INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

ES 611: Algorithms on Advanced Computer Architectures

Report on Computational Lab Project 1.1
by
Shashank Heda
January 23rd, 2013

ABSTRACT

This project is based on key findings about nature and behavior of parallel codes written using MPI. It consists of analysis and basic differences between performance of codes written using Point to point Communication and Collective communication and study based on benefits achieved by using variable number of processors for parallel computing.

Ping-Pong Using 2 Processors on the HPC Cluster

1.1 Introduction

A program to print the values of data sent and received by two processors after they ping each other. This program would help understand the sequence of execution of MPI_Send and MPI_Recv statements when used in a basic parallel code.

1.2 Method of Approach

In order to certify that the two processors send and receive correct values, the processors are made to exchange their ranks with each other as the ranks sent and received can be cross-checked with the actual data known to us. Moreover, it is taken care that the respective send and receive statements for the two processors are in proper sequence as there must not be a condition where a receive command is not found at an appropriate place for a particular send statement.

1.3 Results and Discussion

Output of the Program:

```
Processor 0 sends 0 to processor 1
Processor 0 receives 1 from Processor 1
Total time elapsed: 0.000042 seconds

Processor 1 receives 0 from Processor 0
Processor 1 sends 1 to processor 0
Total time elapsed: 0.000039 seconds
```

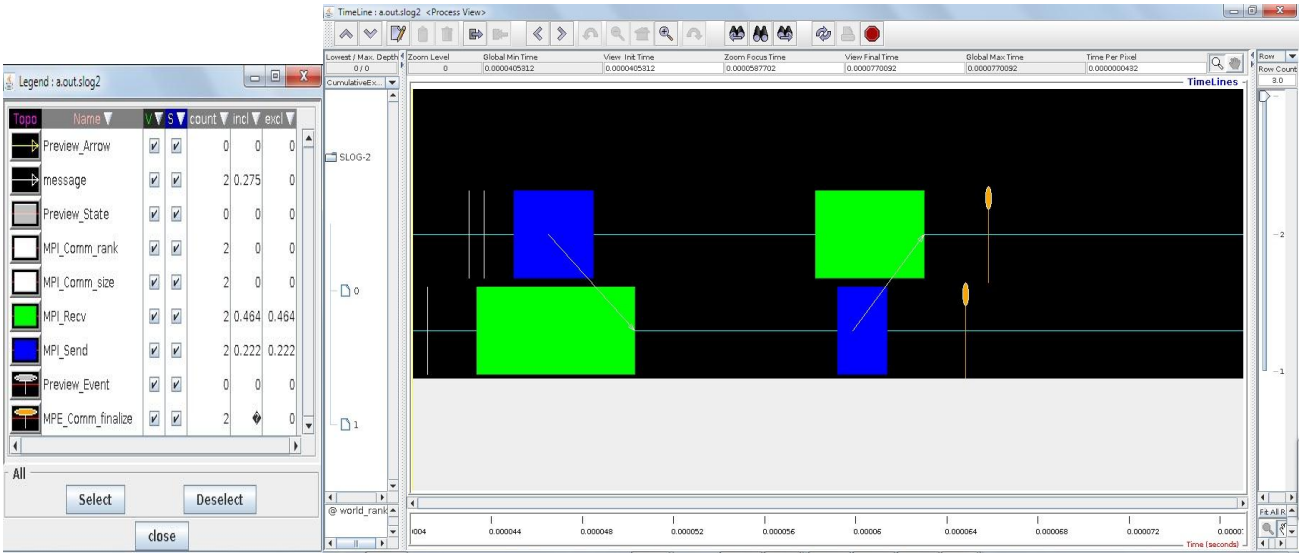
1.3.1 When we use a simple MPI_Send and MPI_Recv, it is noticed that a processor, after sending the required data/fields to another processor, does not idle. Rather, it goes on executing the next set of statements irrespective of the time taken by the data receiving processor in receiving the data sent by first processor. Such situations can alter the execution sequence which the program is intended to follow.

For example, say, a program has a critical section where Processor 0 sends some data to Processor 1. When simple MPI send and receive statements are used, the Processor 0, after sending the required data to processor1, goes on to execute the statements just after the

critical section of the code. On the other hand, Processor 1 takes time to receive the data sent by processor 0, and thus lags behind. This would alter the overall sequence which both the Processors together are intended to follow.

Such situations must be taken care of and should be appropriately put in synchronization by using available commands in MPI Library. In the above case, MPI_Ssend (Synchronous send) or MPI_Barrier may be used in order that both the Processors run at the same pace.

1.3.2 When multiple MPI send and receive statements are used, the sequence of the statements in the code becomes important. It may happen that a processor might not receive data sent by another processor if its MPI_Recv does not occur in synchronization with the MPI_Send of another processor in the parallel code.



(a) **(b)**

Figure 1: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

1.3.3 Tags of the respective MPI send and receive statements must match in order that proper data is sent and received at the two ends. Improper tags may cause a great deal of problems mainly when multiple MPI send and receive statements are used.

1.3.4 The jumpshot (Java based visualization tool) pictures shown in Figure 1: (a) and (b) present a graphical view (through a timeline) of the message passing between processors. We can get a fair idea about the time taken by both the processors in finishing up their jobs. It may also be used in Processor Load optimization in case when idling time for a processor is seen to be quite high when compared to other processors.

1.3.5 It is seen that if the sequence of statements is changed in the two processors, i.e., the statement execution sequence is written as Send, Recv (p=0), Send, Recv (p=2), then also the same jumpshot figure (shown in Fig. 1) is obtained. Thus, we may conclude that normal send and receive operations use system buffer, i.e., the send statements sends its message to system buffer which is then retrieved by the receive statement.

1.3.6 Refer to Appendix for the Parallel code of this question.

Bandwidth and message latency on the HPC Cluster

2. 1 **Introduction**

Program to estimate the message latency t_s and reciprocal of the bandwidth t_c for the SMA Hydra Cluster

2. 2 **Method of Approach**

To estimate the message latency, we may repeatedly send fixed-size messages from one process to another. When the receiving process receives the message, it would reply back to the sending process. This process is then repeated with messages of different sizes. Then Round Trip Timing is calculated several times for each message size using MPI_Wtime(). (Measuring RTT ensures that the two processors send and receive correct values) After dividing the average value for one Round Trip by two, a least-squares-fit line is constructed (message size, time). The intercept on the time vs message size curve gives an estimate of the start-up cost t_s while the gradient of the slope gives an estimate for t_c.

2.3 **Results and Discussion**

Output of the Program:

n=100
Time for the shuttling process between two processors is 0.011245 for size=100 for the variable float

n=200
Time for the shuttling process between two processors is 0.012146 for size=200 for the variable float

n=300
Time for the shuttling process between two processors is 0.009896 for size=300 for the variable float

n=400
Time for the shuttling process between two processors is 0.011306 for size=400 for the variable float

n=500
Time for the shuttling process between two processors is 0.013036 for size=500 for the variable float
.
.
.

n=9700
Time for the shuttling process between two processors is 0.273745 for size=9700 for the variable float

n=9800
Time for the shuttling process between two processors is 0.277254 for size=9800 for the variable float

n=9900
Time for the shuttling process between two processors is 0.278765 for size=9900 for the variable float

n=10000
Time for the shuttling process between two processors is 0.280973 for size=10000 for the variable float

To analyze the message latency and bandwidth, we start with sending message size with 100 bytes, increase by a step size of 100 bytes until the message size reaches 10000 bytes. Thus, we get a total of 100 different message sizes, large enough to construct a least-square-fit line to the resulting (message size, time) pair.

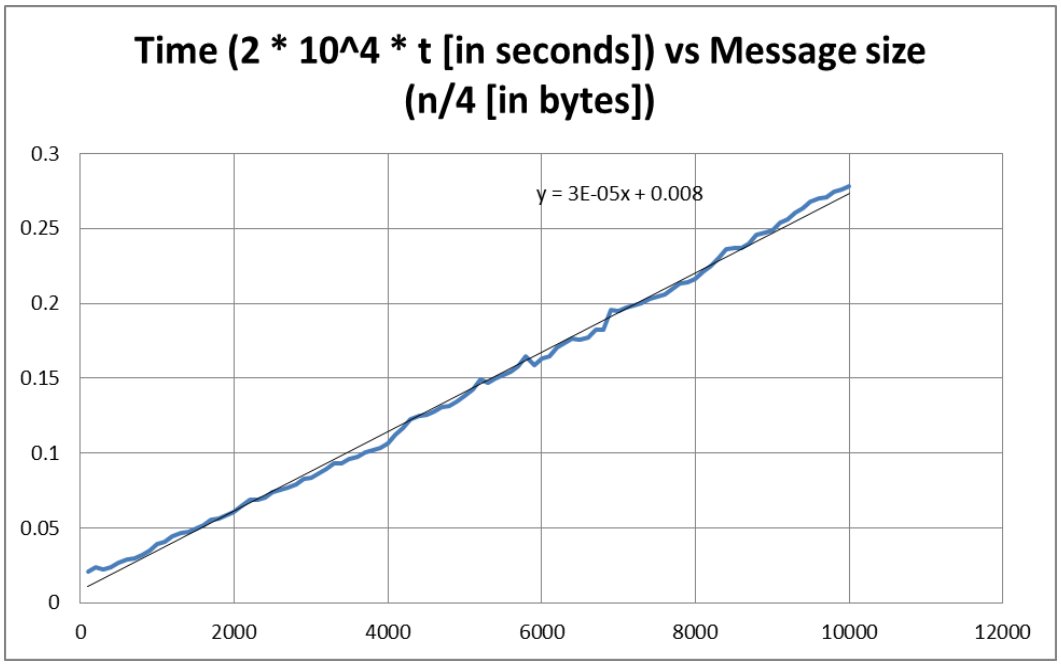


Figure 2: Time (y-axis) (2*10^4*t) [in seconds] vs Message Size (x-axis) (n/4) [in bytes]

From the graph, we see that y-axis represents time (in 20,000 t units [because the time we calculated was for 10,000 Round Trip iterations (2*10,000) of the same data between two processors]).
The x-axis represents message size (number of float variables sent, 4 bytes each).

Now, the y-intercept = 0.008 / ($2*10^4$) = $4 * 10^{-7}$,
where t_s is the time taken to initiate send/receive command.

Hence, slope, t_c = ($3*10^{-5}$) / ($2 * 10^4 * 4$) = $0.375 * 10^{-9}$,
where t_c is the time taken to transfer a single unit of data.

Thus, **Bandwidth** = $1/t_c$ = $2.7 * 10^9$ Hz and **Message Latency** = t_s = $4 * 10^{-7}$ seconds

2.3.1 This method of calculating "Message Latency" does not take into account the distance between the sender and receiver. The Latency takes into account only the time taken to initiate send/receive command and is the minimum time needed by a processor irrespective of the size of the processor

2.3.2 From the graph, we may notice that Round Trip Time (for multiple iterations) increases linearly with increase in message size. But, when message size is 0, there is a small time intercept t_s, which represents Message Latency, i.e., minimum time before which a message can't be transmitted (whatever the size of message be). Thus, t_s must be small in order to have better performance, i.e., Start-up phase overheads which include processes like Message Assembly, Message Tagging and Message Interpretation must be minimal in order that the Cost of Communication is minimised.

2.3.3 Refer to Appendix for the Parallel code of this question.

Parallel integration code

3.1 (a) **Introduction**

A parallel Code to run the parallel integration Program on different number of processes. To realize difference between running a sequential code and a parallelized code using one processor.

3.2 (a) **Method of Approach**

To realize the importance of parallel code, this program is run on multiple cores, number pf cores varying from 1 to 24. The parallel code integral.c contains parallelized code that performs integration and MPE code that performs customized logging.

The serial code, ori_integral.c is then run on a single processor.

Then we analyze the time elapsed in both the cases and make two comparisons:

1. time elapsed in serial code vs time elapsed in parallel code when only 1 processor is used
2. time elapsed for parallel code when multiple cores are used

This program uses Trapezoidal rule to estimate the value of the integral.

Trapezoidal Rule:

We partition the interval [a,b] into 'n' equal subintervals of width h, where h = (b-a)/n and $a = x_0 < x_1 < \ldots < x_{i-1} < x_i < \ldots < x_n = b$. Now, each trapezoid consists of a big rectangle and a small triangle. Thus area of trapezoid is calculated by summing up the area of the rectangle and the triangle. The value of the integral is estimated to be the sum of the areas of such trapezoids.

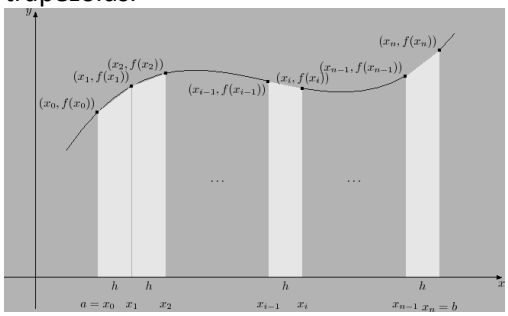


Figure 3: A fig. depicting integral calculation using trapezoidal rule

3.2 (a) **Results**

```
Output of the Serial Code:
With n = 10000000 trapezoids,
our estimate of the integral from 0.00 to 30.00 = 5.792740e-01.
CPU time elapsed: 0.000000 seconds
```

Thus, for the serial code, for n = 1,00,00,000 trapezoids, time taken for calculating integral from a = 0.000000 to b = 30.000000 is 0.000000 seconds. (Refer Appendix for serial code)

```
Output for the parallel code using n=1
With n = 10000000 trapezoids,
our estimate of the integral from 0.000000 to 30.000000 = 5.792740e-01
Time taken for whole computation = 0.498109 seconds

Output for the parallel code using n=2
With n = 10000000 trapezoids,
our estimate of the integral from 0.000000 to 30.000000 = 5.792740e-01
Time taken for whole computation = 0.260333 seconds

Output for the parallel code using n=4
With n = 10000000 trapezoids,
```

```
our estimate of the integral from 0.000000 to 30.000000 = 5.792740e-01
Time taken for whole computation = 0.129541 seconds
```

Output for the parallel code using n=8

```
With n = 10000000 trapezoids,
our estimate of the integral from 0.000000 to 30.000000 = 5.792740e-01
Time taken for whole computation = 0.069916 seconds
```

Output for the parallel code using n=16

```
With n = 10000000 trapezoids,
our estimate of the integral from 0.000000 to 30.000000 = 5.792740e-01
Time taken for whole computation = 0.050110 seconds
```

Thus, for n = 1,00,00,000 trapezoids, the output for various number of processors for calculating integral between a = 0.000000 to b = 30.000000 is

Number of processors	Time Taken for Computation (in seconds)
1	0.498109
2	0.260333
3	0.181201
4	0.129541
8	0.069916
16	0.050110
24	0.061948

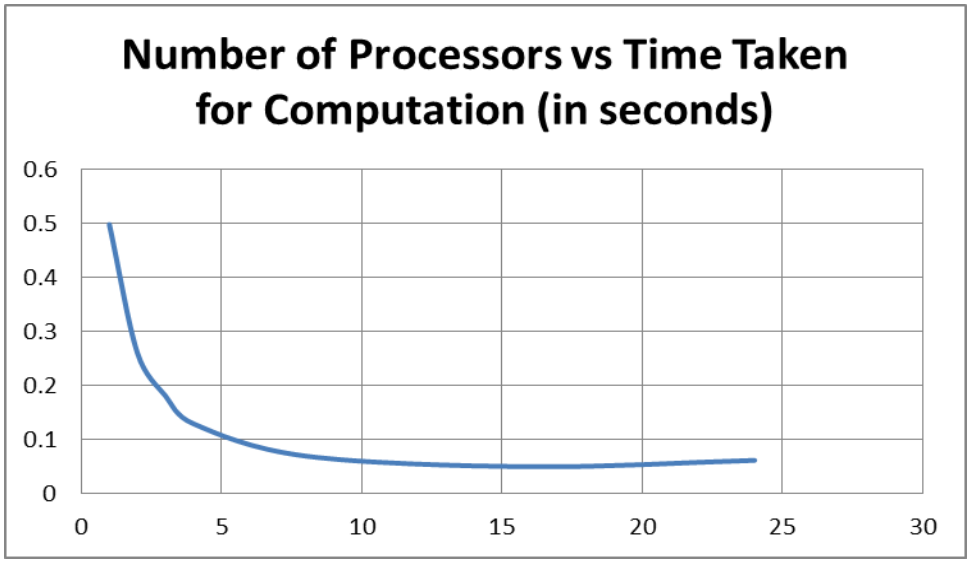


Figure 4: A graph between number of Processors (x-axis) vs Time taken for Computation (y-axis)

Here, we see that the parallel code run on one processor takes more time than a serial code running on one processor. It is because the 3 key factors that determinate the performances of the parallel model are:

- Parallel task granularity - parallel tasks must be "big" enough to justify the overheads of parallelization.
- Communication overhead – can be reduced by minimising the amount of communication and synchronization between parallel tasks ("race conditions"), using redundant computation, asynchronous communications, collective communications and faster communication hardware

- Load balancing among processes - Each process should take approximately the same time to finish their work.

For n=1 in parallel code,

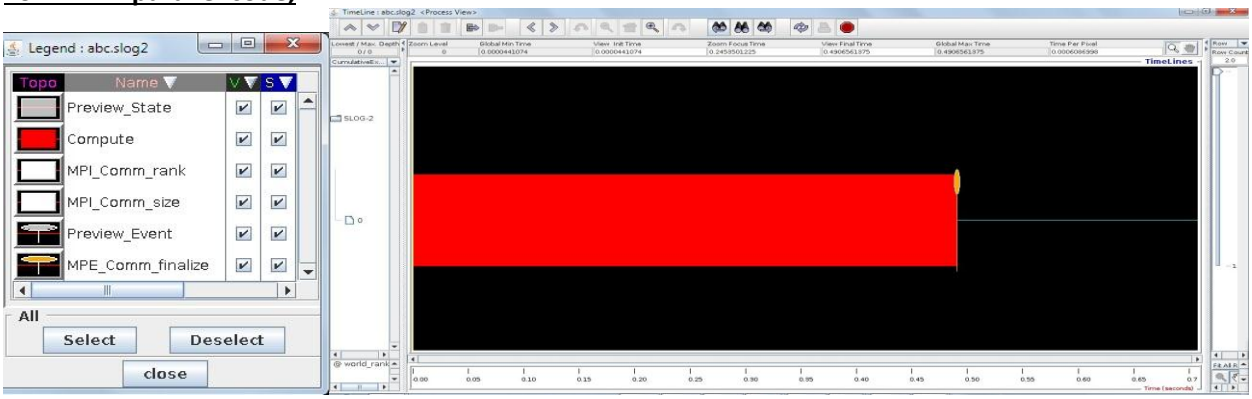


Figure 5: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

For n=2 in parallel code,

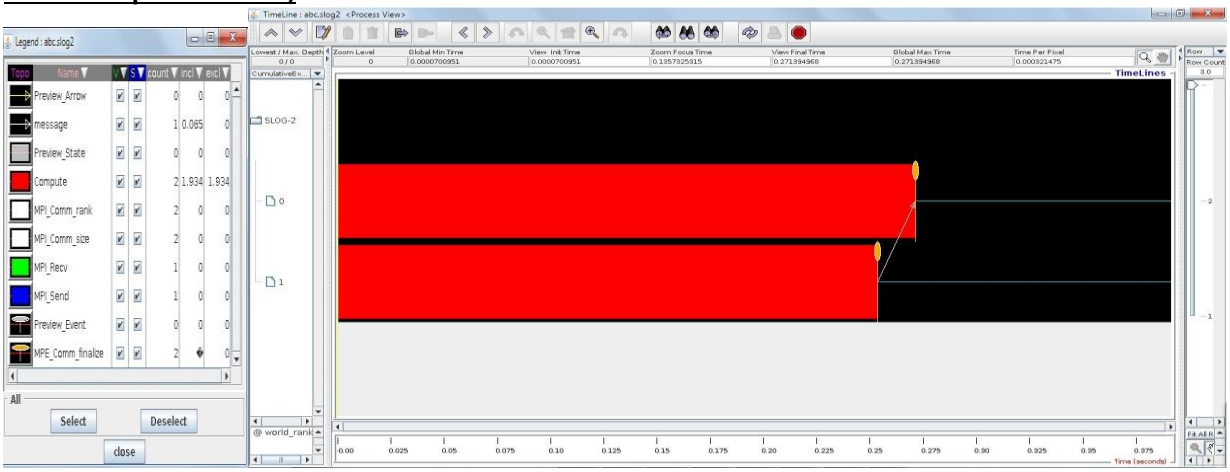


Figure 6: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

For n=4 in parallel code,

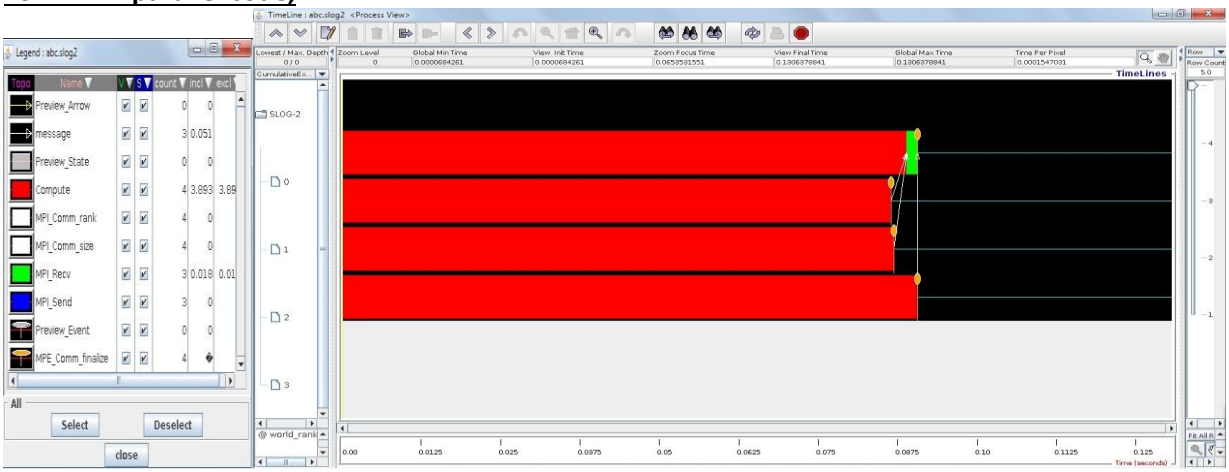


Figure 7: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

For n=8 in parallel code,

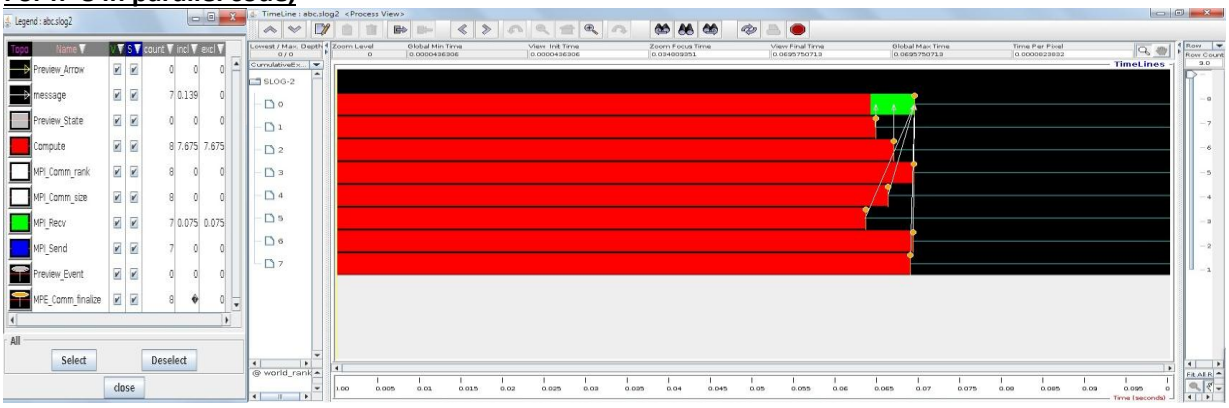


Figure 8: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

For n=16 in parallel code,

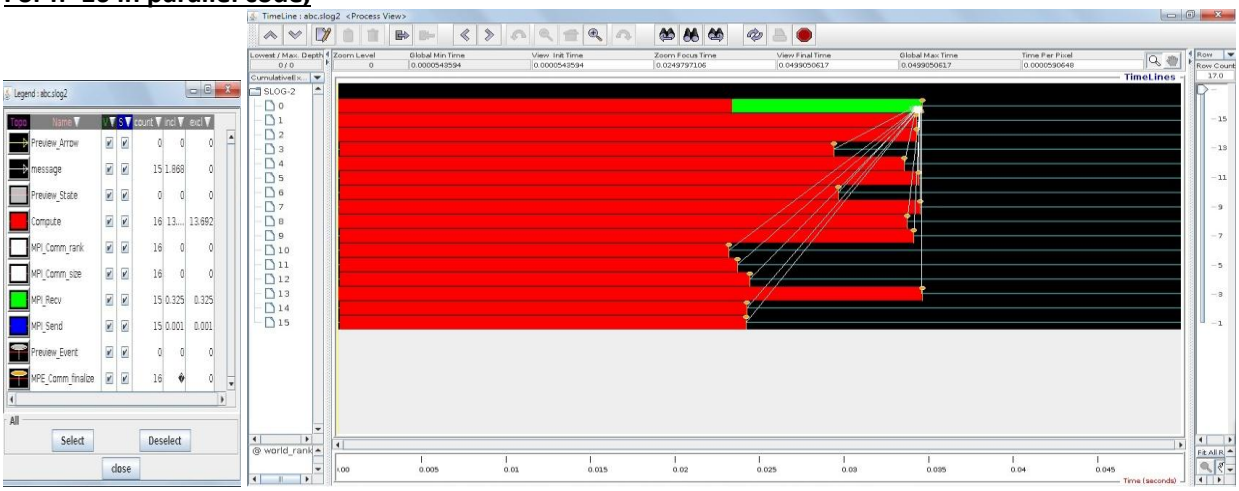


Figure 9: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

For n=24 in parallel code,

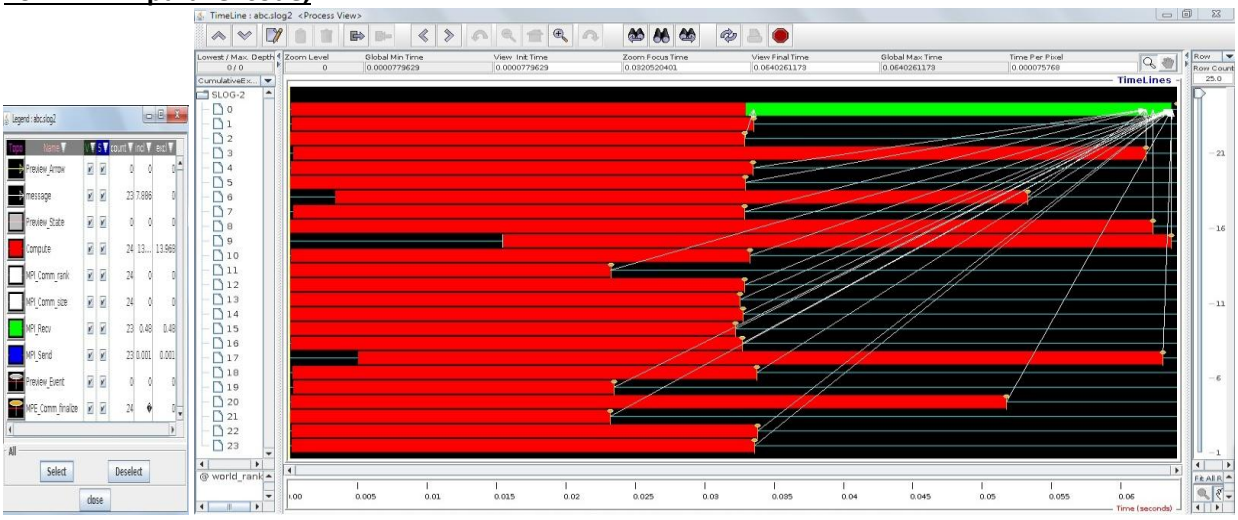


Figure 10: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

3.2.1 (a) The functionality of command mpicc is to give/share all the variables to the specified number of processors. This can be seen from the fact that the statements which are not supposed to be parallelized in a parallel code, i.e., statements which are before MPI_Init() occur as many times as the number of processors specified, if printed. This suggests that mpicc compilation results in distribution of all variables/functions in the program to the specified number of processors (-np 2, –np 4, etc.).

3.2.2 (a) It is seen that there is a gradual decrease in execution time as number of processors increase which is natural because we decrease the workload on individual processors by increasing their number (Graph, Figure 4). But when n=24, i.e., number of processors is maximum, the time elapsed increases quite a bit when compared with time elapsed when n=16. It is because we are giving very small values for computation. So computation time does not matter as it has been found in Problem 2 that bandwidth for HPC Cluster is $2.67*10^9$. So computations involving values less than numbers in the range of 10^9, do not increase/decrease computation time significantly. Instead, giving more resources (number of processors) for such small values increases communication overheads resulting in an increase in total time elapsed when no. of processors = 24 compared to when number of processors = 16.

3.2.3 (a) The jumpshot (Java based visualization tool) pictures shown in Figure 5-10 : (a) and (b) present a graphical view (through a timeline) of the space-time status of the parallel computation involved when different number of processors are used to evaluate the parallel code.

3.2.4(a) Refer to Appendix for the Parallel Code of this Question

Parallel integration code

3.1 (b) **Introduction**

Proper implementation of Simpson's rule using message passing functions to estimate the value of integral of a function and comparing the output with that of Trapezoidal rule

3.2 (b) **Method of Approach**

In order to test if Simpson's rule can serve as a more accurate alternative to the Trapezoidal rule, we make minor changes in the parallel code for the Trapezoidal rule, every even term inside the square brackets [shown below] must be multiplied by 4, while all the odd terms except the 1st and the last term are to be multiplied by 2.

Simpson's Rule:

Simpson's rule is a method of numerical integration that provides an approximation of a definite integral over the interval [*a*,*b*] using parabolas. Generally, the function *f*(*x*) over interval [*a*,*b*] can be approximated as

$$\int_a^b f(x)dx \approx \frac{b-a}{6}[f(a)+4f(\frac{a+b}{2})+f(b)]$$

.

This form works well when the function is "smooth" over [*a*, *b*]; that is, if the function doesn't oscillate much.

If the function is not smooth (which is the more common situation), the interval can be broken into subintervals, and Simpson's rule applied. The integral of a function $f(x)$ over the interval $[a,b]$ with subintervals $a = x_0 < x_1 < \ldots < x_{i-1} < x_i < \ldots < x_n = b$ and subinterval length $h = (b-a)/n$ (n must be even) can be approximated as

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + \ldots 4f(x_{n-1}) + f(x_n)]$$

.

3.3 (b) **Results and Discussions**

```
Output of the Program

Value of integration by Simpson's rule is          : 0.5792739367
Number of processes is                             : 4
Time taken for calculation                         : 0.0000469685
```

3.3.1 (b) It must be made sure that the ratio n/p is even in order that the Simpson's rule works properly. (So, n must be divisible by p and also, the quotient must be even.)

3.3.2 (b) We see that output for the parallel code using trapezoidal method was:

```
Output for the parallel code using n=4 in Trapezoidal Rule
With n = 10000000 trapezoids,
our estimate of the integral from 0.000000 to 30.000000 = 5.792740e-01
Time taken for whole computation = 0.129541 seconds
```

- We see that Simpson's rule produces more precise and accurate values when compared to Trapezoidal rule. Output when integral is estimated using trapezoidal rule was 0.5792740, whereas output when integral is estimated using Simpson's rule is 0.5792739367. (though we may get more precision by using %11.10f etc.)
- Also time elapsed in executing Simpson's rule is quite less when compared to that elapsed using Trapezoidal rule.
  Time elapsed when estimation done using Trapezoidal rule = 0.129541 sec
  Time elapsed when estimation done using Simpson's rule = 0.0000469685 sec

Thus, it can be deduced that Simpson's rule is more efficient than trapezoidal rule.
Refer Appendix for the parallel code of this question

Parallelising sequential code for calculating the value of π using Monte Carlo Method

4.1 **Introduction**

Parallelization of the sequential code to evaluate the value of π by using Monte Carlo Method by using (a) Only MPI_Send() and MPI_Recv() (Point to point operations)
(b) Collective operations

To analyse the execution time (communication time and computation time) by using different values for the number of processors (both, even and odd values) and compare the value of π obtained from both the parallel codes and from the serial code.

To calculate the speedups and efficiencies for both parallel codes with plotting the curves of

(i) Predicted and Actual Speedup

(ii) Predicted and Actual Efficiency of both parallel versions.

4.2 **Method of Approach**

In order that the code to evaluate the value of π (Monte Carlo Method) can be parallelized, we need to firstly decide the number of processors we would be using to parallelize. Then the total number of darts thrown n must be divisible by p such that the result is even, (n/p is even) in order to have proper load balancing. Then the value of pi is calculated for each processor by throwing the specified number of darts, and results are analysed using point-to-point operations. Then the same results are analysed using Collective operations.

4.3 **Results and Discussions**

Output of the Parallel Code using only MPI_Send() and MPI_Recv()

```
 Known value of  PI                                  : 3.14159265358979311600e+00
 Calculated value of pi is                           : 3.12480000000000002203e+00
 Number of processes is                              : 4
 No. of iterations is                                : 1250
 Time for the computation process is                 : 0.000124
 Percentage Error                                    : 0.5345267653
```

Output of the Parallel Code using collective operations

```
Known value of  PI                                  : 3.14159265358979311600e+00
Calculated value of pi is                           : 3.14800000000000013145e+00
Number of processes is                              : 4
No. of iterations is                                : 1250
Time for the computation process is                 : 0.000165
Percentage Error                                    : 0.2039521707
```

Output of the Serial Code for estimating value of π

```
Please enter the no. of iterations to be used to estimate pi: 10000
Total Number of trials= 10000, approx value of pi is : 3.13680000000000003268e+00
Known value of  PI                                    : 3.14159265358979311600e+00
CPU time elapsed                                      : 0.000000 seconds
Percentage Error                                      : 0.1525549019
```

Refer to Appendix for the Serial Code

Actual Speed up, $S_a = T_1/T_n$,

where T_1 is the time taken on a single processor,

T_n is the time taken on 'n' processors.

Actual Efficiency, $e_a = S_a/n$,

where S_a is Actual Speed up,

n is the total number of processes on which program is run.

For case of predicted speedup, the value of a is found to be $S_P = 1/((1-a)+(a/P)+(T_D/T_1))$

Speedup calculation for Point-to-Point Operations

For n = 2, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.154399	0.407670	0.000342	0.000017	0.000325	0.000642	1.8771	93.86
100000	3.144639	0.097000	0.003111	0.000014	0.003297	0.005361	1.7232	86.16
1000000	3.141512	0.002567	0.028246	0.000019	0.028227	0.049851	1.7649	88.24
10000000	3.141869	0.008828	0.202933	0.000015	0.202918	0.375675	1.8512	92.56
100000000	3.141663	0.002245	1.899689	0.000025	1.899676	3.682445	1.9384	96.92

For n = 2, predicted speedup -
a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)
Hence (1-a) = 0 can be neglected.

No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000325	0.000017	19.1176	0.000642	1.8994	94.97
100000	0.003297	0.000014	235.5	0.005361	1.9896	99.48
1000000	0.028227	0.000019	1485.6	0.049851	1.9985	99.92
10000000	0.202918	0.000015	13527.9	0.375675	1.9998	99.99
100000000	1.899676	0.000025	75987.0	3.682445	1.9999	99.99

For n=4, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.131200	0.330804	0.000202	0.000033	0.000169	0.000642	3.7182	92.96
100000	3.144400	0.089360	0.001674	0.000049	0.001634	0.005361	3.2025	80.06
1000000	3.146263	0.148693	0.016442	0.000296	0.016397	0.049851	3.0319	75.79
10000000	3.141700	0.003416	0.100500	0.002727	0.100455	0.375675	3.7380	93.45
100000000	3.141621	0.000904	0.950990	0.005202	0.950932	3.682445	3.8722	96.81

For n = 4, predicted speedup -
a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)
Hence (1-a) = 0 can be neglected.

No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000169	0.000033	5.1212	0.000642	3.3178	82.95
100000	0.001634	0.000049	33.3469	0.005361	3.5716	89.29
1000000	0.016397	0.000296	55.3952	0.049851	3.9072	97.68
10000000	0.100455	0.002727	36.8372	0.375675	3.8871	97.18
100000000	0.950932	0.005202	182.8012	3.682445	3.9775	99.43

For n=8, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.137199	0.139822	0.000140	0.000053	0.000087	0.000642	4.5857	57.32
100000	3.144359	0.088087	0.000838	0.000105	0.000733	0.005361	6.3974	79.97
1000000	3.142392	0.025444	0.011004	0.000684	0.010320	0.049851	4.5303	56.63
10000000	3.141461	0.004159	0.055405	0.001025	0.054380	0.375675	6.7805	84.76
100000000	3.141396	0.006251	0.539320	0.003534	0.535786	3.682445	6.8279	85.35

For n = 8, predicted speedup -
a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)
Hence (1-a) = 0 can be neglected.

No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000087	0.000053	1.6415	0.000642	4.8180	60.22
100000	0.000733	0.000105	6.9809	0.005361	6.9163	86.45
1000000	0.010320	0.000684	15.0877	0.049851	7.2087	90.11
10000000	0.054380	0.001025	53.0536	0.375675	7.8291	97.86
100000000	0.535786	0.003534	151.6089	3.682445	7.9390	99.24

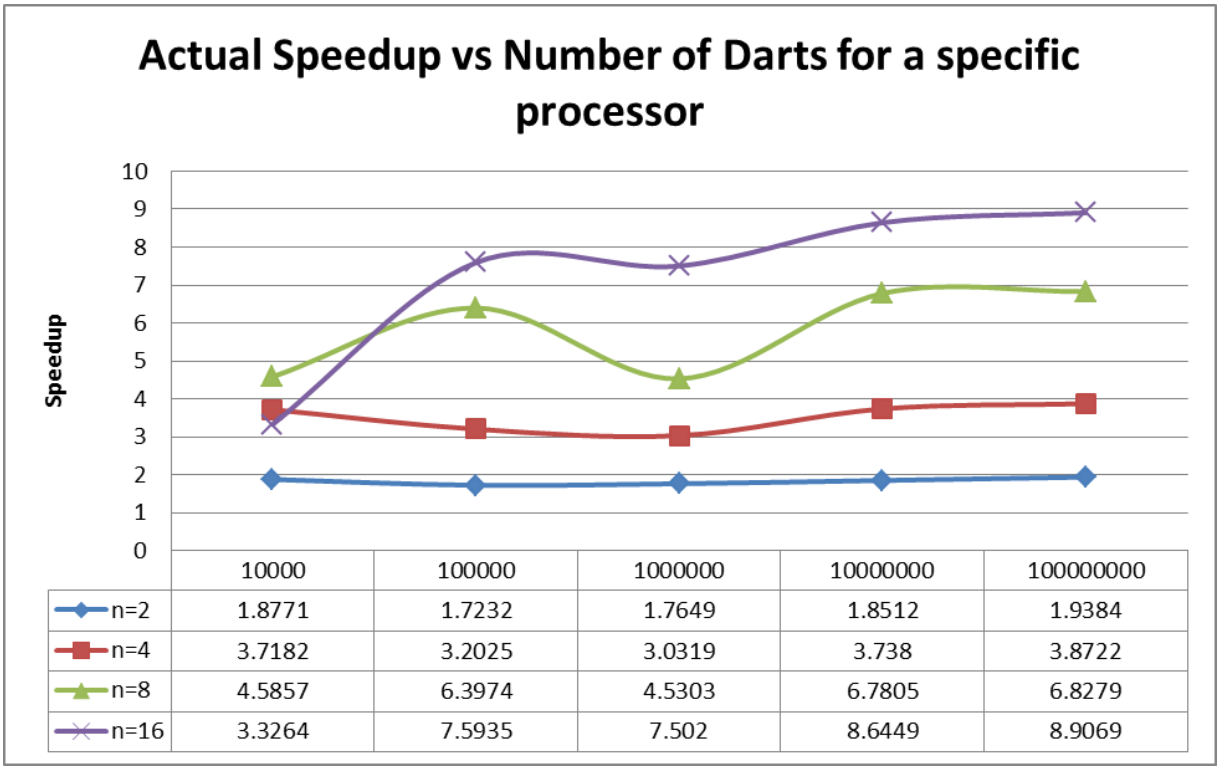
For n=16, actual speedup

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.144800	0.102093	0.000193	0.000144	0.000049	0.000642	3.3264	20.79
100000	3.147280	0.181034	0.000706	0.000220	0.000486	0.005361	7.5935	47.46
1000000	3.142768	0.037412	0.006645	0.002572	0.004073	0.049851	7.5020	46.89
10000000	3.141519	0.002351	0.043436	0.015820	0.027616	0.375675	8.6449	54.03
100000000	3.141380	0.006762	0.405073	0.151821	0.253252	3.682445	8.9069	55.67

For n = 16, predicted speedup -
a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)
Hence (1-a) = 0 can be neglected.

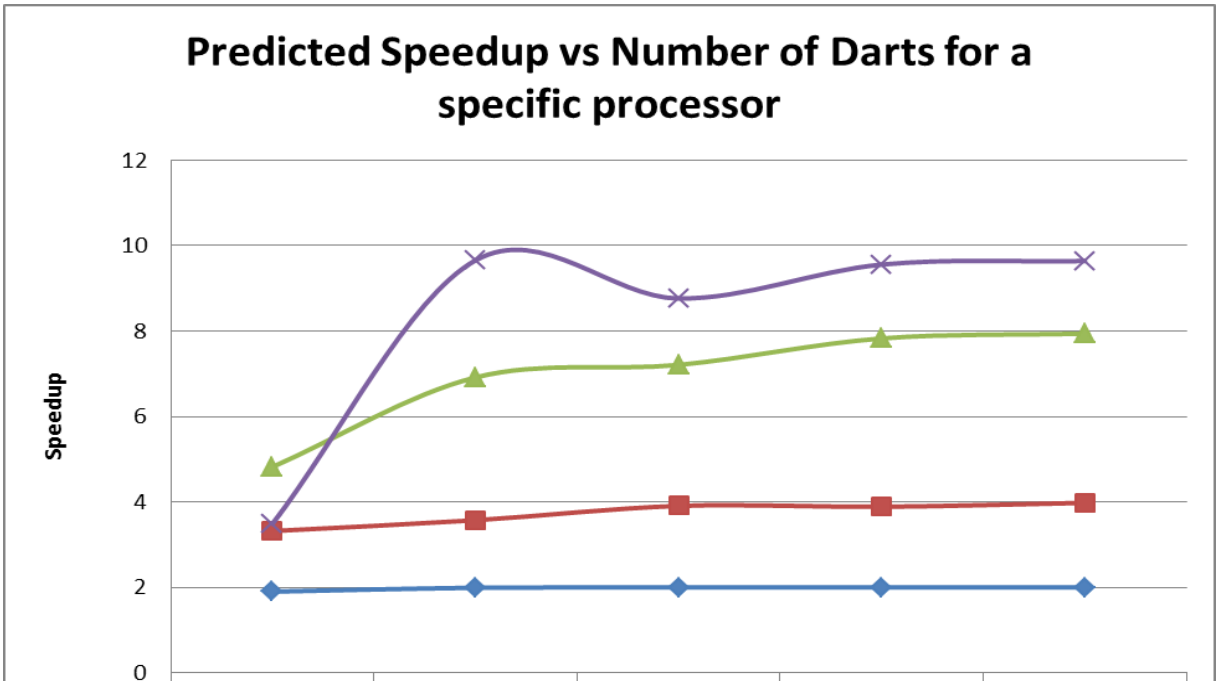
No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000049	0.000144	0.3403	0.000642	3.4868	21.79
100000	0.000486	0.000220	2.2091	0.005361	9.6584	60.36
1000000	0.004073	0.002572	1.5836	0.049851	8.7647	54.78
10000000	0.027616	0.015820	1.7456	0.375675	9.5592	59.74
100000000	0.253252	0.151821	1.6681	3.682445	9.6406	60.25

where,
Total time, t is in seconds,
Communication time, t1 is in seconds,
Calculation time, t2 = t - t1 is in seconds,



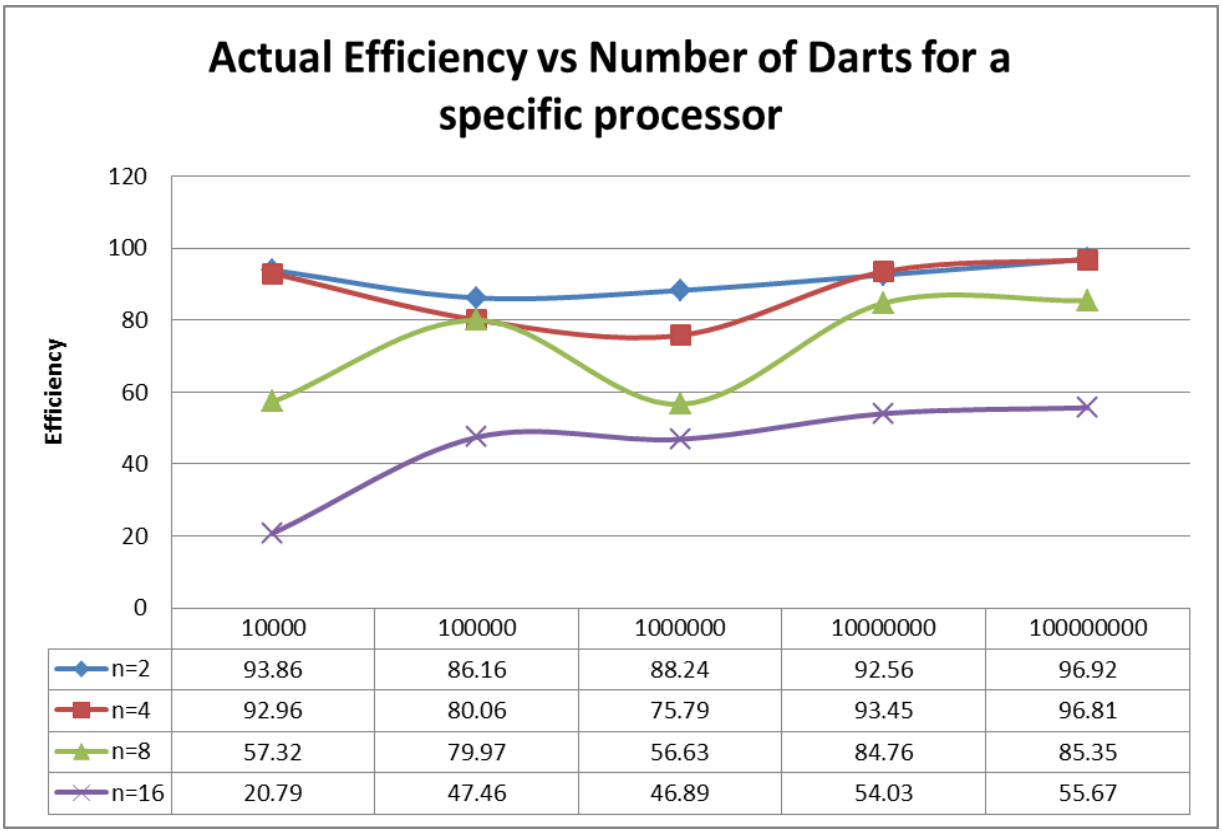
	10000	100000	1000000	10000000	100000000
n=2	1.8771	1.7232	1.7649	1.8512	1.9384
n=4	3.7182	3.2025	3.0319	3.738	3.8722
n=8	4.5857	6.3974	4.5303	6.7805	6.8279
n=16	3.3264	7.5935	7.502	8.6449	8.9069

Figure 11: Actual speedup vs Number of Darts (for a particular processor)



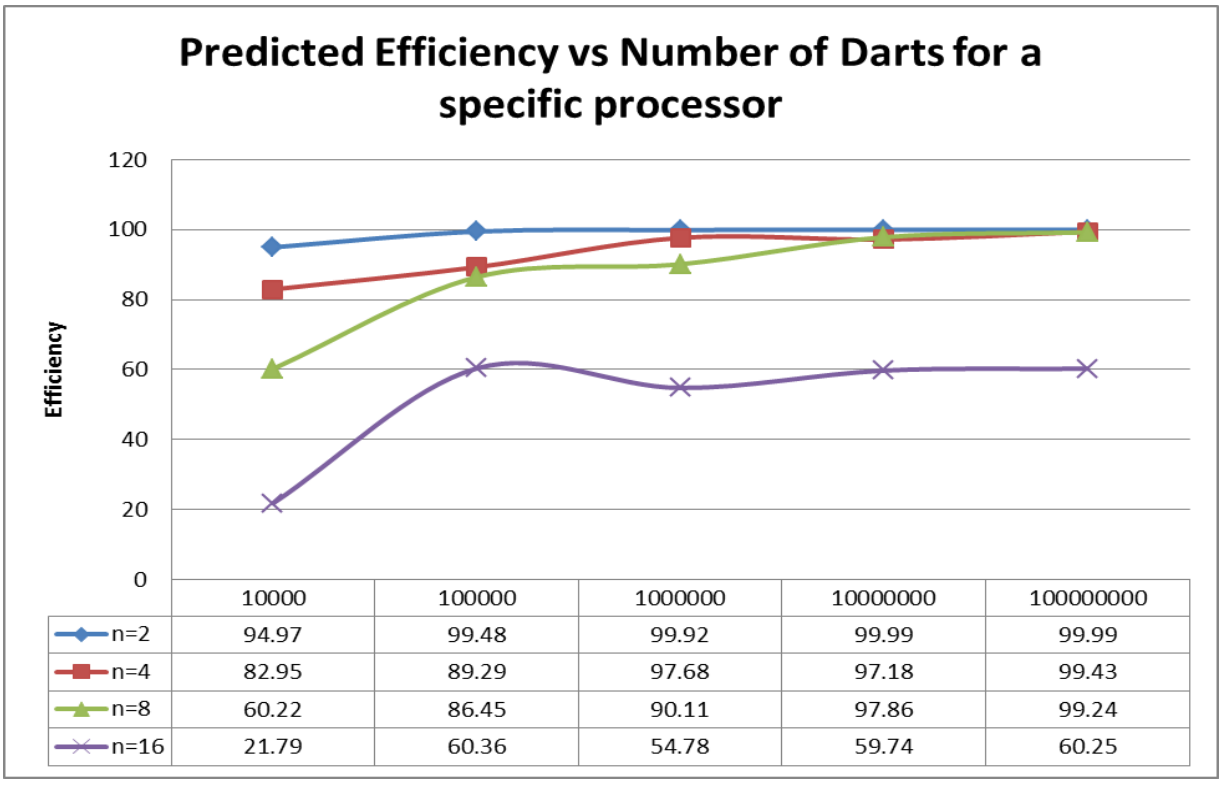
	10000	100000	1000000	10000000	100000000
n=2	1.8994	1.9896	1.9985	1.9998	1.9999
n=4	3.3178	3.5716	3.9072	3.8871	3.9775
n=8	4.818	6.9163	7.2087	7.8291	7.939
n=16	3.4868	9.6584	8.7647	9.5592	9.6406

Figure 12: Predicted speedup vs Number of Darts (for a particular processor)



	10000	100000	1000000	10000000	100000000
n=2	93.86	86.16	88.24	92.56	96.92
n=4	92.96	80.06	75.79	93.45	96.81
n=8	57.32	79.97	56.63	84.76	85.35
n=16	20.79	47.46	46.89	54.03	55.67

Figure 13: Actual Efficiency vs Number of Darts (for a particular processor)



	10000	100000	1000000	10000000	100000000
n=2	94.97	99.48	99.92	99.99	99.99
n=4	82.95	89.29	97.68	97.18	99.43
n=8	60.22	86.45	90.11	97.86	99.24
n=16	21.79	60.36	54.78	59.74	60.25

Figure 14: Predicted Efficiency vs Number of Darts (for a particular processor)

Speedup calculation for Collective Operations

For n=2, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.153999	0.394938	0.000366	0.000030	0.000336	0.000670	1.8306	91.53
100000	3.140600	0.031597	0.003351	0.000036	0.003315	0.005548	1.6556	82.78
1000000	3.140328	0.040255	0.030705	0.000019	0.039686	0.048781	1.5887	79.44
10000000	3.141871	0.008866	0.199382	0.000021	0.199361	0.378867	1.9002	95.01
100000000	3.141505	0.002784	1.860058	0.000025	1.860033	3.684818	1.9810	99.05

For n = 2, predicted speedup -

a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)

Hence (1-a) = 0 can be neglected.

No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000336	0.000030	11.2	0.000670	1.8356	91.78
100000	0.003315	0.000036	92.1	0.005548	1.9744	98.72
1000000	0.039686	0.000019	2088.7	0.048781	1.9984	99.92
10000000	0.199361	0.000021	9493.4	0.378867	1.9997	99.98
100000000	1.860033	0.000025	74401.3	3.684818	1.9999	99.99

For n=4, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.147200	0.178487	0.000192	0.000026	0.000166	0.000670	3.4896	87.24
100000	3.138920	0.085073	0.001609	0.000047	0.002562	0.005548	3.3645	84.11
1000000	3.140784	0.025740	0.016091	0.000033	0.016058	0.048781	3.0316	75.79
10000000	3.141069	0.016662	0.107394	0.000024	0.107370	0.378867	3.5278	88.19
100000000	3.141685	0.002932	1.004446	0.000038	1.004408	3.684818	3.6320	90.80

For n = 4, predicted speedup -

a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)

Hence (1-a) = 0 can be neglected.

No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000166	0.000026	6.4	0.000670	3.4625	86.56
100000	0.002562	0.000047	54.51	0.005548	3.8689	96.72
1000000	0.016058	0.000033	486.6	0.048781	3.9892	99.73
10000000	0.107370	0.000024	4473.7	0.378867	3.9989	99.97
100000000	1.004408	0.000038	26431.7	3.684818	3.9998	99.99

For n=8, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.130399	0.356273	0.000121	0.000037	0.000084	0.000670	5.5372	69.22
100000	3.144479	0.091907	0.001308	0.000028	0.001280	0.005548	4.2416	53.02
1000000	3.140952	0.020392	0.012829	0.000039	0.012790	0.048781	3.8024	47.53
10000000	3.141486	0.003395	0.081433	0.000020	0.081413	0.378867	4.6525	58.16
100000000	3.141497	0.003031	0.517279	0.000040	0.516879	3.684818	7.1235	89.04

For n = 8, predicted speedup -
a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)
Hence (1-a) = 0 can be neglected.

No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000084	0.000037	2.3	0.000670	5.5486	69.36
100000	0.001280	0.000028	45.7	0.005548	7.6895	96.12
1000000	0.012790	0.000039	327.9	0.048781	7.9492	99.36
10000000	0.081413	0.000020	4070.6	0.378867	7.9966	99.96
100000000	0.516879	0.000040	12921.9	3.684818	7.9993	99.99

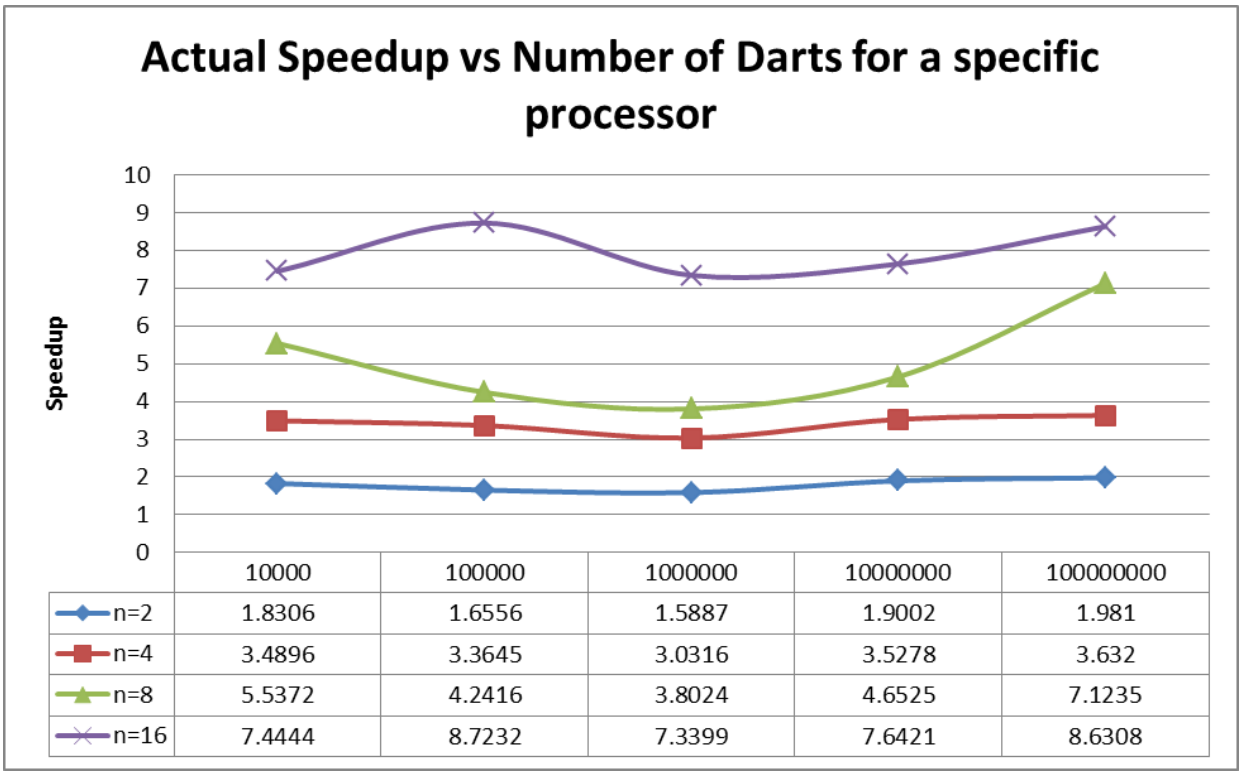
For n=16, actual speedup -

No. of darts	Value of pi	Percentage Error (%)	Total time, t	Comm. time, t1	Calc. time, t-t1	Time when n=1	Speed up	Efficiency (%)
10000	3.148399	0.216684	0.000090	0.000054	0.000126	0.000670	7.4444	46.53
100000	3.140519	0.034143	0.000636	0.000031	0.000605	0.005548	8.7232	54.52
1000000	3.141168	0.013517	0.006646	0.000068	0.006578	0.048781	7.3399	45.87
10000000	3.141294	0.009474	0.049576	0.000120	0.049456	0.378867	7.6421	47.76
100000000	3.141474	0.003765	0.426940	0.000194	0.426746	3.684818	8.6308	53.94

For n = 16, predicted speedup -
a = 0.999999995 == 1 (based on number of operations since most of the program can be parallelized)
Hence (1-a) = 0 can be neglected.

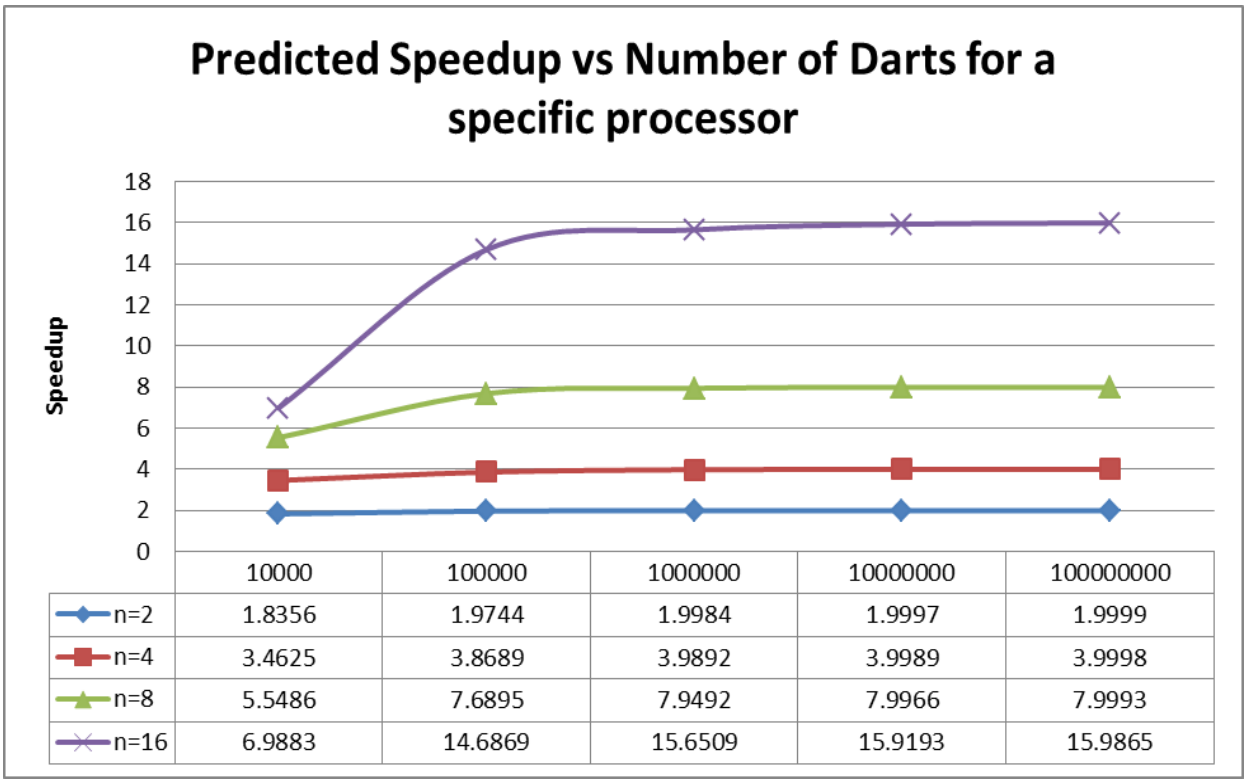
No. of darts	Calc. time, t-t1	Comm. time, t1	(t-t1)/t1	Time when n=1	Predicted Speedup	Predicted Efficiency (%)
10000	0.000126	0.000054	2.33	0.000670	6.9883	43.68
100000	0.000605	0.000031	19.5	0.005548	14.6869	91.79
1000000	0.006578	0.000068	96.73	0.048781	15.6509	97.82
10000000	0.049456	0.000120	412.1	0.378867	15.9193	99.49
100000000	0.426746	0.000194	2199.7	3.684818	15.9865	99.91

where,
Total time, t is in seconds,
Communication time, t1 is in seconds,
Calculation time, t2 = t - t1 is in seconds,



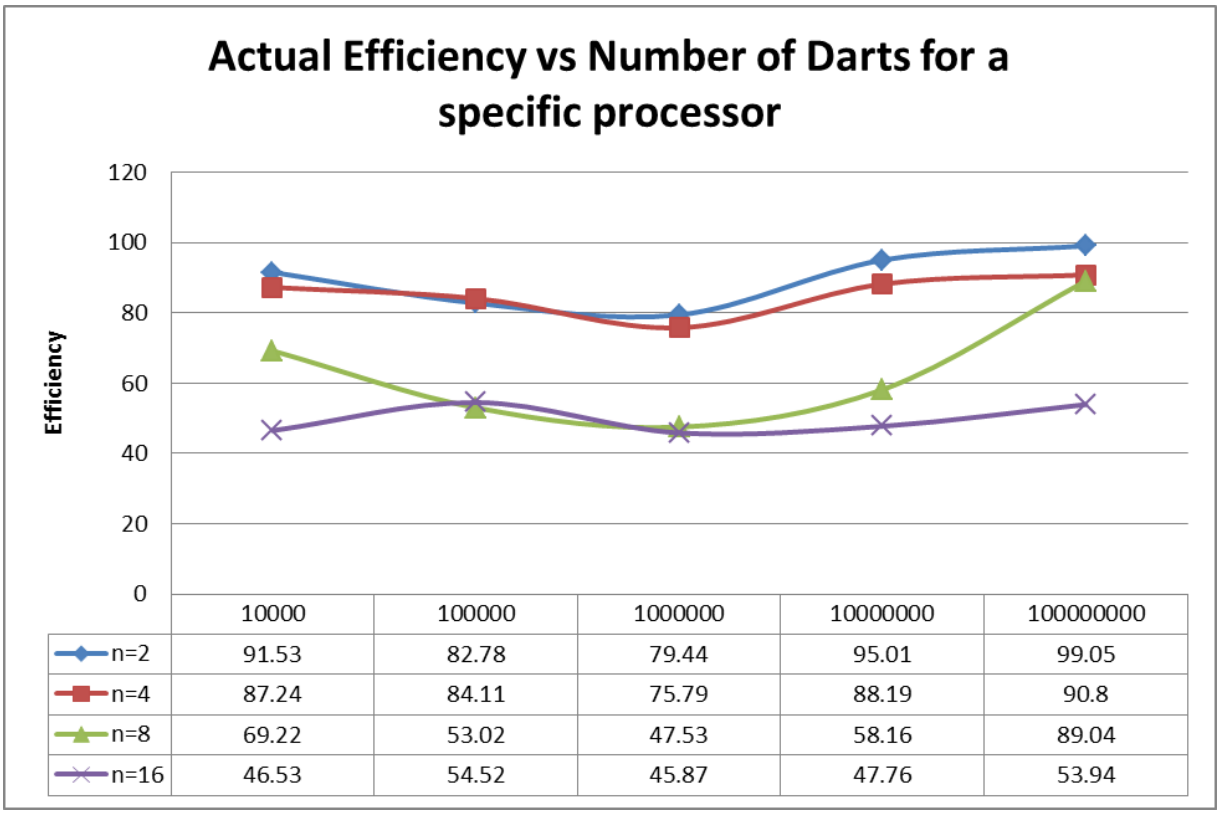
	10000	100000	1000000	10000000	100000000
n=2	1.8306	1.6556	1.5887	1.9002	1.981
n=4	3.4896	3.3645	3.0316	3.5278	3.632
n=8	5.5372	4.2416	3.8024	4.6525	7.1235
n=16	7.4444	8.7232	7.3399	7.6421	8.6308

Figure 15: Actual speedup vs Number of Darts (for a particular processor)



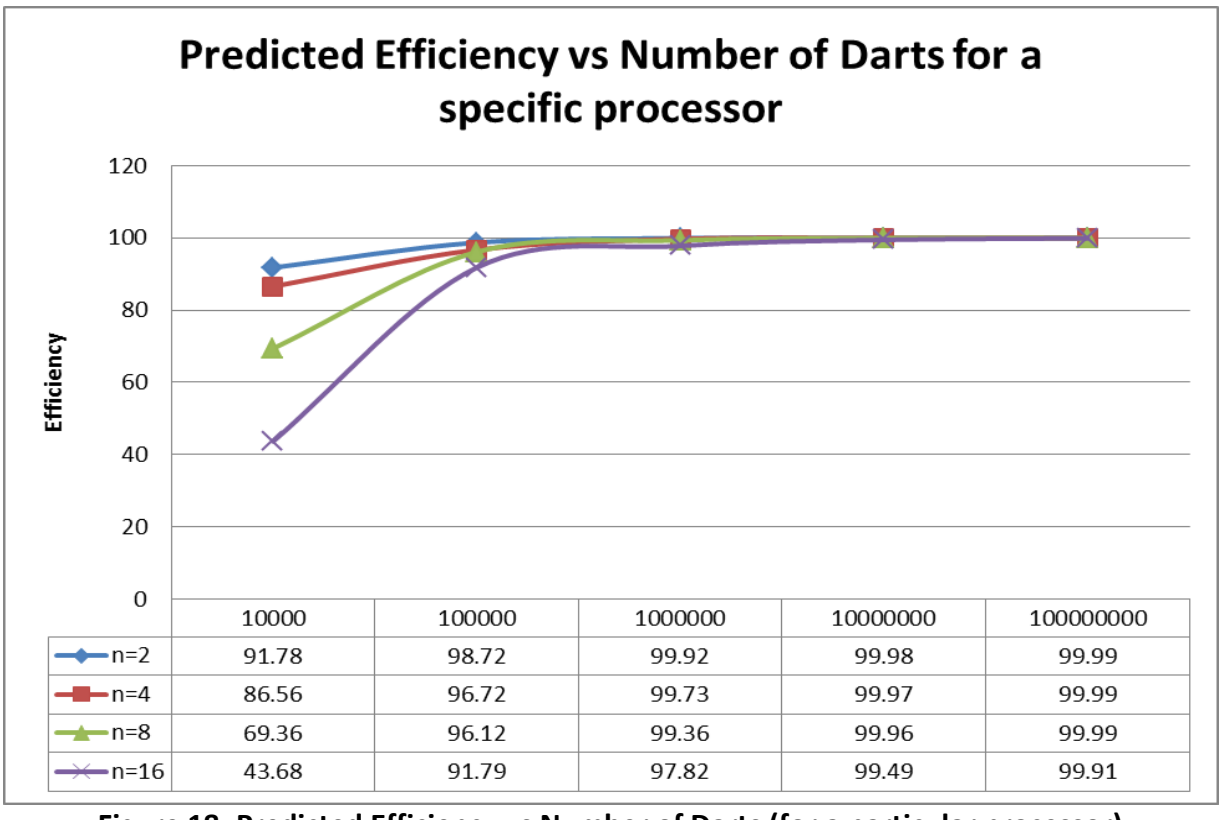
	10000	100000	1000000	10000000	100000000
n=2	1.8356	1.9744	1.9984	1.9997	1.9999
n=4	3.4625	3.8689	3.9892	3.9989	3.9998
n=8	5.5486	7.6895	7.9492	7.9966	7.9993
n=16	6.9883	14.6869	15.6509	15.9193	15.9865

Figure 16: Predicted speedup vs Number of Darts (for a particular processor)



	10000	100000	1000000	10000000	100000000
n=2	91.53	82.78	79.44	95.01	99.05
n=4	87.24	84.11	75.79	88.19	90.8
n=8	69.22	53.02	47.53	58.16	89.04
n=16	46.53	54.52	45.87	47.76	53.94

Figure 17: Actual Efficiency vs Number of Darts (for a particular processor)



	10000	100000	1000000	10000000	100000000
n=2	91.78	98.72	99.92	99.98	99.99
n=4	86.56	96.72	99.73	99.97	99.99
n=8	69.36	96.12	99.36	99.96	99.99
n=16	43.68	91.79	97.82	99.49	99.91

Figure 18: Predicted Efficiency vs Number of Darts (for a particular processor)

n=2 in the Point to Point Operations code

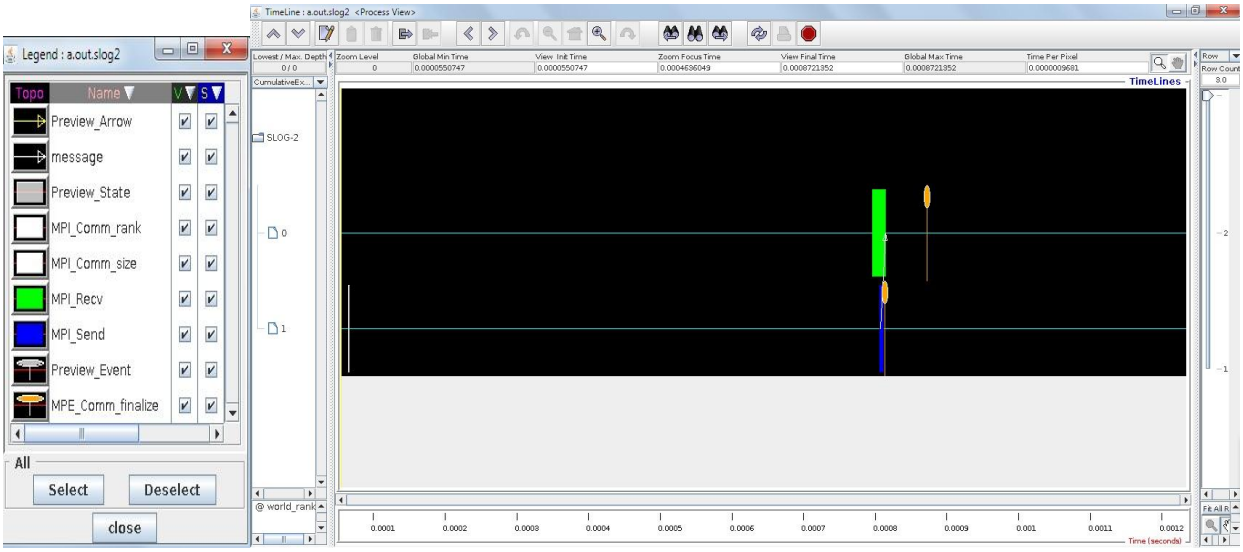


Figure 19: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

n=4 in the Point to Point Operations code

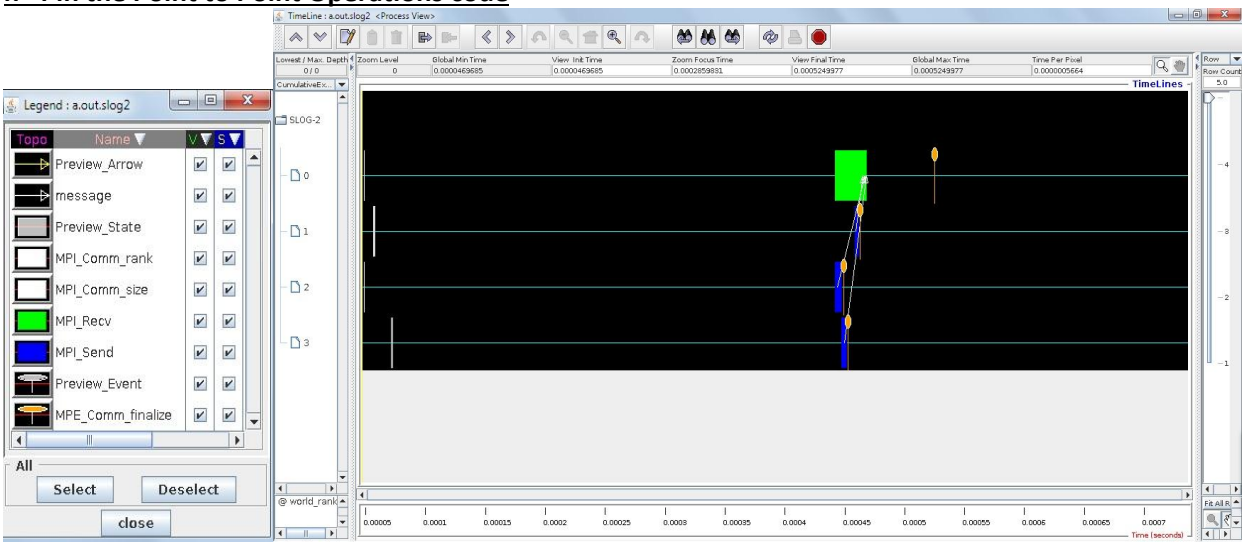


Figure 20: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

n=8 in the Point to Point Operations code

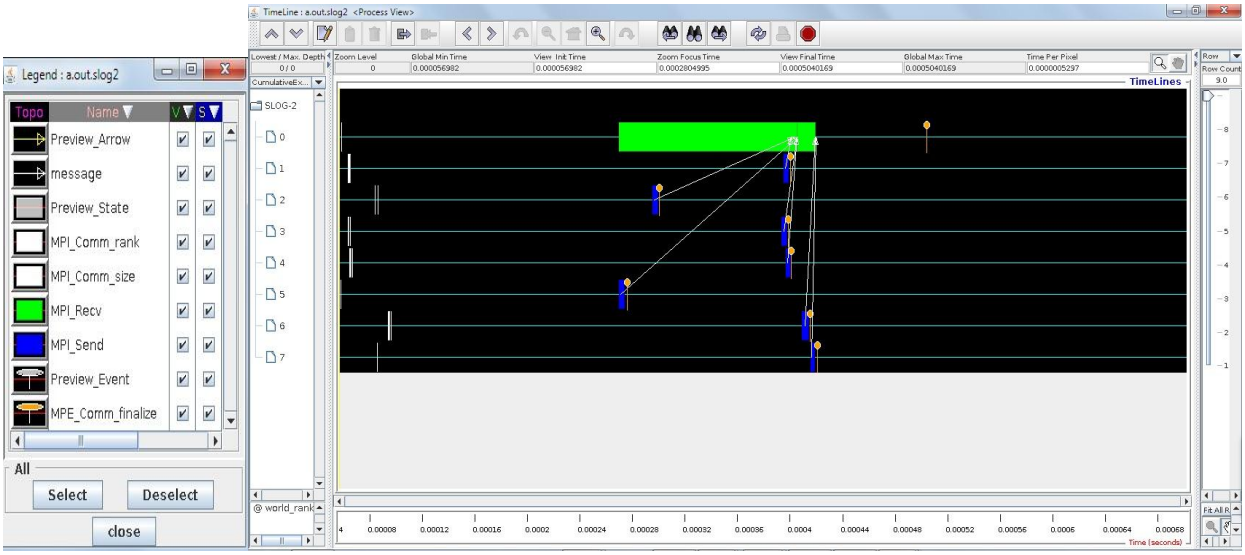


Figure 21: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

n=16 in the Point to Point Operations code

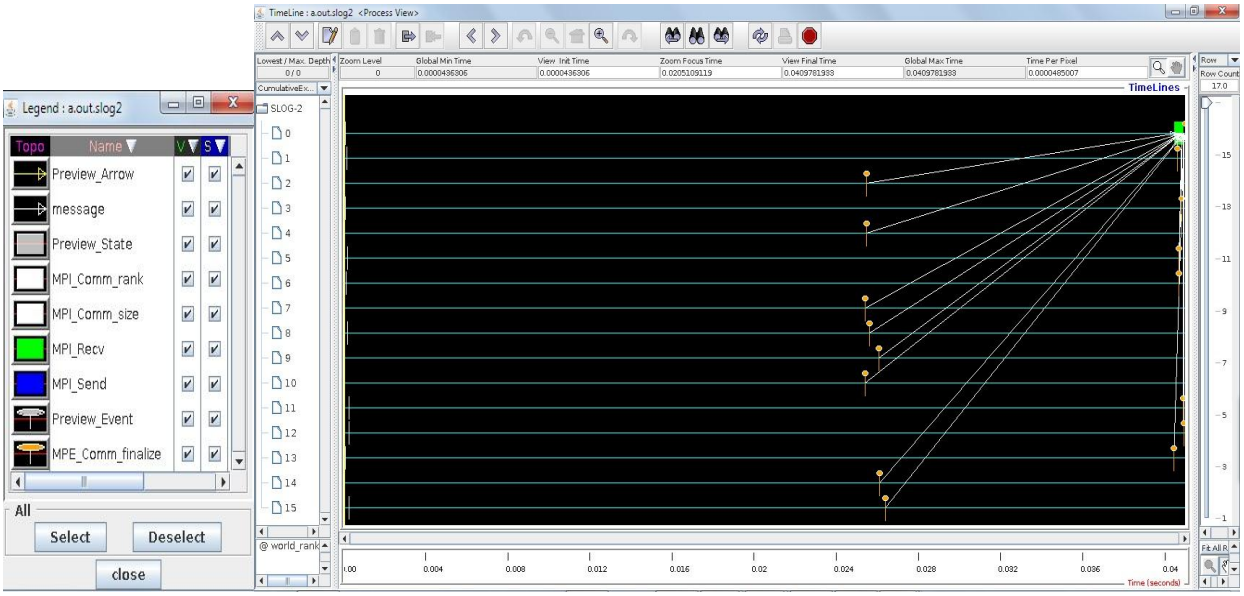


Figure 22: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

- The jumpshot (Java based visualization tool) pictures shown in Figure 19-22: (a) and (b) present a graphical view of the space-time status of the parallel computation involved when different numbers of processes are used to evaluate the parallel code written using Point to Point operations.
- The above pictures depict the timeline of state of different processors during Point to Point Operations, where the final data is collected by the Master Process and all the slaves send their data to it (one by one).

- For all the above cases, the communication time was obtained by finding time taken to execute all MPI_Send, MPI_Recv and MPI_Reduce commands (by placing the code involved in communication between MPI_Wtime statements). The computation time was found by subtracting communication time from the total time.

n=2 in the Collective Operations code

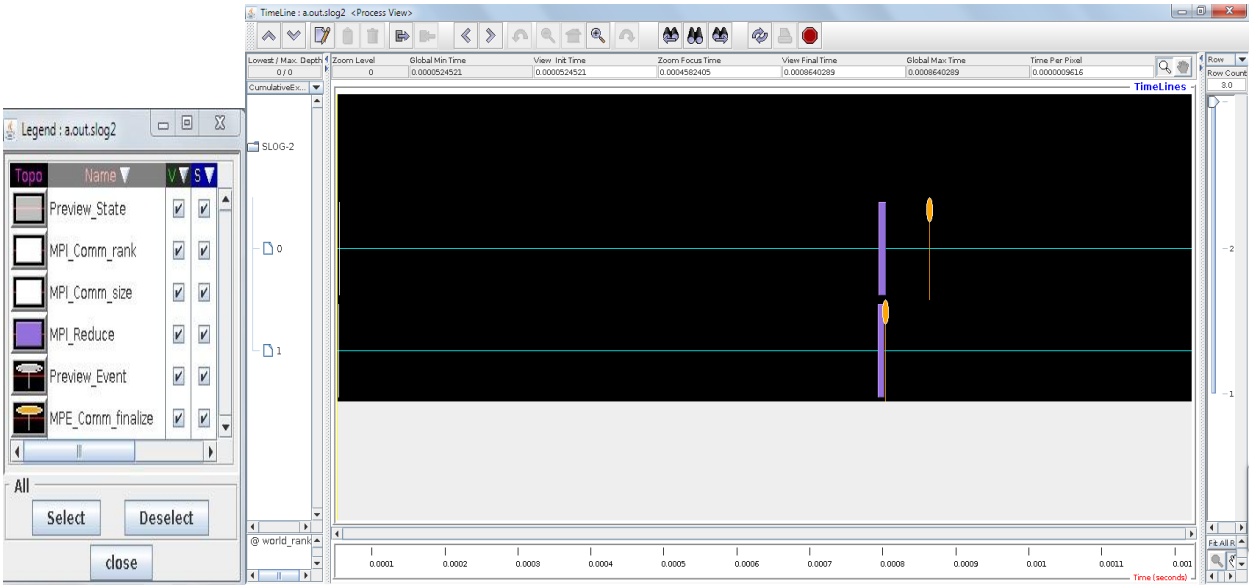


Figure 23: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

n=4 in the Collective Operations code

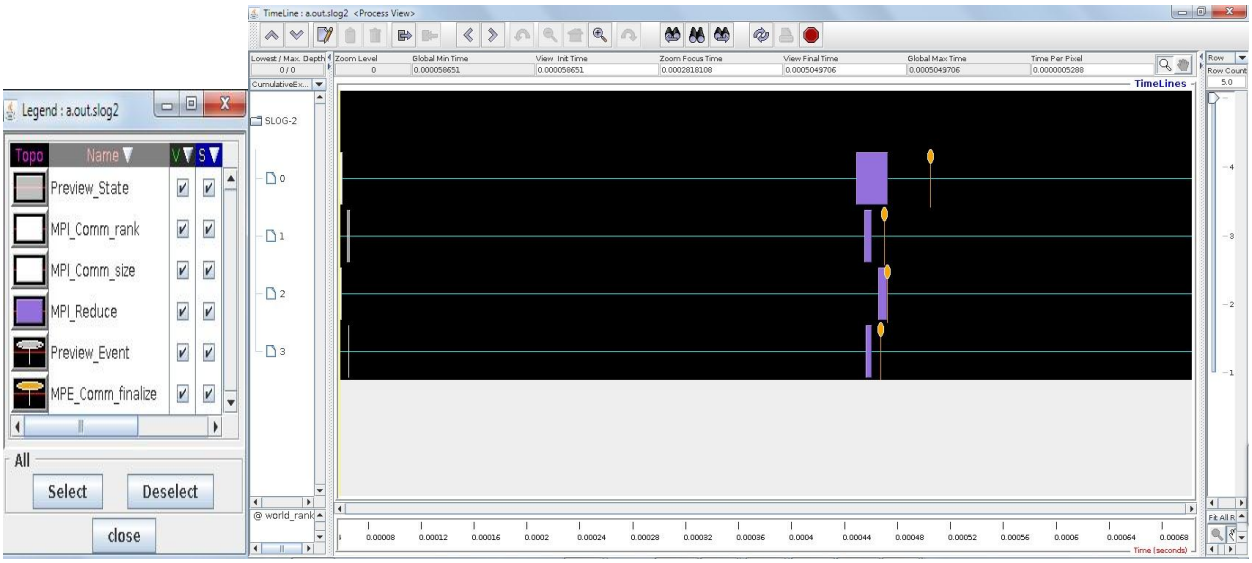


Figure 24: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

n=8 in the Collective Operations code

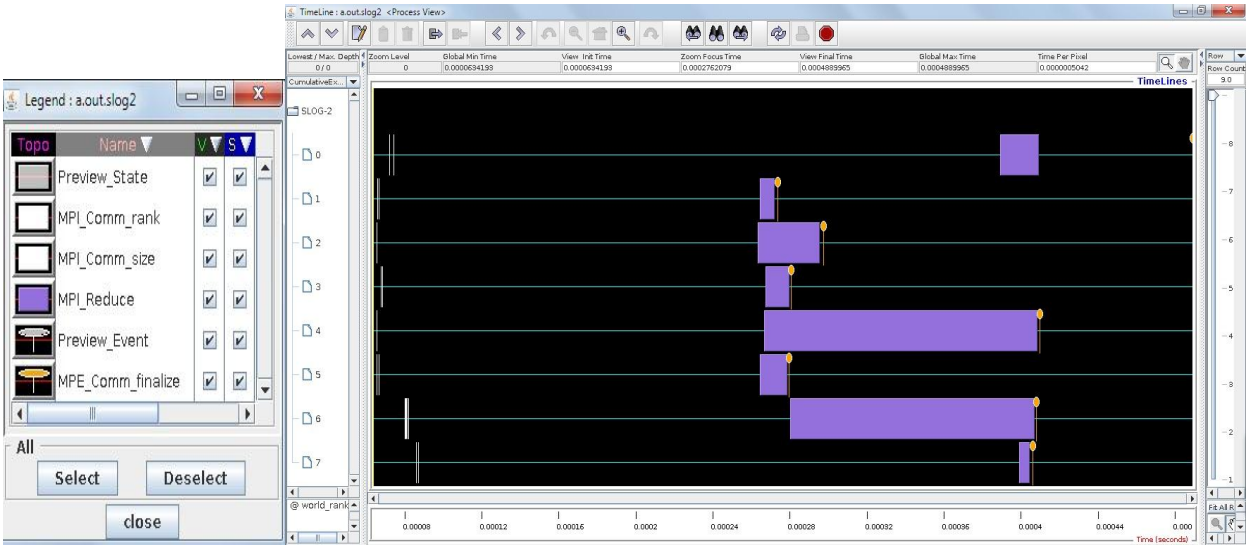


Figure 25: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

n=16 in the Collective Operations code

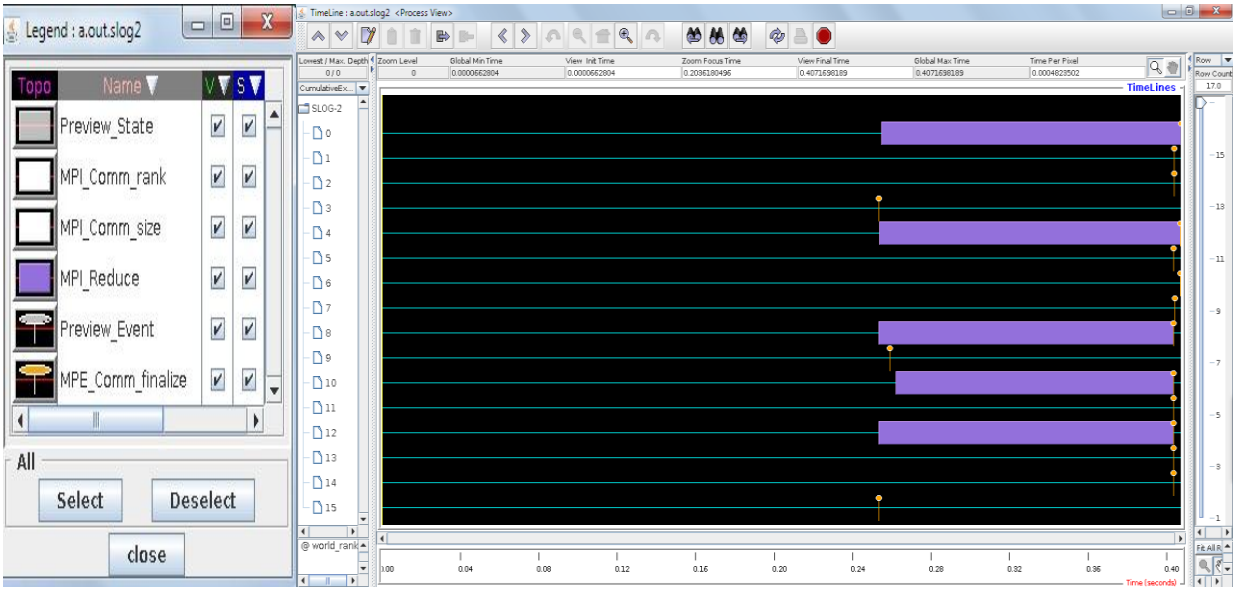


Figure 26: (a) Legend: Output.slog2 (b) Timeline: Output.slog2

- The jumpshot (Java based visualization tool) pictures shown in Figure 23-26: (a) and (b) present a graphical view of the space-time status of the parallel computation involved when different numbers of processes are used to evaluate the parallel code written using Collective Operations.
- The above pictures depict the timeline of state of different processors during Collective Operations, where the final data is sent in a reducing tree fashion by all the processors to each other.

- For all the above cases, the communication time was obtained by finding time taken to execute all MPI_Send, MPI_Recv and MPI_Reduce commands (by placing the code involved in communication between MPI_Wtime statements). The computation time was found by subtracting communication time from the total time.

4.3.1 The value of pi obtained becomes more precise and accurate as number of processors is increased keeping the number of Darts constant. Also, the precision and accuracy increases if number of Darts is increased for a particular value for the number of Processors. Thus, the percentage error tends to decrease as number of processes is increased or if number of Darts thrown is increased, the reason being that more the number of values taken into consideration, better are the approximated values of pi.

4.3.2 It may be noticed that time elapsed is slightly more in case when coding is done using Collective Operations than when done using Point to Point Processes. Also, the overall efficiency when code is written using Point to Point Operations is quite more than when the parallel code is written using Collective Processes. Theoretically, efficiency must increase when parallel code consists of Collective Processes, i.e., Collective processes should have been faster than Point to Point Processes. A possible reason for this anomaly might be that the processors may not be working in synchronization, in case when parallel code contains Collective Processes. This problem can be addressed using MPI_Barrier() before MPI_Reduce(function call). Usage of MPI_Barrier(MPI_COMM_World) would result into synchronization of all the processes. Thus, efficiency of MPI_Reduce() would increase as all the processes would be involved in computations without any process sitting idle waiting for another processes to finish. Figure 27 depicts the synchronization achieved by using MPI_Barrier() [colored in yellow]

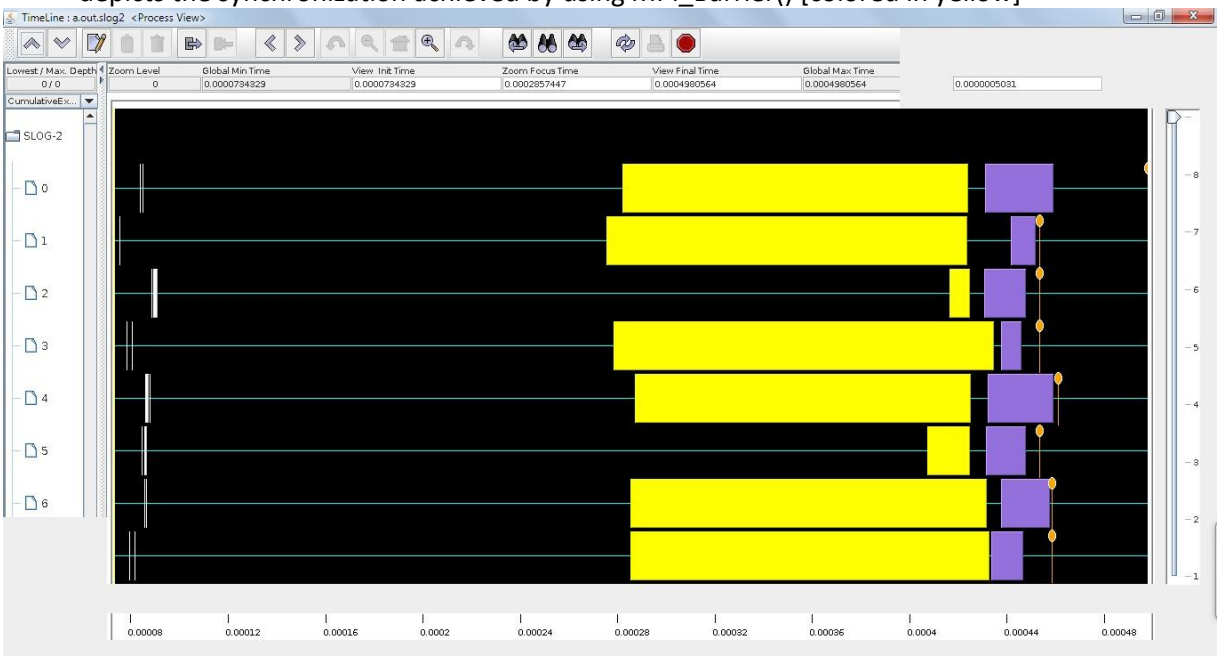


Figure 27: Timeline: Output.slog2

4.3.3 It can be seen from the values recorded that as the number of processes are increased, there is a significant decrease in total time elapsed. Hence parallelizing the code with increased number of processors, keeping the number of darts thrown constant, decreases workload on each processor resulting in a decrease in overall processing time.

4.3.4 As the number of processors increase, the overall efficiency decreases. This may be attributed to the fact that increase in number of processors increases communication time since the parallel

code as well as overheads need to be passed on to more number of processors. Thus, speed up, inversely related to the time elapsed, decreases.

4.3.5 The ratio of computation time to communication time rises drastically as number of darts is increased for a particular processor, the reason being, that communication time remains nearly the same for all cases while computation time increases with increase in value of number of darts thrown.

4.3.6 In general, actual Value of speedup and efficiency tend to increase with increase in number of darts thrown (for both types of operations, see graphs in Figure 11, 13, 15, 17), the reason being that, when the input values reach near the value of bandwidth of the cluster, there is a significant time elapse in each program. This is the point where the parallelism helps, in decreasing overall execution time resulting into an increase in predicted speedup as well as predicted efficiency.

4.3.7 Predicted value of speedup tends to reach the value of number of processors in that case as the value of Number of Darts is increased (for both types of operations, see graphs in Figure 12, 14, 16, 18). It is because, as number of darts increases, the ratio between the time elapsed when a single processor is used to time elapsed when multiple processors are used increases drastically.

4.3.8 It may be seen from the tables that the value of efficiency for each processor is highest for the last case, i.e., when the number of darts thrown is 10^8. It is because as the value for number of darts increases, the resources tend to be fully used because 10^8 is quite closer to the bandwidth of the cluster as estimated before. Hence, when compared to time elapsed when a single processor is used, speedup and efficiency are found to be very high.

4.3.9 Refer Appendix for Parallel code of this question

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help I received from Prof. Murali Damodaran. I would like to extend my sincere thanks to him. I am highly indebted to his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project.

My thanks and appreciations also go to my friend Ravi and classmate Rachit in developing the project tools (jumpshot and many others) and people who have willingly helped me out with their abilities.

APPENDIX

Q1: PingPong

```
//by Shashank Heda
//Question 1 Ping Pong between two processors

#include <stdio.h>
#include<stdlib.h>                          //For the function
exit(1)
#include "mpi.h"
#include "mpe.h"

int main(int argc, char** argv)
{
      int my_rank;
      int p;
      int received_rank;
      double time1, time2;

      MPI_Init(&argc, &argv);                                  // Start
up MPI
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                    //
Get my process rank
      MPI_Comm_size(MPI_COMM_WORLD, &p);                          //
To find number of processes are being used

      time1 = MPI_Wtime();

      // Checking if number of processes is equal to 2
      if (p!=2)
      {
            if (my_rank==0)
                  printf("The number of processes must be 2\n");

            exit(1);
      }


      // Start pingpong-ing between the processors

            if (my_rank == 0)
            {
                  MPI_Send(&my_rank,1,MPI_INT,1,0, MPI_COMM_WORLD);
                  printf("\nProcessor 0 sends %d to processor
1\n",my_rank);

      MPI_Recv(&received_rank,1,MPI_INT,1,2,MPI_COMM_WORLD,MPI_STATUS
_IGNORE);
                  printf("\nProcessor 0 receives %d from Processor
1\n",received_rank);
            }
            else if (my_rank == 1)
            {
```

```
                MPI_Send(&my_rank,1,MPI_INT,0,2, MPI_COMM_WORLD);
                printf("\nProcessor 1 sends %d to processor
0\n",my_rank);
MPI_Recv(&received_rank,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNO
RE);
                printf("\nProcessor 1 receives %d from Processor
0\n",received_rank);
          }
      time2 = MPI_Wtime();
      printf("\nTotal Time Elapsed: %lf\n\n\n",time2-time1);
      /* Shut down MPI */
      MPI_Finalize();

      return 0;
}
```

Q2: Bandwidth

```
//by Shashank Heda
//Program to Estimate bandwidth and message latency on the HPC
Cluster

#include<stdio.h>
#include<stdlib.h>                    //for the function exit
#include "mpi.h"
#include"mpe.h"

int main(int argc, char* argv[])
{
      int my_rank,np,i,j,n;
      float data[10000];
      double time1,time2;

      //Starting MPI
       MPI_Init(&argc,&argv);
       MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
       MPI_Comm_size(MPI_COMM_WORLD,&np);

       if(np!=2)
       {
               printf("\n\n\tSorry!!! The number of processes has to
be 2\n");
               exit(1);
       }

      for(i=1;i<=100;i++)
      {
            n=100*i;

            time1=MPI_Wtime();
              for(j=0;j<10000;j++)
```

```
                {
                        if(my_rank==0)
                        {

MPI_Send(data,n,MPI_FLOAT,1,0,MPI_COMM_WORLD);

MPI_Recv(data,n,MPI_FLOAT,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                        }
                        else
                        {

MPI_Recv(data,n,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

MPI_Send(data,n,MPI_FLOAT,0,0,MPI_COMM_WORLD);

                        }
                }

             time2=MPI_Wtime();

           if(my_rank==0)
           {
                  printf("\nTime for the Ping-Pong process between
two processors is %lf for size=%d for the variable float\n",time2-
time1,n);
           }
      }



      MPI_Finalize();
      exit(1);
}
```

Q3: Trapezoidal Serial Code

```
//by Shashank Heda
/* Serial Code for calculating definite integral using the
trapezoidal rule.
*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#define f(x) log(x + 1)/(x*x + 4*x + 3)

int main()
{
  double integral;  /* Store end-result in integral         */
  float a, b;       /* Left and right endpoints             */
  long int n;       /* Number of trapezoids                 */
```

```
  double h;          /* Trapezoid base width                  */
  double x,timer;
  long int i;
  clock_t start_t, stop_t;

  a = 0.0;
  b = 30.0;
  n = 10000000;

  start_t = clock();

  h = (b - a) / n;
  integral = (f(a) + f(b)) / 2.0;
  x = a;
  for (i=1; i<n; i++)
  {
    x += h;
    integral += f(x);
  }
  integral *= h;

  stop_t = clock();
  timer = (double)((stop_t - start_t) / CLOCKS_PER_SEC);

  printf("With n = %ld trapezoids,\n", n);
  printf("our estimate of the integral from %.2f to %.2f = %e.\n",
a, b, integral);
  printf("CPU time elapsed: %lf seconds\n", timer);

  return 0;
}
```

Q3: Trapezoidal Parallel Code

```
//by Shashank Heda
/* Parallel Trapezoidal Rule

Estimation of the integral from a to b of f(x)
         using the trapezoidal rule and n trapezoids.
*/

#include <stdio.h>
#include <math.h>
#include "mpi.h"
#include "mpe.h"
#define f(x) log(x + 1)/(x*x + 4*x + 3)

// Calculate local integral
double Trap(float local_a, float local_b, long int local_n, double
h);

int main(int argc, char** argv)
{
  int         my_rank;          /* My process rank           */
```

```
  int         p;                /* The number of processes   */
  float       a = 0.0;          /* Left endpoint             */
  float       b = 30.0;         /* Right endpoint            */
  long int    n = 10000000;     /* Number of trapezoids      */
  double      h;                /* Trapezoid base length     */
  float       local_a;          /* Left endpoint my process  */
  float       local_b;          /* Right endpoint my process */
  long int    local_n;          /* Number of trapezoids for  */
                                /* my calculation            */
  double      integral;         /* Integral over my interval */
  double      total_integral;   /* Total integral            */
  int         source;           /* Process sending integral  */
  int         dest = 0;         /* All messages go to 0      */
  int         tag = 0;
  MPI_Status  status;
  int event1a, event1b;
  double startTime, endTime, timeDifference;

  /* Let the system do what it needs to start up MPI */
  MPI_Init(&argc, &argv);
  /* Get my process rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  /* Find out how many processes are being used */
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  event1a = MPE_Log_get_event_number();
  event1b = MPE_Log_get_event_number();

  MPE_Describe_state(event1a, event1b, "Compute", "red");

  if (my_rank==0)
       startTime=MPI_Wtime();

  h = (b - a) / n;    /* h is the same for all processes */
  local_n = n / p;    /* So is the number of trapezoids */

  /* Length of each process' interval of integration = local_n * h.
   * So my interval starts at: */
  MPE_Log_event(event1a, 0, "start compute");
  local_a = a + my_rank * local_n * h;
  local_b = local_a + local_n * h;
  integral = Trap(local_a, local_b, local_n, h);
  MPE_Log_event(event1b, 0, "end compute");

  /* Add up the integrals calculated by each process */
  if (my_rank == 0)
  {
    total_integral = integral;
    for (source=1; source<p; source++)
    {
      MPI_Recv(&integral, 1, MPI_DOUBLE, source, tag,
MPI_COMM_WORLD, &status);
      total_integral = total_integral + integral;
```

```
    }
  }
  else MPI_Send(&integral, 1, MPI_DOUBLE, dest, tag,
MPI_COMM_WORLD);

  if (my_rank==0)
  {
    endTime = MPI_Wtime();
    timeDifference = endTime - startTime;
  }

  /* Print the result */
  if(my_rank == 0)
  {
    printf("With n = %ld trapezoids, \n", n);
    printf("our estimate of the integral from %f to %f = %e\n", a,
b, total_integral);
    printf("Time taken for whole computation = %f seconds\n",
timeDifference);
  }

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
}

double Trap(float    local_a   /* in */,
            float    local_b   /* in */,
            long int local_n   /* in */,
            double   h         /* in */)
{
  double integral;   /* Store result in integral  */
  double x;
  long int i;

  integral = (f(local_a) + f(local_b))/2.0;
  x = local_a;
  for(i=1; i<=local_n-1; i++)
  {
    x = x + h;
    integral = integral + f(x);
  }
  integral = integral*h;

  return integral;
}
```

Q3(b)Simpson's rule:

```
//by Shashank Heda
//To calculate integral by Simpson's rule
#include<stdio.h>
```

```
#include<math.h>
#include<stdlib.h>
#include "mpi.h"
#include "mpe.h"
#define f(x) log(x + 1)/(x*x + 4*x + 3)


int main(int argc, char* argv[])
{
int my_rank,np,process;
long int i,n=1000, local_n;
double a=0.0 , b=30.0, h, local_a, local_b,x;
double local_int, total_int;
double time1,time2,error;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);

time1=MPI_Wtime();
h=(b-a)/n;
local_n=n/np;

local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;


local_int=f(local_a)+f(local_b);

x=local_a;
for(i=1;i<local_n;i=i+2)
{
x=local_a+i*h;
local_int = local_int + 4*f(x);
}

x=local_a;
for(i=2;i<local_n-1;i=i+2)
{
x=local_a+i*h;
local_int = local_int + 2*f(x);
}

local_int=local_int*h/3.0;

if(my_rank!=0)
{
     MPI_Send(&local_int,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
else
{
     total_int=local_int;
```

```
      for(process=1;process<np;process++)
      {

      MPI_Recv(&local_int,1,MPI_DOUBLE,process,0,MPI_COMM_WORLD,MPI_S
TATUS_IGNORE);
            total_int+=local_int;
      }
}

time2=MPI_Wtime();

if(my_rank==0)
{
      printf("\n\tValue of integration by Simpson's rule is
: %e",total_int);
       printf("\n\tNumber of processes is
: %d",np);
      printf("\n\tTime taken for calculation
: %e",time2-time1);
}

MPI_Finalize();
return 0;
}
```

Q4:Sequential Monte Carlo

```
//by Shashank Heda
// Serial code to compute Pi using Monte Carlo method

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include<time.h>
#define PI      3.14159265358979323846264643
#define SEED 36584136

main(int argc, char* argv)
{
   int iter=0;
   double x,y;
   int i,count=0; /* # of points in the 1st quadrant of unit circle
*/
   double z,pi,timer,error;
   clock_t start_t, stop_t;

   printf("\n\tPlease enter the number of iterations used to
estimate pi: ");
   scanf("%d",&iter);

   start_t = clock();
```

```
   /* initialize random numbers */
   srand(SEED);
   count=0;
   for ( i=0; i<iter; i++) {
      x = (double)rand()/RAND_MAX;
      y = (double)rand()/RAND_MAX;
      z = x*x+y*y;
      if (z<=1) count++;
      }
   pi=(double)count/iter*4;

   stop_t = clock();

   timer = (double)((double)(stop_t - start_t) /
(double)CLOCKS_PER_SEC);

   error = fabs((pi - PI)/PI) * 100;

   printf("\n\tTotal Number of trials= %d , approximate value of pi
is %11.20e",iter,pi);
   printf("\n\tKnown value of  PI                              :
%11.20e",PI);
   printf("\n\tCPU time elapsed                              :%lf
seconds", timer);
   printf("\n\tPercentage Error                                :
%11.10f\n", error);

}
```

Q4:Monte_Carlo using Point to Point operations

```
//by Shashank Heda
/*Program to calculate the value of p using Point to Point
Operations*/

#include <stdio.h>  // Core input/output operations
#include <stdlib.h> // Conversions, random numbers, memory
allocation, etc.
#include <math.h>   // Common mathematical functions
#include <time.h>   // Converting between various date/time formats
#include <mpi.h>    // MPI functionality
#include "mpe.h"
#define PI      3.14159265358979323846264 3

int main (int argc, char *argv[]) {


  int    proc_id,       // Process ID
         np,             // Number of processors
         llimit,        // Lower limit for random numbers
         ulimit,        // Upper limit for random numbers
         n_circle,      // Number of darts that hit the circle
         i;             // Dummy/Running index
  long int ndarts=100000000/16;
```

```
  double pi_current,    // PI calculated by each WORKER
         pi_sum,        // Sum of PI values from each WORKER
         x,             // x coordinate, betwen -1 & +1
         y,             // y coordinate, betwen -1 & +1
         z,             // Sum of x^2 and y^2
         error,         // Error in calculation of PI
         time1,         // Wall clock - start time
         time2,         // Wall clock - end time
        time3,      // Wall clock - communication start time
        time4;      // Wall clock - communication end time

 struct timeval stime;

  llimit   = -1;
  ulimit   = 1;
  n_circle = 0;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  time1 = MPI_Wtime();

  gettimeofday(&stime, NULL);
  srand(stime.tv_usec * stime.tv_usec * stime.tv_usec *
stime.tv_usec);


      for (i = 1; i <= ndarts; i++)
      {
           x = ((ulimit- llimit) * ((double) rand()/(double)
RAND_MAX)) + llimit;
           y = ((ulimit- llimit) * ((double) rand()/(double)
RAND_MAX)) + llimit;

           z=x*x+y*y;
           if (z <= 1.0)
             {
                  n_circle++;
           }
      }

      pi_current = 4.0 * (double)n_circle/(double)ndarts;

      if(proc_id==0)
      pi_sum=pi_current;

      time3 = MPI_Wtime();

      if(proc_id!=0)
      {
```

```
            MPI_Send(&pi_current,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
      }

      else
      {
            for(i=1;i< np;i++)
            {

      MPI_Recv(&pi_current,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,MPI_STATUS
_IGNORE);
                  pi_sum = pi_sum+pi_current;
            }

            time4 = MPI_Wtime();

            pi_sum=pi_sum/np;
      }


      time2 = MPI_Wtime();


      error = fabs((pi_sum - PI)/PI) * 100;



      if(proc_id==0)
        {
            printf("Known value of  PI
: %11.20e\n",PI);
            printf("Calculated value of pi is
: %11.20e\n",pi_sum);
            printf("Number of processes is
: %d\n",np);
             printf("No. of iterations is
: %ld\n",ndarts);
             printf("Time for the computation process is
: %lf\n",time2-time1);
             printf("Time for the communication process is
: %lf\n",time4-time3);
             printf("Percentage Error
: %11.10f\n", error);

       }

MPI_Finalize();
return 0;
}
```

Q4:Monte Carlo using Collective Operations

```
//by Shashank Heda
//Montecarlo using Collective Operations (Reduce)
#include <stdio.h>  // Core input/output operations
```

```
#include <stdlib.h> // Conversions, random numbers, memory
allocation, etc.
#include <math.h>   // Common mathematical functions
#include <time.h>   // Converting between various date/time formats
#include <mpi.h>    // MPI functionality
#include "mpe.h"
#define PI      3.14159265358979323846264 3

int main (int argc, char *argv[]) {


  int    proc_id,       // Process ID
         np,             // Number of processors
         llimit,        // Lower limit for random numbers
         ulimit,        // Upper limit for random numbers
         n_circle,      // Number of darts that hit the circle
         i;             // Dummy/Running index
  long int ndarts=10000000/16;

  double pi_current,    // PI calculated by each WORKER
         pi_sum,        // Sum of PI values from each WORKER
         x,             // x coordinate, betwen -1 & +1
         y,             // y coordinate, betwen -1 & +1
         z,             // Sum of x^2 and y^2
         error,         // Error in calculation of PI
         time1,         // Wall clock - start time
         time2,         // Wall clock - end time
       time3,      // Wall clock - start communication
       time4;      // Wall clock - end communication

     struct timeval stime;

     llimit   = -1;
     ulimit   = 1;
     n_circle = 0;

     MPI_Init(&argc, &argv);

     MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
     MPI_Comm_size(MPI_COMM_WORLD, &np);

     time1 = MPI_Wtime();

     gettimeofday(&stime, NULL);
     srand(stime.tv_usec * stime.tv_usec * stime.tv_usec *
stime.tv_usec);


     for (i = 1; i <= ndarts; i++)
     {
          x = ((ulimit- llimit) * ((double) rand()/(double)
RAND_MAX)) + llimit;
```

```
            y = ((ulimit- llimit) * ((double) rand()/(double)
RAND_MAX)) + llimit;

            z=x*x+y*y;
            if (z <= 1.0)
            {
                  n_circle++;
            }
      }

      pi_current = 4.0 * (double)n_circle/(double)ndarts;

      if(proc_id==0)
      pi_sum=pi_current;
      MPI_Barrier(MPI_COMM_WORLD);
      time3 = MPI_Wtime();

      MPI_Reduce(&pi_current, &pi_sum, 1, MPI_DOUBLE, MPI_SUM, 0 ,
MPI_COMM_WORLD);

      time4 = MPI_Wtime();

      pi_sum=pi_sum/np;


      time2 = MPI_Wtime();

      error = fabs((pi_sum - PI)/PI) * 100;

      if(proc_id==0)
            {
            printf("Known value of  PI
: %11.20e\n",PI);
            printf("Calculated value of pi is
: %11.20e\n",pi_sum);
            printf("Number of processes is
: %d\n",np);
             printf("No. of iterations is
: %ld\n",ndarts);
            printf("Time for the computation process is
: %lf\n",time2-time1);
            printf("Time for the communication process is
: %lf\n",time4-time3);
             printf("Percentage Error
: %11.10f\n", error);
       }

MPI_Finalize();
return 0;
}
```