



Xi'an Jiaotong-Liverpool University

西交利物浦大學

XJTLU Entrepreneur College (Taicang) Cover Sheet

Module code and Title	DTS205TC High Performance Computing	
School Title	School of AI and Advanced Computing	
Assignment Title	Assessment 2 – Lab Report	
Submission Deadline	April 22nd, 2024 @ 23:59	
Final Word Count	N/A	
If you agree to let the university use your work anonymously for teaching and learning purposes, please type “yes” here.		Yes

I certify that I have read and understood the University’s Policy for dealing with Plagiarism, Collusion and the Fabrication of Data (available on Learning Mall Online). With reference to this policy I certify that:

- My work does not contain any instances of plagiarism and/or collusion.
My work does not contain any fabricated data.

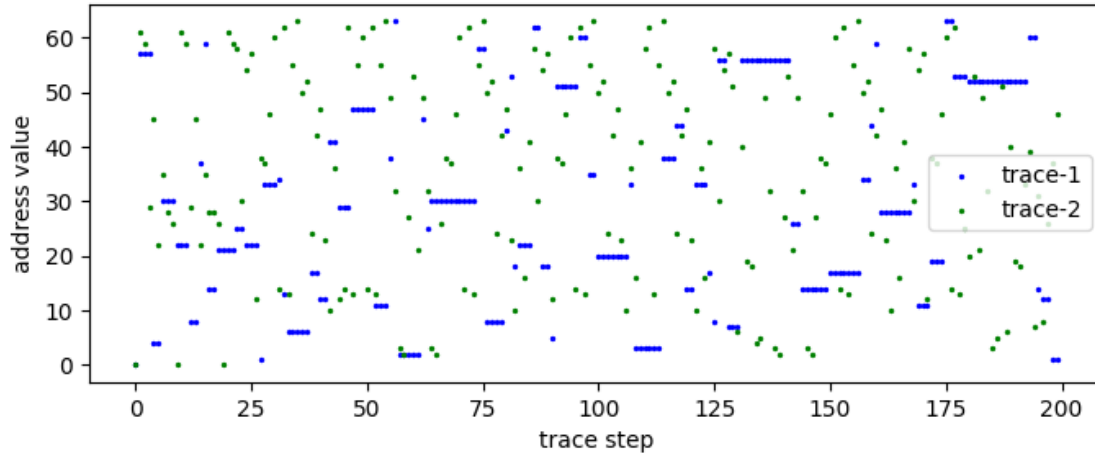
By uploading my assignment onto Learning Mall Online, I formally declare that all of the above information is true to the best of my knowledge and belief.

Scoring – For Tutor Use					
Student ID			2142116		
Stage of Marking	Marker Code	Learning Outcomes Achieved (F/P/M/D) (please modify as appropriate)			Final Score
		A	B	C	
1 st Marker – red pen					
Moderation – green pen	IM Initials	The original mark has been accepted by the moderator (please circle as appropriate):			Y / N
		Data entry and score calculation have been checked by another tutor (please circle):			Y
2 nd Marker if needed – green pen					
For Academic Office Use		Possible Academic Infringement (please tick as appropriate)			
Date Received	Days late	Late Penalty	<input type="checkbox"/> Category A		Total Academic Infringement Penalty (A,B, C, D, E, Please modify where necessary) _____
			<input type="checkbox"/> Category B		
			<input type="checkbox"/> Category C		
			<input type="checkbox"/> Category D		
			<input type="checkbox"/> Category E		



Q1:

1) Firstly, the first 200 memory accesses of the two datasets, trace1 and trace2, were selected for visualization as shown in the figure below.



Running the given program, we obtained the data in the following table. For trace1, both the Random and FIFO policies exhibited good cache hit rates, and this good cache access was almost independent of the cache size. As shown in the figure, this was due to the characteristic of trace1 having multiple consecutive requests for the same address, which made subsequent memory requests, except for the first one, always hit the cache, thereby achieving a high hit rate independent of the cache size. On the other hand, the memory requests in trace2 showed no obvious pattern, and both the Random and FIFO policies exhibited extremely low cache hit rates, with the hit rates increasing as the cache size increased. Upon careful comparison of the two policies, it was found that the randomness of the Random policy was better able to adapt to the scattered memory access pattern of trace2, resulting in a slightly better cache hit rate than the FIFO .

Cache size		1	2	3	4	5
Trace1	Random	0.6105	0.6158	0.6221	0.6287	0.6345
	FIFO	0.6105	0.6162	0.6228	0.6290	0.6350
	RP	0.6105	0.6168	0.6230	0.6269	0.6332
Trace2	Random	0.0018	0.0080	0.0235	0.0438	0.0712
	FIFO	0.0018	0.0072	0.0142	0.0181	0.0231
	RP	0.0018	0.0142	0.0264	0.0462	0.0760

The visualization code is as follows:



```
# the first 200-step traces for visualization
x_line = range(200)
plt.figure(figsize=(8, 3))
plt.scatter(x_line, traces[0][:200], label='trace-1', s=2, color='blue')
plt.scatter(x_line, traces[1][:200], label='trace-2', s=2, color='green')
plt.legend()
plt.xlabel('trace step')
plt.ylabel('address value')
plt.savefig('lab1_trace.png')
plt.show()
```

2) As mentioned in the first question analysis, randomness seemed to be more suitable for the addressing pattern of trace2. Therefore, a Random Probability Policy (RP) was designed. This policy differs from the original Random policy in that, when the cache needs to be updated, it no longer discards items with equal probability. Instead, it normalizes the time since the last hit for each address in the cache into a probability and updates the cache based on these probabilities. If an address in the cache has not been hit for a longer time, it has a higher probability of being discarded. To implement this, a time list is used to track time information, and the initial time is set to 1 to avoid using 0 as a divisor. The main code for the implementation is included, and the cache hit performance of the Random Probability Policy in trace2 is also shown in the original table. By introducing the last hit time as a basis for probability, the Random policy is given a time dependency, improving its cache hit performance and surpassing the original two policies.

Implementation for Random Probability Policy:

```
# Random Probability Policy
class RPPolicy:
    def __init__(self, size):
        self.size = size
        self.cache = []
        self.time = []
        self.name = 'RP'

    def access(self, current):
        if current in self.cache:
            # update time + 1 for last cache hit except current
            self.time = [t + 1 for t in self.time]
            index = self.cache.index(current)
            # 1 set to start time for current cache hit, avoid 0 sum for probability computing
            self.time[index] = 1
            return True
        else:
            if len(self.cache) > self.size:
                assert False
            elif len(self.cache) == self.size: # full
                # normalize time to probability
                hit_time = 1 / np.array(self.time)
                hit_prob = hit_time / np.sum(hit_time)
                drop_index = np.random.choice(range(self.size), p=hit_prob)
                del self.cache[drop_index]
                del self.time[drop_index]
            self.cache.append(current)
            self.time.append(1)
            return False
```



Q2: The following two questions rely primarily on the functions provided by the networkx network analysis toolkit.

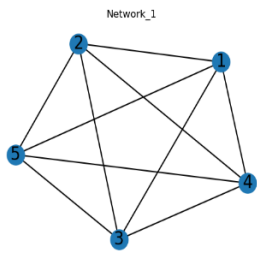
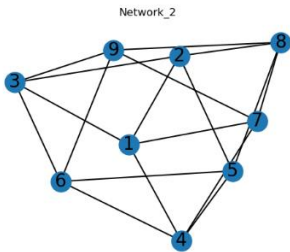
1) Network Visualization: The visualization function draw() provided by the networkx is used to convert the given data into its corresponding graph format and visualize it on a given matplotlib canvas. The core code for the visualization is as follows:

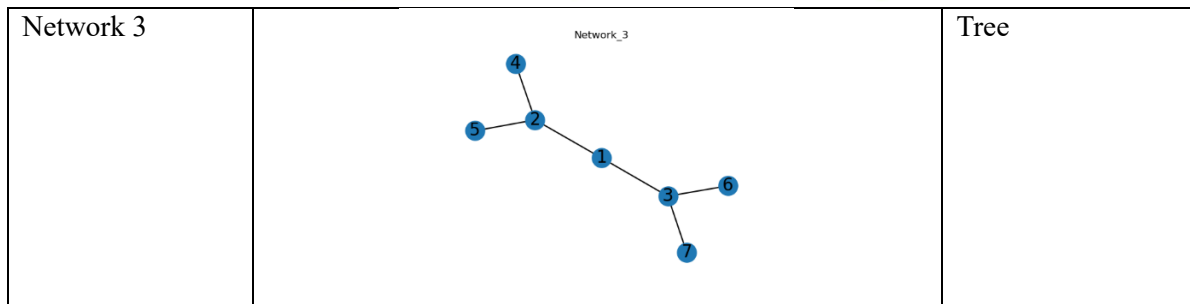
```
# Load bandwidth and latency data from CSV files
bandwidth = pd.read_csv(f"./lab2/{name}_Bandwidth.csv", header=None).to_numpy()
latency = pd.read_csv(f"./lab2/{name}_Latency.csv", header=None).to_numpy()
N = bandwidth.shape[0] # Number of nodes in the network

edges, band_s, laten_s = list(), list(), list() # Lists to store edge information, bandwidth, and latency
for i in range(N):
    for j in range(N):
        if not np.isnan(bandwidth[i, j]):
            # node start from 1
            edges.append([i + 1, j + 1])
            band_s.append(bandwidth[i, j])
            laten_s.append(latency[i, j])
            # Create a networkx directed graph object
G = networkx.Graph()
# add nodes
G.add_nodes_from(range(1, N + 1))
# add edges with weight
for edge, band, laten in zip(edges, band_s, laten_s):
    G.add_edge(edge[0], edge[1], bandwidth=band, latency=laten)
graphs.append(G)
print(f'{name} has {G.number_of_nodes()} nodes and {G.number_of_edges()} edges')

# visualization network
fig, ax = plt.subplots()
networkx.draw(G, ax=ax, node_size=500, width=1.5, with_labels=True, font_size=20)
plt.title(f'{name}')
plt.savefig(f'./lab2/{name}.png', dpi=100)
```

The visualization results and the inferred network topology types based on the results are shown in the table below:

Network	Visualization	Topology
Network 1		Fully-connected
Network 2		2D-Mesh



2) Shortest delay: To find the shortest delay and its path from node 1 to node 5 on a specified network, we need to search for the shortest weighted path in a weighted directed graph where latency is used as the edge weight. This can be achieved by utilizing the `dijkstra_path` function from the `networkx` library, which implements the Dijkstra's algorithm to find the shortest path. The main code for this implementation is as follows:

```
# calculate the shortest path using latency as the weight
source = 1
target = 5
path = networkx.dijkstra_path(G, source, target, weight='latency')
min_latency = sum([G[path[i]][path[i + 1]]['latency'] for i in range(len(path) - 1)])
print(f"{name} shortest delay")
print(f"Path: {path} \t Latency: {min_latency:.3f}")
```

Maximum Throughput: Unlike delay, which for each path is equal to the sum of the latencies of all edges on that path, throughput is defined as the minimum bandwidth of any edge along that path. In this implementation strategy, we identify all simple paths from Node 1 to Node 5, compute the throughput for each such path, and ultimately determine the Maximum Throughput along with the corresponding path.

Key code:

```
# calculate the maximum throughput
flow, cost = networkx.maximum_flow(G, source, target, capacity='bandwidth')
```

The results for the three networks are also presented in the table below:

Network	Network 1		Network 2		Network 3	
Shortest delay	0.298	[1, 3, 5]	0.6780	[1, 3, 2, 5]	1.781	[1, 2, 5]
Maximum throughput	242		145		86	

Output:

```
Network_1 has 5 nodes and 10 edges
Network_1 shortest delay
Path: [1, 3, 5] Latency: 0.298
Network_1 maximum throughput 242.0 {1: {2: 86.0, 3: 16.0, 4: 48.0, 5: 92.0}, 2: {1: 0, 3: 0, 4: 0, 5: 89.0}, 3: {1: 0, 2: 3.0, 4: 0, 5: 13.0}, 4: {1: 0, 2: 0, 3: 0, 5: 48.0}, 5: {1: 0, 2: 0, 3: 0, 4: 0}}
Network_2 has 9 nodes and 18 edges
Network_2 shortest delay
Path: [1, 3, 2, 5] Latency: 0.678
Network_2 maximum throughput 145.0 {1: {2: 86.0, 3: 0, 4: 53.0, 7: 25.0}, 2: {1: 0, 3: 0, 5: 3.0, 8: 87.0}, 3: {1: 19.0, 2: 4.0, 6: 0, 9: 0}, 4: {1: 0, 5: 92.0, 6: 6.0, 7: 0}, 5: {2: 0, 4: 0, 6: 0, 8: 0}, 6: {3: 23.0, 4: 0, 5: 43.0, 9: 0}, 7: {1: 0, 4: 45.0, 8: 0, 9: 13.0}, 8: {2: 0, 5: 7.0, 7: 33.0, 9: 47.0}, 9: {3: 0, 6: 60.0, 7: 0, 8: 0}}
Network_3 has 7 nodes and 6 edges
Network_3 shortest delay
Path: [1, 2, 5] Latency: 1.781
Network_3 maximum throughput 86.0 {1: {2: 86.0, 3: 0}, 2: {1: 0, 4: 0, 5: 86.0}, 3: {1: 0, 6: 0, 7: 0}, 4: {2: 0}, 5: {2: 0}, 6: {3: 0}, 7: {3: 0}}
```



Q3:

1) This parallel program follows a master-slave pattern, where the master sends data to the slaves, and upon completion of sampling, the slaves transmit their data back to the master. Under the assumption of point-to-point blocking communication and a transmission latency of $L = 0$ seconds, the communication time is equal to the total amount of communicated data divided by the network bandwidth:

$$t = \frac{C}{B} = \frac{C_1 + C_2}{B} = \frac{4D(P-1) + 4D\frac{N}{P}(P-1)}{B} = 4D(P-1)\frac{P+N}{BP} \quad (s)$$

2)

Mathematical basis

Let X_1, X_2, \dots, X_P represent distinct data segments of a dataset, where each segment X_i contains n_i observations. The mean of each segment X_i is defined as:

$$\bar{X}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} X_{ij}$$

where X_{ij} denotes the j -th observation in the i -th segment.

The overall mean \bar{X} of the dataset, combining all segments, can then be expressed as a weighted average of the segment means:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^P n_i \bar{X}_i$$

where $N = \sum_{i=1}^P n_i$ is the total number of observations across all segments.

Linearity of Expectation

This characteristic enables each process in a distributed system to calculate a partial mean on its local data. When these partial means are averaged, the result is equivalent to what you'd get if the mean were computed across the entire dataset in a centralized manner. By sending only the means instead of entire resampled datasets, the total volume of data transmitted across the network is significantly decreased.

The changes are founded on the principle of linearity of expectation, a core concept in probability theory, which states that the expected value of a sum of random variables is equivalent to the sum of their individual expected values. This can be mathematically represented as:

$$E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i]$$

Only modifying the content within the red box, since the sampling portion is already fixed and the size of the data after sampling by each process is predetermined, the objective is to reduce the communication volume during the second communication by processing and compressing this sampled data. Considering that the main purpose of the program is to calculate the variance of sample means, and each process already possesses multiple samples of data, it would be entirely feasible for each process to first compute its own sample mean, followed by collecting these means at the master for the subsequent variance calculation. At this point, the communication volume will be significantly reduced, and the new formula for calculating communication time becomes:

$$t = \frac{C}{B} = \frac{C_1 + C'_2}{B} = \frac{4D(P-1) + 4D\frac{N}{P}(P-1)}{B} = 4(P-1)\frac{DP+N}{BP} \quad (s)$$

The modified code is as follows:



```
def question2():  
    if R == 0: # master  
        # generate data -- this is only for test!  
        data = np.arange(D)  
        # send data  
        for i in range(1, P):  
            comm.send(data, dest=i)  
        # receive samples  
        samples = bootstrap(data) # do my part  
  
        # changed to calculate mean before communication  
        mean = np.mean(samples, axis=1)  
        for i in range(1, P):  
            mean = np.vstack((mean, comm.recv(source=i)))  
  
        result = np.var(mean)  
        # output  
        print(f'proc {R}: var={result}')  
    else: # slave  
  
        # receive data  
        data = comm.recv(source=0)  
        # resample  
        samples = bootstrap(data)  
        # calculate mean  
        mean = np.mean(samples, axis=1)  
        # send mean back  
        comm.send(mean, dest=0)  
        print(f'proc {R}: done!')
```



Q4:

Mathematical basis

When memory capacity restricts each process from performing a full sampling of the data, it is necessary for all processes to collaborate in completing one sampling, with each process sampling a D/P -sized portion of the data. To compute the mean of this sample, we can use the following formula:

$$m = \frac{n_1 + n_2 + \dots + n_D}{D} = \frac{n_1 + \dots + n_D}{D} \frac{1}{P} + \dots + \frac{n_{\frac{(P-1)N}{P}+1} + \dots + n_D}{D}$$

That is, each process computes the sum of its partial sampled data, divides it by D , and then all processes' results are collected and summed to obtain the mean of the sample. The main skeleton of its implementation is as follows:

```
# changed to sample partial data for each sample and get the sum / D
# to reduce the memory usage
def bootstrap_changed(data):
    sum_D = np.zeros(shape=(N,))
    for i in range(N):
        sample = np.random.choice(data, size=len(data), replace=True)
        sum_D[i] = np.sum(sample) / D
    return sum_D
if R == 0: # master
    # generate data -- this is only for test!
    data = np.arange(round(D/P))
    # resample
    sum_D = bootstrap_changed(data)
    # the sum of all sum_D is the mean of the samples
    for i in range(1, P):
        sum_D = sum_D + comm.recv(source=i)
    # calculate the variance
    result = np.var(sum_D)
    # output
    print(f'proc {R}: var={result}')
else: # slave
    # generate data -- this is only for test!
    data = np.arange(round(D/P))
    # resample
    sum_D = bootstrap_changed(data)
    comm.send(sum_D, dest=0)
    print(f'proc {R}: done!')
```

Under this implementation, the master only needs to collect N -sized data from all slaves once, with each process requiring N samples. Given the specified constraints, the estimated formula for the program's runtime is:

$$t = t_{com} + t_{sam} = \frac{4N(P-1)}{B} + \frac{N}{S} (s)$$



Q5:

To implement this requirement from a loop perspective, we first need to iterate over the dictionary representing the graph's node pointers, identifying the list of nodes pointed to by each node. Then, we must iterate over these lists to count the occurrences of each node.

To restructure this approach, we begin by concatenating these lists, eliminating the outermost loop. Here, we use the *sum* function to concatenate all pointer lists. Next, we perform the frequency counting using the *reduce* function. We define a dictionary to store the node frequencies and use *reduce* to iterate over the concatenated list, updating the frequency dictionary accordingly. Finally, we employ the *sorted* function to sort this frequency dictionary based on the frequency values.

Main code structure like this:

```
import pickle
from functools import reduce

with open('./lab5/links.pkl', 'rb') as f:
    links = pickle.load(f)

# concatenate the links target nodes into a list
target_nodes = sum(links.values(), [])

# count the frequency of each node
# from collections import defaultdict
# counter = defaultdict(int)
# counter = reduce(lambda x, y: counter.update({y: x[y] + 1}) or x, target_nodes, counter)
# count the frequency of each node
counter = {}
counter = reduce(lambda x, y: x.update({y: x.get(y, 0) + 1}) or x, target_nodes, counter)

# Sort the nodes by their frequency
linkin_count = sorted(counter.items(), key=lambda x: x[1], reverse=True)

# Output the top 10 nodes
for node, count in linkin_count[:10]:
    print(f"Page {node} is referred to {count} times.")
```

And the output is:

```
Page 579 is referred to 22 times.
Page 433 is referred to 22 times.
Page 46 is referred to 20 times.
Page 772 is referred to 19 times.
Page 32 is referred to 19 times.
Page 713 is referred to 19 times.
Page 156 is referred to 19 times.
Page 181 is referred to 19 times.
Page 744 is referred to 18 times.
Page 3 is referred to 18 times.
```