

## Polynomial Multiplication Using the FFT

This assignment is due at class time Thursday, 29 November 2001.

This assignment is to make sure you know how to use an FFT implemented in parallel. You are to write a program to do polynomial multiplication by means of the FFT.

Input to your program will come from a file containing

1. degree  $n$  of  $f(x)$ ;
2. the  $n + 1$  coefficients of  $f(x)$ ;
3. degree  $m$  of  $g(x)$ ;
4. the  $m + 1$  coefficients of  $g(x)$ ;
5. the number  $k$  of check values;
6. an integer modulus  $p$  for checking with;
7. the  $k$  integer check values themselves.

The output of your program should be the coefficients of

$$h(x) = f(x) \cdot g(x).$$

You will need to round up the larger of  $n$  and  $m$  to the next higher power of 2, then double that, and use that as the size of the FFT. For example, if  $n = 18$  and  $m = 24$ , then round up the larger number 25 to 32, double that to get 64, and do a 64-point FFT using the 64-th roots of unity.

You will have to pad the coefficient arrays of both  $f(x)$  and  $g(x)$  with zeros. All this is done so that the product  $h(x)$  has fewer coefficients than the number of points in the FFT and you won't get errors.

Your program should run on polynomials of degree from 7 to 511. Your program should run on any variable number of processors, with 4, 8, and 16 processors being the target. Parallelize in the obvious way. If you need, for example, a  $2^9 = 512$ -point FFT done on 8 processors, then you'll need to do 8 64-bit FFTs in parallel. Think of this as if of three recursive splits of the FFT into 2, 4, and 8 sub-FFTs (on the high three bits of the 9 bits of the FFT), and then do the FFT for the low six bits on each individual processor.

The coefficients of your polynomials will be 32-bit integers. Keep the input coefficients of  $f$  and of  $g$  smaller than 10 in absolute value so that there won't be any possibility of arithmetic overflow. (With 1024 coefficients, you'll be adding up 1024 things that are only as big as 100, so everything will be fine.)

Now, think of how to test this. If this were a million point FFT, how could you do this? That's where the check values come in. Your program should be prepared to read in "some reasonable number" (less than 100, say) of check values. You can evaluate  $f(x)$  and  $g(x)$  at each of these values  $x$ , multiply the values together, and check this against your putative  $h(x)$  evaluates. But if this really were done on a degree-512 polynomial, then evaluating at something even as small as 10 would result in a number like  $10^{512}$  added to a number as small as 10. Not good.

So long as the coefficients of the original polynomial are integers, you can do the checking by using modular arithmetic. You will have read in

the modulus (a prime number less than 1000, say). You will read in some number of check values  $x$ . Do all the checking of  $f(x)$ ,  $g(x)$ , and  $h(x)$  using the check values and doing the arithmetic modulo  $p$ . What's the chance that the  $h(x)$  will pass all your tests just by chance? That's like  $(1/p)^k$ , assuming randomness and independence. That number can be made as small as you want.

Note that no matter how careful you are, you're going to be taking "integer" coefficients, munging them with complex floating point numbers, and then trying to use them as integers. Invariably you're going to be getting values back that are an integer minus some epsilon, and if you convert to integer naively you'll wind up truncating the value down. To make sure that you get the right value, you'll need to convert the computed floating point value to the **nearest** integer within some tolerance. There are ways to determine what that tolerance ought to be, but you can put this in as a defined constant (maybe 0.0001?) and test against that value so you can change the tolerance if necessary.

Note that you'll have to do this as complex arithmetic, so you'll be wise to implement an add, a subtract, and a multiply function for complex numbers. This isn't hard:

$$(x_1 + y_1\sqrt{-1}) + (x_2 + y_2\sqrt{-1}) = (x_1 + x_2) + (y_1 + y_2)\sqrt{-1}$$

similarly for subtraction, and

$$(x_1 + y_1\sqrt{-1})(x_2 + y_2\sqrt{-1}) = (x_1x_2 - y_1y_2) + (x_2y_1 + x_1y_2)\sqrt{-1}$$

so complex arithmetic can be implemented just by keeping the real and imaginary parts as elements in a 2-long array. You won't need to divide by a complex number, and dividing by a real number is easy:

$$\frac{1}{x_1}(x_2 + y_2\sqrt{-1}) = \frac{x_2}{x_1} + \frac{y_2}{x_1}\sqrt{-1}.$$