Discrete Fourier Transform

Nikhil Agarwal
Roll No : 11012323
IIT Guwahati

Introduction

Polynomial multiplication takes $O(n^2)$ time usins simple two for loop algorithm. So, DFT algorithm is designed where two polynomials are represented by point representation method and then after multiplying them they are converted to polynomial representation again. An attempt is made to understand the complexity and efficiency of such an algorithm.

Algorithm

- First take two polynomials as input. The coefficients of the poynomial are stored in an array.
- Then, convert the polynomials into point representation. We have implemented a function recursive_fft which takes as input the polynomial and outputs it's point representation form. The function is a recursive functions which find the y values for n roots of unity.
- Once, we have the point representation we multiply the y values for similar x values.
- Lastly, we convert the point representation into polynomial representation again. For this we have implemented a function inverse_recursive_fft which takes input in the point form and outputs the polynomial form in the form of array.

Theoretical Analysis of Algorithm

- First step takes $O(n)$ time since we just input our array.
- For converting to point representation, we require $O(nlogn)$ time. Since the recursive function is called $(logn)$ times and each time we have to deal with n amount of data so in total it takes $O(nlogn)$ time.
- For multiplying we require $O(n)$ time.
- We again require $O(nlogn)$ to convert to polynomial representation by similar logic as described in 2.

Therefore, in total we require $O(nlogn)$ time. So if we proceed in parallel we can finish it in $O(n)$ time.

Table 1: **Program Analysis**

n	Sequential Algorithm			DFT based sequential Algorithm			Multithreaded DFT Algorithm		
	real	user	sys	real	user	sys	real	user	sys
10	0.701	0.002	0.002	0.711	0.002	0.002	0.761	0.003	0.003
100	0.864	0.002	0.002	0.937	0.004	0.003	1.125	0.005	0.003
500	0.950	0.007	0.002	0.964	0.008	0.003	0.970	0.010	0.004
1000	1.248	0.015	0.002	1.185	0.013	0.005	1.003	0.014	0.005
2000	1.552	0.036	0.002	1.379	0.021	0.006	1.373	0.023	0.007
5000	1.454	0.200	0.003	1.483	0.066	0.019	1.245	0.069	0.019
10000	2.599	0.766	0.003	1.948	0.124	0.036	1.400	0.129	0.036
20000	4.382	3.031	0.006	1.712	0.248	0.069	1.541	0.253	0.068
50000	20.723	18.959	0.088	2.769	0.498	0.136	2.246	0.523	0.134
100000	-	-	-	3.144	1.015	0.264	2.754	1.050	0.262

Experimental Analysis

We know thar User+Sys tells us how much actual CPU time our process used. Note that this is across all CPUs, so if the process has multiple threads it could potentially exceed the wall clock time reported by Real. So,

- For smaller values of n CPU time is almost same for both sequential as well as DFT based sequential algorithm but CPU time is more for mulithreaded algorithm because of the overhead due to more number of threads.
- As the value of n increases CPU time for DFT sequential is quite small compared to general sequential algorithm but it is almost similar to that of mulithreaded one. The reason being the multithreaded user time is the sum of time taken across all the processing units. So, we compare real time in such case and we can clearly see the real time for mutithreaded is quite less that of DFT based sequential.

Conclusion

DFT algorithm is quite efficient compared to the polynomial one. Further using multithreaded programming on DFT algorithm makes it more efficient.

Parallel DFA Algorithm

Nikhil Agarwal
Roll No : 11012323
IIT Guwahati

Introduction

A Deterministic Finite Automaton (DFA) is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation of the automaton for each input string. We present a parallel implementation of such finite machine and compare it's performance with that of serial implementation.

Algorithm

- Firstly, we store the transition table in the form of 2D Array and for each input string we decide whether the DFA accepts/rejects it.
- For serial implementation: $state$ -> $\delta(state, str[i])$ where i varies from 0 to length of string. If we reach the final state, then we accept the string otherwise we reject it.
- For parallel implementation: we partition the string into 4 parts and then for each partition except the first partition, we make transition from each possible state and store the final state. Each state behaves as a separate entity. Hence, we can parallelize this aspect. Finally we track the final states for each partition and output whether the string is accepted or not.

Theoretical Analysis of Algorithm

Time Complexity of Serial Algorithm is $O(n)$. Time Complexity of Parallel Algorithm is $O(n/p+p)$ where n = length of string, p = number of threads running in parallel. Space complexity is $O(Q*M)$ where Q is the number of states and M is the maximum number of symbols a particular state can process.

Program Analysis

n	**Sequential DFA Algorithm**			**Parallelized DFA Algorithm**		
	real	user	sys	real	user	sys
1000	0.004	0.002	0.002	0.006	0.004	0.003
10000	0.004	0.002	0.002	0.014	0.015	0.003
100000	0.006	0.004	0.002	0.049	0.051	0.003
1000000	0.024	0.022	0.002	0.353	0.341	0.006
10000000	0.166	0.163	0.003	3.296	3.275	0.117

Experimental Analysis

As the value of n increases, the performance of parallel DFA Algorithm goes on decreasing. For very large values of n($> 10^8$), parallel DFA takes a large amount of time compared to the serial DFA.

Conclusion

Serial Implementation of DFA is better than that of parallel implementation without using any external hardware. Instead of openmp, if we use Cuda, parallel DFA may perform better compared to serial DFA for our implementation.

Canon Algorithm

Nikhil Agarwal
Roll No : 11012323
IIT Guwahati

Introduction

Canon Algorithm is a distributed algorithm for matrix multiplication. The main advantage of the algorithm is that its storage requirements remain constant and are independent of number of processors.

Algorithm

- Firstly, we partition two matrix A and B into p square blocks of $n/\sqrt(p)$ x $n/\sqrt(p)$ each. Then we use cartesian topology to set up the process grid in MPICH. Process P_{ij} initially stores A_{ij} and B_{ij} and computes block C_{ij} of the result matrix.
- For matrix A, we row shift by i(row no.) blocks and for matrix B, we column shift by j(col no.) blocks initially. Then for $\sqrt(p) - 1$ times , we shift both A and B by 1 blocks in row and column respectively.
- Instead of using periodic function of mpich for shifting, we have implemented left shift and right shift functions. For shifting, we have calculated the rank of the process, we want to send our output to, and use MPI_SEND and MPI_RECV functions in row and columns respectively.
- During each shifting, as soon as process have both blocks A and B, temporary matrix C is calculated by multiplying blocks of A and B and then we add to the previously obtained C. Repeating in this manner, we obtain our resultant matrix C.

Theoretical Analysis of Algorithm

- Time Complexity of Serial Algorithm is $O(n^3)$ for matrix multiplication.
- If we use all to all broadcast: $\sqrt{p}$ rows of all to all broadcast, each is among a group of $\sqrt{p}$ processes. A message size is n^2/p. Therefore, the communication time is $t_s log\sqrt{p} + t_w \frac{n^2}{p}(\sqrt{p} - 1)$. Same time is taken for $\sqrt{p}$ columns of all to all broadcasts. Time taken for computation is $\sqrt{p}$ x $(\frac{n}{\sqrt{p}})^3 = \frac{n^3}{p}$. Total parallel time taken is: $\frac{n^3}{p} + 2(t_s log\sqrt{p} + t_w \frac{n^2}{p}(\sqrt{p} - 1))$
- If we use Send and Recv Algorithm: Maximum distance over which the blocks shifts is $\sqrt{p} - 1$. The circular shift operation in row and column directions take time $2(t_s + \frac{t_w n^2}{p})$. Each of $\sqrt{p}$ single step shifts in the compute and shift phase takes time: $t_s + \frac{t_w n^2}{p}$. So total parallel time taken is $\frac{n^3}{p} + 2\sqrt{p}(t_s + \frac{t_w n^2}{p}) + 2(t_s + \frac{t_w n^2}{p})$

Table 2: **Program Analysis for Canon Algorithm**

n	1 Processor			4 Processors			16 processors			Serial Transpose Implementation		
	real	user	sys	real	user	sys	real	user	sys	real	user	sys
100	0.064	0.040	0.025	1.081	0.118	0.066	1.230	0.446	0.316	0.017	0.015	0.002
500	1.049	1.026	0.024	1.519	1.807	0.113	1.774	2.257	0.642	0.863	0.858	0.005
1000	10.304	9.615	0.084	4.954	14.719	0.901	5.816	18.418	0.559	7.013	6.949	0.059
1200	17.890	17.746	0.135	8.662	27.336	0.501	9.960	34.180	0.678	12.189	12.059	0.115

Experimental Analysis

We know thar User+Sys tells us how much actual CPU time our process used. Note that this is across all CPUs, so if the process has multiple threads it could potentially exceed the wall clock time reported by Real. So,

- For smaller values of n(< 500), we can see that time taken by Serial Transpose Implementation < Canon 1 processor < Canon 4 processor < Canon 16 processor. This means that for lower values of n, serial transpose implementation is the best.
- For large values of n(> 500), we can see that time taken by Canon 4 processor < Canon 16 processor < Serial Transpose Implementation < Canon 1 processor. This means that Canon Algorithm with proper utilisation of processors is the best for larger values of n.

Conclusion

Canon Algorithm is quite efficient compared to the normal matrix multiplication. Further using multithreaded programming in an efficient way i.e with proper utilisation of processors is more efficient.
