Investigation of different methods of fast polynomial evaluation

Gavin S. Reynolds

September 2010

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2010

Abstract

This work summarises an investigation into polynomial evaluation in computer codes. A polynomial evaluation library has been written in Fortran containing four polynomial evaluation methods named: Brute Force, Brute Force - Optimised, Horner's form and Estrin's method. In this project, two versions of the parallelisable Estrin's method were written, using OpenMP and Message Passing Interface (MPI).

This report describes the background, the theory and the development of these polynomial evaluation methods. Although this project deals primarily with uni-variate polynomials, a significant attempt was made to extend the project to multi-variate polynomials. Unfortunately, due to time constraints and a number of substantial problems encountered, the work has been limited to uni-variate polynomials for the most part.

The methods contained within PolyEval were evaluated in terms of both performance and precision across a number of test cases; this report presents these results. The work was tested on a Sun Fire X4600 based at EPCC named `ness.epcc.ed.ac.uk`. Additional validation tests were conducted on a Cray XT6 system (Phase 2b of HECToR, the UK national supercomputer).

This report finds that in terms of both performance and precision, the serial Horner's form is the optimum polynomial evaluation method.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I would like to thank my supervisor Adrian Jackson at EPCC for his support and direction throughout the project. I am also appreciative of the support of my supervisors at the Department of Civil Engineering at the University of Glasgow: Professor Chris Pearce and Dr Lukasz Kaczmarczyk.

I would like also to extend my sincere thanks to all those who provided support, encouragement and understanding throughout this year, including Dr Judy Hardy (Director of Studies for the MSc programme), family and friends.

This thesis is typeset using open source software: the LyX document processor and the LaTeX 2_ε document markup language for the TeX typesetting system.

Chapter 1

Introduction and Background

1.1 Project Basis

The main aim of this project is to investigate different methods of evaluating polynomials. The motivation for the project is to improve the evaluation of polynomials in a code called YaFFEMS - a parallel implementation of the Finite Element Method, developed by the Department of Civil Engineering, at the University of Glasgow.

At higher polynomial orders, there is a loss of precision and the current implementation is perceived to be inflexible by its authors. The creators of YaFFEMS believe that alternative forms of polynomial representation would result in perhaps a moderate performance increase as well as possibly increase in the level of precision.

The proposal agreed is as follows:

> "Methods for polynomial evaluation such as Horner form and Estrin's method will be investigated. The implementation will be optimised and evaluated on a high performance computer, and potentially parallelised if necessary. Benchmark data sets from the Department of Civil Engineering, University of Glasgow will be used to evaluate the different methods."

This project combines techniques from computer science and parallel computing research with computational engineering. Currently there are no freely available libraries in Fortran or C to calculate polynomials using both serial Horner's form and parallel Estrin's method. The aim is produce a library which contains a number of different functions for evaluating arbitrary polynomials in a standardised fashion. The library is intended for release under the GPL and will be written in Fortran.

The project is to be limited in scope and the goals incrementally staged, to ensure they are achievable. The limited time available for the dissertation may necessitate alteration or curtailment of the stated aims. The aims of the project are:

1. Investigate polynomial representation and evaluation in computer codes
2. Implement a polynomial evaluation library for arbitrary uni-variate and multi-variate polynomials in Fortran
3. Undertake testing of the polynomial evaluation library
4. Investigate the testing results
5. Integrate the work and polynomial evaluation library with YaFFEMS to improve the evaluation of polynomials within that code.

1.2 Background & Literature Review

1.2.1 YaFFEMS

YaFFEMS is an on-going EPSRC funded project to develop a parallel implementation of the Finite Element Method (FEM) suitable for HPC, written in a mixture of C and C++. The code allows the Department of Civil Engineering, at the University of Glasgow to solve very large scale multi-physics problems using multi-scale modelling. The implementation uses a finite element formulation based on Hybrid Trefftz elements. The strategy in developing the software was to develop a computational tool which could take maximum advantage of high performance computers. To achieve this they have integrated advanced software such as MOAB (A Mesh-Orientated datABase), PETSc (Portable, Extensible Toolkit for Scientific Computation), ZOLTAN (Parallel Partitioning, Load Balancing and Data-Management Services), METIS (Family of Multilevel Partitioning Algorithms) as well as several others [1].

Multi-scale modelling is different from traditional FEM. For example, to model a concrete beam in traditional FEM, the beam would be approximated to a line element, given average material properties and its behaviour analysed. However, in multi-scale modelling, information from the behaviour of the material at the macroscopic level and below is used to inform the analysis; in this example, it would be represented as a beam, made up of cement & aggregate which when mixed form concrete. The behaviour and interaction of the cement & aggregate would influence the overall behaviour of the beam. Consequently, this method is extremely demanding on computational resources but leads to a more realistic analysis.

For the purposes of this project, it is sufficient to know that in its computations, YaFFEMS executes many polynomials in its solutions of FEM problems. Currently, the polynomials in YaFFEMS are generated in a specialised MAPLE file, which does the necessary pre-processing to produce the polynomials in a reasonably optimised form by using common sub-expression elimination (CSE). The resulting polynomials are then output into source code files by MAPLE and compiled into the project. The authors of the project believe that an alternative polynomial evaluation method could lead to both performance and precision gains. Additionally, they would like to be able to change the polynomials more easily, therefore an alternative polynomial representation is necessary.

1.2.1.1 Polynomials in YaFFEMS

Currently in YaFFEMS, polynomials are stored in a factored form in 18 separate C source code files each containing a single function which returns a double precision array. These files are generated by specialised MAPLE worksheets, which perform the necessary preparatory steps for generating the polynomials. The 18 code files contain 3 sets of polynomials: those representing stress, strain and displacement (hereafter referred to as Sv, Ev and Uv respectively). Each set of polynomials is split into 6 orders, with order 1 being the least complex and order 6 being the most complex. The files are named in the following scheme: [Uv,Ev,Sv]_poly#.c (for example: Uv_poly1.c would contain the function Uv_poly1).

There are only 6 orders of polynomials because above the 6th order, the current polynomial representation and evaluation method is not precise enough. The polynomials are stored in 1D arrays which use arithmetic to simulate the functionality of a 2D array. For example, for the 1st order Uv polynomials, the array is defined as follows:

```
inline void Uv_poly1(double x,double y,double z,double lambda,double mu,double *Uv)
{
Uv[0*3+0] = x;
Uv[0*3+1] = 0;
Uv[0*3+2] = 0;
Uv[1*3+0] = 0;
Uv[1*3+1] = y;
Uv[1*3+2] = 0;
```

```
Uv[2*3+0] = 0;
Uv[2*3+1] = 0;
Uv[2*3+2] = z;
Uv[3*3+0] = y;
Uv[3*3+1] = x;
Uv[3*3+2] = 0;
Uv[4*3+0] = z;
Uv[4*3+1] = 0;
Uv[4*3+2] = x;
Uv[5*3+0] = 0;
Uv[5*3+1] = z;
Uv[5*3+2] = y;
}
```

The 1st order polynomials are very simple; each entry in the array is a complete polynomial. However, for the higher order polynomials, the MAPLE worksheets perform common sub-expression elimination (CSE) in order to simplify the polynomials since there are a number of common monomials or blocks of monomials. The array for the 2nd order Uv polynomials is defined as follows:

```
inline void Uv_poly2(double x,double y,double z,double lambda,double mu,double *Uv)
{
double t1 = x * x;
double t2 = mu * t1;
double t3 = 0.2e1 * t2;
double t4 = y * y;
double t5 = lambda * t4;
double t7 = mu * t4;
double t10 = 0.1e1 / mu;
double t13 = x * y;
double t14 = lambda * t1;
double t17 = 0.5000000000e0 * t10 * (t2 + t14);
double t18 = x * z;
double t19 = z * z;
double t21 = 0.2e1 * mu * t19;
double t22 = 0.2e1 * t7;
double t30 = 0.5000000000e0 * t10 * (-0.1e1 * t7 - 0.1e1 * t5);
double t31 = 0.2e1 * t14;
double t32 = 0.4e1 * t2;
double t36 = y * z;
Uv[0*3+0] = 0.5000000000e0 * t10 * (t3 - 0.2e1 * t5 - 0.4e1 * t7);
Uv[0*3+1] = 0.0e0;
Uv[0*3+2] = 0.0e0;
Uv[1*3+0] = t13;
Uv[1*3+1] = -t17;
Uv[1*3+2] = 0.0e0;
Uv[2*3+0] = t18;
Uv[2*3+1] = 0.0e0;
Uv[2*3+2] = -t17;
Uv[3*3+0] = 0.5000000000e0 * t10 * (t21 - t22);
Uv[3*3+1] = 0.0e0;
Uv[3*3+2] = 0.0e0;
Uv[4*3+0] = t30;
Uv[4*3+1] = t13;
Uv[4*3+2] = 0.0e0;
Uv[5*3+0] = 0.0e0;
Uv[5*3+1] = -0.5000000000e0 * t10 * (-t22 + t31 + t32);
Uv[5*3+2] = 0.0e0;
Uv[6*3+0] = 0.0e0;
Uv[6*3+1] = t36;
Uv[6*3+2] = -t17;
Uv[7*3+0] = 0.0e0;
Uv[7*3+1] = -0.5000000000e0 * t10 * (t3 - t21);
Uv[7*3+2] = 0.0e0;
Uv[8*3+0] = t30;
Uv[8*3+1] = 0.0e0;
Uv[8*3+2] = t18;
Uv[9*3+0] = 0.0e0;
Uv[9*3+1] = 0.0e0;
Uv[9*3+2] = -0.5000000000e0 * t10 * (-t22 + t3);
Uv[10*3+0] = 0.0e0;
Uv[10*3+1] = -t17;
Uv[10*3+2] = t36;
Uv[11*3+0] = 0.0e0;
Uv[11*3+1] = 0.0e0;
Uv[11*3+2] = -0.5000000000e0 * t10 * (t32 + t31 - t21);
}
```

To allow the polynomial functions to be easily iterated over, in the UvEvSv.c source file, the 18 functions are grouped together by pointers for each of the Sv, Ev and Uv quantities as follows:

```
static void (*ptSvFunc[])(double,double,double,double,double,double *Sv) =
  { Sv_poly1, Sv_poly2, Sv_poly3, Sv_poly4, Sv_poly5, Sv_poly6 };
static void (*ptUvFunc[])(double,double,double,double,double,double *Uv) =
  { Uv_poly1, Uv_poly2, Uv_poly3, Uv_poly4, Uv_poly5, Uv_poly6 };
static void (*ptEvFunc[])(double,double,double,double,double,double *Ev) =
  { Ev_poly1, Ev_poly2, Ev_poly3, Ev_poly4, Ev_poly5, Ev_poly6 };
```

To illustrate how the polynomial functions are called, an example function is shown below. The order argument specifies the maximum order of the polynomial to return, the coord argument is an array with coordinates in three dimensions and lambda & mu are parameters of the Hybrid Trefftz equations.

```
void Uv_poly(int order,double *coord,double lambda,double mu,double *Uv)
{
  assert(order <= T_max_poly);
  double x = coord[0],y = coord[1], z = coord[2];
  int i = 0,shift = 0;
  for(; i<=order; i++) {
    (*ptUvFunc[i])(x,y,z,lambda,mu,&(Uv[3*shift]));
    shift += T_size_tab[i];
  }
}
```

An example test program for YaFFEMS is presented below:

```
#include "UvEvSv.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static double eps = 1e-12;
static void print(int ii,double x) {
  if(fabs(x) < eps) x = 0.;
  printf("%d %+3.2f\n",ii,x);
}

int main(void) {
    double *Sv,*Uv;
    int order = 4,ii;
    double coord[] = { 0.5, -0.5, -0.5 };

    Sv = (double *)malloc( sizeof(double)*T_size_tab_cumulative[order]*6 );
    Uv = (double *)malloc( sizeof(double)*T_size_tab_cumulative[order]*3 );
    Sv_poly(order,coord,1.,1.,Sv);
    Uv_poly(order,coord,1.,1.,Uv);
    puts("\nSv");
    for(ii = 0; ii<T_size_tab_cumulative[order]*6; ii++)
      print(ii,Sv[ii]);
    puts("\nUv");
    for(ii = 0; ii<T_size_tab_cumulative[order]*3; ii++)
      print(ii,Uv[ii]);
    free(Sv);
    free(Uv);

    return EXIT_SUCCESS;
}
```

The issue with the above method is that a significant amount of pre-processing is required to change the polynomials. The creators of YaFFEMS are primarily concerned with precision, followed by performance. However, it is the author's understanding that it would be advantageous if it were possible to change more easily the polynomials being evaluated.

1.2.2 Polynomial evaluation techniques in computer codes

In general, if asked to represent a 7th order polynomial such as $y = cx^7 + bx^5 + ax^3 + x$, there are a number of ways in which it could be represented in code. One such method a beginner programmer would likely use would be to write it out long hand i.e. $y = c * x * x * x * x * x * x * x + b * x * x * x * x * x * x + a * x * x * x * x + x$ or using power operators i.e. $y = c * power(x, 7) + b * power(x, 5) + a * power(x, 3) + x$. Obviously these methods, especially the former one, become impractical for polynomials of a larger size.

There are a few techniques for efficiently representing polynomials in computer codes, such as the Horner form and Estrin's method. Although these techniques have been in existence for some time, there is relatively little reference material available regarding them.

One presentation which covers both the Horner form and Estrin's method is one by Robin Green, a Research & Development programmer with Sony Computer Entertainment America. It is entitled "Fast Math Functions" and is concerned with the writing of Math functions, optimising for speed, accuracy and memory usage. A section of the presentation is concerned with "Fast polynomial evaluation" and covers the advantages & disadvantages of using the Horner form and Estrin's method [2]. The Horner form and Estrin's method are also mentioned briefly in "The Art of Computer Programming" by Donald Knuth [3].

The first alternative, the Horner form, would express the above 7th order polynomial as:

$$z = x * x;\ y = (((c * z + b) * z + a) * z + 1) * x) \tag{1.1}$$

The Horner form is suitable for processors with fused multiply-add instructions, however the calculation is purely serial. The issue with Horner's form is that later instructions are reliant on earlier results, leading to a long, thin pipeline. It is likely that due to the fused multiply-adds used in Horner's form (if the host processor supports them), the reduced rounding error inherent to these instructions will make Horner's form more precise.

The second alternative, Estrin's method, would express the above 7th order polynomial as:

$$y = ((c * x) * x^2 + b * x)) * x^4 + ((a * x) * x^2 + (x)) \tag{1.2}$$

The additional advantage of Estrin's method is that it can be vectorised or even evaluated in parallel. If coded correctly, Estrin's method is suitable for parallelisation as each bracketed sub-expression is independent of any other sub-expression, therefore the instruction pipeline can be wide, rather than long.

1.2.3 Technologies Used

1.2.3.1 Fortran

Fortran is a general purpose programming language well suited to numeric, scientific and engineering computation. It is a strongly, statically typed programming language originally developed by IBM in the 1950s. It has been developed over many decades in the form of several published standards from the early IBM versions to FORTRAN 66 (1966), FORTRAN 77 (1977), Fortran 90 (1991), Fortran 95 (1997) and beyond. It is now one of the most popular programming languages in HPC.

Fortran will be the programming language of choice for this project.

1.2.3.2 OpenMP

The OpenMP standard is a parallel programming model for shared memory systems. It uses compiler directives and library routines to achieve parallelism. It uses multi-threading to execute blocks of code in parallel. Both data parallelism and task parallelism can be achieved in OpenMP. It is a widely implemented standard in HPC that programmers with a simple, high level approach for introducing parallelism into their codes.

It can only be run on shared memory systems, but is sometimes used in SMP nodes of a distributed memory system in what is known as mixed mode programming (both MPI and OpenMP).

OpenMP will be used to parallelise Estrin's method in this dissertation.

1.2.3.3 MPI

The Message-Passing Interface standard defines a library, not a language and is a specification, not an implementation. It is a widely accepted standard in HPC. It could also perhaps be described as the "assembler" of the parallel programming world, because of its low level approach to programming. It is a collection of library routines that can be called from existing languages, using standard compilers and by linking in the MPI library.

MPI is based on the basic message passing model, where communication is explicit as messages are passed between processors. At the most basic level, it is a collection of processes communicating with messages.

It can be run on both shared and distributed memory systems, although it is primarily used on the latter.

MPI will also be used to parallelise Estrin's method in this dissertation.

1.2.3.4 MAPLE

The mathematics package MAPLE is a computer algebra system used for technical computing. It allows infinite precision calculations and is more accurate than implementations of the IEEE Standard of Floating-Point Arithmetic (IEEE 754). By symbolically evaluating equations, highly accurate results can be achieved. Equations are entered in traditional mathematical notation.

MAPLE will be used to compare the numerical output of the polynomial evaluation methods tested in this work to a symbolically evaluated result for the polynomial to arbitrary precision (known as "infinite" or "unlimited" precision on the MAPLE website) [4].

Chapter 2

Development

2.1 Uni-variate polynomials

A polynomial is a mathematical expression composed of constants and variables, using only addition, subtraction, multiplication and positive integer powers. A uni-variate polynomial is a polynomial in one independent variable, whereas a multi-variate polynomial is a polynomial in multiple independent variables.

Initially, the techniques of Horner's form and Estrin's method were implemented for uni-variate polynomials. Additionally, brute-force methods were implemented for comparison, which evaluate the polynomial directly, rather than use a special algorithm.

However, before work could begin, it was necessary to create a way of representing a polynomial. This was achieved by using Fortran derived types.

2.1.1 Uni-variate Polynomial Derived Type

In order to represent a uni-variate polynomial, three variables are required: a variable to hold the number of terms in the polynomial, an array to hold the coefficients for each monomial and the value of the independent variable.

```
integer, parameter :: prec = kind(1.0d0) !Specify double precision for reals

type polynomial
        integer :: n
        real (kind=prec), allocatable, dimension(:) :: f !Coefficients of polynomial
        real (kind=prec) :: x !Value of independent variable
end type
```

It is not required to store the powers of the polynomial for a uni-variate polynomial, as they are sequential. For example, for a 7th order polynomial ($hx^7 + gx^6 + fx^5 + ex^4 + dx^3 + cx^2 + bx + a$), if there was a powers array it would contain $[7, 6, 5, 4, 3, 2, 1, 0]$ and is thus entirely redundant. Therefore, it removes the need to store and load the powers. Indeed, as will be seen with Horner's form a powers array would never be used because of the factorisation used within that method.

2.1.2 Horner's Form

For a uni-variate polynomial, Horner's Form is renowned as the fastest way to compute its value. This is because it is the method with the smallest possible number of arithmetic operations (additions and multiplications). Horner's form solves polynomial equations of any degree by converting the expression into a nested set of multiply-add instructions; therefore Horner's form is good for processors with fused multiply-add instructions. A fused multiply-add instruction calculates the result of the sum $a + b \times c$ in one step, with a single rounding error. This is an improvement over processors without fused multiply-add instructions as they would evaluate first $b \times c$, round the result, then add the result to a and round the result. Therefore processors with a fused multiply-add instruction improve the accuracy and speed of polynomial evaluation [5, 6].

The pseudo-code for evaluating a polynomial using Horner's form is as follows. The array *coefficient* is an array of the $C_x$values i.e. $C_3x^3 + C_2x^2 + C_1x + C_0$ would be represented in an array as $[C_3, C_2, C_1, C_0]$.

Algorithm 2.1 Horner's form pseudo-code

1. result = coefficient(1)*x
2. do i = 2, n - 1
   (a) result = (result + coefficient(i))*x
3. end do
4. result = result + coefficient(n)

Horner's rule computes $f(x)$ by nested multiply-adds as follows:

$$f(x) = \sum_{i=0}^{n} a_i x^i \tag{2.1}$$

For example, to evaluate a 7th order polynomial ($hx^7 + gx^6 + fx^5 + ex^4 + dx^3 + cx^2 + bx + a$), Horner's form would evaluate the polynomial in the following fashion:

$$((((((hx + g)x + f)x + e)x + d)x + c)x + b)x + a \tag{2.2}$$

Horner's form is often the default method for evaluating polynomials in many libraries; i.e. polynomials are stored as arrays of coefficients and evaluated using a Horner's form function. However, the major downside to Horner's form is that it is purely serial; later results rely on the results of previous instructions i.e. the expression tree is deep rather than wide . On modern architectures a deep expression tree is generally not desirable; rather the expression tree should be wide which would allow the sub-trees to be evaluated in parallel. [2, 7]

A method which theoretically improves on Horner's form by introducing potential parallelism is Estrin's method.

2.1.3 Estrin's Method

Estrin's method comes from parallel computing research. It works by dividing the polynomial into an implied tree of multiply-adds, allowing each level of the tree to be evaluated in parallel. The main idea of Estrin's method is to separate the polynomial into terms of the form $(Bx + A)$, evaluating them and then using the results as the coefficients in the next level of $(Bx + A)$. However, unlike Horner's form, it does not reduce the number of arithmetic operations required to evaluate a polynomial. The expression x^{2N} is used as the "glue" between expressions in the next level N+1 [2, 7]. Estrin's method stems from the observation that regular patterns emerge when polynomials are factorised. For example, a 3^{rd} order expression can be factorised into sub-expressions of $(Bx + A)$ as follows.

$$f(x) = dx^3 + cx^2 + bx + a = (dx + c)x^2 + (bx + a) \tag{2.3}$$

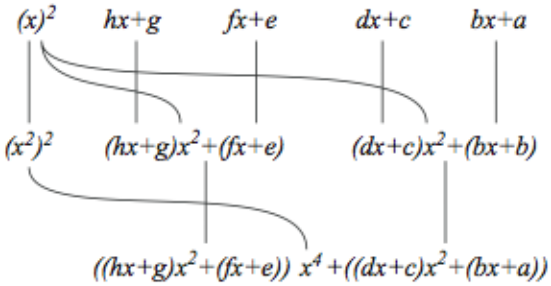
Extending this to higher order polynomials, an Estrin's method table can be built up:

$p_0(x) =$ a
$p_1(x) =$ $(bx+a)$
$p_2(x) =$ $(c)x^2+(bx+a)$
$p_3(x) =$ $(dx+c)x^2+(bx+a)$
$p_4(x) =$ $(e)x^4+((dx+c)x^2+(bx+a))$
$p_5(x) =$ $(fx+e)x^4+((dx+c)x^2+(bx+a))$
$p_6(x) =$ $((g)x^2+(fx+e))x^4+((dx+c)x^2+(bx+a))$
$p_7(x) =$ $((hx+g)x^2+(fx+e))x^4+((dx+c)x^2+(bx+a))$
$p_8(x) =$ $(i)x^8+(((hx+g)x^2+(fx+e))x^4+((dx+c)x^2+(bx+a)))$
$p_9(x) =$ $(jx+i)x^8+(((hx+g)x^2+(fx+e))x^4+((dx+c)x^2+(bx+a)))$
$p_{10}(x) =$ $((k)x^2+(jx+i))x^8+(((hx+g)x^2+(fx+e))x^4+((dx+c)x^2+(bx+a)))$
$p_{11}(x) =$ $((lx+p)x^2+(jx+i))x^8+(((hx+g)x^2+(fx+e))x^4+((dx+c)x^2+(bx+a)))$

(Source: [2])

Figure 2.1: Estrin's Method table

For example, the evaluation of a 7^{th} order polynomial ($hx^7 + gx^6 + fx^5 + ex^4 + dx^3 + cx^2 + bx + a$) looks like this:



(Source: [2])

Figure 2.2: Evaluation of a 7^{th} order polynomial using Estrin's method

The terms in each row of Figure 2.2 are evaluated in parallel; all the terms on a row must be fully evaluated before the next row can be started. Excluding the calculation of x^2 and $(x^2)^2$, it can be observed that at each step, the number of operations required divides by two i.e. 4 multiply-adds for the first step, 2 multiply-adds for the second step and 1 multiply-add for the

last step in the above example. Therefore, in order for the algorithm to work, the degree of the polynomial must be a power of two. If the degree of the polynomial is not a power of two, it must be rounded up to the nearest power of two i.e. for a polynomial of degree 12, Estrin's method would pad the coefficients of the polynomial with zeros so that the degree of the polynomial would be rounded up to 16. This ensures that the divisions do not result in fractions.

Estrin's method uses an implicit binary evaluation tree implied by splitting $f(x)$ as:

$$\left(\sum_{0\leq i<h} a_h + i^{x^i}\right) x^h + \left(\sum_{0\leq i<h} a_i x^i\right) \tag{2.4}$$

where $h = \frac{(n+1)}{2}$ is a power of 2 [8]. An N^{th} degree polynomial, requires $log_2(N) + 1$ rows of expressions to evaluate.

The pseudo-code for evaluating a polynomial using Estrin's method is as follows. The coefficients of the polynomial are inserted into the first row of the array *coefficient* and subsequently as the polynomial is evaluated the newly computed $(Bx + A)$ coefficients are inserted into the next rows of this array. Finally, the result can be found at the first entry of $numSteps$ row.

Algorithm 2.2 Estrin's Method Pseudo-code

1. nearestPowerOfTwo = $2^{ceiling(log_2(n))}$
2. if (modulus of (n, nearestPowerOfTwo) doesn't equal 0) then
   (a) shift = nearestPowerOfTwo - modulus(n, nearestpoweroftwo)
   (b) npow2 = degree + shift
3. else
   (a) shift = 0
   (b) npow2 = n
4. end if
5. numSteps = $2^{(log_2(npow2))}$
6. coefficient(:, 0) = coefficients of the polynomial
7. powersArray(1) = x
8. do i = 2, numSteps
   (a) powers(i) = powers(i-1)^2
9. end do
10. do i = 1, numSteps
    (a) do j = 1, $\frac{npow2}{2^i}$
        i. coefficient(j, i) = coefficient(2*j-1, i-1) * powers(i) + coefficient(2*j, i-1)
    (b) end do
11. end do
12. result = coefficient(1, numSteps)

2.1.3.1 Implementation of a parallel Estrin's Method

Two approaches were taken to the parallelisation of Estrin's Method. Initially an OpenMP version was created, simply by placing an !$OMP PARALLEL DO around the appropriate calculation DO loop.

In order to obtain a performance comparison, an MPI version was also created. The MPI version works by sharing out portions or chunks of the calculation DO loop to each MPI process. At the end of each calculation step, all MPI processes communicate their results back to the root MPI process. The root MPI process then collates the results into the coefficient array and then broadcasts the updated coefficient array to each MPI process before the next calculation step begins.

OpenMP In order to implement the OpenMP version of Estrin's Method a PARALLEL DO compiler directive was placed around the inner j loop of the calculation step loop.

```
do i = 1, numsteps
        !$omp parallel do default(none) shared(coeff, powers, npow2, i) private(j)
        do j = 1, npow2/(2**i)
                coeff(j, i) = coeff(2*j-1, i-1)*powers(i)+coeff(2*j, i-1)
        end do
        !$omp end parallel do
end do
```

MPI In order to implement the MPI version of Estrin's Method, a greater amount of work was required. Whilst OpenMP hides the complexity of sharing out the work across the threads, with MPI a more low-level approach is required.

Algorithm 2.3 Pseudo-code for MPI version of Estrin's Method

1. Calculate nearest power of two from the number of coefficients (npow2). If the number of MPI processes is greater than npow2, abort.
2. Calculate the number of calculation steps required to reach the result (numSteps)
3. Build the powers array (powers) and populate the first (0) row of the coefficient array with the polynomial coefficients
4. Evaluate using Estrin's Method:
   (a) Do i = 1, numSteps
      i. Work out the number of elements in the current array row that are "active".
      ii. If the array width in use at this step is greater than the active/effective number of MPI processes at this step then reduce the effective number of MPI processes.
      iii. Work out the size of the chunk of data to be shared to each MPI active/effective process
      iv. Get the lower limit and upper limit in this array row for this MPI process
      v. If the rank is less than or equal to the effective number of MPI processes, do the calculation.
      vi. If this is not the last step, send calculation results to root MPI process.
      vii. On the root MPI process, collate the received results and broadcast the updated array row to all the other processes.
   (b) End Do
5. result = coefficient(1, numSteps)

At each step the number of “active” elements in the current array row is half that of the number of “active” elements in the previous row. For example, for a 512 coefficient poly, at step 1 there are 512 coefficients but at step 2 there are only 256 coefficients. Therefore, array elements greater than 256 in this row at this current step are unused. At step 3 there will be 128 coefficients etc.

Therefore, it follows that there will become a point where the number of processors available will be greater than the number of coefficients. (The number of processes is limited to either 1 or a power of two). If at the current step there are only 8 calculations to be performed (i.e. array width of 16) and there are 16 MPI processes, then reduce the effective number of MPI processes by a factor of two. Therefore, in this example, 8 processors would have one calculation to perform and 8 processors would be idle.

2.1.3.2 Issues with Estrin’s Method

Upon inspection of Estrin’s method, a problem of load balance in the parallelisation of Estrin’s method is apparent. Given say a polynomial of 512, which requires 9 steps to evaluate, and say, 16 processors/threads upon which to evaluate it using OpenMP, it can be observed that there is a rapid decrease in the utilisation of the processors as the evaluation executes. As the number of coefficients divides by two at each step, therefore, for example, at step 5 there are only 32 coefficients i.e. only 16 multiply-add operations (one per processor/thread). In the following steps 6 through 9, there are an increasing number of entirely idle processors (although it could be argued that at step 5, with only one operation per thread that each processor is virtually idle). This presents a load balance problem which is inherent to the algorithm. The load balance is distributed in a triangular fashion, with the number of operations dividing by two at each step until it reaches 1 operation at the final step.

It should also be noted that Estrin’s method was originally intended to use parallelism in the form of processor pipelines or floating point units. By creating a wide expression tree, as opposed to the deep expression tree found in Horner’s form, it is possible to parallelise it. However, in this work, Estrin’s method will be parallelised by the multi-processor method rather than parallelism on a single processor.

2.1.4 Brute Force Methods

In order to allow performance comparisons to be made between Horner’s form and Estrin’s method, another class of evaluation method was required as a reference point.

Whilst Horner’s form and Estrin’s method use (admittedly simple) algorithms, the “Brute Force” methods do not use a specialised approach but rather evaluate the polynomial directly, using a “brute force” approach. Whilst the term “brute force” is often used in the context of encryption and authentication attacks, in this context the author has taken it to mean an easy, blunt, direct, unsophisticated approach without finesse or subtlety. For example, for the uni-variate case:

Algorithm 2.4 Pseudo-code for Brute-force Method

1. n=number of polynomial coefficients
2. do i = 1, n -1
   (a) monomial(i) = coefficients(i)*x**(n - i)
3. end do
4. monomial(n) = coefficients(n)
5. result = sum of the monomials

2.1.4.1 Optimised Brute Force

The Optimised Brute Force function pre-calculates the value of x^2 and use it where possible in building up the powers for each monomial. For example, if a monomial contains x^7, instead of 7 multiplications of x by x, the function would multiply x^2 by x^2 three times then multiply it by x once. In this example, this would save three multiplications.

Algorithm 2.5 Pseudo-code for Brute Force - Optimised Method

1. n=number of polynomial coefficients
2. x2 = x*x
3. do i = 1, n-1
   (a) monomial(i) = coefficients(i)
   (b) numSteps = (n-i)/2
   (c) do j = 1, numSteps
      i. monomial(i) = monomial(i) * x2
   (d) end do
   (e) if (modulus of (n-i, 2) is not equal to 0) then
      i. monomial(i) = monomial(i) * x
   (f) end if
4. end do
5. monomial(n) = coefficients(n)
6. result = sum of the monomials

2.1.5 Basic Evaluation of Polynomials

Additionally, in order to gain a further reference point for comparing the performance of the above polynomial evaluation methods, a further method was required (henceforth named the "Basic" method).

As mentioned in the introduction, polynomials can be evaluated "long hand" i.e. written out fully in the source code. Due to the size and length of the polynomials used in the test cases for this project, it was necessary to develop a short script to automate the generation of these polynomial representations. The following shell script is a basic illustrative example of the code used. (*Note: "\c" suppresses the new line usually output by the echo command.*)

```
#!/bin/sh
for i in {14..1}
do
    echo "coeff($i)*x**$i + \c"
done
echo "coeff(0) \c"
```

The above script would generate the following output for a 15 term polynomial:

```
coeff(14)*x**14 + coeff(13)*x**13 + coeff(12)*x**12 + coeff(11)*x**11 + coeff(10)*x
    **10 + coeff(9)*x**9 + coeff(8)*x**8 + coeff(7)*x**7 + coeff(6)*x**6 + coeff(5)*x
    **5 + coeff(4)*x**4 + coeff(3)*x**3 + coeff(2)*x**2 + coeff(1)*x**1 + coeff(0)
```

The output from the script would then be inserted inside a block of timing statements and evaluated under the same conditions as the other methods.

2.2 Multi-variate Polynomials

The above methods of Horner's form and Estrin's method are limited to polynomials in single variables i.e. uni-variate. However, the polynomials used in YaFFEMS contain at least 3 variables. As such, a method to handle such polynomials is required.

2.2.1 Multi-variate Polynomial Derived Type

In order to represent a multi-variate polynomial, five variables are required: a variable to hold the number of terms in the polynomial, a variable to store the number of independent variables in the polynomial, an array to hold the coefficients of each monomial, an array to store the values of the independent variables and an array to store the powers for each variable in each monomial.

```
integer, parameter :: prec = kind(1.0d0) !Specify double precision for reals
integer, parameter :: long = selected_int_kind(12) !Specify length of integer

type polynomial_multi
        integer :: n
        integer :: m
        real (kind=prec), allocatable, dimension(:) :: f !Coefficients of polynomial
        real (kind=prec), allocatable, dimension(:) :: vars !Independent variables
        integer (kind=long), allocatable, dimension(:,:) :: powers
end type
```

2.2.2 Multi-variate Horner Form

An extension of Horner's form which is able to evaluate polynomials of more than one variable is the Multi-variate Horner form.

There are a number of variants, described in several research papers, most notably Carnicer & Gasca [9], Pena & Sauer [10] and Ceberio & Kreinovich [11]. These papers present several methods for evaluating polynomials using multi-variate Horner's form. From the research to date, it is not clear which of the several different methods is optimal. Additionally, none present implementations or pseudo-code of their algorithms, preferring instead to use high level mathematical notation to describe their algorithms.

Unfortunately, despite the best efforts of both myself, my supervisor and a colleague of my supervisor with a PhD in Mathematics, we were unable to understand the papers sufficiently to implement the algorithms contained within.

This severely impeded the progress of the project. It was later decided to refocus the project solely on uni-variate polynomials in order to progress with the project.

2.2.2.1 Carnicer & Gasca

Carnicer & Gasca takes a tree based approach using graph theory to solving the problem of multi-variate Horner's form. They present a series of highly technical mathematical algorithms and related definitions which reformulate the polynomial into a tree mapping based on multi-indices and then traverses the tree in an ordering defined mathematically; put simply, the algorithm traverses the tree from the furthest node, along the paths/branches, until it reaches the root of the tree.

The approach presented in this paper was highly technical and required a very high level of mathematical knowledge in order to interpret it.

2.2.2.2 Pena & Sauer

Pena & Sauer present an approach based upon storing the coefficients and multi-indices in inverse lexicographical order. The algorithm they develop separates the calculation into intermediate summation processes, rather than executing a recursive evaluation based upon the sets defined in the paper.

2.2.2.3 Ceberio & Kreinovich

The approach presented in Ceberio & Kreinovich is to apply Horner's form to a multi-variate polynomial by selecting a variable (say x) and treating all other variables as if they are constants. The resulting coefficients are then functions of all the remaining variables. In order to compute these coefficients, one of the remaining variables must be selected and Horner's scheme applied using this second variable, etc etc. The main issue with this approach is that it is difficult to determine the appropriate order of variables programmatically. It is possible, by selecting a sub-optimal order, to produce a factorisation which takes more arithmetic operations to calculate.

The authors present a greedy, heuristic method for finding the optimal order of variables for the multi-variate Horner's form.

2.2.3 Brute-Force

As with the uni-variate case, it was relatively simple to implement a blunt and direct approach to evaluating the polynomials. The methods presented below are a simple extension to Algorithms 2.4 & 2.5.

Algorithm 2.6 Pseudo-code for Multi-variate Brute Force

1. n=number of polynomial coefficients
2. m=number of independent variables
3. do i = 1, n-1
   (a) monomial(i) = coefficients(i)
   (b) do j = 1, m
      i. monomial(i) = monomial(i)*variables(j)**powers(j,i)
   (c) end do
4. end do
5. monomial(n) = coefficients(n)
6. result = sum of the monomials

Algorithm 2.7 Pseudo-code for Multi-variate Brute Force - Optimised

1. n=number of polynomial coefficients
2. m=number of independent variables
3. do i = 1, m
   (a) variables2(i) = variables(i)*variables(i)
4. end do
5. do i = 1, n-1
   (a) monomial(i) = coefficients(i)
   (b) do j = 1, m
      i. numSteps = (powers(j,i))/2
      ii. do k = 1, numSteps
         A. monomial(i) = monomial(i) * variables2(j)
      iii. end do
      iv. if (modulus of (powers(j,i), 2) is not equal to 0) then
         A. monomial(i) = monomial(i) * variables(j)
      v. end if
   (c) end do
6. end do
7. monomial(n) = coefficients(n)
8. result = sum of the monomials

2.3 PolyEval library

In order to combine the work on polynomial evaluation techniques, a library of functions for evaluating both uni-variate & multi-variate polynomials was created. This included Horner's form and Estrin's method, as well as other techniques for evaluating polynomials. It was decided that the library would be hosted on Google Code, in order to facilitate its future use by other groups. The site (http://code.google.com/p/polyeval/) hosts the project SVN repository.

The PolyEval library is written in Fortran and intended to contain a number of methods for evaluating an arbitrary polynomial, in any number of variables. For example: Horner's form, Estrin's method, Brute Force evaluation and optimised Brute Force methods.

The library uses Fortran Derived Data-types to represent polynomials in either one or multiple variables. There are functions to evaluate a polynomial in one or more variables using a basic Brute Force method as well as an optimised method. There is also Horner's form and Estrin's method for polynomials in one variable. It was planned for there to be a multi-variate Horner's form function also but major difficulties were encountered in implementing the function.

2.3.1 YaFFEMS and PolyEval

Although it was not possible to integrate the PolyEval library with YaFFEMS within the time-frame of the project - due to the problems with Multi-variate polynomials - it is possible to show the benefits of such an integration.

In the uni-variate case, only the coefficients of the polynomial need to be stored. However, in the multi-variate case, both the coefficients and the multi-indices (the powers to which each variable is raised to in each monomial) need to be stored.

To represent and evaluate a bi-variate polynomial such as $gx^2y^2 + fx^2y + exy^2 + dxy + cx + by + a$ for the values of $x = 2$, $y = 3$, the following code would be required. (*Note: the variables a to g in the definition of the coefficient array are used for convenience and clarity in this example.*)

```
use modPolyEval
real (kind=prec) :: result
!modPolyEval defines prec as "integer, parameter :: prec = kind(1.0d0)" i.e. double precision

type(polynomial_multi) :: poly

!Define arrays
poly%n = 7
poly%m = 2
allocate(poly%f(poly%n))
allocate(poly%vars(poly%m))
allocate(poly%powers(poly%m, poly%n-1))

!Assign coefficients
poly%f = (/g,f,e,d,c,b,a/) !Where a-g are variables containing the coefficients.

!Assign values of independent variables
poly%vars(1)=2.0
poly%vars(2)=3.0

!Assign multi-indices (powers to which the variables are raised to in each monomial)
poly%powers(1,:) = (/2,2,1,1,1,0/)
poly%powers(2,:) = (/2,1,2,1,0,1/)

!Evaluate polynomial
result = Eval_multi(poly)
```

Therefore, with a small amount of wrapper/initialisation code (such as above) for the PolyEval library, different polynomials can be represented and evaluated.

In the context of YaFFEMS, a penta-variate polynomial would need to be defined (5 variables: x, y, z, lambda and mu). In order to be efficient, it would be shrewd to ensure that any given polynomial is only defined once and evaluated many times with different permutations of variable values. This is different from the current workings of YaFFEMS, where many polynomials are evaluated for a particular set of values.

Although re-factoring of YaFFEMS would be required to make the best, efficient use of the polynomial representation and evaluation methods of PolyEval, it is likely that some performance and precision gain would be made over the existing methods in YaFFEMS.

2.4 GPGPU

As an aside to the project, a small investigation was undertaken into the viability of using GPGPUs (general-purpose computing on graphics processing units) to evaluate polynomials. However, it quickly became apparent that in order to efficiently utilise a GPGPU, the polynomial evaluation kernel would need to run for a reasonable amount of time. Since Estrin's method is the only parallel polynomial evaluation technique, an attempt was made to port this to a machine with NVIDIA Tesla C2050 GPGPUs using the PGI CUDA for Fortran compiler.

To enable Estrin's method to be evaluated on a GPGPU, the PGI Accelerator Directives were used:

```
do i = 1, numsteps
        !$acc region
        do j = 1, npow2/(2**i)
                coeff(j, i) = coeff(2*j-1, i-1)*powers(i)+coeff(2*j, i-1)
        end do
        !$acc end region
end do
```

However, the overhead of making the initial connection to the GPGPU (roughly of the order of a second), far outweighed the kernel evaluation time for the degrees of polynomials tested. It was surmised that Estrin's method and therefore polynomial evaluation in the scope of this project was not suitable for GPGPUs. In order for GPGPUs to be useful, the degree of the polynomial and the number of terms would need to be extremely large, likely far beyond the size of polynomials used by YaFFEMS.

2.5 Integration of the work into YaFFEMS

As YaFFEMS is a mixed C/C++ code and the PolyEval library has been written in Fortran, in order to enable the integration of the work in this project into YaFFEMS, mixed language compilation would be required. Before the difficulties with Multi-variate Horner's form were encountered, some work was undertaken to implement this mixed language compilation.

It was discovered that it is moderately difficult to expose Fortran derived types to C. However, when those derived types use Fortran allocatable arrays, it becomes impossible without writing an "interface" to sit between the calling C code and the Fortran code. The purpose of this interface is to take in the input arguments from the calling YaFFEMS C code, allocate the Fortran derived type arrays based upon the input, repackage the data into the polynomial derived type and then pass the polynomial derived type to the appropriate PolyEval library function.

Additionally, all arguments in Fortran are passed by reference and not by value. Therefore, for YaFFEMS to successfully interact with PolyEval, it must pass arguments to the Fortran code by reference. It should also be noted that C and Fortran have different column orderings for multi-dimensional arrays. To pass multi-dimensional array data between C and Fortran, the programmer must implement a workaround for this. The workaround can either be code to reverse the column orderings or in say the case of two dimensional arrays, simply split the data into separate one dimensional arrays and pass those separately.

It is possible to include both C and Fortran code within the same executable, because they are compiled languages and as such their compiled object files can be linked together. However, one thing to note is that the naming of Fortran functions will differ from those defined in the source code. For example, to call a function named CEval in a Fortran module named modPolyEvalCInterface, the function name required in the C code would be "__modpolyevalcinterface_MOD_ceval". Additionally, the function must be declared as a C external function i.e. "extern double __modpolyevalcinterface_MOD_ceval ();".

The Makefile created, but ultimately unused because of the later change in direction of the project, is listed below. (*Note: The [Uv,Sv,Ev]poly#.c files are the relevant polynomial code files from YaFFEMS*)

```
MF= Makefile

FC = gfortran

FFLAGS =

CC = gcc
CFLAGS =

LFLAGS=

EXE= yaffems_polyeval

FSRC= \
        modPolyEval.f90 \
        modPolyEvalCInterface.f90

CSRC= \
        Uv_poly1.c \
        Uv_poly2.c \
        Uv_poly3.c \
        Uv_poly4.c \
        Uv_poly5.c \
        Uv_poly6.c \
        Sv_poly1.c \
        Sv_poly2.c \
        Sv_poly3.c \
        Sv_poly4.c \
        Sv_poly5.c \
        Sv_poly6.c \
        Ev_poly1.c \
        Ev_poly2.c \
        Ev_poly3.c \
        Ev_poly4.c \
        Ev_poly5.c \
        Ev_poly6.c \
        OmegaSv_poly0.c \
        OmegaSv_poly1.c \
        OmegaSv_poly2.c \
        OmegaSv_poly3.c \
        UvEvSv.c \
        yaffems_fem.c

#
# Makefile Rules
#

.SUFFIXES:
.SUFFIXES: .f90 .o
.SUFFIXES: .c .o

FOBJ=   $(FSRC:.f90=.o)
FMOD=   $(FSRC:.f90=.mod)
COBJ=   $(CSRC:.c=.o)

.f90.o:
        $(FC) $(FFLAGS) -c $<

.c.o:
        $(CC) $(CFLAGS) -c $<

all:    $(EXE)

$(EXE): $(FOBJ) $(COBJ)
        $(FC) $(FFLAGS) -o $@ $(FOBJ) $(COBJ) $(LFLAGS)

$(OBJ): $(MF)

clean:
        rm -f $(FOBJ) $(COBJ) $(EXE) $(FMOD) core
```

Chapter 3

Testing

3.1 Testing of polynomial evaluation techniques

In order to test the evaluation techniques for uni-variate polynomials, a series of test cases were created.

Test Case	Number of terms
1	5
2	15
3	30
4	100
5	500
6	1000
7	2000
8	4000

Table 3.1: Polynomial evaluation test cases

The performance tests were undertaken on `ness.epcc.ed.ac.uk`, a Sun Fire X4600 system with two boxes of 16 x 2.6 GHz AMD Opteron (AMD64e) each with 2 GB of memory. Each test case will be evaluated 1,000 times and the execution times recorded will be the average iteration time. For each test run, an entire box of 16 processors was reserved using the queue system.

For each test case, the different evaluation methods are used to obtain numeric results and execution times: the Basic method, the two Brute force methods, Horner's form and Estrin's method.

The PolyEval library and test program were compiled on `ness` using PGI Fortran (`pgf90`) with the `-fast` flag.

3.2 Data Collection Methodology

For each test case, each evaluation method is executed 1,000 times and the average iteration time calculated. The fastest average iteration time is selected for the results. The average of the iteration times across the multiple executions of a test was not selected because it can be influenced by extreme times. There are many factors which can cause a code to run slower but very few - if any - which can cause a code to run faster (except in edge cases such as

errors and premature termination). In order to ensure results are reliable, for each test run all 16 processors on a Ness node were reserved using the queue system. This should reduce distortion/degradation of execution times from outside influences such as OS operations.

3.3 Random Coefficients

For each test case, consisting of between 5 and 4,000 term polynomials, an array of random coefficients is generated in the range 0 to 10. The seed for the random number generator is manually set as otherwise the compiler will seed the generator with a static number i.e. on subsequent executions the values generated will be the same. The Fortran intrinsic random_number fills the entire array rndArray with random numbers between 0 and 1, which is then multiplied by 10 to get coefficients in the range 0 to 10.

```
subroutine generateRandomArray(size, rndArray)
        integer, intent(IN) :: size
        real(kind=prec), allocatable, intent(OUT), dimension(:) :: rndArray
        integer :: seedSize,date(8)
        integer, allocatable  :: seed(:)

        call date_and_time(values=date)
        call random_seed(SIZE=seedSize)
        allocate( seed(seedSize) )
        call random_seed(GET=seed)
        seed = seed * date(8)       ! date(8) is milliseconds
        call random_seed(PUT=seed)

        allocate(rndArray(size))

        call random_number(rndArray)

        rndArray(:) = rndArray(:) * 10

end subroutine generateRandomArray
```

As was stated in previous sections, two versions of PolyEval exist: an OpenMP version and an MPI version. With MPI, each MPI task is a separate process whereas with OpenMP a single master process executes until a parallel region is encountered and then slave threads are spawned. This gives rise to an obvious problem regarding the generation of random coefficients. To ensure each MPI process uses the same coefficients for Estrin's method, the randomly generated array must be broadcast from process rank 0 to all other processes. Whereas with OpenMP, the coefficients array can simply be shared from the master thread to all the slave threads with a compiler directive.

Due to the generation of random coefficients for each test case for each test run, code was added to the test program to output a formula representing the polynomial, including numeric coefficients. This was necessary in order to check the precision of each polynomial evaluation method. The polynomial formula could then be symbolically evaluated in MAPLE. A MAPLE worksheet was created to take the numeric results of the polynomial evaluation methods and calculate the percentage difference to the symbolically evaluated result.

3.4 Test Program

In the OpenMP version of PolyEval, timing uses *omp_get_wtime()* and in the MPI version of PolyEval *MPI_Wtime()* is used. It should be noted that only Estrin's method executes on more than more thread. This is achieved in the OpenMP version by placing the appropriate OMP compiler directive around the calculation loop in Estrin's method. For the MPI version: Brute Force, Brute Force - Optimised and Horner's form execute on process rank 0 and Estrin's method is parallelised across all processes.

A single run of the test program will execute all 8 test cases and output the numeric results & average iteration times for each evaluation method in CSV format. The test program takes the following form in pseudo-code:

Algorithm 3.1 Pseudo-code for PolyEval Test Program

1. Set numTestCases = 8
2. Set testCases array = (5, 15, 30, 100, 500, 1000, 2000, 4000)
3. Do i = 1, numTestCases
   (a) numTerms = testCases(i)
   (b) Set x value of polynomial
   (c) Set n value of polynomial = numTerms
   (d) Generate random array of coefficients for this test case
   (e) Call all 4 evaluation methods
      i. Brute Force
      ii. Brute Force - Optimised
      iii. Horner's Form
      iv. Estrin's Method
   (f) Output numeric results for the evaluation methods in CSV form
   (g) Output average iteration times for the evaluation methods in CSV form
4. End do

An excerpt of the final two test cases from a typical test case run is shown below:

```
===============

Test Case            7
Number of terms:          2000

Brute Force                | Brute Force - Opt          | Horners Form               | Estrins Method
Results:
 3.40534834054662773E+084 ,  3.40534834054649616E+084 ,  3.40534834054649099E+084 ,  3.40534834054662730E+084 ,
Average Iteration Times:
 1.02996826171875000E-004 ,  1.82318687438964844E-003 ,  5.96046447753906250E-006 ,  3.81278991699218750E-002 ,

===============

Test Case            8
Number of terms:          4000

Brute Force                | Brute Force - Opt          | Horners Form               | Estrins Method
Results:
 1.99126351861883737E+167 ,  1.99126351861868339E+167 ,  1.99126351861868575E+167 ,  1.99126351861883737E+167 ,
Average Iteration Times:
 1.78098678588867188E-004 ,  7.24482536315917969E-003 ,  1.21593475341796875E-005 ,  0.14080405235290527       ,

===============
```

Chapter 4

Results and Analysis

4.1 Performance & precision of polynomial evaluation techniques

For each test case, the four methods of evaluation are executed: the two Brute force methods, Horner's form and Estrin's method. Estrin's method is evaluated using 16 threads using OpenMP as well as 1 thread, without OpenMP. Additionally. for comparison, the "Basic" method discussed above was included.

In terms of performance, the average execution times for the polynomial evaluation techniques are graphed and tabulated below. The execution times are quoted in nanoseconds.

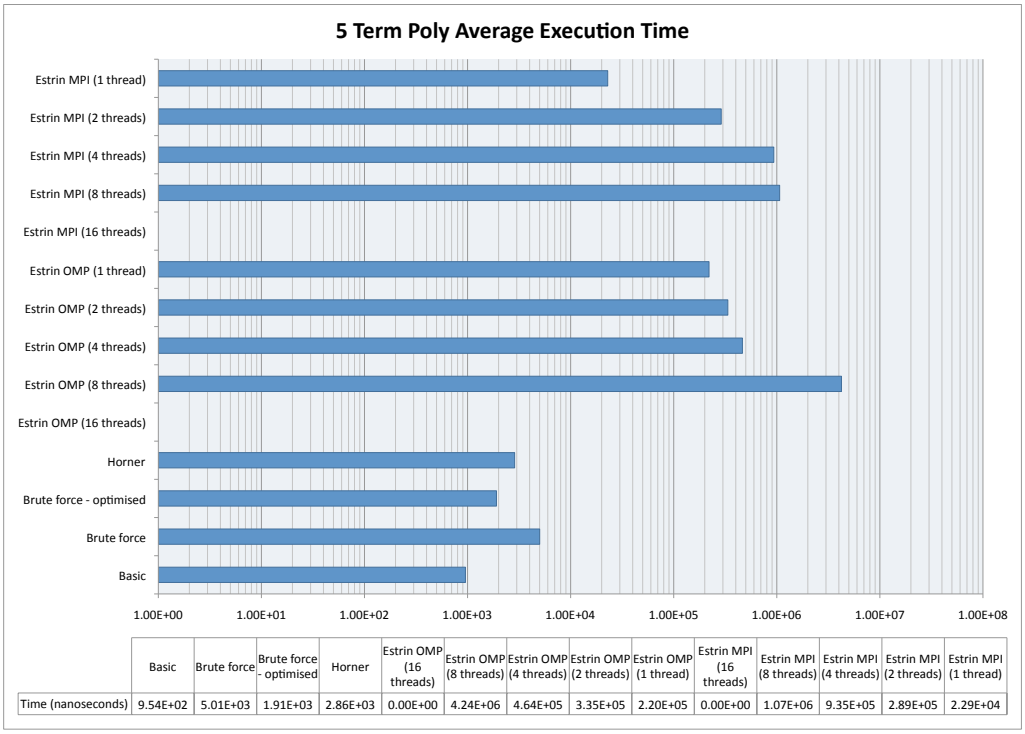
In terms of precision, the results from the polynomial evaluation techniques are compared to the result from MAPLE, a symbolic math package which supports "unlimited" precision. In order to compare the results from the different methods, the percentage difference with respect to the symbolic result was calculated in MAPLE.

Additionally, when evaluating test case 8, the 4000 term poly, symbolically in MAPLE, it was discovered that it was unable to interpret a 4000 term expression. Therefore, the polynomial had to be split into several chunks to allow MAPLE to interpret the expression and calculate the symbolic result.

Note: The precision graphs are unlabeled on the horizontal axis for clarity, due to the large number of decimal places in the quantities involved.

4.1.1 Test case 1: 5 term polynomial

4.1.1.1 Performance



	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	9.54E+02	5.01E+03	1.91E+03	2.86E+03	0.00E+00	4.24E+06	4.64E+05	3.35E+05	2.20E+05	0.00E+00	1.07E+06	9.35E+05	2.89E+05	2.29E+04

Figure 4.1: Test case 1 performance results

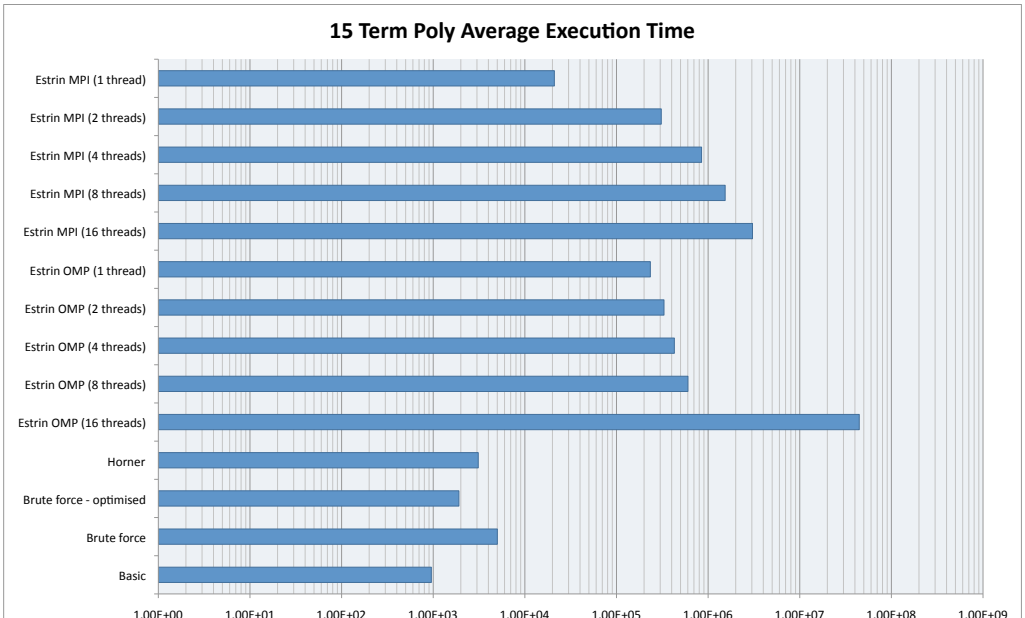
For a polynomial of degree 4, i.e. 5 terms, the fastest execution time is the “basic” method and the slowest is Estrin’s method using OMP on 8 threads. Estrin’s method executes faster using OMP than with MPI, with the exception of the 1 thread case.

There is no result shown for Estrin’s method using 16 threads as the size of the coefficient array is 8 (i.e. nearest power of two to 5 is 8). Therefore, it is not sensible to use 16 processors when at most 8 threads can be utilised at the first step of the calculation.

4.1.1.2 Precision

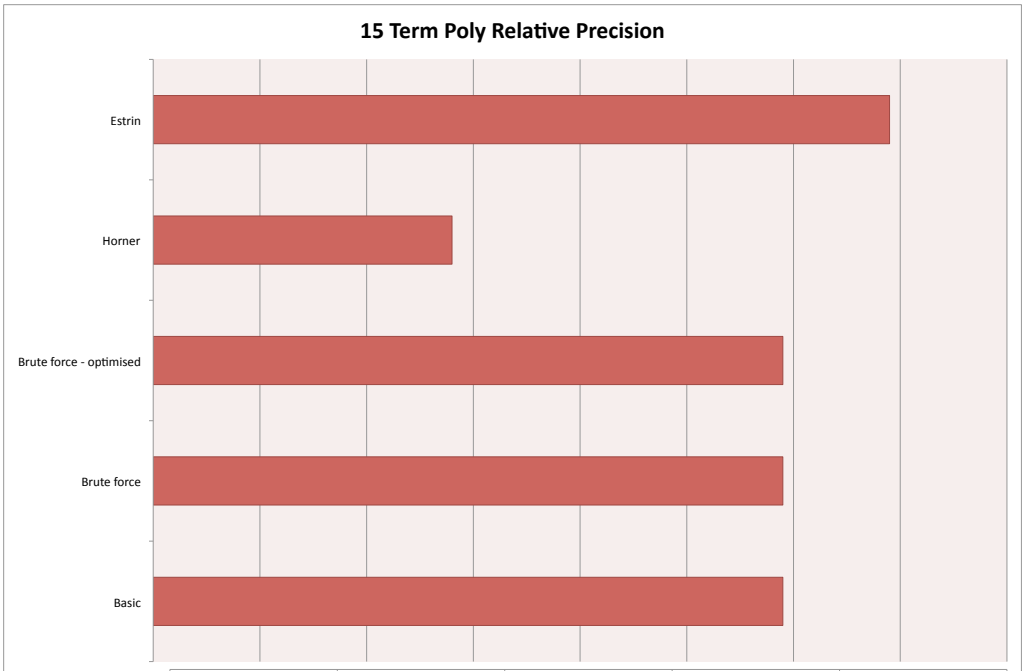
For the polynomial evaluated in Test case 1, all the evaluation methods produced exactly the same result to the accuracy of double precision floating point.

Comparing the result produced on Ness to that produced by MAPLE, there was a 6.50335E-06% difference between the two, with respect to the symbolically evaluated MAPLE figure.



	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	9.54E+02	5.01E+03	1.91E+03	3.10E+03	4.46E+07	6.02E+05	4.28E+05	3.30E+05	2.34E+05	3.05E+06	1.53E+06	8.48E+05	3.08E+05	2.10E+04

Figure 4.2: Test case 2 performance results



	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	2.0687560379E-05	2.0687560379E-05	2.0687560379E-05	2.0687560348E-05	2.0687560389E-05

Figure 4.3: Test case 2 precision results

4.1.2 Test case 2: 15 term polynomial

4.1.2.1 Performance

As can be seen in Figure 4.2, the fastest evaluation method for the 14th degree polynomial, i.e. 15 terms, is the “Basic” method, closely followed by the “Brute force - optimised” method; Horner’s form is slightly faster than the “Brute force” method. Estrin’s method using OMP on 16 threads is the slowest by a large margin. However, Estrin’s method using OMP on 2-8 threads is faster than using MPI, although on 1 thread the MPI version of the algorithm is faster.

4.1.2.2 Precision

For the 15 term uni-variate polynomial, Horner’s form was the most accurate as seen in Figure 4.3, with the two brute force methods and the basic method producing exactly the same result. Estrin’s method is the least accurate.

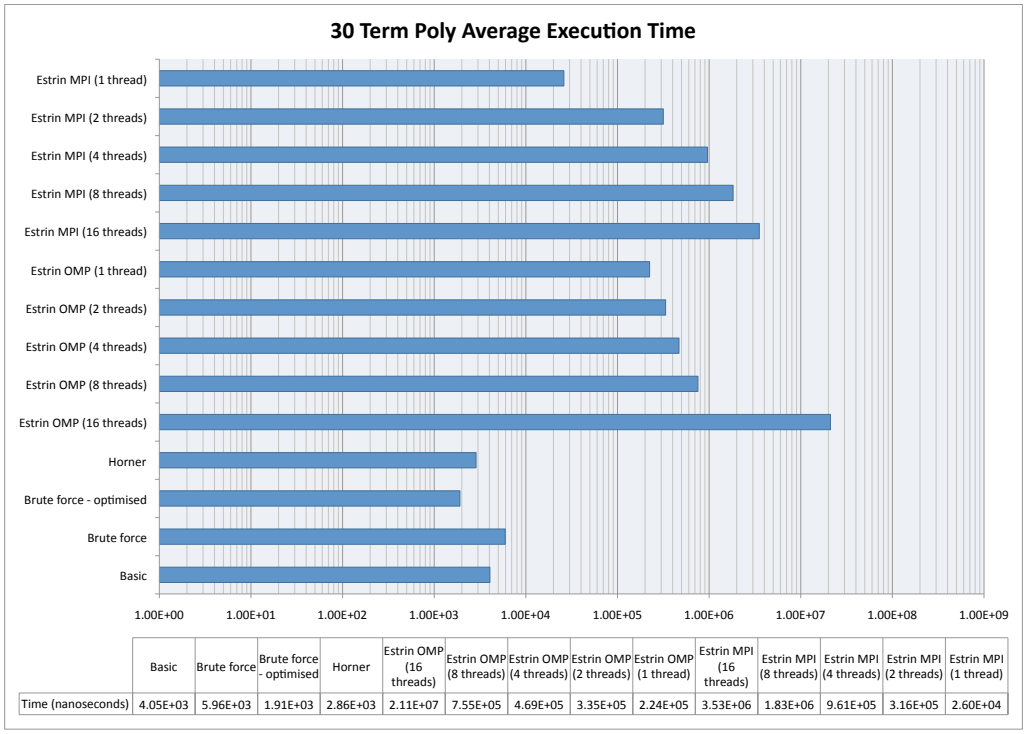
4.1.3 Test case 3: 30 term polynomial

4.1.3.1 Performance

For the 30 term i.e. 29th degree polynomial, the fastest evaluation method is “Brute force - optimised” as shown in Figure 4.4, closely followed by Horner’s form. As with the 5 and 15 term cases, Estrin’s method is notably slower; in the case of Estrin’s method using OMP on 16 threads, the difference in performance is many orders of magnitude.

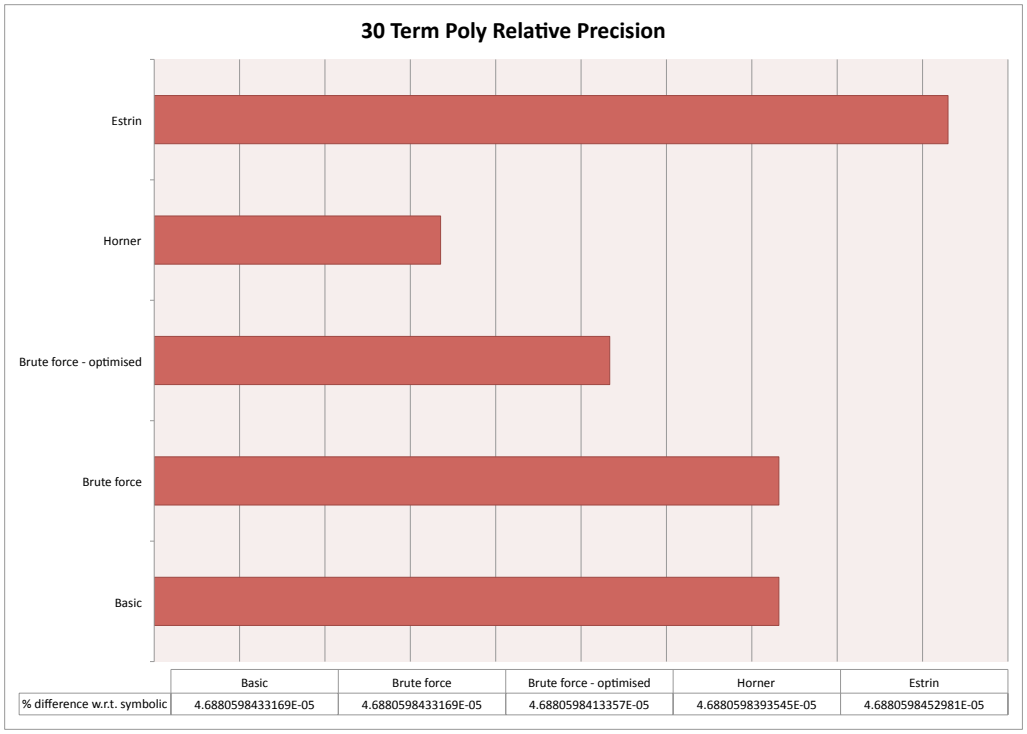
4.1.3.2 Precision

Horner’s form was found to be the most accurate method for this test case, with Brute Force - Optimised close behind. Estrin’s method is the least accurate. As can be seen from the data table Figure 4.5, the relative differences in the percentages are minimal; to find a difference in the figure, the reader must look beyond the 7th decimal place before the figures change.



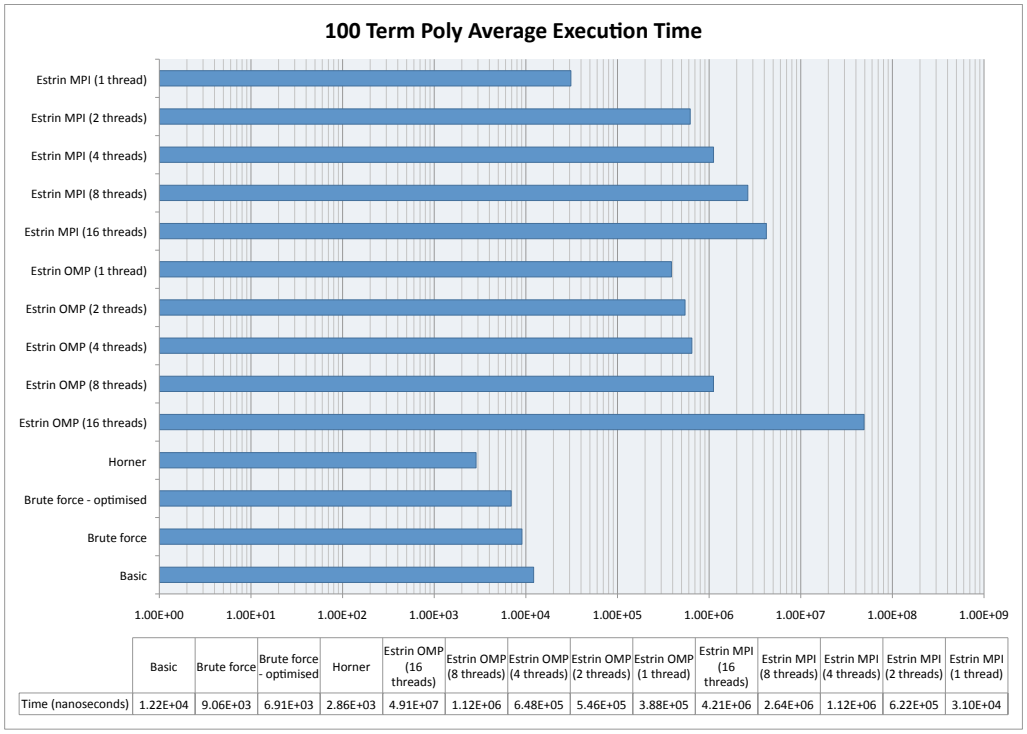
	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	4.05E+03	5.96E+03	1.91E+03	2.86E+03	2.11E+07	7.55E+05	4.69E+05	3.35E+05	2.24E+05	3.53E+06	1.83E+06	9.61E+05	3.16E+05	2.60E+04

Figure 4.4: Test case 3 results



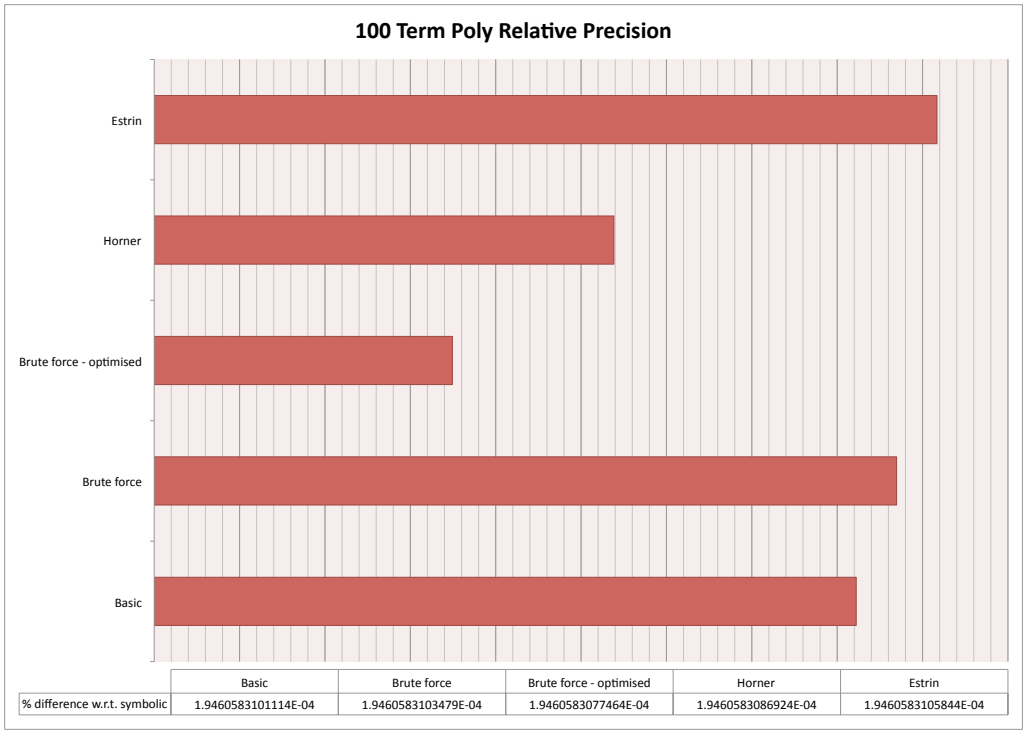
	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	4.6880598433169E-05	4.6880598433169E-05	4.6880598413357E-05	4.6880598393545E-05	4.6880598452981E-05

Figure 4.5: Test case 3 precision results



	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	1.22E+04	9.06E+03	6.91E+03	2.86E+03	4.91E+07	1.12E+06	6.48E+05	5.46E+05	3.88E+05	4.21E+06	2.64E+06	1.12E+06	6.22E+05	3.10E+04

Figure 4.6: Test case 4 results



	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	1.9460583101114E-04	1.9460583103479E-04	1.9460583077464E-04	1.9460583086924E-04	1.9460583105844E-04

Figure 4.7: Test case 4 precision results

4.1.4 Test case 4: 100 term polynomial

4.1.4.1 Performance

For the 100 term polynomial, Horner's form is the fastest evaluation method as shown in Figure 4.6. The same performance pattern with the previous test cases is observed with Estrin's method - both OMP and MPI versions.

4.1.4.2 Precision

As can be seen in Figure 4.7, Brute Force - Optimised is the most accurate of the methods, with Horner's form close behind. The Basic, Brute Force and Brute Force - Optimised methods are approximately similar. Again, the differences in these percentages are minimal.

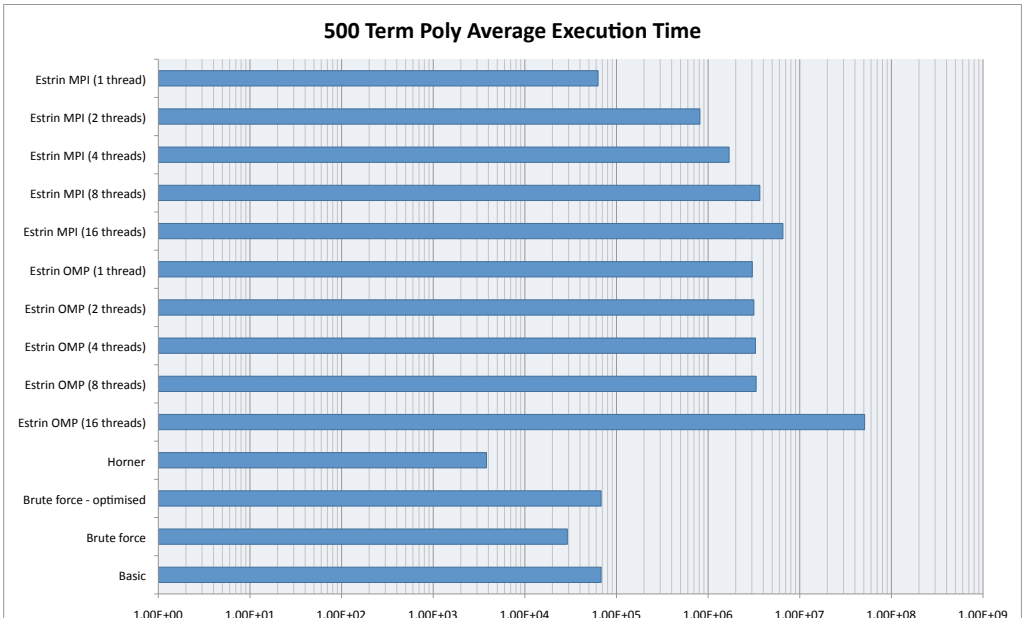
4.1.5 Test case 5: 500 term polynomial

4.1.5.1 Performance

The performance gain from using Horner's method for the 500 term polynomial is an entire order of magnitude over the "Basic" and "Brute force" methods. Estrin's method using MPI is faster than using the OMP version for less than 8 threads. Estrin's method using OMP on 16 threads is again records the largest average iteration time. The results are graphed in Figure 4.8. It should be noted that this is the first test case that Brute Force - Optimised is slower than Brute Force.

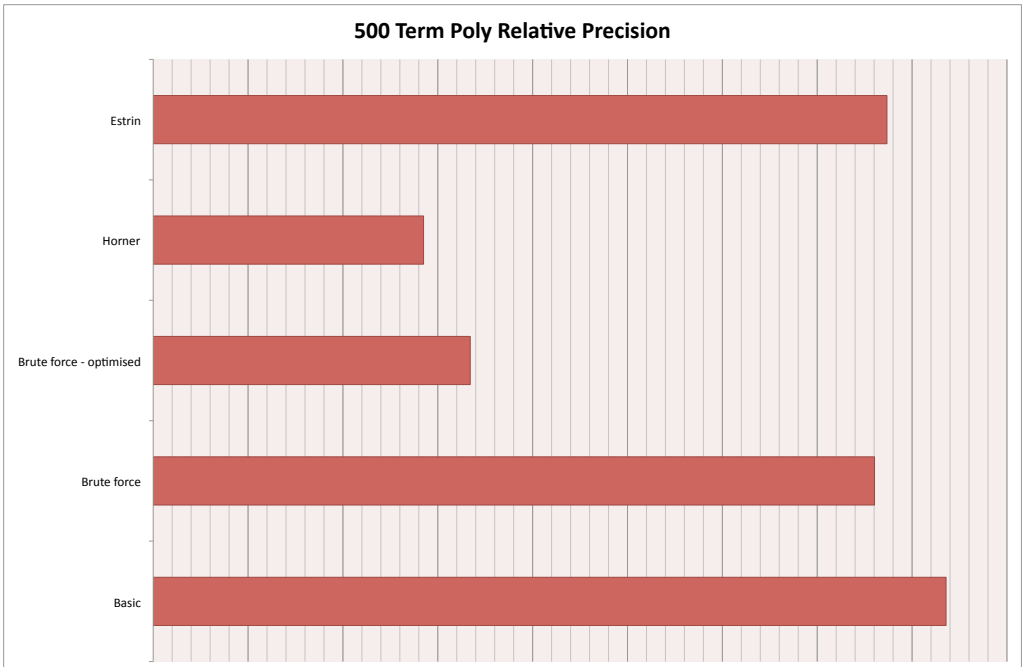
4.1.5.2 Precision

For test case 5, Horner's form and Brute Force - Optimised are the most accurate with the Basic method being the least accurate. Estrin's method and Brute Force perform approximately similarly. The results are shown in Figure 4.9.



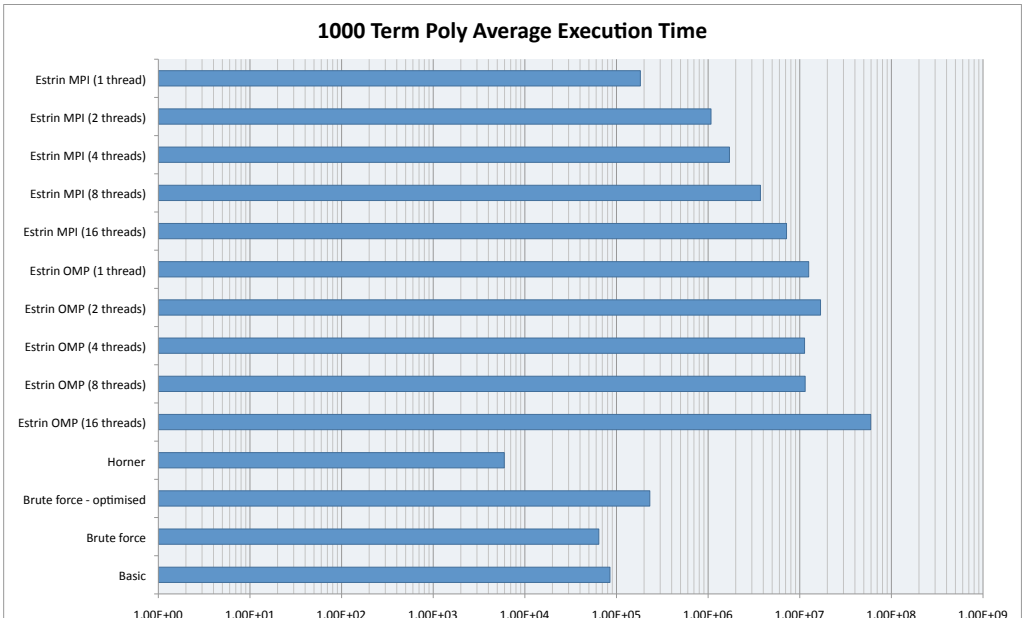
	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	6.79E+04	2.91E+04	6.79E+04	3.81E+03	5.10E+07	3.34E+06	3.28E+06	3.16E+06	3.03E+06	6.55E+06	3.67E+06	1.69E+06	8.14E+05	6.29E+04

Figure 4.8: Test case 5 results



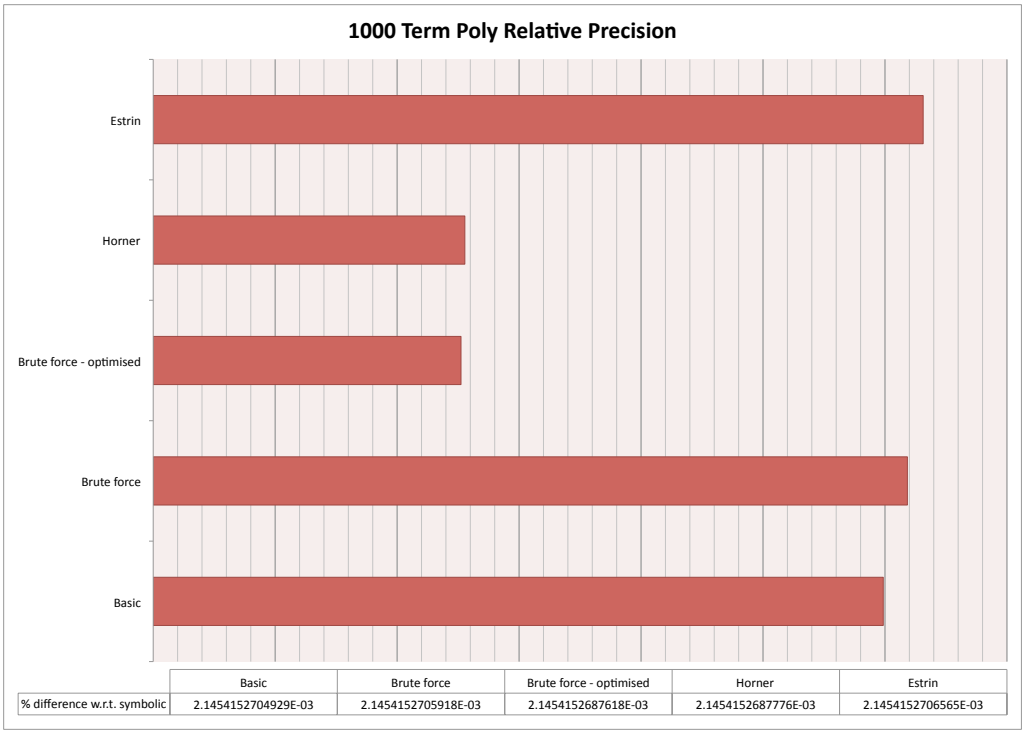
	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	1.0619885490716E-03	1.0619885489208E-03	1.0619885480684E-03	1.0619885479700E-03	1.0619885489470E-03

Figure 4.9: Test case 5 precision results



	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	8.49E+04	6.41E+04	2.31E+05	5.96E+03	5.95E+07	1.15E+07	1.13E+07	1.69E+07	1.25E+07	7.17E+06	3.72E+06	1.71E+06	1.08E+06	1.82E+05

Figure 4.10: Test case 6 results



	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	2.1454152704929E-03	2.1454152705918E-03	2.1454152687618E-03	2.1454152687776E-03	2.1454152706565E-03

Figure 4.11: Test case 6 precision results

4.1.6 Test case 6: 1000 term polynomial

4.1.6.1 Performance

For the 1000 term polynomial, as shown in Figure 4.10, Horner's form is the clear performance winner by an order of magnitude, with the Brute Force, Basic and Brute Force - Optimised methods being approximately similar. Estrin's method with MPI performs better than the OMP version, in line with the previous test cases.

4.1.6.2 Precision

As shown in Figure 4.11, for test case 6, the 1000 term polynomial, the Brute Force - Optimised method and Horner's form are the most accurate, with Estrin's method being the least accurate - although Basic and Brute Force are approximately similar.

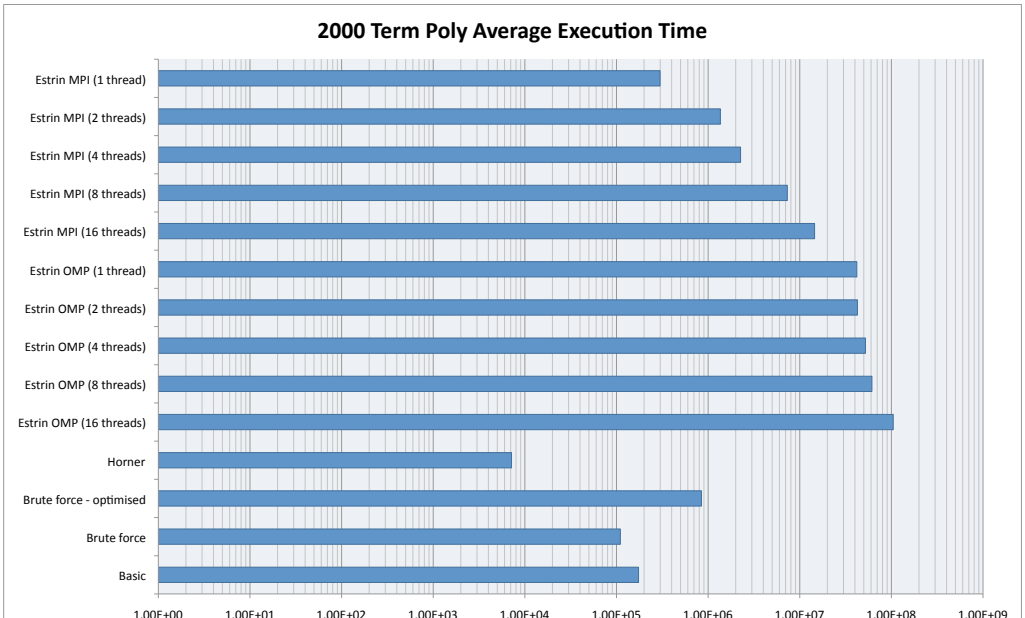
4.1.7 Test case 7: 2000 term polynomial

4.1.7.1 Performance

For the 2000 term poly, Horner's form is the clear performance winner as shown in Figure 4.12. Brute Force - Optimised is two orders of magnitude slower than Horner's form, with Basic & Brute Force being one order of magnitude slower. The MPI version of Estrin's method is faster then then OMP version by at least an order of magnitude.

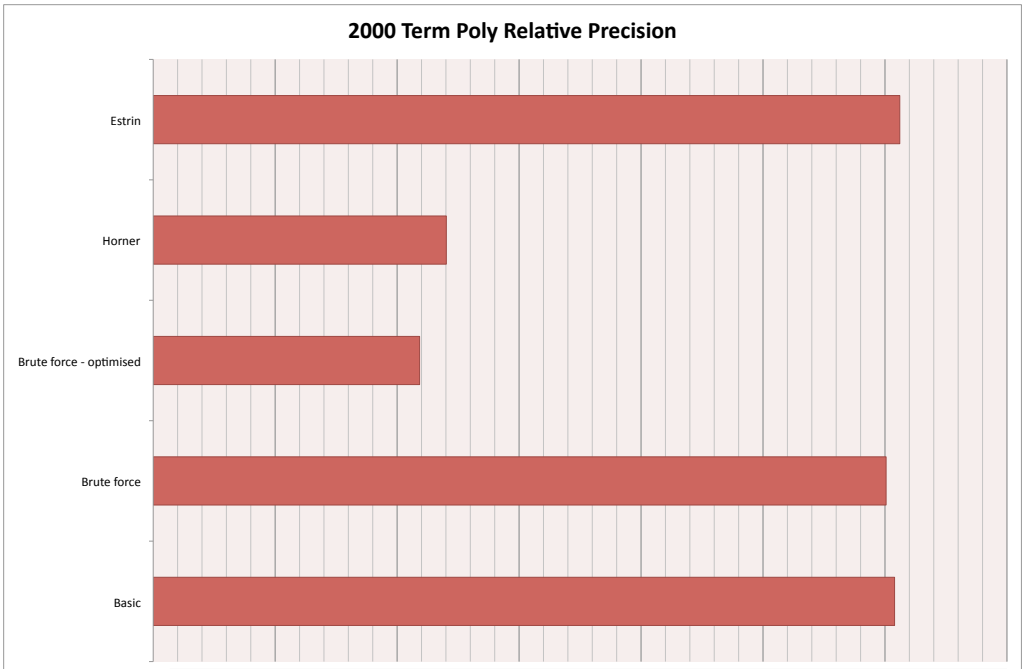
4.1.7.2 Precision

For test case 7, the 2000 term polynomial, Brute Force - Optimised and Horner's form are again the most accurate with the other three methods producing almost exactly the same answer. The results are graphed in Figure 4.13.



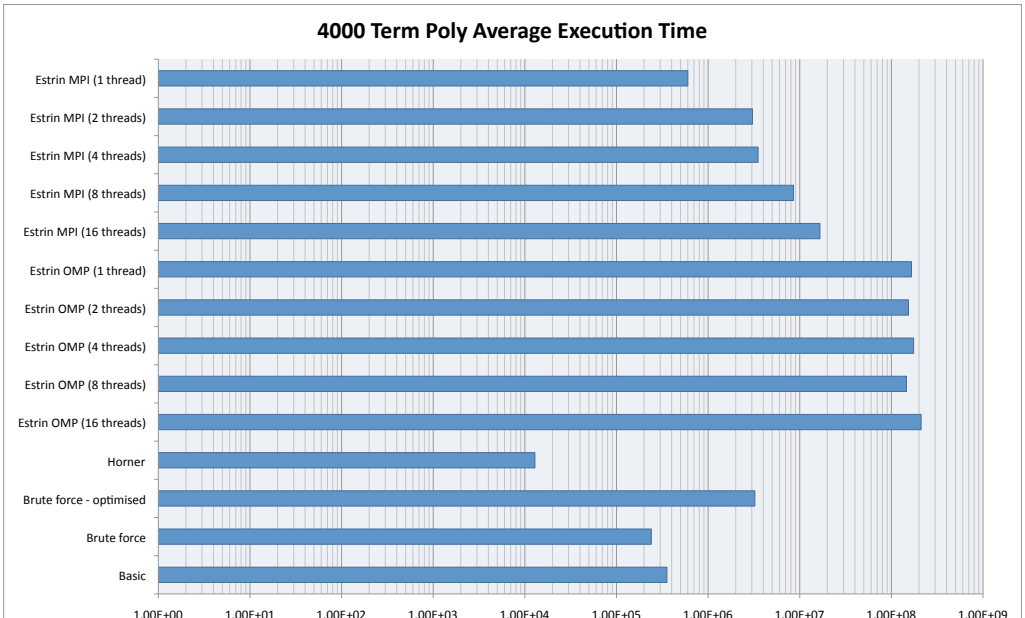
	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	1.74E+05	1.10E+05	8.46E+05	7.15E+03	1.05E+08	6.13E+07	5.21E+07	4.26E+07	4.20E+07	1.45E+07	7.31E+06	2.26E+06	1.36E+06	2.99E+05

Figure 4.12: Test case 7 results



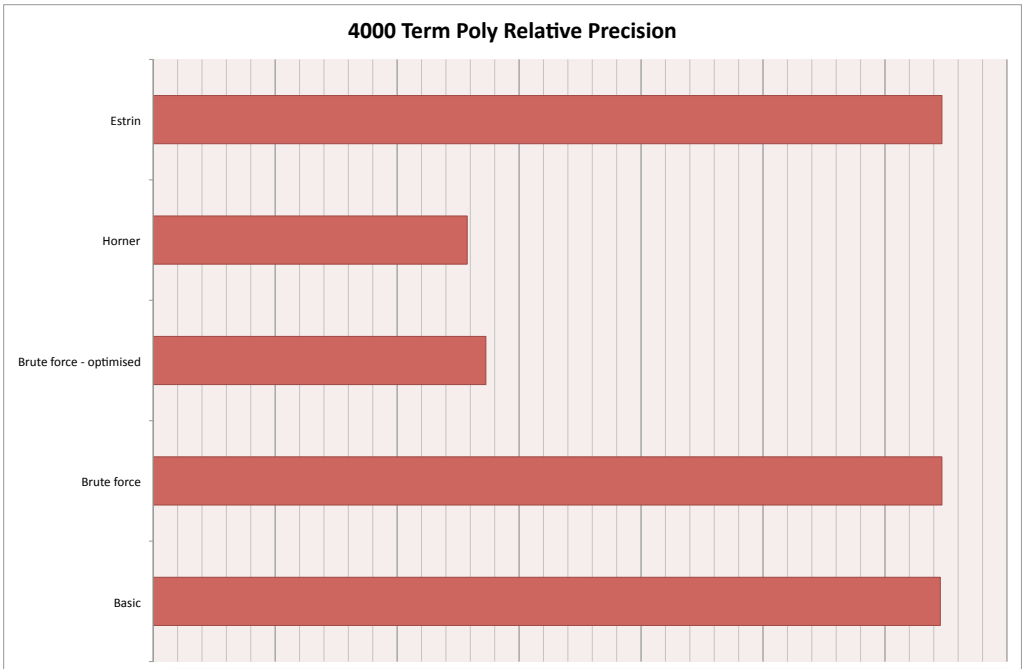
	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	4.3129265010786E-03	4.3129265010091E-03	4.3129264971842E-03	4.3129264974032E-03	4.3129265011213E-03

Figure 4.13: Test case 7 precision results



	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Time (nanoseconds)	3.55E+05	2.40E+05	3.24E+06	1.29E+04	2.11E+08	1.46E+08	1.75E+08	1.54E+08	1.66E+08	1.66E+07	8.55E+06	3.51E+06	3.05E+06	6.00E+05

Figure 4.14: Test case 8 results



	Basic	Brute force	Brute force - optimised	Horner	Estrin
% difference w.r.t. symbolic	8.6480858089073E-03	8.6480858089336E-03	8.6480858014554E-03	8.6480858011490E-03	8.6480858089336E-03

Figure 4.15: Test case 8 precision results

4.1.8 Test case 8: 4000 term polynomial

4.1.8.1 Performance

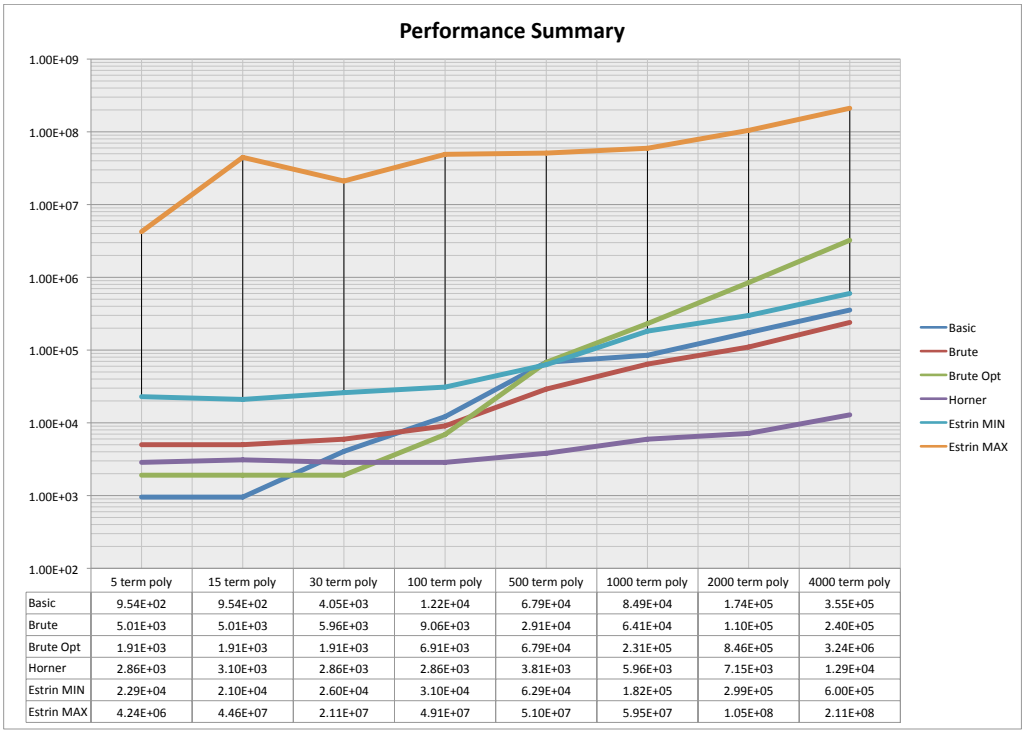
For the 4000 term polynomial, Horner's form is the performance winner by a large margin as shown in Figure 4.14. Estrin's method with MPI with 1 & 2 threads performs faster than Brute Force - Optimised. The Basic and Brute Force methods perform approximately the same. The Brute Force - Optimised method performs two orders of magnitude slower than Horner's form. Estrin's method parallelised using OMP perform similarly and as with other test cases, the MPI version performs better than the OMP version.

4.1.8.2 Precision

For the last test case, the 4000 term polynomial, Horner's form and Brute Force - Optimised are the most accurate, with the other 3 methods again producing approximately the same result. The results are graphed in Figure 4.15.

4.2 Summary of Results

4.2.1 Performance



	5 term poly	15 term poly	30 term poly	100 term poly	500 term poly	1000 term poly	2000 term poly	4000 term poly
Basic	9.54E+02	9.54E+02	4.05E+03	1.22E+04	6.79E+04	8.49E+04	1.74E+05	3.55E+05
Brute	5.01E+03	5.01E+03	5.96E+03	9.06E+03	2.91E+04	6.41E+04	1.10E+05	2.40E+05
Brute Opt	1.91E+03	1.91E+03	1.91E+03	6.91E+03	6.79E+04	2.31E+05	8.46E+05	3.24E+06
Horner	2.86E+03	3.10E+03	2.86E+03	2.86E+03	3.81E+03	5.96E+03	7.15E+03	1.29E+04
Estrin MIN	2.29E+04	2.10E+04	2.60E+04	3.10E+04	6.29E+04	1.82E+05	2.99E+05	6.00E+05
Estrin MAX	4.24E+06	4.46E+07	2.11E+07	4.91E+07	5.10E+07	5.95E+07	1.05E+08	2.11E+08

Figure 4.16: Evaluation Methods performance summary

Note: For clarity, only the maximum execution time and the minimum execution time are show for Estrin's Method. During testing, both the OMP and MPI versions of Estrin's method were tested on 16, 8, 4, 2 and 1 threads. To avoid cluttering the graph, the minimum and maximum values were taken as a summary. The minimum value in fact corresponds to the MPI version on 1 thread and the maximum to OMP on 16 threads.

As can be seen from Figure 4.16, for the 5 & 15 term polynomial the "Basic" method performs the best. For the 30 term polynomial, Brute Force - Optimised is the fastest. However, for the 100 term polynomial and above, Horner's form is the fastest performer by a clear margin.

4.2.2 Precision

In order to evaluate the precision of the methods, the results for each test case were compared to the symbolic results from MAPLE.

Unfortunately, due to the relative size of the quantities involved, even representing them on a logarithmic graph did not allow differences to be seen between the test cases for each method. Therefore, the results are presented in a qualitative tabular form.

Test Case	Most Accurate	⇒			Least Accurate
4000 poly	**Horner**	Brute-Opt	Basic	Brute	**Estrin**
2000 poly	**Brute-Opt**	Horner	Brute	Basic	**Estrin**
1000 poly	**Brute-Opt**	Horner	Basic	Brute	**Estrin**
500 poly	**Horner**	Brute-Opt	Brute	Estrin	**Basic**
100 poly	**Brute-Opt**	Horner	Basic	Brute	**Estrin**
30 poly	**Horner**	Brute-Opt	Brute = Basic		**Estrin**
15 poly	**Horner**	Basic = Brute = Brute-Opt			**Estrin**
5 poly	Basic = Brute = Brute-Opt = Horner = Estrin				

Table 4.1: Precision Summary

As can be seen from Table 4.1, Estrin's method is the least accurate method with Brute Force - Optimised and Horner's form being the most accurate consistently. With the 5 term polynomial, all the methods produced exactly the same result.

4.3 Performance Issues with Estrin's Method

As can be seen from the results presented above, the performance of Estrin's Method is disappointing. Executing both the OpenMP and MPI versions of Estrin's method with more threads is slower than running them only with one thread. As a general statement, using OpenMP & 16 threads is the slowest and MPI & 1 thread is the fastest of the Estrin's method times.

The overheads of parallelism are partly to blame. Profilers were used to investigate the performance problems with both versions of the code: namely OpenMP Profiler and VAMPIR. A number of the test cases were profiled and a summary of the results are presented below.

With the OpenMP version, the profiler identified that between 85% and 95% of the execution time was lost to overheads; of which roughly 50% was load imbalance and 50% thread management.

Similarly with the MPI version, VAMPIR identified that between 85% and 99% of the execution time was taken up by MPI communication, between 0.30% and 10% by the VAMPIR profiling API and between 0.70% and 5% by application code.

There is simply not enough work in evaluating a uni-variate polynomial to be worth parallelising the problem. The overheads of parallelisation are greater than the computations required on each thread. For each step of Estrin's method, the work of evaluating that step is spread across OpenMP threads or MPI processes. On each thread there is only a few operations that take place at each step: several loads, a multiply-add and a store. Since it is possible to complete these operations in the order of nano- or micro-seconds, the overheads simply dominate the execution time. Additionally, although parallelism is possibly beneficial in the first few steps of Estrin's Method, it is likely to become a hindrance in the final steps.

For say a 1024 term polynomial, evaluated on 16 processors, 10 steps are required to evaluate the result. At the first step there are $2^{10}/2/16 = 32$ results to calculate on each processor. However, by the 6th step, only 1 result ($2^5/2/16 = 1$) needs to be to calculated on each processor. Therefore, parallelism is no longer beneficial after this step and arguably not beneficial for several of the preceding steps either. This is because the amount of work required in latter steps is simply far too small to be worth parallelising.

However, the overheads of parallelism are not entirely to blame. If Estrin's method is run serially, that is without OpenMP or MPI, then performance is still roughly of the same order of magnitude as the parallel versions. Therefore, it is appropriate to attribute the performance problems to the algorithm itself.

As has been noted above, the advantages of using parallelisation diminish quickly as the calculation steps of Estrin's Method progress. Therefore, there is likely a point in the method where parallelism is no longer beneficial and the calculation would be more efficiently evaluated serially on a single processor.

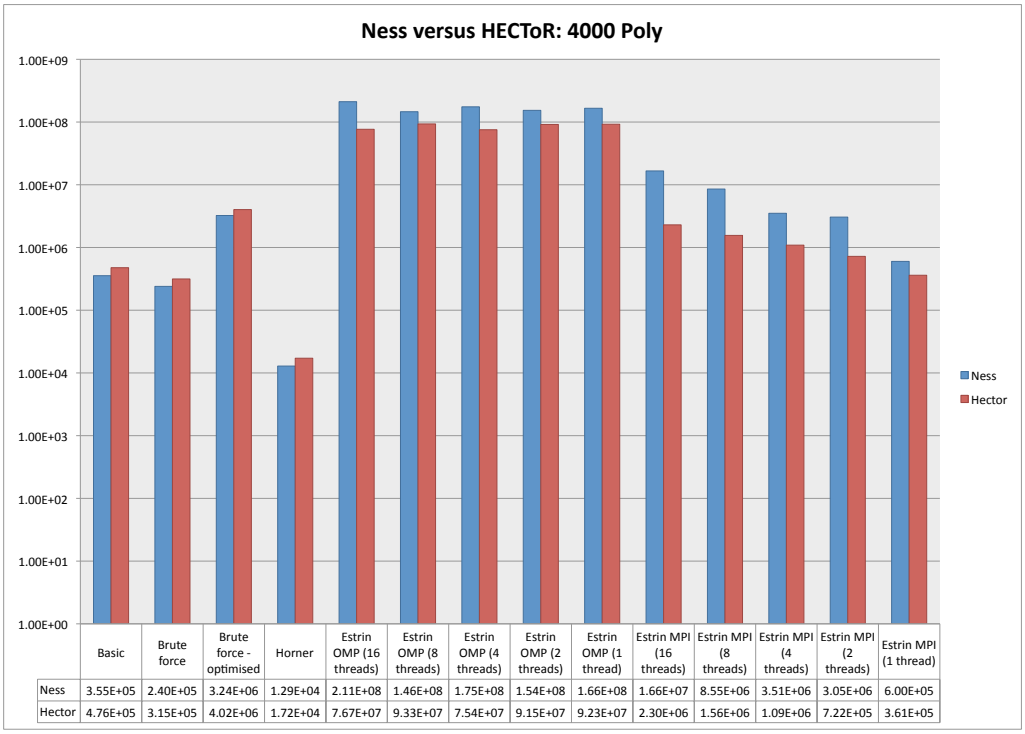
Essentially, if Estrin's Method was visualised as a binary tree, calculations could be evaluated in parallel at the thicker levels of the tree and calculations could be serial at the thinner levels. As the number of entries in each successive level of the binary tree reduces by a factor of two, there will come a point where there is simply not enough work available to warrant parallelism. The results of the parallelisation would be passed to a serial process to continue the final stages of calculation.

As has been mentioned in Section 2.1.3.2, when Estrin's method was proposed, it was not intended for multi-processor parallelisation. Rather it was targeted at wide processor pipelines and floating point units, making use of multiple pipelines to evaluate a wide expression tree, rather than multi-processor parallelisation. This work has shown that Estrin's method does not scale to the multi-processor level. However, when compared to serial Horner's form, Estrin's method executed in serial or in parallel are both vastly inferior in terms of performance.

4.4 Comparison of Performance on Ness and HECToR

Due to the disappointing performance of Estrin's method using both OpenMP and MPI, a comparison of performance between Ness and HECToR was made. HECToR (High End Computing Terascale Resource) is the UK's academic national super computer. The PolyEval code was run on the Phase 2b system, which as of writing comprises a 20 cabinet installation of a Cray XT6 system. Each compute node has two 12 core AMD Opteron 6100 processors and 32GB of memory. For the purposes of this investigation, all 24 cores were reserved on a single node for both the OpenMP and MPI versions of the code, although only utilising 16, 8, 4, 2 or 1 cores.

Test case 8, the 4000 term poly, was used for the comparison.



	Basic	Brute force	Brute force - optimised	Horner	Estrin OMP (16 threads)	Estrin OMP (8 threads)	Estrin OMP (4 threads)	Estrin OMP (2 threads)	Estrin OMP (1 thread)	Estrin MPI (16 threads)	Estrin MPI (8 threads)	Estrin MPI (4 threads)	Estrin MPI (2 threads)	Estrin MPI (1 thread)
Ness	3.55E+05	2.40E+05	3.24E+06	1.29E+04	2.11E+08	1.46E+08	1.75E+08	1.54E+08	1.66E+08	1.66E+07	8.55E+06	3.51E+06	3.05E+06	6.00E+05
Hector	4.76E+05	3.15E+05	4.02E+06	1.72E+04	7.67E+07	9.33E+07	7.54E+07	9.15E+07	9.23E+07	2.30E+06	1.56E+06	1.09E+06	7.22E+05	3.61E+05

Figure 4.17: Ness versus HECToR: 4000 Poly

As can be seen from Figure 4.17, execution times are broadly similar and unfortunately follow the same performance pattern; instead of a performance increase as the number of threads is increased, performance decreases with an increase in the number of threads.

Similarly, although the numeric results produced by HECToR were different to those produced by Ness, they were of comparable order i.e. Horner's form and Brute Force - Optimised were the most accurate with Estrin's method being the least accurate.

Chapter 5

Discussion and Conclusions

5.1 Discussion

Research into polynomial evaluation in computer codes goes back to the early days of computing. Both Horner's form and Estrin's method are mentioned in "The Art of Computer Programming" by Donald E. Knuth [3] from 1968. However, Estrin's method predates this work by 8 years, first proposed in 1960 by G. Estrin. Even earlier and long before the arrival of computers, Horner's form was first proposed in the 19th century to the Royal Society of London as part of a procedure for calculating polynomial roots. In fact, Issac Newton proposed a similar technique in a work titled "De Analysi per Æquationes Infinitas", originally written in 1669. Clearly therefore, the techniques have existed for some time.

One of the aims of this work was to draw together the body of research that exists in the area of polynomial evaluation and provide an overview of the relative merits of those methods. It has been shown that Horner's form is correctly held as the optimal method of evaluating polynomials, in terms of both performance and precision. The Brute Force - Optimised method created for this project was often more precise than Horner's form, but excepting the first few test cases, Horner's form out-performed that method.

The problems in successfully parallelising Estrin's method to achieve a performance speedup have been shown. Indeed, a performance decrease was the result of parallelising Estrin's method. Whilst the OpenMP version of Estrin's method was simpler to create than the MPI version, it was out-performed by the latter. However, not even the MPI version of Estrin's method was able to challenge the performance dominance of Horner's form in the 8 test cases presented above. The project has shown that Estrin's method does not scale to multi-processor parallelism; even in serial, Estrin's method is at least an order of magnitude or more slower than Horner's form.

As can be seen from the results, even for test case 8 (a polynomial of 4,000 terms), the time required to evaluate a polynomial is of the magnitude of micro or nanoseconds. Consequently, a code must be evaluating many millions of polynomials - or perhaps polynomials with numbers of terms in the millions - before polynomial evaluation becomes a performance bottleneck. The main reason the parallelisation of Estrin's method was such a demonstrable failure was because there is simply not enough work available to overcome the inherent overheads of parallelism. Therefore, highly efficient but serial methods of evaluation, namely Horner's form, are the most appropriate for the evaluation of polynomials. It is also convenient, that the highest performing evaluation method is more often than not the most precise, especially when evaluated on processors with support for fused multiply-adds.

Nonetheless, should parallelism in evaluating polynomials be desired, there is another option not investigated in this work which may be of use. If a code requires to evaluate, say, a thousand

polynomials, at each iteration, a worthwhile introduction of parallelism is still possible. Rather than parallelising the evaluation of each polynomial by spreading the calculation across multiple processors (such as in Estrin's method), it is possible to parallelise across all the polynomials required to be evaluated. For example, each processor could be assigned a chunk of the polynomials to be evaluated and then use Horner's form to evaluate them all. Thus, parallelism has been introduced to an otherwise serial problem, in a much more efficient manner than Estrin's method has been shown to be.

5.2 Conclusions

The principal aim of this project was to investigate the different methods of polynomial evaluation. However this dissertation was set in the context of improving the polynomial evaluation performance and precision of YaFFEMS. Unfortunately, the project was unable to achieve its intended aim of integrating the work back into YaFFEMS and thus use benchmark data sets to evaluate the different methods. This was primarily because of the problems encountered with Multi-variate Polynomials and Multi-variate Horner's Form. As a result of these difficulties, the project has been limited to uni-variate polynomials. This has limited its effectiveness with respect to the parent project. Improving polynomial evaluation performance and precision in YaFFEMS changed from being an aim of the project to being a motivation for the project.

The project has been successful in its intended aim of creating a polynomial evaluation library in Fortran. The PolyEval library uses a standard representation of a polynomial in the form of a Fortran derived type. Each of the four polynomial evaluation techniques (Brute Force, Brute Force - Optimised, Horner's form and Estrin's method) use this derived type. Estrin's method was parallelised using two methods: OpenMP and MPI.

The polynomial evaluation functions in the PolyEval library were tested with a range of polynomial test cases between 5 and 4000 terms long. The execution times were compared against a "Basic" method to provide a reference point. In terms of precision, the numeric results were compared for precision by evaluating the polynomial symbolically in MAPLE with "unlimited" precision. It was shown that the overheads of parallelism in Estrin's method are simply to great for the problem of polynomial evaluation. Similarly, an attempt to use GPGPUs for Estrin's method encountered the same problems i.e. that there is simply not enough computational work available to overcome the overheads of parallelisation.

The results show that Estrin's method does not scale to multi-processor level parallelism and performs slower in parallel than in serial. The project has provided a body of evidence which shows that for polynomials, a highly optimised serial evaluation method, in the guise of Horner's Form, is more advantageous to a parallelised method, in the form of Estrin's method. Additionally, the project has shown that Horner's form is more precise and faster than evaluating a polynomial by the "Basic" method (i.e. writing out the polynomial formula in full in the source code).

In the context of a High Performance Computing dissertation, it is disappointing that introducing parallelism to a problem does not increase performance. However, the project has shown that a highly optimised serial evaluation method is preferable to a parallel method when the quantity of computation is relatively small. Using parallelisation in the context of polynomials has been shown to be counter-productive as the amount of work available simply does not overcome the overheads of parallelisation.

However, by implementing a better method for polynomial representation than is currently used in YaFFEMS and showing that using specialised polynomial evaluation methods can increase performance & precision, the project has successfully achieved most of its aims.

5.3 Future Work

In order to progress the work presented here, it is critical that the challenge of multi-variate Horner's form is surmounted. By implementing this, PolyEval would be able to be used to evaluate any arbitrary polynomial in any number of variables, using a range of techniques. Currently, PolyEval can only evaluate multi-variate polynomials using Brute Force and Brute Force - Optimised. It is likely that multi-variate Horner's form would outperform the multi-variate Brute Force methods in PolyEval, as Horner's form did in the uni-variate case. This is because of the reduced number of arithmetic operations inherent in the factorisation used in Horner's form.

If PolyEval were able to evaluate arbitrary multi-variate polynomials using multi-variate Horner's form, then the library could be integrated with YaFFEMS and replace the current methods of polynomial representation and evaluation.

5.4 Project Review

In order to review a project, consideration must be given to the positive & negative aspects of it.

On the whole, the project has been successful and has achieved most of its stated aims. It has compared and contrasted the different polynomial evaluation methods and has shown that Horner's form is indeed the optimal method.

The performance of Estrin's method has been particularly disappointing, especially in the context of a High Performance Computing dissertation. It was expected that parallelising Estrin's method would have lead to faster execution times. However, it was ultimately shown that polynomial evaluation is simply not sufficiently computationally expensive enough to warrant parallelisation. The overheads of thread management, load imbalance and communication simply outweigh the amount of work available in even a particularly large polynomial. However, it has been proposed that if a code has a large number of polynomials to evaluate, instead of parallelising the execution of each polynomial, a more successful strategy would be to parallelise across the number of polynomials to be evaluated.

There were long delays associated with attempting to implement the Multi-variate Horner's form. It ultimately proved to be beyond myself, my supervisor plus several contacts with Maths backgrounds to understand how to implement the algorithms for Multi-variate Horner's form. This lead to considerable delays and severely weakened the project as a whole. The intention to integrate the work back into YaFFEMS was consequently relegated down to only a motivation for the project. Although Multi-variate Brute Force and Brute Force - Optimised were implemented, without Multi-variate Horner's form testing would have lacked the validity of the uni-variate testing.

With limited time available enforced limits upon what was practically achievable and the incremental approach taken to the project has proved successful. As discussed in Section 1.1, five goals were set out for this project. Unfortunately due to the problems in implementing Multi-variate Horner's form, the project was limited in scope to uni-variate polynomials. Therefore, it was not possible to integrate the work undertaken in this project back into YaFFEMS within this dissertation. Consequently, of the five goals set out for this dissertation, one was moved outside the scope of the project, one was partially achieved and three have been fully achieved,

Despite the problems encountered, the project aims have been largely achieved.

Bibliography

[1] Yaffems on google code. URL http://code.google.com/p/yaffems/.

[2] Robin Green. Faster math functions, March 2003. URL http://www.research.scea.com/gdc2003/fast-math-functions.html.

[3] Donald Ervin Knuth. *The Art of Computer Programming*. Addison-Wiley, Reading, Mass.„ 1968. ISBN 020103803 0201038021.

[4] Maplesoft website. URL http://www.maplesoft.com/products/maple.

[5] W. S. Dorn. A generalization of horner's rule for polynomial evaluation. In *Proceedings of the 1961 16th ACM national meeting*, pages 61.501–61.502, New York, NY, USA, 1961. ACM. doi: http://doi.acm.org/10.1145/800029.808522.

[6] Ian Munro and Allan Borodin. Efficient evaluation of polynomial forms. *Journal of Computer and System Sciences*, 6(6):625 – 638, 1972. ISSN 0022-0000. doi: DOI: 10.1016/S0022-0000(72)80033-3. URL http://www.sciencedirect.com/science/article/B6WJ0-4RFMC96-6/2/31f969d2dae0daded6f47dc914eca81e.

[7] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser Boston, 2006. URL http://www.springerlink.com/content/m01841/?p=38ebabb6e0b54e4ea4923af43a221ccc&pi=9.

[8] Jean-Michel Muller. *Handbook of Floating-Point Arithmetic*. Springer, 2010.

[9] J. Carnicer and M. Gasca. Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, 54(189):231–243, January 1990.

[10] J. M. Pena and Thomas Sauer. On the multivariate horner scheme. *Society for Industrial and Applied Mathematics*, 37(4):1186–1197, 2000.

[11] Martine Ceberio and Vladik Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *SIGSAM Bull.*, 38(1):8–15, 2004. ISSN 0163-5824. doi: http://doi.acm.org/10.1145/980175.980179.

The reference list above is generated & formatted automatically using LaTeX 2_ε and the default BibTeX numerical style.

Appendix A

Individual Performance Graphs

A.1 Basic

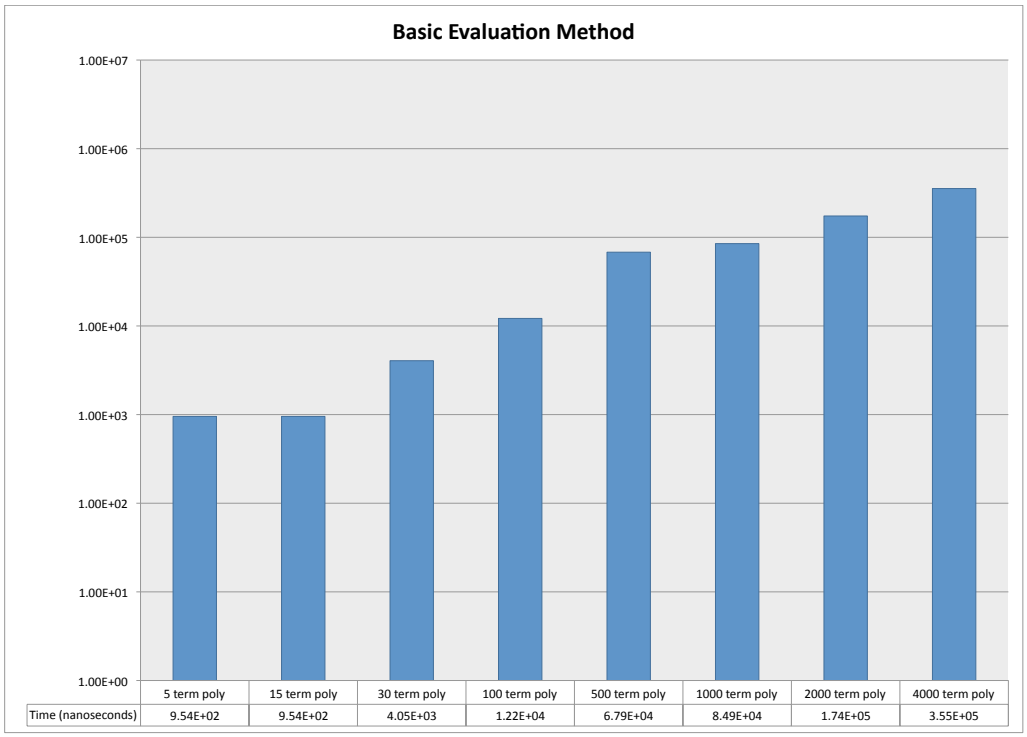


Figure A.1: Basic performance summary

A.2 Brute Force

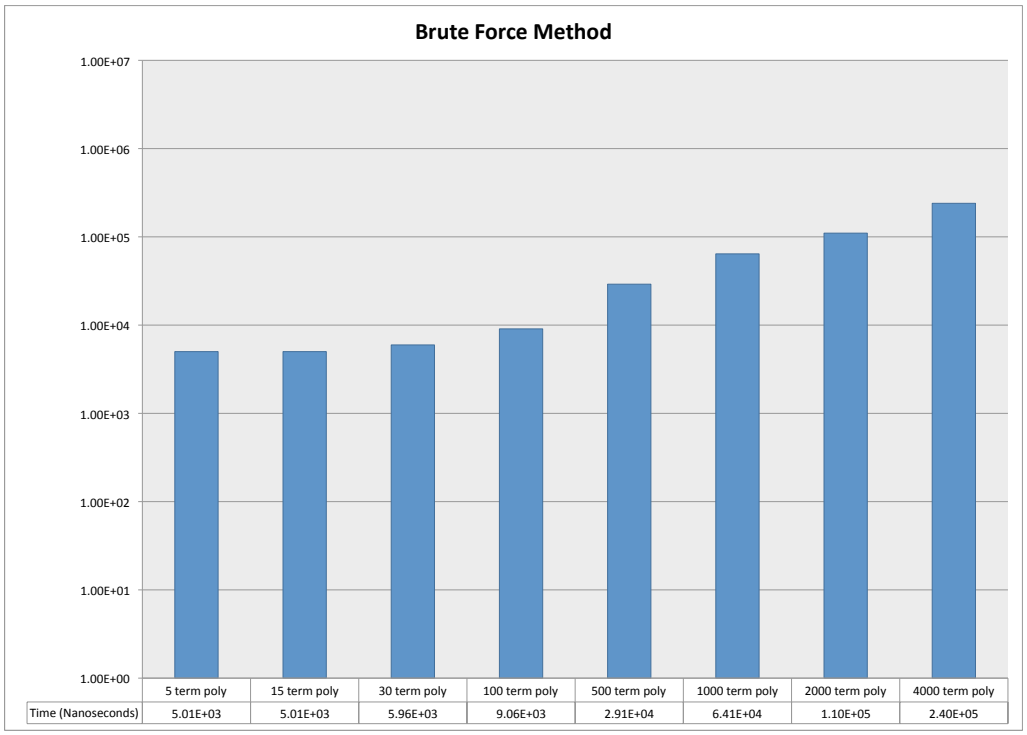


Figure A.2: Brute Force performance summary

A.3 Brute Force - Optimised

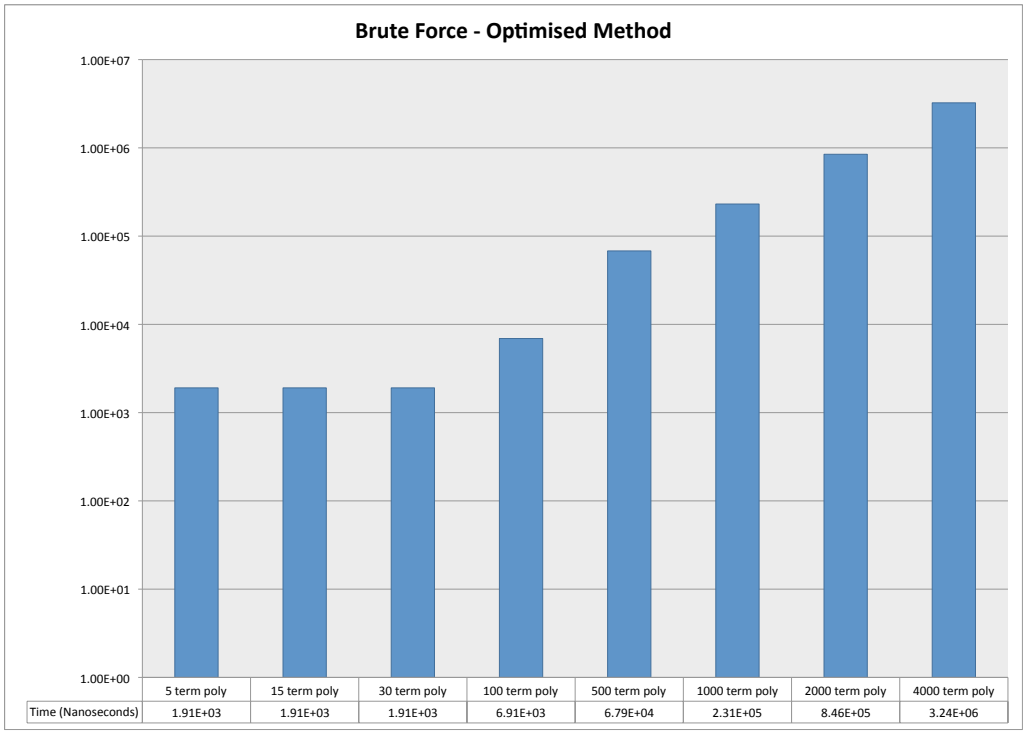


Figure A.3: Brute Force - Optimised performance summary

A.4 Horner's Form

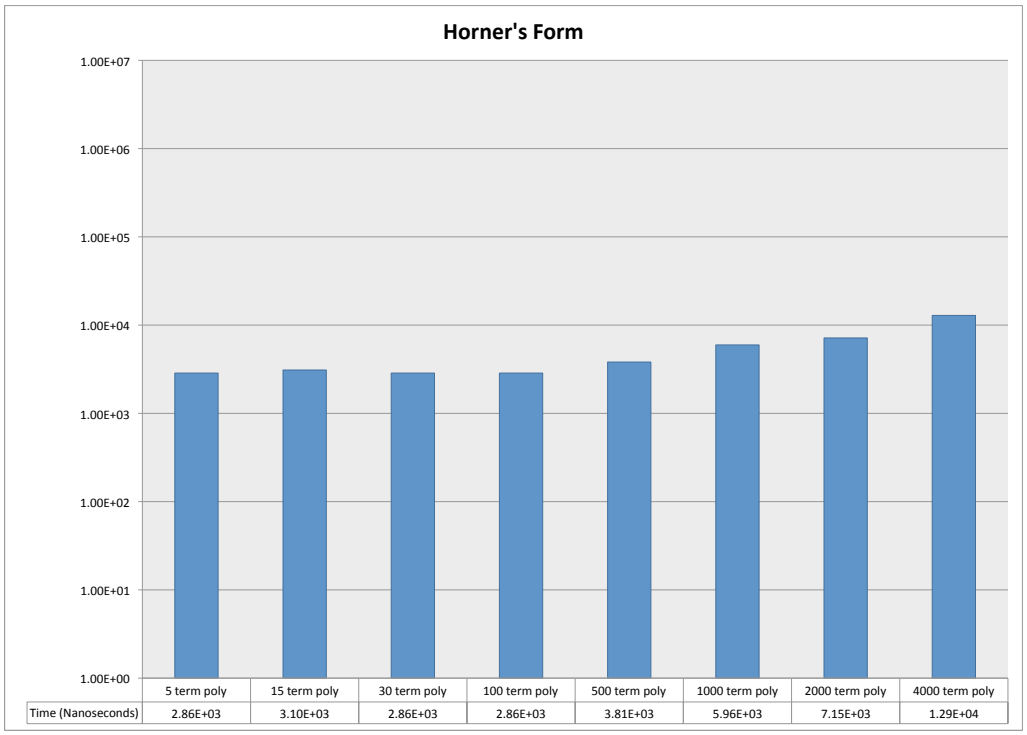


Figure A.4: Horner's Form performance summary

A.5 Estrin's Method

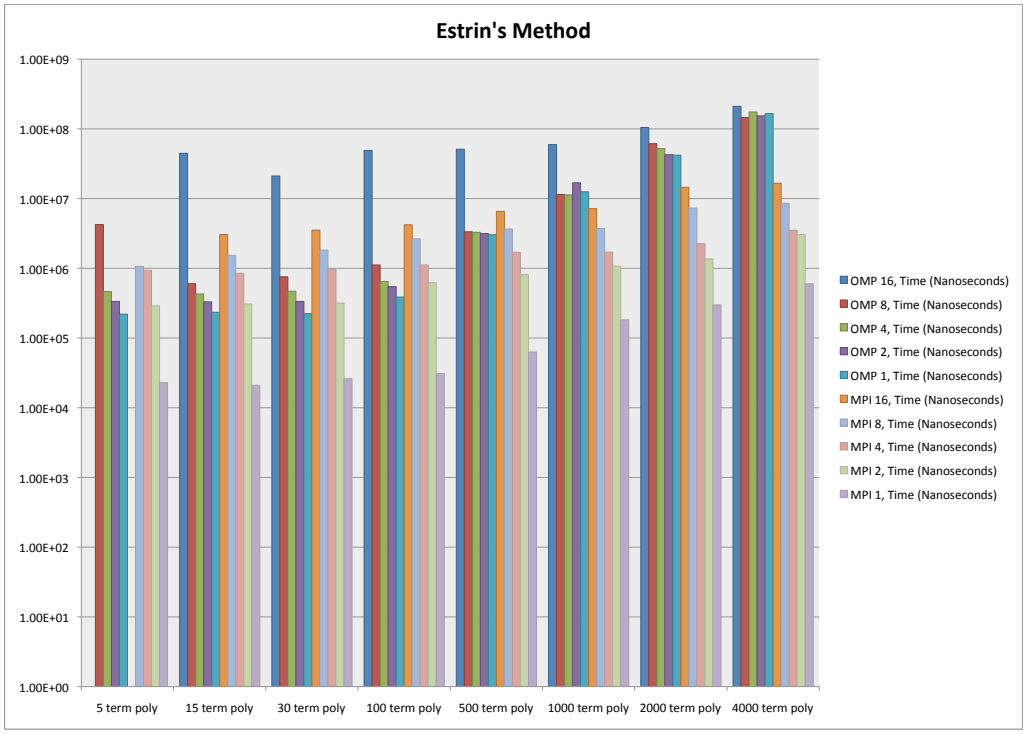


Figure A.5: Estrin's Method performance summary

Appendix B

Source Code

B.1 PolyEval Test Program

B.1.1 OpenMP version

```
program PolyEval_test
        use omp_lib
        use modPolyEval
        implicit none

        integer, parameter :: MAXITER = 1000
        real(kind=prec), allocatable, dimension(:) :: rndArray
        integer :: i, numTerms
        integer, parameter :: numTestCases = 8
        integer, dimension(numTestCases) :: testCases
        type(polynomial) :: poly

        testCases = (/5,15,30,100,500,1000,2000,4000/)

        do i = 1, numTestCases

                if (i > 1) then
                        deallocate(poly%f)
                end if

                numTerms = testCases(i)

                write(*,*) 'Test Case ', i
                write(*,*) 'Number of terms: ', numTerms
                write(*,*)

                poly%x = 1.1
                poly%n = numTerms
                allocate(poly%f(poly%n))

                call generateRandomArray(poly%n, rndArray)

                poly%f(:) = rndArray(:)
                deallocate(rndArray)

                call evaluationMethods(poly)

                write(*,*)
                write(*,*) '==============='
                write(*,*)

        end do

contains

        subroutine generateRandomArray(size, rndArray)
                integer, intent(IN) :: size
                real(kind=prec), allocatable, intent(OUT), dimension(:) :: rndArray
                integer :: seedSize,date(8)
                integer, allocatable  :: seed(:)
```

```
            call date_and_time(values=date)
            call random_seed(SIZE=seedSize)
            allocate( seed(seedSize) )
            call random_seed(GET=seed)
            seed = seed * date(8)       ! date(8) is milliseconds
            call random_seed(PUT=seed)

            allocate(rndArray(size))

            call random_number(rndArray)

            rndArray(:) = rndArray(:) * 10

        end subroutine generateRandomArray

        subroutine evaluationMethods(poly)

            type(polynomial), intent(IN) :: poly
            integer, parameter :: numMethods = 4
            real (kind=prec), dimension(numMethods) :: results, averageIterTime
            real (kind=prec) :: startTime, endTime, timeDiff
            integer :: i, loop

            !Brute force

            startTime = omp_get_wtime()
            do loop = 1, MAXITER
                    results(1) = Eval(poly)
            end do
            endTime = omp_get_wtime()

            timeDiff = endTime - startTime
            averageIterTime(1) = timeDiff/MAXITER

        !Brute force (optimised)

            startTime = omp_get_wtime()
            do loop = 1, MAXITER
                    results(2) = EvalOpt(poly)
            end do
            endTime = omp_get_wtime()

            timeDiff = endTime - startTime
            averageIterTime(2) = timeDiff/MAXITER

            !Horners Form

            startTime = omp_get_wtime()
            do loop = 1, MAXITER
                    results(3) = EvalHorner(poly)
            end do
            endTime = omp_get_wtime()

            timeDiff = endTime - startTime
            averageIterTime(3) = timeDiff/MAXITER

            !Estrins Method

            startTime = omp_get_wtime()
            do loop = 1, MAXITER
                    results(4) = EvalEstrin(poly)
            end do
            endTime = omp_get_wtime()

            timeDiff = endTime - startTime
            averageIterTime(4) = timeDiff/MAXITER

            write(*,*) 'Brute Force               |  Brute Force - Opt        |  Horners Form
                        |  Estrins Method'
            write(*,*) 'Results:'
            write(*,*) (results(i),',', i=1,numMethods)
            write(*,*) 'Average Iteration Times:'
            write(*,*) (averageIterTime(i),',', i=1,numMethods)

        end subroutine evaluationMethods

end program PolyEval_test
```

B.1.2 MPI version

```
program PolyEval_test
        use mpi
        use modPolyEval
        implicit none

        integer, parameter :: MAXITER = 1!000
        real(kind=prec), allocatable, dimension(:) :: rndArray
        integer :: i, numTerms
        integer, parameter :: numTestCases = 8
        integer, dimension(numTestCases) :: testCases
        type(polynomial) :: poly

        !MPI variables
        integer :: ierr, comm, rank, size

        !Initialise MPI
        comm = MPI_COMM_WORLD

        call MPI_Init(ierr)

    call MPI_Comm_Rank(comm, rank, ierr)

    call MPI_Comm_Size(comm, size, ierr)

        !Number of processes must either be 1 or a power of two
        if (size .ne. 1) then
            if ((iand(size, (size - 1))) .ne. 0) then
                if (rank==0) then
                        write(*,*) 'Number of processes must be a power of two'
                end if
                call MPI_Finalize(ierr)
                stop
            end if
    end if

        testCases = (/5,15,30,100,500,1000,2000,4000/)

        do i = 1, numTestCases

                if (i > 1) then
                        deallocate(poly%f)
                        deallocate(rndArray)
                end if

                numTerms = testCases(i)

                poly%x = 1.1
                poly%n = numTerms
                allocate(poly%f(poly%n))
                allocate(rndArray(poly%n))

                if (rank == 0) then

                        write(*,*) 'Test Case ', i
                        write(*,*) 'Number of terms: ', numTerms
                        write(*,*)

                        call generateRandomArray(poly%n, rndArray)

                end if

                call MPI_Bcast(rndArray, numTerms, MPI_DOUBLE_PRECISION, 0, comm, ierr)

                poly%f(:) = rndArray(:)

                call evaluationMethods(poly, rank)

                if (rank == 0) then
                        write(*,*)
                        write(*,*) '==============='
                        write(*,*)
                end if

        end do

        call MPI_Finalize(ierr)

contains
```

```
subroutine generateRandomArray(size, rndArray)
        integer, intent(IN) :: size
        real(kind=prec), allocatable, intent(OUT), dimension(:) :: rndArray
        integer :: seedSize,date(8)
        integer, allocatable  :: seed(:)

        call date_and_time(values=date)
        call random_seed(SIZE=seedSize)
        allocate( seed(seedSize) )
        call random_seed(GET=seed)
        seed = seed * date(8)       ! date(8) is milliseconds
        call random_seed(PUT=seed)

        allocate(rndArray(size))

        call random_number(rndArray)

        rndArray(:) = rndArray(:) * 10

end subroutine generateRandomArray

subroutine evaluationMethods(poly, rank)

        type(polynomial), intent(IN) :: poly
        integer, parameter :: numMethods = 4
        real (kind=prec), dimension(numMethods) :: results, averageIterTime
        real (kind=prec) :: startTime, endTime, timeDiff
        integer :: i, loop, rank

        if (rank == 0) then

                !Brute force

                startTime = MPI_Wtime()
                do loop = 1, MAXITER
                        results(1) = Eval(poly)
                end do
                endTime = MPI_Wtime()

                timeDiff = endTime - startTime
                averageIterTime(1) = timeDiff/MAXITER

            !Brute force (optimised)

                startTime = MPI_Wtime()
                do loop = 1, MAXITER
                        results(2) = EvalOpt(poly)
                end do
                endTime = MPI_Wtime()

                timeDiff = endTime - startTime
                averageIterTime(2) = timeDiff/MAXITER

                !Horners Form

                startTime = MPI_Wtime()
                do loop = 1, MAXITER
                        results(3) = EvalHorner(poly)
                end do
                endTime = MPI_Wtime()

                timeDiff = endTime - startTime
                averageIterTime(3) = timeDiff/MAXITER

        end if

        !Estrins Method

        startTime = MPI_Wtime()
        do loop = 1, MAXITER
                results(4) = EvalEstrin(poly, rank, size, comm)
        end do
        endTime = MPI_Wtime()

        timeDiff = endTime - startTime
        averageIterTime(4) = timeDiff/MAXITER

        if (rank == 0) then
                write(*,*) 'Brute Force                 | Brute Force - Opt         |
                    Horners Form              | Estrins Method'
                write(*,*) 'Results:'
                write(*,*) (results(i),',', i=1,numMethods)
```

```
                write(*,*) 'Average Iteration Times:'
                write(*,*) (averageIterTime(i),',', i=1,numMethods)
            end if

        end subroutine evaluationMethods

end program PolyEval_test
```

B.2 modPolyEval

```
module modPolyEval
    implicit none

        integer, parameter :: prec = kind(1.0d0)
        integer, parameter :: long = selected_int_kind(12)

        type polynomial
                integer :: n !Number of coefficients
                real (kind=prec), allocatable, dimension(:) :: f !Coefficients of polynomial
                real (kind=prec) :: x !Value of independent variable
        end type

        type polynomial_multi
                integer :: n !Number of coefficients
                integer :: m !Number of variables
                real (kind=prec), allocatable, dimension(:) :: f !Coefficients of polynomial
                real (kind=prec), allocatable, dimension(:) :: vars !Independent variables
                integer (kind=long), allocatable, dimension(:,:) :: powers
        end type

        !Evaluate by brute force
        interface Eval
                module procedure Eval, Eval_multi
        end interface

        !Evaluate by optimised brute force method
        interface EvalOpt
                module procedure EvalOpt, EvalOpt_multi
        end interface

    contains

    !Function to evaluate a univariate polynomial by brute force
        double precision function Eval(poly)
                type(polynomial), intent(IN) :: poly
                integer :: i
                real (kind=prec), dimension(poly%n) :: monomial

                do i = 1, poly%n-1
                        !Build up the monomial value by taking the coefficient
                        !and then multiplying by the variable values raised to the respective power
                        monomial(i) = poly%f(i)*poly%x**(poly%n-i)
                end do

                monomial(poly%n) = poly%f(poly%n)

                Eval = sum(monomial(:))

        end function Eval

        !Function to evaluate a univariate polynomial by brute force, with optimisations
        double precision function EvalOpt(poly)
                type(polynomial), intent(IN) :: poly
                integer :: i, j, numSteps
                real (kind=prec), dimension(poly%n) :: monomial
                real (kind=prec) :: x2

                x2 = poly%x*poly%x

                do i = 1, poly%n-1
                        monomial(i) = poly%f(i)
                        !Build up the power required from x^2 and x
                        !i.e. x^7 = x^2 * x^2 * x^2 * x
                        numSteps = (poly%n-i)/2
                        do j = 1, numSteps
                                monomial(i) = monomial(i) * x2
                        end do
                        if (mod(poly%n-i, 2) .ne. 0) then
                                monomial(i) = monomial(i) * poly%x
                        end if
                end do

                monomial(poly%n) = poly%f(poly%n)

                EvalOpt = sum(monomial(:))

        end function EvalOpt
```

```
    !Function to evaluate a multivariate polynomial by brute force
    double precision function Eval_multi(poly)
            type(polynomial_multi), intent(IN) :: poly
            integer :: i, j
            real (kind=prec), dimension(poly%n) :: monomial

            do i = 1, poly%n-1
                    !Build up the monomial value by taking the coefficient
                    !and then multiplying by the variable values raised to the respective power
                    monomial(i) = poly%f(i)
                    do j = 1, poly%m
                            monomial(i) = monomial(i)*poly%vars(j)**poly%powers(j,i)
                    end do
            end do

            monomial(poly%n) = poly%f(poly%n)

            Eval_multi = sum(monomial(:))

    end function Eval_multi

    !Function to evaluate a multivariate polynomial by brute force, with optimisations
    double precision function EvalOpt_multi(poly)
            type(polynomial_multi), intent(IN) :: poly
            integer :: i, j, k, numSteps
            real (kind=prec), dimension(poly%n) :: monomial
            real (kind=prec), dimension(poly%m) :: vars2 !variables, squared

            !Pre-calculate all the x^2, y^2 etc variables
            do i = 1, poly%m
                    vars2(i) = poly%vars(i)*poly%vars(i)
            end do

            do i = 1, poly%n-1
                    monomial(i) = poly%f(i)

                    do j = 1, poly%m
                            !Build up the power required from x^2 and x
                            !i.e. x^7 = x^2 * x^2 * x^2 * x
                            numSteps = (poly%powers(j,i))/2
                            do k = 1, numSteps
                                    monomial(i) = monomial(i) * vars2(j)
                            end do
                            if (mod(poly%powers(j,i), 2) .ne. 0) then
                                    monomial(i) = monomial(i) * poly%vars(j)
                            end if
                    end do
            end do

            monomial(poly%n) = poly%f(poly%n)

            EvalOpt_multi = sum(monomial(:))

    end function EvalOpt_multi

    !Function to evaluate a polynomial using Horner's form
double precision function EvalHorner(poly)
            type(polynomial), intent(IN) :: poly
            integer :: i

            !Evaluate using Horner's Method
            EvalHorner = poly%f(1)*poly%x
            do i = 2, poly%n-1
                    EvalHorner = (EvalHorner + poly%f(i))*poly%x
            end do
            EvalHorner = EvalHorner + poly%f(poly%n)

    end function EvalHorner

    !Function to evaluate a polynomial using Estrin's method
    double precision function EvalEstrin(poly)
            use omp_lib

            type(polynomial), intent(IN) :: poly
            real (kind=prec), allocatable, dimension(:) :: powers
            real (kind=prec), allocatable, dimension(:,:) :: coeff
            integer :: i,j,numsteps, shift, nearestpoweroftwo, npow2

            !Round up the number of coefficients to the nearest power of two
            nearestpoweroftwo = 2**ceiling(log(real(poly%n))/log(2.0d0))
            if (mod(poly%n, nearestpoweroftwo) .ne. 0) then
                    shift = nearestpoweroftwo-mod(poly%n, nearestpoweroftwo)
```

```
                npow2 = poly%n + shift
        else
                shift = 0
                npow2 = poly%n
        end if

        allocate(coeff(npow2, 0:npow2))
        coeff(:,:) = 0

        !Calculate the number of calculation steps required to reach the result
        numsteps = log(real(npow2))/log(2.0d0)
        allocate(powers(numsteps))

        !Build the powers array
        powers(1) = poly%x
        do i = 2, numsteps
                powers(i) = powers(i-1)**2
        end do

        !Populate the first (0) row of the coeff array with the polynomial coefficients
        coeff(1+shift:npow2, 0) = poly%f(:)

        !Evaluate using Estrin's Method
        do i = 1, numsteps

                !$omp parallel do default(none) shared(coeff, powers, npow2, i) private(j)
                do j = 1, npow2/(2**i)
                        coeff(j, i) = coeff(2*j-1, i-1)*powers(i)+coeff(2*j, i-1)
                end do
                !$omp end parallel do

        end do

        EvalEstrin = coeff(1, numsteps)

    end function EvalEstrin

end module modPolyEval
```

B.3 Estrin's Method in MPI

```
!Function to evaluate a polynomial using Estrin's method
double precision function EvalEstrin(poly, rank, size, comm)
        use mpi

        type(polynomial), intent(IN) :: poly
        integer, intent(IN) :: rank, size, comm
        real (kind=prec), allocatable, dimension(:) :: powers
        real (kind=prec), allocatable, dimension(:,:) :: coeff
        integer :: i,j,numsteps, shift, nearestpoweroftwo, npow2, ll, ul, source, sourcell,
             sourceul, sizeofchunk, arraywidthinuse
        integer :: effectivesize

        !MPI Variables
        integer, dimension(MPI_STATUS_SIZE) :: status
        integer :: ierr

        !Round up the number of coefficients to the nearest power of two
        nearestpoweroftwo = 2**ceiling(log(real(poly%n))/log(2.0d0))

        if (size > nearestpoweroftwo) then
                if (rank==0) then
                        write(*,*) 'Number of processes is greater than the width of the array.'
                        write(*,*) 'Num procs: ', size, ', numcoeffs: ', poly%n, '
                             nearestpoweroftwo: ', nearestpoweroftwo
                end if
                call MPI_Finalize(ierr)
                stop
        end if

        if (mod(poly%n, nearestpoweroftwo) .ne. 0) then
                shift = nearestpoweroftwo-mod(poly%n, nearestpoweroftwo)
                npow2 = poly%n + shift
        else
                shift = 0
                npow2 = poly%n
        end if

        !Calculate the number of calculation steps required to reach the result
        numsteps = log(real(npow2))/log(2.0d0)
        allocate(coeff(npow2, 0:numsteps))
        coeff(:,:) = 0
        allocate(powers(numsteps))

        !Build the powers array
        powers(1) = poly%x
        do i = 2, numsteps
                powers(i) = powers(i-1)**2
        end do

        !Populate the first (0) row of the coeff array with the polynomial coefficients
        coeff(1+shift:npow2, 0) = poly%f(:)

        effectivesize = size

        !Evaluate using Estrin's Method
        do i = 1, numsteps
                !Work out the number of elements in the current array row that are "active"/in use.
                !i.e. for a 512 coefficient poly, at step 1 there are 512 coefficients but at step
                     two there are only 256 coefficients
                !therefore, elements > 256 are unused i.e. zero
                arraywidthinuse = npow2/(2**(i-1))


                if (arraywidthinuse < effectivesize) then
                        !If the array width in use at this step is greater than the active/
                             effective number of MPI processes at this step
                        !then reduce the effective number of MPI processes.
                        !i.e. if at this step there is only 8 coefficients (i.e. array width of 8)
                             and there are 16 MPI processes,
                        !then reduce the effective number of MPI processes by a factor of two.
                        effectivesize = effectivesize/2
                end if

                !Size of chunk is the portion of the array row at this step assigned to each MPI
                     process
                sizeofchunk = arraywidthinuse/effectivesize

                if (i .ne. numsteps) then
                        ll = rank*sizeofchunk+1
```

```
                ul = ll+sizeofchunk-1
            else
                !If this is the last step, then lower limit and upper limit are 1, no
                    matter what the rank
                ll=1
                ul=1
            end if

            if (rank <= effectivesize - 1) then
            !If the rank is less than or equal to the active/effective number of MPI processes
                at this step then do the calc
                do j = ll, ul
                    coeff(j, i) = coeff(2*j-1, i-1)*powers(i)+coeff(2*j, i-1)
                end do
            end if

            if (i .ne. numsteps) then

                if (rank .ne. 0) then
                    if (rank <= effectivesize - 1) then
                        !Don't send data outside of the width of the array in use
                            at this step.
                        !Data outside the effective width of the array in use at
                            this step will only consist of zeros.

                        call MPI_SSend(coeff(ll:ul, i), ul-ll+1,
                            MPI_DOUBLE_PRECISION, 0, 0, comm, ierr)
                    end if
                else
                    do source = 1, effectivesize-1
                        !Only expect to recieve from processes with data inside the
                            effective width of the array.
                        sourcell = source*sizeofchunk+1
                        sourceul = sourcell+sizeofchunk-1

                        !Root recieves and inserts data from other processes into
                            coeff array
                        call MPI_Recv(coeff(sourcell:sourceul, i), ul-ll+1,
                            MPI_DOUBLE_PRECISION, source, 0, comm, status, ierr)
                    end do
                end if

                !Broadcasting updated data back to processes other than root
                call MPI_Bcast(coeff(:, i), npow2, MPI_DOUBLE_PRECISION, 0, comm, ierr)
            end if
        end do

        EvalEstrin = coeff(1, numsteps)

end function EvalEstrin
```