Parallelizing The Discrete Fourier Transform

Nelson Morrow

May 12, 2018

Abstract

The Fourier Transform is used in signal processing to decompose signals into their component frequencies - called the 'fundamentals' by computing a linear combination of sinusoidal functions. The Fast Fourier Transform is a method of computing the Discrete Fourier Transform that reduces the complex ity from $O(n^2)$ to $O(n \log n)$. For large N, this algorithm is significantly more efficient than DFT. Here, I implement a novel (but naive) implementation of DFT parallelized with OpenMPI and compare its scaling capabilities to Gnu Scientific Libraries FFT.

Code and Discussion

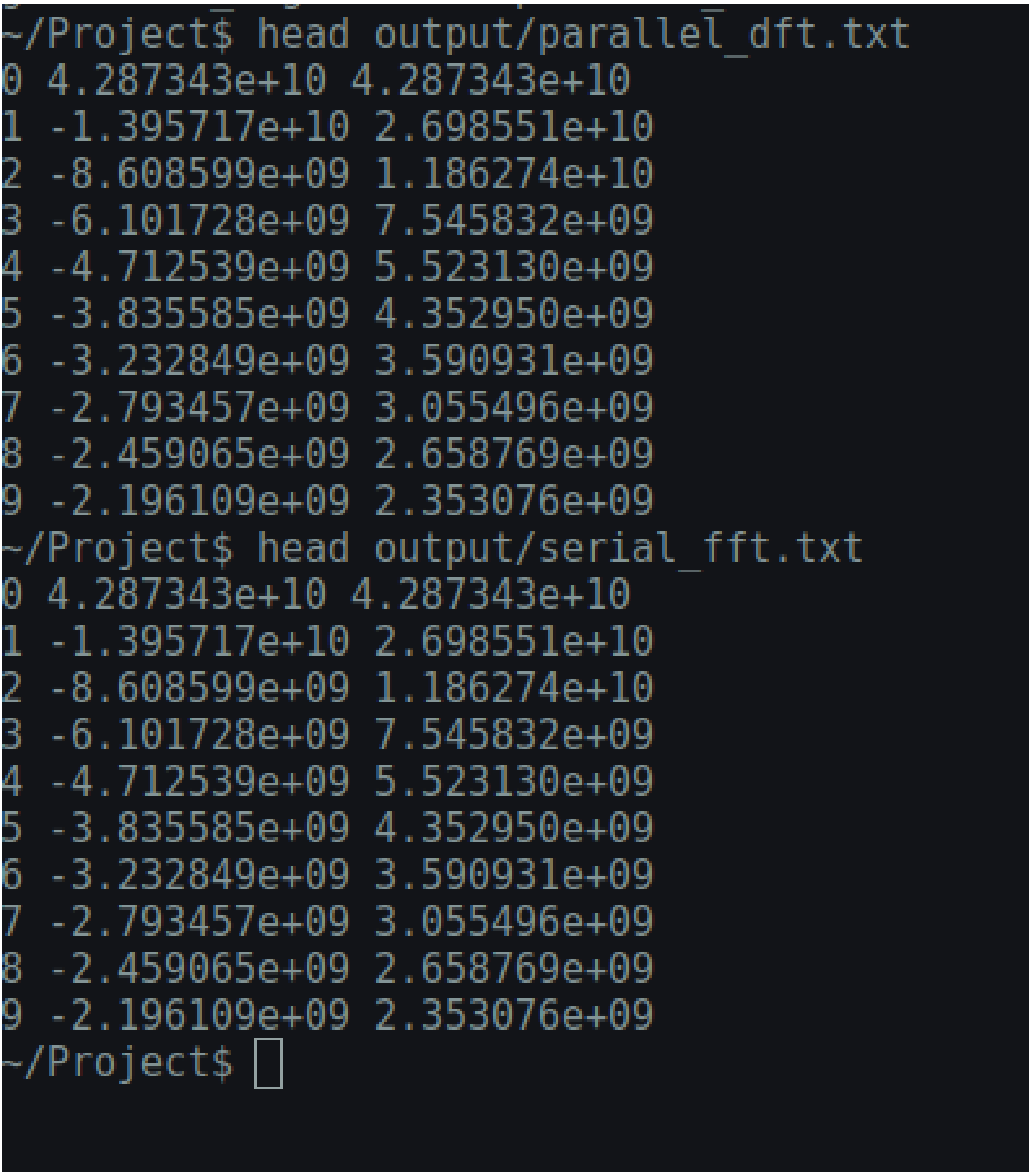
Initially, I attempted to parallelize GSL's implementation of the FFT. This failed (I am certain) because the bitreverse algorithm, which is essential to FFT, maps elements of input arrays with index k to index $n-k$ with n being the length of the signal vector. My attempts to feed GSL's FFT smaller arrays from different nodes created wholly inaccurate transformations. Since I could not do this, I implemented my own version of an FFT - a parallelized DFT. The code that I wrote to parallelize the DFT is below. Notice the call to gsl_fft_complex_radix2_forward, which I used to verify the correctness of my code.

```
static void dft_kernel(double *input, double* sumreal,
double* sumimag, int k, int min, int size) {
  double loc_real_sum = *sumreal;
  double loc_imag_sum = *sumimag;
  for (int t = 0; t < size; t++) {  // For each input element
      double angle = 2 * M_PI * (t+min) * k / N;
      loc_real_sum += REAL(input, t) * cos(angle) +
      IMAG(input, t)* sin(angle);
      loc_imag_sum += -REAL(input, t) * sin(angle) +
      IMAG(input, t)* cos(angle);
  }
  *sumreal = loc_real_sum;
  *sumimag = loc_imag_sum;
}

//In function 'main'
start = omp_get_wtime();
min =  sharded_arr_size*world_rank;
for (int k = 0; k < N; k++) {  // For each output element
  local_real_sum = 0;
  local_imag_sum = 0;
  dft_kernel(shard, &local_real_sum, &local_imag_sum,
  k, min, sharded_arr_size);
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Reduce(&local_real_sum, &global_real_sum, 1,
  MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
  MPI_Reduce(&local_imag_sum, &global_imag_sum, 1,
  MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
  if (world_rank == 0) {
    REAL(parallel_dft, k) = global_real_sum;
    IMAG(parallel_dft, k) = global_imag_sum;
  }
}
```

```
end = omp_get_wtime();
times[0]  = end-start;

gsl_fft_complex_radix2_forward (complex_polynomial, 1, N);
```

Results

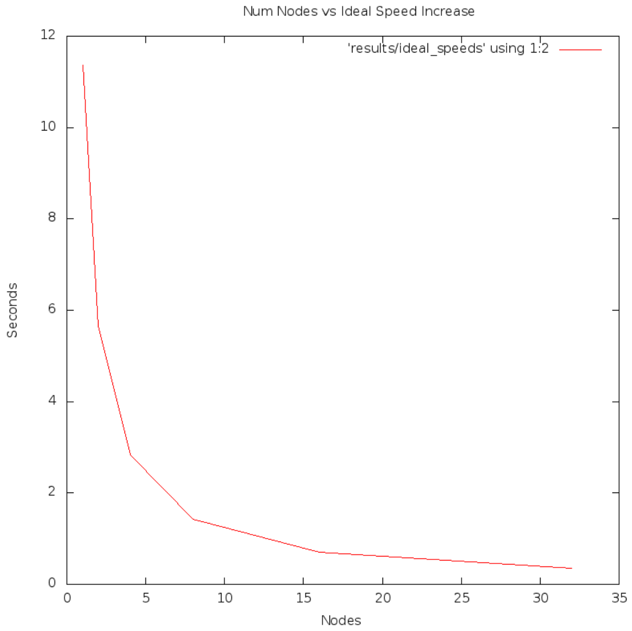
This was a novel implementation of DFT; it is completely inferior to FFT because of the need to synchronize, and at best scaling was N^2/p where p is the number of nodes. Comparing the output of the DFT to the FFT in the output/ folder shows that the results are completely identical. It works, but it is not nearly as fast as FFT. Here is output from both functions, confirming the result:

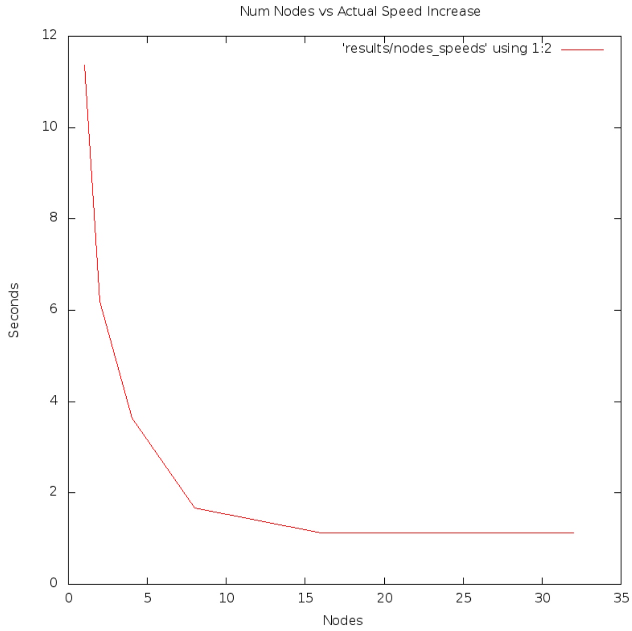


```
~/Project$ head output/parallel_dft.txt
0 4.287343e+10 4.287343e+10
1 -1.395717e+10 2.698551e+10
2 -8.608599e+09 1.186274e+10
3 -6.101728e+09 7.545832e+09
4 -4.712539e+09 5.523130e+09
5 -3.835585e+09 4.352950e+09
6 -3.232849e+09 3.590931e+09
7 -2.793457e+09 3.055496e+09
8 -2.459065e+09 2.658769e+09
9 -2.196109e+09 2.353076e+09
~/Project$ head output/serial_fft.txt
0 4.287343e+10 4.287343e+10
1 -1.395717e+10 2.698551e+10
2 -8.608599e+09 1.186274e+10
3 -6.101728e+09 7.545832e+09
4 -4.712539e+09 5.523130e+09
5 -3.835585e+09 4.352950e+09
6 -3.232849e+09 3.590931e+09
7 -2.793457e+09 3.055496e+09
8 -2.459065e+09 2.658769e+09
9 -2.196109e+09 2.353076e+09
~/Project$
```

Scaling information

Scaling was actually close to ideal, and the more nodes tasked to the assignment the better the performance. The most time consuming kernel, which I parallelized with dft_kernel, had a performance increase hindered greatly by synchronizing necessities; for example, MPI_Barrier() slowed it down greatly. Here are some graphics which make the argument that scaling wasn't too far in fact from ideal - but compared to the FFT on the same dataset each time was comparatively very slow. For example, with $p = 32$ nodes the DFT ran at 1.344 seconds and the FFT ran at 0.005987 seconds.


Num Nodes vs Ideal Speed Increase
'results/ideal_speeds' using 1:2
Seconds
Nodes
0
2
4
6
8
10
12
0
5
10
15
20
25
30
35


Num Nodes vs Actual Speed Increase
'results/nodes_speeds' using 1:2
Seconds
Nodes
0
2
4
6
8
10
12
0
5
10
15
20
25
30
35