

Implementing a Point Domination Query using MPI

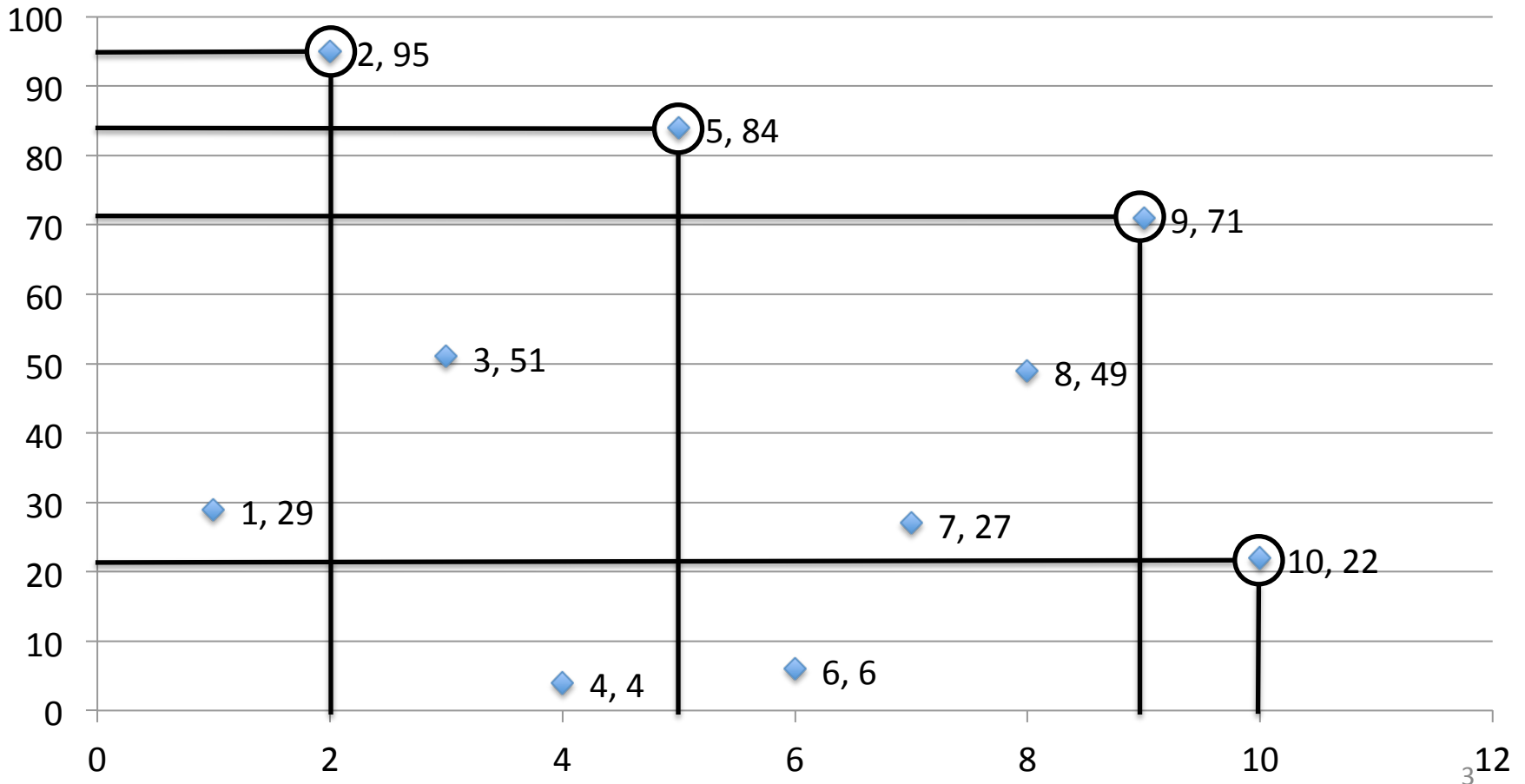
Eric Nagler
ednagler@buffalo.edu

What is a Point Domination Query?

- A Point Domination Query is from the field of computational geometry where we are interested if a point “dominates” another point
- The query will return the dominating points that encompass the other non-dominated points in the dataset
- A point “dominates” another point $q_1 = (x_1, y_1)$, $q_2 = (x_2, y_2)$ if and only if $x_1 > x_2$ and $y_1 > y_2$

Graphical Example

- Example Values : 29, 95, 51, 4, 84, 6, 27, 49, 71, 22
- Domination Points : 95, 84, 71, 22



Implementations of the Query

- Three implementations of the query
 - RAM Implementation
 - Implement the RAM version and run against different sizes of the dataset and benchmark the performance
 - Master/Worker Implementation
 - Implement a version that a master node will distribute different sizes of the dataset to the worker nodes to process. The worker nodes will process the data using the RAM algorithm. When the worker nodes are complete they will communicate their local postfix result to the master, then the master will take a final postfix to retrieve the final result
 - Parallel Postfix Implementation
 - Implement a version that will perform a parallel postfix operation between nodes and in the first node the final answer will be collected. The workers will run the local RAM algorithm and then pass their local results to their next postfix node

Testing Details

- All tests were conducted on the IBM eight cores per node CCR machines
- All tests were conducted on a variety of input data set sizes. The final query result is smaller than the input data set. The final query results number of data items are shown in parentheses
 - 1,000,000 items (5847 items)
 - 5,000,000 items (25274 items)
 - 10,000,000 items (28359 items)
 - 20,000,000 items (35869 items)
 - 30,000,000 items (47687 items)
- All items passed between processing elements are integers
- For the parallel implementations, the data set was evenly distributed among the processing elements for querying
- The run times represent the average over three tests of the algorithms

RAM Algorithm Details

Pseudo Code:

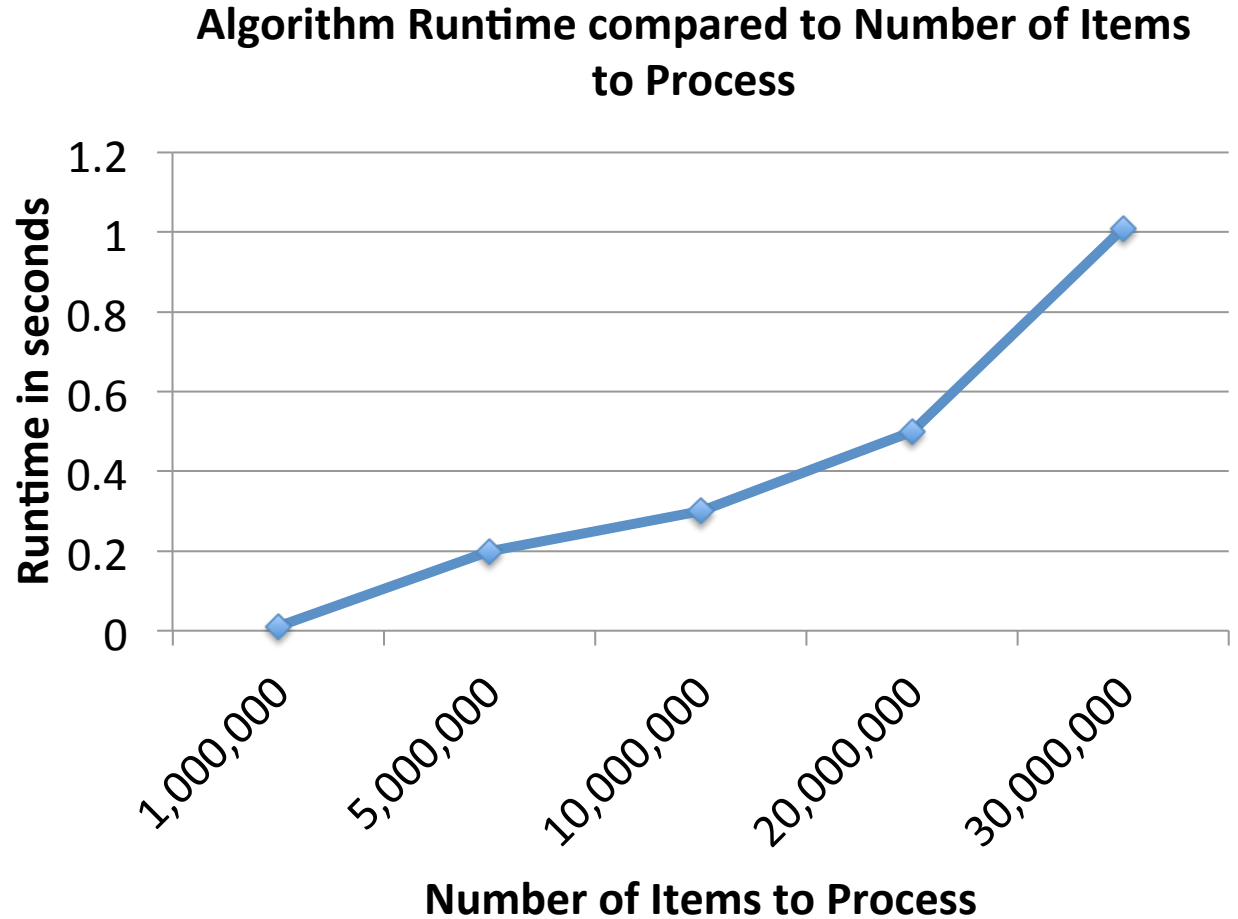
- Initially, postfixArray and valueArray contain the read in dataset

```
int valueArray[ARRAYSIZE], postfixArray[ARRAYSIZE];  
for (x = ARRAYSIZE-1 to 0)  
    postfixArray[x-1] = max(postfixArray[x], valueArray[x-1]);  
end for
```

- Run time of the RAM algorithm is $\Theta(n)$

RAM Performance Results

Number of Items	Runtime (seconds)
1,000,000	0.01
5,000,000	0.2
10,000,000	0.3
20,000,000	0.5
30,000,000	1.01



Master/Worker Implementation

- Algorithm Outline
 - Processor zero will distribute the input dataset to all of the child processors
 - When the file is completely distributed, the worker processors will take their local query and send their results back to processor zero
 - The master processor will collect the workers results append them together and take a final query and write the results to disk

Master/Worker Results

Table of runtime in seconds with multiple dataset sizes on multiple numbers of processing elements

Number of data items to process	Two processing elements (1 Master 1 Worker)	Three processing elements (1 Master 2 Workers)	Four processing elements (1 Master 3 Workers)	Five processing elements (1 Master 4 Workers)	Eight processing elements (1 Master 7 Workers)
1,000,000	0.025623	0.014198	0.010928	0.009412	0.007821
5,000,000	0.341468	0.179464	0.14994	0.136634	0.12355
10,000,000	0.593014	0.248078	0.208775	0.190851	0.166605
20,000,000	1.098337	0.656923	0.35209	0.315826	0.276557
30,000,000	2.300388	1.300863	0.962853	0.553948	0.518589

All tests were ran on one node with 8 processing elements

Master/Worker Results

Table of runtime in seconds with multiple dataset sizes on multiple numbers of processing elements

Number of data items to process	Nine processing elements (1 Master 8 Workers)	Sixteen processing elements (1 Master 15 Workers)	Seventeen processing elements (1 Master 16 Workers)	Thirty-Two processing elements (1 Master 31 Workers)	Thirty-Three processing elements (1 Master 32 Workers)	Sixty-Four processing elements (1 Master 63 Workers)
1,000,000	0.008436	0.00701	0.007159	0.007022	0.007336	0.008803
5,000,000	0.12237	0.118056	0.11527	0.114039	0.114701	0.115285
10,000,000	0.163099	0.151837	0.150433	0.145029	0.14584	0.144701
20,000,000	0.270884	0.251032	0.249424	0.241156	0.240604	0.237576
30,000,000	0.489801	0.459813	0.459662	0.419091	0.434146	0.438205

Number of nodes needed:
(8 PE's each)

2 nodes

3 nodes

4 nodes

5 nodes

8 nodes

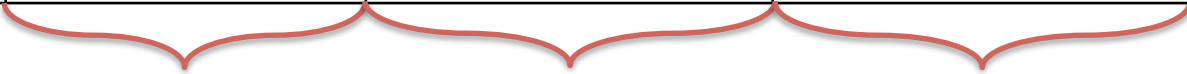
Master/Worker Results

Table of runtime in seconds with multiple dataset sizes on multiple numbers of processing elements

Number of data items to process	Ninety-Six processing elements (1 Master 95 Workers)	128 processing elements (1 Master 127 Workers)	256 processing elements (1 Master 255 Workers)
1,000,000	0.010756	0.009933	0.020966
5,000,000	0.117194	0.118332	0.126832
10,000,000	0.147281	0.149588	0.194236
20,000,000	0.239105	0.2401713	0.252065
30,000,000	0.437994	0.441021	0.448289

Number of nodes needed:
(8 PE's each)

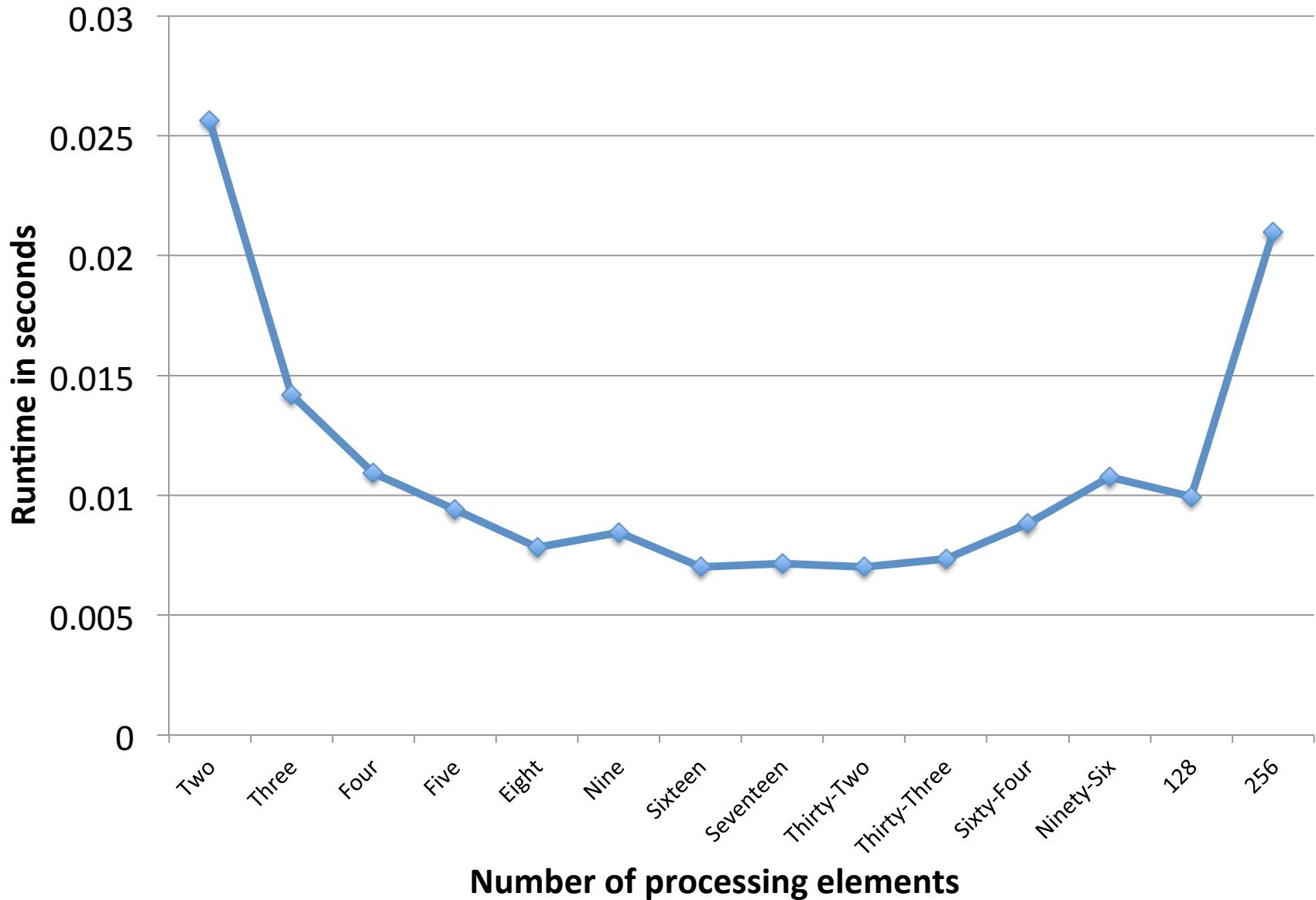
12 nodes 16 nodes 32 nodes



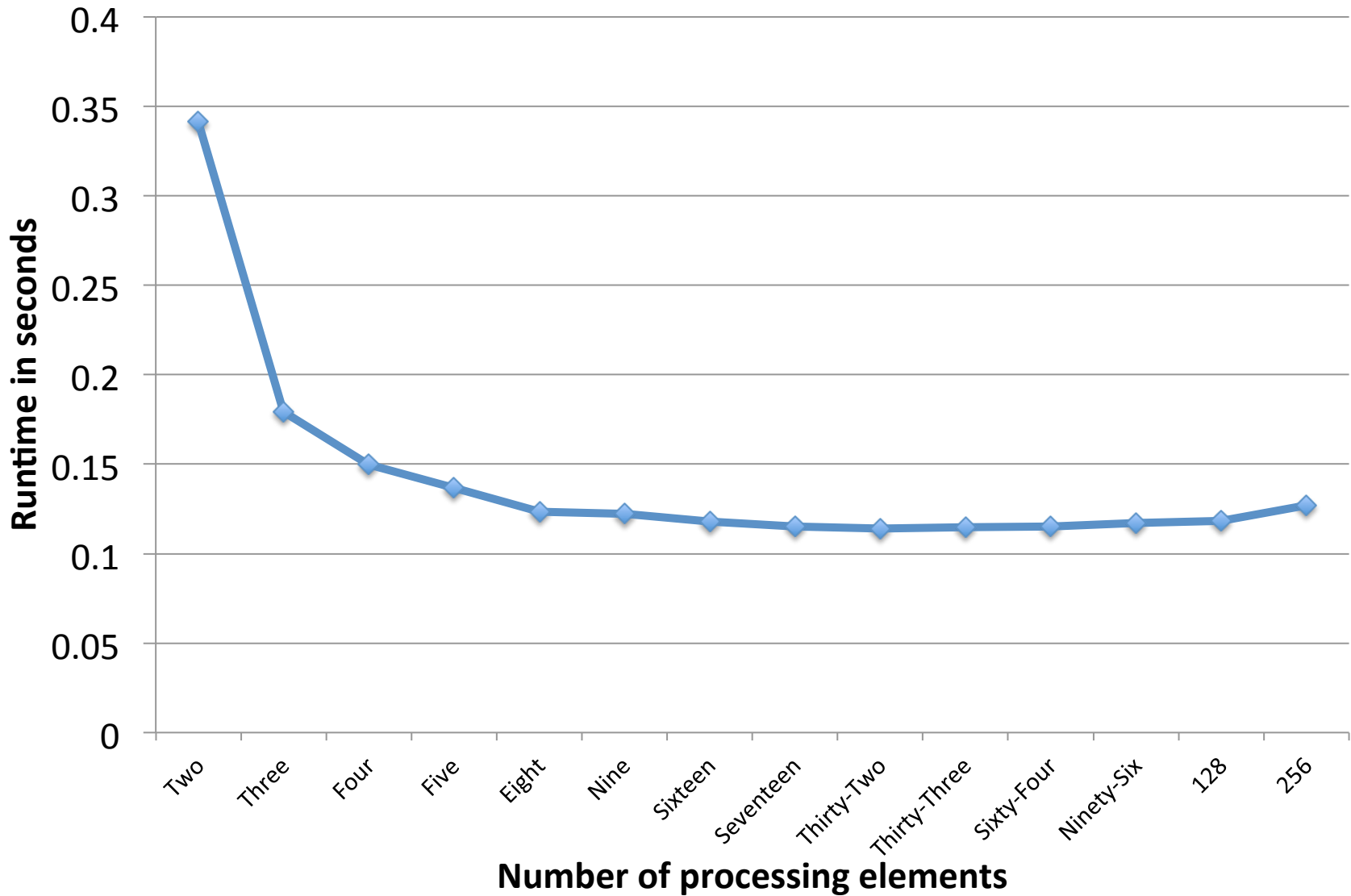
Master/Worker Performance Observations

- When a test was conducted that required only one process on a separate node the run time was slightly higher. This was most likely because of the inter-node communication
- When the number of nodes increases, the most amount of time taken by the algorithm is by the workers waiting to send their local postfix results to the master.
 - This can be observed easily because at 256 processing elements the algorithm running time starts to increase.

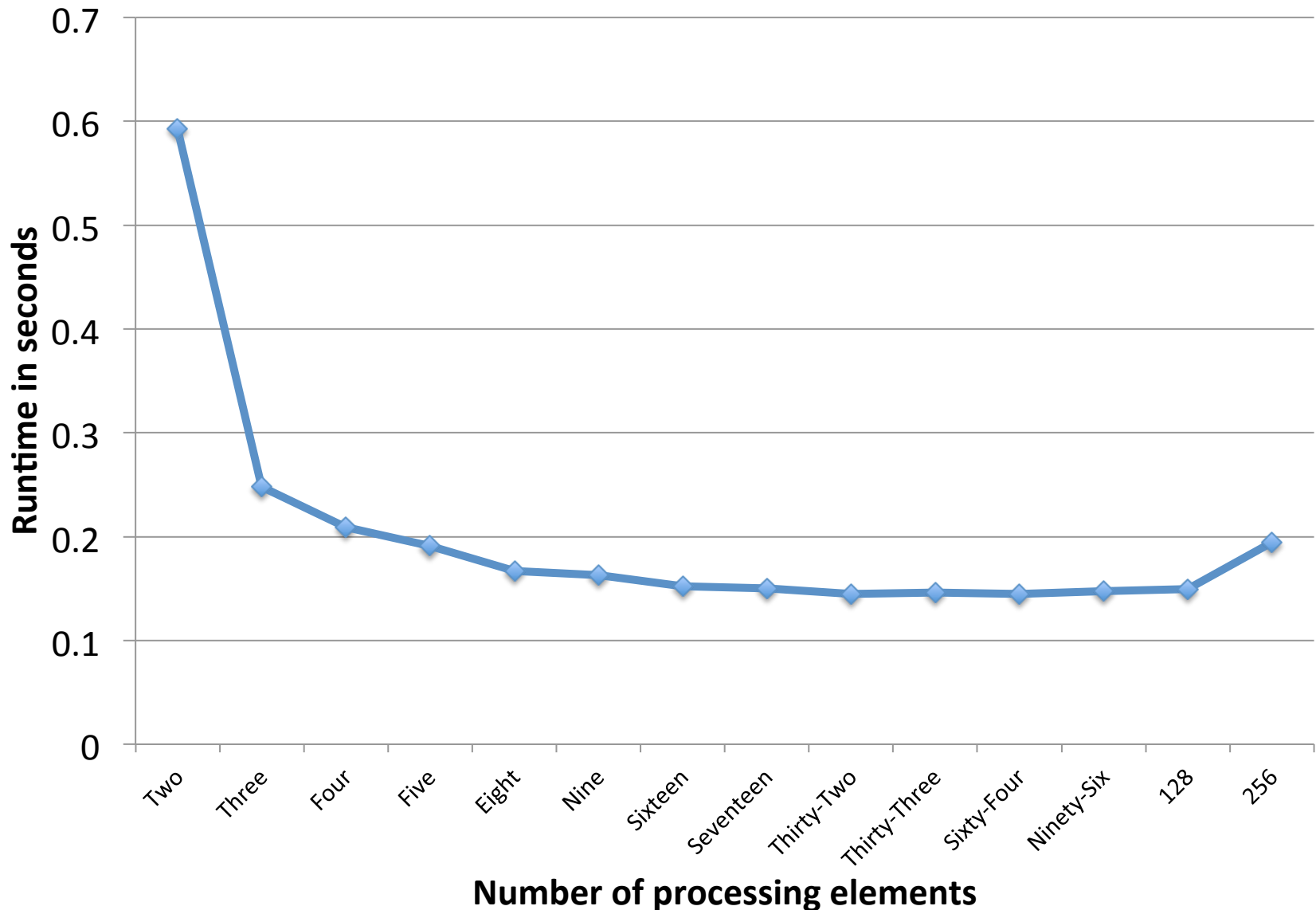
Time in seconds needed to process 1,000,000 items divided evenly among N processing elements



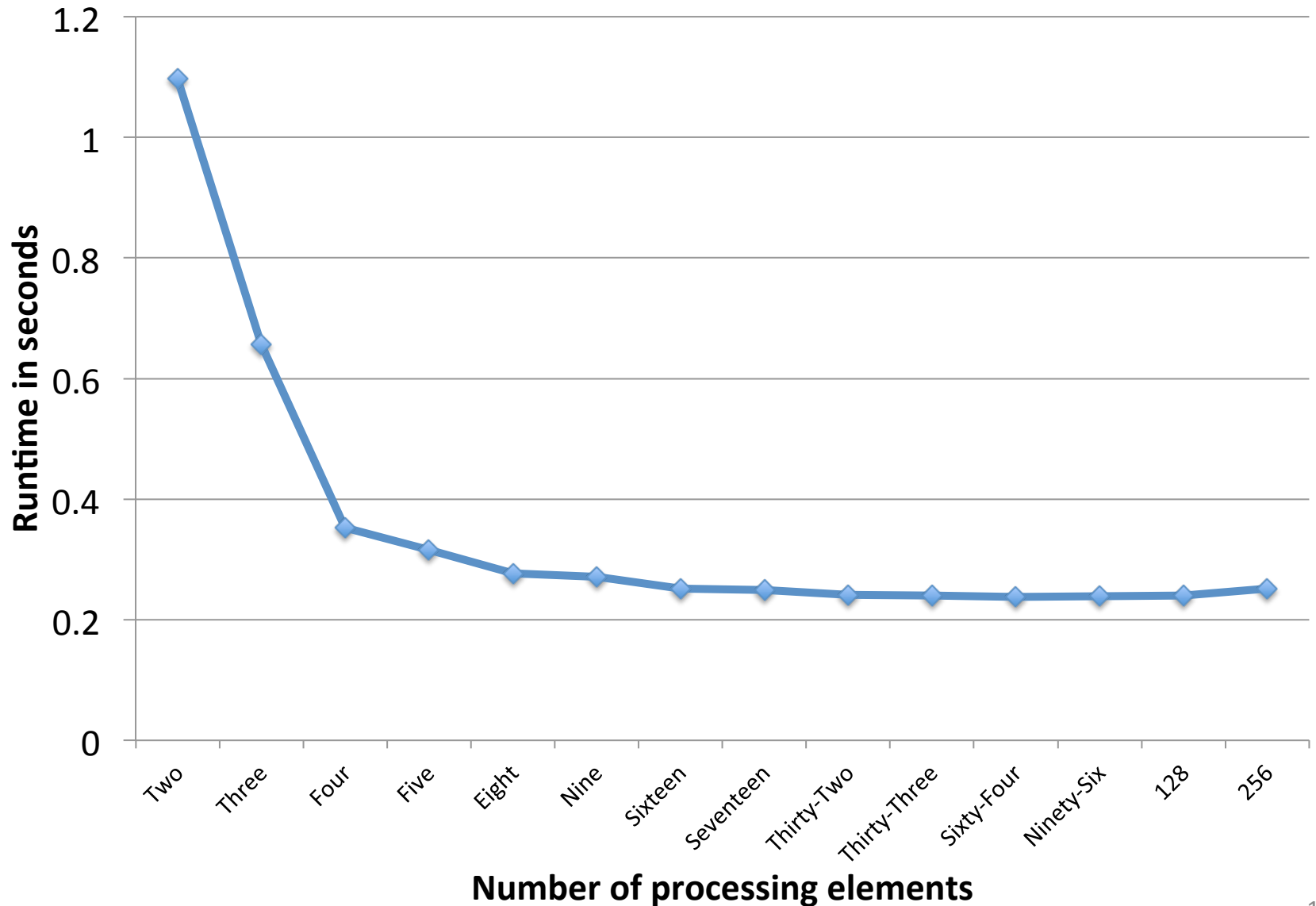
Time in seconds needed to process 5,000,000 items divided evenly among N processing elements



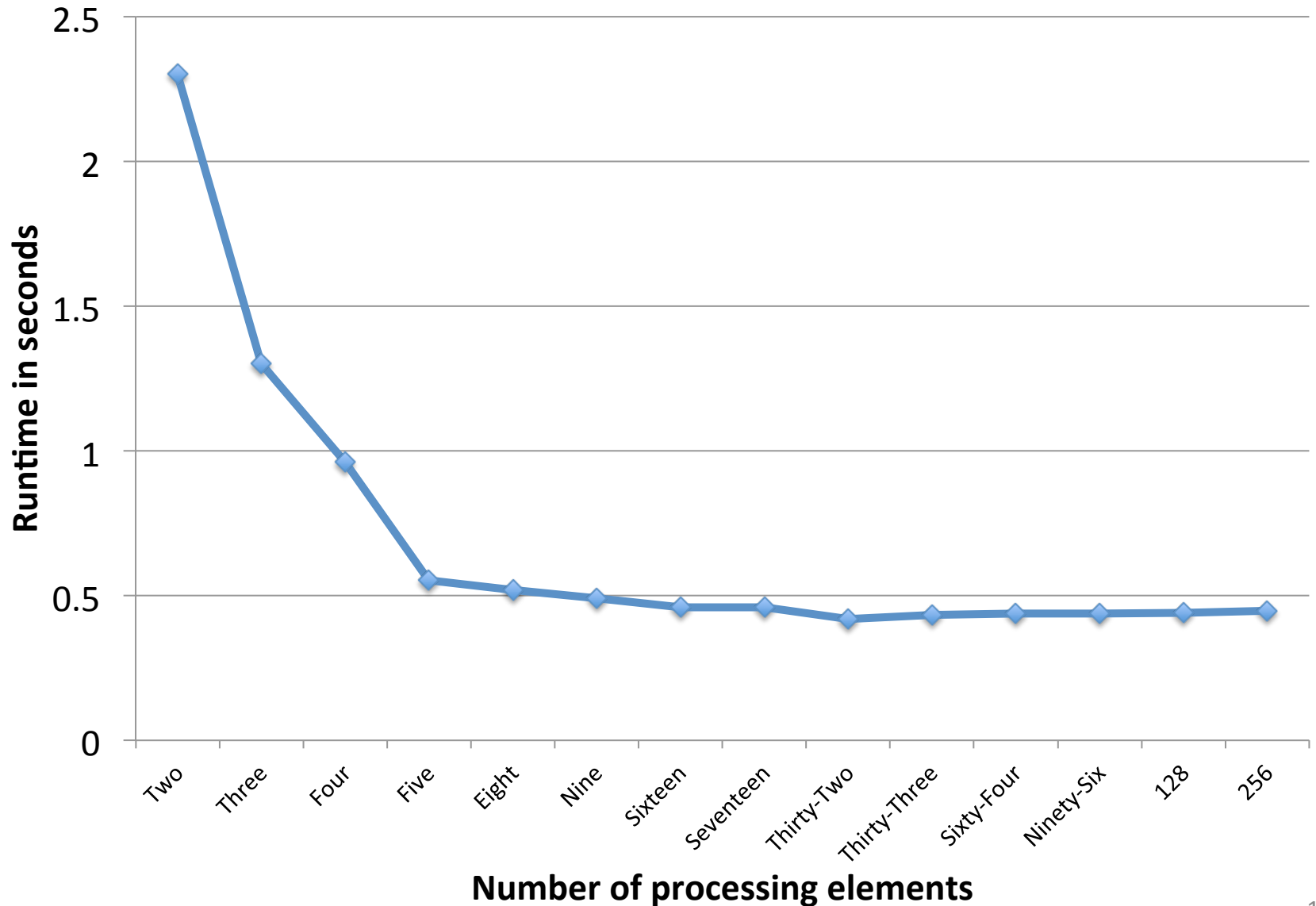
Time in seconds needed to process 10,000,000 items divided evenly among N processing elements



Time in seconds needed to process 20,000,000 items divided evenly among N processing elements



Time in seconds needed to process 30,000,000 items divided evenly among N processing elements



Parallel Postfix Implementation

- Implemented parallel postfix using MPI by modifying the parallel prefix algorithm in the Algorithms Sequential and Parallel textbook
- The original algorithm pseudo code is shown below:





```
For i = 1 to n, do in parallel
  pi.prefix = xi;
  pi.first_in_segment = i;
End For
```

```
For i = 2 to n, do in parallel
  While pi.first_in_segment > 1, do
    j = pi.first_in_segment - 1;
    pi.prefix = pj.prefix ⊗ pi.prefix;
    pi.first_in_segment = pj.first_in_segment;
  End While
End For
```

Parallel Postfix Implementation

- To successfully implement this algorithm two modifications were needed
- Modification One
 - Each processing element needs to know where they are sending their values to and where they are receiving from. Each processing element can calculate this by adding one for their receiving from their first in segment for receiving and subtract one for their sending iteration

Parallel Postfix Example

Node i					
Initial First in segment	0	1	2	3	Set up Step $\log_2(4) = 2$ iterations
j.send_to	-1	0	1	2	
j.recv_from	1	2	3	4	Iteration 1
New First in segment	1	2	3	4	
j.send_to	0	1	2	3	Iteration 2
j.recv_from	2	3	4	5	
New First in segment	3	4	5	6	

- After the first iteration, the calculation the j.recv_from is correct, but the j.send_to is incorrect
- Node 2 should be sending to Node 0 and Node 3 should be sending to Node 1

Parallel Postfix Implementation

- The second modification to the algorithm is not calculating the `j.send_to` value correctly. We can correct for this by modifying the calculation of `j.send_to`. At the end of a postfix iteration we multiply a counter initialized at one by two





$$\text{countIterations} = \text{countIterations} * 2$$

- Then to calculate the `j.send_to` value simply subtract the processing element `i`'s identification value from the `countIterations` value

$$j.\text{send_to} = \text{node}_i.\text{id} - \text{countIterations}$$

- This equation will calculate the correct processing element to send the postfix data to correcting the sending problem

Parallel Postfix Example

Node i					
Initial first in segment	0	1	2	3	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> Set up Step $\text{Log}_2(4) = 2$ iterations </div>
j.send_to	-1	0	1	2	
j.recv_from	1	2	3	4	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> Iteration 1 IterationNumber = 1 </div>
New first in segment	1	2	3	4	
j.send_to	-2	-1	0	1	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> Iteration 2 IterationNumber = 2 </div>
j.recv_from	2	3	4	5	
New first in segment	3	4	5	6	

- After the second modification, the calculation of j.send_to and j.recv_from is correct
- With this calculation correct, the postfix will run for any number iterations correctly

Parallel Postfix Pseudo Code

```
For i = 1 to n, do in parallel
    pi.postfix = xi;
    pi.first_in_segment = i;
End For

totalIterations = ceil(log2(n));
countIterations = 1;

For i = 1 to n, do in parallel
    While totalIterations > 0, do
        j.send_to = pi.id - countIterations;
        j.recv_from = pi.first_in_segment + 1;
        if(j.send_to >= 0)
        {
            send pi.postfix to pj.send_to
        }
        if(j.recv_from < n)
        {
            recv recvedprefix from j.recv_from;
            pi.prefix = pi.prefix ⊗ recvedprefix;
        }
        if(j.send_to >= 0)
        {
            send pi.first_in_segment to j.send_to
        }
        if(j.recv_from < n)
        {
            recv newFIS from j.recv_from;
            pi.first_in_segment = newFIS
        }
        totalIterations--;
        countIterations=countIterations*2;
    End While
End For
```

Parallel Postfix Results

Table of runtime in seconds with multiple dataset sizes on multiple numbers of processing elements

Number of data items to process	Two processing elements	Three processing elements	Four processing elements	Five processing elements	Eight processing elements
1,000,000	0.031092	0.022284	0.019831	0.018036	0.017211
5,000,000	0.262823	0.213076	0.216446	0.25277	0.20309
10,000,000	0.418481	0.313178	0.303652	0.383399	0.290436
20,000,000	0.95053	0.550902	0.51744	0.631492	0.501563
30,000,000	1.785445	1.160773	1.064436	1.652871	0.945759

All tests were ran on one node with 8 processing elements

Parallel Postfix Results

Table of runtime in seconds with multiple dataset sizes on multiple numbers of processing elements

Number of data items to process	Nine processing elements	Sixteen processing elements	Seventeen processing elements	Thirty-Two processing elements	Thirty-Three processing elements	Sixty-Four processing elements
1,000,000	0.019451	0.015108	0.018363	0.01571	0.016884	0.015406
5,000,000	0.2607153	0.240822	0.311957	0.223178	0.318088	0.220724
10,000,000	0.372847	0.341503	0.42765	0.31045	0.431284	0.301486
20,000,000	0.5850726	0.499397	0.69038	0.514603	0.729233	0.491735
30,000,000	1.7470815	0.9880633	1.48113	1.05995	1.63448	1.07413

Number of nodes needed:
(8 PE's each)

2 nodes

3 nodes

4 nodes

5 nodes

8 nodes

Parallel Postfix Results

Table of runtime in seconds with multiple dataset sizes on multiple numbers of processing elements

Number of data items to process	Ninety-Six processing elements	128 processing elements	256 processing elements
1,000,000	0.020267	0.021986	0.036986
5,000,000	0.239257	0.221295	0.23376
10,000,000	0.358634	0.3039986	0.3106978
20,000,000	0.551017	0.525583	0.5351053
30,000,000	1.162384	1.05421	1.111625

Number of nodes needed:
(8 PE's each)

12 nodes

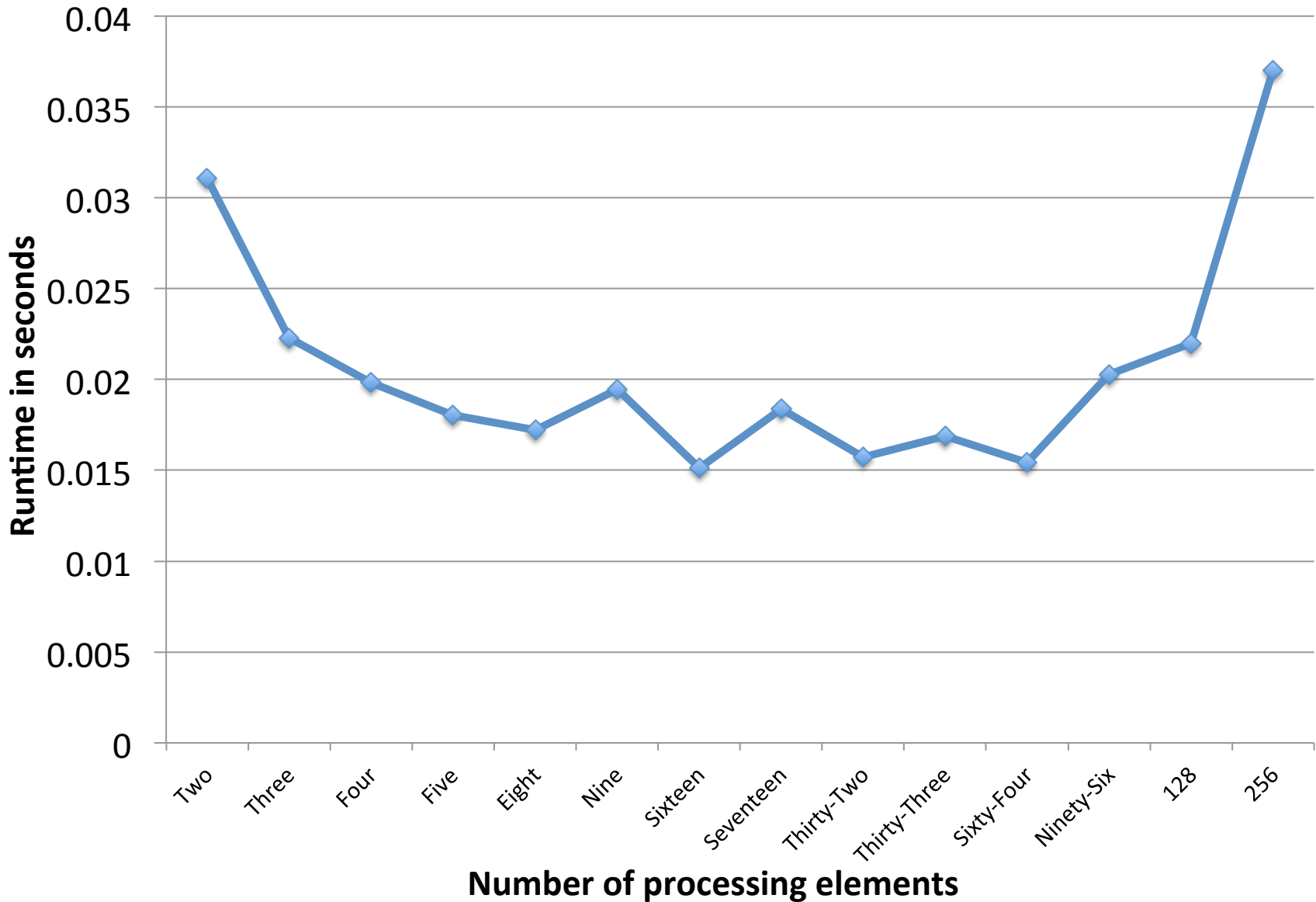
16 nodes

32 nodes

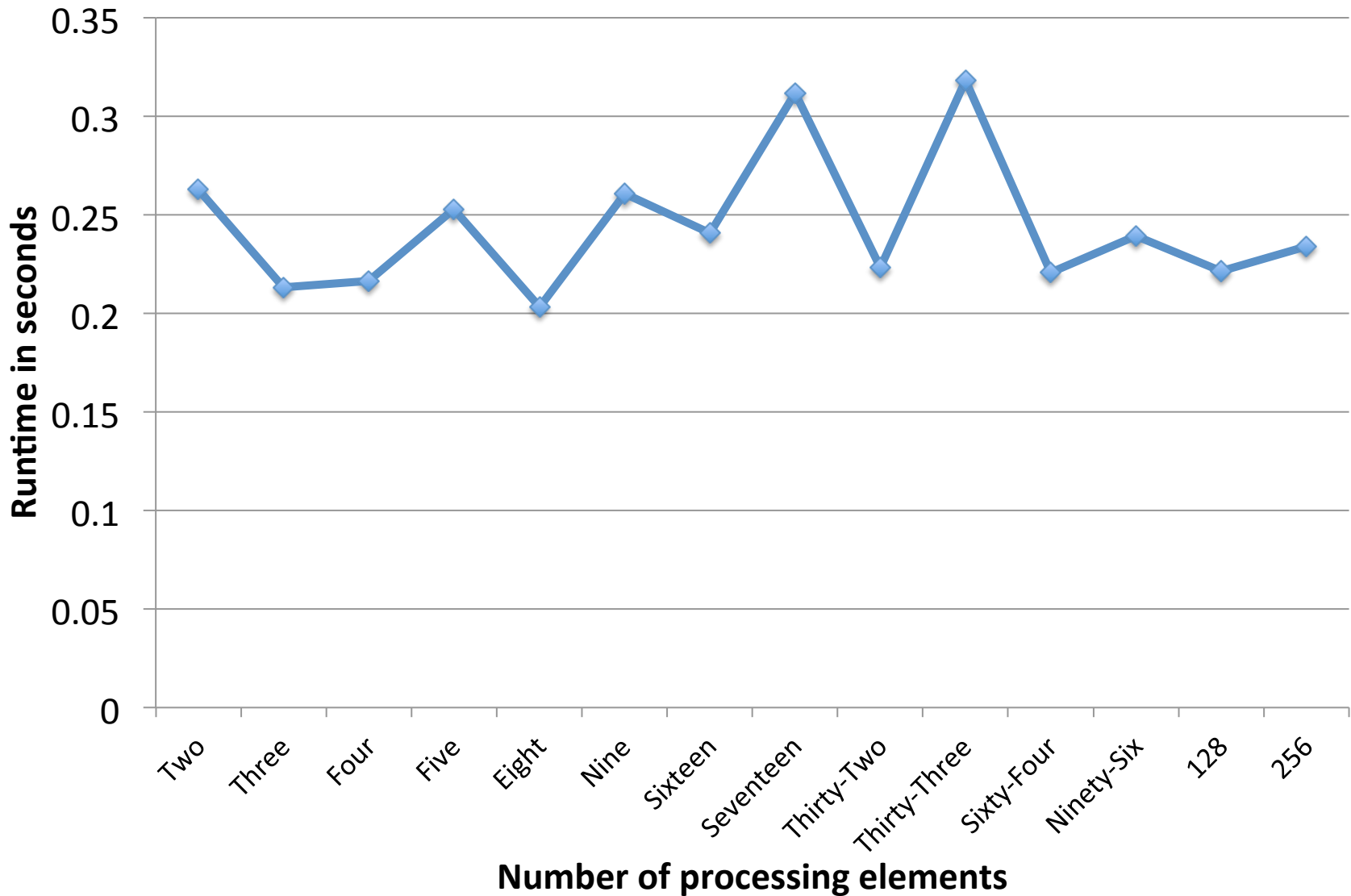
Parallel Postfix Performance Observations

- When the number of iterations calculated is not an integer value, for example $\text{Log}_2(4) = 2$, $\text{Log}_2(8) = 3$, then the runtime is longer because an extra postfix is required
- When inter-node communication is needed then the running time is longer. This occurs because the processing elements need to communicate between nodes in later iterations of the postfix
- The runtime on larger processing element tests run longer because of barriers in the algorithm that ensure that the processing elements are at the same step of the postfix

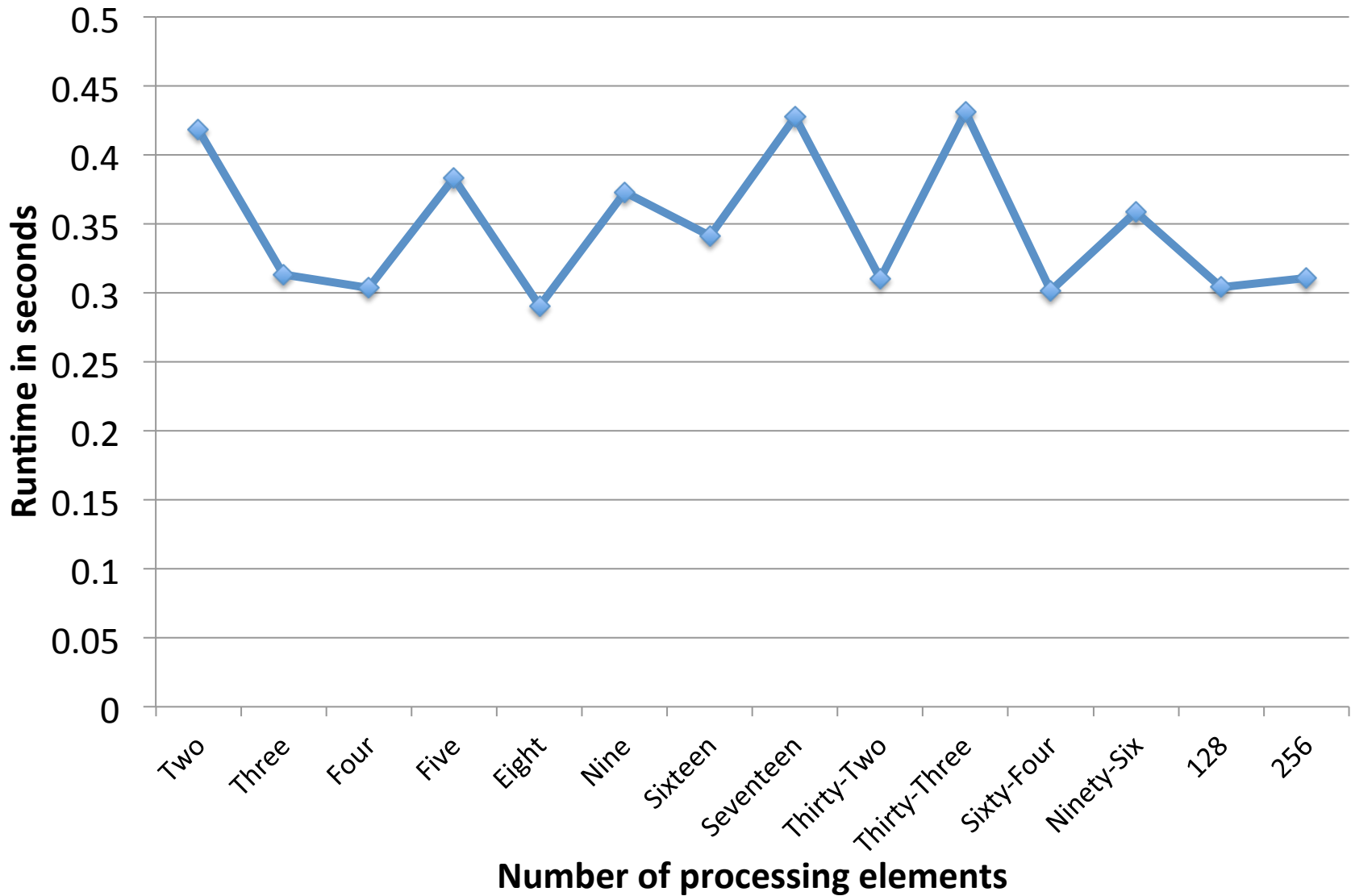
Time in seconds needed to process 1,000,000 items divided evenly among N processing elements



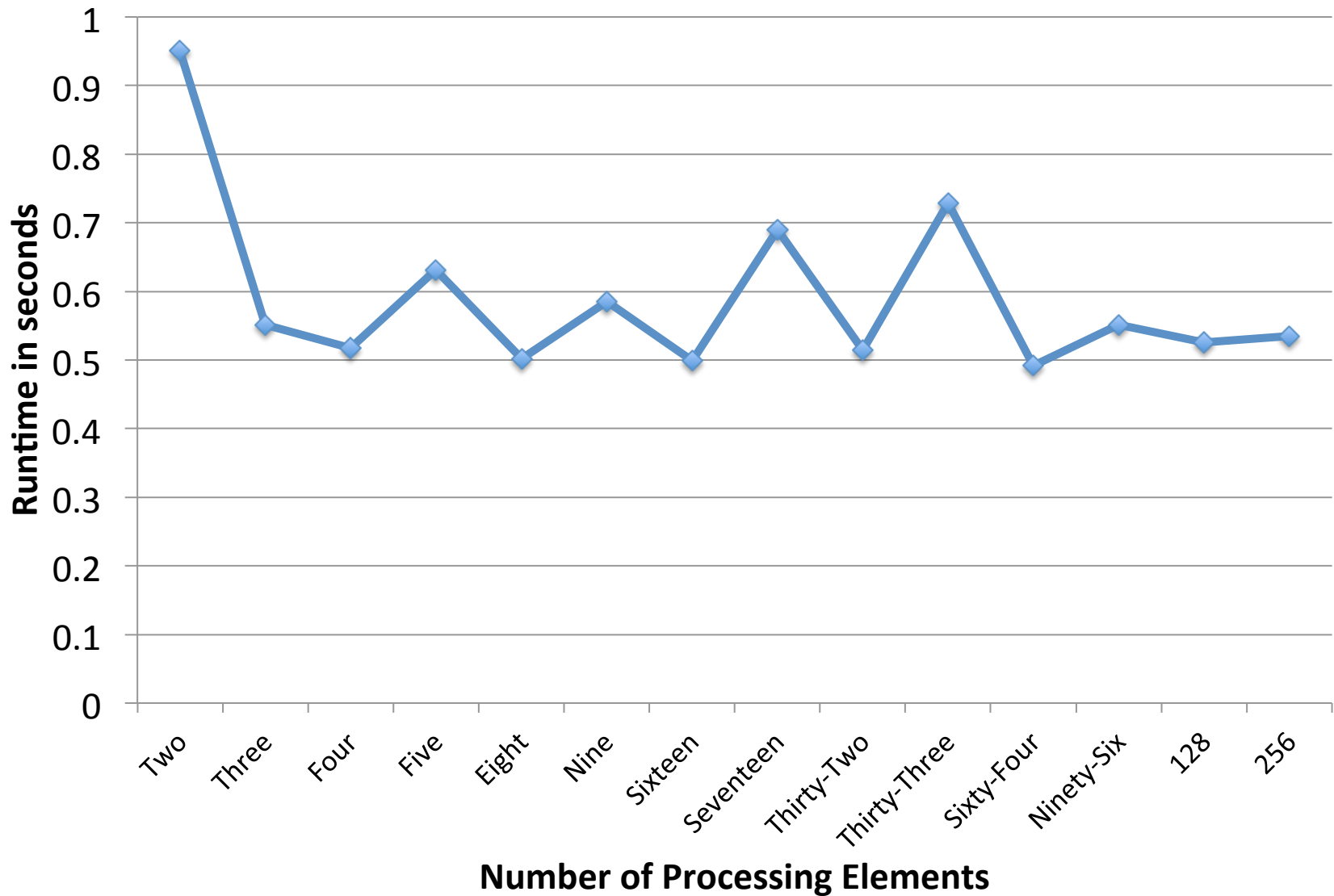
Time in seconds needed to process 5,000,000 items divided evenly among N processing elements



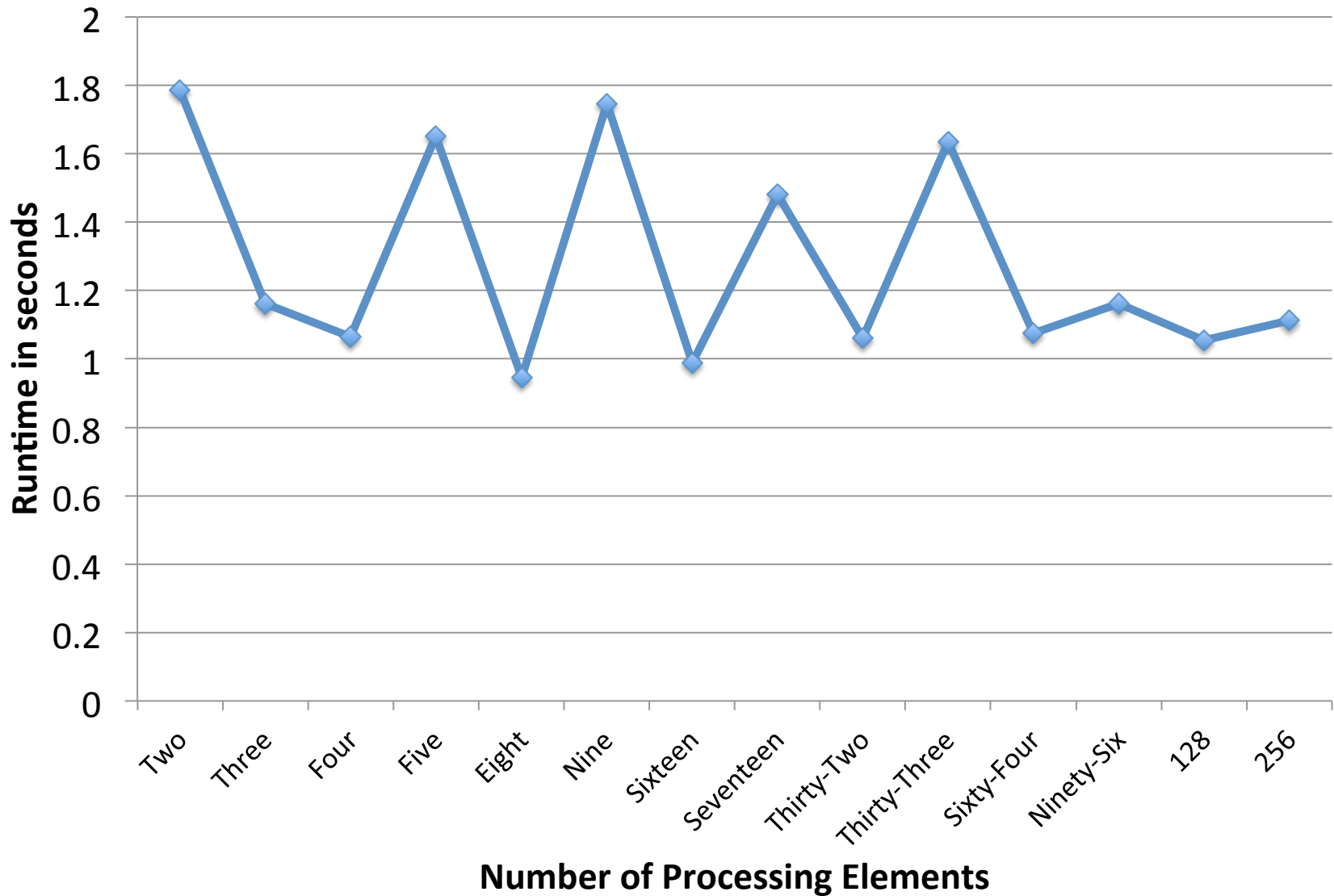
Time in seconds needed to process 10,000,000 items divided evenly among N processing elements



Time in seconds needed to process 20,000,000 items divided evenly among N processing elements



Time in seconds needed to process 30,000,000 items divided evenly among N processing elements



Final Results

- For both the parallel postfix and master/worker implementation immediate algorithm runtime benefits can be seen when the number of processing elements working on the problem increases
- For all of the test cases the master/worker implementation is faster than the parallel postfix implementation. This is most likely because the parallel prefix algorithm requires multiple iterations to achieve a final answer
- But, the parallel prefix algorithm calculates every individual prefix on every processing element where the master/worker algorithm just calculates the final result

Future Work

- Distribute and collect the dataset using MPI Scatter and Gather
 - Currently using sends and receives to distribute the portions of the data set to the nodes
- Implement the algorithm in OpenMP and compare the performance of the two solutions

Questions?

References

- R. Miller and L. Boxer, *Algorithms Sequential and Parallel, A Unified Approach*, Prentice Hall, Upper Saddle River, NJ (2000).

Thank you!