Problem 1

Given 64 elements in an array in shared memory and 8 PEs, design an efficient parallel computing algorithm to compute the sum. Assuming it takes 1 second to add two numbers and 2 seconds to read or write a number, describe your design and calculate the total time required to compute the sum.

Solution:

First, divide the array into 8 sub-arrays with 8 numbers each. Each PE adds elements 0-7 to a local varible `total`, then write that value back to index 0 of its sub array. This creates 8 values that can be added in a merge as follows: PE_0 to PE_3 add up elements at index pairs (0, 8), (16, 24), (32, 40), and (48, 56), putting the result into the first index of the pair. Then PE_0 and PE_1 add up pairs (0, 16) and (32, 48), respectively. Then PE_0 adds values at 0 and 32, storing the result in 0, completing the computation.

Python-ish pseudocode and timing analysis is given.

Listing 1: p is the index of the processing element, from 0 to 7. It is assumed that all thread are running in simultaneous lockstep so no synchonization is needed.

```
1  thread_entry(Int p):
2      Int total = 0
3
4      for i from 0 to 7:
5          total += A[8*p+i]
6      # Partial sums stored at 0 8 16 24 | 32 40 48 56
7      A[8*p] = total
8
9      if p >= 4:
10         halt()
11     # Partial sums stored at 0 8 16 24
12     A[8*p] += A[8*p + 32]
13
14     if p >= 2:
15         halt()
16     # Partial sums stored at 0 8
17     A[8*p] += A[8*p + 16]
18
19     if p == 1:
20         halt()
21     # Final sum stored at 0
22     A[0] += A[8]
23     halt()
```

Line 5 is called 8 times by each processor, and consists of one read (fetching the element from the array) and one addition (adding to `total`). This takes a total of $(2 + 1) \times 8 = \mathbf{24}$

seconds (8 compute, 16 memory). `total` is stored in the start of each processor's subarray, which takes only one write, **2 seconds** (0 compute, 2 memory). After half of the processors drop out, the remaining four processors each add one element of the array to another and store the result in the array, on line 12. This takes two reads, one addition, and one write, which takes 7 seconds. This is repeated on line 17 and 22, for a total of **21 seconds** (3 compute, 18 memory).

In total, this gives 47 seconds to compute the sum. Only 11 of these are spent on additions, showing the I/O bottleneck in distributed computing.

Problem 2

Consider the Weak Scalability Analysis for $n = 1024 \times p$ on slide 17.

Problem 2.a

Give the equations for Speedup and Efficiency in terms of p, respectively.

In Equation 1, we are given an equation for $T(2^q, 2^k)$. Since equations for speedup and efficiency are given in terms of T of p, we need to first find $T(p, n)$ then substitute in for speedup and efficiency.

$$T(p,n) = T(2^q, 2^k) = 2^{k-q} - 1 + 7q \tag{1}$$

$$p = 2^q \text{ and } n = 2^k$$

$$q = \log_2(p) \text{ and } k = \log_2(n) \tag{2}$$

$$T(p,n) = 2^{\log_2(n) - \log_2(p)} - 1 + 7\log_2(p)$$

$$T(p,n) = \frac{n}{p} - 1 + 7\log_2(p) \tag{3}$$

We can then find speedup:

$$\text{Speedup} = \frac{T(1, 1024 \times p)}{T(p, 1024 \times p)} = \frac{1024 \times p - 1}{1023 + 7\log_2(p)} \tag{4}$$

and efficiency:

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{1024 \times p - 1}{p(1023 + 7\log_2(p))} \tag{5}$$

Problem 2.b

Define the equations of Speedup and Efficiency as R functions and verify the Speedup and Efficiency values given on slide 17 using these R functions.

Listing 2: R code to plot efficieny and speedup.

```
require(scales)
library(ggplot2)
library(patchwork)

t <- function(p, n) {
    n/p - 1 + 7*log2(p)
}
tprime <- function(p) {
    1023 + 7*log2(p)
}
speedup <- function(p) {
    (1024*p-1)/(1023+7*log2(p))
}
efficiency <- function(p) {
    100*(1024*p-1)/(p*(1023+7*log2(p)))
}

xdata=c(1,2,4,8,16,32,64,128,256,512)
df <- data.frame(x = xdata, speedup=speedup(xdata), efficiency=efficiency(xdata))
left <- ggplot(df, aes(x = x, y = speedup)) +
    geom_line() +
    scale_x_continuous(trans = log2_trans(), breaks = xdata)
right <- ggplot(data=df, aes(x=x, y=efficiency)) +
    geom_bar(stat=identity) +
    scale_x_continuous(trans = log2_trans(), breaks = xdata)
ggsave( /Users/max/Repos/fall2020/parallel/project1/plot2.png,
       plot = left+right,
       device=png(),
       width=8, height=4, dpi=100)
```

Problem 2.c

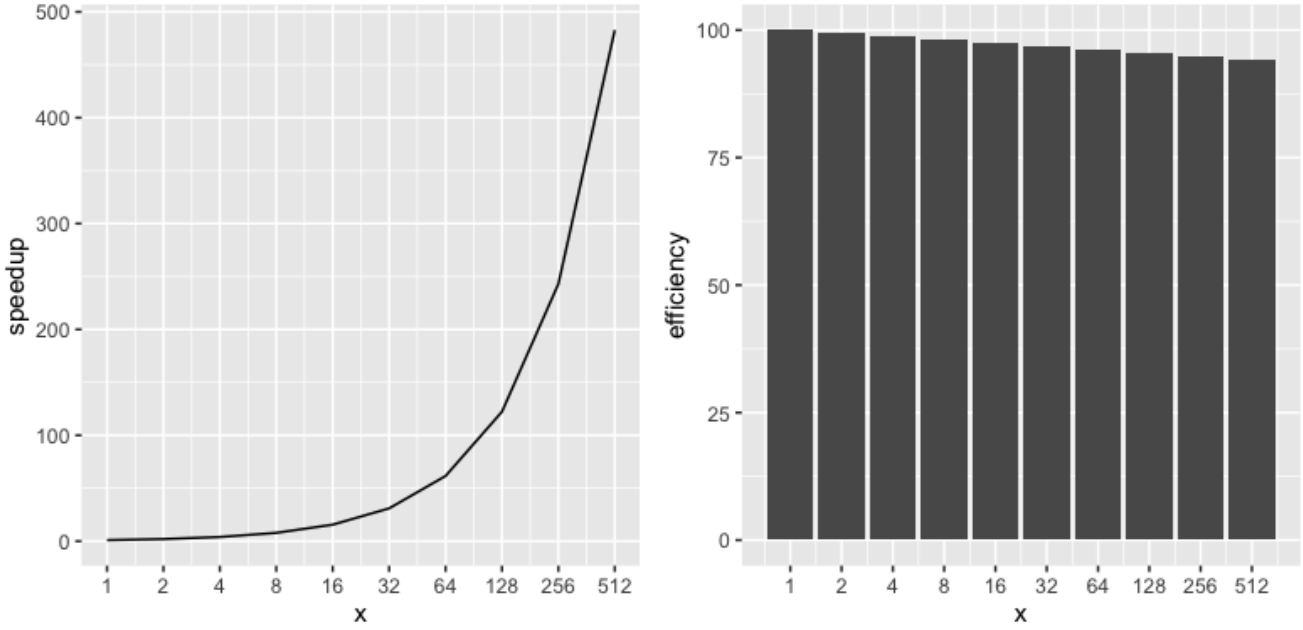
Use R to plot Speedup and Efficiency similar to those given on slide 17.

See Figure 1 on page 4

Problem 3

Consider slides 18-20 (the general case of computing sum using p PEs).

Figure 1: Plots of speedup and efficiency.



Problem 3.a

Give the equation for the Time, T(p, n, a, b)

Solution:

Using the substitition from Equation 2, we can state T in terms of p and n.

$$\begin{aligned} T(p,n,a,b) &= 2b\log_2 p + a\left(2^{\log_2 n - \log_2 p} - 1 + \log_2 p\right) \\ &= 2b\log_2 p + a\left(\frac{n}{p} - 1 + \log_2 p\right) \end{aligned} \tag{6}$$

Problem 3.b

Give the equation for the Speedup, S(p, n, r)

Solution:

Using the substitition from Equation 2, we can state s in terms of p and n.

$$S(p,n,r) = \frac{r\left(2^{\log_2 n}-1\right)}{2\log_2 p + r\left(2^{\log_2 n - \log_2 p} - 1 + q\right)} = \frac{r(n-1)}{2\log_2 p + r\left(\frac{n}{p} - 1 + \log_2 p\right)} \tag{7}$$

Problem 3.c

Give the equation for the optimal number of PEs, P(n, r)

Solution:

This equation is also given and only a substition needs to be performed. Since the function P is not expicitely given, I will restate what is given at the start of my work.

$$2^q = \frac{r\ln 2}{2+r}2^k \tag{8}$$

$$P(n,r) = p = \frac{r\ln 2}{2+r}n \tag{9}$$

Problem 3.d

Use R to define the functions in 3.a, 3.b, and 3.c. Use these R functions to calculate the optimal P for r = 1/3 and n = 1024 and the corresponding speedup, S. Discuss your calculated results compared with those given on the bottom of slide 20

Solution:

The optimal p given $n = 1024$ and $r = 1/3$ is 101.3975, which is close to 100, the approximate number given in the slides. The corresponding speedup is 18.35109, which seems reasonable given the setup. Code is given in Listing 3.

Listing 3: Code to find optimal p and corresponding speedup.

```
t <- function(p, n, a, b) {
    2*b*log2(p) + a*(n/p - 1 + log2(p))
}
s <- function(p, n, r) {
    r*(n-1)/(2*log2(p) + r*(n/p - 1 + log2(p)))
}
p_opt <- function(n, r) {
    n*r*log(2)/(2+r)
}

print(optimal p given n=1024 and r=1/3)
```