Parallel Computing Lab

<u>Polynomial Multiplication using FFT</u>

By: Dhruv Kohli
Roll No.:120123054
Dept.:M&C

Introduction

Let A = [a0 a1 a2 ... a(n-1)], B = [b0 b1 b2 ... b(n-1)] be the vectors of coefficients of two n-degree-bound polynomials A(x) and B(x) respectively.

Assumption: n is a power of 2

Problem Statement: To compute C = [c0 c1 c2 ... c(2n-2)] which is a vector of coeffecients of an (2n-1)-degree-bound polynomial C(x) which is computed as:

C(x) = A(x)*B(x)

Brute-force polynomial multiplication algorithm has a time complexity of $O(n^2)$.

Another algorithm which makes use of point-wise representation of a polynomial (i.e. every n-degree polynomial can be uniquely represented by (n+1) distinct points {(xi, yi)|i=1,2...n}), which we can derive in O(lg(n)) time by interpolating the given polynomial on 2nth complex roots** of unity using sequential FFT, and then computes C(x) on those 2nth complex roots of unity using C(xi)=A(xi)*B(yi) in O(n) time and finally extrapolates the computed values of C using inverse FFT in O(log(n)) time again to get back the coeffecients of C and hence leading to a final time compexity of O(nlog(n)) for this algorithm.

Through this assignment, we demonstrate that, given infinite number of processors, we can achieve a time complexity of O(lg(n)) and hence a parallelism of O(1/n) using the so called parallel FFT algorithm for interpolation and extrapolation step. Note that the pointwise multiplication step takes constant time O(1) given infinite number of processors.

**We chose 2nth complex roots of unity instead of n because we require 2n-1 points to represent C pointwise since it is a polynomial of degree 2n-2.

Parallel FFT Algorithm

```
Recursive_Parallel_FFT(A, IN_OR_EX) {
      n=A.length;
      if(n==1)
            return A;
      w_n=exp(2*IN_OR_EX*pi*i/n);
      w=1;
      A_e <- complex_empty_vector;
      A_o <- complex_empty_vector;
      PARALLEL_THREAD_ARRAY_T
      for(int i = 0; i < n/2; ++i) {
                  A_e.push_back(A[2*i]);
                  A_o.push_back(A[2*i+1]);
      }
      SYNCH_BARRIER_THREAD_ARRAY_T

      PARALLEL_THTREAD_T1 {
            Y_e <- Recursive_Parallel_FFT(A_e)
      }
      PARALLEL_THREAD_T2 {
            Y_o <- Recursive_Parallel_FFT(A_o)
      }
      SYNCH_BARRIER_T1
      SYNCH_BARRIER_T2

      Y <- complex_vector(size=n);
      PARALLEL_THREAD_ARRAY_T
      for(int i = 0; i < n/2; ++i) {
                  Y[k] <- Y_e[k] + w*Y_o[k];
                  Y[k+n/2] <- Y_r[k] - w*Y_o[k];
                  w *= w_n
      }
      SYNCH_BARRIER_THREAD_ARRAY_T
      return Y
}
```

Proposed Algorithm

Fast Polynomial Multiplication:
Let A_in = [a0 a1 a2 ... a(n-1)], B_in = [b0 b1 b2 ... b(m-1)] and let A and B are complex vectors of same elements as A_in and B_in but with appropriate number of zeros appended so as to make the size of both A and B equal to nearUpperPowerOf2(degree(C) + 1) where degree(C) = m+n-2.

```
Fast_Poly_Multi(A,B) {
	PARALLEL_THREAD_T1 {
		A_complex <- Recursive_Parallel_FFT(A, +1);
	}
	PARALLEL_THREAD_T2 {
		B_complex <- Recursive_Parallel_FFT(B, +1)'
	}
	SYNCH_BARRIER_T1
	SYNCH_BARRIER_T2

	n =  A_complex.length
	C_complex <- empty complex vector

	PARALLEL_THREAD_ARRAY_T
	for(int i = 0; i < n; ++i) {
			C_complex.push_back(A_complex[i]*B_complex[i]);
	}
	SYNCH_BARRIER_THREAD_ARRAY_T

	C <- Recursive_Parallel_FFT(A, -1);

	return C.real;    //ehere C.real[i] = C[i].real
}
```

Theoretical Analysis

CASE 1: #Proc == 1
Proposed Algo is equivalent to sequential FFT. Hence, T(1) = O(nlg(n))

Case 2: #Proc == infinite
Blue colored region in proposed algorithm takes O(lg(n)) time.
REASON:
Each call to Recursive_Parallel_FFT with an 'n' sized vector results in the formation of a tree of height lg(n).
Argument: All Vertices at same level can be processed simultaneously (since we have infinite # of processors). Therefore, total time complexity is of order of height of the tree i.e. O(lg(n)).
Orange Colored Region in proposed algorithm takes constant time with infinite # of processors (launch one thread per computation and hence perform the net computation simultaneously)
Hence, T(infinite) = O(lg(n))

Therefore Parallelism = T(infinite)/T(1) = O(1/n)

Case 3: #proc = p
The blue colored region will take time of order O(nlg(n)/p).
Argument: Assuming that each processor has equal share of computation and since the sequential time complexity was O(nlg(n)), the net time complexity of blue colored region with p processor will be O(nlg(n)/p).

The orange colored region will take time of order O(n/p).

Hence, T(p) = O(nlg(n)/p).

openMP implementation

*par_dft is equivalent to Fast_Poly_Multi

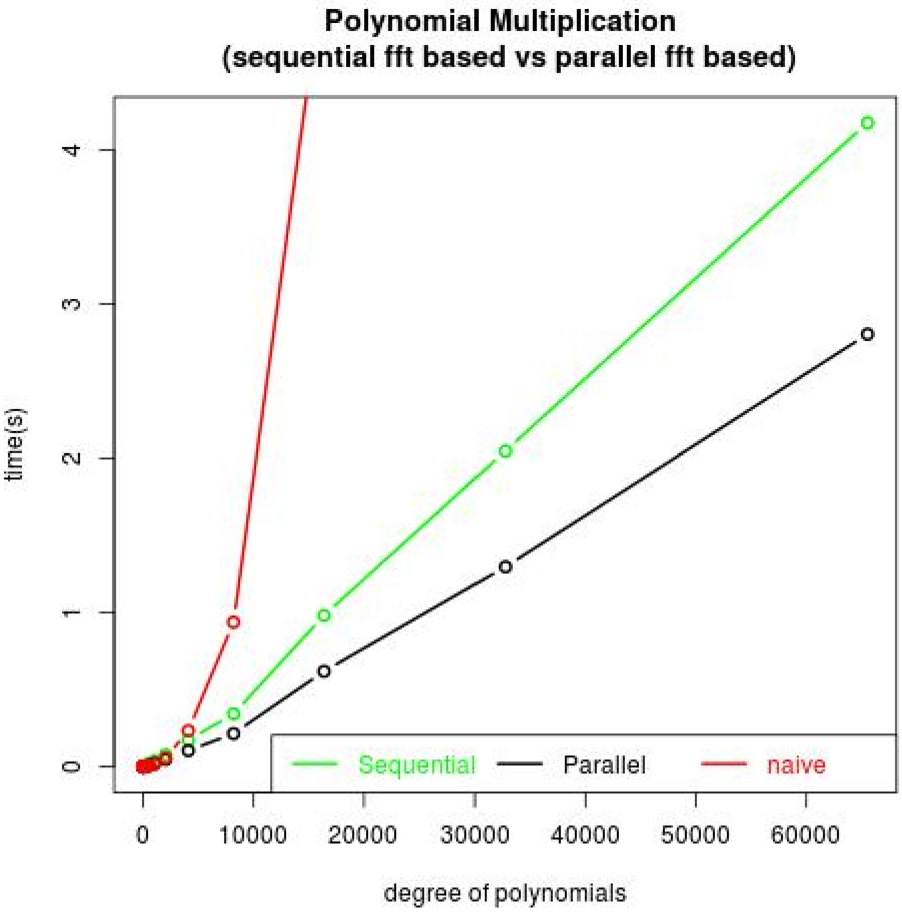
```
std::vector<value_type> par_dft(std::vector<value_type> A, std::vector<value_type> B) {
    std::complex<value_type> zero (0,0);
    int degC = A.size() + B.size() - 1;
    int npo2_degC = nearPo2(degC);

    std::vector< std::complex<value_type> > A_c(npo2_degC, zero), B_c(npo2_degC, zero),
    A_ex_c(npo2_degC, zero), B_ex_c(npo2_degC, zero), C_c(npo2_degC, zero),
    C_in_c(npo2_degC, zero);


    #pragma omp parallel for
    for(int i = 0; i < (A.size()); ++i) {
        addElem(&A_c[i], A[i]);
    }
    #pragma omp parallel for
    for(int i = 0; i < (B.size()); ++i) {
        addElem(&B_c[i], B[i]);
    }
    #pragma omp parallel num_threads(2)
    {
        int i = omp_get_thread_num();

        if (i == 0){
            ex_o_in_polate_parallel(&A_c, EXTRAPOLATE, &A_ex_c);
        }
        if (i == 1 || omp_get_num_threads() != 2){
            ex_o_in_polate_parallel(&B_c, EXTRAPOLATE, &B_ex_c);
        }
    }

    //multiply pointwise

    #pragma omp parallel for
    for(int i = 0; i < (A_ex_c.size()); ++i) {
        multiply_kernel(A_ex_c[i], B_ex_c[i], &C_c[i]);
    }

    ex_o_in_polate_parallel(&C_c, INTERPOLATE, &C_in_c);

    std::vector<value_type> res(degC, 0);

    #pragma omp parallel for
    for(int i = 0; i < (degC); ++i) {
        addElem_v(&res[i], C_in_c[i].real()/(1.0*npo2_degC));
    }

    return res;
}
```

*ex_o_in_polate_parallel is equivalent to Recursive_Parallel_FFT

```
void ex_o_in_polate_parallel(std::vector< std::complex<value_type> > *A,
                             int TYPE,
                             std::vector< std::complex<value_type> > *A_res) {
    if(A->size() == 1) {
        (*A_res)[0] += (*A)[0];
        return;
    }
    unsigned long N = A->size();        //will always be a power of 2
    std::complex<value_type> w_N(std::cos(TYPE*2.0*M_PI/N), std::sin(TYPE*2.0*M_PI/N));
    std::complex<value_type> w(1.0, 0);
    std::complex<value_type> zero(0,0);
    std::vector< std::complex<value_type> > A_o(N/2, zero), A_e(N/2, zero);

    int cntO = 0, cntE = 0;
    if(omp_get_thread_num() > 1) {
        #pragma omp parallel for
        for(int i = 0; i < N/2; ++i) {
            addElem_c(&A_e[cntE++], (*A)[i*2]);
            addElem_c(&A_o[cntO++], (*A)[i*2+1]);
        }
    } else {
        for(int i = 0; i < N/2; ++i) {
            addElem_c(&A_e[cntE++], (*A)[i*2]);
            addElem_c(&A_o[cntO++], (*A)[i*2+1]);
        }
    }

    std::vector< std::complex<value_type> > A_o_c(A_o.size(), zero), A_e_c(A_e.size(), zero);

    #pragma omp parallel num_threads(2)
    {
        int i = omp_get_thread_num();

        if (i == 0){
            ex_o_in_polate_parallel(&A_e, TYPE, &A_e_c);
        }
        if (i == 1 || omp_get_num_threads() != 2){
            ex_o_in_polate_parallel(&A_o, TYPE, &A_o_c);
        }
    }

    if(omp_get_num_threads() > 1) {
        #pragma omp parallel for
        for(int i = 0; i < N/2; ++i) {
            addElem_c(&((*A_res)[i]), A_e_c[i]+w*A_o_c[i]);
            addElem_c(&((*A_res)[i+N/2]), A_e_c[i]-w*A_o_c[i]);
            w = w*w_N;
        }
    } else {
        for(int i = 0; i < N/2; ++i) {
            addElem_c(&((*A_res)[i]), A_e_c[i]+w*A_o_c[i]);
            addElem_c(&((*A_res)[i+N/2]), A_e_c[i]-w*A_o_c[i]);
            w = w*w_N;
        }
    }
}
```

Experimental Analysis and Conclusion

Following graph represents the experimental analysis with # of processors equal to 8.



We conclude that the fast polynomial multiplication algorithm is approximately 1.5 times faster (for degree of polynomial of order ~ 2^20) and 2 times faster (for degree of polynomial of order ~2^16) than the sequential one in practice on an 8 core machine and we are affirmative that the practical speedup will be much more higher on GPUs with thousand of processors.

NOTE: A C++11 thread based implementation has also been done but probably we are making some conceptual mistakes preventing any speed up even in comparison with sequential dft code.