Description of Algorithms:

- Parallel polynomial evaluation: we try to implement the Horners Rule (**mpi_poly_eval1()**):
    - The last processor (processor with the highest rank) p_{n-1} has a polynomial value equal to the last constant value in the global constant array, or the first constant value of the constant chunk it receives after the scatter call on the source_rank processor. The last processor is also the last receiver in order to prevent infinite information sending within its machine cluster. Therefore, it will only send its value to the next lower rank (the rank difference between the two processor is 1).

      $P_x = a[0] = a_{n-1}$, a_{n-1} is the first element of the local constant array a of p_{n-1}

    - The lower rank processor, after receiving the polynomial value from the higher rank, will use this polynomial evaluation and the broadcast value x it receives from the source_rank to calculate its polynomial as follows:

      $P_{x-1} = P_x * x + a[0] = a_{n-1} x + a_{n-2}$, a_{n-2} is an first element of the local constant array a of p_{n-2}

      This communication continues down to the processor right after the lowest ranked processor (processor index 0):

      $$P_{x-n-1} = P_{x-n-2} * x + a[0] = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + ... + a_1 x$$

    - When the polynomial evaluation is sent to processor 0. Processor 0 only receives from this value and does not pass on its polynomial evaluation after the polynomial is evaluated, i.e. processor 0 is the last receiver for this algorithm. This is to prevent infinite information sending within its cluster. Then, processor 0 will compute its polynomial value similar to the its higher rank processor:

      $$P_{x-n} = P_{x-n-1} * x + a[0] = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + ... + a_1 x + a_0$$

    - Broadcast this final polynomial to all other processors so they have result polynomial evaluation and return the polynomial.

- Parallel prefix_sum:
    - Step 1: Calculate number of steps of hypercubic communications (called prefix_steps() in our program, locates in utils.cpp). By assuming the number of processors n is of base 2, the number of steps of hypercubic communications is $\log_2(n)$. We find this result by continuously divide n by 2 in a loop until the quotient is less than 1.

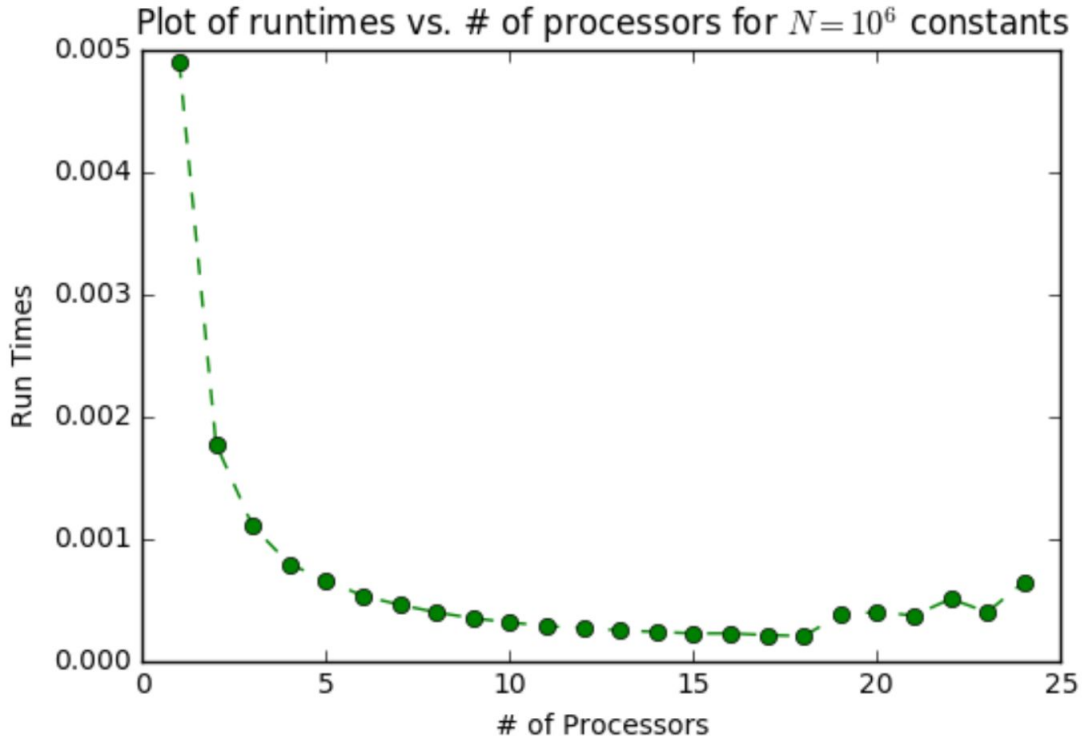
    - Step 2: Have a counter “key” initialized to 1 to keep track of the distance between the send processor (sender) and the receive processor (receiver). “key” will be increased by factor 2 after each iteration (or communication step).
        - At each iteration, senders from the lower rank will send their respective local sums on to the receivers which is “key” rank higher. The last sender will be the processor right before the highest ranked processor.
        - After the receiver receives values from the lower rank, it will add this value to its local sum and update this new sum to be its local sum.
        - “key” is increased by factor 2.
    - Step 3: return the final prefix_sum which is the local sum at the last processor.

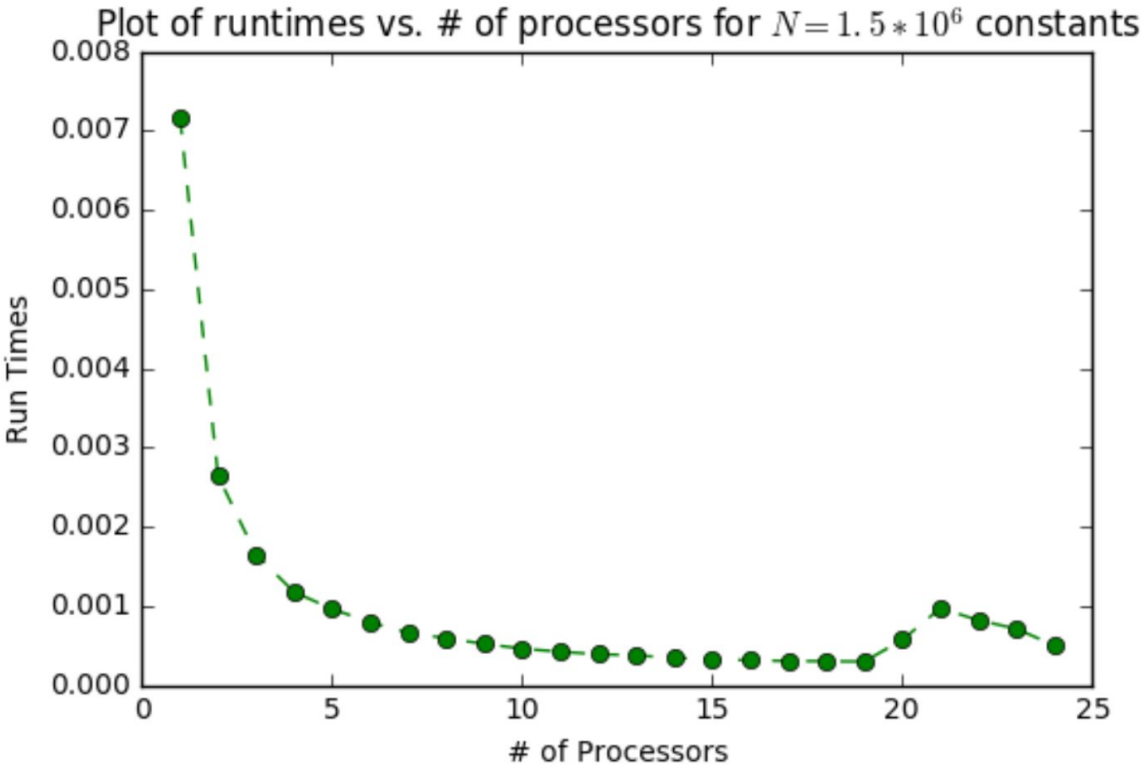
- Parallel prefix_product: similar to prefix_sum. But instead of having a local sum, a local product is used and updated.

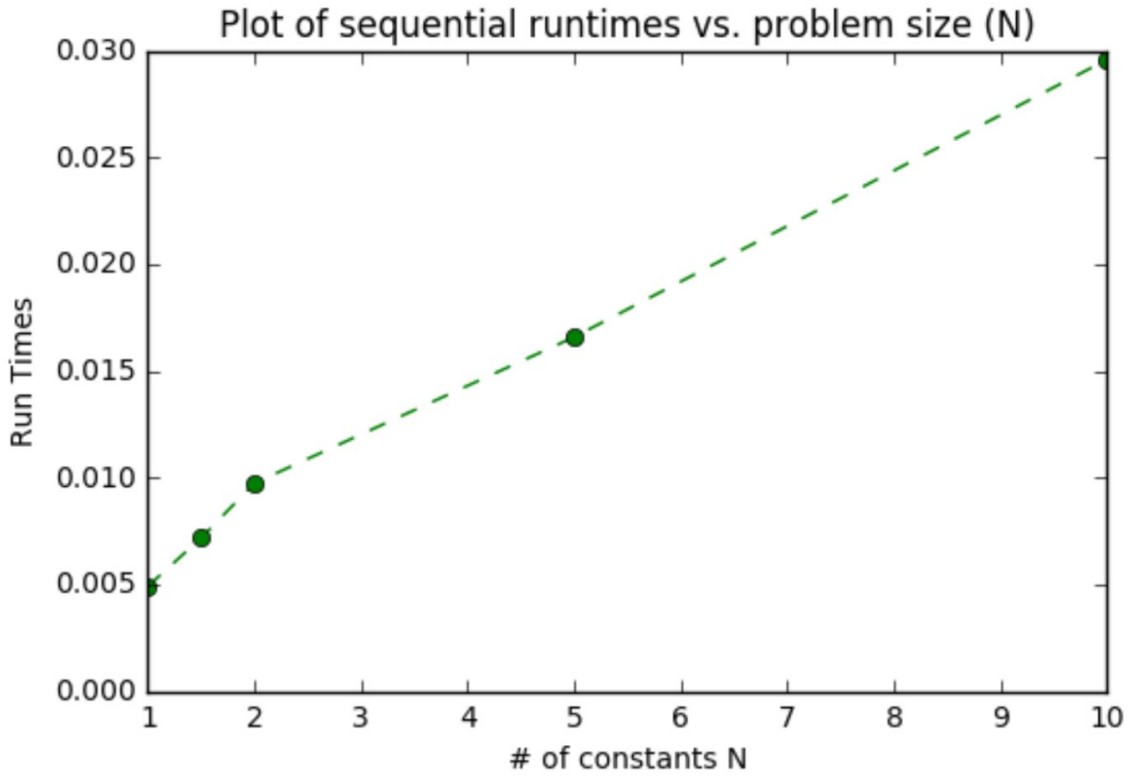
- Scatter:
    - Step 1: Calculate the size of the message block to be sent (floor(n/p)) and any extra elements (in case the problem size n is not divisible by p). For example, if n = 11, p = 8 ⇒ $message\ block = floor(\frac{n}{p})$ = 1, extra = n%p = 3.
    - Step 2: Use loop to iterate through p processors. If the current processor is the source_rank, it will keep the first chunk of n (message block) starting at index 0 of the value array and send the next chunks of n (index 0 + message block) to the next processors. If the current processor is not the source_rank, it will receive from source_rank.
    - Step 3: Since each processor should receive more than the others maximum one element, Scatter function will distribute $message\ block = floor(\frac{n}{p})$ + 1(this extra 1 element is from the “extra” ⇒ $extra - 1\ after\ each\ distribution$) elements to the source_rank, source_rank + 1, source_rank + 2, …, until “extra” is exhausted. All the remaining processor will get $message\ block = floor(\frac{n}{p})$ elements.

- Broadcast:
    - Step 1: source_rank will send the value to processor 0 and the processor 0 will receive the value.
    - Step 2: compute number of times that the value will be broadcasted, which equals to log(n).
    - Step 3: In i = 0 to log(n) iteration, the processors that know the value will send to the processor that is (p/n^i) away from it.

Graph plots:


Plot of runtimes vs. # of processors for $N = 10^6$ constants
Run Times
# of Processors
0.000
0.001
0.002
0.003
0.004
0.005
0
5
10
15
20
25


Plot of runtimes vs. # of processors for $N = 1.5 * 10^6$ constants
Run Times
# of Processors
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0
5
10
15
20
25



Run-times analysis:

- With large problem size (or number of constants) up to 1,000,000, the run time decreases as more processors are added. However, the parallel slowdown begins to occurs when the 21st processor is added. This makes sense because the communication time to broadcast among processors becomes larger than the time that the processors evaluate their polynomial.
- Therefore, when the problem size n is made bigger, the processors spend more time to do useful computation work. At this point, we observe a desired decrease in run time again.
- The value of x is relatively small in order to prevent overflow at the last polynomial result since the maximum number length that double type can hold is 10^{308}

	Broadcast	**PolyEvaluator**	**x**	**n**	**p**	**Efficient**
Big-O	O(log (n)) increases in proportion with p	O(n/p) decreases in proportion with p	1-e < x < 1+e	10^6	1-24	Slow down at processor 19th
				$1.5*10^6$	1-24	Slow down at processor 21th

				$2*10^6$	1-24	More efficient after processor 21^{th} is added