Determining Line Segment Visibility with MPI

CSE 633: Parallel Algorithms
Fall 2012

Jayan Patel

Problem Definition

- Computational Geometry
- From *Algorithms Sequential and Parallel*:
  - Given a set of *n* pair-wise disjoint line segments in the first quadrant of the Euclidean plane, each of which has one of its endpoints on the x-axis, compute the piece of each line segment that is observable from the origin.

Problem Definition

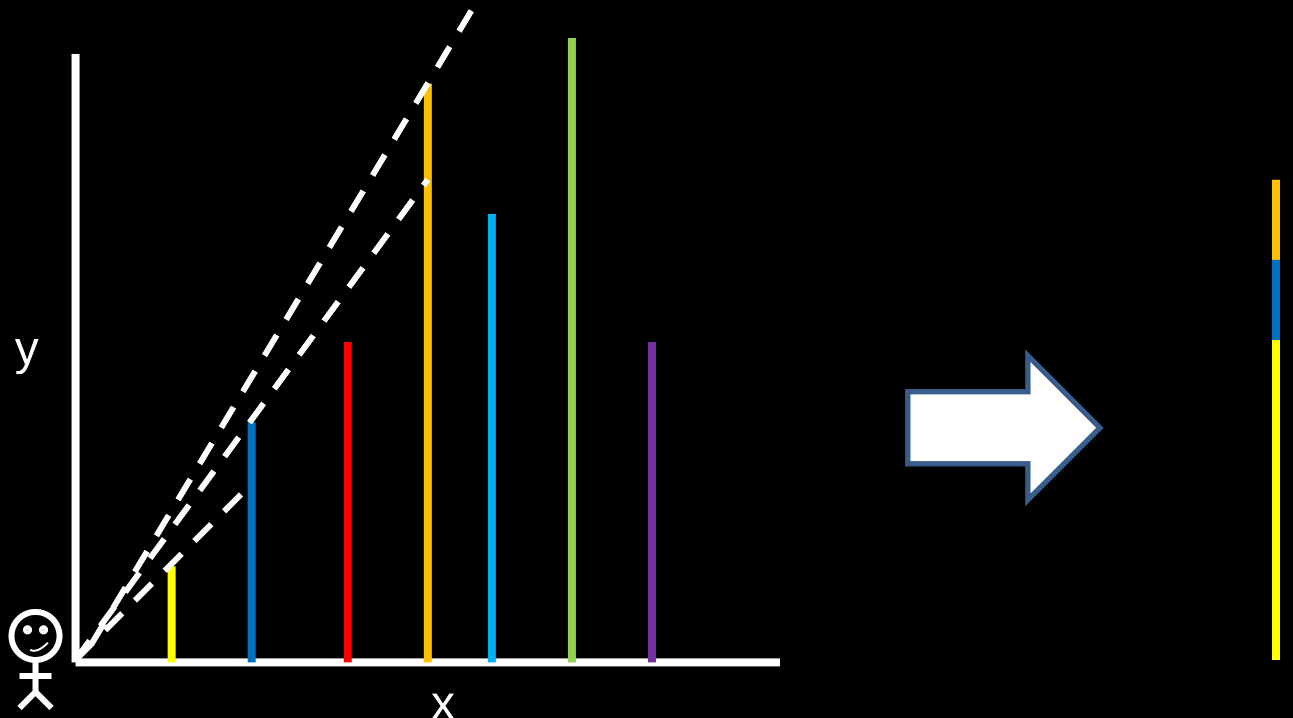
- Assumptions:
  - The input data is ordered from left to right (i.e., by increasing *x* values).
  - The viewer does not have X-ray vision.

Example

- Atlanta City Skyline

Source: http://en.wikipedia.org/wiki/File:Atlanta_Skyline_from_Buckhead.jpg

Example



Practical Application

- Simple form of occlusion culling
  - Popular optimization for graphics-based applications
- Parallelization is critical to achieve acceptable real-time performance
  - Rendering pipeline

Implementation Plan

- Code in C/C++
- Develop serial (RAM) algorithm for benchmarking and verification
- Develop parallel algorithm using MPI

Implementation Plan

- Input
  - Single binary file containing *n* (*x*,*y*) pairs
  - Sorted by increasing *x*

- Output
  - Single binary file containing triplets of the form (*x*,*y*,*visibleLength*), where *visibleLength* > 0
  - Sorted by increasing *x*

Serial (RAM) Algorithm

```
1. Read binary file into arrays $x$ and $y$
2. $slope[1] = y[1] / x[1]$
3. Push ($x[1], y[1], y[1]$) onto results queue
4. For $i = 2$ to $n$, do
   a. $slope[i] = \max(slope[i-1], y[i] / x[i])$
   b. If $slope[i] > slope[i-1]$ then
      i.  $visibleLength = y[i] - slope[i-1] * x[i]$
      ii. Push($x[i], y[i], visibleLength$) onto results queue
5. Write each result to output file
```

- $\Theta(n)$ time

Parallel Algorithm

```
1. In parallel, each of $p$ processors, do
   a. Read block from data file into arrays $x$ and $y$
   b. $slope[1] = y[1] / x[1]$
   c. $P_0$ only: Push($x[1]$, $y[1]$, $y[1]$) onto results queue
   d. For $i = 2$ to $n/p$, do
      i. $slope[i] = \max(slope[i-1], y[i] / x[i])$
```

- $\Theta(n/p)$ time

Parallel Algorithm

2. In parallel, each of *p* processors, do
    a. Compute global parallel prefix (operation = maximum) for the set of *p* right-most prefixes

---

- Using PRAM-like recursive doubling process, requires *Θ(log(p))* iterations of simultaneous MPI operations

Parallel Algorithm

```
3. In parallel, each of $p$ processors, do
   a. If not $P_0$ then
      i.   prevSlope = global prefix from $P_{k-1}$
      ii.  slope[1] = max(prevSlope, slope[1])
      iii. if slope[1]  > prevSlope then push result
   b. For $i$ = 2 to $n/p$, do
      i.   If slope[i] > slope[i-1] then
           a. visibleLength = y[i] - slope[i-1] * x[i]
           b. push(x[i], y[i], visibleLength) onto
              results queue
      ii.  else slope[i] = slope[i-1]
```

- $\Theta(n/p)$ time

Parallel Algorithm (cont.)

4. In parallel, each of *p* processors, do
   a. Write each result to unique output file
   b. Enter barrier

5. P_0 concatenates the results files in processor order

- $\Theta(n/p)$ time (worst case)

Test Plan

- Vary size of data set
- Vary number of processes
  - 12- core Dell compute nodes will be used
- Measure running time, compute speedup
- Tabulate and graph results
- Explain trends

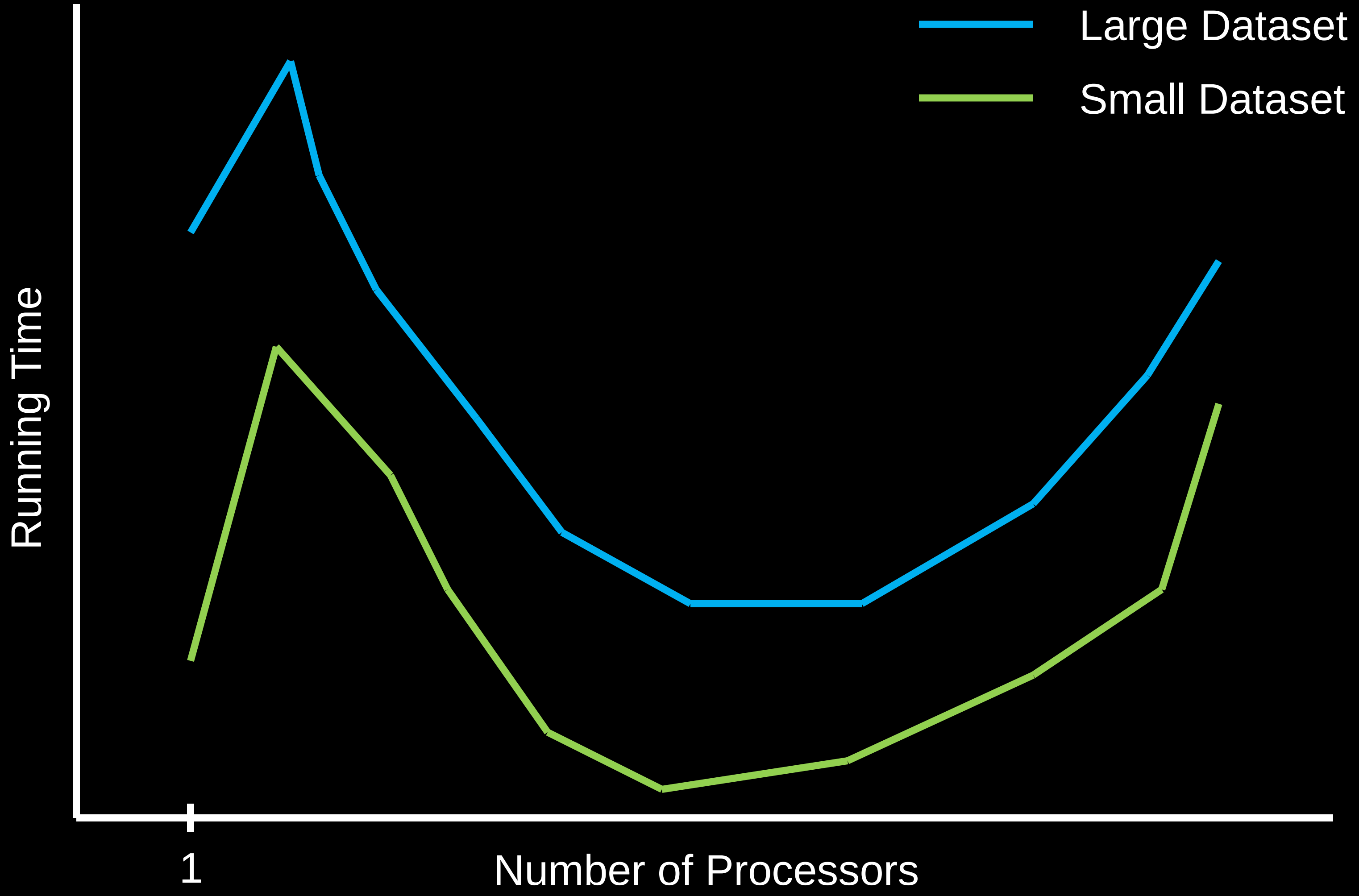
Expectations

- For fixed data size:
  - Small number of processors will have slower run times than RAM algorithm
  - Increasing the number of processors will eventually lead to reduced execution times
  - Eventually inter-processor communication will come into play

Expectations

- For fixed number of processors:
  - RAM will perform best for smaller data sizes
  - As data size increases, performance of parallel algorithm will exceed RAM
  - Large data sizes will be required to see parallel performance exceed RAM performance

Expectations



Implementation

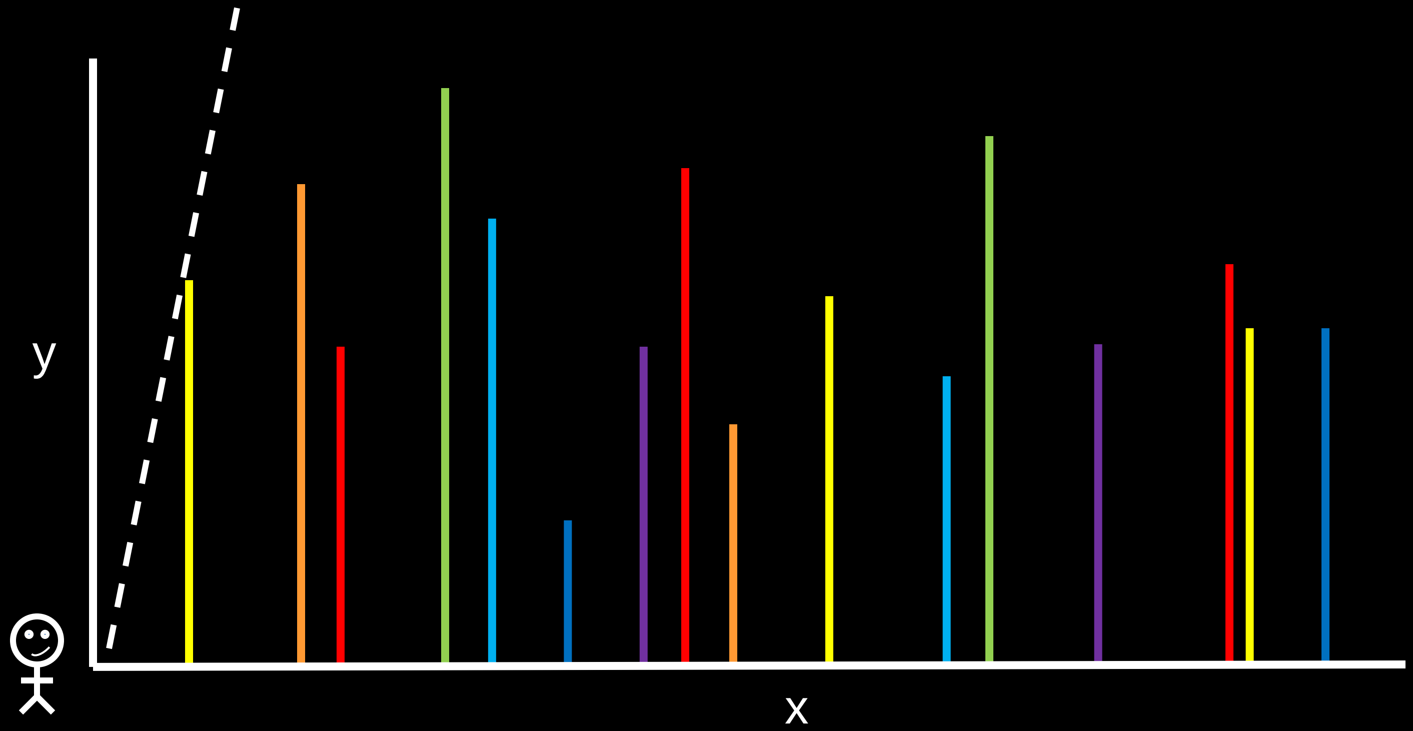
Implementation Outline

- Generation of Input Data
- Details of Parallel Approach
- Measurements and Test Setup
- Results

Input Data – Initial Approach

- Separate program
- Randomly generate pairs of numbers representing *x* and *y* coordinates
- Sort data by increasing *x* coordinate
- Write data to binary data file as (*x*, *y*) pairs

Input Data – Initial Approach



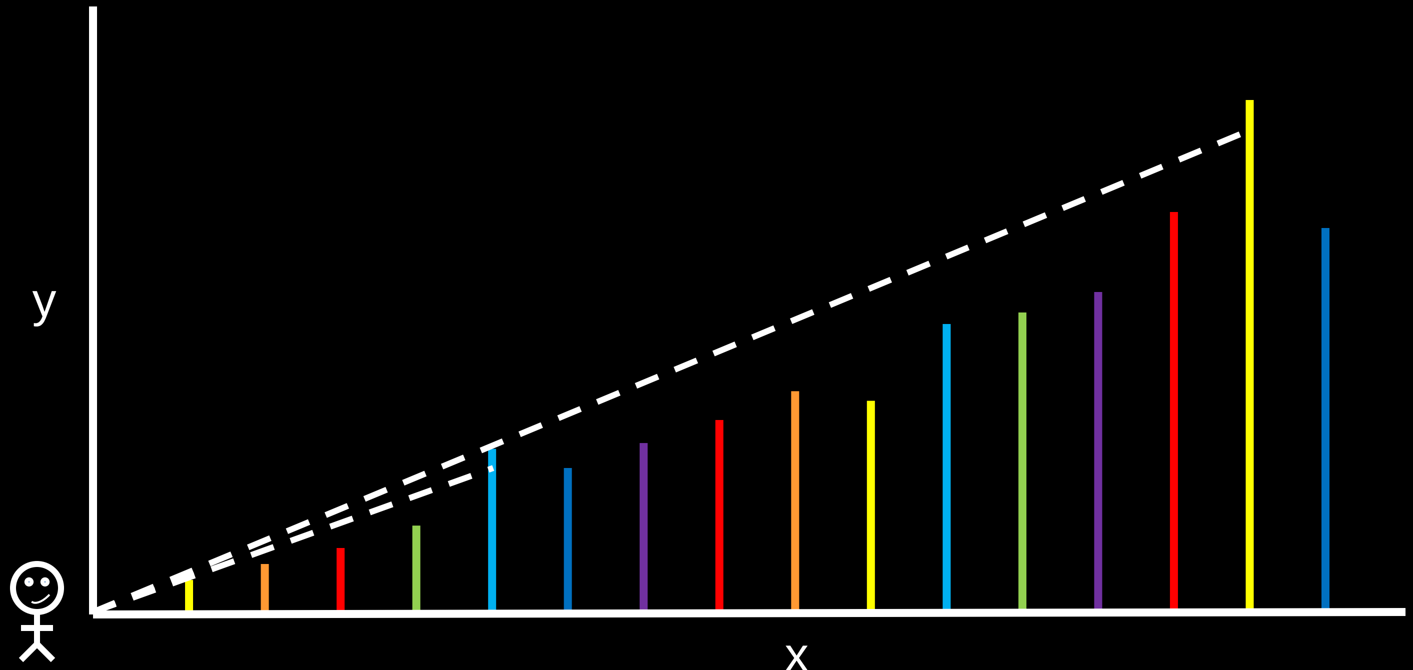
Input Data – Issues

- Very few visible line segments
- All results allocated to first process
- Uninteresting

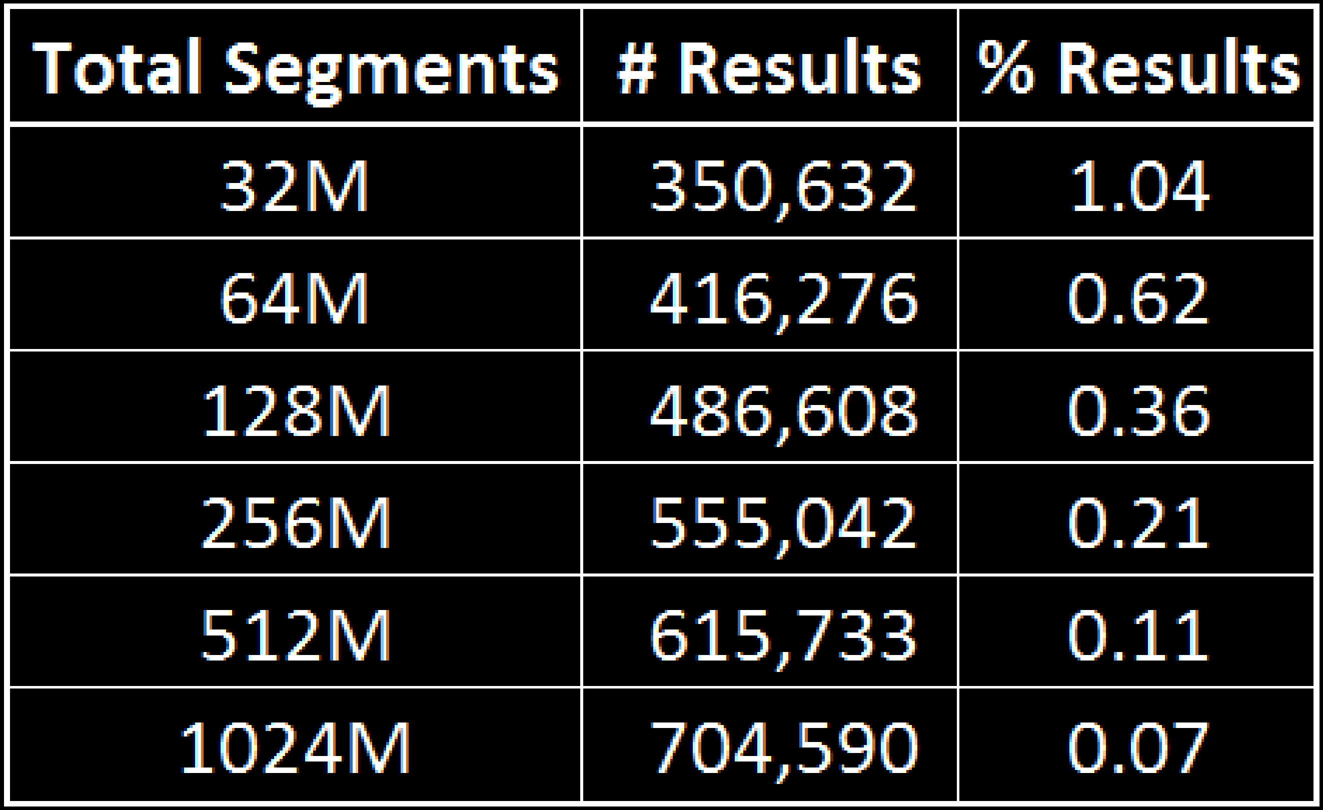
Input Data – Revised Approach

- Separate program
- Generate data in order of increasing *x* coordinate
  - Limit the increase in successive *x* coordinates
  - Factor in current index when generating *y* coordinate
- Write data to binary data file as (*x*, *y*) pairs

Input Data – Revised Approach

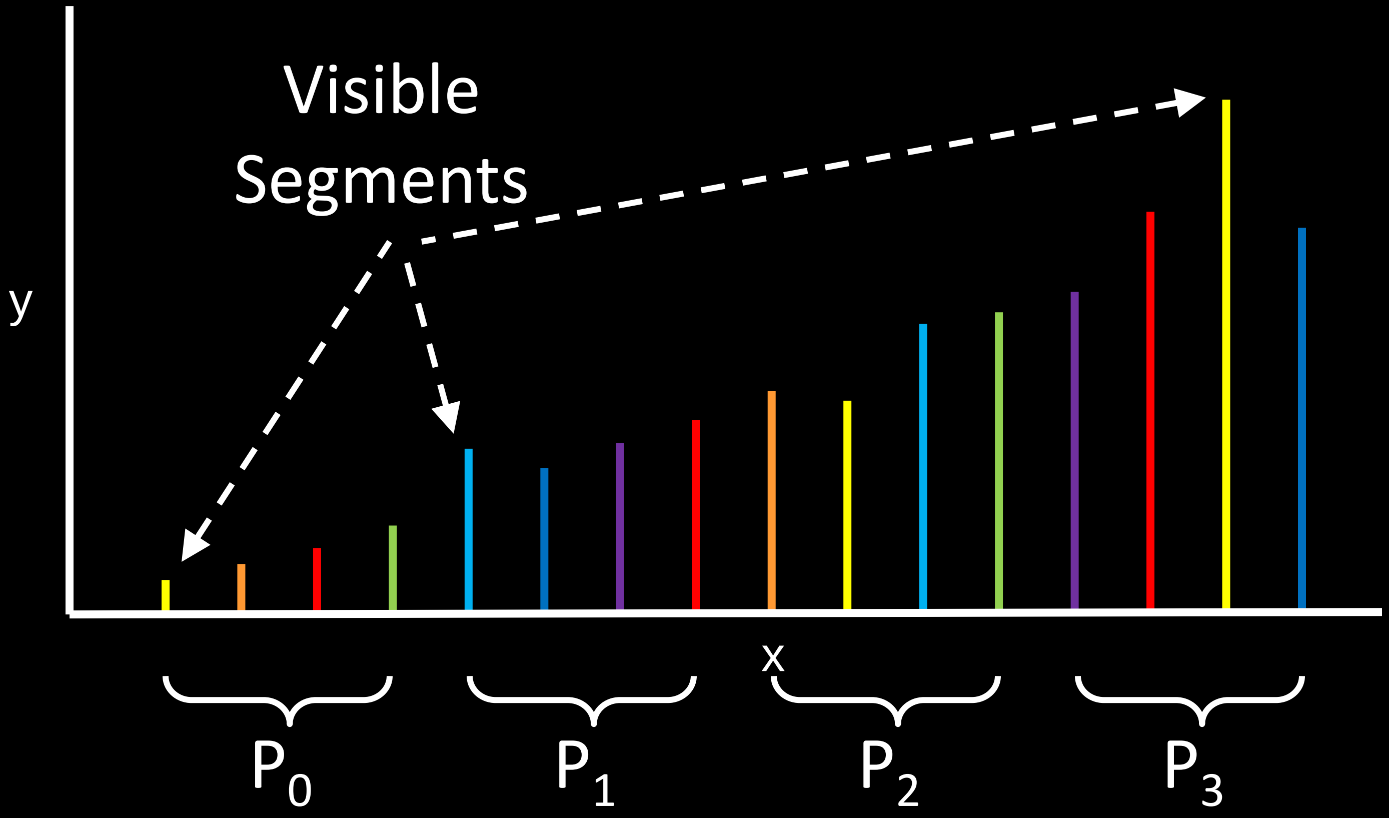


Input Data – Summary



Total Segments	# Results	% Results
32M	350,632	1.04
64M	416,276	0.62
128M	486,608	0.36
256M	555,042	0.21
512M	615,733	0.11
1024M	704,590	0.07

Parallel Approach



Desired Results: Segments 1, 5, and 15

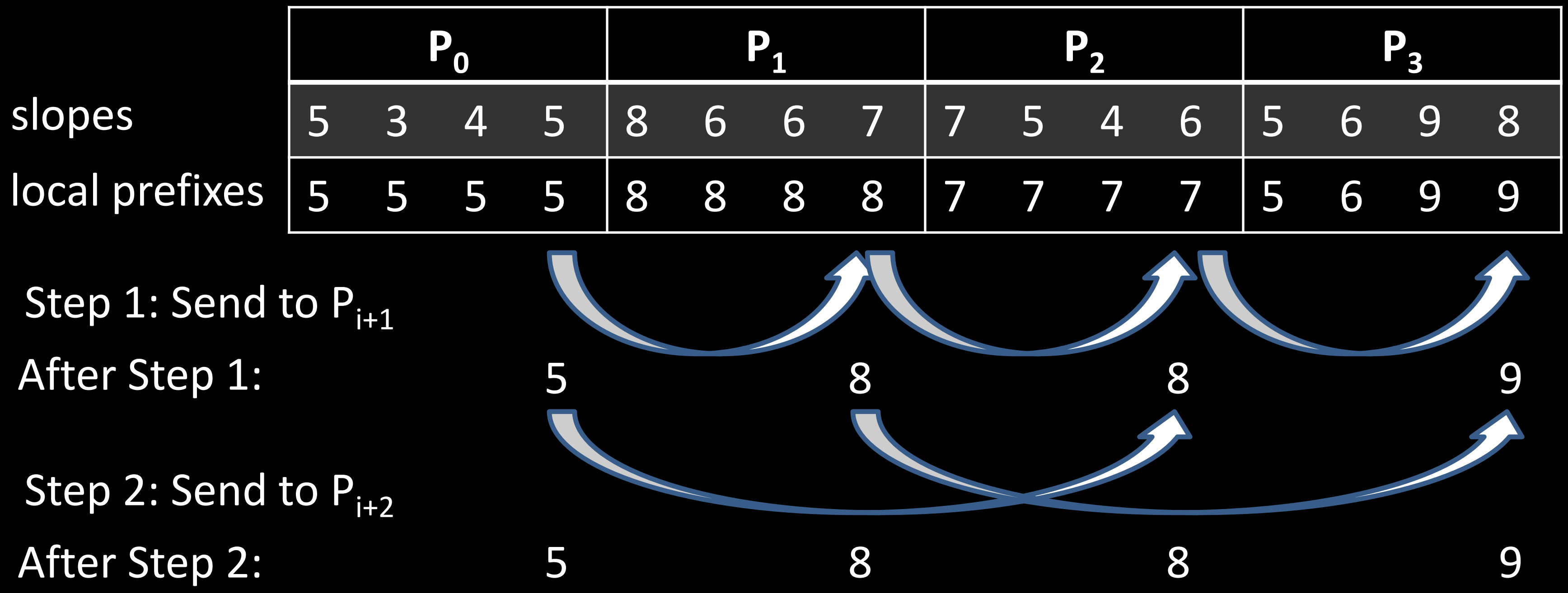
Parallel Approach – Step 1

- In parallel, read input binary data file and compute local parallel prefix
  - Operation = max(slope)

	P_0				P_1				P_2				P_3			
slopes	5	3	4	5	8	6	6	7	7	5	4	6	5	6	9	8
local prefixes	5	5	5	5	8	8	8	8	7	7	7	7	5	6	9	9

Parallel Approach – Step 2

- In parallel, compute global prefixes for right-most local prefixes
  - Operation = max(slope)



	P_0				P_1				P_2				P_3			
slopes	5	3	4	5	8	6	6	7	7	5	4	6	5	6	9	8
local prefixes	5	5	5	5	8	8	8	8	7	7	7	7	5	6	9	9

Step 1: Send to P_{i+1}

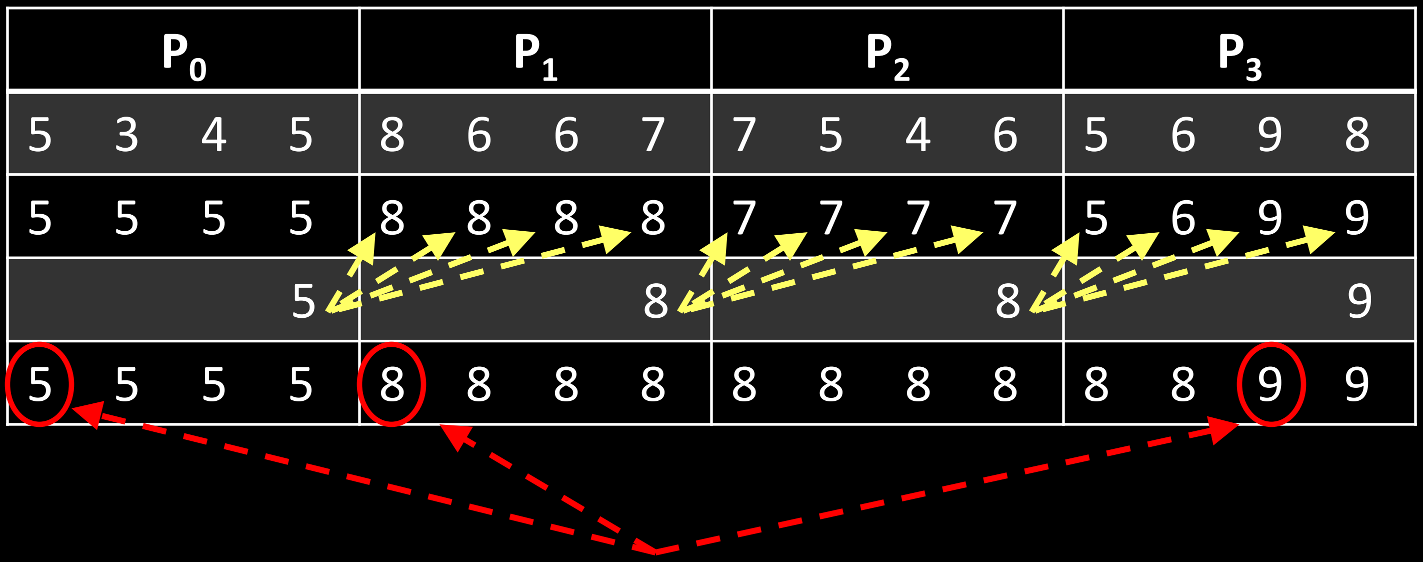
After Step 1: 5 8 8 9

Step 2: Send to P_{i+2}

After Step 2: 5 8 8 9

Parallel Approach – Step 3

- In parallel, distribute global prefix locally and push results
  - Operation = max(slope)

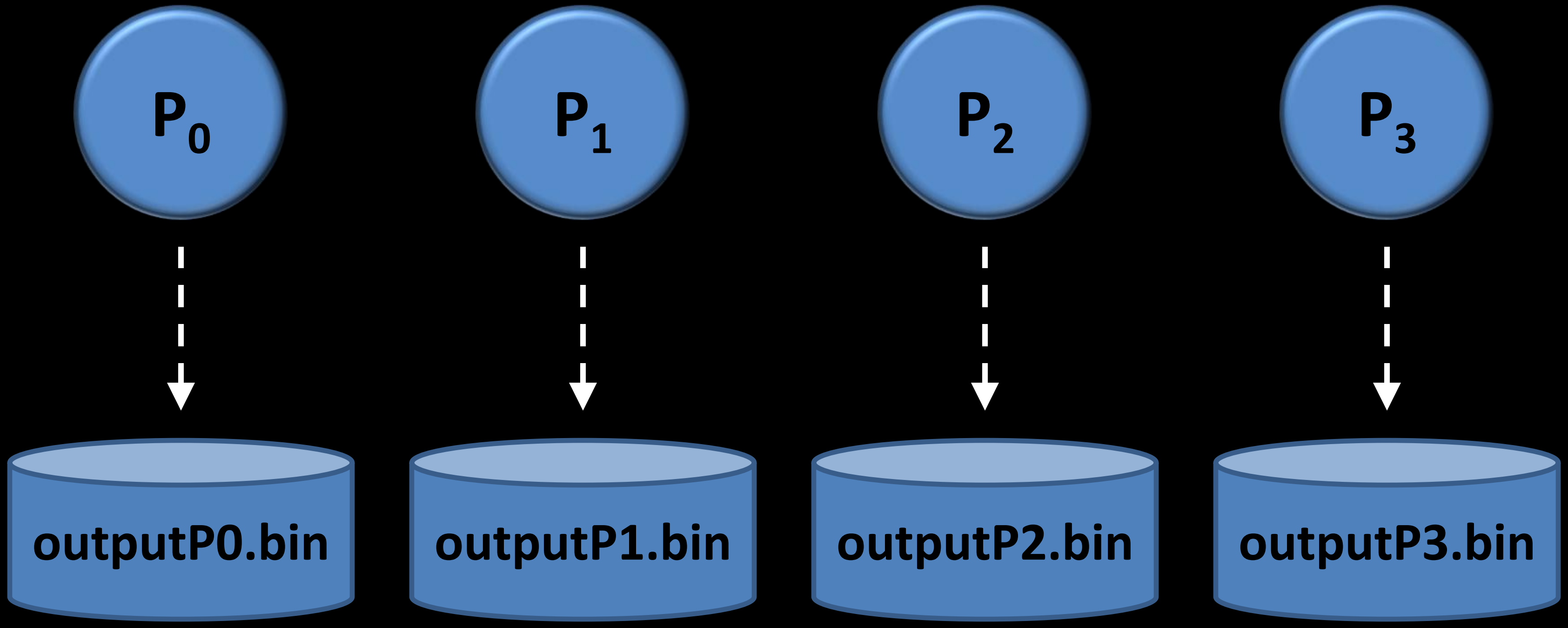


	P_0				P_1				P_2				P_3			
slopes	5	3	4	5	8	6	6	7	7	5	4	6	5	6	9	8
local prefixes	5	5	5	5	8	8	8	8	7	7	7	7	5	6	9	9
global prefixes				5				8				8				9
final prefixes	5	5	5	5	8	8	8	8	8	8	8	8	8	8	9	9

results

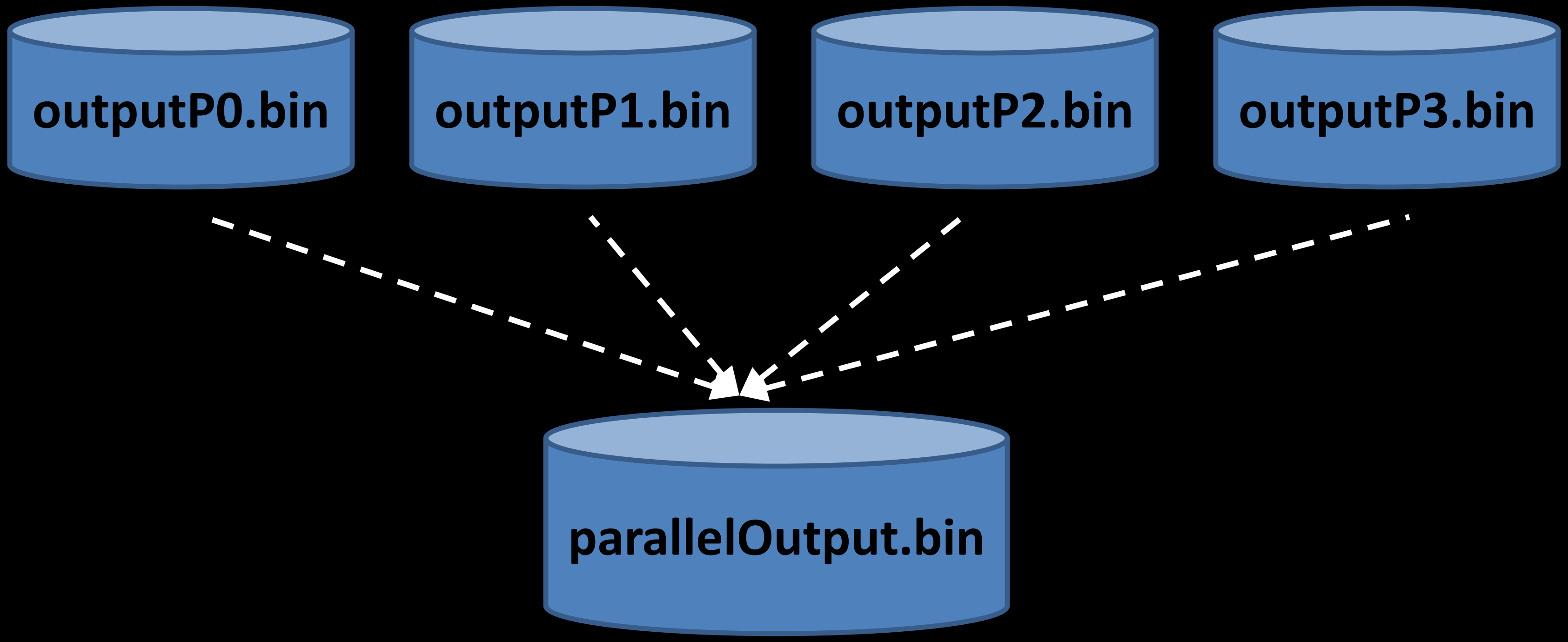
Parallel Approach – Step 4

- In parallel, write each result to unique binary output file



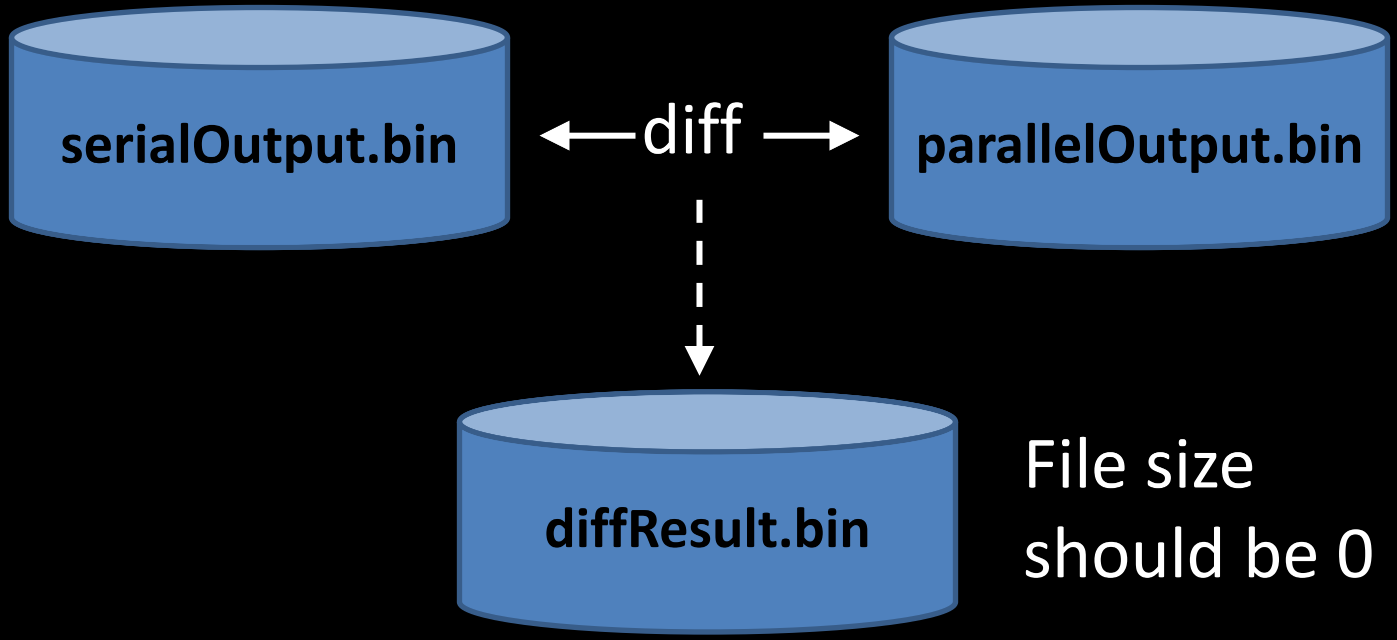
Parallel Approach – Step 5

- Combine results from each processor into a single binary output file
  - Used pbs-hydra script



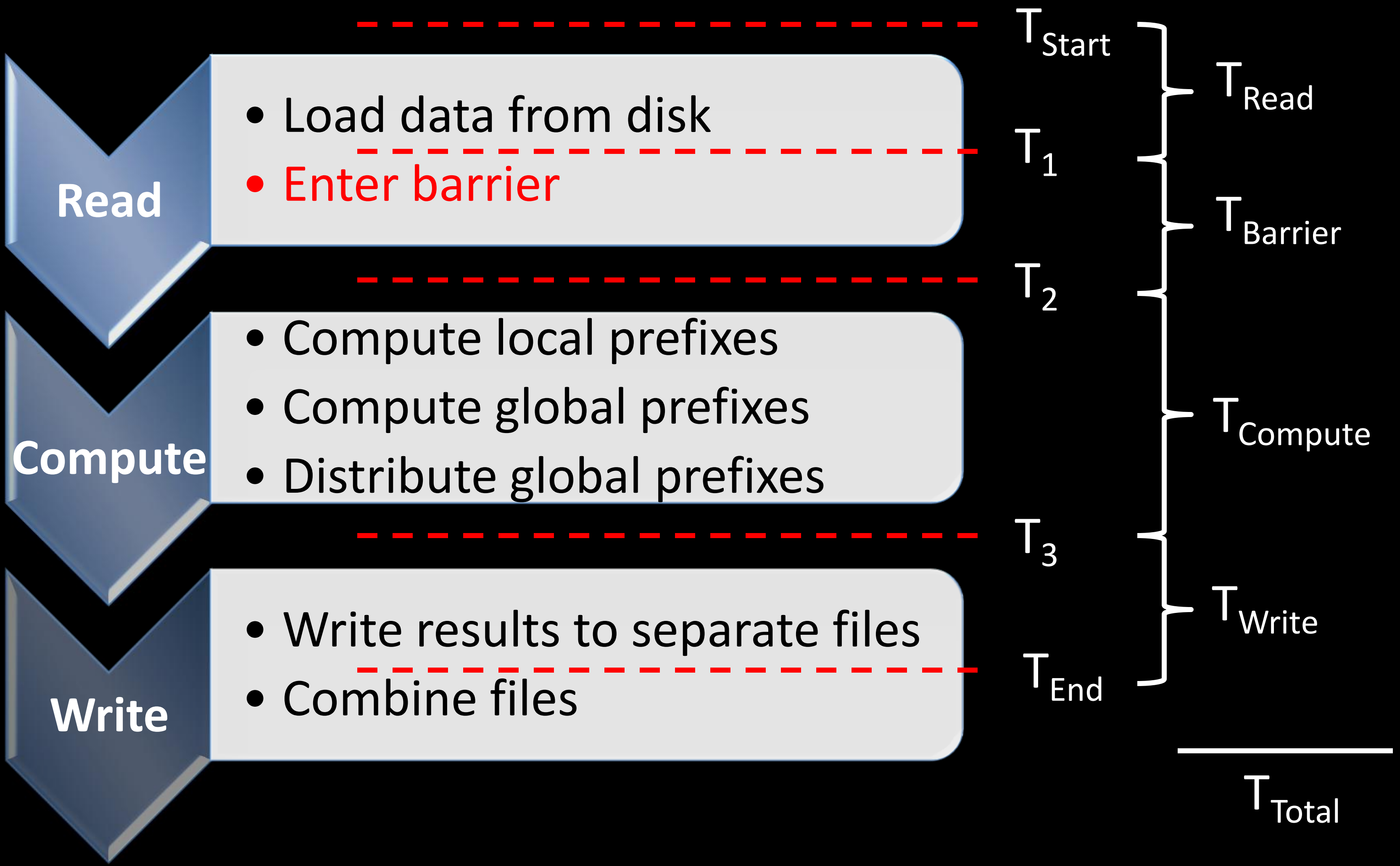
Parallel Approach – Verification Step

- Compare the parallel output file to the serial output file
  - Used pbs-hydra script



File size should be 0

Timing Measurements



Read

- Load data from disk
- Enter barrier

Compute

- Compute local prefixes
- Compute global prefixes
- Distribute global prefixes

Write

- Write results to separate files
- Combine files

T_{Start}

T_1

T_2

T_3

T_{End}

T_{Read}

$T_{Barrier}$

$T_{Compute}$

T_{Write}

T_{Total}

Testing Details

- For each test in each process, T_{Read}, $T_{Barrier}$, $T_{Compute}$, and T_{Write} were measured.
    - MPI_Reduce was used to record the min and max of each of these in P_0 for the run.
- Each test for each test configuration was repeated 100 times.
    - Min and max for T_{Read}, $T_{Barrier}$, $T_{Compute}$, and T_{Write} were tracked across all 100 runs.
    - Averages for T_{Read}, $T_{Barrier}$, $T_{Compute}$, and T_{Write} were also computed.

Testing Configuration Summary

- Used minimum number of 12-core compute nodes required to support one process per core
  - 2.40 GHz, 48 GB RAM, Infiniband (QL) network

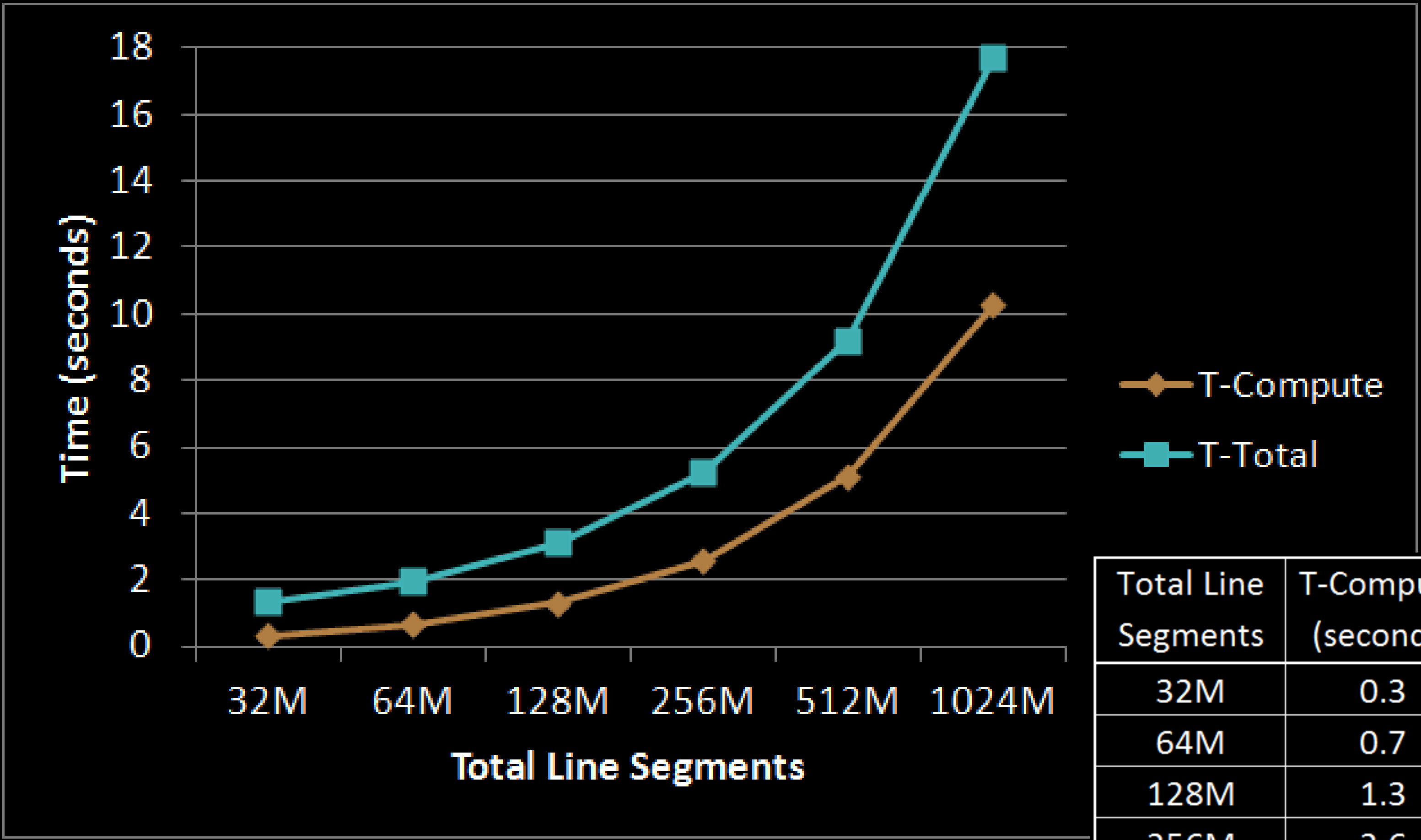
- Recorded min, max, and average for T_{Read}, $T_{Barrier}$, $T_{Compute}$, and T_{Write} for each run

Num Processes

Num Segments

	1	2	4	8	16	32	64	128	256
32M									
64M									
128M									
256M									
512M									
1024M									

Serial (RAM) Results



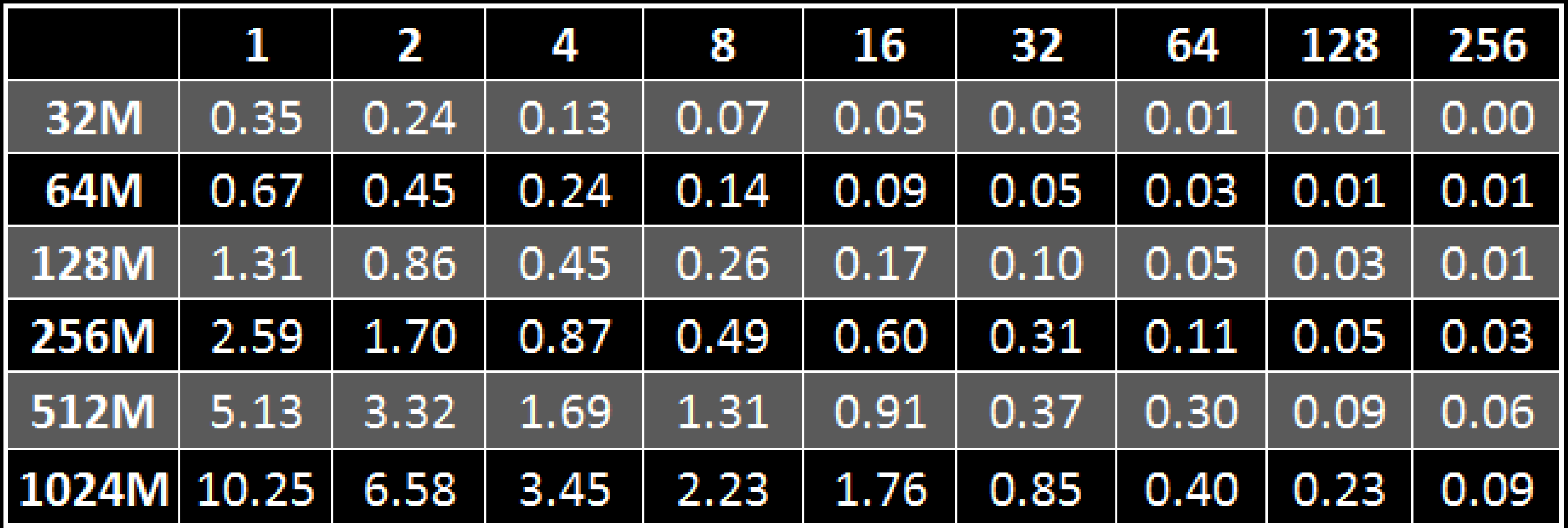
Total Line Segments	T-Compute (seconds)	T-Total (seconds)
32M	0.3	1.3
64M	0.7	2.0
128M	1.3	3.1
256M	2.6	5.2
512M	5.1	9.2
1024M	10.3	17.7

Parallel Results – $T_{Compute}$

- Compute Time (seconds)

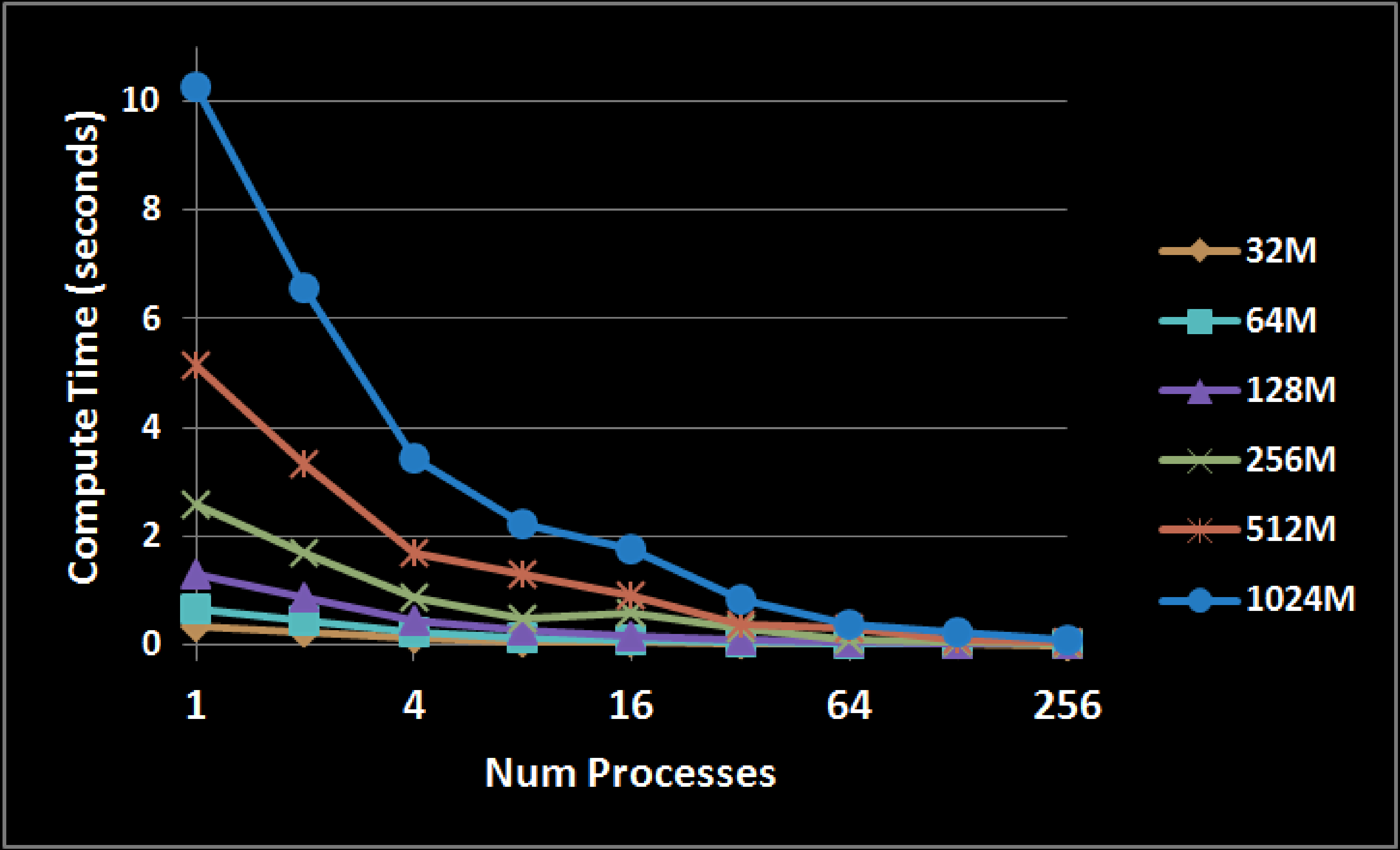
Num Processes

Num Segments

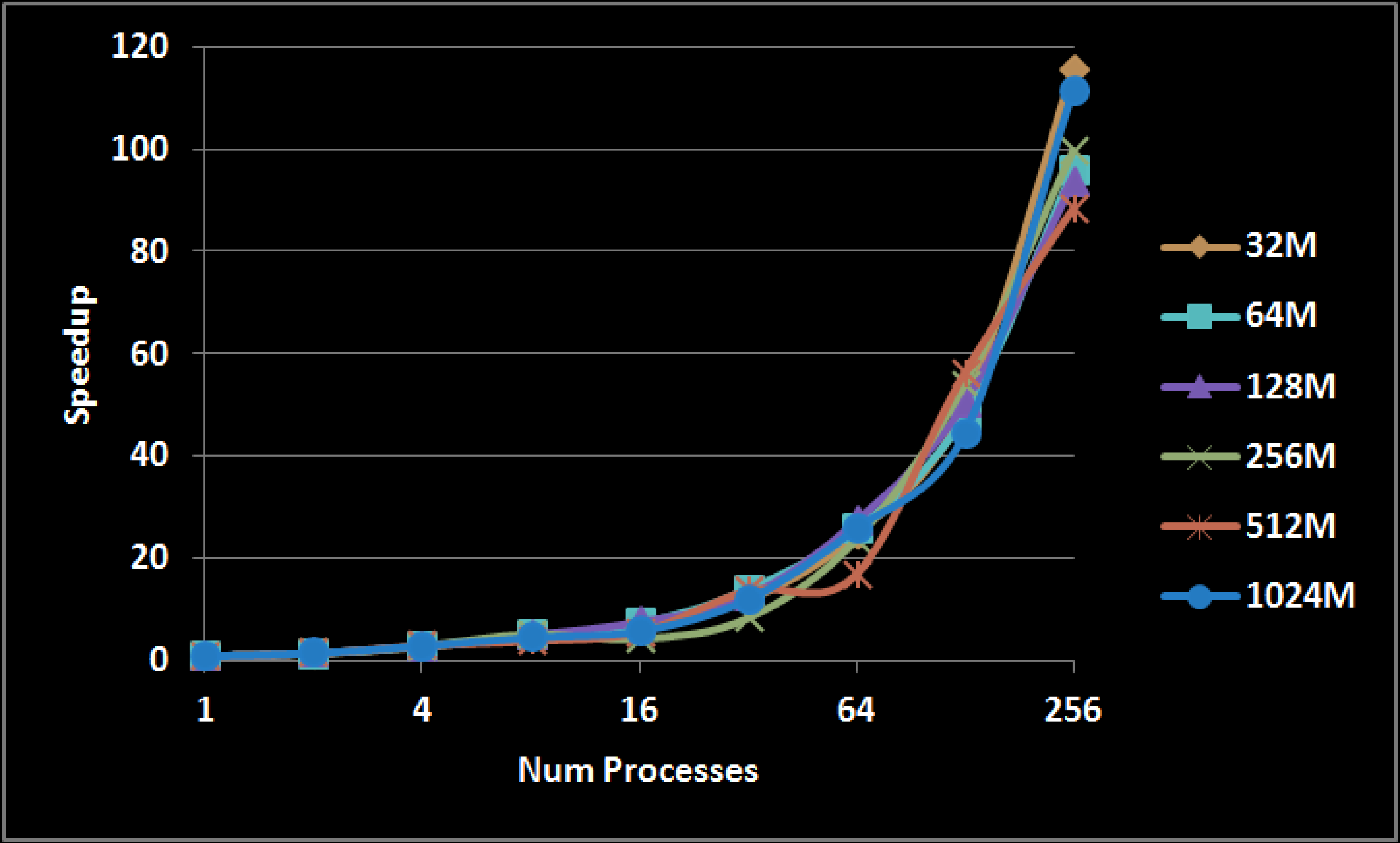


	1	2	4	8	16	32	64	128	256
32M	0.35	0.24	0.13	0.07	0.05	0.03	0.01	0.01	0.00
64M	0.67	0.45	0.24	0.14	0.09	0.05	0.03	0.01	0.01
128M	1.31	0.86	0.45	0.26	0.17	0.10	0.05	0.03	0.01
256M	2.59	1.70	0.87	0.49	0.60	0.31	0.11	0.05	0.03
512M	5.13	3.32	1.69	1.31	0.91	0.37	0.30	0.09	0.06
1024M	10.25	6.58	3.45	2.23	1.76	0.85	0.40	0.23	0.09

Parallel Results – $T_{Compute}$



Parallel Results – $T_{Compute}$

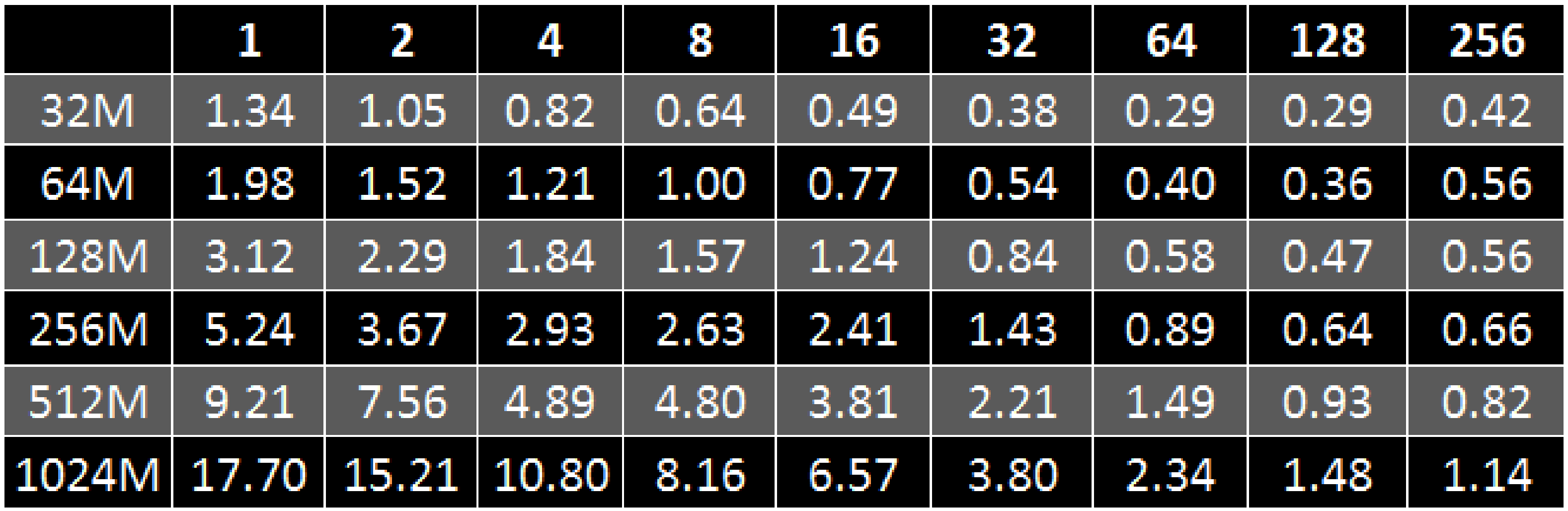


Parallel Results – T_{Total}

- Total Time (seconds)

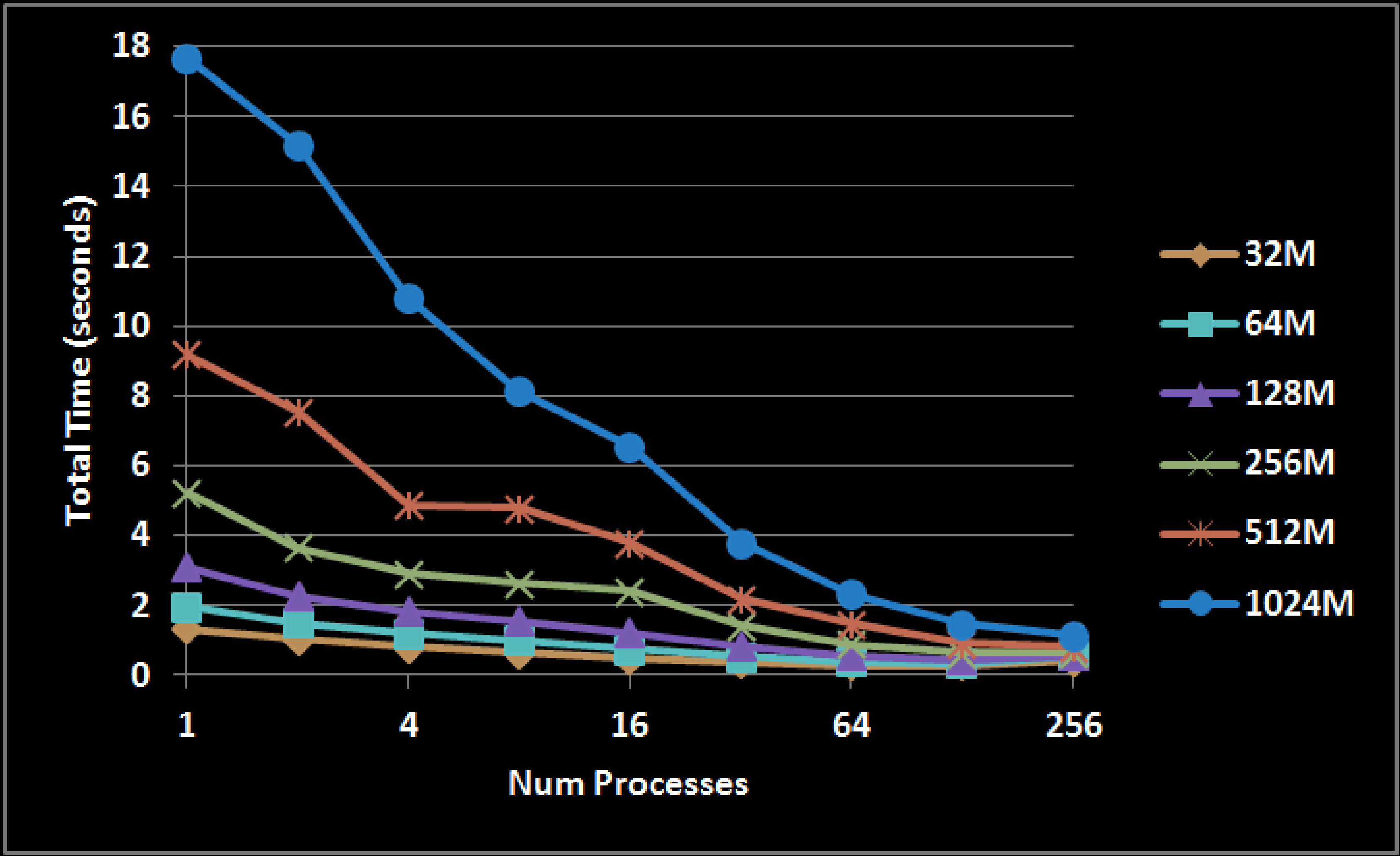
Num Processes

Num Segments

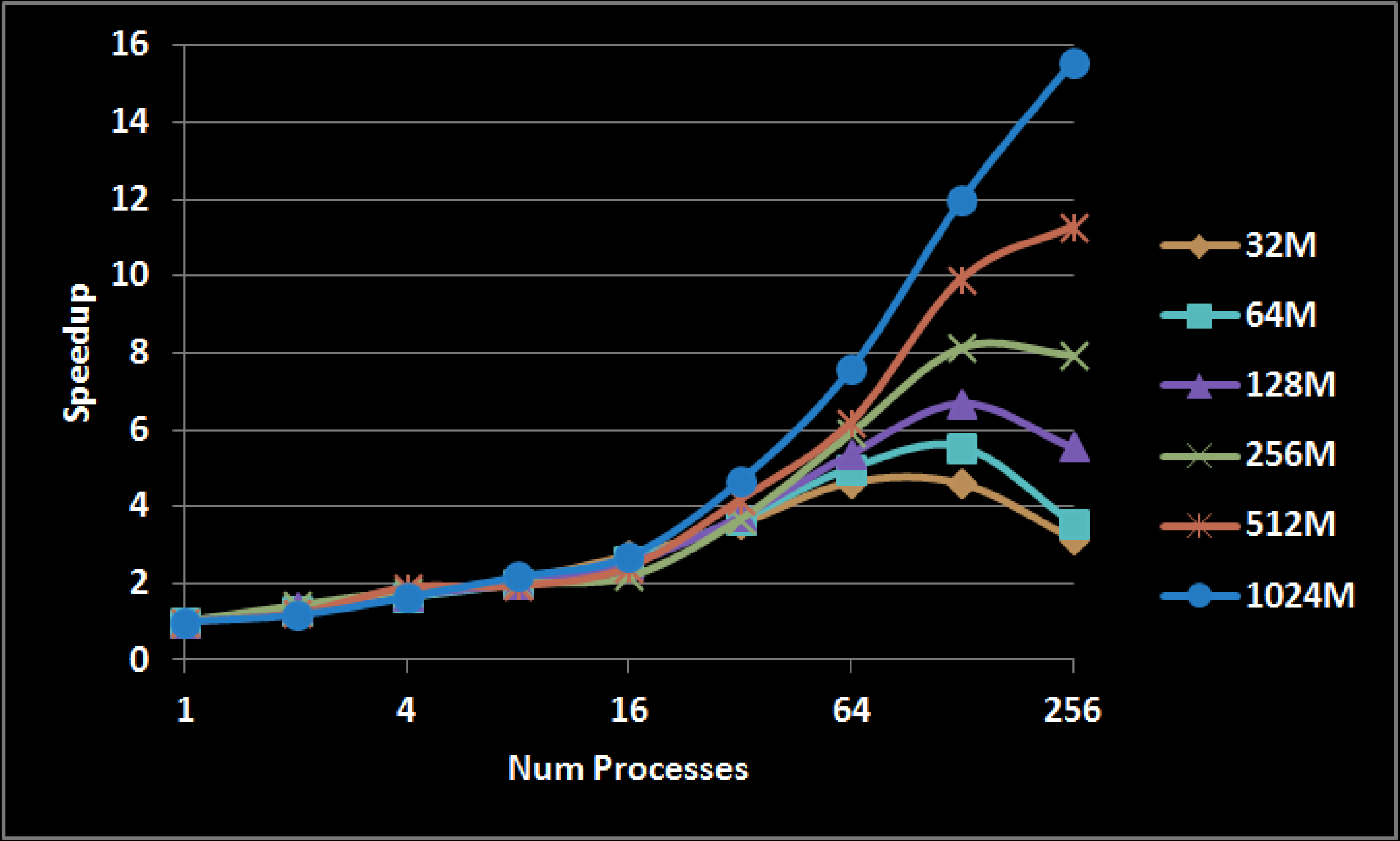


	1	2	4	8	16	32	64	128	256
32M	1.34	1.05	0.82	0.64	0.49	0.38	0.29	0.29	0.42
64M	1.98	1.52	1.21	1.00	0.77	0.54	0.40	0.36	0.56
128M	3.12	2.29	1.84	1.57	1.24	0.84	0.58	0.47	0.56
256M	5.24	3.67	2.93	2.63	2.41	1.43	0.89	0.64	0.66
512M	9.21	7.56	4.89	4.80	3.81	2.21	1.49	0.93	0.82
1024M	17.70	15.21	10.80	8.16	6.57	3.80	2.34	1.48	1.14

Parallel Results – T_{Total}



Parallel Results – T_{Total}



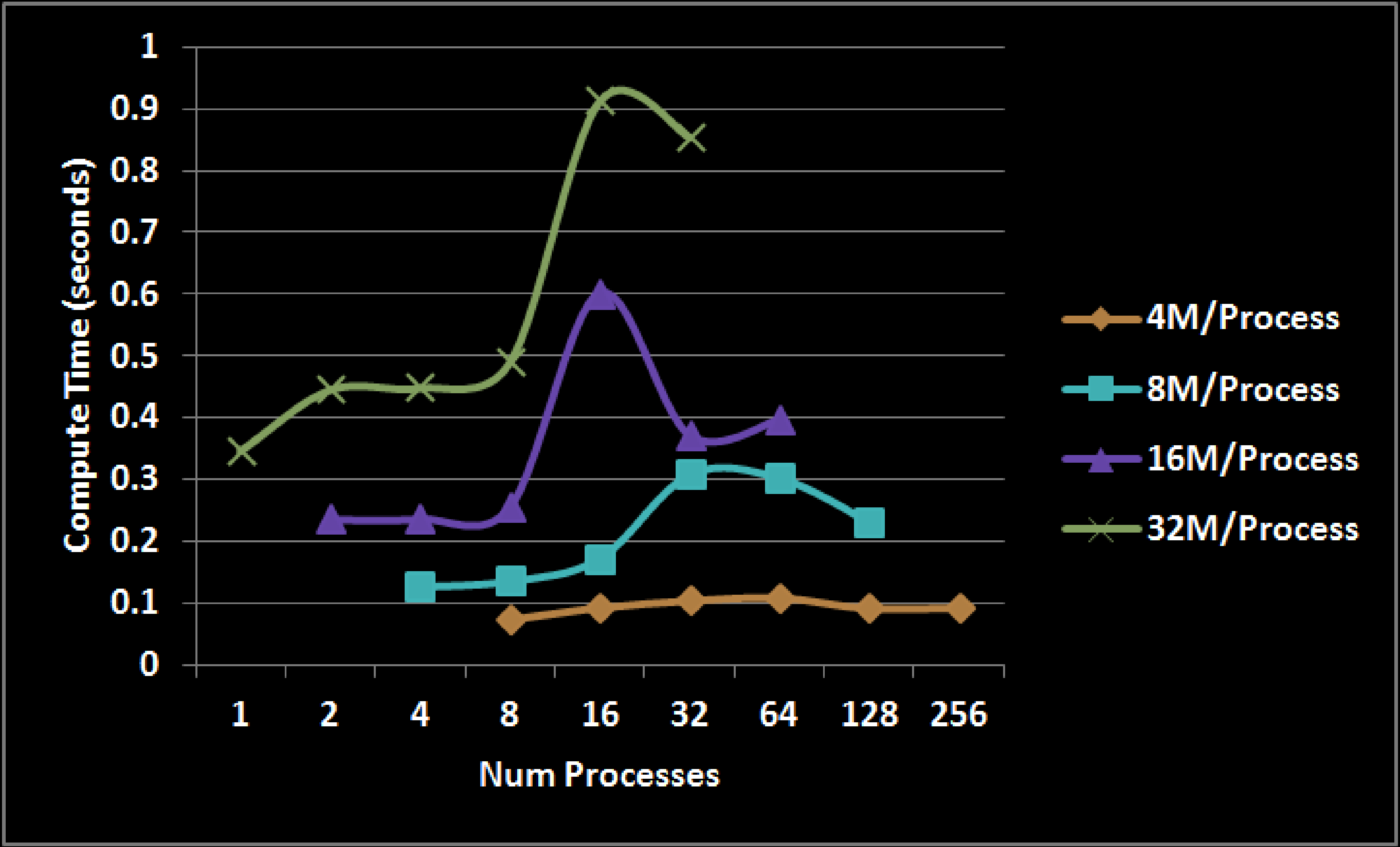
Parallel Results – $T_{Compute}$

- Compute Time (seconds)

Num Processes

Num Segments	1	2	4	8	16	32	64	128	256
32M	0.35	0.24	0.13	0.07	0.05	0.03	0.01	0.01	0.00
64M	0.67	0.45	0.24	0.14	0.09	0.05	0.03	0.01	0.01
128M	1.31	0.86	0.45	0.26	0.17	0.10	0.05	0.03	0.01
256M	2.59	1.70	0.87	0.49	0.60	0.31	0.11	0.05	0.03
512M	5.13	3.32	1.69	1.31	0.91	0.37	0.30	0.09	0.06
1024M	10.25	6.58	3.45	2.23	1.76	0.85	0.40	0.23	0.09

Parallel Results – $T_{Compute}$

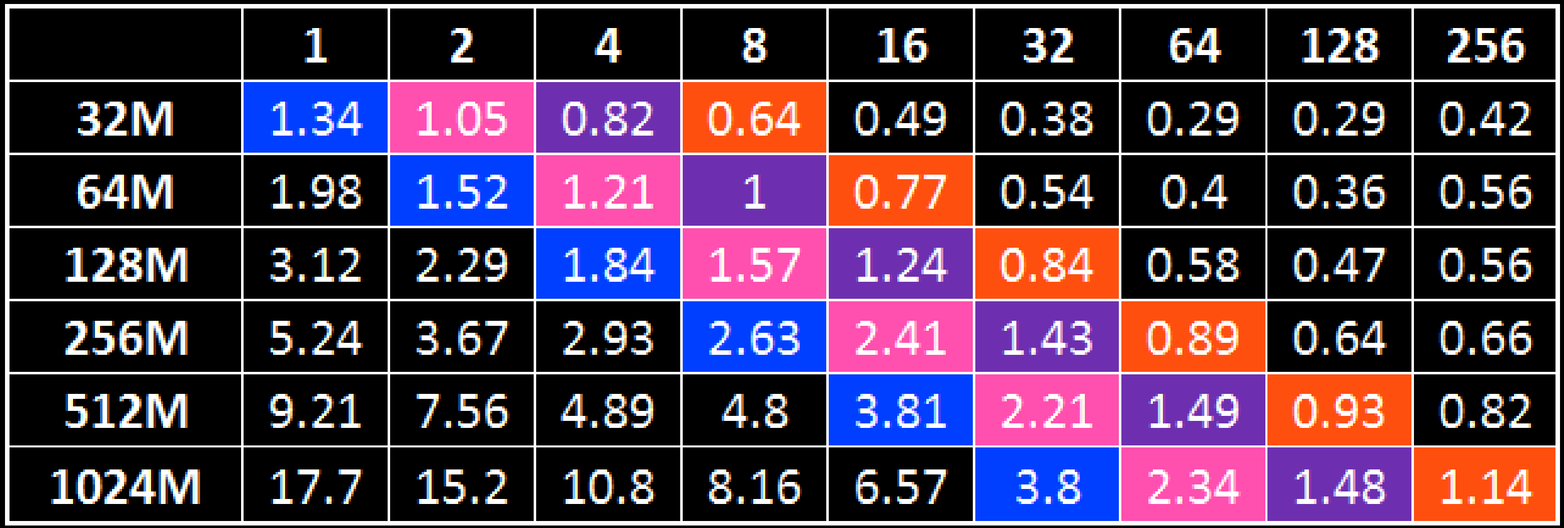


Parallel Results – T_{Total}

- Total Time (seconds)

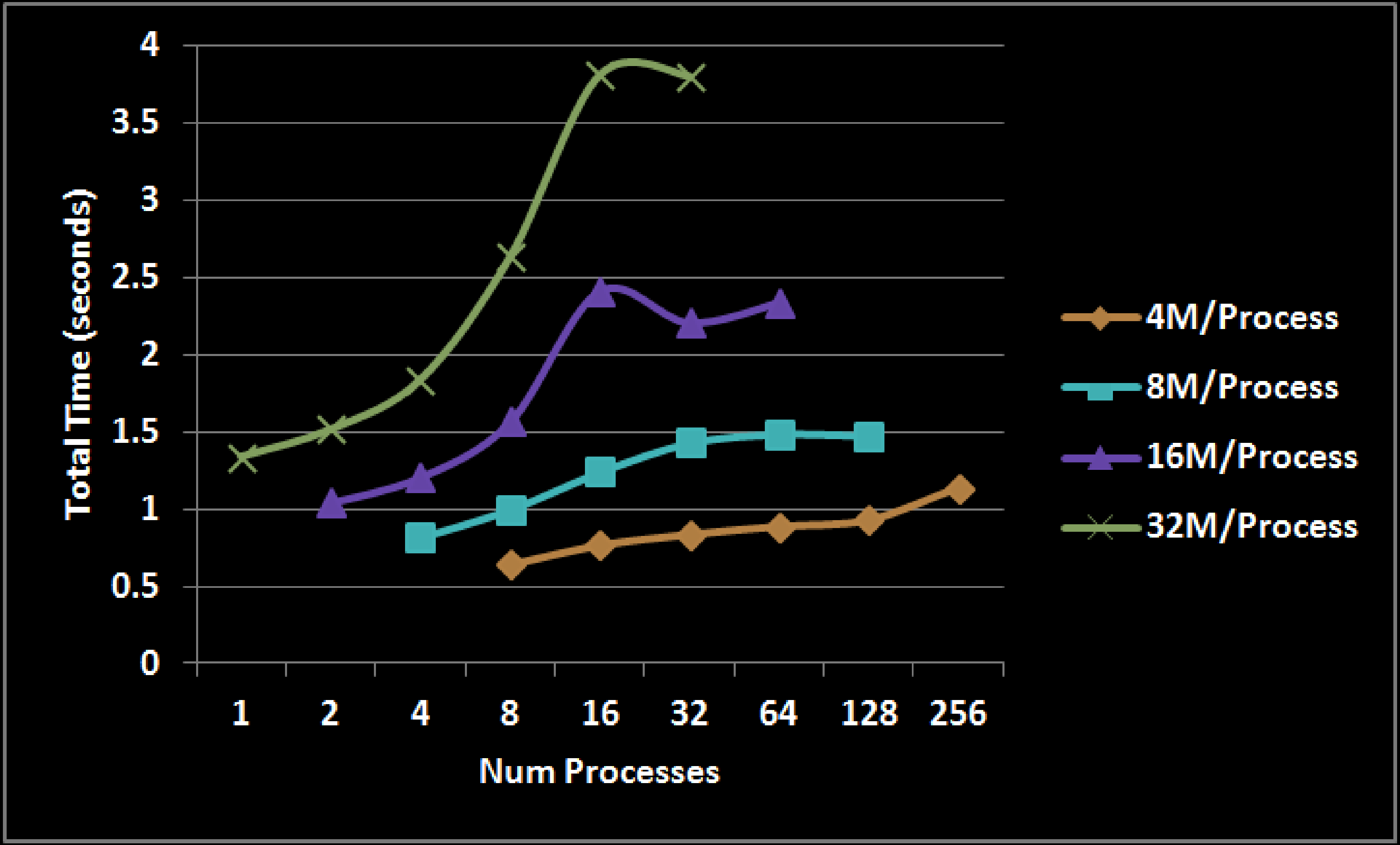
Num Processes

Num Segments



	1	2	4	8	16	32	64	128	256
32M	1.34	1.05	0.82	0.64	0.49	0.38	0.29	0.29	0.42
64M	1.98	1.52	1.21	1	0.77	0.54	0.4	0.36	0.56
128M	3.12	2.29	1.84	1.57	1.24	0.84	0.58	0.47	0.56
256M	5.24	3.67	2.93	2.63	2.41	1.43	0.89	0.64	0.66
512M	9.21	7.56	4.89	4.8	3.81	2.21	1.49	0.93	0.82
1024M	17.7	15.2	10.8	8.16	6.57	3.8	2.34	1.48	1.14

Parallel Results – T_{Total}



Conclusions

- For the given data sets and test configurations:
  - Compute time ($T_{Compute}$) continued to decrease and associated speedup was good
  - The larger the data set, the larger the optimal number of processes
  - The smaller the amount of fixed data per process, the more scalable the parallel algorithm appeared to be, although this requires further investigation

Follow-On Items

- Adapt serial algorithm to use OpenMP
- Remove assumption that input data is ordered by increasing *x*

Questions

?

References

- Miller, Russ and Laurence Boxer. *Algorithms Sequential and Parallel: A Unified Approach*. Hingham, MA: Charles River Media, 2005. Print.

- http://en.wikipedia.org/wiki/File:Atlanta_Skyline_from_Buckhead.jpg

Determining Line Segment Visibility with MPI

CSE 633: Parallel Algorithms
Fall 2012

Jayan Patel