CSE 625 Parallel Programming – Project 1

September 1, 2021 (100 points)

Due: September 18 (Saturday) midnight (Submit your project report, and Python notebooks p3.ipynb and p4.ipynb to the Blackboard.)

**Note: Name your project report file like this: <your-last-name>.pdf**

Related Materials

- 01\_Introduction PowerPoint and CSE628\_speedup.ipynb Notebook

- chrono.cpp C++ source file (demos of applications of the chrono class e.g., timing)

- Python runtime timing using the magic command %timeit and %time

- Python Data Science Handbook Notebook, Chapter 2 NumPy and Chapter 4 Matplotlib

Assignments

1(15 points) Test the dot product function of two float vectors given in chrono.cpp.

Run the function for two ones-vectors of size = 14,000,000, 16,000,000, 16,700,000,   
 16,777,216 and 17,000,000, respectively, and use the chrono class to measure the   
 runtime. Collect the results into the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vector size | 14,000,000 | 16,000,000 | 16,700,000 | 16,777,216 | 17,000,000 |
| dot result | 14000000 | 16000000 | 16700000 | 16777216 | 16777216 |
| runtime (secs) | 0.446244200 sec | 0.505624600 sec | 0.530180100 sec | 0.533600000 sec | 0.539189100 sec |

The dot result for vectors of size 17,000,000 is not accurate. Explain why and find a  
 way to fix the problem by modifying the dot function (using float vectors).

The float case in C++ is a 32-bit value with 1 bit being used for the sign of the value (positive or negative) and the next 8 bits being used as the exponent value for the number. This leaves a total of 23 bits + 1 for the actual number value of the float which in out case means that the max value a float can be calculated as is 2^24 which equals 16777216. To fix this I can instead change the result variable to be a double and cast the float values in the vector as doubles during the calculations since a double will have more bit space to represent a larger number. I then return this double casted back as a float to keep the value that was calculated.

float dot(const std::vector<float>& v1, const std::vector<float>& v2)

{

size\_t length = (v1.size() <= v2.size() ? v1.size() : v2.size());

double result = 0;

for (int i = 0; i < length; ++i)

result += (double)v1[i] \* (double)v2[i];

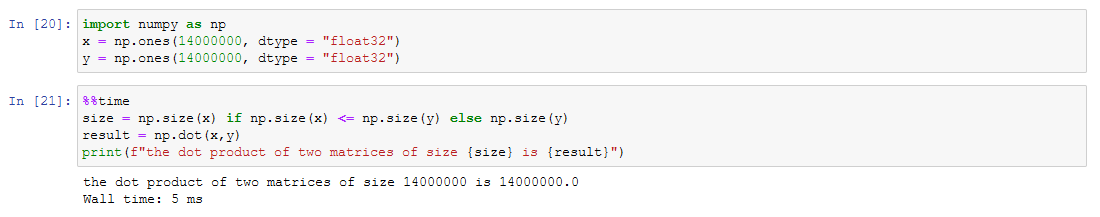
return (float)result;

}

2(15 points) Redo Problem 1 in Python using NumPy. Similarly, collect the results into the   
 table as shown in Problem 1. Discuss and compare the results with those found in  
 Problem 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vector size | 14,000,000 | 16,000,000 | 16,700,000 | 16,777,216 | 17,000,000 |
| dot result | 14000000 | 16000000 | 16700000 | 16777216 | 17000000 |
| runtime (secs) | 0.005 | 0.006 | 0.00658 | 0.006 | 0.00598 |

List your Python code in your project report.



import numpy as np

x = np.ones(17000000, dtype = "float32")

y = np.ones(17000000, dtype = "float32")

%%time

size = np.size(x) if np.size(x) <= np.size(y) else np.size(y)

result = np.dot(x,y)

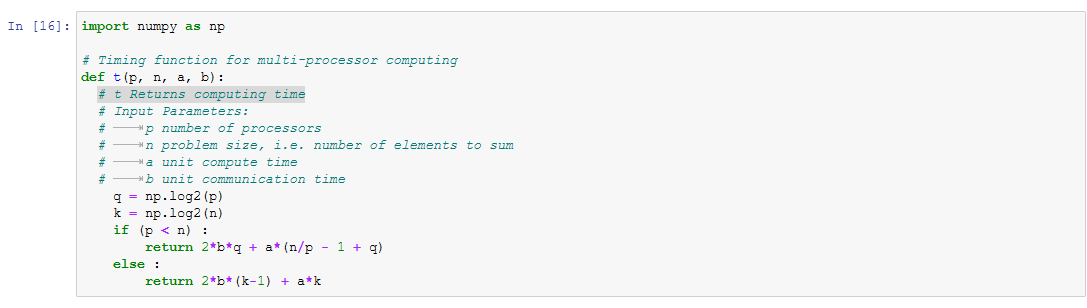
print(f"the dot product of two matrices of size {size} is {result}")

3(30 points) Consider the Weak (Efficiency) Scalability Analysis for *n* = 1024×*p* as   
 shown on slide 17 of 01\_Introduction PowerPoint. Write a Python notebook called,   
 p3.ipynb, to perform the weak scalability analysis,

for n = 1024 and p =1, 2, 4, 6, 8, 16,3 2, 64, 126,2 58, 512.

List your Python code and show the plots (as given on slide 17) in your project report.

I used the following python code to develop the graphs seen bellow to demonstrate week scalability as seen in the lecture slides 17.



import numpy as np

# Timing function for multi-processor computing

def t(p, n, a, b):

# t Returns computing time

# Input Parameters:

# p number of processors

# n problem size, i.e. number of elements to sum

# a unit compute time

# b unit communication time

q = np.log2(p)

k = np.log2(n)

if (p < n) :

return 2\*b\*q + a\*(n/p - 1 + q)

else :

return 2\*b\*(k-1) + a\*k

This segment of code is a function taken from the example .ipynb given in class. It is used as a timing function for the multi-processor computing we are simulating in this question.

# define speedup for 0, 1, 2, 3 communication time unit, respectively

# q is number of processors

# this speedup is for week scalabilty anallysis where the problem size changes with the processor amount

n = 1024

def weak\_s\_0(q):

return t(1,n\*(2\*\*q), 1,0) / t(2\*\*q, n\*(2\*\*q), 1,0)

def weak\_s\_1(q):

return t(1,n\*(2\*\*q), 1,1) / t(2\*\*q, n\*(2\*\*q), 1,1)

def weak\_s\_2(q):

return t(1,n\*(2\*\*q), 1,2) / t(2\*\*q, n\*(2\*\*q), 1,2)

def weak\_s\_3(q):

return t(1,n\*(2\*\*q), 1,3) / t(2\*\*q, n\*(2\*\*q), 1,3)

This function is a formula to measure the speedup of the multi-core calculation. It was also found in the example .ipynb provided in class. It has been modified however to account for the week scalability speedup as apposed to the strong scalability speedup that was calculated in the example file shared in class. In this case the problem size that is being passed into the timing function is changed to have a relation with how many processors are designated to be used. We can see the use of these two functions in the next segment of code.

x = np.linspace(0, 9, 100)

y3 = [weak\_s\_3(q) for q in x]

y2 = [weak\_s\_2(q) for q in x]

y1 = [weak\_s\_1(q) for q in x]

y0 = [weak\_s\_0(q) for q in x]

Here we are getting 100 values between 0 and 9 that represent the number of processors being tested for scalability. 0 to 9 represents the p values of 1, 2, 4, 6, 8, 16, 32, 64, 126, 258, 512 when passed into the speedup function. They are calculated using different communication costs from 0 to 1 to show the change in how much a communication cost effects the speedup. In the case of the plots that are found on slide 17 of the lecture notes the only speedup plot that is shown is the one that has communication cost of 3 however, so for this report I will only display the plot with a communication cost of 3. The other three communication cost plots of 2, 1, and 0 are in the p3.ipynb attached to this assignment.

import matplotlib as mpl

import matplotlib.pyplot as plt

%matplotlib inline

# Plot Speedup vs # of processors for 3 comm.cost

plt.plot(x, y3, '-r', label = "0 comm. unit")

plt.ylabel("Speedup")

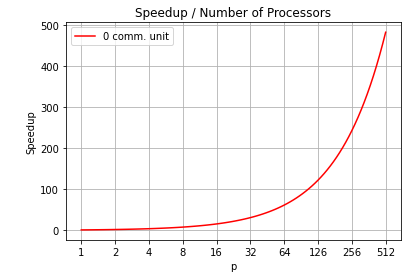
plt.xlabel("p")

plt.legend()

plt.title("Speedup / Number of Processors")

plt.grid(True)

plt.xticks([0,1,2,3,4,5,6,7,8,9], ['1','2','4','8','16','32','64','126','256','512']) # positions and labels



This function seen below calculates the efficiency of the week scalability analysis of the question. This efficiency equation assumes that the communication cost of that we are analyzing is 3 which is what is seen in the slides from the notes.

# Efficiency (for n =1024, comm. cost = 3)

def efficiency(q) :

return (s\_3(q) \* 100.0) / (2\*\*q)

processors = [0,1,2,3,4,5,6,7,8,9]

speedupEfficiency = []

for q in processors:

speedupEfficiency.append(efficiency(q))

speedupEfficiency

# Efficiency scalability bar plot

fig = plt.figure()

ax = fig.add\_axes([0, 0, 1, 1])

ax.bar(processors, speedupEfficiency)

ax.set\_ylabel("Efficiency in %")

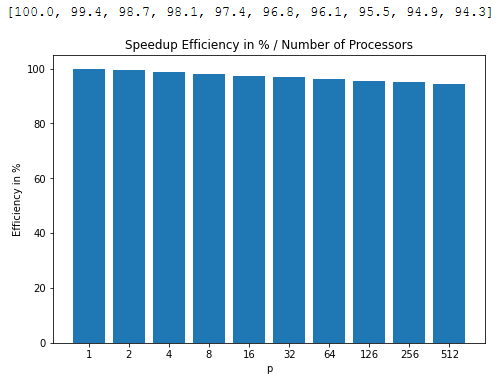
ax.set\_xlabel("p")

ax.set\_title("Speedup Efficiency in % / Number of Processors")

ax.set\_xticks(ex)

ax.set\_xticklabels(['1','2','4','8','16','32','64','126','256','512'])

[round(num, 1) for num in speedupEfficiency]



4(30 points) Study slides 18-22 (the general case of computing sum using *p* PEs) of   
 01\_Introduction PowerPoint. Write a Python notebook called, p4.ipynb, to perform   
 the following tasks:  
  
 (4.1) Define a function to return the number of PEs, which has a largest speedup given   
 *γ* (gamma) and *n* (number of elements). List the function and show the function  
 return value for *γ* = 1/3 and *n* = 1024 in your project report. (See slide 22)

import numpy as np

# function to return the optimal number of PEs

def optimalPE(y, k):

# optimalPE returns numer of PEs

# y (gamma)

# k exponent value for the n number of elements

optiaml = ((y\*np.log(2))/(2+y))\*2\*\*k

return optimal

k=10

y=1/3

pe = optimalPE(y, k)

print(pe)

math.log(pe, 2)

101.39753041334056

6.663878704997498

(4.2) Plot the function defined in (4.1) for *n* = 1024. The x-axis of the plot is *γ*   
 (gamma) in the interval (0, 1] and the y-axis is *p* (number of PEs). Make your plot   
 pretty and informative.  
  
 Show the plot in your project report.

import matplotlib as mpl

import matplotlib.pyplot as plt

%matplotlib inline

# Plot Number of PEs vs gamma

plt.plot(x, y, '-r', label = "number of PEs based on gamma")

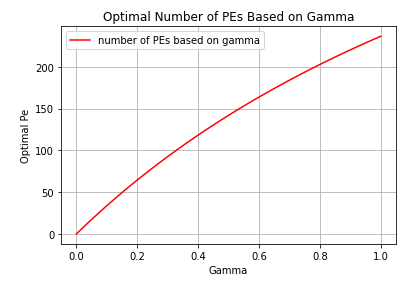
plt.ylabel("Optimal Pe")

plt.xlabel("Gamma")

plt.legend()

plt.title("Optimal Number of PEs Based on Gamma")

plt.grid(True)



(4.3)(Extra-credit) Do a 3D plot of the following 2-variable function,   
  
 *F(γ,q)* = *Sγ(p, n)*, for *n* = 1024,

as shown on slide 18 of 01\_Introduction PowerPoint.

Below is the python code used to try and calculate and print a 3D graph close to what was seen in the slides.

import numpy as np

# function to return the optimal number of PEs

def optimalSpeedUp(y, k, q):

# optimalPE returns numer of PEs

# y (gamma)

# k exponent value for the n number of elements

# q is the exponent value for optimal PEs

optiamlSpeed = ((y\*(2\*\*k - 1))/(2\*q + y\*(2\*\*(k-q) - 1 + q)))

return optiamlSpeed

import math

k=10

x = np.linspace(0, 0.35, 100)

x = np.delete(x, 0)

q = [math.log(optimalPE(g, k), 2) for g in x]

speedup = [optimalSpeedUp(g, k, math.log(optimalPE(g, k), 2)) for g in x]

# Plot Number of PEs vs gamma

ax = plt.axes(projection='3d')

ax.plot3D(x, speedup, q, '-r', label = "optimal speedup")

ax.set\_ylabel("F(gamma,q)")

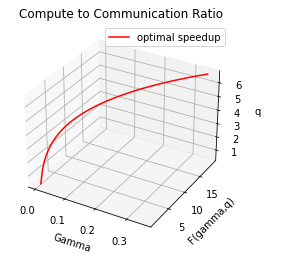
ax.set\_xlabel("Gamma")

ax.set\_zlabel("q")

ax.legend()

ax.set\_title("Compute to Communication Ratio")

ax.grid(True)



5(10 points) Google it to find the top supercomputer in June 2021. Give the name of the   
 supercomputer and describe the CPU used and its basic configuration. Cite the   
 references of your answer.

Fugaku was the number 1 fastest supercomputer in the world in June 2021. It had 7630848 cores which it used for processing. It CPU is based on the [ARM](https://en.wikipedia.org/wiki/ARM_architecture) [version 8.2A](https://en.wikipedia.org/wiki/AArch64#ARMv8.2-A) processor architecture and adopted scalable vector extensions. This is a new optional extension to the ARM CPU being used which allows for specific high-performance computing of scientific workloads specifically using vectors. Its CPU is called the A64FX CPU and it has four core memory groups of which each has 13 cores that only 12 are used for computing and the last one is used as an assistant core. It has a level 2 cache and memory controller as well as a ring bus network on a chip that is used to connect them with the Tofu Interconnect D interface and PCI Express interface.

<https://www.top500.org/lists/top500/2021/06/>

<https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article03.html>

<https://www.fujitsu.com/global/about/innovation/fugaku/specifications/>