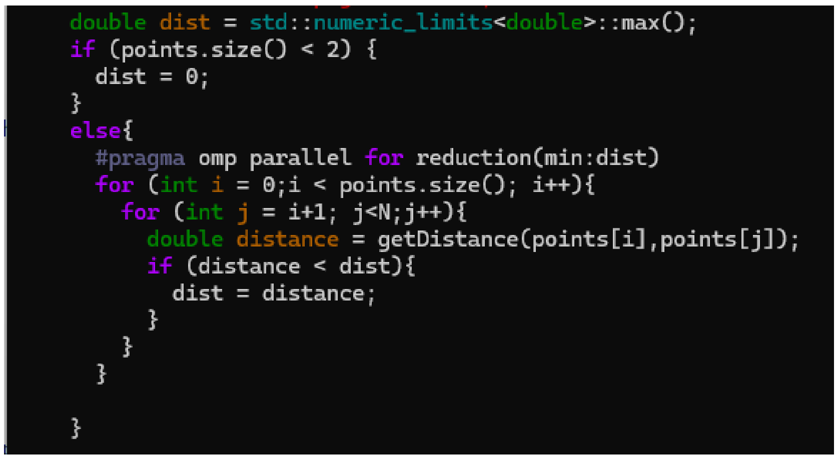
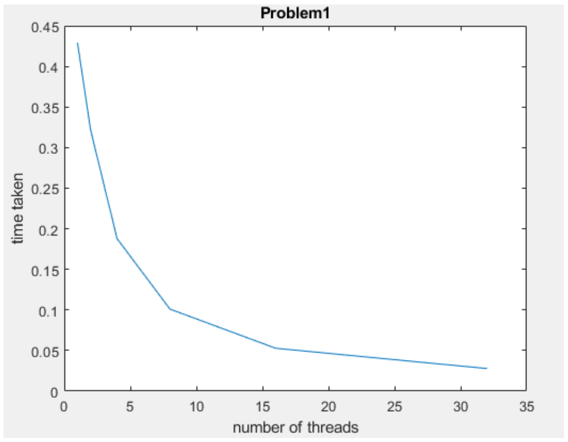
Problem1



```
  double dist = std::numeric_limits<double>::max();
  if (points.size() < 2) {
    dist = 0;
  }
  else{
    #pragma omp parallel for reduction(min:dist)
    for (int i = 0;i < points.size(); i++){
      for (int j = i+1; j<N;j++){
        double distance = getDistance(points[i],points[j]);
        if (distance < dist){
          dist = distance;
        }
      }
    }

  }
```

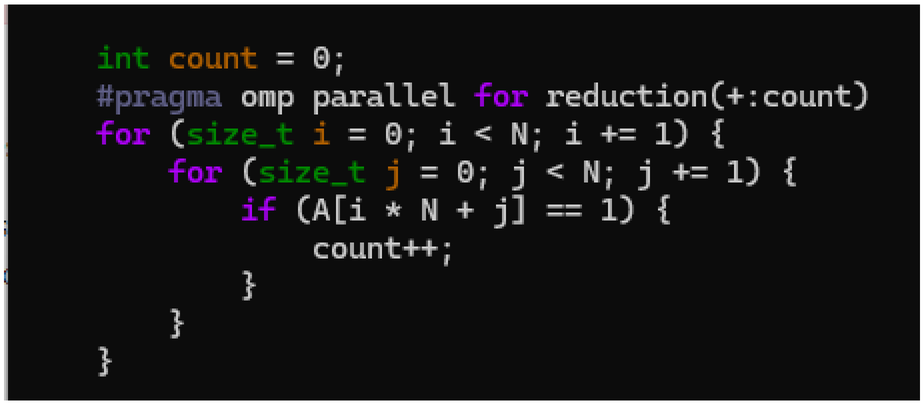
I firstly declare a global variable dist to store the minimum distance. Then I use if statement to check whether the number of points is only 1. If so, let the minimum distance be 0. To parallelize the program, I added the OpenMP pragma directives with reduction clause. The operator min:dist would automatically set dist to be private for each thread and let the final global variable dist be the minimum among all the local variables dist.



The performance on different number of threads is shown on the figure. Basically, the time taken decreases exponentially as the number of threads increases exponentially.

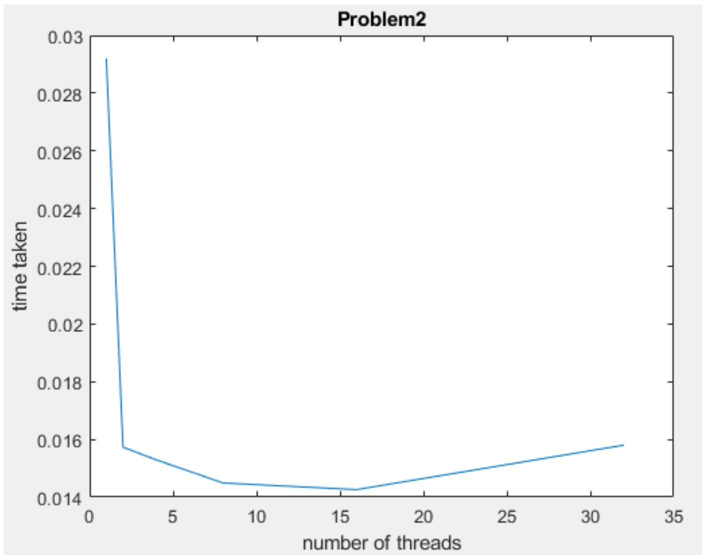
This kind of parallelism will face load imbalance. The first thread needs to calculate the minimum distance among N points, the second thread needs to calculate among N-1 points and so on. To further improve speedup, an idea is to let the first thread to compute iteration 1 and iteration N-1, let the second thread to compute the iteration 2 and iteration N-2. In all, if distributing threads in this strategy, every thread needs to compute among N points equally.

Problem2



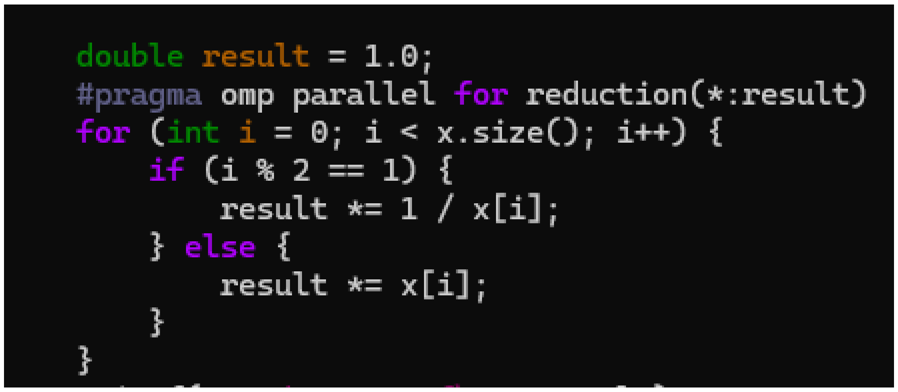
```
int count = 0;
#pragma omp parallel for reduction(+:count)
for (size_t i = 0; i < N; i += 1) {
    for (size_t j = 0; j < N; j += 1) {
        if (A[i * N + j] == 1) {
            count++;
        }
    }
}
```

To parallelize this algorithm, firstly declare count as global variable and set it to be 0 for getting cumulative summation. Then add a pragma directives with reduction clause with +:count operator. Local variable count will automatically become private to avoid reading/writing issues. Each thread will compute the total number of edges starting from the vertices sent to that thread. The final global variable count will be the summation of all local variable count. This problem is upon directed graph so there is no load imbalance.



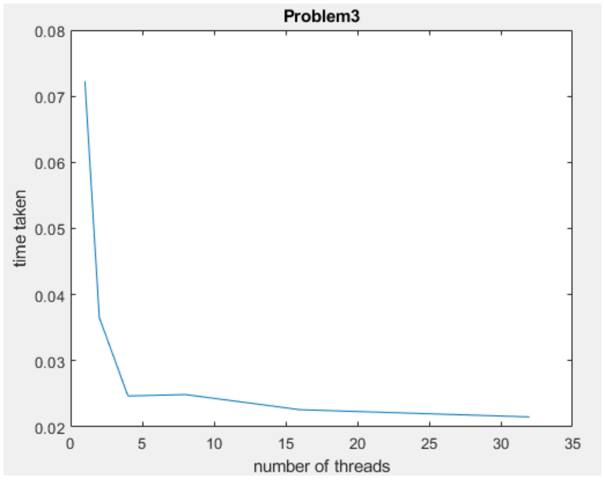
The algorithm took the shortest time while running on 16 threads, but unexpectedly took more time as the number of threads further increased. One possible reason is the overhead outweighed the gain in speedup.

Problem3



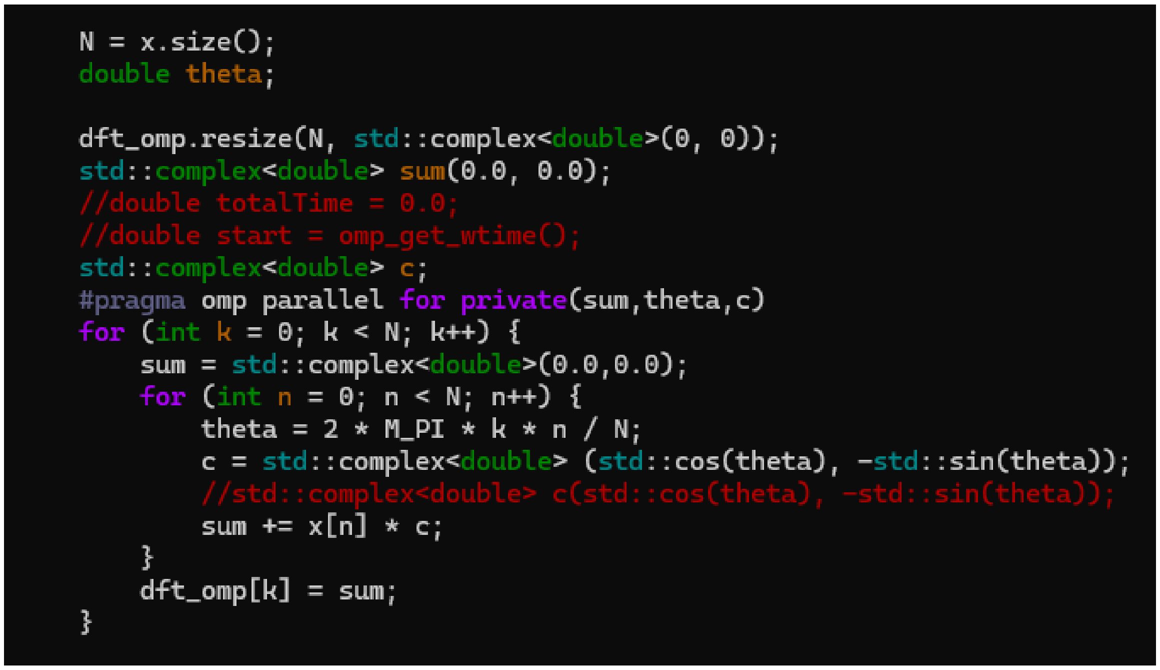
```
double result = 1.0;
#pragma omp parallel for reduction(*:result)
for (int i = 0; i < x.size(); i++) {
    if (i % 2 == 1) {
        result *= 1 / x[i];
    } else {
        result *= x[i];
    }
}
```

To parallelize this algorithm, firstly declare the global varible result and set it to be 1.0 for getting accumulative product. Then add a pragma directives with reduction clause with *:result operator. Local variable result will automatically become private to avoid reading/writing issues. Each thread will work on a certain subset of indices of the vector and calculate the cumulative product locally. The final global variable result will be the product of all local cumulative products. Whether one element should be inverted or not only depends on the oddity of indices so this algorithm can be parallelized simply.



The time taken almost decreases as the number of threads increases except it runs a little bit faster on 4 threads than on 8 threads. Also, exponential decay only happens when the number of threads increases from 1 to 2, but the later decay is pretty small. This algorithm involves only scalar multiplication and some if statement, which are all simple. It is likely the overhead overweighed the improvement on computation.

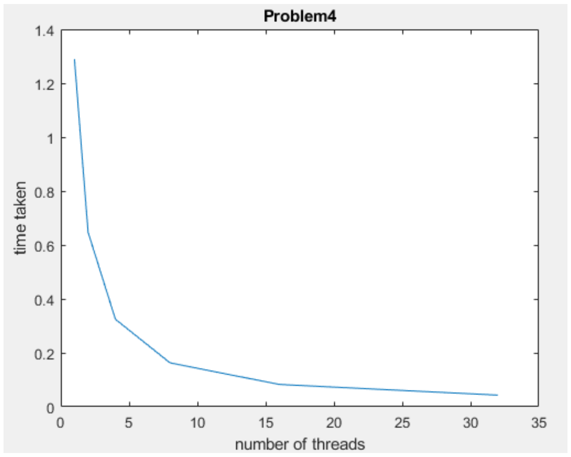
Problem4



```
N = x.size();
double theta;

dft_omp.resize(N, std::complex<double>(0, 0));
std::complex<double> sum(0.0, 0.0);
//double totalTime = 0.0;
//double start = omp_get_wtime();
std::complex<double> c;
#pragma omp parallel for private(sum,theta,c)
for (int k = 0; k < N; k++) {
    sum = std::complex<double>(0.0,0.0);
    for (int n = 0; n < N; n++) {
        theta = 2 * M_PI * k * n / N;
        c = std::complex<double> (std::cos(theta), -std::sin(theta));
        //std::complex<double> c(std::cos(theta), -std::sin(theta));
        sum += x[n] * c;
    }
    dft_omp[k] = sum;
}
```

To parallelize this algorithm, I only need to set sum, theta and c to be private to avoid reading/writing conflict and don't need to change anything from the sequential code. This is because Fourier transform on different index is independent so it can be parallelized simply. It is also fine to declare local variable sum, theta and c inside the for loop and thus no private clause is needed. However in that case, the time taken will be longer. Maybe this is because it takes more time to clean the memory and declare new variables every time.



The performance is an exponential decay in time as the number of threads increases. The computation of Fourier transforms does requires some computations, so the advantage of parallelism can be seen.