**1.1 Classic Sequential Multiplication**

**Description:**

* This algorithm uses a straightforward nested loop approach to compute the product of two polynomials.
* Time Complexity: O(n^2) where n is the degree of the polynomials.

**Steps:**

1. Create a result list of size p1.degree+p2.degree+1, initialized to zeros.
2. For each coefficient i in p1:
   * For each coefficient j in p2:
     + Compute the product of the coefficients at i and j.
     + Add this product to the corresponding position in the result (i+j).
3. Return the result as a new polynomial.

**1.2 Classic Parallel Multiplication**

**Description:**

* This is a parallelized version of the classic multiplication algorithm, dividing the computation into smaller tasks executed concurrently.
* Time Complexity: O(n2) but parallel execution reduces the runtime depending on the number of threads.

**Steps:**

1. Divide the result polynomial into segments based on the number of threads.
2. Assign each segment to a PolynomialTask that calculates the partial results for that segment.
3. Use a ThreadPoolExecutor to execute the tasks concurrently.
4. Wait for all tasks to complete, then return the final result.

**1.3 Karatsuba Multiplication**

**Description:**

* A divide-and-conquer algorithm that reduces the number of coefficient multiplications compared to the classic approach.
* Time Complexity: O(n log2​(3))≈O(n1.59).

**Steps:**

1. **Base Case:** If the degree of the polynomials is small, use the classic sequential algorithm.
2. Split each polynomial into "low" and "high" halves:
   * p1=lowP1+xm⋅highP1
   * p2=lowP2+xm⋅highP2
3. Compute three partial products:
   * z1=lowP1⋅lowP2
   * z2=(lowP1+highP1)⋅(lowP2+highP2)
   * z3=highP1⋅highP2
4. Combine the results:
   * Final result = z3⋅x2m+(z2−z3−z1)⋅xm+z1

**1.4 Karatsuba Parallel Multiplication**

**Description:**

* A parallelized version of the Karatsuba algorithm that executes the recursive calls for z1, z2, and z3​ concurrently.
* Time Complexity: : O(n log2​(3))≈O(n1.59). with reduced runtime due to parallel execution.

**Steps:**

1. **Base Case:** If the depth of recursion exceeds a threshold or the degree is small, fall back to the sequential Karatsuba algorithm.
2. Split the polynomials into "low" and "high" halves.
3. Use Callable tasks to compute z1​, z2 and z3​ in parallel.
4. Combine the results using the same approach as the sequential Karatsuba algorithm.

**2. Synchronization in Parallelized Variants**

**2.1 Classic Parallel Multiplication**

* **Shared State:** The result polynomial is shared among threads.
* **Synchronization:**
  + Each thread operates on a separate segment of the result polynomial, avoiding race conditions.
  + No explicit synchronization is required due to non-overlapping segments.

**2.2 Karatsuba Parallel Multiplication**

* **Shared State:** Recursive tasks operate on different polynomial segments, so there is no shared state in intermediate computations.
* **Synchronization:**
  + A ThreadPoolExecutor manages tasks, ensuring that threads are reused efficiently.
  + Futures are used to retrieve results from concurrent tasks, and the awaitTermination method ensures all tasks complete before combining results.

**3. Performance Measurements**

**Environment:**

* **Processor:** Modern multi-core processor (e.g., Intel Core i5-8300H).
* **Input:** Polynomials of degree 10,000.
* **Number of Threads:** 2 threads for classic parallel and dynamic threads for Karatsuba parallel.

**Results (Example):**

| **Algorithm** | **Execution Time (ms)** |
| --- | --- |
| Classic Sequential | 1354 |
| Classic Parallel (2 threads) | 999 |
| Karatsuba Sequential | 846 |
| Karatsuba Parallel | 296 |

**Analysis:**

1. **Classic Sequential vs. Parallel:**
   * Parallelization significantly reduces runtime for large polynomials by leveraging multiple CPU cores.
   * The benefit diminishes for smaller polynomials due to thread management overhead.
2. **Karatsuba Sequential vs. Classic Sequential:**
   * The Karatsuba algorithm is faster for large inputs due to reduced multiplications.
   * It outperforms classic multiplication for polynomials with a degree greater than ~500.
3. **Karatsuba Parallel:**
   * The parallelized version of Karatsuba achieves the best performance, as it combines algorithmic efficiency with concurrency.
   * Scalability improves with more threads, but diminishing returns may occur due to overhead and limited CPU resources.

  
