Report

November 8, 2019

0.1 Section 0: (warm-up)

- Compute Theoretical Peak performance for your laptop/desktop
  - The CPU of my computer is **Intel(R) Core(TM) i7-4720HQ**
  - Clock-rate (frequency) is 3.6 GHz (Turbo frequency)
  - It has 4 cores
  - Number_of_FP_operation is **16** (for 64 bit architecture) (source)

[1]:
```
Clock_rate = 3.6 #turbo frequency
cores = 4
FP_Operation = 16

Theoretical_Peak_Performance = Clock_rate * FP_Operation * cores

print("Theoretical Peak Performance of my computer is:
 ↪",Theoretical_Peak_Performance, "GLOPS")
```

```
Theoretical Peak Performance of my computer is: 230.4 GLOPS
```

	Your model	CPU	Frequency	number of core	Peak performance
Laptop	MSI GE70 2QE APACHE PRO	Intel(R) Core(TM) i7-4720HQ	3.6 GHz	4	230.4 GLOPS

- Compute sustained Peak performance for your cell-phone
  - My phone is iPhone 6, its CPU is **Apple A8 a 64 bit**
  - Clock-rate (frequency) is 1.4 GHz
  - It has 2 cores
  - Number_of_FP_operation is **12.8** (for 64 bit architecture) source
- Compute sustained Peak performance for your cell-phone
  - I have used MobileLinpack app from App Store.
  - Below you can find the results.

0.1.1 First Run

Default Parameters

- Matrix start size = 32
- Matrix finish size = 257
- Step = 16
- Number of iterations = 200
- **Result** 2.55 GLOPS

0.1.2 Second Run

- Matrix start size = 64
- Matrix finish size = 257
- Step = 16
- Number of iterations = 150
- **Result** 2.55 GLOPS

0.1.3 Third Run

- Matrix start size = 128
- Matrix finish size = 512
- Step = 16
- Number of iterations = 150
- **Result** 2.09 GLOPS

0.1.4 Fourth Run

- Matrix start size = 32
- Matrix finish size = 512
- Step = 8
- Number of iterations = 150
- **Result** 2.58 GLOPS

0.1.5 Fifth Run

- Matrix start size = 16
- Matrix finish size = 700
- Step = 8
- Number of iterations = 200
- **Result** 2.58 GLOPS
- Extra Bonus: Identify the CPU and calculate the Peak performance of the CPU

[3]:
```
Clock_rate = 1.4
cores = 2
FP_Operation = 12.8

Theoretical_Peak_Performance = Clock_rate * FP_Operation * cores

print("Theoretical Peak Performance of my cell phone is:
 ↪",Theoretical_Peak_Performance, "GLOPS")
```

```
Theoretical Peak Performance of my cell phone is: 35.839999999999996 GLOPS
```

	Your model	sustained performance	size of matrix	Peak performance	memory
SmartPhone	iPhone 6, Apple A8 (CPU)	2.58 GLOPS	700	35.84 GLOPS	1 GB

- Find out in which year your cell phone/laptop could have been in top 1% of Top500

According to results my phone is 505.7 times are slower than Intel ASCI Red (1998 TOP500 #1) and 3307.4 times are faster than ES-1045 (1979)

In Top500, sustained peak performance is used for benchmark, that is why to answer the below question, I also find the sustained peak performance of my laptop by using **Intel® Math Kernel Library (Intel® MKL) Benchmarks Suite.**

Parameters for test is Number of equations to solve (problem size): 30000, Leading dimension of array: 30000, Number of trials to run: 8

Average sustained performance is **131.2885 GLOPS**, sustained peak performance is **136.3127 GLOPS**

Even for 1993 my smart phone can not enter the %1 of the TOP500.

My laptop can enter the %1 of TOP500 in 1996 (November) as #5.

	Your model	performance	Top500 year	number 1 HPC system	number of processors (TOP500)
SmartPhone	iPhone 6, Apple A8 (CPU)	2.58 GLOPS	Can not enter(1993)	Numerical Wind Tunnel	140

	Your model	performance	Top500 year	number 1 HPC system	number of processors (TOP500)
Laptop	Intel(R) Core(TM) i7-4720HQ	136.3 GLOPS	1996 (November)	CP-PACS/2048	2048

0.2 Section 1: theoretical model

- devise a performance model for a simple parallel algorithm: sum of N numbers
  - Serial Algorithm : n-1 operations

    $T_{serial} = N * T_{comp}$ where T_{comp} *is time to compute a floating point operation*
- Parallel Algorithm : master-slave
  - read N and distribute N to P-1 slaves ===> $T_{read} + (P-1) * T_{comm}$ where T_{comm} is *time each processor takes to communicate one message, i.e. latency..* T_{read} = *time master takes to read*
  - N/P sum over each processors (including master) ===> T_{comp}/N
  - Slaves send partial sum ===> $(P-1) * T_{comm}$
  - Master performs one final sum ===> $(P-1) * T_{comp}$

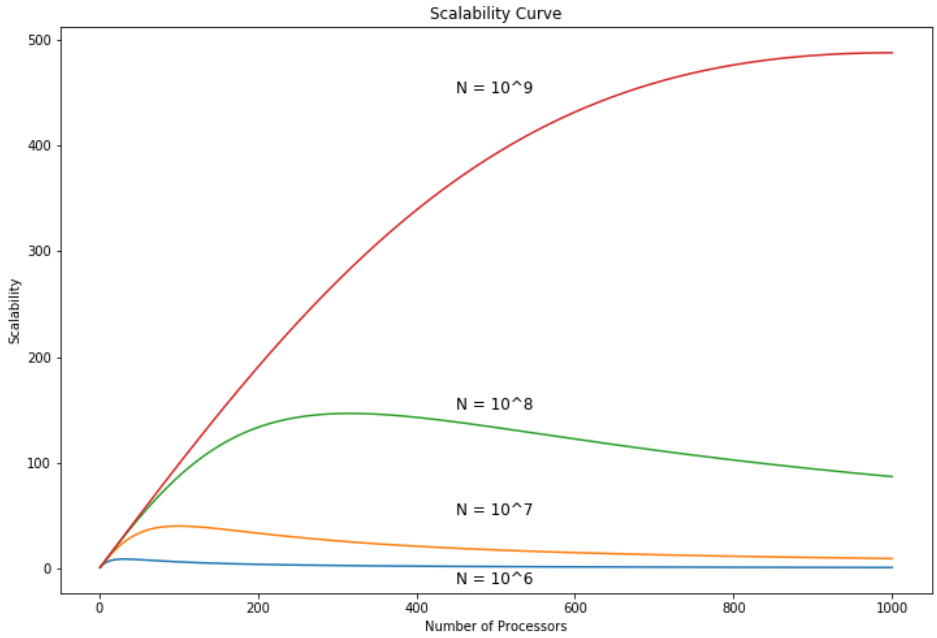
    the final model ===> $T_p = T_{comp} * (P - 1 + n/P) + T_{read} + 2(P-1) * T_{comm}$
- N: numbers to be sum
- T_comp: time to compute a floating point operation
- T_read: time master takes to read
- P: number of processors

[4]:
```
import numpy as np

#Assumptions

T_comp = 2e-09
T_read = 1e-04
T_comm = 1e-06
P = np.arange(1,1000+1)
```

- compute scalability curves for such algorithm and make some plots
- Play with some value of N and plot against P (with P ranging from 1 to 1000)

Refer to ../codes/section1_scalability.py

- Serial algorithm time for N = 1000000 is 0.002
  - For N = 1000000 best speed up value when P is 32
- Serial algorithm time for N = 10000000 is 0.02
  - For N = 10000000 best speed up value when P is 100
- Serial algorithm time for N = 100000000 is 0.2
  - For N = 100000000 best speed up value when P is 316
- Serial algorithm time for N = 1000000000 is 2.0
  - For N = 1000000000 best speed up value when P is 1000



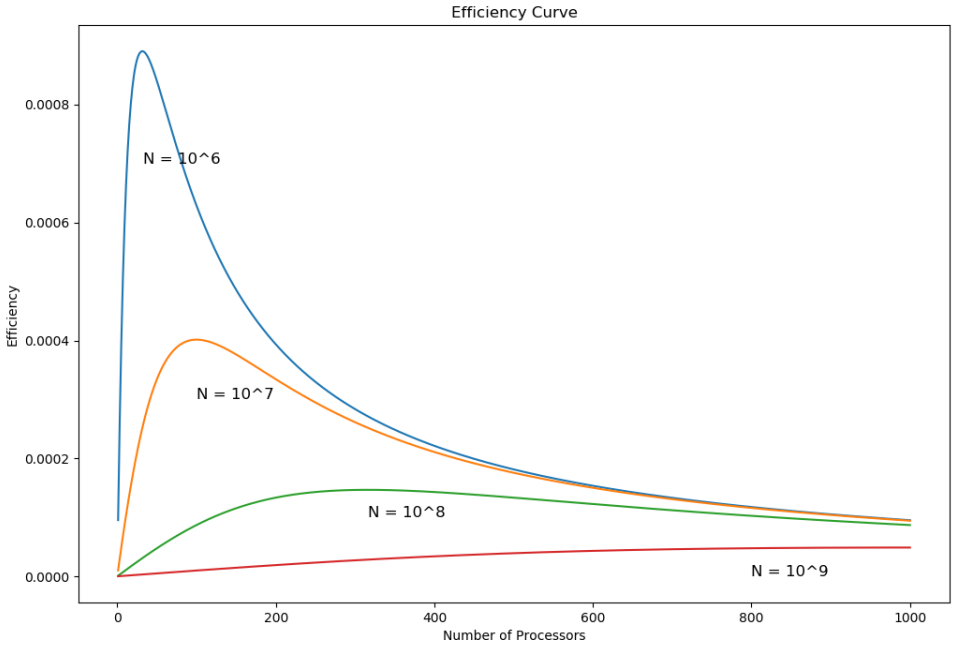
- For which values of N do you see the algorithm scaling ?
- For which values of P does the algorithm produce the best results ?

For P = [1,1000], when N gets larger, algorithm tends to scale better.

Best scaling P values are highly dependent on N, when N is smaller best scaling value of P is small as well. For instances: For N = 10^9 best scaling value of P is 1000, for N = 10^8 best scaling value of P is 316, for N 10^7 best scaling value of P is 100 and for N = 10^6 best scaling value of P 32. It means that if N is large enough more processors give better results up to specific point. For smaller N, best P tends to be smaller as well.

Refer to ../codes/section1_efficiency.py

- Serial algorithm time for N = 1000000 is 0.002
  - For N = 1000000 best efficiency value is 0.000890614702267505 with P 32
- Serial algorithm time for N = 10000000 is 0.02
  - For N = 10000000 best efficiency value is 0.00040144520272982734 with P 100
- Serial algorithm time for N = 100000000 is 0.2
  - For N = 100000000 best efficiency value is 0.0001466766669209062 with P 316
- Serial algorithm time for N = 1000000000 is 2.0
  - For N = 1000000000 best efficiency value is 4.878048780487805e-05 with P 1000

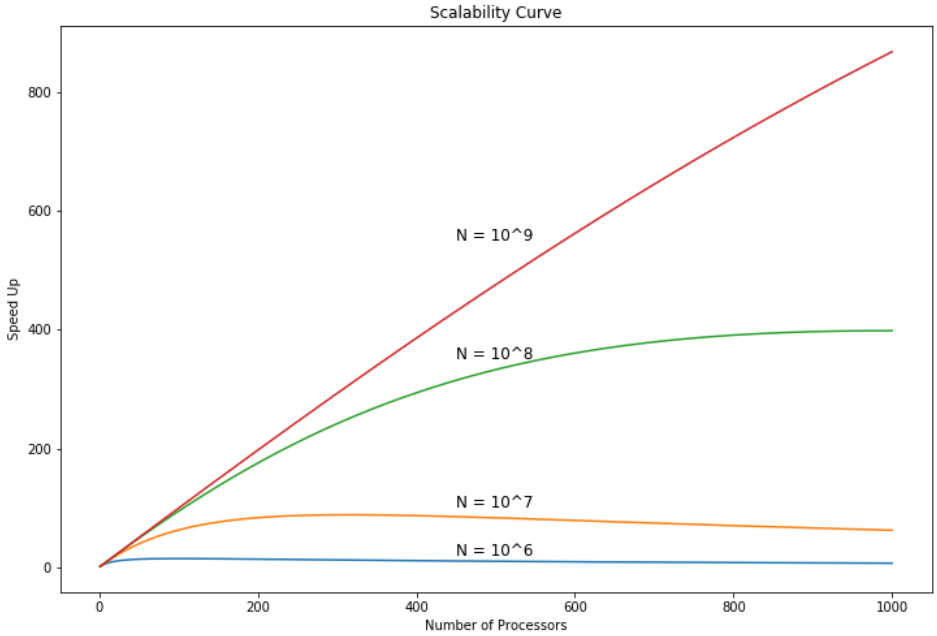


As it can be seen from the plot, for the changing values of N, best P values have the most efficient points as same as previous plot. For N = 10^6 algorithm is most efficient at P = 32, after that point efficiency decrease. For N = 10^9 , it has more way to best efficient point. However for the range of P = [1,1000], most efficient point is P = 1000.

- Can you try to modify the algorithm sketched above to increase its scalability ?
  - Reducing communication time should increase the scalability

Refer to ../codes/section1_better_comm.py

- Serial algorithm time for N = 1000000 is 0.002
  - For N = 1000000 best speed up value when P is 100
- Serial algorithm time for N = 10000000 is 0.02
  - For N = 10000000 best speed up value when P is 315
- Serial algorithm time for N = 100000000 is 0.2
  - For N = 100000000 best speed up value when P is 995
- Serial algorithm time for N = 1000000000 is 2.0
  - For N = 1000000000 best speed up value when P is 1000



Indeed, when I reduce the communication time (for example using collective operations) algorithm tends to scale better. For example, N = 10^6 best speed up value when P is 100 and for N = 10^7 best speed up value when P is 315 and so on.

0.3 Section 2: play with MPI program

0.3.1 2.1: compute strong scalability of a mpi_pi.c program

- Compile the serial and parallel version.
  - I compiled pi.c and mpi_pi.c files using `gcc pi.c -o pi.x` and `mpicc mpi_pi.c -o mpi_pi.x` commands in Ulysses.
- Determine the CPU time required to calculate PI with the serial calculation using 10000000 (10 millions) iterations (stone throws). Make sure that this is the actual run time and does not include any system time.

  - After that by using `/usr/bin/time ./pi.x 10000000` and `/usr/bin/time mpirun -np 4 ./mpi_pi.x 10000000` commands I have determined the required time and estimated pi value. You can find the results below.

- Result of pi.x:

of trials = 10000000 , estimate of pi is 3.141396400

walltime : 0.25000000

0.25user 0.00system **0:00.25elapsed** 99%CPU (0avgtext+0avgdata 1936maxresident)k 0inputs+0outputs (0major+143minor)pagefaults 0swaps

- Elapsed time for serial program is **0.25 seconds**
- Result of mpi_pi.x:

of trials = 10000000 , estimate of pi is 3.141961600

walltime on master processor : **0.07639313**

walltime on processor 1 : 0.06366587

walltime on processor 2 : 0.07195497

walltime on processor 3 : 0.06781292

4.83user 0.25system **0:01.70elapsed** 299%CPU (0avgtext+0avgdata 103008maxresident)k 0inputs+8outputs (1major+26408minor)pagefaults 0swaps

- Elapsed time for parallel pi program is **1.70 seconds**
- The parallel code writes walltime for all the processor involved. Which of these times do you want to consider to estimate the parallel time ?
  - We should consider the **maximum walltime of processors** to estimate the parallel time. In this case it is **0.07639313**
- First let us do some running that constitutes a strong scaling test.
- Make a plot of run time versus number of nodes from the data you have collected.

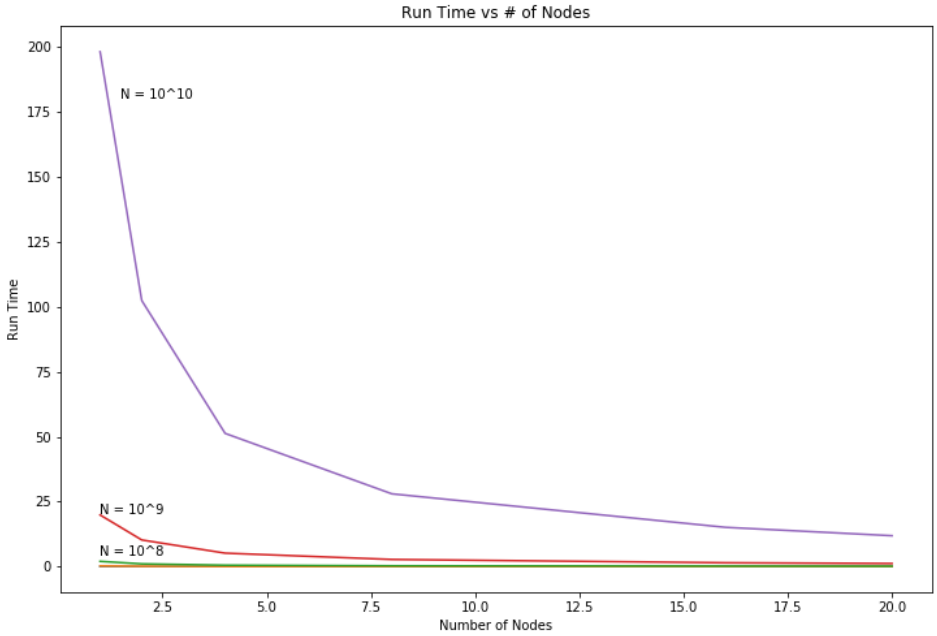
`bash for procs in 1 2 4 8 16 20 ; do /usr/bin/time mpirun -np ${procs} mpi_pi.x 10000000 done > <output.txt files>` by using above command, I get the output.txt files, after that by using python I get the maximum walltime for each run with different number of processors and with these run times I plotted the strong scalability of algorithm

0.3.2 Strong Scaling with Max Walltime

- Strong scalability here would yield a straight line graph. Comment on your results. Repeat the work playing with a large dataset.
- Provide a final plot with at least 3 different size and for each of the size report and comment your final results.

Refer to ../codes/section2_strong_runTime_maxTime.py

- For N = 10^6 you can find run times for different processors below
  - 1 cores run has maximum run time: 0.01990104 scalability is 1.0
  - 2 cores run has maximum run time: 0.01076198 scalability is 1.849198753389246
  - 4 cores run has maximum run time: 0.00544691 scalability is 3.653638484939167
  - 8 cores run has maximum run time: 0.00333095 scalability is 5.974583827436617
  - 16 cores run has maximum run time: 0.00339699 scalability is 5.8584334955357535
  - 20 cores run has maximum run time: 0.00313091 scalability is 6.3563117432312
- For N = 10^7 you can find run times for different processors below
  - 1 cores run has maximum run time: 0.19876003 scalability is 1.0
  - 2 cores run has maximum run time: 0.10333514 scalability is 1.9234505319294095
  - 4 cores run has maximum run time: 0.051332 scalability is 3.8720492090703655
  - 8 cores run has maximum run time: 0.0273509 scalability is 7.267038013374331
  - 16 cores run has maximum run time: 0.01808906 scalability is 10.987858407236196
  - 20 cores run has maximum run time: 0.01323915 scalability is 15.013050686788805
- For N = 10^8 you can find run times for different processors below
  - 1 cores run has maximum run time: 1.98622179 scalability is 1.0
  - 2 cores run has maximum run time: 1.0219152 scalability is 1.9436268195247512
  - 4 cores run has maximum run time: 0.51305103 scalability is 3.8713922667692535
  - 8 cores run has maximum run time: 0.27128315 scalability is 7.321581860133961
  - 16 cores run has maximum run time: 0.14891601 scalability is 13.337866022598915
  - 20 cores run has maximum run time: 0.11698103 scalability is 16.97900753652109
- For N = 10^9 you can find run times for different processors below
  - 1 cores run has maximum run time: 19.85913587 scalability is 1.0
  - 2 cores run has maximum run time: 10.22568107 scalability is 1.9420844180504055
  - 4 cores run has maximum run time: 5.14612985 scalability is 3.8590429019197794
  - 8 cores run has maximum run time: 2.72066188 scalability is 7.299376675943281
  - 16 cores run has maximum run time: 1.44949389 scalability is 13.700737896866883
  - 20 cores run has maximum run time: 1.16037512 scalability is 17.114410269327387
- For N = 10^10 you can find run times for different processors below
  - 1 cores run has maximum run time: 198.1946258 scalability is 1.0
  - 2 cores run has maximum run time: 102.394792 scalability is 1.9355928356199994
  - 4 cores run has maximum run time: 51.2605071 scalability is 3.8664195306019518
  - 8 cores run has maximum run time: 27.986582 scalability is 7.08177317973306
  - 16 cores run has maximum run time: 15.0945041 scalability is 13.130250883829964
  - 20 cores run has maximum run time: 11.862725 scalability is 16.70734386913631

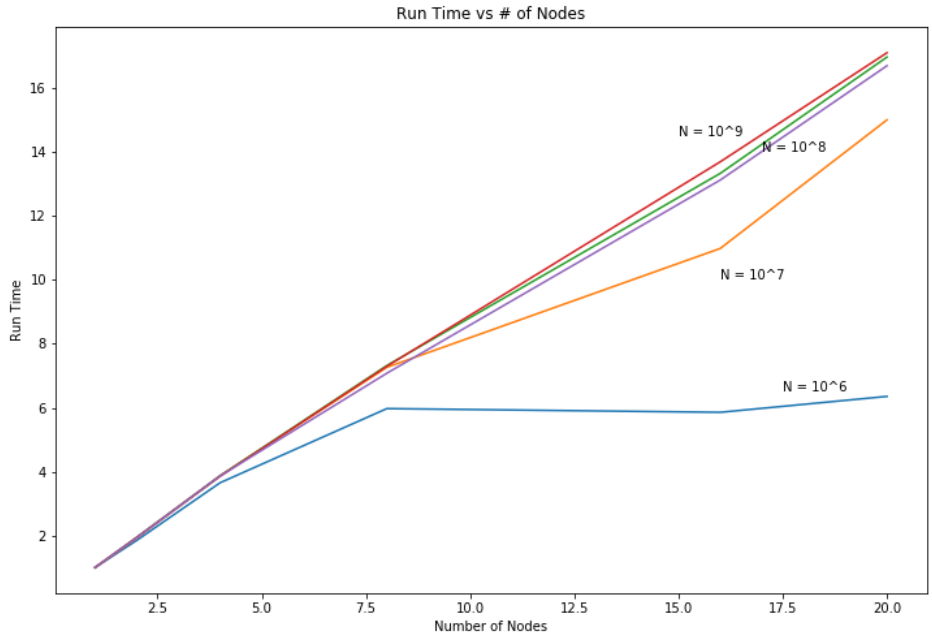


Firstly run time decrease visibly, after specific point for number of nodes it begin to stop.It is compatible with the theoretical model.

Refer to ../codes/section2_strong_scalability_maxTime.py

- For N = 10^6 you can find run times for different processors below
  - 1 cores run has maximum run time: 0.01990104 scalability is 1.0
  - 2 cores run has maximum run time: 0.01076198 scalability is 1.849198753389246
  - 4 cores run has maximum run time: 0.00544691 scalability is 3.653638484939167
  - 8 cores run has maximum run time: 0.00333095 scalability is 5.974583827436617
  - 16 cores run has maximum run time: 0.00339699 scalability is 5.8584334955357535
  - 20 cores run has maximum run time: 0.00313091 scalability is 6.3563117432312
- For N = 10^7 you can find run times for different processors below
  - 1 cores run has maximum run time: 0.19876003 scalability is 1.0
  - 2 cores run has maximum run time: 0.10333514 scalability is 1.9234505319294095
  - 4 cores run has maximum run time: 0.051332 scalability is 3.8720492090703655
  - 8 cores run has maximum run time: 0.0273509 scalability is 7.267038013374331
  - 16 cores run has maximum run time: 0.01808906 scalability is 10.987858407236196
  - 20 cores run has maximum run time: 0.01323915 scalability is 15.013050686788805

- For N = 10^8 you can find run times for different processors below
  - 1 cores run has maximum run time: 1.98622179 scalability is 1.0
  - 2 cores run has maximum run time: 1.0219152 scalability is 1.9436268195247512
  - 4 cores run has maximum run time: 0.51305103 scalability is 3.8713922667692535
  - 8 cores run has maximum run time: 0.27128315 scalability is 7.321581860133961
  - 16 cores run has maximum run time: 0.14891601 scalability is 13.337866022598915
  - 20 cores run has maximum run time: 0.11698103 scalability is 16.97900753652109
- For N = 10^9 you can find run times for different processors below
  - 1 cores run has maximum run time: 19.85913587 scalability is 1.0
  - 2 cores run has maximum run time: 10.22568107 scalability is 1.9420844180504055
  - 4 cores run has maximum run time: 5.14612985 scalability is 3.8590429019197794
  - 8 cores run has maximum run time: 2.72066188 scalability is 7.299376675943281
  - 16 cores run has maximum run time: 1.44949389 scalability is 13.700737896866883
  - 20 cores run has maximum run time: 1.16037512 scalability is 17.114410269327387
- For N = 10^10 you can find run times for different processors below
  - 1 cores run has maximum run time: 198.1946258 scalability is 1.0
  - 2 cores run has maximum run time: 102.394792 scalability is 1.9355928356199994
  - 4 cores run has maximum run time: 51.2605071 scalability is 3.8664195306019518
  - 8 cores run has maximum run time: 27.986582 scalability is 7.08177317973306
  - 16 cores run has maximum run time: 15.0945041 scalability is 13.130250883829964
  - 20 cores run has maximum run time: 11.862725 scalability is 16.70734386913631



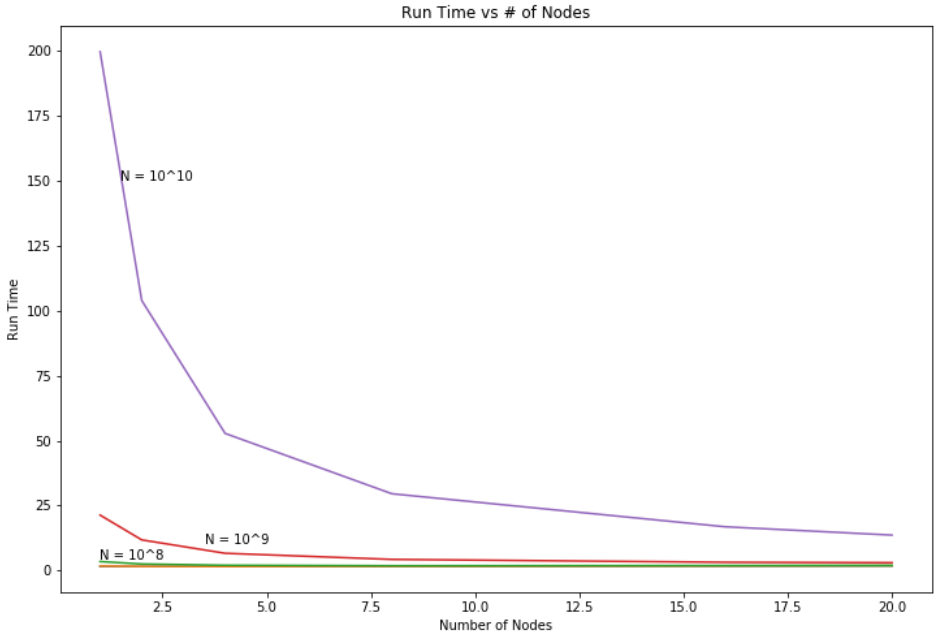
Parallel to theoretical model, if we increase the N, algorithm tend to scale good up to specific points (Ahmdals Law). For smaller N even if we increase the number of nodes algorithm doesn't scale good. For N = 10^6 using more than 8 cores is not logical for other values of N there are still place for improved scalability. But there is one interesting point for N = 10^{10}, so for this value it doesn't

scale better than 10^9. In my opinion it is because of memory.

0.3.3 Strong Scaling with Elapsed Time

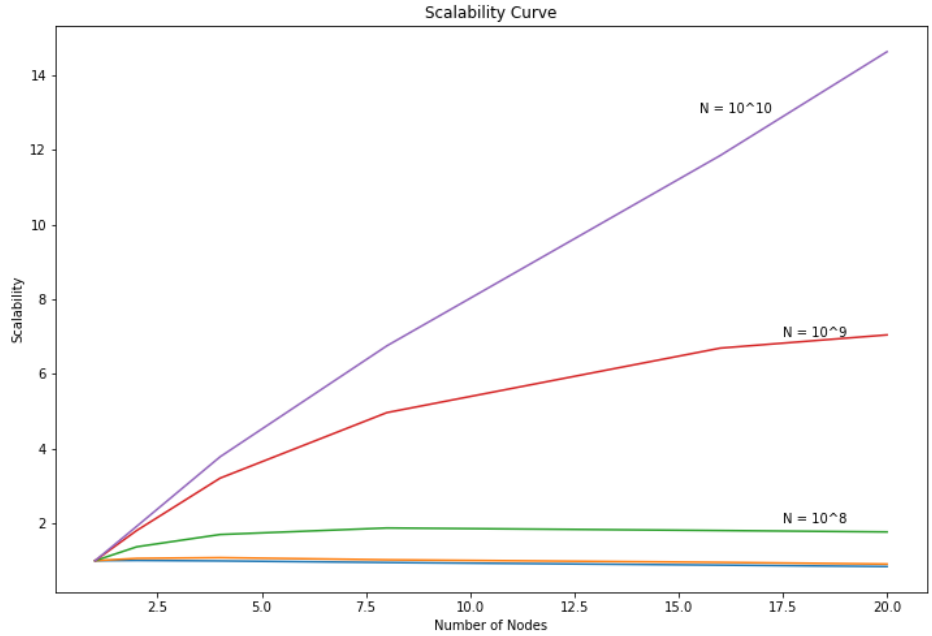
Refer to ../codes/section2_strong_runTime_elapsedTime.py

- For N = 10^6 you can find elapsed times for different processors below
  - 1 cores run has elapsed time: 1.57
  - 2 cores run has elapsed time: 1.56
  - 4 cores run has elapsed time: 1.58
  - 8 cores run has elapsed time: 1.65
  - 16 cores run has elapsed time: 1.78
  - 20 cores run has elapsed time: 1.86
- For N = 10^7 you can find elapsed times for different processors below
  - 1 cores run has elapsed time: 1.72
  - 2 cores run has elapsed time: 1.62
  - 4 cores run has elapsed time: 1.59
  - 8 cores run has elapsed time: 1.68
  - 16 cores run has elapsed time: 1.8
  - 20 cores run has elapsed time: 1.89
- For N = 10^8 you can find elapsed times for different processors below
  - 1 cores run has elapsed time: 3.47
  - 2 cores run has elapsed time: 2.53
  - 4 cores run has elapsed time: 2.04
  - 8 cores run has elapsed time: 1.85
  - 16 cores run has elapsed time: 1.92
  - 20 cores run has elapsed time: 1.96
- For N = 10^9 you can find elapsed times for different processors below
  - 1 cores run has elapsed time: 21.35
  - 2 cores run has elapsed time: 11.8
  - 4 cores run has elapsed time: 6.65
  - 8 cores run has elapsed time: 4.3
  - 16 cores run has elapsed time: 3.19
  - 20 cores run has elapsed time: 3.03
- For N = 10^10 you can find elapsed times for different processors below
  - 1 cores run has elapsed time: 199.69
  - 2 cores run has elapsed time: 103.92
  - 4 cores run has elapsed time: 52.79
  - 8 cores run has elapsed time: 29.57
  - 16 cores run has elapsed time: 16.85
  - 20 cores run has elapsed time: 13.65



Similar to max wall times, algorithm run time firstly decrease visibly, after some specific point this decrease begin to stop

Refer to ../codes/section2_strong_scalability_elapsedTime.py



Similar results are found for the same analyisis but this time with elapsed time. As previously mentioned algorithm tends to scale better when N is increased. However after some specific points for P, algorithm doesn't scale even P is increased. For example, it is not logical to increase P after 8 for N = 10^8. Because elapsed time increase.

0.3.4 2.2: identify a model for the parallel overhead

Tasks in a program can be split into serial tasks executed by a single worker and parallel tasks distributed to multiple workers. Then, the total runtime (execution time) of a program with n workers is given as:

$T(n) = T_p/n + T_s + P(n)$

where T_p is the runtime of parallelizable tasks with a single core, T_s is the runtime of a serial code, and $P(n)$ is parallelization overhead. When a set of tasks is distributed to n workers, then the ideal computing time of the parallelized tasks should be T_p/n. However, the actual time to compute the tasks in parallel requires extra P(n) seconds. Therefore, the parallelization overhead P(n) is defined as the additional time incurred from parallelizing a chunk of tasks to multiple processors. source

I couldn't determine the serial part in mpi_pi.c file and since it is constant it is not crucial to understand the overhead change with increasing number of P so I considered T_S as negligible, so P(n) is $P(n) = (T(n) - T_s) - T_p/n$.Final equaition for parallel overhead is $P(n) = T(n) - T_p/n$.

Refer to ../codes/section2_overhead_maxTime.py

0.3.5 Max Wall Time

- For N = 10^6 you can find run times and overheads for different processors below
  - 1 cores run has maximum run time: 0.01990104 overhead is 0.0
  - 2 cores run has maximum run time: 0.01076198 overhead is 0.00538099
  - 4 cores run has maximum run time: 0.00544691 overhead is 0.00413537
  - 8 cores run has maximum run time: 0.00333095 overhead is 0.0029799675
  - 16 cores run has maximum run time: 0.00339699 overhead is 0.0032974356249999997
  - 20 cores run has maximum run time: 0.00313091 overhead is 0.0030608625000000003
- For N = 10^7 you can find run times and overheads for different processors below
  - 1 cores run has maximum run time: 0.19876003 overhead is 0.0
  - 2 cores run has maximum run time: 0.10333514 overhead is 0.05166757
  - 4 cores run has maximum run time: 0.051332 overhead is 0.03850049
  - 8 cores run has maximum run time: 0.0273509 overhead is 0.0239531375
  - 16 cores run has maximum run time: 0.01808906 overhead is 0.017179314375
  - 20 cores run has maximum run time: 0.01323915 overhead is 0.01265295
- For N = 10^8 you can find run times and overheads for different processors below
  - 1 cores run has maximum run time: 1.98622179 overhead is 0.0
  - 2 cores run has maximum run time: 1.0219152 overhead is 0.5109576
  - 4 cores run has maximum run time: 0.51305103 overhead is 0.3848860224999999
  - 8 cores run has maximum run time: 0.27128315 overhead is 0.23737275624999998
  - 16 cores run has maximum run time: 0.14891601 overhead is 0.13988595124999997
  - 20 cores run has maximum run time: 0.11698103 overhead is 0.111198986
- For N = 10^9 you can find run times and overheads for different processors below
  - 1 cores run has maximum run time: 19.85913587 overhead is 0.0
  - 2 cores run has maximum run time: 10.22568107 overhead is 5.112840535
  - 4 cores run has maximum run time: 5.14612985 overhead is 3.8623858725000004
  - 8 cores run has maximum run time: 2.72066188 overhead is 2.3805810225000004
  - 16 cores run has maximum run time: 1.44949389 overhead is 1.359477256875
  - 20 cores run has maximum run time: 1.16037512 overhead is 1.1026759640000001
- For N = 10^10 you can find run times and overheads for different processors below
  - 1 cores run has maximum run time: 198.1946258 overhead is 0.0
  - 2 cores run has maximum run time: 102.394792 overhead is 51.197396
  - 4 cores run has maximum run time: 51.2605071 overhead is 38.445380325
  - 8 cores run has maximum run time: 27.986582 overhead is 24.586386649999998
  - 16 cores run has maximum run time: 15.0945041 overhead is 14.1934852875
  - 20 cores run has maximum run time: 11.862725 overhead is 11.286937649999999

When we increase the number of cores overhead tends to be smaller. Overhead must be zero for 1

cores because for 1 cores $T_n = T_s, and, n = 1$ it makes $P(n) = 0$.

Refer to ../codes/section2_overhead_elapsedTime.py

0.3.6 Elapsed time

- For N = 10^6 you can find elapsed times for different processors below
  - 1 cores run has elpased time: 1.5
  - 2 cores run has elpased time: 1.56
  - 4 cores run has elpased time: 1.58
  - 8 cores run has elpased time: 1.65
  - 16 cores run has elpased time: 1.78
  - 20 cores run has elpased time: 1.86

Overhead for N 10^6 [0.0, 0.775, 1.1875, 1.4537499999999999, 1.681875, 1.7815]

- For N = 10^7 you can find elapsed times for different processors below
  - 1 cores run has elpased time: 1.72
  - 2 cores run has elpased time: 1.62
  - 4 cores run has elpased time: 1.59
  - 8 cores run has elpased time: 1.68
  - 16 cores run has elpased time: 1.8
  - 20 cores run has elpased time: 1.89

Overhead for N 10^7 [0.0, 0.7600000000000001, 1.1600000000000001, 1.4649999999999999, 1.6925000000000001, 1.8039999999999998]

- For N = 10^8 you can find elapsed times for different processors below
  - 1 cores run has elpased time: 3.47
  - 2 cores run has elpased time: 2.53
  - 4 cores run has elpased time: 2.04
  - 8 cores run has elpased time: 1.85
  - 16 cores run has elpased time: 1.92
  - 20 cores run has elpased time: 1.96

Overhead for N 10^8 [0.0, 0.7949999999999997, 1.1724999999999999, 1.41625, 1.703125, 1.7865

- For N = 10^9 you can find elapsed times for different processors below
  - 1 cores run has elpased time: 21.35
  - 2 cores run has elpased time: 11.8
  - 4 cores run has elpased time: 6.65
  - 8 cores run has elpased time: 4.3

  - 16 cores run has elpased time: 3.19
  - 20 cores run has elpased time: 3.03

Overhead for N 10^9 [0.0, 1.125, 1.3125, 1.6312499999999996, 1.8556249999999999, 1.9624999999999997]

- For N = 10^10 you can find elapsed times for different processors below
  - 1 cores run has elapsed time: 199.69
  - 2 cores run has elapsed time: 103.92
  - 4 cores run has elpased time: 52.79
  - 8 cores run has elpased time: 29.57
  - 16 cores run has elpased time: 16.85
  - 20 cores run has elpased time: 13.65

Overhead for N 10^10 [0.0, 4.075000000000003, 2.8674999999999997, 4.608750000000001, 4.369375000000002, 3.6654999999999998]

0.3.7 2.3: weak scaling with max walltime

- Now let us do some running that constitutes a weak scaling test.
  - This means increasing the problem size simultaneously with the number of nodes being used. In the present case, increasing the number of iterations, Niter.

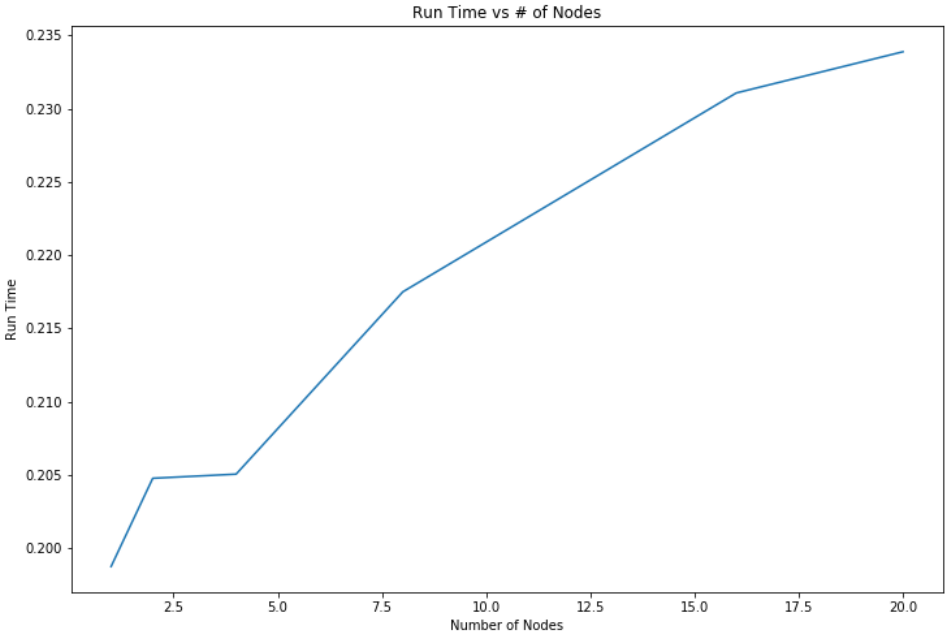
I have used this bash script to increase the problem size simultaneously with the number of nodes being used.

```
bash for procs in 1 2 4 8 16 20 ; do /usr/bin/time mpirun -np ${procs} ./mpi_pi.x
$((${procs}*10000000)); done &> output_weakscaling.txt
```

- Record the run time for each number of nodes and make a plot of the run time versus number of computing nodes.

Refer to ../codes/section2_weakscaling_maxTime.py

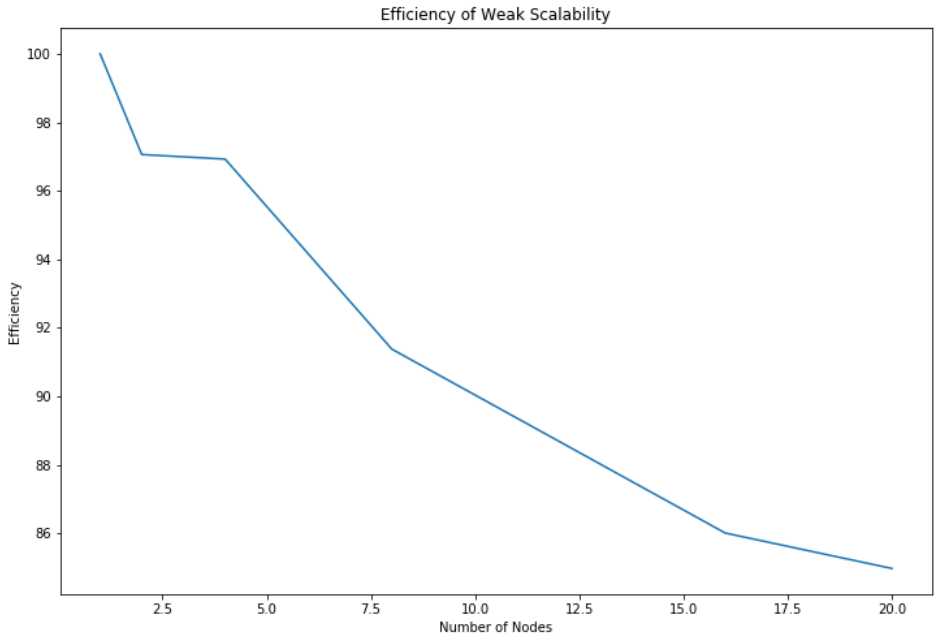
- 1 cores run has maximum run time: 0.198735
- 2 cores run has maximum run time: 0.20475984
- 4 cores run has maximum run time: 0.20504093
- 8 cores run has maximum run time: 0.21748805
- 16 cores run has maximum run time: 0.23106313
- 20 cores run has maximum run time: 0.2338779



According to logic of weak scalability when we increase the number of cores with the N run time supposed to be same. However for this example there are small differences. There might be some room for optimizaiton of parallelization part.

- Plot on the same graph the efficiency (T(1)/T(p)) of weak scalability for different number of moves and comment the results obtained.

Refer to ../codes/section2_weakscaling_maxTime_2.py

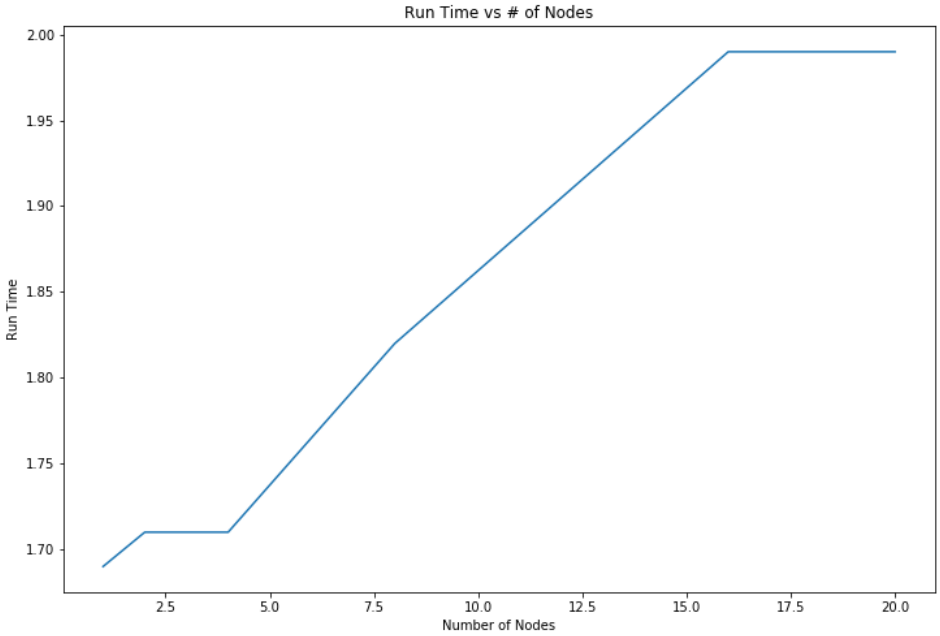


Similar results as previous findings, for the efficiency of weak scalability decrease when N is increased.

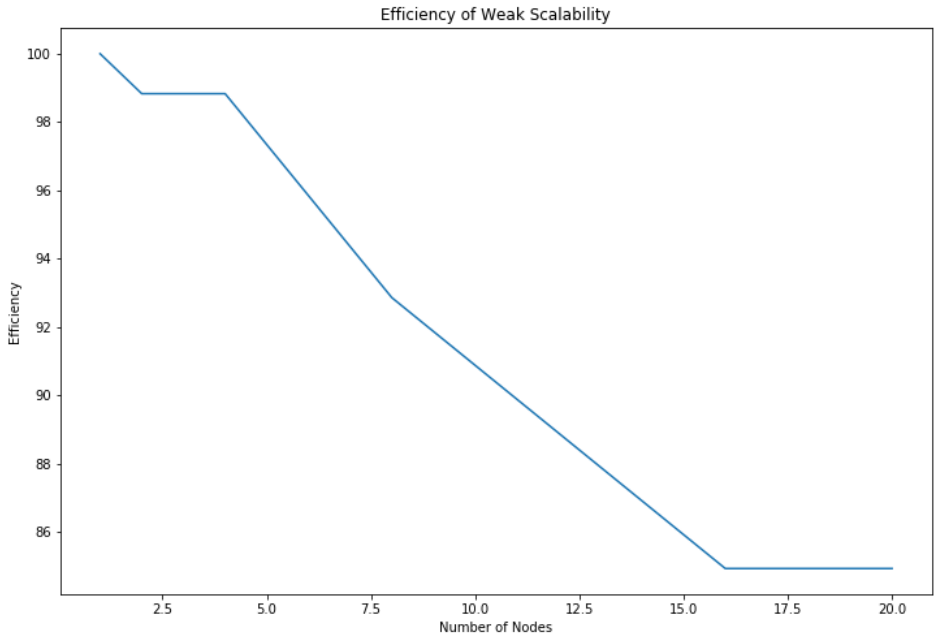
0.3.8 weak scaling with elapsed time

Refer to ../codes/section2_weakscaling_elapsedTime.py

- 1 cores run has elapsed time: 1.69
- 2 cores run has elapsed time: 1.71
- 4 cores run has elapsed time: 1.71
- 8 cores run has elapsed time: 1.82
- 16 cores run has elapsed time: 1.99
- 20 cores run has elapsed time: 1.99



Refer to ../codes/section2_weakscaling_elapsedTime.py



Similar results are obtained from the analysis of weak scalability for elapsed time.

0.4 Section 3: implement a parallel program using MPI

0.4.1 Naive Communicaiton

For section 3, I have used python to implement the algorithm. Program read an input from file input_file.txt. Every processors sum its partial array and send it to master processor, master processor collect the all partial results and if there is reminder deal with it. Finally master processor adds all the parital sums and reminder sum if there is.

Refer to ../codes/sumNumbers_mpi.py

First performance of program was not good. Therefore I tried to make it better. For example I was creating whole array for all processors from 1 to N. I modified and eliminate this part. Run time decreased after this change.

0.4.2 Collective Communicaiton

Most of the logic is same for this approach, only difference is communication channel which is reduce for this part.

Refer to ../codes/sumNumbers_mpi_collective.py

- measure the reading time, the communication times (initial distribution phase and final collectin phase) and the computation time (hints: use MPI_walltime)

I modified sumNumbers.py file to determine reading time, communicaiton times and computation time. I run my program for 4 cores in Ulysses and save the results section4_output_assumptions.txt. After that I sum all times to find final results

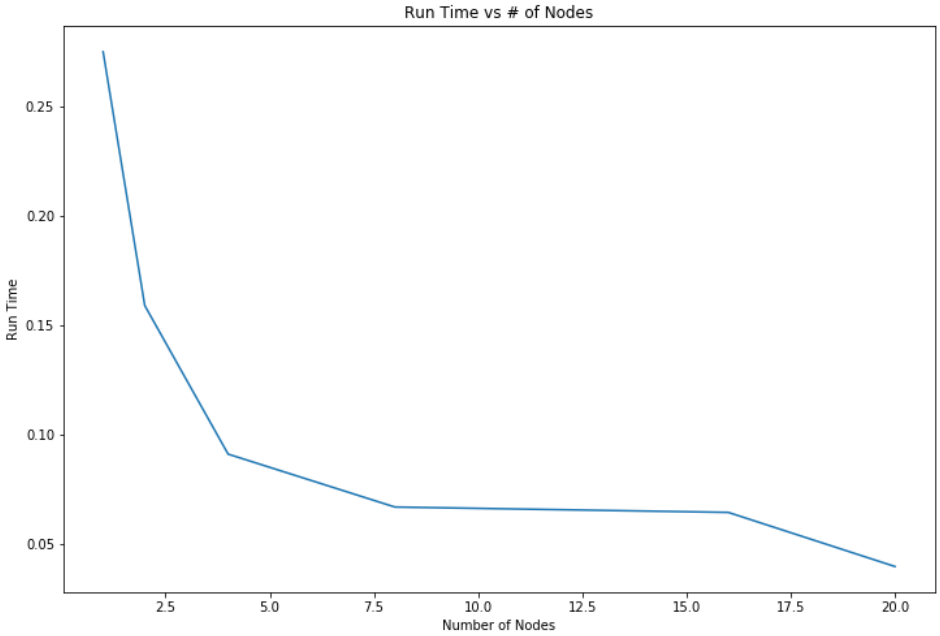
0.5 Section 4 (run and compare)

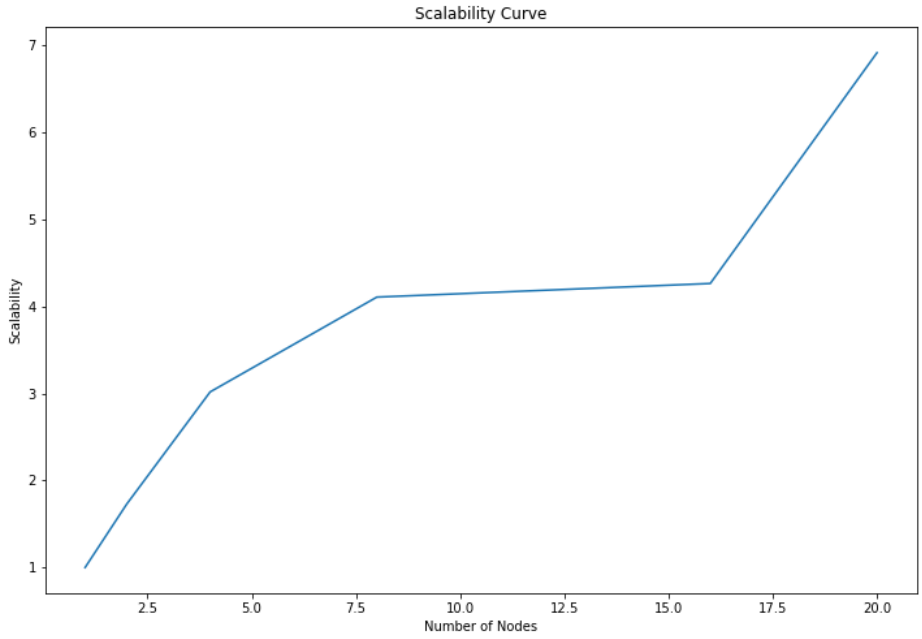
- Run the implemented program for different large enough sizes of the array
  - I run the program for different large enough sizes of the array by using `for arrays in 1000 10000 99999 1000000 9999999 10000000 ; do mpiexec -n 4 python sumNumbers_mpi.py ${arrays} ; done > section4_output_different_size_arrays.txt`.
- Plot as in section 3 scalability of the program

0.5.1 Max Time Naive

Refer to ../codes/section4_runtime&scalability_naive_maxTime.py

- 1 cores has maximum run time: 0.2748489379882812
- 2 cores has maximum run time: 0.1589272022247314
- 4 cores has maximum run time: 0.0910718441009521
- 8 cores has maximum run time: 0.066909074783325
- 16 cores has maximum run time: 0.0644559860229492
- 20 cores has maximum run time: 0.0397398471832275



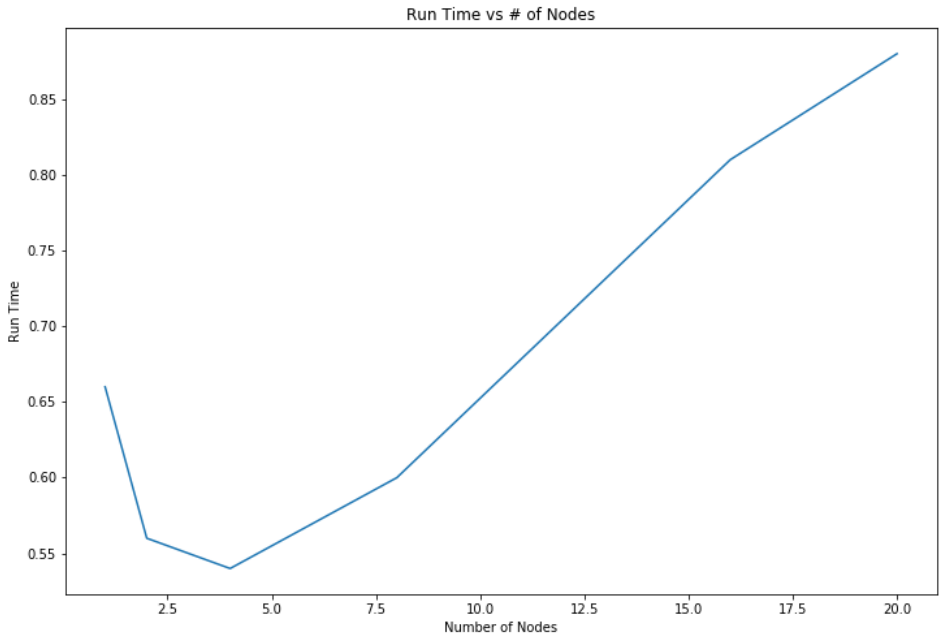


According to results when I increase the number of nodes run time decrease and it scales. But there are some problem about unbalance (especially when there is reminder run time of master processors take more time e.g node = 16) of work between the processors.

0.5.2 Elapsed Time Naive

Refer to ../codes/section4_runtime&scalability_naive_elapsed.py

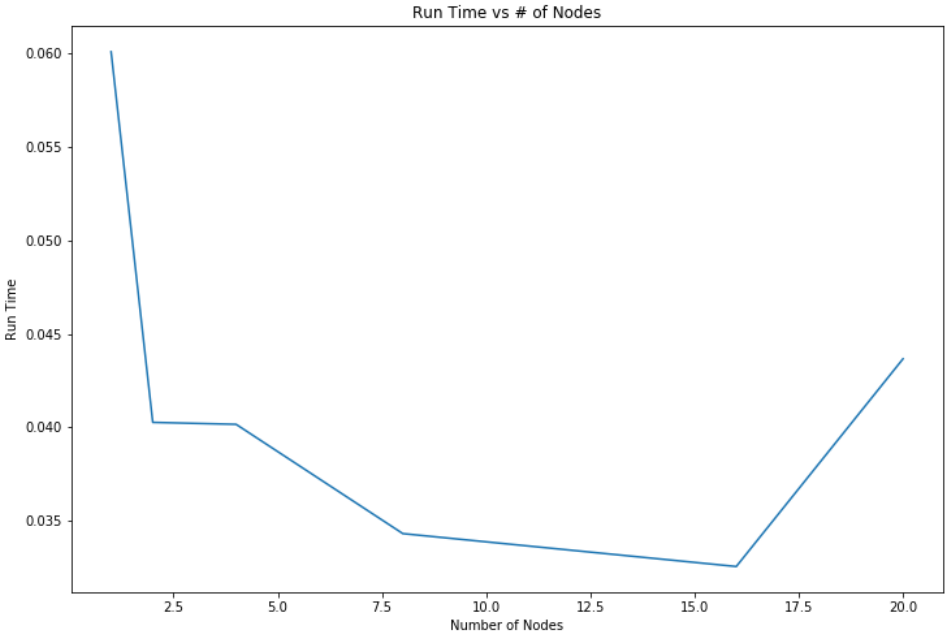
- 1 cores run has elapsed time: 0.66
- 2 cores run has elapsed time: 0.56
- 4 cores run has elapsed time: 0.54
- 8 cores run has elapsed time: 0.6
- 16 cores run has elapsed time: 0.81
- 20 cores run has elapsed time: 0.88

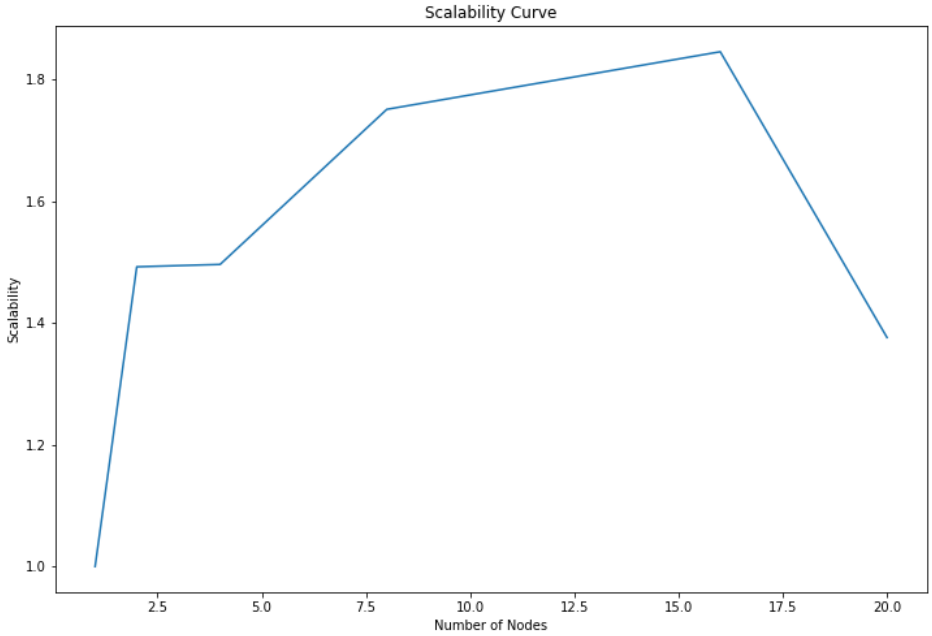


0.5.3 Max Time Collective

Refer to ../codes/section4_runtime&scalability_collective_maxTime.py

- 1 cores has maximum run time: 0.0600860118865966
- 2 cores has maximum run time: 0.0402710437774658
- 4 cores has maximum run time: 0.040165901184082
- 8 cores has maximum run time: 0.03432393074035644
- 16 cores has maximum run time: 0.0325651168823242
- 20 cores has maximum run time: 0.0436670780181884



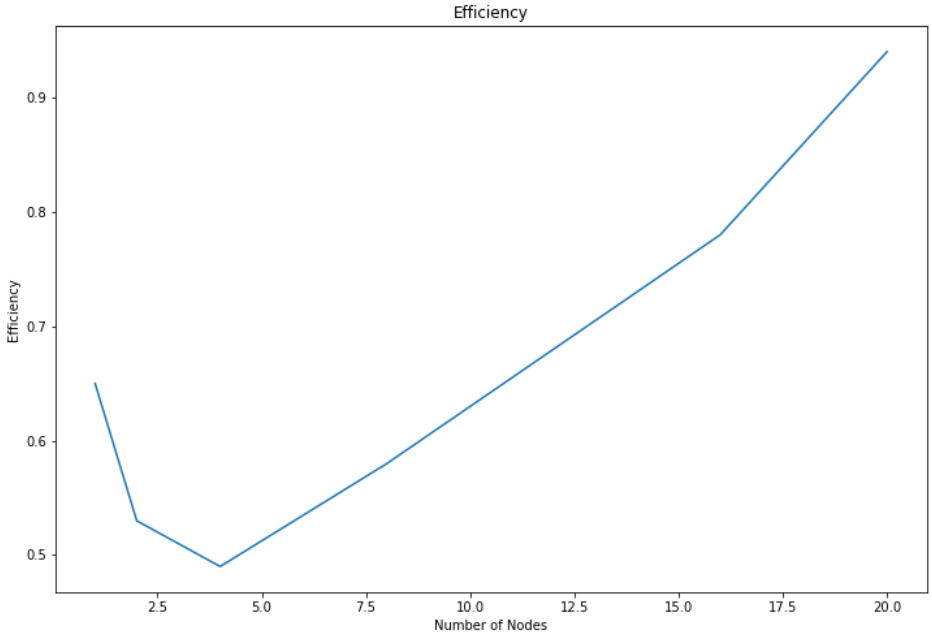


My program for collective operation has some problem. Maybe run time dominating from communication time when I increase the node.

0.5.4 Elapsed Time Collective

Refer to ../codes/section4_runtime&scalability_collective_elapsed.py

- 1 cores run has elapsed time: 0.65
- 2 cores run has elapsed time: 0.53
- 4 cores run has elapsed time: 0.49
- 8 cores run has elapsed time: 0.58
- 16 cores run has elapsed time: 0.78
- 20 cores run has elapsed time: 0.94



- compare performance results obtained against the performance model elaborated in section 2

Program in the section 2 was more stable then my program and it scales better then my program. Apperantly the correction that I have done is not sufficient make it more stable and scalable. Extra steps should be applied.

- comment on the assumption made in section 1 about times: are they correct ? do you observe something different ?

[6]:
```
#Assumptions
T_comp = 2e-09
T_read = 1e-04
T_comm = 1e-06
```

I measure reading time for master processor by modifying sumNumbers_mpi.py file for 4 cores and reading time is 0.00017404556274414062 which is more or less same with the assumption(for master node)

Computaiton time for master is 0.03702187538146973 and slaves (0.03630995750427246, 0.029160022735595703, 0.027962923049926758) totally 0.10129475593566895 which is different from theoretical model.

Lastly I measure the communication time (sending and receiving seperately) slave 1 to master node ending is 0.00012493133544921875 , slave 2 to master node sending 0.029160022735595703, slave 3 to master node sending time is 0.027962923049926758 which is totaly 0.05724787712097168 sending time.

Receving time respectively $1.4066696166992188e-05$, $7.891654968261719e-05$ and which is totaly 0.03824305534362793. Total communicaiton time is 0.09549093246459961. It is also different from

assumptions.