***GOVERNMENT COLLEGE UNIVERSITY***

****

***NAME:***

***Muhammad Bilal***

***ROLL NO:***

***0134-BSCS-19(B)***

***SEMESTER-7(2019-2023)***

***SUBJECT:***

***“Parallel & Distributed Computing”***

***SUBMITTED TO:***

***Sir Umair Sadiq***

**Sieve of Eratosthenes**

Sieve of Eratosthenes commonly known as Seive’s Algo is a simple algorithm to find prime numbers from 1 - upper limit.Eratosthenes was an ancient Greek Mathematician. It is one of the most efficient algorithms and works iteratively by checking till sqrt(N) or N/2 only ,then marking/cutting off the numbers which are not prime, starting from 2, For example :

Pre-Condition : 2, 3 ,4 ,5 ,6 ,7 ,8 ,9 ,10

Post-Condition : 2, ~~3~~ ,4 ,~~5~~ ,6 ,~~7~~ ,8 ,9 ,10

This algo is parallelizable. We first do data distribution block/cyclic then keep broadcasting elements from 2 to sqrt(N) or N/2 to all processes to cancel their multiples in their part of data.

Important Note :

* we should not employ more processors than the number of cores of our system. Because as soon as we employ more processors than the cores of the system all the performance drastically takes a nosedive.
* Parallel code only makes a difference if we have l;arge array otherwise there's no significant change in performance and it may also decrease little because of inter process communication and small array size.

**SourceCode**

#include <mpi.h>

#include <cstdlib>

#include <stdio.h>

void getMyNumbersArray(int arr[] , int size,int processNumber, int noOfProcesses)

{

int j = 0;

int i;

for (i = processNumber+2; j < size; i+=noOfProcesses, j++) {

arr[j] = i;

}

}

int findNextIndexOfMultipleToMark(int arr[] ,int size,int currentIndex)

{

int i;

for(i=currentIndex; i < size; i++)

{

if(arr[i]!= -1)

return i;

}

return -1;

}

int getNoOfTasksToSolve(int processNumber, int totalSize, int noOfProcesses) {

int remain = totalSize % noOfProcesses;

int addMore = 1;

if (remain <= processNumber)

addMore = 0;

int noOfTasks = (totalSize / noOfProcesses) + addMore;

return noOfTasks;

}

int getTotalTasksSolved(int processNumber, int totalSize, int noOfProcesses)

{

int extraTasks = 0;

int equalTasks = (totalSize / noOfProcesses);

if(totalSize < noOfProcesses)

{

if(processNumber+1 <= totalSize)

extraTasks = processNumber;

}

else if(totalSize > noOfProcesses)

{

extraTasks = totalSize % noOfProcesses;

if(processNumber+1 <= extraTasks)

extraTasks = processNumber;

}

else

extraTasks = 0;

return (equalTasks \* processNumber) + extraTasks;

}

int main(int argc, char \*\*argv)

{

int myrank, noOfProcesses ;

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myrank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&noOfProcesses);

int totalNumbers = 20000;

int totalSize = totalNumbers - 1;

const int mySize = getNoOfTasksToSolve(myrank,totalSize,noOfProcesses);

if(mySize != 0)

{

int myNumbers[mySize];

getMyNumbersArray(myNumbers,mySize,myrank,noOfProcesses);

int current\_Index = 0 , i;

int numberToMarks = 0;

int flagHolder = 0;

int j;

for(j=0; j < (totalSize % noOfProcesses) + (totalSize / noOfProcesses); j++)

{

if(flagHolder == myrank)

{

current\_Index = findNextIndexOfMultipleToMark(myNumbers,mySize,current\_Index);

if(current\_Index != -1)

numberToMarks = myNumbers[current\_Index];

else

numberToMarks = -1;

current\_Index++;

}

MPI\_Bcast(&numberToMarks,1,MPI\_INT,flagHolder,MPI\_COMM\_WORLD);

if(numberToMarks != -1)

{

for(i=current\_Index; i < mySize; i++)

{

if(myNumbers[i] % numberToMarks == 0)

myNumbers[i] = -1;

}

}

flagHolder = (flagHolder +1) % noOfProcesses;

}

for(i=0; i<mySize; i++)

{

if(myNumbers[i] != -1)

printf("%i is Prime\n",myNumbers[i]);

}

}

MPI\_Finalize();

}

**Parallel Evaluation**

**Key :**

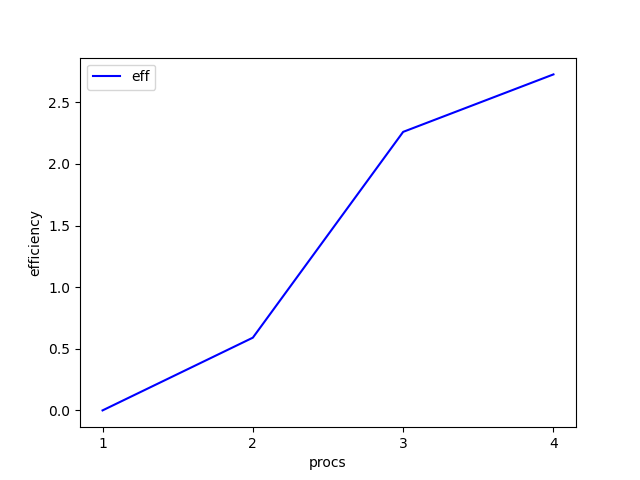
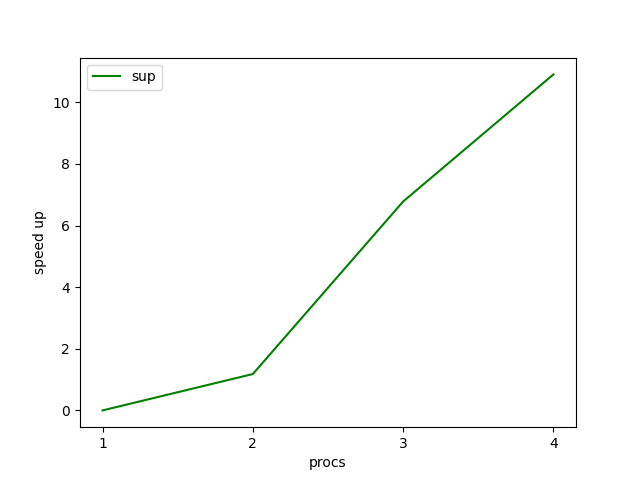
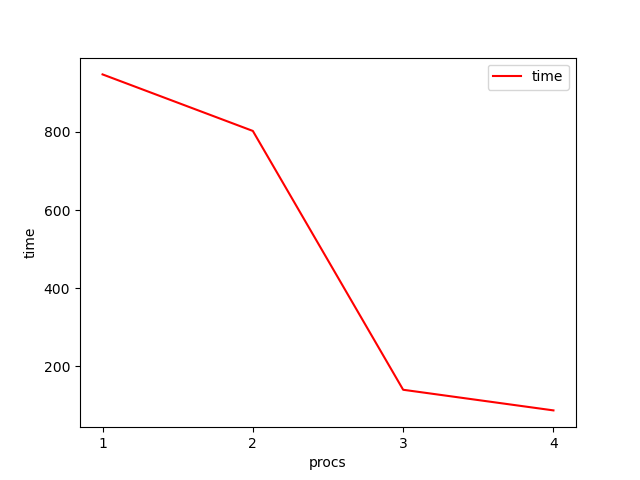
* **Red : proc vs time**
* **Green : proc vs speedup**
* **Blue :** proc vs efficiency

**Scaling the values down by factor of 10**

Array Size : 500000 mapped on all good possible number of processes

As the size of the array is very large,it is performing really well.

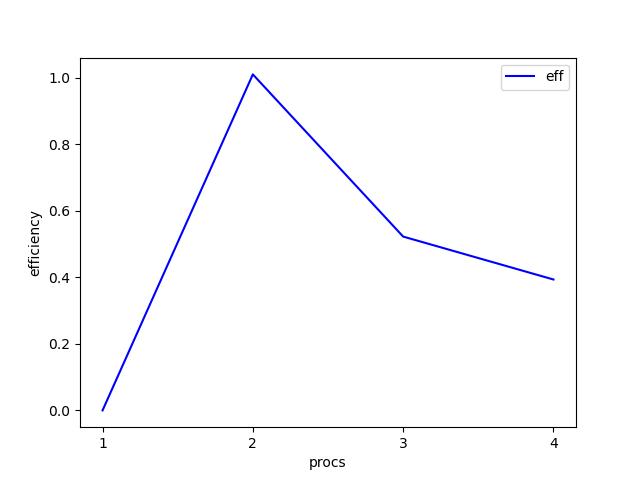
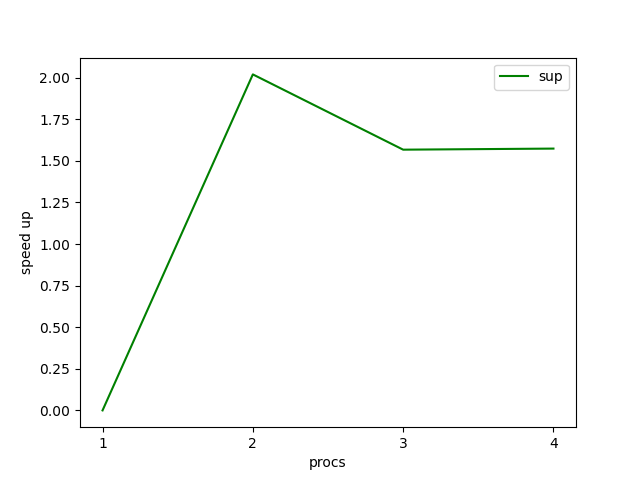
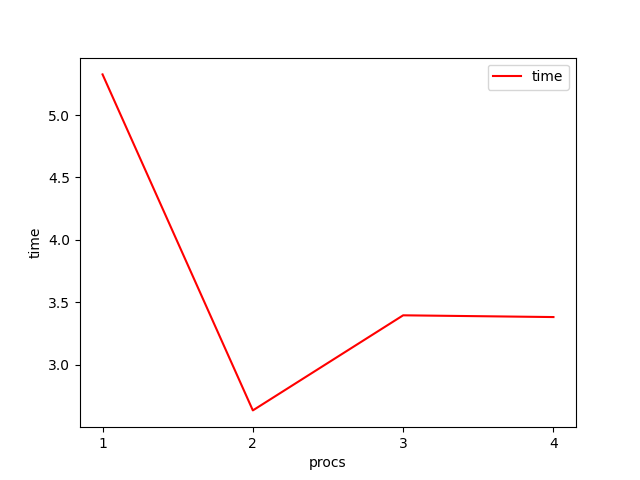
| Procs | Value | time | time in secs | speedup | Efficiency |
| --- | --- | --- | --- | --- | --- |
| 4 | 500000 | 1m26.875s | 86.875 | 10.90650935 | 2.726627338 |
| 3 | 500000 | 2m19.736s | 139.736 | 6.780664968 | 2.260221656 |
| 2 | 500000 | 13m22.682s | 802.682 | 1.180421387 | 0.5902106936 |
| 1 | 500000 | 15m47.503s | 947.503 | 0 | 0 |



Array Size : 50000 mapped on all good possible number of processes

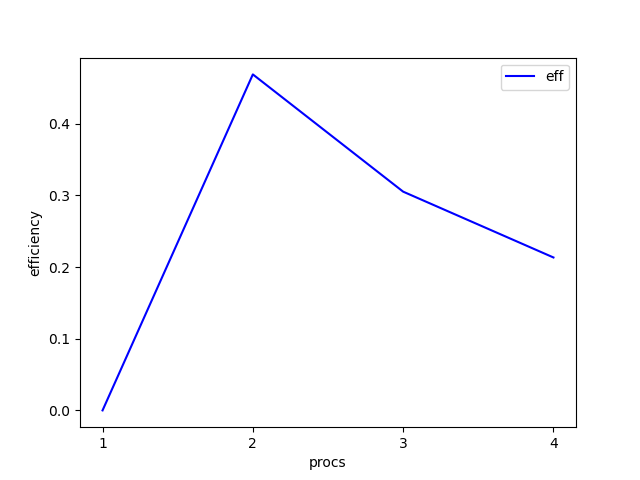
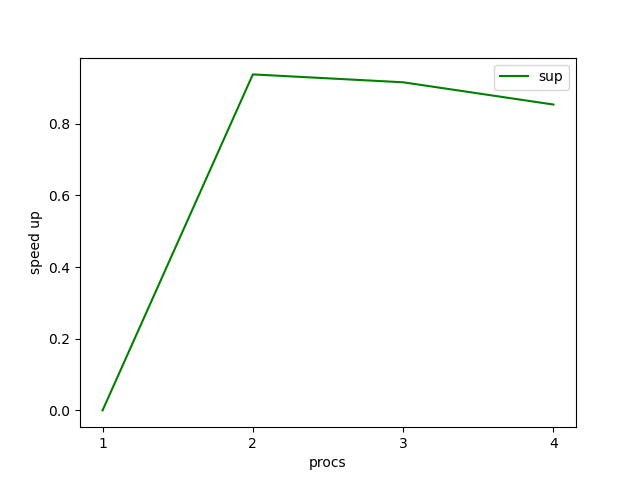
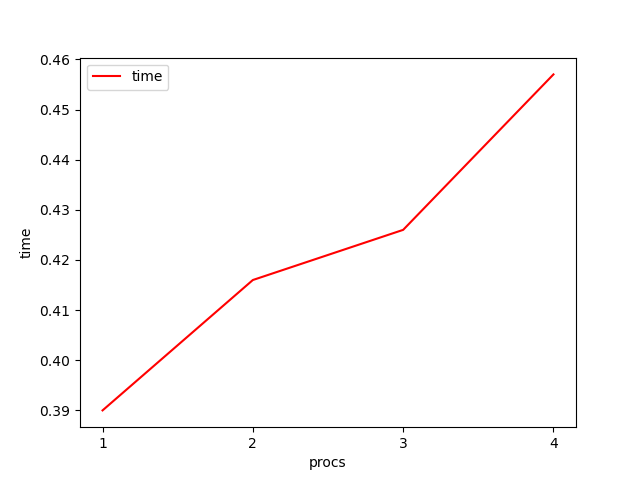
Still a significant gain by increase in processors.

|  |  | ⬇️ |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Procs | Value | time | time in secs | speedup | Efficiency |
| 4 | 50000 | 0m3.383s | 3.383 | 1.574046704 | 0.393511676 |
| 3 | 50000 | 0m3.397 | 3.397 | 1.567559611 | 0.5225198705 |
| 2 | 50000 | 0m2.636s | 2.636 | 2.020106222 | 1.010053111 |
| 1 | 50000 | 0m5.325s | 5.325 | 0 | 0 |



Array Size : 5000 mapped on all good possible number of processes

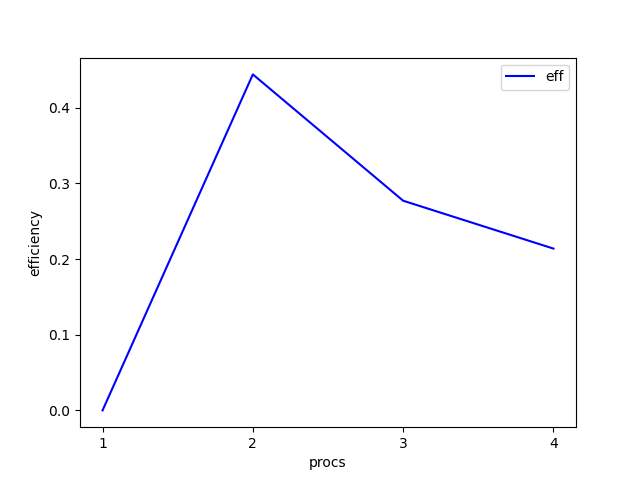
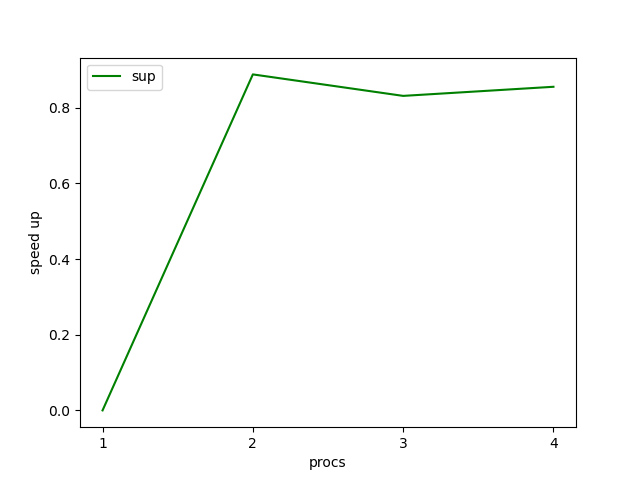
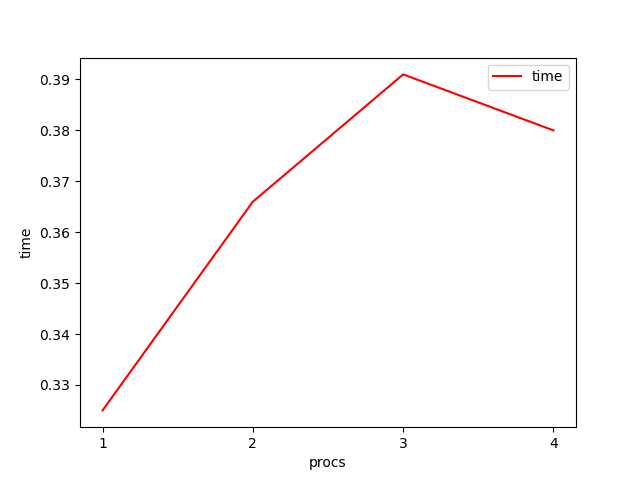
|  |  | ⬇️ |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Procs | Value | time | time in secs | speedup | Efficiency |
| 4 | 5000 | 0m0.457s | 0.457 | 0.8533916849 | 0.2133479212 |
| 3 | 5000 | 0m0.426s | 0.426 | 0.9154929577 | 0.3051643192 |
| 2 | 5000 | 0m0.416s | 0.416 | 0.9375 | 0.46875 |
| 1 | 5000 | 0m0.39s | 0.39 | 0 | 0 |



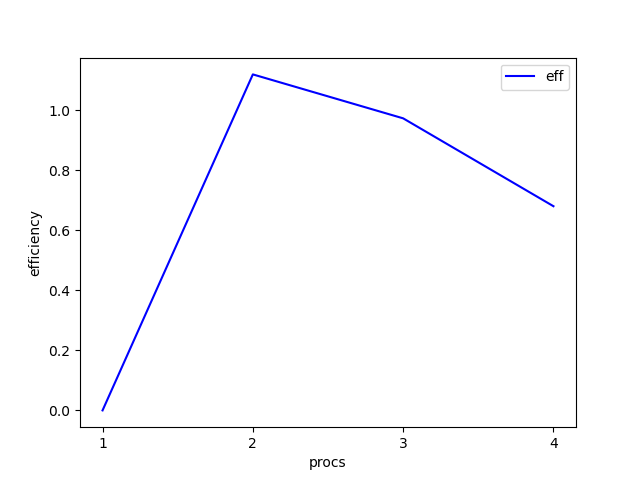
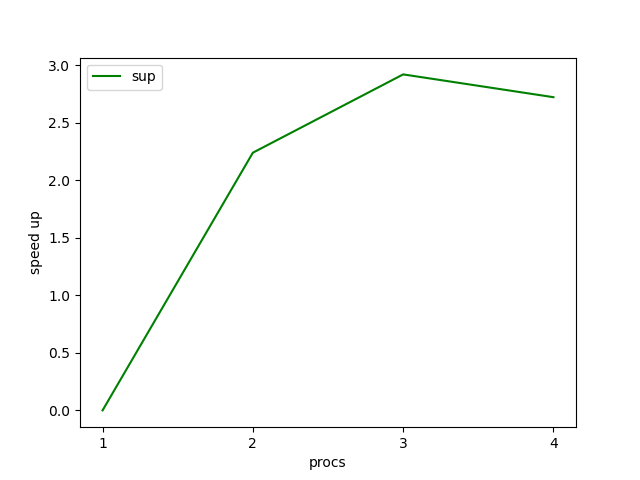
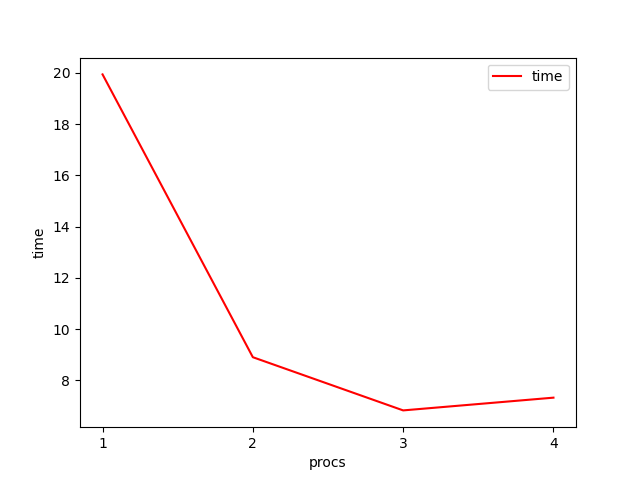
Array Size : 500 mapped on all good possible number of processes

As the array is very small so there's no considerable increase in performance instead it has deteriorated a little bit.

|  |  | ⬇️ |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Procs | Value | time | time in secs | speedup | Efficiency |
| 4 | 500 | 0m0.38s | 0.38 | 0.8552631579 | 0.2138157895 |
| 3 | 500 | 0m0.391s | 0.391 | 0.831202046 | 0.2770673487 |
| 2 | 500 | 0m0.366s | 0.366 | 0.8879781421 | 0.443989071 |
| 1 | 500 | 0m0.325s | 0.325 | 0 | 0 |

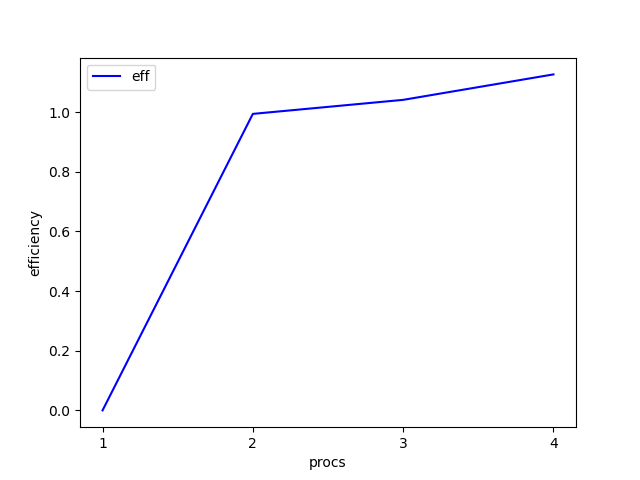
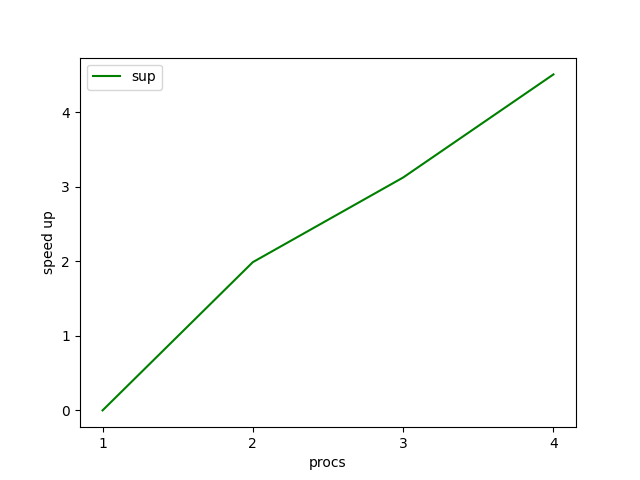
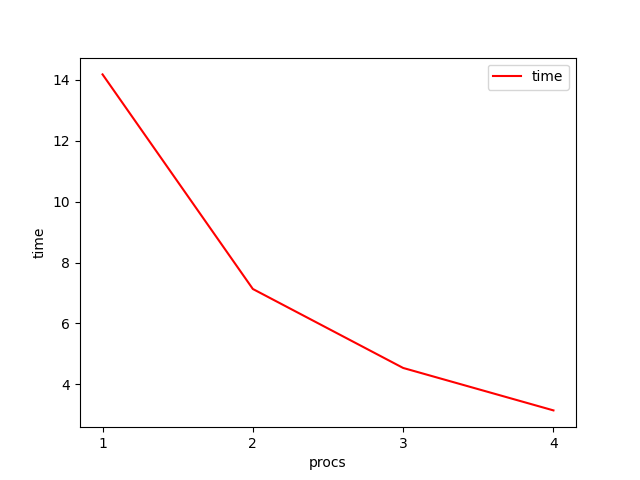


|  |  | **Some more random values** | |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Array Size : 100000 mapped on all good possible number of processes | | | |  |
| Procs | Value | time | time in secs | speedup | Efficiency |
| 4 | 100000 | 0m7.326s | 7.326 | 2.721130221 | 0.6802825553 |
| 3 | 100000 | 0m6.829s | 6.829 | 2.919168253 | 0.9730560843 |
| 2 | 100000 | 0m8.904s | 8.904 | 2.238881402 | 1.119440701 |
| 1 | 100000 | 0m19.935s | 19.935 |  |  |



Array Size : 85000 mapped on all good possible number of processes

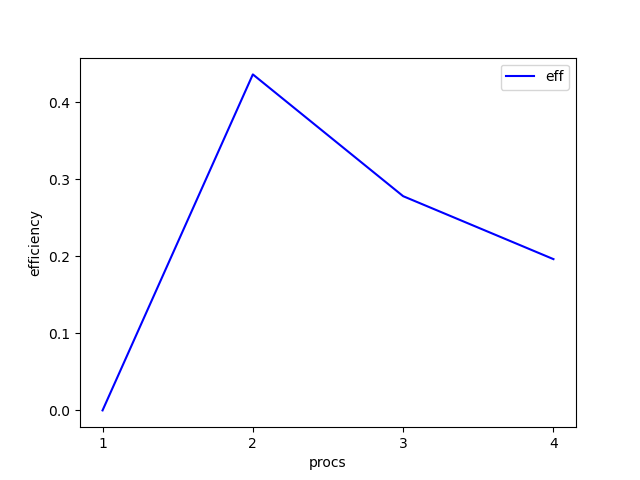
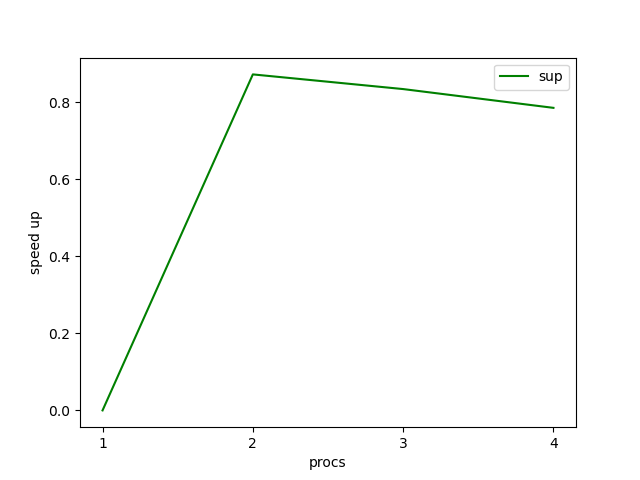
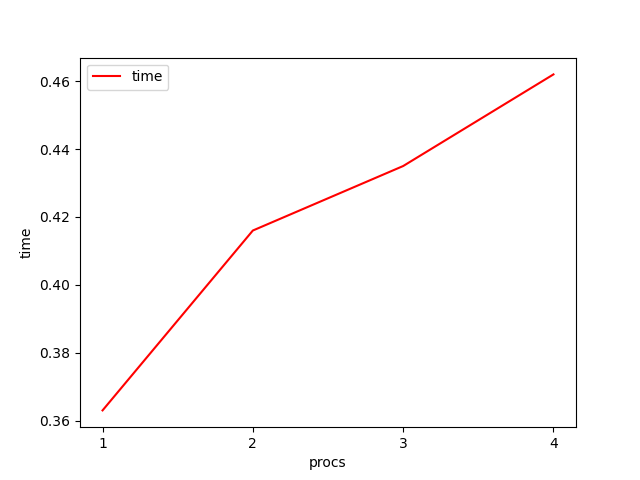
| Procs | Value | time | time in secs | speedup | Efficiency |
| --- | --- | --- | --- | --- | --- |
| 4 | 85000 | 0m3.147s | 3.147 | 4.5042898 | 1.12607245 |
| 3 | 85000 | 0m4.54s | 4.54 | 3.122246696 | 1.040748899 |
| 2 | 85000 | 0m7.132s | 7.132 | 1.987521032 | 0.993760516 |
| 1 | 85000 | 0m14.175s | 14.175 | 0 | 0 |



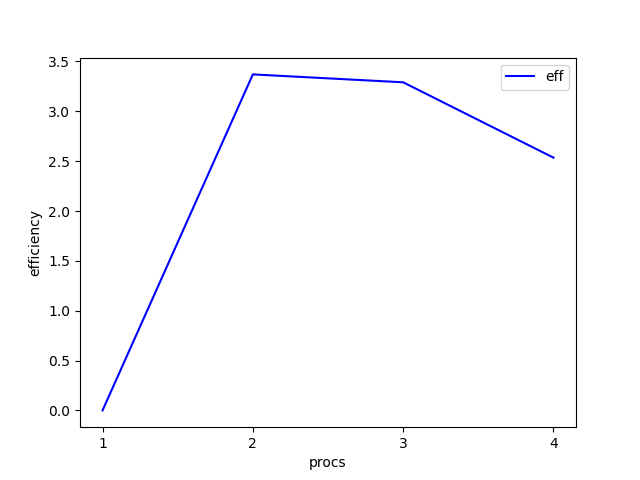
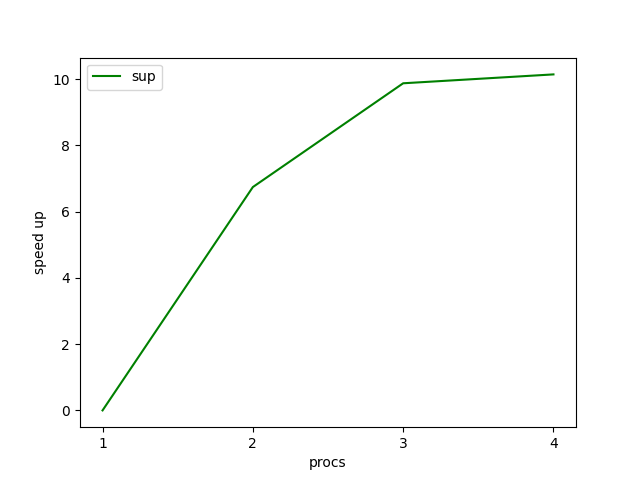
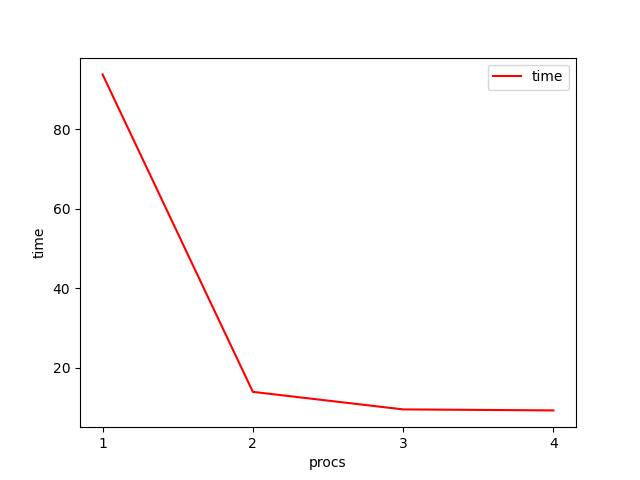
Array Size : 850 mapped on all good possible number of processes

Again the array is very small so there's no considerable increase in performance instead it has deteriorated a little bit and took more time.

| Procs | Value | time | time in secs | speedup | Efficiency |
| --- | --- | --- | --- | --- | --- |
| 4 | 850 | 0m0.462s | 0.462 | 0.7857142857 | 0.1964285714 |
| 3 | 850 | 0m0.435s | 0.435 | 0.8344827586 | 0.2781609195 |
| 2 | 850 | 0m0.416s | 0.416 | 0.8725961538 | 0.4362980769 |
| 1 | 850 | 0m0.363s | 0.363 | 0 | 0 |



| **Exceptionally large value**  Array Size : 500000 mapped on all good possible number of processes  Here the value is very large so it's performing well, resulting in significant speedup. | | | | | |
| --- | --- | --- | --- | --- | --- |
| Procs | Value | time | time in secs | speedup | Efficiency |
| 4 | 125671 | 0m9.243s | 9.243 | 10.1415125 | 2.535378124 |
| 3 | 125671 | 0m9.494s | 9.494 | 9.873393722 | 3.291131241 |
| 2 | 125671 | om13.906s | 13.906 | 6.740831296 | 3.370415648 |
| 1 | 125671 | 1m33.738 | 93.738 | 0 | 0 |



**Special Case/Major Issue**

As soon as we employ more processors than the cores of our system all the performance drastically takes a nosedive.   
Time : increases many folds

Speed up : dives fully

Efficiency : also immediately goes down.

| Procs | Value | time | time in secs | speedup | Efficiency |
| --- | --- | --- | --- | --- | --- |
| 2 | 20000 | 0m2.67s | 2.67 | 0.4539325843 | 0.2269662921 |
| 4 | 20000 | 0m1.49s | 1.49 | 0.8134228188 | 0.2033557047 |
| 5 | 20000 | 0m38.353s | 38.353 | 0.03160117853 | 0.00632023575 |
| 3 | 20000 | 0m1.563s | 1.563 | 0.7754318618 | 0.2584772873 |
| 1 | 20000 | 0m1.212s | 1.212 | 0 | 0 |

