Parallel Sudoku Solver using MPI and C

8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

Name: Rohit H. Shreekant

Ubit Name: rohithol

UB # 5009 7252

CSE 633

Instructor : Prof. Russ Miller

Monday, May 5th, 2014

What is Sudoku!

- Logic puzzle
- Given a grid: player can deduce all the remaining symbols
- Rules : Must have 9 unique symbols (1 to 9)
    1. Each Row
    2. Each Column
    3. Each 3x3 Block
    4. All Numbers from 1 – 9
    5. Fill in all the blank spots

8	3	5	4	1	6	9	2	7
2	9	6	8	5	7	4	3	1
4	1	7	2	9	3	6	5	8
5	6	9	1	3	4	7	8	2
1	2	3	6	7	8	5	4	9
7	4	8	5	2	9	1	6	3
6	5	2	7	8	1	3	9	4
9	8	1	3	4	5	2	7	6
3	7	4	9	6	2	8	1	5

Standard Sudoku Grid : 9 X 9

Solving a Sudoku

8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5

- Two Recursive Steps

1. Constraint Propagation

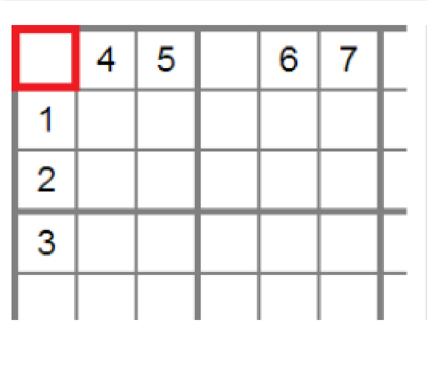
- Reduce the amount of possibilities for each cell to 1 number!

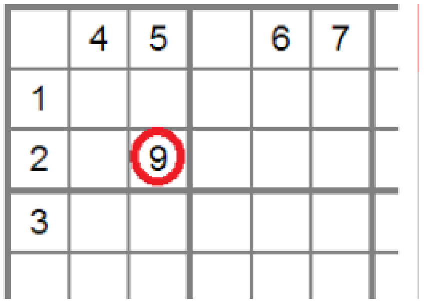
2. Search

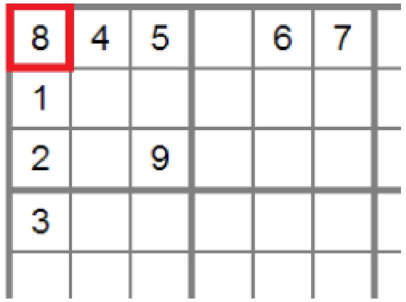
- A cell is chosen to assume one of its possible values, then Constraint Propagation is repeated.

Constraint Propagation

- Rule 1
  a. For any cell, if a number already exists in its row, column or box (the cell's peers), the possibility of that number for that cell is removed.

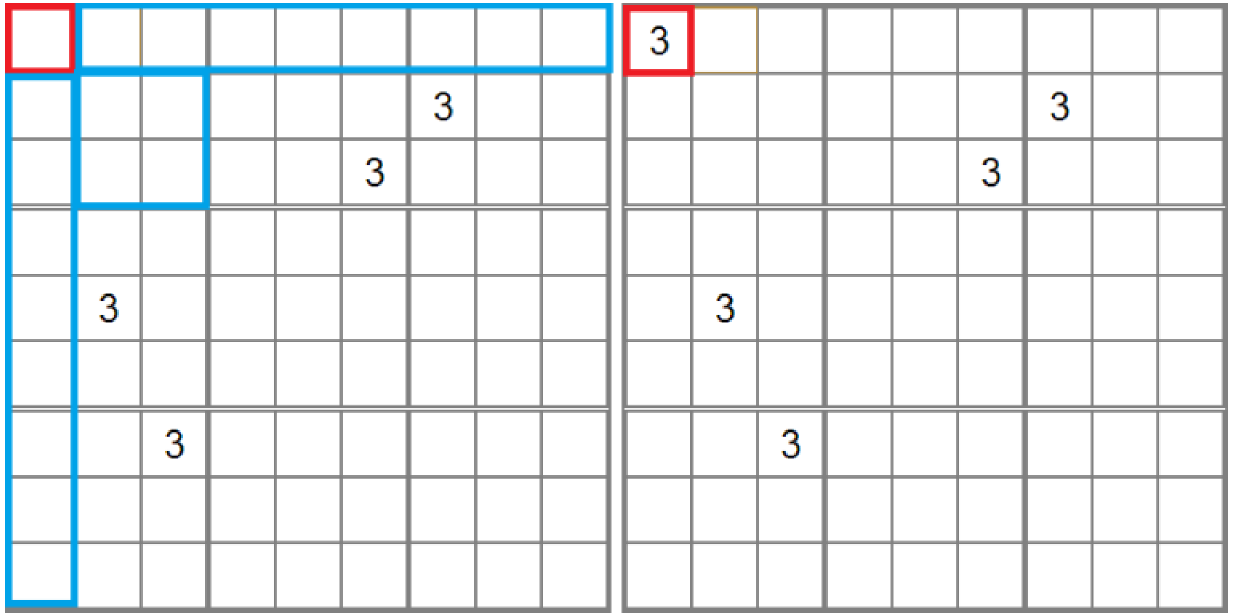






Constraint Propagation

- Rule 2
  a. For any cell, if all of its peers has a specific number removed, the cell itself must contain that number.



Search

- A Single cell is chosen to assume one of its possible values.
- Contraint_prop()

- If (assumption is TRUE) -> eventually arrive at the solution.

- If (assumption is FALSE) or we reach a contradiction -> Initial assumption was wrong.

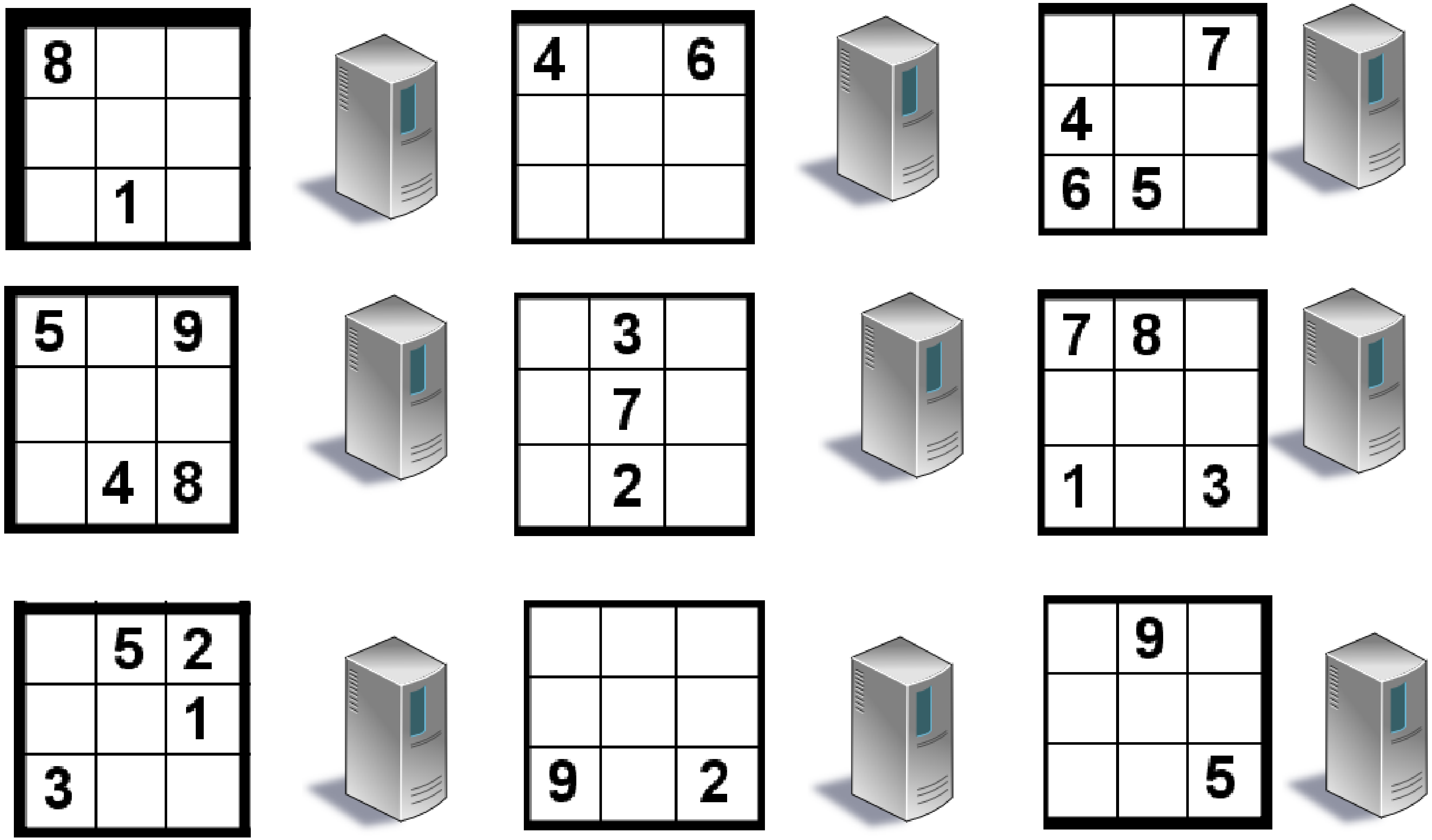
- Remove that assumption from the possibilities list.

Recursive Calls

- CP() -> Search() -> CP() -> Search() …

Parallel Solution

- Parallelizing Constraint Propagation



Approach

- 1 Master + n worker nodes
- Master inputs a number based on constraints.
- Distributes the grid amongst the workers.
- Workers perform constraint_propagation()
- Masters gathers all the data.
- Repeat till all entries have been made.

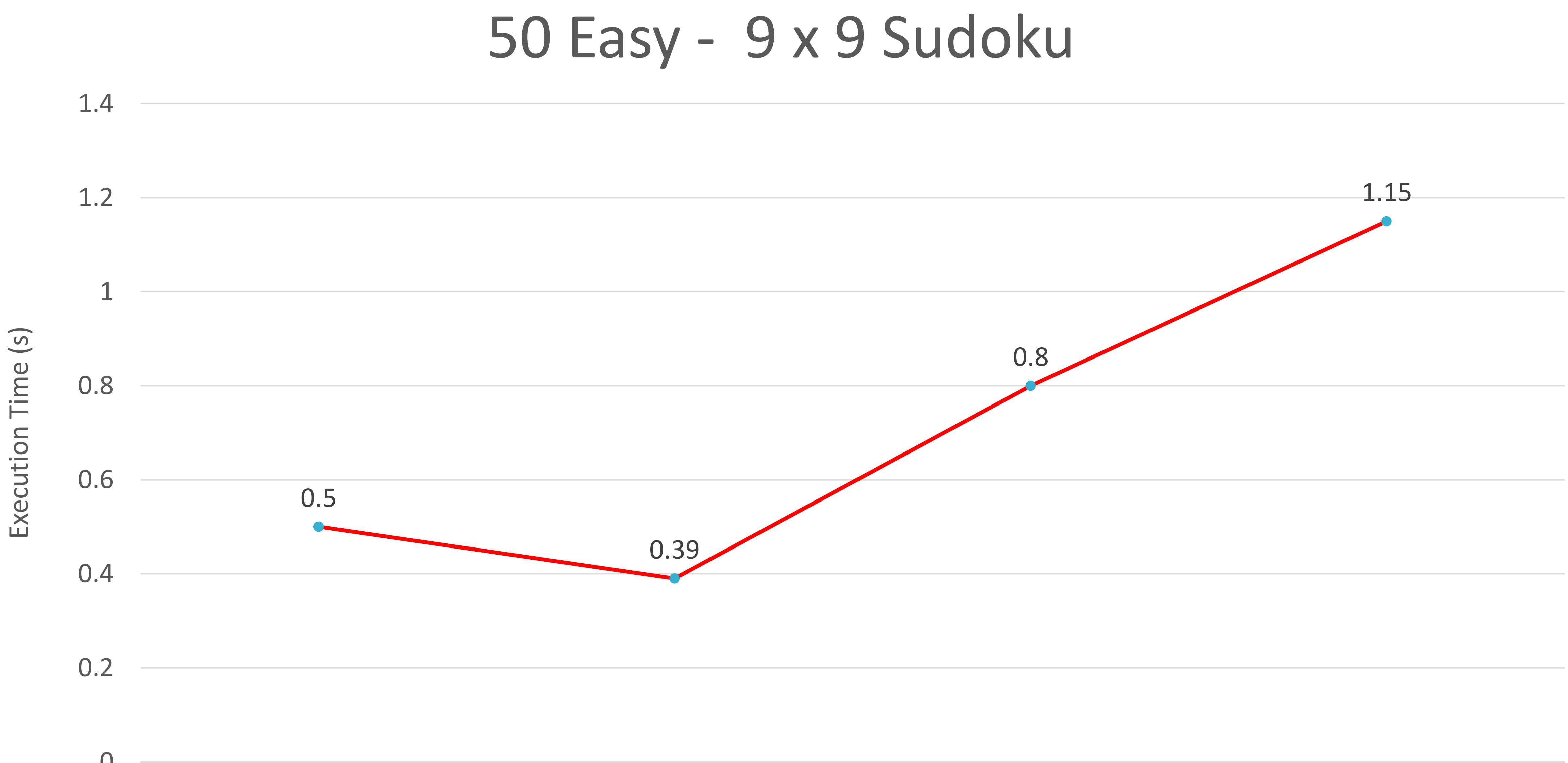
Important :

- Please note – chosen inter process communication over efficiency
- Dell – 2.40 Gz Intel Xeon E5645 (Batch System)
  - 372 Total Nodes
  - 12 Cores each
  - Main Memory : 48 GB
- **<u>1 Core / Node</u>**
- Why?
  - MPI handles send recv automatically.
  - Cores on the same node use quickest communication medium = shared memory.
  - For Uniformity.

Experiment 1

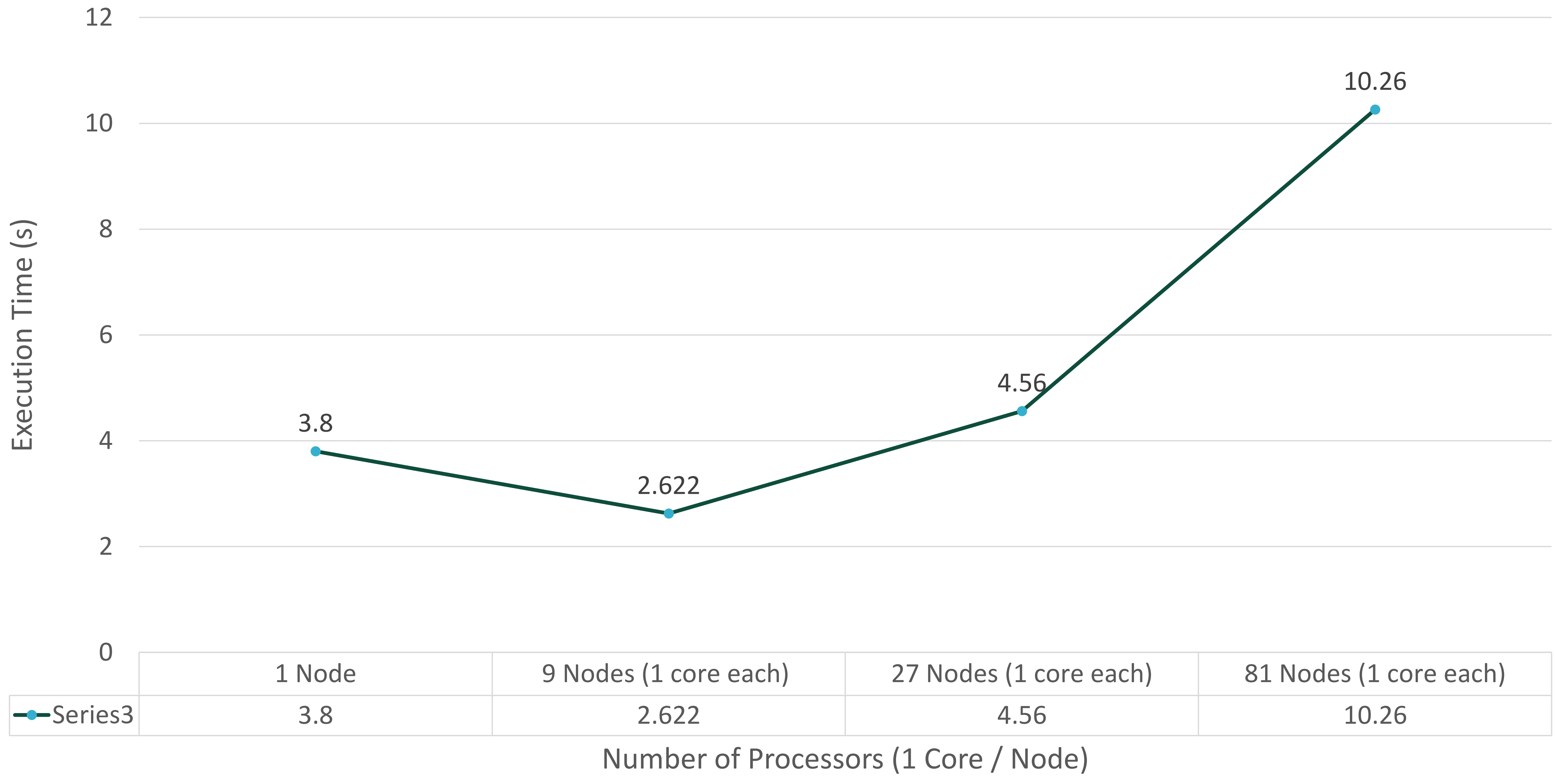
- Keeping Data Constant and Increasing the Number of Nodes (Processors)
- Data = 50 Easy Sudoku + 90 Hard Sudoku
- Easy = Given ~ (25 to 30)
- Difficult = Given ~ (19 to 25)
- 4 Rounds
  1. Serial
  2. 3x3 cell - > Each Node
  3. Arr[3] -> Each Node
  4. Arr[1] -> Each Node

8			4		6			7
						4		
	1					6	5	
5		9		3		7	8	
				7				
	4	8		2		1		3
	5	2					9	
		1						
3			9		2			5



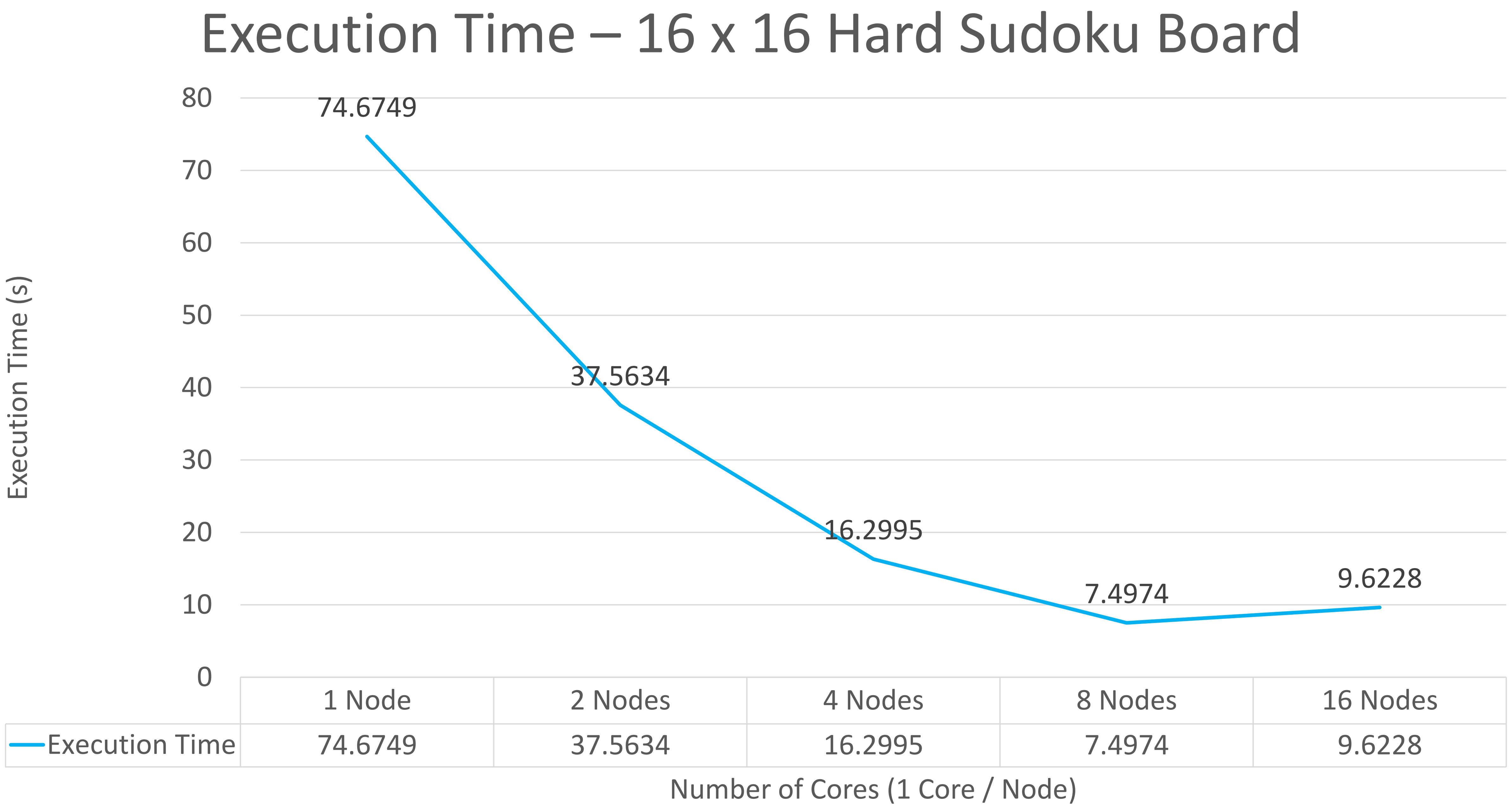
	1 Node	9 Nodes (1 core each)	27 Nodes (1 core each)	81 Nodes (1 core each)
Series3	0.5	0.39	0.8	1.15

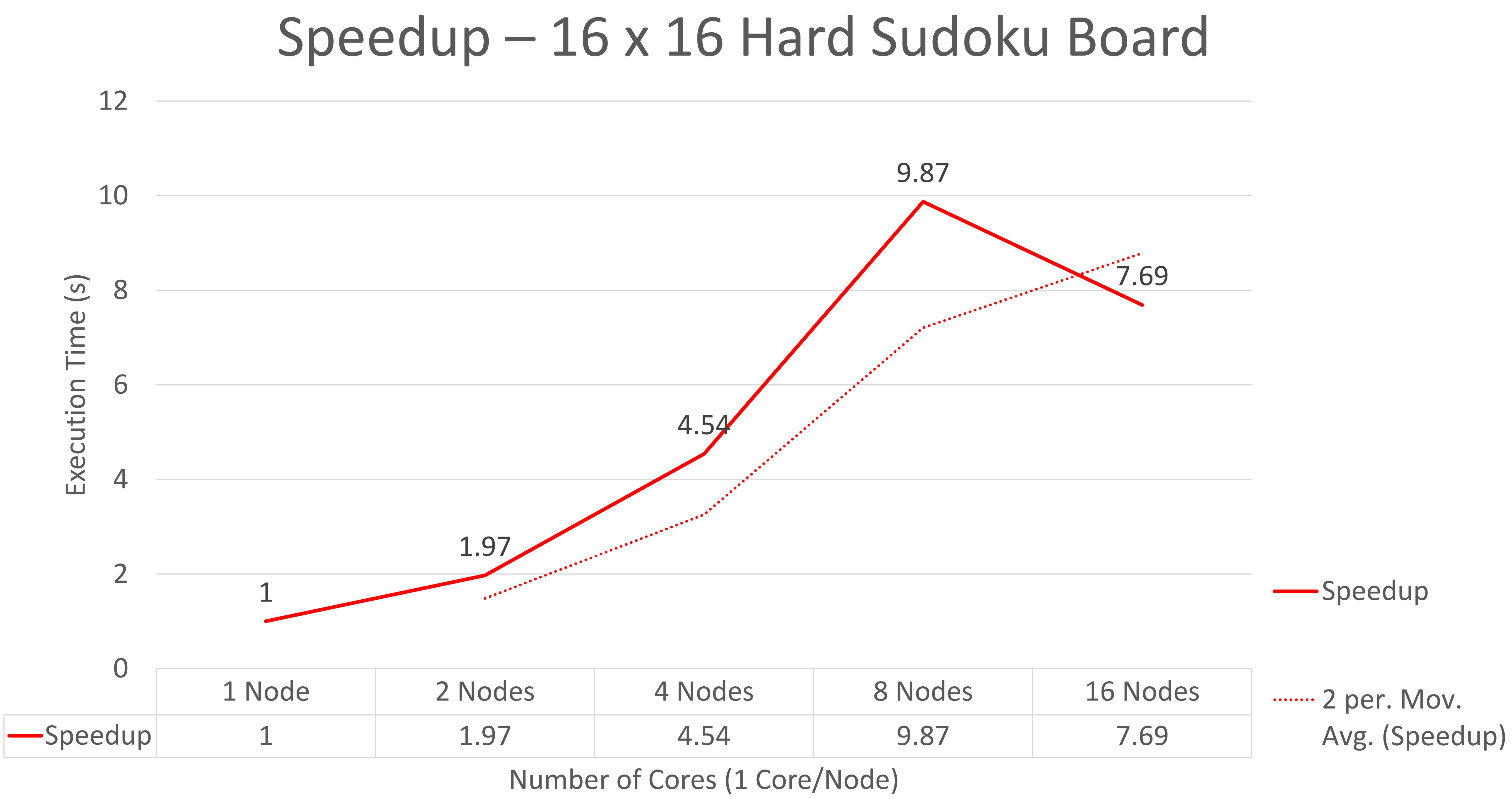
90 Hard – 9 x 9 Sudoku



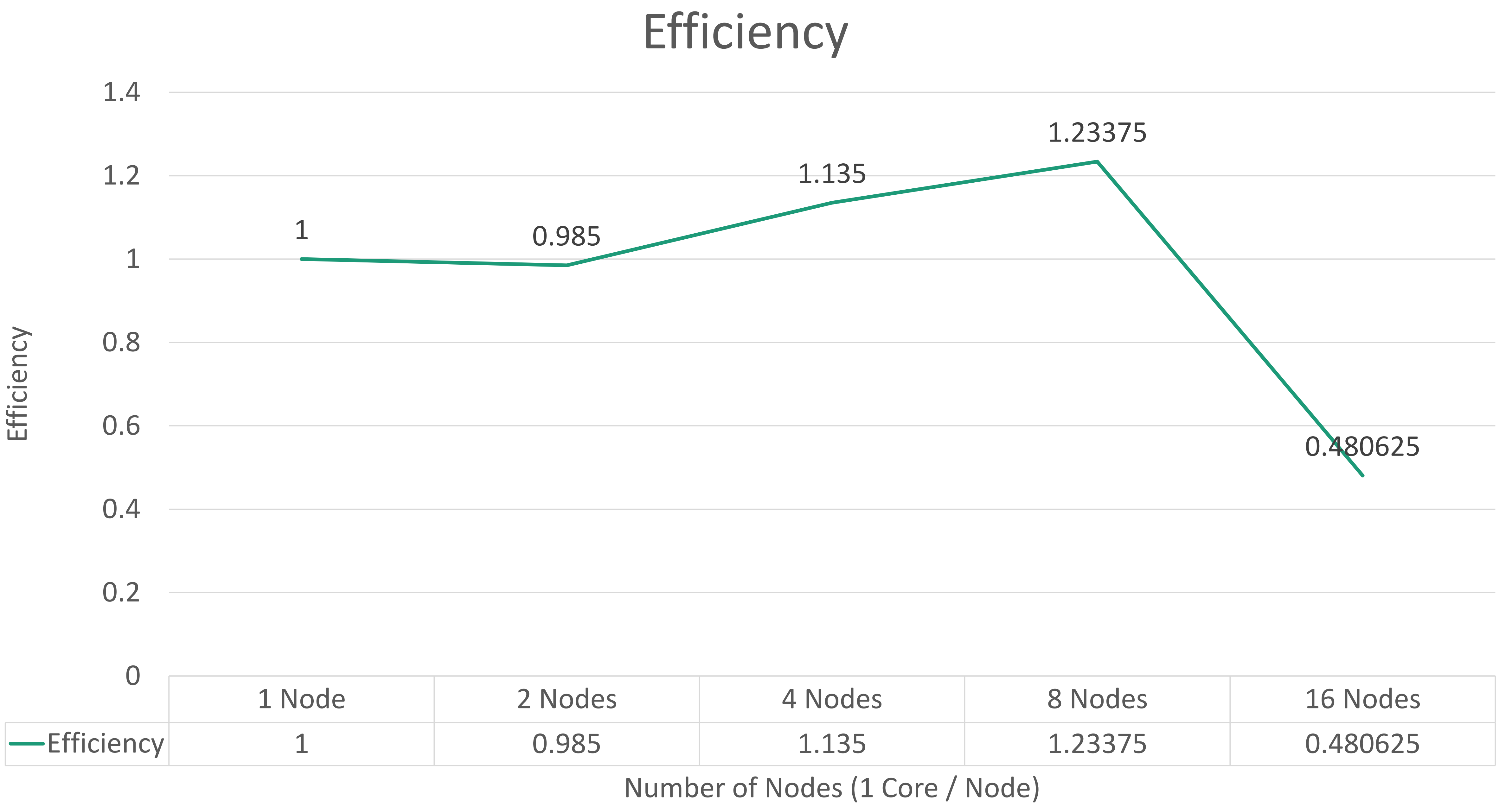
Experiment 2 – Speed Up

- Initially Idea – Run many 25 x 25 Sudoku boards
- Problems with 25 x 25 – Take too long!
- A 9 x 9 is solved really fast
- Best size for analysis – 16 x 16 hard
- Hard -> 104 – 115 cells are filled (16 * 16 = 256)


Execution Time – 16 x 16 Hard Sudoku Board
Execution Time (s)
80
70
60
50
40
30
20
10
0
74.6749
37.5634
16.2995
7.4974
9.6228
1 Node
2 Nodes
4 Nodes
8 Nodes
16 Nodes
Execution Time
74.6749
37.5634
16.2995
7.4974
9.6228
Number of Cores (1 Core / Node)



Speedup – 16 x 16 Hard Sudoku Board
Execution Time (s)
0
2
4
6
8
10
12
1
1.97
4.54
9.87
7.69
1 Node
2 Nodes
4 Nodes
8 Nodes
16 Nodes
Speedup
1
1.97
4.54
9.87
7.69
Number of Cores (1 Core/Node)
Speedup
2 per. Mov. Avg. (Speedup)


Efficiency
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Efficiency
1
0.985
1.135
1.23375
0.480625
1 Node
2 Nodes
4 Nodes
8 Nodes
16 Nodes
Efficiency
1
0.985
1.135
1.23375
0.480625
Number of Nodes (1 Core / Node)

Results & Observations :

- Super Linear Speedup

$$S_p = \frac{T_1}{T_p}$$

  - Usually Linear Speed up
  - Generally Noticeable in Open MPI – Cache Effect
  - Occurred Due to my implementation – Broadcasting cell values after constraint propagation.

- Efficiency > 1 ?
  - Due to Super Linear Speedup

$$E_p = \frac{S_p}{p}$$

- Balance of Processors used and Data Distribution -> Best Efficiency

- Easy problems are solved too quickly (serially) -> Inaccurate Speedup
  - Difficult to analyze.

Results & Observations:

- Modified Brute Force approach
  - Good Speedup
  - Poor Execution Time
  - Hard Problems : ~7.5 s
  - Expert Problems : exceeded 15 min quota
  - Other implementation took over 6 hours.

- Parallel Programming is really hard! – Very Interesting at the same Time!

References

- Parallelization of Sudoku – (University of Toronto) http://individual.utoronto.ca/rafatrashid/Projects/2012/SudokuReport.pdf
- Parallel Sudoku Solver – Carnegie Mellon University, Hilda Huang, Lindsay Zhong
- Arbitrary Size Parallel Sudoku Creation – William Dudziak http://www.dudziak.com/ArbitrarySizeSudokuCreation.pdf
- Solving Every Sudoku Puzzle – Peter Norvig

  http://norvig.com/sudoku.html