Experimenting with the schedule clause, in the parallel for directive, when using OpenMP to calculate the Pearson Correlation Coefficient

Frances Crouch

I have used the following formula to calculate the Pearson correlation coefficient (the rationale for using this formula is included in my report on the previous assignment):

$$\rho(X,Y) = \frac{\sum xy - \frac{\sum x \sum y}{n}}{\sqrt{\sum x^2 - \frac{(\sum x)^2}{n}}\sqrt{\sum y^2 - \frac{(\sum y)^2}{n}}}$$

Program structure - See Assignment2.c for code.

Serial Program

(1) Use a `for` statement to iterate through the X and Y arrays. In each iteration: assign values to the element of the array (such that $x[i] = \sin(i)$ and $y[i] = \sin(i+2)$) and calculate $\sum x$, $\sum y$, $\sum xy$, $\sum x^2$, $\sum y^2$

(2) Calculate the Pearson correlation coefficient as per the above formula

Parallel Program

In contrast to the MPI parallel program, the OpenMP parallel program is very similar to the serial code; the only difference being the use of the `parallel for` directive with the `reduction` and `schedule` clauses (to divide the iterations across several threads), instead of the standard `for` statement.

Experimenting with the schedule clause

The `schedule` clause allows the user to set how the iterations of the `parallel for` directive are divided among the threads.

For this assignment, I have experimented with four scheduling options: static, dynamic, guided and auto. The scheduling type can be set at run-time by using the `runtime` schedule type (within the C code) and setting the `OMP_SCHEDULE` environment variable within a bash file. This means we do not need to re-compile the file to change the scheduling type.

I ran the the program for each schedule type with a chunk size (number of iterations given to a thread in one call) of between 1 and 1,000,000. Each combination of schedule type and chunk size was run 10 times and I have used the average value in my analysis. All results were obtained using the University Linux computers. See Assignment2.sh for an example of the bash file used. Also note, that the array size was set to 2,000,000 in all experiments.

Results

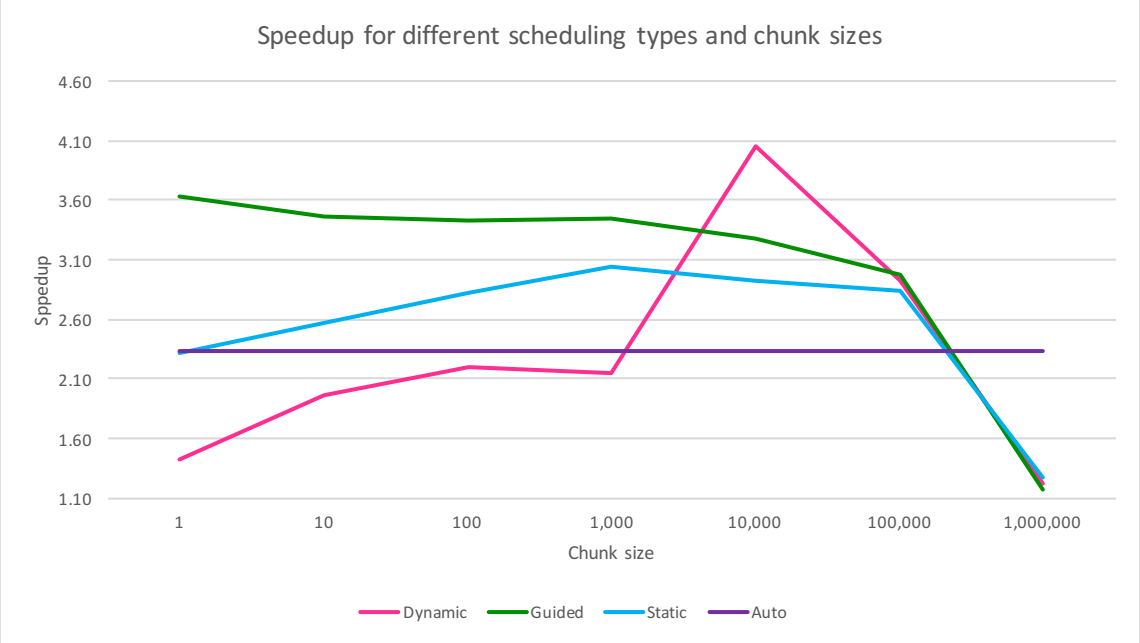
Initial Observations

The `omp_get_max_threads( )` function can be used to obtain the maximum number of threads available, and by using this function I determined that the University Linux computers have 8 threads.

I experimented with setting the number of threads to 5 and 10. In both cases the speedup was reduced compared to using 8 threads. This is an obvious results when just 5 threads are used. As the speedup was also reduced when 10 threads were used, it would appear that there is extra overhead associated with creating "pseudo-threads".

All further experiments used 8 threads.

Speedup



Note: the user does not specify the chunk size for the `auto` schedule, as this is done by the run-time system. The `auto` schedule was run just 10 times but has been represented as a straight line in the graph above to aid comparison between the schedule types.

The speedup with the `static` schedule increases with the chunk size due to an improvement in the locality of the calculations performed by each thread, and also because the grain size increase with the chunk size. Once the chunk size reaches 1,000,000 the program is forced to run with just 2 threads and hence the speedup is dramatically reduced (we see this across all the schedule types).

It is clear that the run-time system has set the `auto` schedule to be of type `static`, with a chunk size of 1.

The `guided` schedule divides the iterations among the threads during the program execution. Once a thread has completed the iterations assigned to it, it will request another chunk of iterations and with each request, the chunk size decreases. The chunk size will reduce down to the value set by the user e.g. when the chunk size is set to 1, the size of chunks given to the threads decreases down to 1. Therefore, as the chunk size (set by the user) increases, there is a tighter restriction on how the `guided` schedule can divide the iterations. These results suggest that the `guided` schedule performs best without any restrictions. It also easily out performs the `static` schedule, despite having the greatest overhead of all the schedules.

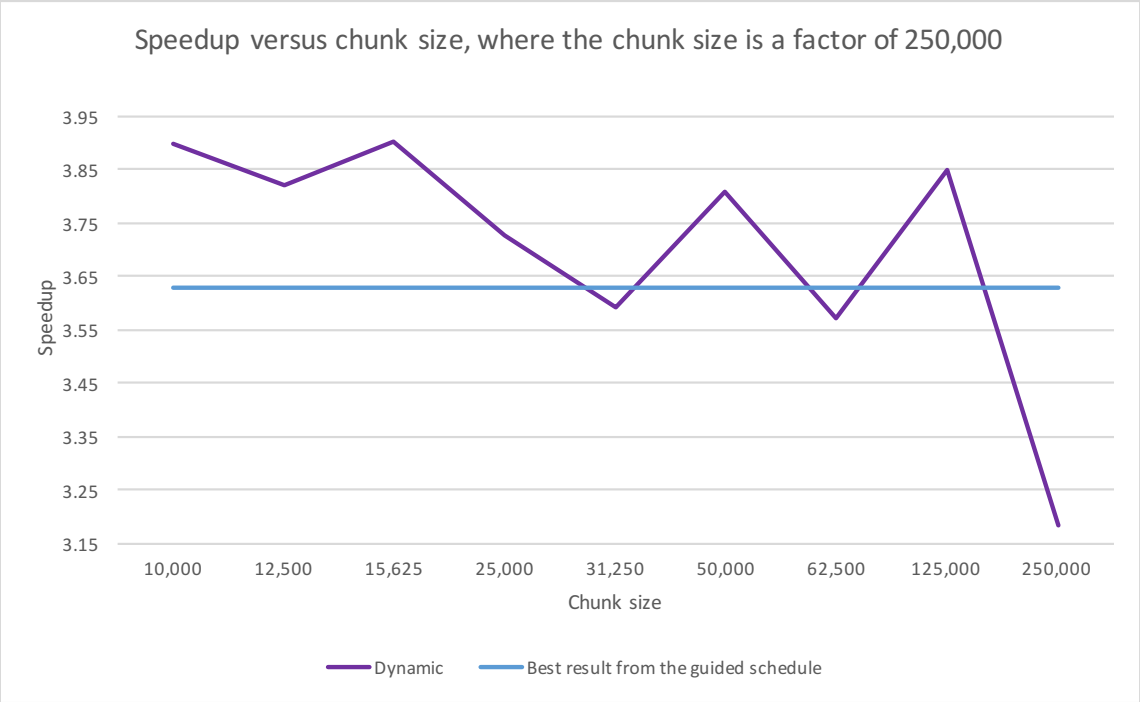
The `dynamic` schedule offers some interesting results. Initially, the speedup is very poor, just 1.43, due to the overhead associated with dynamically allocating the iterations, the poor locality of the calculations performed by each thread and a small grain size. The speedup then increases significantly to 4.05 (the highest result across all the schedule types) and then falls again to be inline with the `static` and `guided` schedules.

For these experiments, the size of the array was set to be 2,000,000 and the number of threads was 8. With a chunk size of 1,000 the iterations over the array are split into 2,000 chunks and hence each thread could be called up to 250 times (2,000/8). This obviously creates a large overhead for the system. When the chunk size is increase by a factor of 10 (to 10,000), each thread is called just 25 times, reducing the overhead and also improving the locality of the calculations and increasing the grain size. It might be intuitive to think that as the chunk size increases to 100,000, we would see a further reduction in overhead and improvement in the locality; but the speedup actually falls sharply from 4.05 to 2.92. When the chunk size is 100,000, each thread is called on average 2.5 times. Once the first 1,600,000 iterations have been completed, only 4 (of the 8 threads available) are actually used to complete the remaining iterations, and hence the resources are not being fully utilised.

This could suggest that to improve the speedup of the `dynamic` schedule, the chunk size should be a factor of 250,000 (2,000,000/8), or in the general case: $\frac{number\ of\ iterations}{number\ of\ threads}$.

Using factors of 250,000 as the chunk size with the dynamic schedule

I ran the code with the chunk size set to factors of 250,000 (starting from 10,000). The graph below shows the results:



I would have expected the speedup to increase as the chunk size increases from 10,000 to 250,000, due to improvements in the locality and increase in grain size. However, instead we see that the graph is fairly jagged (although the results are plotted against a small scale). This could be just due to fluctuations in the resources available while the code was executed, or could suggest that at certain chunk sizes the grain size is optimised.

Interestingly, it does not appear to be optimal to only call each thread once, e.g. have a chunk size of 250,000, which suggests having smaller chunk sizes gives extra flexibility, e.g. if one thread is taking longer than other threads, the the run-time system can give more iterations to the other threads.

The previous results show that the `guided` schedule can achieve a speedup of 3.63 (demonstrated as the horizontal line on the graph above). The majority of these experiments have a better speedup that this. Although, it should be noted, that these results were obtained from running the code at a completely different time to the original experiments, and hence the two may not be entirely comparable.

Conclusion

Using the maximum number of threads available, and not more, gives the optimal speedup. The `guided` schedule with a chunk size of 1 provides a good speedup with minimal investigation required, however, the best speedup was obtained by using the `dynamic` schedule and a chunk size that is a factor of $\frac{number\ of\ iterations}{number\ of\ threads}$.

OpenMP coding was a lot more straight forward compared with the previous assignment using MPI, and also achieved much a much better speedup, just 1.490 with MPI compared with 4.05 with OpenMP (note: these results are not entirely comparable, as the MPI program was only run on my personal laptop).

Further investigations could include: increasing array size to see if there is more of a difference between the scheduling types, or trying to introduce the idea of using factors as the chunk size in the `guided` schedule, as a way of combining the two best performing schedules.