On the Fast Evaluation of Polynomials

P. Abascal Fuentes, F. Fueyo Tirado, D. García Quintas, J. Jiménez Meana and A. Palacio Muñiz

Department of Mathematics, University of Oviedo, Edificio Polivalente Campus de Viesques, 33203-Gijón, Asturias, España.

Authors’ contributions
This work was carried out in collaboration among all authors. All authors read and approved the final manuscript.

Article Information
DOI: 10.9734/JAMCS/2022/v37i630457

Abstract
Minimizing the computational cost of polynomial evaluation is a main problem in Computational Science. Horner’s algorithm efficiently solves the problem of evaluating a polynomial of degree n in time $O(n)$. It is also used to evaluate multivariate polynomials and, as an extension, matrix polynomials. If, in addition, a parallel execution of this method can be carried out, the computational cost of this problem would be further minimized. This makes it especially important to develop a strategy for parallel execution of Horner’s method for fast and efficient polynomial evaluation.

In this paper we present a parallelization of Horner’s method based on polynomial partitioning, as well as a modification of this method to exploit its advantages in the evaluation of sparse polynomials.

We also provide an analysis of the numerical error between the proposed parallel method and the classic algorithm.

Keywords: Parallel polynomial evaluation; parallel algorithms; sparse polynomials.

*Corresponding author: E-mail: abascal@uniovi.es;
1 Introduction

Polynomial evaluation, either dense or sparse, has been studied extensively due to its many applications. For instance, high degree polynomials are common in Coding Theory (cf. [1]). Error correcting codes are ubiquitous and their importance only increases as demand for higher speeds in computing naturally tend to introduce errors. The importance of minimizing algorithmic complexity in this field is mentioned in [2]. Any improvements based on simple principles are worthwhile.

Horner’s algorithm efficiently solves the problem of evaluating a polynomial of degree \( n \) in time \( O(n) \). This algorithm, however, does not take full advantage of any sparsity of \( p(x) \) since it always needs \( n \) multiplications and \( n \) additions even when the input polynomial has most of its coefficients equal to zero.

In [3], the authors, in order to achieve their objectives, propose a method for evaluating certain sparse polynomials.

In this paper, we present a modification of Horner’s method to exploit its advantages in the evaluation of all kinds of sparse polynomial.

Similarly to polynomials in one variable, Horner’s algorithm also optimizes the evaluation of multivariate polynomials as shown in [4] and [5] or [6]. In [7] authors studied the computational cost when using Horner’s algorithm to solve problems inherent to matrix calculations.

The use of parallel algorithms for the evaluation of high degree polynomials has already been proposed in different articles, see [8] or [9]. Consequently, the parallelization of Horner’s method can assist in addressing problems in the aforementioned fields.

The idea of partitioning has already been used in some previous papers, (see [10], [11] or [9]). Similarly, the method proposed in this paper partitions the input polynomial in order to exploit multiple computational units working in parallel.

Our partition proposal is simple because it is obtained directly by separating the polynomial into subpolynomials of fixed size so that each subpolynomial can be evaluated in parallel by Horner’s method.

In addition, the size of each subpolynomial can be established, adapting the partitioning to the size of the coefficients so that the per-core cache is not exceeded and thus further optimize the evaluation.

In [12], the authors already proposed an implementation for the parallel evaluation of sparse polynomials.

Here, we will see that polynomial partitioning combined with the proposed modifications to Horner’s algorithm will also allow us to benefit of both the advantages of the algorithm and the fact that we are working with sparse polynomials.

This paper is organized as follows: Section 2 reviews Horner’s algorithm for polynomial evaluation. Section 3 shows the proposed modifications to Horner’s method for its application to sparse polynomials. Section 4 presents the partitioning technique that will allow us to work in parallel improving
over the traditional sequential algorithm. Section 5 presents the performance results and compares them against the traditional Horner’s method. Section 6 analyzes the error bounds for both the classic and proposed Horner algorithms.

Finally, conclusions and further research will be presented in Section 7.

2 Horner’s Algorithm

A polynomial is defined as an expression of the form \( a_0 + a_1 x + \ldots + a_n x^n \). Scalars \( a_i \) are called coefficients, usually real or complex numbers; \( x \) is considered as a variable, that is, substituting an arbitrary number \( \alpha \) for \( x \), a well-defined number \( a_0 + a_1 \alpha + \ldots + a_n \alpha^n \) is obtained. The arithmetic of polynomials is governed by the usual rules for the sum and product operations.

Horner’s method evaluates \( P \) at a point \( \alpha \) as follows:

\[
P(\alpha) = a_0 + (a_1 + \ldots + (a_{n-1} + a_n \cdot \alpha) \cdot \alpha) \cdot \alpha
\]

where \( q_n = a_n \)
\( q_r = a_r + \alpha \cdot q_{r+1} \) \( r = n-1, \ldots, 1, 0 \)

Algorithm 2.1: Horner’s Method

1: \textbf{function} Horner(coefficients \( a_0, \ldots, a_n, \alpha \in \mathbb{R} \))
2: \hspace{1em} result \leftarrow a_n
3: \hspace{1em} \textbf{for} \( i = n-1 \rightarrow 0 \) \textbf{do}
4: \hspace{2em} result = result \cdot \alpha + a_i \hspace{1em} \triangleright n \text{ additions, } n \text{ products}
5: \hspace{1em} \textbf{end for}
6: \hspace{1em} \textbf{return} result \hspace{1em} \triangleright \text{ result } \equiv P(\alpha) = a_0 + a_1 \alpha + \ldots + a_n \alpha^n
7: \textbf{end function}

It performs \( n \) additions and \( n \) multiplications. Horner’s method is optimal for the evaluation of arbitrary polynomials inasmuch any other method would perform at least the same number of operations (see [11]).

3 Sparse Polynomial Evaluation

The traditional Horner method assumes all or most of the polynomial coefficients are non-zero, iterating over them sequentially in increments of one over the coefficient indices. That is, the algorithm does not distinguish if we have null coefficients or not, so it does not take advantage of the possible sparsity of a polynomial.

Ignoring the zero coefficients of a sparse polynomial saves unnecessary computation. Instead, we calculate the difference between consecutive pairs of exponents of non-zero coefficients.

So if \( p(x) = a_0 x^{r_0} + a_1 x^{r_1} + a_2 x^{r_2} + \ldots + a_n x^{r_n} \) we can consider the following sequence of exponent differences \( h_0 = r_0 \) and \( h_i = r_i - r_{i-1} \) for \( i = 1, 2, \ldots, n \). The coefficients \( (a_0, a_1, a_2, \ldots, a_n) \) are combined with \( (x^{h_0}, x^{h_1}, x^{h_2}, \ldots, x^{h_n}) \) as follows, in the usual Horner fashion:

\[
p(x) = \left[ a_0 + \left[ a_1 + \left[ a_2 + \ldots + (a_{n-1} + a_n \cdot x^{h_{n-1}}) \cdot x^{h_{n-1}} \right] \cdot x^{h_2} \right] \cdot x^{h_1} \right] \cdot x^{h_0}
\]
Therefore, a polynomial is uniquely determined by the sequences $(a_0, a_1, a_2, \ldots, a_n)$ and $(h_0, h_1, h_2, \ldots, h_n)$.

Algorithm 3.1 we propose a slightly modified version of Horner’s method for the evaluation of sparse polynomials, to be used in the recombination of the subproblems.

**algorithm 3.1 Modified Horner for Sparse Polynomials**

```plaintext
1: function HornerSparse(coefficients $a_0, \ldots, a_n$, $\alpha \in \mathbb{R}$, differences $h_0, \ldots, h_n$)
2: for all $i = 0 \rightarrow n$ do
3: \hspace{1em} powers[i] $\leftarrow \alpha^{h_i}$
4: end for
5: result $\leftarrow a_n \cdot powers[n]$
6: for $i = n - 1 \rightarrow 0$ do
7: \hspace{1em} result $= (a_i + result) \cdot powers[i]$
8: end for
9: return result
10: end function
```

In this regard, it should be noted that the evaluation is performed, unlike the classical method, on the powers of the degree differences $h_i$ as shown in the 'for' loop of lines 2 to 4.

## 4 Polynomial Partitioning

The idea of partitioning has already been used by the authors in the parallelization of methods for error correcting codes (see [13]). Also in [2] the authors perform a partition of a polynomial for its parallel evaluation that is generalized by the one we propose here. Based on this idea of partitioning we will separate the input polynomial into smaller pieces with which we will perform the evaluation process. By adequately partitioning the polynomial into sub-polynomials it becomes possible to evaluate them independently, and, consequently in parallel.

### 4.1 Partitioning method

The simplest partitioning mechanism would be to divide $P(x)$ into chunks of equal number of coefficients. If the data type of the coefficients is of constant size (such as `float` or `double`), that would result in equally-sized subproblems. However, if precision-preserving types are used for the coefficients, their sizes will vary.

A better way to partition the vector of coefficients of varying sizes is to have a maximum byte size per partition, linearly placing the dividing boundaries in such a way that the sum of the coefficient sizes for the partition be less than said maximum byte size. This uses the subproblem size as a proxy for its computational cost. Choosing the subproblem size as a function of the size of the system caches makes it possible to exploit the difference in speed of the memory hierarchy.

This article focuses on the case of constant-sized coefficients, that is, coefficients are represented with a constant number of bits, interpreted as floating point numbers.

Let $P(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial over $\mathbb{R}$, $\alpha \in \mathbb{R}$ the point where we want to evaluate the
polynomial and \( t \) the desired number of partitions.

- We construct the vector \( \vec{v} \) from the coefficients in \( P(x) \).
- We split \( \vec{v} \) into \((\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{t-1})\) consecutive components of \( \vec{v} \).
- The first \( t - 1 \) vectors, \( \vec{v}_i \) for \( i = 0, \ldots, t - 2 \), have length \( w = \lceil \frac{n+1}{t} \rceil \). The last one, \( \vec{v}_{t-1} \), has length \( r = (n \mod w) + 1 \).

The construction of these vectors is straightforward:

\[
\vec{v}_i = \begin{cases} 
(a_{iw}, \ldots, a_{(i+1)w-1}) & \text{if } i = 0, \ldots, t - 2, \\
(a_{iw}, \ldots, a_n) & \text{if } i = t - 1.
\end{cases}
\]

This partitioning is trivially performed in \( w \) steps.

### 4.1.1 Subproblem size selection

Let \( t \) be the number of parallel threads available and \( n \) be the degree of the polynomial. In order to balance load across subproblems evenly—which, given that all coefficient are of the same size, corresponds to the amount of computation—subproblem sizes should be bounded by \( \lceil \frac{n+1}{t} \rceil \). For example, it’s expected that for \( t = 3 \), performance would peak around subproblem sizes \( \approx \frac{n+1}{3} \) and diminish from that point onwards, as problem sizes become more and more dissimilar.

Note that for \( t = 1 \), the subproblem size is \( n \). In other words, no partitioning happens when executing over a single thread: the proposed method reduces to the usual iterative algorithm 2.1, with no performance impact. This allows the proposed method to be a drop-in replacement for the iterative one, with no downside in the single-threaded case and all the benefits of parallel execution if multiple threads are present.

### 4.2 Subproblem Evaluation

Let the values of \( \vec{v}_i \) conform the coefficients of a polynomial \( p_i(x) \). By considering \( y = x^w \), we can rewrite the original polynomial \( P(x) \) as:

\[
P(x) = p_0(x) + p_1(x)y + p_2(x)y^2 + \cdots + p_{t-1}(x)y^{t-1}
\]

Each of the \( p_i(x) \) subpolynomials can be evaluated at \( \alpha \) in parallel. Except for the last one, they are all \((w-1)\)-degree polynomials: each subproblem is expected to require about the same amount of computation. The optimal degree (“width”) \( w \) of each subproblem is likely to be a fraction of one of the system caches. Note that subproblems are, like the original problem, a dense polynomial. Each is evaluated using the iterative Horner algorithm described in 2.1.

The last step, combining the results for the \( t \) subproblems, is reduced to the evaluation of a \((t-1)\)-degree polynomial at \( y = x^w \). However, in this case, the polynomial is sparse.

Naturally, if \( t \) were large enough, the partitioning and parallel evaluation method could be applied again. Likewise for the actual evaluation of the \( p_i(x) \) subpolynomials.

In any event, once the degree of the subproblem is small enough, the base case evaluation is performed by means of the usual iterative version of Horner’s method.
4.3 Example of evaluation

Computation of \( p(\alpha) \) in the polynomial

\[
p(x) = a_0 + a_5x^5 + a_7x^7 + a_{13}x^{13} + a_{17}x^{17} + a_{23}x^{23} + a_{28}x^{28} + a_{36}x^{36} + a_{80}x^{80}
\]

- If we evaluate it by the Horner method, we would perform 160 operations. But most of them are additions by 0.
- If we take \( t = 3 \) threads and use the partitioning strategy, the size of each polynomial would be \( w = \lceil \frac{81}{3} \rceil = 27 \) and we would write the polynomial \( p(x) \) in the form:

\[
p(x) = a_0 + x(a_1 + x(a_2 + \ldots x(a_{25} + xa_{26}))\]) + \\
+ x^{27}\left( a_{27} + x(a_{28} + \ldots x(a_{52} + xa_{53}))\right) + \\
+ x^{54}\left( a_{54} + x(a_{55} + \ldots x(a_{79} + xa_{80}))\right)
\]

Finally, as the length of \( \alpha \) are:

\[
\alpha^{3}, \quad \alpha^{4}, \quad \alpha^{5} = \alpha^{1+4}, \quad \alpha^{6} = \alpha^{2+4}, \quad \alpha^{8} y \alpha^{44} = \alpha^{4+8+32}, \quad \text{then 9 operations are required.}
\]

To obtain the final result we have performed 25 operations.

- If we consider the coefficient sequence \( (a_0, a_5, a_7, a_{13}, a_{17}, a_{23}, a_{28}, a_{36}, a_{80}) \) and the exponent difference sequence \( (0, 5, 2, 6, 4, 6, 5, 8, 44) \) to define \( p(x) \) as a sparse polynomial, as in section 3, we can write \( p(x) \) in the next way:

\[
p(x) = a_0 + x^5(a_5 + x^2(a_7 + x^6(a_{13} + x^4(a_{17} + \\
+ x^6(a_{23} + x^5(a_{28} + x^8(a_{36} + x^{14}a_{80})))))))
\]

So we need calculate powers of \( \alpha \), in this case: \( \alpha^{2}, \quad \alpha^{4}, \quad \alpha^{5} = \alpha^{1+4}, \quad \alpha^{6} = \alpha^{2+4}, \quad \alpha^{8} y \alpha^{44} = \alpha^{4+8+32} \), then 9 operations are required.

Finally, as the length of \( p(x) \) is 9, we need 16 operations to evaluate \( p(\alpha) \) by Horner’s algorithm.

To obtain the final result we have performed 25 operations.

- Taking \( t = 3 \) threads the length of the subpolynomials is \( w = \lceil \frac{25}{3} \rceil = 3 \).

To be evaluated by Horner’s method in parallel, we can write the polynomial in the following form:

\[
p(x) = a_0 + x^5(a_5 + x^2a_7) + \\
+ x^{13}\left( a_{11} + x^4(a_{17} + x^6a_{23})\right) + x^{15}\left( a_{28} + x^8(a_{36} + x^{14}a_{80})\right)
\]

Thus, in a previous step to the parallel evaluation, the powers of \( \alpha \) that are going to be needed are: \( \alpha^{2}, \quad \alpha^{4}, \quad \alpha^{5} = \alpha^{1+4}, \quad \alpha^{6} = \alpha^{2+4}, \quad \alpha^{8}, \quad \alpha^{13} = \alpha^{1+4+8}, \quad \alpha^{15} = \alpha^{1+2+4+8} y \alpha^{44} = \alpha^{4+8+32} \) and 14 operations are required.
And each thread evaluates each $p_i(x) = \beta_i$ in 4 operations. Finally, the last evaluation

$$p(\alpha) = \beta_1 + \alpha^{13}(\beta_2 + \alpha^{15}\beta_3)$$

is achieved by 2 sums and 2 multiplications, so we have performed $3 \cdot 4 + 14 + 4$ operations. But at runtime this is equivalent to 22 operations.

5 Results

The following results were obtained on a dual socket 4-core Intel Xeon L5420 running at 2.50 GHz, for a total of eight cores. The benchmarking as well as the rest of the source code is available at: https://github.com/dgquintas/ParallelHorner under an Apache v2 license.

The test polynomial is the Taylor expansion of the exponential function, evaluated at $\alpha = 2.2$.

5.1 Efficiency

As described in section 5, the number of physical cores in the test system is 8, 4 per CPU socket. The efficiency of a parallel algorithm is its speedup relative to a baseline normalized by the number of threads. In the following results, the baseline is the usual iterative version of the Horner algorithm, as described in 2.1. Fig.1 and 2 present the efficiency results for different degrees $n$, number of threads $t$ and partition size as a percentage of $n$. The horizontal axis represents the subproblem size as a percentage of $n$ (see section 4.1.1 for details).

As expected, performance peaks for subproblems of size $n/t$, which corresponds to $100/t\%$ in terms of the values on the horizontal axis. See section 5.1.2 below.

![Fig. 1. Efficiency for $n = 1000$](image_url)

5.1.1 Maximum efficiencies

The main appeal of the proposed method is its use in parallel environments. Executions over a single thread reduce to the usual iterative algorithm (see section 4.1.1). Efficiency results are highly dependent on the architecture of the system, especially with respect to the organization of the per-core caches. Fig. 3 presents the maximum efficiency per number of threads.
Efficiency increases with the problem size (polynomial degree). At $n = 4000$, efficiency is $\geq 50\%$ for any number of threads backed by a physical core. For the number of threads available within a single CPU socket (in our case, 4), which minimizes the contention on the system bus, efficiency surpasses 50% for polynomials of degree $\geq 1000$.

5.1.2 Optimal Subproblem Size

Table 1. empirically shows the intuition, presented in section 4.1.1, that equally sized subproblems scale best. Note how, once the degree has become sufficiently large for the number of working threads $t$, the optimal subproblem size converges to 100%/t.

6 Error analysis

In [?], the author studied the error in the application of the Horner algorithm. In this section we analyze the error incurred when applying the Horner method in both its classic and parallel versions, but in a different way.
Table 1. Partition Size (as a % of \(n\)) for best efficiency

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>54</td>
<td>56</td>
<td>56</td>
<td>58</td>
<td>58</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>200</td>
<td>56</td>
<td>56</td>
<td>56</td>
<td>56</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>500</td>
<td>52</td>
<td>34</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>18</td>
</tr>
<tr>
<td>1000</td>
<td>52</td>
<td>34</td>
<td>24</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>2000</td>
<td>54</td>
<td>32</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>3000</td>
<td>54</td>
<td>32</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>4000</td>
<td>52</td>
<td>34</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>

Firstly, we assume that, computationally, each algebraic operation of a process is subject to a relative error bounded by \(\epsilon\), that is,

\[
| (a \perp b) - (a \perp b)^* | \leq \epsilon |(a \perp b)| \leq \epsilon (|a| \perp |b|) \tag{6.1}
\]

where \(a \perp b\) represents the exact value of the operation, \((a \perp b)^*\) the computed result and \(\perp\) the arithmetic operation + or \(\cdot\).

6.1 Some previous results

Now, we define some algebraic expressions that are going to be very useful hereafter.

**Definition 6.1.** Let \(a_0, a_1, \ldots, a_n\) be real numbers. We define iteratively a basic algebraic expression \(Q_n\) with \(n\) operations as:

- \(Q_1 = a_0 \perp_1 a_1\).
- for \(2 \leq k \leq n\), \(Q_k = Q_{k-1} \perp_k a_k\).

where \(\perp_k\) is the arithmetic operation + or \(\cdot\).

Now we can define the algebraic expressions which are obtained as compositions of basic algebraic expressions.

**Definition 6.2.** The algebraic expressions are defined recursively as follows:

- The basic algebraic expressions are algebraic expressions.
- If \(Q\) and \(R\) are algebraic expressions with \(n\) and \(k\) operations, respectively, then \(Q \perp R\) is an algebraic expression with \(n + k + 1\) operations.

**Example 6.1.** Given the polynomial \(P(x) = 3x^3 + 4x^2 - 2x + 1\), if we want evaluate it at \(x = \alpha\), we could use either the monomial form or the Horner method.

We can interpret the evaluation using the Horner method as a basic algebraic expression, however, if we use the monomial form the evaluation is an algebraic expression.

The first column shows eight operations need be performed for the monomial form (3 operations from \(Q_3\), 1 from \(R_1\), 1 from \(S_1\), and 3 from \(T_3\)), while only six are required for Horner’s method.

**Remark 6.1.** Given \(Q\) an algebraic expression, we consider the following notation:

- \(Q^*\) will be the computed output we obtain when we evaluate \(Q\), that is,
Given Definition 6.3. Let $n \in \mathbb{N}$, let $\mu_n$ denote the function $\mu_n : \mathbb{R}^+ \rightarrow \mathbb{R}$ defined as: $\mu_n(\epsilon) = (1 + \epsilon)^n - 1$.

**Definition 6.3.** Given $n \in \mathbb{N}$, let $\mu_n$ denote the function $\mu_n : \mathbb{R}^+ \rightarrow \mathbb{R}$ defined as: $\mu_n(\epsilon) = (1 + \epsilon)^n - 1$.

**Proposition 6.1.** Let $n, k$ be natural numbers, the function $\mu_n$ verifies the following properties:

1. $\mu_{n+1}(\epsilon) = \mu_n(\epsilon) + \epsilon \cdot (1 + \mu_n(\epsilon))$
2. $\mu_{n+k+1}(\epsilon) = \mu_n(\epsilon) + \mu_k(\epsilon) \cdot (1 + \mu_n(\epsilon)) + \epsilon \cdot (1 + \mu_n(\epsilon)) \cdot (1 + \mu_k(\epsilon))$
3. $\mu_n(\mu_k(\epsilon)) = \mu_{n \cdot k}(\epsilon)$.
4. $\mu_n(\epsilon) = n \cdot \epsilon + O(\epsilon^2)$
5. $\mu_n(\epsilon) \leq n \cdot \epsilon \cdot (1 + \epsilon)^{n-1}$

**Proof.** 1. From the definition of the function $\mu_n$ we have

$$\mu_n(\epsilon) + \epsilon \cdot (1 + \mu_n(\epsilon)) = \left((1 + \epsilon)^n - 1\right) + \epsilon \cdot (1 + (1 + \epsilon)^n - 1) =$$

$$(1 + \epsilon)^n + \epsilon \cdot (1 + \epsilon)^n - 1 = (1 + \epsilon)^{n+1} - 1 = \mu_{n+1}(\epsilon)$$

2. In the same way as for the previous equality, by the definition of $\mu_n$ we have

$$\mu_n(\epsilon) + \mu_k(\epsilon) \cdot (1 + \mu_n(\epsilon)) + \epsilon \cdot (1 + \mu_n(\epsilon)) \cdot \mu_k(\epsilon) =$$

$$= \left((1 + \epsilon)^n - 1\right) + \left((1 + \epsilon)^k - 1\right) \cdot (1 + \epsilon)^n - 1 +$$

$$= (1 + \epsilon)^n - 1 + (1 + \epsilon)^k \cdot (1 + \epsilon)^n + \epsilon \cdot (1 + \epsilon)^n \cdot (1 + \epsilon)^k - 1 =$$

$$(1 + \epsilon)^{k+n} + \epsilon \cdot (1 + \epsilon)^{k+n} - 1 = (1 + \epsilon)^{k+n+1} - 1 = \mu_{n+k+1}(\epsilon)$$

<table>
<thead>
<tr>
<th>Monomial form</th>
<th>Horner method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \cdot \alpha$</td>
<td>$Q_1$</td>
</tr>
<tr>
<td>$Q_1 \cdot \alpha$</td>
<td>$Q_2$</td>
</tr>
<tr>
<td>$Q_2 \cdot 3$</td>
<td>$Q_3$</td>
</tr>
<tr>
<td>$Q_1 \cdot 4$</td>
<td>$R_1$</td>
</tr>
<tr>
<td>$(-2) \cdot \alpha$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$Q_3 + R_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_1 + S_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>$T_2 + 1$</td>
<td>$T_3$</td>
</tr>
</tbody>
</table>
3. We now prove that the composition of two $\mu$ functions is the product function.

\[ \mu_n(\mu_k(\epsilon)) = (1 + \mu_k(\epsilon))^{n-1} = (1 + (1 + \epsilon)^k - 1)^{n-1} = (1 + \epsilon)^{k+1} - 1 = \mu_{kn}(\epsilon) \]

4. Expanding the expression by the Binomial Theorem we obtain that $\mu_n$ is of order 1

\[ \mu_n(\epsilon) = (1 + \epsilon)^n - 1 = \binom{n}{0} \cdot \epsilon + \binom{n}{1} \cdot \epsilon^2 + \binom{n}{2} \cdot \epsilon^3 + \cdots + \binom{n}{n} \cdot \epsilon^n - 1 = \epsilon \cdot n + \epsilon^2 \cdot \binom{n}{2} + \epsilon^3 \binom{n}{3} + \cdots + \epsilon^{n-1} \binom{n}{n-1} = \epsilon \cdot n + O(\epsilon^2) \]

5. Using \( \binom{n}{k} \leq n \cdot \frac{n-1}{k-1} \) for \( 1 \leq k \leq n \),

\[ \frac{\mu_n(\epsilon)}{\epsilon} \leq \left( \frac{n}{1} \right) \cdot \epsilon + \left( \frac{n}{2} \right) \cdot \epsilon^2 + \cdots + \left( \frac{n}{n} \right) \cdot \epsilon^{n-1} \leq \epsilon \cdot n \cdot (1 + \epsilon)^{n-1} \]

We now introduce a result we will need in order to prove subsequent theorems about the algorithms’s numerical errors.

**Lemma 6.2.** If $\beta \in (0, 1)$, given $a \in \mathbb{R}$, its approximation $a' \in \mathbb{R}$, and $\overline{a} \in \mathbb{R}^+$ such that $|a| \leq \overline{a}$ and $|a - a'| \leq \beta \cdot \overline{a}$ then:

\[ |a'| \leq (1 + \beta) \cdot \overline{a} \]

**Proof.** As $|a - a'| \leq \beta \cdot \overline{a}$, then $-\beta \cdot \overline{a} \leq a' - a \leq \beta \cdot \overline{a}$ and $-\beta \cdot \overline{a} \leq a - a' \leq \beta \cdot \overline{a}$

therefore, $a - \beta \cdot \overline{a} \leq a' \leq a + \beta \cdot \overline{a}$ and $-a - \beta \cdot \overline{a} \leq -a' \leq -a + \beta \cdot \overline{a}$

- If $a \geq 0$, then $|a| = a$ and

\[ -a - \beta \cdot \overline{a} \leq a - \beta \cdot \overline{a} \leq a' \leq a + \beta \cdot \overline{a} \]

So \[ |a'| \leq a + \beta \cdot \overline{a} = |a| + \beta \cdot \overline{a} \leq \overline{a} + \beta \cdot \overline{a} = (1 + \beta) \cdot \overline{a} \]

- If $a < 0$, then $|a| = -a$ and

\[ -(-a) - \beta \cdot \overline{a} \leq -a - \beta \cdot \overline{a} \leq -a' \leq (-a) + \beta \cdot \overline{a} \]

So \[ |a'| = |a - a'| \leq (-a) + \beta \cdot \overline{a} = |a| + \beta \cdot \overline{a} \leq \overline{a} + \beta \cdot \overline{a} = (1 + \beta) \cdot \overline{a} \]

We obtain the same result in both cases: $|a'| \leq (1 + \beta) \cdot \overline{a}$.

In the next theorem, we derive bounds for the error the computer commits when evaluating a basic algebraic expression

**Theorem 6.3.** The error introduced when computing a basic algebraic expression $Q_n$, with $n$ operations, is bounded by the function $\mu_n$:

\[ |Q_n - Q_n'| \leq \mu_n(\epsilon) \cdot \overline{Q}_n = (n \cdot \epsilon + O(\epsilon^2)) \cdot \overline{Q}_n \leq \epsilon \cdot n \cdot (1 + \epsilon)^{n-1} \cdot \overline{Q}_n \]

(6.2)
Proof. We will proceed by induction over \( n \).
The \( n = 1 \) case is trivial by (6.1):
\[
| (a_0 \perp 1 \, a_1) - (a_0 \perp 1 \, a_1)^* | \leq \epsilon \cdot |a_0 \perp 1 \, a_1| \leq \epsilon \cdot (|a_0| \perp 1 \, |a_1|)
\]
We suppose the statement is true for \( n \):
\[
|Q_n - Q_n^*| \leq \mu_n(\epsilon) \cdot \overline{Q}_n
\]
Applying Lemma 6.2 with \( a = Q_n \), \( a' = Q_n^* \), \( \beta = \mu_n(\epsilon) \) and \( \pi = \overline{Q}_n \):
\[
|Q_n^*| \leq (1 + \mu_n(\epsilon)) \cdot \overline{Q}_n
\]
(6.3)
For the \( n + 1 \) case:
\[
|Q_{n+1} - Q_{n+1}^*| = |(Q_n \perp_{n+1} a_{n+1}) - (Q_n^* \perp_{n+1} a_{n+1})^*|
\]
\[
= |(Q_n \perp_{n+1} a_{n+1}) - (Q_n^* \perp_{n+1} a_{n+1}) + (Q_n^* \perp_{n+1} a_{n+1}) - (Q_n^* \perp_{n+1} a_{n+1}^*)|^* \leq
\]
\[
|Q_n \perp_{n+1} a_{n+1}) - (Q_n^* \perp_{n+1} a_{n+1}) + |(Q_n^* \perp_{n+1} a_{n+1}) - (Q_n^* \perp_{n+1} a_{n+1}^*)|^* \leq
\]

• If \( \perp_{n+1} = \cdot \), by the induction hypothesis, the relative error introduced by the operation and the inequation (6.3)
\[
|Q_n - Q_n^*| \cdot |a_{n+1}| + \epsilon \cdot |Q_n^*| \cdot |a_{n+1}| \leq \mu_n(\epsilon) \cdot \overline{Q}_n \cdot |a_{n+1}| + \epsilon \cdot (1 + \mu_n(\epsilon)) \cdot \overline{Q}_n \cdot |a_{n+1}| =
\]
\[
= (\mu_n(\epsilon) + \epsilon \cdot (1 + \mu_n(\epsilon))) \cdot \overline{Q}_n \cdot |a_{n+1}| = \mu_{n+1}(\epsilon) \cdot \overline{Q}_{n+1}
\]

• If \( \perp_{n+1} = + \), then in the same way as before
\[
|Q_n - Q_n^*| + \epsilon \cdot |Q_n^*| + |a_{n+1}| \leq \mu_n(\epsilon) \cdot \overline{Q}_n + \epsilon \cdot (1 + \mu_n(\epsilon)) \cdot \overline{Q}_n + |a_{n+1}| \leq
\]
adding \( |a_{n+1}| \) to \( \overline{Q}_n \) in the first summand and \( \mu_n(\epsilon)|a_{n+1}| \) in the second factor in the second summand:
\[
\leq \mu_n(\epsilon) \cdot (\overline{Q}_n + |a_{n+1}|) + \epsilon \cdot (1 + \mu_n(\epsilon)) \cdot (\overline{Q}_n + |a_{n+1}|) \leq
\]
\[
(\mu_n(\epsilon) + \epsilon \cdot (1 + \mu_n(\epsilon))) \cdot (\overline{Q}_n + |a_{n+1}|) = \mu_{n+1}(\epsilon) \cdot (\overline{Q}_n \perp_{n+1} |a_{n+1}|) =
\]
\[
= \mu_{n+1}(\epsilon) \cdot \overline{Q}_{n+1}
\]
In both cases, \( |Q_{n+1} - Q_{n+1}^*| \leq \mu_{n+1}(\epsilon) \cdot \overline{Q}_{n+1} \).
And, finally, applying Proposition 6.1 we obtain
\[
|Q_n - Q_n^*| \leq \mu_n(\epsilon) \cdot \overline{Q}_n = (n \cdot \epsilon + O(\epsilon^2)) \cdot \overline{Q}_n \leq \epsilon \cdot n \cdot (1 + \epsilon)^{n-1} \cdot \overline{Q}_n
\]
\[
\square
\]
Now, we’re going to prove that the operations are compatible with the bounds. So, if \( S = Q \perp R \),
where \( Q \) and \( R \) are algebraic expressions verifying equation (6.2), \( S \) is bounded in the same way.

**Theorem 6.4.** Let \( Q \) and \( R \) be two algebraic expressions with \( n \) and \( k \) operations, respectively,
verifying equation (6.2) in Theorem 6.3.
Then, if \( S = Q \perp R \), the error when computing \( S \) is bounded by the function \( \mu_l \) with \( l = n + k + 1 \),
that is,
\[
|S - S^*| \leq \mu_l(\epsilon) \cdot \overline{S} = (l \cdot \epsilon + O(\epsilon^2)) \cdot \overline{S} \leq \epsilon \cdot l \cdot (1 + \epsilon)^{n+k} \cdot \overline{S}
\]
(6.4)
Proof. We assume $S = Q \perp R$ with $l = n + k + 1$ operations.

\[ |S - S^*| = |(Q \perp R) - (Q^* \perp R^*)| = \]

\[ |(Q \perp R) - (Q^* \perp R) - (Q^* \perp R^*) - (Q^* \perp R^*)| \leq \]

\[ |(Q \perp R) - (Q^* \perp R)| + |(Q^* \perp R) - (Q^* \perp R^*)| + |(Q^* \perp R^*) - (Q^* \perp R^*)| \leq \]

- If $\leq -$ then

\[ |Q - Q^*| \cdot |R| + |Q^*| \cdot |(R - R^*)| + \epsilon \cdot |Q^*| \cdot |R| \leq \]

\[ \mu_n(e) \cdot \overline{Q} \cdot \overline{R} + \mu_k(e) \cdot \overline{R} \cdot |Q^*| + \epsilon \cdot |Q^*| \cdot |R^*| \leq \]

\[ \mu_n(e) \cdot \overline{Q} \cdot \overline{R} + \mu_k(e) \cdot \overline{R} \cdot (1 + \mu_n(e)) \cdot \overline{Q} + \epsilon (1 + \mu_n(e)) \cdot \overline{Q} + (1 + \mu_n(e)) \cdot \overline{R} = \]

\[ (\mu_n(e) + \mu_k(e) \cdot (1 + \mu_n(e))) + \epsilon (1 + \mu_n(e)) \cdot (1 + \mu_k(e))) \cdot (\overline{Q} \cdot \overline{R}) = \]

\[ \mu_i(e) \cdot (\overline{Q} \perp \overline{R}) \]

- If $\leq +$, then

\[ |Q - Q^*| + |(R - R^*)| + \epsilon \cdot |(Q^* + |R^*)| \leq \mu_n(e) \cdot \overline{Q} + \mu_k(e) \cdot \overline{R} + \epsilon \cdot |(Q^* + |R^*)| \leq \]

\[ \mu_n(e) \cdot (\overline{Q} + \overline{R}) + \mu_k(e) \cdot (1 + \mu_n(e)) \cdot \overline{R} + \epsilon \cdot (1 + \mu_n(e)) \cdot \overline{Q} + (1 + \mu_n(e)) \cdot \overline{R} = \]

\[ \mu_n(e) \cdot (\overline{Q} + \overline{R}) + \mu_k(e) \cdot (1 + \mu_n(e)) \cdot (\overline{Q} + \overline{Q}) + \epsilon \cdot (1 + \mu_n(e)) \cdot (1 + \mu_k(e)) \cdot (\overline{Q} + \overline{R}) = \]

\[ (\mu_n(e) + \mu_k(e) \cdot (1 + \mu_n(e))) + \epsilon (1 + \mu_n(e)) \cdot (1 + \mu_k(e))) \cdot (\overline{Q} + \overline{R}) = \]

\[ \mu_i(e) \cdot (\overline{Q} \perp \overline{R}) \]

In both cases, we obtain

\[ |(Q \perp R) - (Q^* \perp R^*)| \leq \mu_i(e) \cdot (\overline{Q} \perp \overline{R}) = \mu_i(e) \cdot (\overline{Q} \perp \overline{R}) \]

Then, applying Proposition 6.1, we conclude

\[ |S - S^*| \leq \mu_i(e) \cdot \overline{S} = (l \cdot \epsilon + O(e^2)) \cdot \overline{S} \leq \epsilon \cdot l \cdot (1 + \epsilon)^n \cdot \overline{S} \]

\[ \square \]

It is therefore straightforward to prove that algebraic expressions are bounded, and to find their bounds.

**Corollary 6.5.** The error introduced when computing an algebraic expression $S$ is bounded as follow:

\[ |S - S^*| \leq \mu_m(e) \overline{S} \]

where $m$ is the number of operations we need to obtain $S$.

**Proof.** The proof is concluded from Theorem 6.3 and Theorem 6.4 and from Definition 6.1 of algebraic expression. \( \square \)

### 6.2 Horner's Algorithm Error

Now, we can find out bounds for Horner’s algorithm.

**Corollary 6.6.** Given a polynomial $P(x) = \sum_{i=0}^{n} a_i \cdot x^i$ and a number $\alpha \in \mathbb{R}$, the error introduced by the classic Horner method when computing $P(\alpha)$ is bounded by $\mu_2n(e) \overline{P}(|\alpha|)$, that is,

\[ |P(\alpha) - (P(\alpha))^H| \leq \mu_2n(e) \cdot \overline{P}(|\alpha|) \]

where $^H$ denotes the computed result when we apply the classic Horner algorithm and $\overline{P}(x) = \sum_{i=0}^{n} |a_i| \cdot x^i$. 

32
Proof. The Horner algorithm is described by the algebraic expression $Q_{2n}$ defined recursively,

1. $Q_1 = a_n \cdot \alpha$,
2. If $k = 2i$ for $i = 1, \ldots, n$ then $Q_k = Q_{k-1} + a_{n-i}$
3. If $k = 2i + 1$ for $i = 1, \ldots, n-1$ then $Q_k = Q_{k-1} \cdot \alpha$

By applying Theorem 6.3 to $Q_{2n}$, we conclude

$$\left| P(\alpha) - (P(\alpha))^{*H} \right| = \left| Q_{2n} - Q_{2n}^{*} \right| < \mu_{2n}(\epsilon) \cdot \overline{Q}_{2n} = \mu_{2n}(\epsilon) \cdot \mathcal{P}(\alpha)$$

6.3 Parallel Horner’s Algorithm Error

Let $P(x)$ be a polynomial of degree $n$ and $t$ the number of subpolynomials, we can rewrite the polynomial

$$P(x) = \sum_{i=0}^{n} a_i \cdot x^i = \sum_{k=0}^{t-1} p_k(x) \cdot y^k$$

where $w = \left\lceil \frac{n+1}{t} \right\rceil$ and $y = x^w$. It verifies that $\deg(p_k(x)) = w - 1$ for $1 \leq k \leq t - 2$ and $\deg(p_{t-1}(x)) = r - 1 = n \mod w$.

Given $\alpha \in \mathbb{R}$, we consider the polynomial of degree $t - 1$

$$H(y) = \sum_{k=0}^{t-1} h_k \cdot y^k \text{ where } h_k = p_k(\alpha)$$

Applying the parallel Horner algorithm is equivalent to computing $H(\beta)$ with $\beta = \alpha^w$.

**Corollary 6.7.** Under the previous notation, given a polynomial of degree $n$, $t$ the number of subpolynomials and $\alpha \in \mathbb{R}$, a bound of the error, when we apply the parallel Horner algorithm, is $\mu_d(\epsilon) \cdot \overline{P}(\alpha)$ with $d = 3n - \deg(H(x)) - \deg(p_{t-1}(x))$, that is,

$$\left| P(\alpha) - (P(\alpha))^{*PH} \right| \leq \mu_d(\epsilon) \cdot \overline{P}(\alpha) \text{ where } \overline{P}(x) = \sum_{i=0}^{n} |a_i| \cdot x^i$$

where $^{*PH}$ denotes the computed result when we apply the parallel Horner algorithm and, as in the Corollary 6.6, $\overline{P}(x) = \sum_{i=0}^{n} |a_i| \cdot x^i$.

Proof. When we apply the parallel Horner algorithm, we first compute the numbers $h_k = p_k(\alpha)$ and $\beta = \alpha^w$. By Theorem 6.3, the bounds of the errors are

\[
\begin{align*}
|h_k - (h_k)^{*H}| & = |p_k(\alpha) - (p_k(\alpha))^{*H}| \leq \mu_{2w-2}(\epsilon) \cdot \overline{P}_k(\alpha) \quad \text{for } 0 \leq k \leq t - 2 \\
|h_{t-1} - (h_{t-1})^{*H}| & = |p_{t-1}(\alpha) - (p_{t-1}(\alpha))^{*H}| \leq \mu_{2w-2}(\epsilon) \cdot \overline{P}_{t-1}(\alpha)
\end{align*}
\]

\[|\beta - (\beta)^{*}| = |\alpha^w - (\alpha^w)^{*}| \leq \mu_{w-1}(\epsilon) \cdot |\alpha|^w\]

We now apply Horner to find the value of $H(\beta)$, that is, we consider the algebraic expressions, $Q_{2m-2}$ and $(Q_{2m-2})^{*}$, defined recursively

1. $Q_1 = h_{t-1} \cdot \beta$ and $(Q_1)^{*} = ((h_{t-1})^{*H} \cdot (\beta)^{*})$
2. If $k = 2i$ for $i = 1, \ldots, t - 1$ then

$$Q_k = Q_{k-1} + h_{t-1-i} \text{ and } (Q_k)^{*} = ((Q_{k-1})^{*} + (h_{t-1-i})^{*H})^{*}$$
3. If \(k = 2i + 1\) for \(i = 1, \ldots, t - 2\) then
\[
Q_k = Q_{k-1} \cdot \beta \quad \text{and} \quad (Q_k)^* = ((Q_{k-1})^* \cdot (\beta)^*)^\
\]

By applying the theorem 6.4 to \(Q_k\),
1. \(|Q_1 - (Q_1)^*| \leq \mu_{d_1}(\epsilon) \cdot Q_1^\epsilon\) with \(d_1 = 2 \cdot r - 2 + w - 1 + 1\)
2. If \(k = 2i\) for \(i = 1, \ldots, t - 1\) then \(|Q_k - (Q_k)^*| \leq \mu_{d_k}(\epsilon) \cdot Q_k^\epsilon\) with \(d_k = d_{k-1} + 2 \cdot w - 1 + 1\)
3. If \(k = 2i + 1\) for \(i = 1, \ldots, t - 2\) then \(|Q_k - (Q_k)^*| \leq \mu_{d_k}(\epsilon) \cdot Q_k^\epsilon\) with \(d_k = d_{k-1} + w - 1 + 1\)

Finally, the algebraic expression to compute parallel Horner algorithm, \(Q_{2t-2}\), is bounded as follows:
\[
|P(\alpha) - (P(\alpha))^*_{PH}| = |Q_{2t-2} - (Q_{2t-2})^*| \leq \mu_{d_{2t-2}}(\epsilon) \cdot Q_{2t-2}^\epsilon = \mu(\epsilon) \cdot P(|\alpha|)
\]

where
\[
d = d_{2t-2} = 2 \cdot r - 2 + (3 \cdot w - 3 + 2) \cdot (t - 1) = 3 \cdot (r - 1 + w \cdot (t - 1)) - (t - 1) - (r - 1) = 3n - \deg(H(x)) - \deg(p_{t-1}(x))
\]

\hspace{1cm}\Box

7 Conclusions and Further Research

In this paper, we have proposed a polynomial partitioning strategy that allows parallel evaluation of a polynomial at a point by Horner’s method.

We have found that this parallelization substantially improves the classical application of the algorithm. The proposed method is a good candidate as the default general polynomial evaluation method. By construction, it reduces to the iterative algorithm when executing over a single thread. Multi-threaded environments would always benefit from a speedup, even more as larger the polynomial degree is: applications likely to deal with large polynomials (\(N \geq 2000\)) should expect efficiency gains in the 40% - 90% range with no downside in the single-threaded case. The method is also simple to implement and reason about, leaving little room for programming errors.

It has also been seen that with just small modifications in the application of Horner’s method we can benefit from its advantages when it comes to evaluating sparse polynomials.

We have also verified that the error introduced by the parallelized method is of the same order as that of the classical one. In other words, in addition to speeding up the classic method, any error introduced is not of higher order.

Further research can explore more elaborate partition schemes suitable for variable size coefficients and study how to adapt our strategy of parallelization of the Horner method to the evaluation of multivariate polynomials.

Competing Interests

Authors have declared that no competing interests exist.

References


© 2022 Fuentes et al.; This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution and reproduction in any medium, provided the original work is properly cited.

Peer-review history:
The peer review history for this paper can be accessed here (Please copy paste the total link in your browser address bar)
https://www.sdiarticle5.com/review-history/89359