# CX 4220 / CSE 6220 Programming Assignment 1

Yuho Hsieh, Courtney Wong

February 5, 2019

## Overview

This report describes and analyzes the results obtained from a parallelized pi-approximation dartboard algorithm. In brief, this algorithm involves the generation of $N$ uniformly distributed points ($N \geq 5,000,000$) within a circle of radius 1. The known geometric relationship between the areas of a circle and its inscribed square ($\frac{M}{N}$) are used to approximate the value of pi. The algorithm is parallelized among $P$ processors and repeated $R$ times ($R \leq 100$). The average pi value from each run is averaged to improve accuracy.

## Results

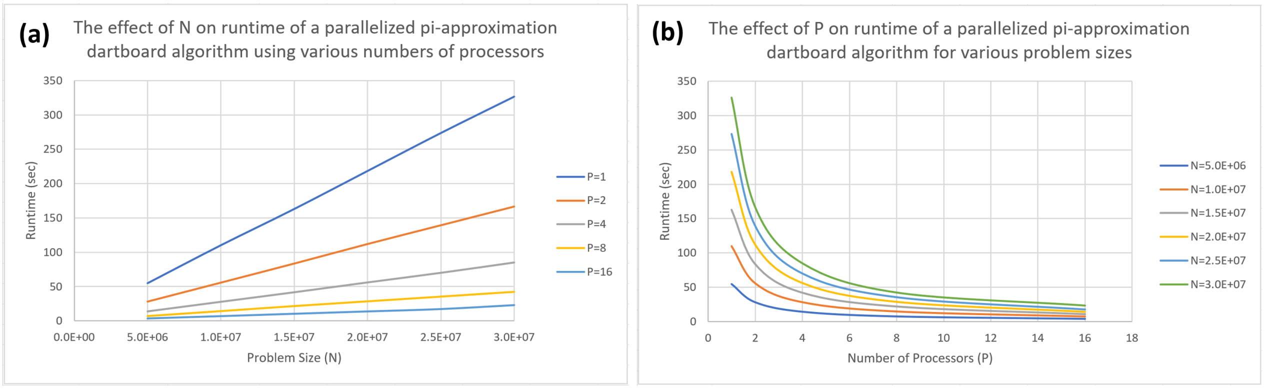

**Figure 1.** Empirical runtime data collected by varying values of $N$ and $P$ independently with $R = 100$. Runtime is defined as maximum time needed by any processor to finish computations and excludes initial broadcasting from the master node. **(a)** Problem size (number of darts) was incremented by 5e+06 while the number of processors were kept constant for each series. **(b)** Number of processors was doubled while the problem size was kept constant for each series.

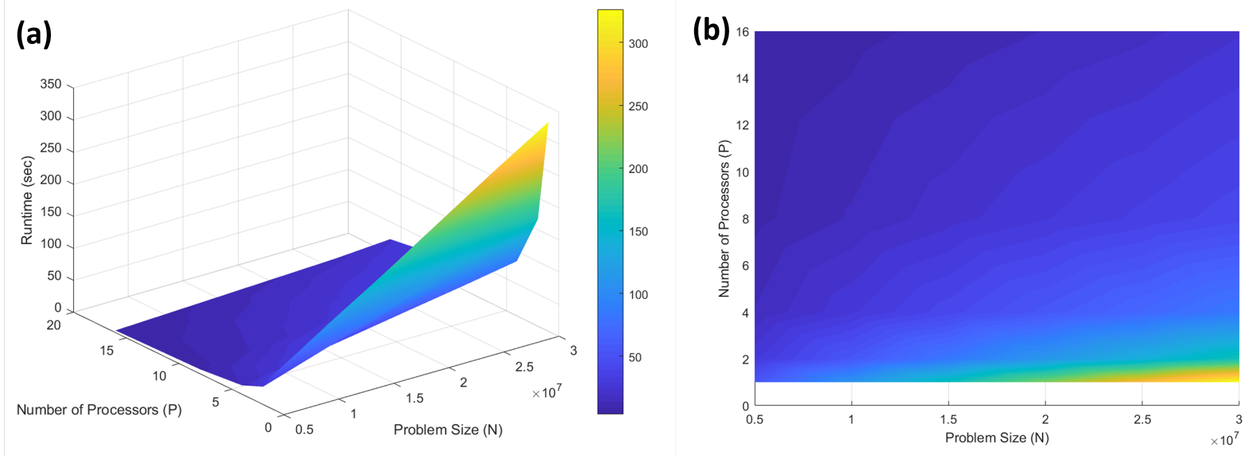

**Figure 2.** Surface meshes generated in MATLAB. **(a)** All data from Figure 1 was represented in 3-D to demonstrate the effect of problem size (x-axis) and number of processors used (y-axis) on algorithm runtime (z-axis). Color gradient generated using interpolative shading. **(b)** Heat map for problem size vs. number of processors. Refer to (a) for color scale.

# Analysis

## Initial Observations

**Figure 1** indicates that runtime grows linearly with $n$ while $p$ is fixed and decays inversely with $p$ while $n$ is fixed. Doubling $n$ results in doubled runtime, and doubling $p$ results in halved runtime. In other words, in order to maintain a desired runtime, $n$ and $p$ must be kept proportional. **Figure 2** allows for analysis of both these variables at the same time and indicates that the immense improvement each time the number of processors is doubled— the non-blue region is quite small and is clustered mainly for $p = 1$, and problem size is hardly an issue for $p = 16$.

## Speedup

Based on the observations, the runtime complexity of this algorithm is $T(n, p) = \Theta(n/p)$. In order to analyze speedup, it can be speculated that a linear dartboard algorithm would involve the sequential generation of $n$ points and a constant-time calculation at the end to produce a pi approximation. This hypothetical algorithm would therefore have a runtime of $T(n, 1) = \Theta(n)$. Speedup is thus optimal: $S(p) = \frac{\Theta(n)}{\Theta(n/p)} = p$. This makes sense, because the work (generating random points) is perfectly divided up between all $P$ processors, such that no processor is idle other than at the very end when the negligible, constant-time pi calculation and file I/O is performed by the master node. This other point worth mentioned is that the communication between processors only happens during broadcasting the number of experiment (how many time to generate randomly distributed points) and reducing the accumulated results. Hence, the communication does not happen frequently within the experiments. Hence the overhead for parallel computing is minimized.

## Efficiency and Scalability

Efficiency is speedup divided by $p$. This algorithm's efficiency is therefore optimally efficient: $E(p) = p/p = 1$. Since work in this algorithm is always divided up evenly among $p$ processors (remainders of $n/p$ are negligible if they are spread out), this algorithm can run on any number of processors smaller than $n$ (if $p > n$, then some processors would be idle, resulting in suboptimal efficiency). The fact that $p$ is scalable with $n$ is excellent news, because it means that optimal efficiency can be maintained even for very large problem sizes— if we double $n$, the same runtime can be maintained if we double $p$, and the $p < n$ ratio will never be exceeded.

## Constants and Latency

Both algorithms must generate $n$ points and calculate pi, repeating $R$ times. The parallel algorithm however must also involve sending data to the master node $(p - 1) * R$ times, via MPI_Reduce. This latency worsens as $p$ becomes large, although it is fortunate that $p << n$. In foresight, this algorithm can be improved by generating all $n * R$ points locally first, keeping each result in a vector, before sending the message to the master node all at once. This would result in a slight improvement because the latency cost will only be paid once per processor, as it is independent of message size: transfer time = latency + message size / network bandwidth.

## Pi Accuracy

Although the accuracy of the pi estimation is not within the scope of this parallel algorithm study, it can be noted that, as expected, the relative error was observed to be generally independent of $P$ and indirectly proportional to both $N$ and $R$. From the empirical data collected, relative error spanned a small range of 0.0005% to 0.0040% and was largely luck-based.