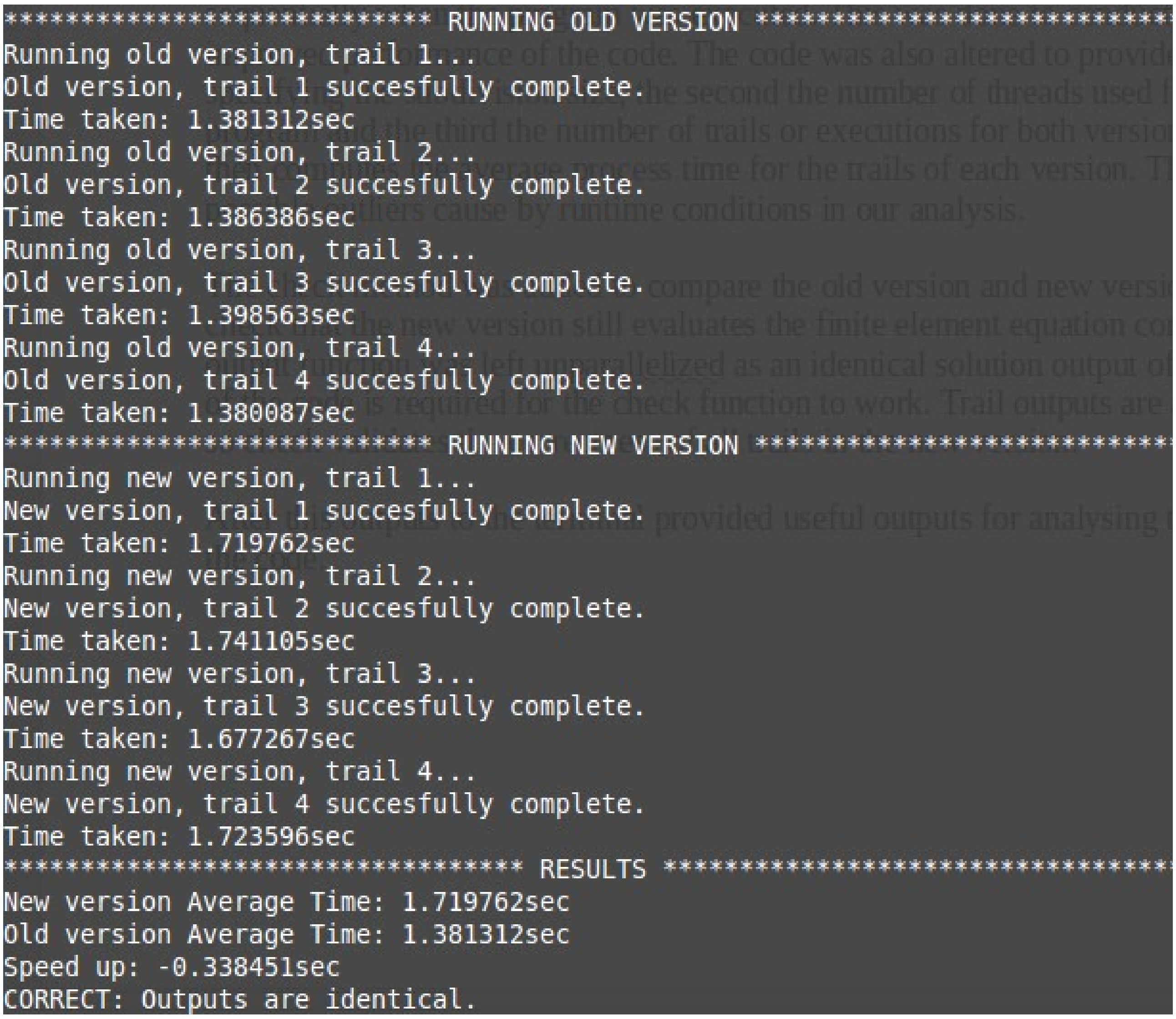
The check method was added to compare the old versions and new versions output solutions and check that the new version still evaluates the finite element equation correctly. Trail outputs are append to the solution file so check validates the correctness of all trails in the new version in a single call.

After this outputs to the terminal provided useful information for analysing the effects of parallelizing the code. The first code parallelized was the for loops inside the geometry method. Here all loops could be parallelized using the OpenMP for construct. Here no variables assigned multiple times in the loop so all threads maintained independence making the outcome correct. After compiling (with -fopenmp flag) The following command was run from terminal:



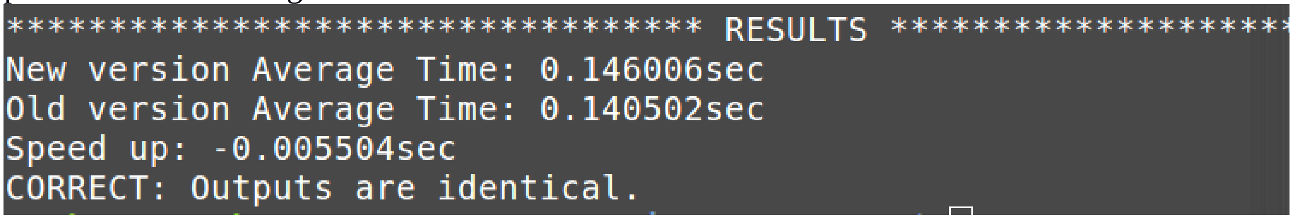
```
./fem 100000 20 4 4
```

This ran the program with 100,000 subdivisions, using a basis function of 20 and 4 trails for both the parallelized version and the original version. The parallelized version used 4 threads in execution. The following output was produced.



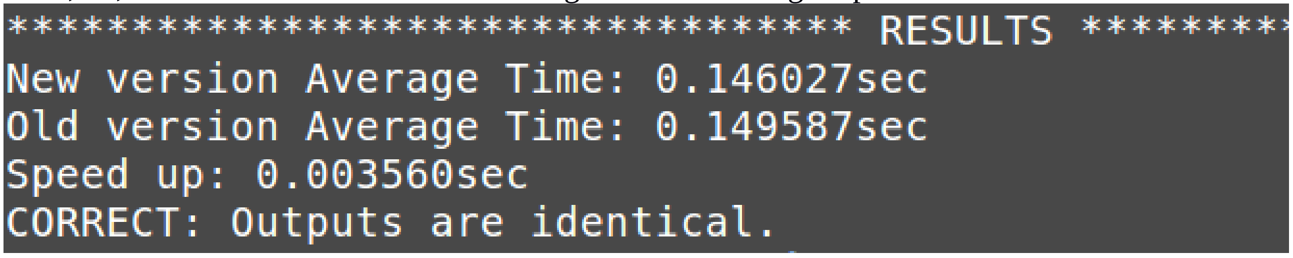
```
****************************** RUNNING OLD VERSION ******************************
Running old version, trail 1...
Old version, trail 1 succesfully complete.
Time taken: 1.381312sec
Running old version, trail 2...
Old version, trail 2 succesfully complete.
Time taken: 1.386386sec
Running old version, trail 3...
Old version, trail 3 succesfully complete.
Time taken: 1.398563sec
Running old version, trail 4...
Old version, trail 4 succesfully complete.
Time taken: 1.380087sec
****************************** RUNNING NEW VERSION ******************************
Running new version, trail 1...
New version, trail 1 succesfully complete.
Time taken: 1.719762sec
Running new version, trail 2...
New version, trail 2 succesfully complete.
Time taken: 1.741105sec
Running new version, trail 3...
New version, trail 3 succesfully complete.
Time taken: 1.677267sec
Running new version, trail 4...
New version, trail 4 succesfully complete.
Time taken: 1.723596sec
*********************************** RESULTS ************************************
New version Average Time: 1.719762sec
Old version Average Time: 1.381312sec
Speed up: -0.338451sec
CORRECT: Outputs are identical.
```

As seen above although outputs were correct for the parallelized code the average time increased, causing a speed down of more than 0.3 seconds. The slow down was credited to the fprintf statements inside the parallized region. Slow interactions with I/O streams and unbuffered data made these poor functions to parallelize. These where removed from the parallelized region and executed sequentially, while no wait constructs where also added to the parallelized loops who's succeeding loops where not dependant on their changes. This way threads did not have to wait for loop barriers before continuing execution thus utilising the computers CPUs more effectively reducing run time. The same was also done for the loops that initialise global arrays to zero in the assemble function. Re-compiling and running 100 trails with 10,000 subdivision and four threads produced the following result:



```
*********************************** RESULTS *******************
New version Average Time: 0.146006sec
Old version Average Time: 0.140502sec
Speed up: -0.005504sec
CORRECT: Outputs are identical.
```

It seemed the speed down had been reduced to less than 6 milliseconds meaning performance had been improved but was still just under the performance of the original code. Further modications were made. Loops in both the assemble and geometry methods were reduced to a single parallelized loop. This drastically reduced the iterations made by the program. The ouptut function was also parallelized. By changing u to an array which stored the output for later printing the output could be calculated in a single parallelized loop improving performance. Running these updates with 1000 trails, 10,000 subdivisions and four threads gave the following output.



```
********************************** RESULTS ********
New version Average Time: 0.146027sec
Old version Average Time: 0.149587sec
Speed up: 0.003560sec
CORRECT: Outputs are identical.
```

The parallelized version now shows a speed up in performance. With the code close to an optimised algorithm the next analysis involved testing the effects of different numbers of subdivisions, threads, and basis functions.

A major change which increased the reliability of analysis was changing the program to toggle between trails of old and new versions of the process. This meant that changes in uncontrollable run time factors such as the number of operating system and background process running would effect both old and new trails executions equally and inaccuracies between time differences would be minimised in the results.

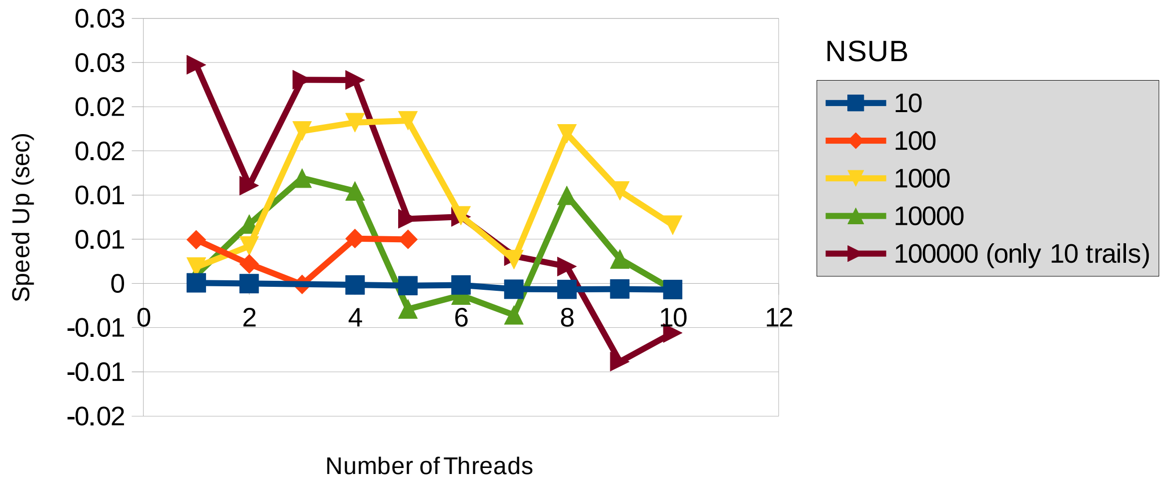
**Analysis:**

The optimisation of threads involved testing speed up time at various levels of subdivisions with different numbers of threads.

**./fem a 40 x 50**

| | | | **NSUB (a)** | | |
|---|---|---|---|---|---|
| **threads (x)** | **10** | **100** | **1000** | **10000** | **100000 (only 10 trails)** |
| 1 | 0.000059 | 0.004951 | 0.001847 | 0.000922 | 0.024747 |
| 2 | -0.000017 | 0.002213 | 0.004228 | 0.00671 | 0.01107 |
| 3 | X | -0.000116 | 0.017251 | 0.011922 | 0.023051 |
| 4 | -0.000163 | 0.005079 | 0.01821 | 0.010464 | 0.023018 |
| 5 | -0.00025 | 0.004995 | 0.018422 | -0.002889 | 0.00732 |
| 6 | -0.000182 | X | 0.00767 | -0.001336 | 0.007532 |
| 7 | -0.000638 | X | 0.002687 | -0.003571 | 0.003097 |
| 8 | -0.000663 | X | 0.016941 | 0.009965 | 0.001918 |
| 9 | -0.000622 | X | 0.010479 | 0.002763 | -0.008844 |
| 10 | -0.000684 | X | 0.006601 | -0.000692 | -0.005596 |
| 20 | -0.000496 | | -0.000262 | 0.011384 | 0.01639 |
| 50 | -0.004586 | 0.004632 | 0.00403 | 0.010346 | 0.002918 |
| 100 | -0.005958 | -0.002216 | 0.005709 | -0.000739 | 0.00446 |
| 1000 | -0.041831 | -0.023262 | -0.008171 | -0.026873 | -0.015738 |
| 10000 | -0.273046 | -0.252266 | -0.260528 | -0.251605 | -0.244457 |

Performance at Varying Number of Threads with Increasing Subdivision



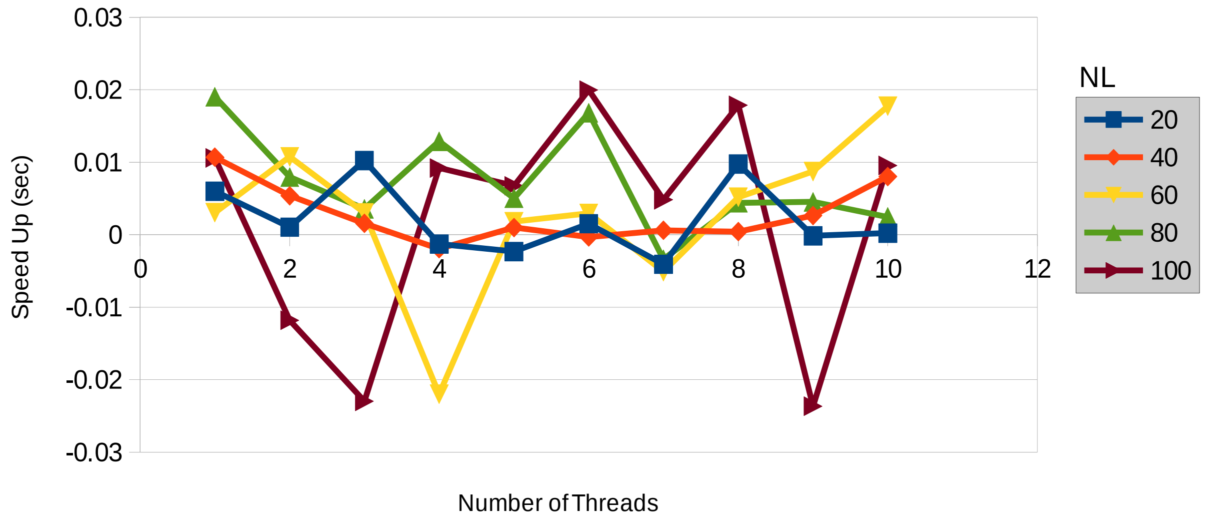
The above results show speed up for thread between the range of 1 to 10 threads. Speed up is very volatile, showing max performance at round 3 to 4 threads and 8 threads for most subdivision sizes. The rapid changes in performance could be due to the relation between loop lengths and the division of labour between the threads. 3, 4 or 8 Threads possibly divide the entire loops workload very evenly between the processors 4 cores while 7 threads poorly allocates workloads causing more cache misses in runtime causing greater calls to memory and a loss in performance. The results also show that even with a single thread the new version improves performance meaning even acting sequentially the performance is improved. This is due to the reduction in the number of for loops in the code meaning less iterations.

Next different numbers of threads were tested with different degrees of polynomial base function to see how this effected performance.

**./fem 10000 a x 50**

| | | | | **NL (a)** | |
|---|---|---|---|---|---|
| **threads (x)** | **20** | **40** | **60** | **80** | **100** |
| 1 | 0.005988 | 0.01068 | 0.00303 | 0.019039 | 0.010591 |
| 2 | 0.001027 | 0.005372 | 0.010743 | 0.007945 | -0.011813 |
| 3 | 0.01022 | 0.001546 | 0.003003 | 0.003554 | -0.02299 |
| 4 | -0.001294 | -0.001937 | -0.02201 | 0.012882 | 0.009201 |
| 5 | -0.002329 | 0.000987 | 0.001793 | 0.005028 | 0.006734 |
| 6 | 0.001506 | -0.000316 | 0.002923 | 0.016806 | 0.019959 |
| 7 | -0.004087 | 0.0006 | -0.005095 | -0.003395 | 0.004802 |
| 8 | 0.009738 | 0.000417 | 0.005166 | 0.0044 | 0.017851 |
| 9 | -0.000159 | 0.00264 | 0.008726 | 0.004515 | -0.023668 |
| 10 | 0.000216 | 0.008019 | 0.017717 | 0.002441 | 0.009545 |
| 20 | 0.001459 | -0.002727 | 0.006884 | 0.066023 | 0.01269 |
| 50 | -0.000053 | -0.000388 | 0.002178 | 0.006704 | 0.010033 |
| 100 | 0.000154 | -0.003089 | 0.005035 | 0.018212 | 0.025168 |
| 1000 | -0.016789 | -0.026498 | -0.015567 | -0.010417 | -0.019755 |
| 10000 | -0.255935 | -0.264381 | -0.257858 | -0.262046 | -0.253137 |

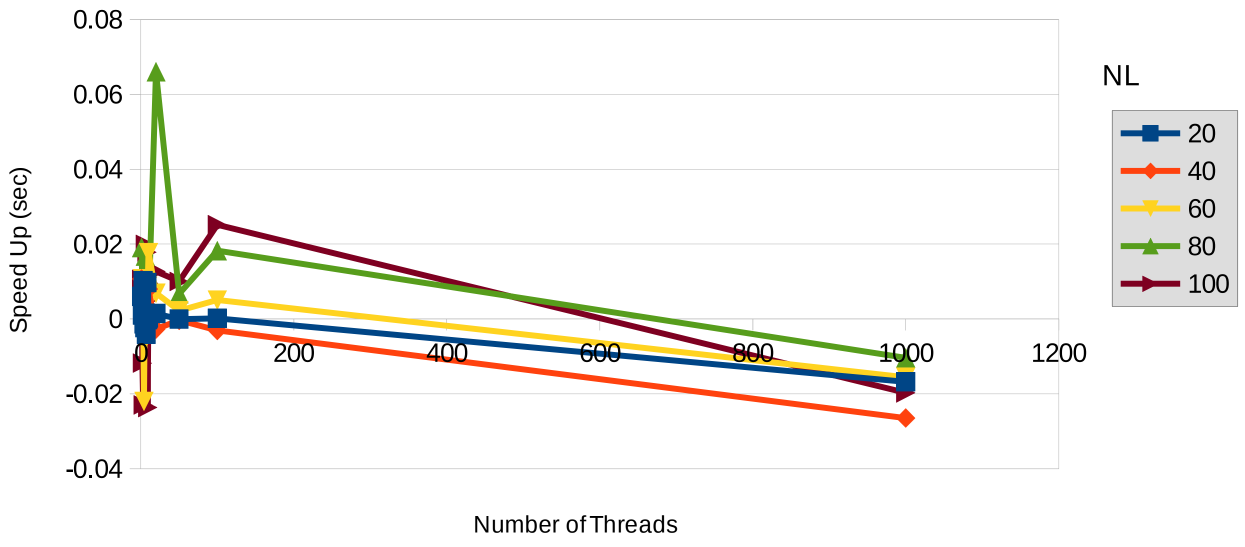
Performance at Varying Number of Threads with Increasing Degree of Basis Functions



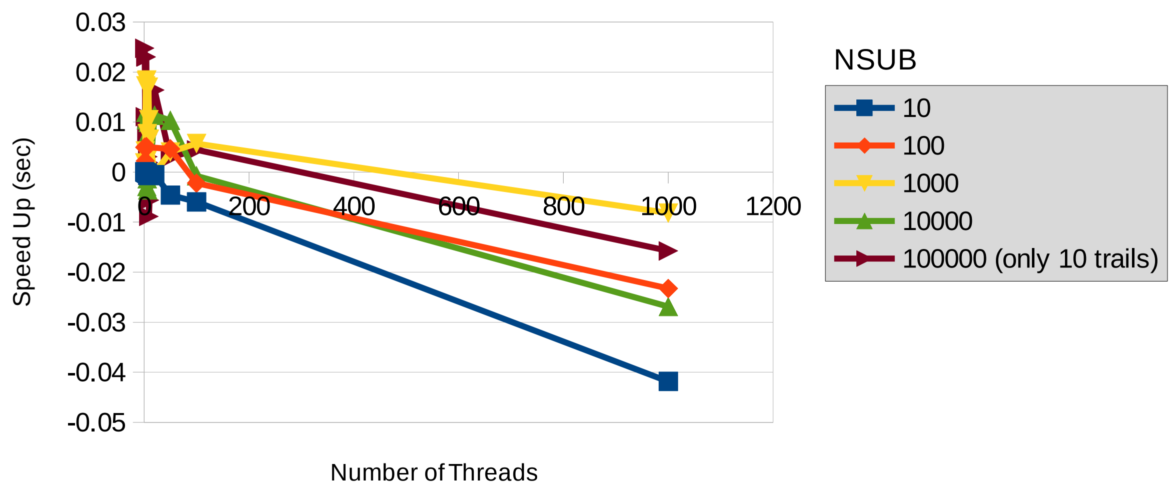
As you can see again the trend is very volatile and provides very little conclusion on the effects of base function degree. One point of note is that as the function degree is increased the performance trend becomes more volatile with more rapid changes between .
Speed up and speed down.

Next performance was tested when both thread number and workload were scaled.

Performance at Varying Number of Threads with Increasing Degree of Basis Functions



Performance at Varying Number of Threads with Increasing Subdivision

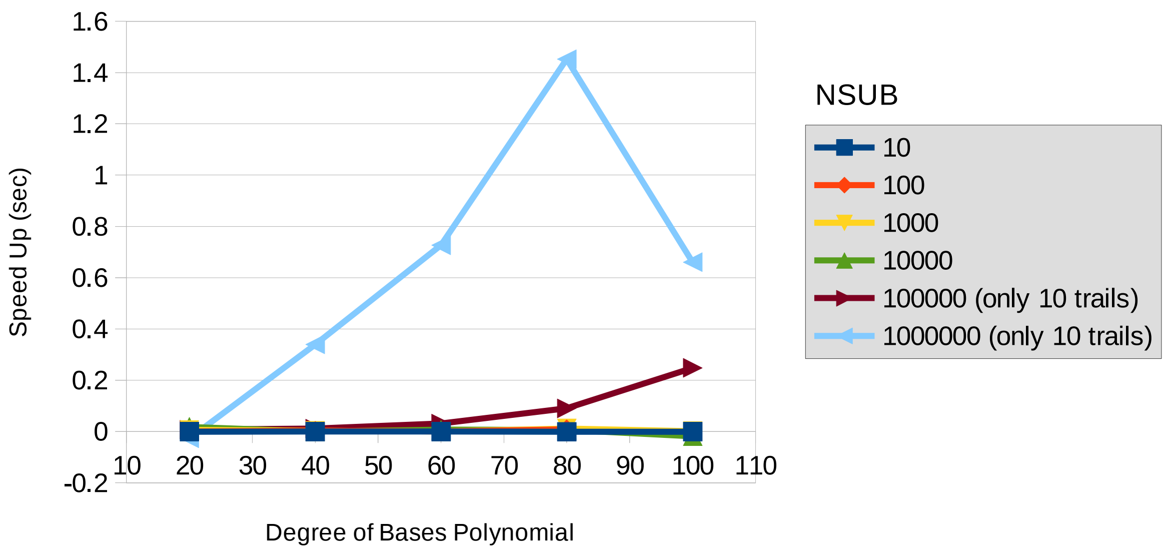


Above the effects of thread scaling can be seen. As the number of threads goes towards 10,000 the overhead produced by the parallelized execution causes a rapid decline in performance. This can be seen above both with varying levels of subdivision and degrees of base functions.

**./fem a b 4 50**

| | | | | **NSUB (a)** | | |
|---|---|---|---|---|---|---|
| **NL(b)** | **10** | **100** | **1000** | **10000** | **100000 (only 10 trails)** | **1000000 (only 10 trails)** |
| 20 | -0.00031 | -0.0002 | 0.006283 | 0.01838 | 0.007553 | -0.026495 |
| 40 | -0.000108 | 0.003304 | 0.003096 | 0.000784 | 0.01178 | 0.339631 |
| 60 | -0.000185 | -0.000196 | 0.000264 | 0.00723 | 0.031443 | 0.726371 |
| 80 | -0.000657 | 0.008974 | 0.011865 | 0.006761 | 0.090394 | 1.452302 |
| 100 | -0.000356 | X | 0.001662 | -0.018155 | 0.248205 | 0.660732 |

Scalablity of Varying Workloads



Above shows that increased performance can be seen with increases in workload. At 1,000,000 subdivisions with a 8th degree base function the program executes at over 1.4 seconds faster than the original code. However this is trivial relative to the execution time (more than 3mins). However this does show that performance of the parallelized regions scales relative with workload.