# Project 2: Pi by MPI

Team 7: Ian Freeman, Nick Schwartz, Anna Brandl, Jingyao Tang, Xinrui Yu

## Warm-Up Exercises

## Exercise 2.18

**Exercise 2.18.** Explain the problem with the following code:

```
// serial initialization
for (i=0; i<N; i++)
  a[i] = 0.;
#pragma omp parallel for
for (i=0; i<N; i++)
  a[i] = b[i] + c[i];
```

The issue is that multiple threads are writing to the same memory location (a[i]) at the same time without synchronization. Section 2.6.1.5 “Race Conditions, Thread Safety, and Atomic Operations” discusses this issue of race conditions and the importance of having synchronization mechanisms. The way the code is currently set up can lead to inaccurate results since the final value of a[i] depends on the sequence in which the threads are executed.

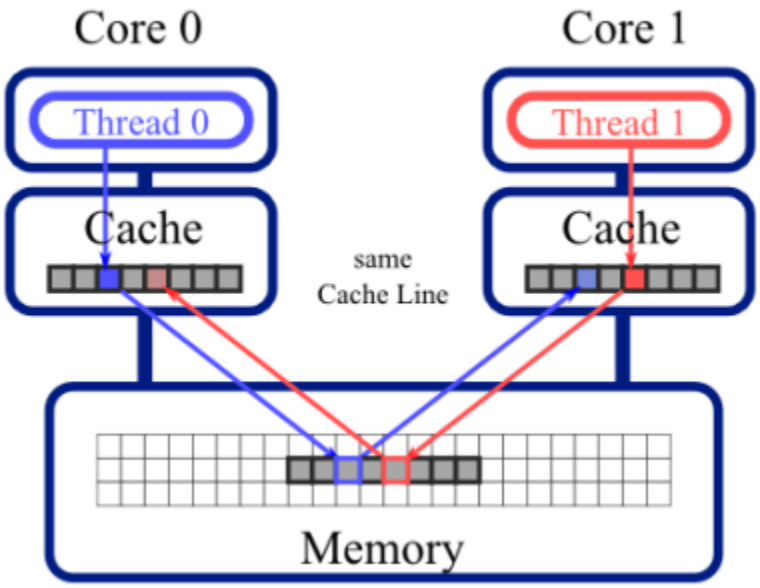
## Exercise 2.19

**Exercise 2.19.** Let’s say there are $t$ threads, and your code looks like

```
for (i=0; i<N; i++) {
  a[i] = // some calculation
}
```

If you specify a chunksize of 1, iterations $0, t, 2t, \ldots$ go to the first thread, $1, 1+t, 1+2t, \ldots$ to the second, et cetera. Discuss why this is a bad strategy from a performance point of view. Hint: look up the definition of *false sharing*. What would be a good chunksize?

False sharing happens when threads on separate cores change variables that are located close to each other in memory, even if the variables are not related. This slows down the performance. The exercise discusses assigning adjacent iterations to different threads, which may lead to the threads accessing adjacent elements of a[]. Then, all those threads must be synchronized between the cores, which slows down performance. This is that false sharing concept.




In our case, several threads try to update on a[].

# Exercise 2.21

**Exercise 2.21.** There is still a problem left with this code: the boundary conditions from the original, global, version have not been taken into account. Give code that solves that

The code in question on the left, the "original, global code" on the right:

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
if (myTaskID==0) leftproc = MPI_PROC_NULL;
  else leftproc = myTaskID-1;
if (myTaskID==nTasks-1) rightproc = MPI_PROC_NULL;
  else rightproc = myTaskID+1;
MPI_Sendrecv( &b[LocalProblemSize-1], &bfromleft,  rightproc );
MPI_Sendrecv( &b[0],                  &bfromright, leftproc);
```

```
for (i=0; i<ProblemSize; i++) {
  if (i==0)
    a[i] = (b[i]+b[i+1])/2
  else if (i==ProblemSize-1)
    a[i] = (b[i]+b[i-1])/2
  else
    a[i] = (b[i]+b[i-1]+b[i+1])/3
}
```

The new (left) code only handles getting the appropriate data to the correct location so the original (right) code can be executed. To solve the problem of boundary conditions, the calculated variable "`LocalProblemSize`" needs to be invoked to determine when we have reached the boundary of our initialized function. First the code on the left should execute, storing `bleft=b[i-1]` and `bright=b[i+1]`, from the global case. Then, we should handle boundary conditions and replace the code on the right with:

```
for (i=0; i<LocalProblemSize; i++) {
      // Left boundary condition - At the beginning of the first processor
      if (i==0 && myTaskID==0) {a[i] = (b[i]+bright)/2;}

      // Right boundary condition - At the end of the last processor
      else if (i==LocalProblemSize && myTaskID==nTasks-1) {a[i] = (b[i]+bleft)/2;}

      // All other times
      else {a[i] = (b[i]+bleft+bright)/3;}
}
```

This implementation assumes a somewhat linear distribution of the vector b such that the first entry of b is on the first processor and similarly with the last entry.

# Exercise 2.22

**Exercise 2.22.** Take another look at equation (2.5) and give pseudocode that solves the problem using non-blocking sends and receives. What is the disadvantage of this code over a blocking solution?

```
initialize recvbuffer;
MPI_Recv(&recvbuffer, x[N-1]);
y[0] = y[0] + recvbuffer;

For (i=1; i<n_processor-1; i++) {

      MPI_IRecv(x[i-1], recvbuffer);
      y[i] = y[i] + recvbuffer;
      MPI_Wait();
}
```

Note: we have omitted most of the details of MPI_IRecv, as they are ill-posed for this question. Specifically, the datatype, where the data is stored, and the communicator object.

# Exercise 2.23

**Exercise 2.23.** Analyze the discussion in the last item above. Assume that the bandwidth between the two nodes is only enough to sustain one message at a time. What is the cost savings of the hybrid model over the purely distributed model? Hint: consider bandwidth and latency separately.

Looking at bandwidth savings – using hybrid has the potential to reduce the bandwidth from four messages to one bundled message, hence there could be a savings of about 75%. However, since the bandwidth between the nodes can only sustain one message at a time, then there is no difference between the models.

Looking at latency savings – using hybrid reduces the number of messages passed and instead does a simple data copy, which reduces the latency. In hybrid, the bundled message causes the latency to be lower than the purely distributed model, improving the communication time.

# Exercise 2.27

**Exercise 2.27.** How much can you gain from overlapping computation and communication? Hint: consider the border cases where computation takes zero time and and there is only communication, and the reverse. Now consider the general case.

Consider the pseudocode below, with local variables a, b, c, and variable x stored on processor 3. And the "operation" method takes a long amount of time.

```
a = operation(b);          \\ Line 1
MPI_Sendrecv(&d, &x, 3);   \\ Line 2
a = a + operation(c);      \\ Line 3
a = a + operation(x);      \\ Line 4
```

Notice line 3 does not rely on the presence of variable x. In the limiting case, let's say operation() takes 0 time, but communication time is slow. We waste time in line 2 waiting for communication to occur, when the code *could* be completing line 3 while the communication occurs. Conversely, let's say computation time is long and communication is instant. Now, processor 3 must wait for line 1 to finish before sending variable x. This means processor 3 is wasting time.

In general, one of the two processors will always have to wait some amount of time for this communication to occur, unless we utilize asynchronous communication. If line 2's functionality was replaced by an asynchronous MPI call (and we swapped line 2 and 3) we would waste far less computing time because both processors would no longer have to wait on either processor to finish either of their tasks.

# MPI Basics

- After running the “Hello World” source file using the MPI parallel run command
- When adding the commands MPI_Init and MPI_Finalize and the three different print statements, we ran the code in serial and received the following output (observations below):

```
Pre-Initialize Output
Mid-Initialize Output
dev-intel18
I am processor 0 out of 1
I, processor 0 out of 1 am unique!
Post-Finalize Output
```

- Exercises 2.3, 2.4, and 2.5 from the Parallel Programming book are completed in the final code, and verbalized in the following output:

```
Pre-Initialize Output
Pre-Initialize Output
Pre-Initialize Output
Pre-Initialize Output
Pre-Initialize Output
Mid-Initialize Output
lac-350
I am processor 0 out of 4
I, processor 0 out of 4 am unique!
Mid-Initialize Output
Mid-Initialize Output
Mid-Initialize Output
lac-351
lac-352
I am processor 1 out of 4
I am processor 2 out of 4
lac-400
I am processor 3 out of 4
Post-Finalize Output
Post-Finalize Output
Post-Finalize Output
Post-Finalize Output
```

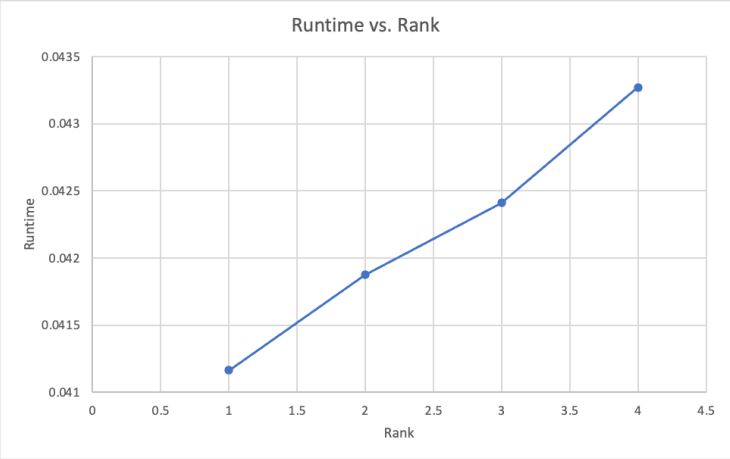
In serial, the outputs are in order. However, in parallel, some processes finish before others and the outputs *seem* out of order. This is to be expected because not all processors finish their tasks at the same time. That said, the pre-initialization and post-finalization outputs are always together because those are blocking MPI commands. Finally, each node has a different name (when we submitted them as a slurm job).

# Eat Some Pi

- (#2) For the first iteration, perform the same number of "rounds" on each MPI rank. Measure the total runtime using MPI_WTIME(). Vary the number of ranks used from 1 to 4. How does the total runtime change?

Varying the number of ranks used from 1 to 4, the runtime is increasing. When running this code, our group changed the rank size and kept the number of rounds the same.

| Rank | Runtime (seconds) |
|---|---|
| 1 | 0.0411634 |
| 2 | 0.0418742 |
| 3 | 0.0424099 |
| 4 | 0.0432706 |




- (#3) Now, divide the number of "rounds" up amongst the number of ranks using the appropriate MPI routines to decide how to distribute the work. Again, run the program on 1 to 4 ranks. How does the runtime vary now?

Runtime varying with the division of number of rounds amongst number of ranks shows much bigger decreases in runtime than compared to the table above. For this portion, our group ran the code such that we are throwing x number of darts which is calculated by dividing the total amount of darts by the number of rounds.

| Rank | Runtime (seconds) |
|---|---|
| 1 | 0.000408 |
| 2 | 0.000166 |
| 3 | 0.000083 |
| 4 | 0.000035 |

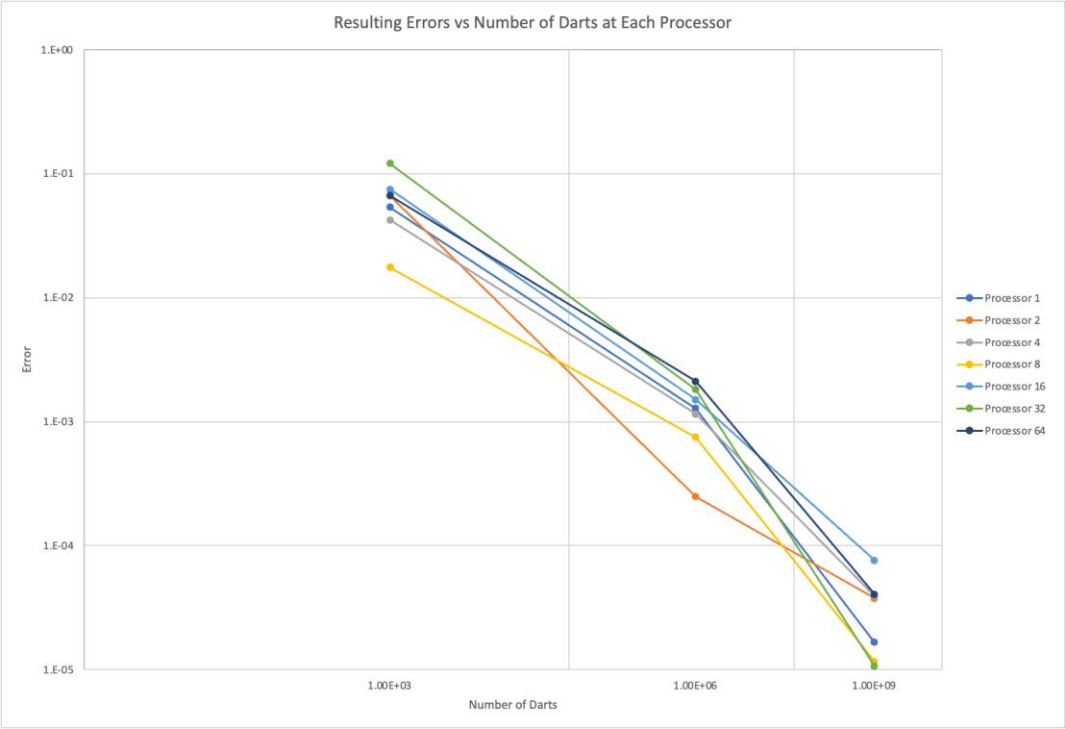
- (#4) Now let's change the number of "darts" and ranks. Use your MPI program to compute pi using total numbers of "darts" of 1E3, 1E6, and 1E9. For each dart count, run your code on HPCC with processor counts of 1, 2, 4, 8, 16, 32, and 64. Keep track of the resulting value of pi and the runtimes. Use non-interactive jobs and modify the submitjob.sb script as necessary.

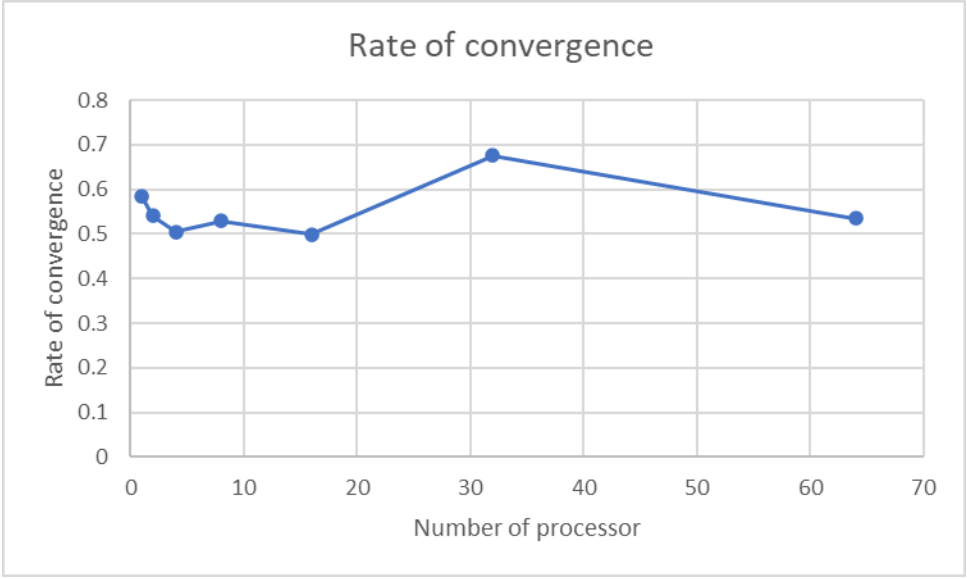
For this problem, we read the question as needed to have a total of 1E3, 1E6, etc. darts after the total number of rounds instead of doing that many darts per round. We did this to save time as running the 1E9 darts 100 times on a single processor would have taken more than three hours. Our data and graphs still look how we expect them to look. It should be noted that we only did strong scaling here and not weak scaling.

| Number of Darts | Processor Counts | Resulting Pi Value | Runtime (seconds) |
|---|---|---|---|
| **1E3** | 1 | 3.088000 | 0.000053 |
| | 2 | 3.208000 | 0.000033 |
| | 4 | 3.184000 | 0.000020 |
| | 8 | 3.124000 | 0.000016 |
| | 16 | 3.066667 | 0.000016 |
| | 32 | 3.020833 | 0.000013 |
| | 64 | 3.075000 | 0.000013 |
| **1E6** | 1 | 3.142872 | 0.043304 |
| | 2 | 3.141344 | 0.021528 |
| | 4 | 3.142744 | 0.010845 |
| | 8 | 3.140844 | 0.005465 |
| | 16 | 3.143100 | 0.002698 |
| | 32 | 3.143408 | 0.001376 |
| | 64 | 3.143712 | 0.002105 |
| **1E9** | 1 | 3.141576 | 42.629151 |
| | 2 | 3.141555 | 21.284142 |
| | 4 | 3.141632 | 10.640985 |
| | 8 | 3.141581 | 5.335699 |
| | 16 | 3.141669 | 2.661769 |
| | 32 | 3.141582 | 1.331284 |
| | 64 | 3.141552 | 0.665052 |

- (#5) For each processor count, plot the resulting errors in your computed values of pi compared to the true value as functions of the number of darts used to compute it. Use log-log scaling on this plot. What is the rate of convergence of your algorithm for computing pi? Does this value make sense? Does the error or rate of convergence to the correct answer vary with processor count? Should it?

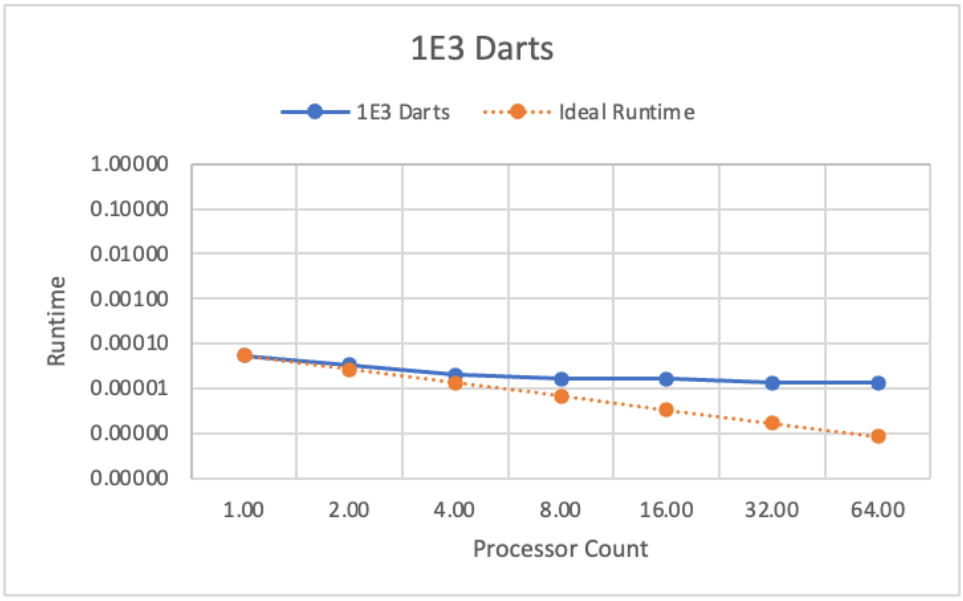
The plots for the resulting errors in the computed values of pi compared to the true function are plotted below on a single figure. The rate of convergence of our algorithm for computing pi is around 0.6. This was done by calculating the slopes of each line in the log-log plot shown below. This makes sense because to get a magnitude more accurate error, we must add 1000 times more darts. Therefore, the convergence rate of about 0.6 makes sense. The rate of convergence does vary with the processor count, which can be seen in the information below, but not by too much. It does make sense that the rate of convergence varies with processor count due to the random number generator.

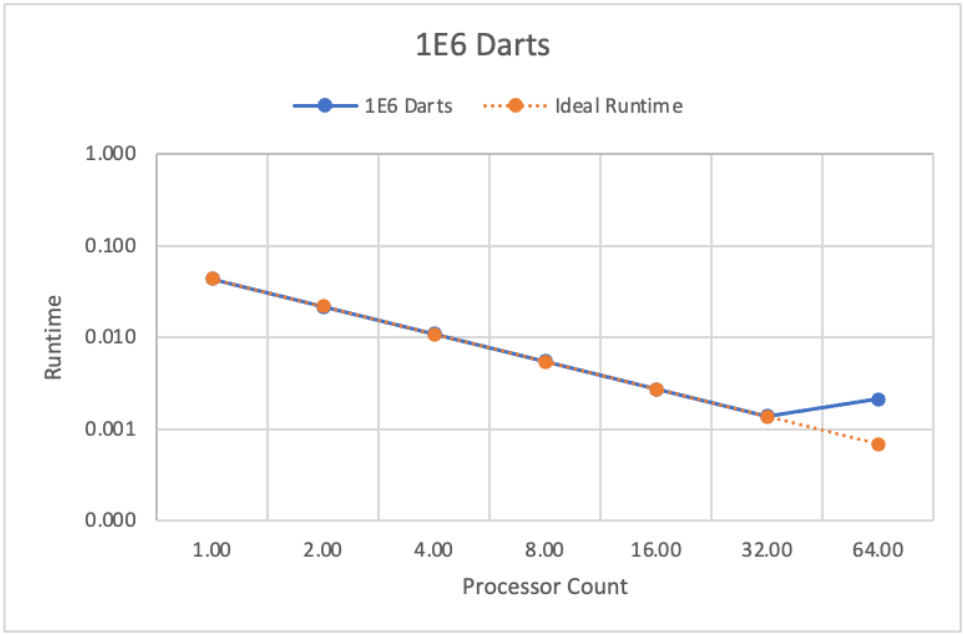


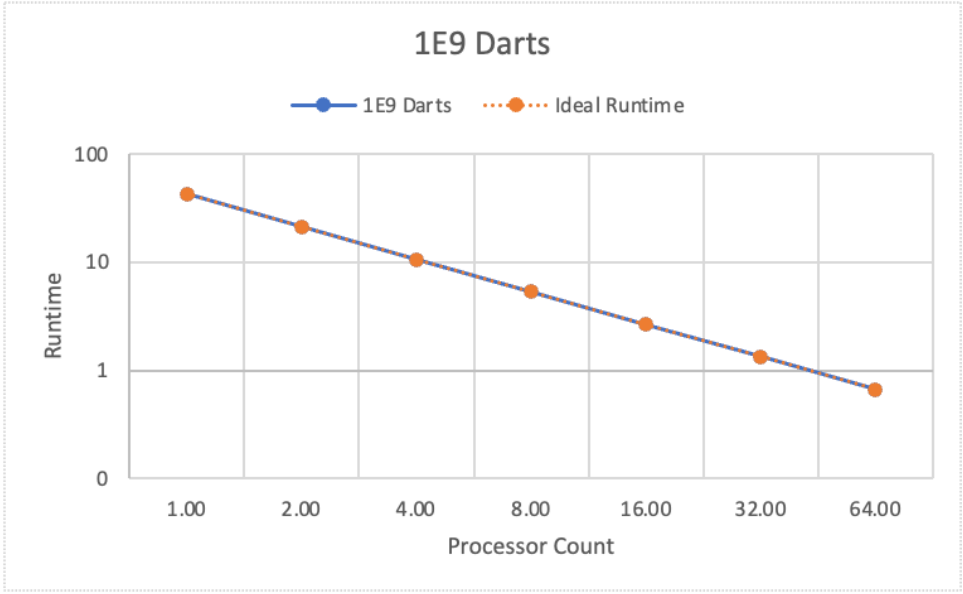



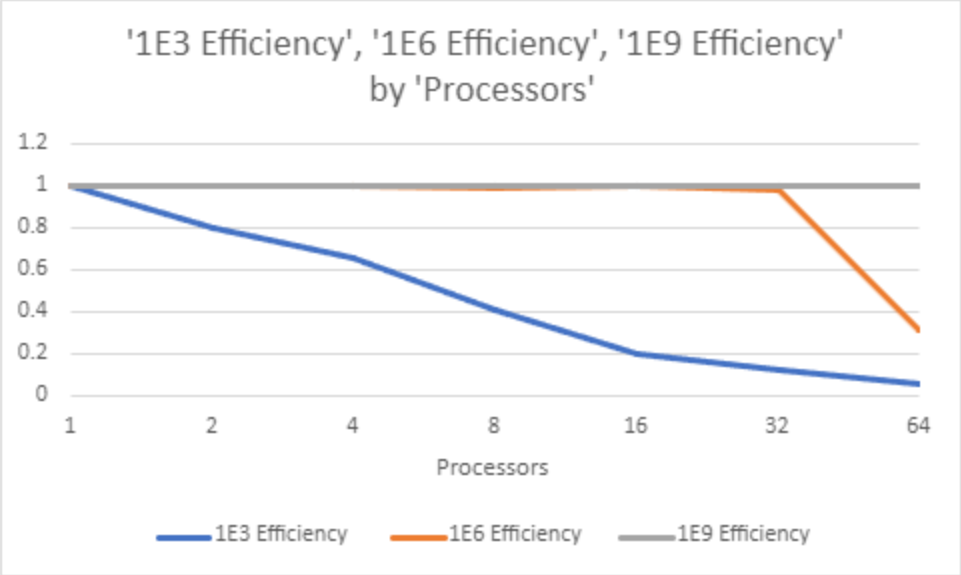

- (#6) For each dart count, make a plot of runtime versus processor count. Each line represents a "strong scaling" study for your code. For each dart count, also plot the "ideal" scaling line. Calculate the parallel scaling efficiency of your code for each dart count. Does the parallel performance vary with dart count? Explain your answer.

The graphs below show runtime vs. processor count for each dart count. Each plot also has the “ideal” scaling line. The fourth plot shows the parallel scaling efficiency for each dart count. As the dart count increases, the parallel scaling efficiency gets closer to one. This suggests that a higher dart count leads to increased efficiency, and estimations of pi calculated from low dart counts are not to be trusted as they are not accurate.


1E3 Darts
1E3 Darts
Ideal Runtime
1.00000
0.10000
0.01000
0.00100
0.00010
0.00001
0.00000
0.00000
Runtime
1.00
2.00
4.00
8.00
16.00
32.00
64.00
Processor Count


1E6 Darts
1E6 Darts
Ideal Runtime
1.000
0.100
0.010
0.001
0.000
Runtime
1.00
2.00
4.00
8.00
16.00
32.00
64.00
Processor Count


1E9 Darts
1E9 Darts
Ideal Runtime
100
10
1
0
Runtime
1.00
2.00
4.00
8.00
16.00
32.00
64.00
Processor Count




- (#7) Going further. Try running your code on different node types on HPCC with varied core counts. In particular, try to ensure that your runs utilize multiple nodes so that the communication network is used. Do you see a change in the communication cost when the job is run on more than one node?

When run on amd20 nodes, the process ran (almost) twice as fast as the intel18 nodes. This varies greatly from node to node, but generally, amd20 is faster than intel18. On multiple nodes (and many darts) we did not observe an increased communication cost, only faster computation.