Parallelization strategy

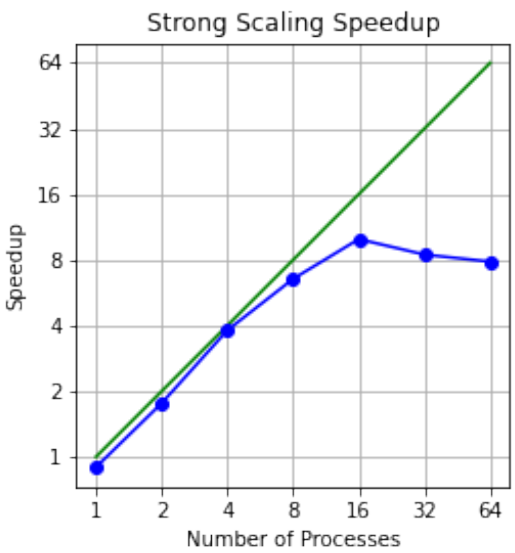
The problem of parallelization primary comes with overhead, poor memory access patterns, and data race. To address these problem, we apply **loop optimization** to improve the use of memory, specify **data environment to avoid data race**, and add **nowait** to decrease the unnecessary time spent by threads waiting at implicit barrier.

Firstly, a suitable reorganization of the computation in loop nests to exploit cache can significantly improve a program's performance. A number of loop construct transformations can help achieve this. For example, in both "init" and "step" function, we explicitly update u and v matrices in separate loops to streamline the computation. In addition, the nowait clause is used to omit the implied barrier at the end of the first and the second loop.

Another factor to take into account is the optimization of parallel regions. Indiscriminate use of multiple parallel regions may hinder the improvement of overall performance, because overheads are strongly associated with starting and terminating a parallel region. While multiple parallel region could require more time to execute, a single large parallel region offers more opportunities for using data in cache and provide a bigger context for other compiler optimizations. For example, if we choose multiple combined parallel work-sharing loops rather than single parallel region enclosing all work-sharing for loops, each parallelized loop would add to the parallel overhead and has an implied barrier that cannot be omitted. Therefore it is worthwhile to minimize the number of parallel regions.

Lastly, to optimize the performance as well as to avoid data race problem in parallel region, data environment should be specified properly. This is importance because efficient parallelization often involves minimizing the use of shared variables to reduce contention and synchronization overhead. Private variables can enhance performance by avoiding unnecessary synchronization between threads. For instance, specifying private for counter "i" and "j" and other shared variable has significantly reduced synchronization overhead, contributing to improved overall performance.

Scalability Graph



Amdahl's law

speedup= 1/(s+p/N)

Speedup is calculated by T(1)/T(N)

where **s** is the proportion of execution time spent on the serial part, **p** is the proportion of execution time spent on the part that can be parallelized, and **N** is the number of processors. According to Amdahl's law an upper bound for the speedup when running, for example, on 4 threads is given by S = 1/(0.98/4 + 0.02) =3.8