Homework 3

Md Ali A20439433
CS 546: Parallel and Distributed Processing

October 16, 2021

Exercise 1. Your company has bought a new 8-core processor, and you have been asked to optimize your software for this processor. You will run two applications on this 8-core processor, but the resource requirements are not equal. The first application needs 80% of the resources, and the other only 20% of the resources.

a) Given that 40% of the first application is parallelizable, how much speedup would you achieve with at application if run in isolation?
b) Given that 99% of the second application is parallelizable, how much speedup would this application observe if run in isolation?
c) Given that 40% of the first application is parallelizable, how much overall system speedup would you observe if you parallelized it?
d) Given that 99% of the second application is parallelizable, how much overall system speedup would you get?
e) If we follow fixed-time scalable up principle, and assume only parallelizable portion will scale up in size, what is the fixed-time speedup of question a) and b), respectively?
f) If we further assume that the applications are dense matrix computations, that is the memory requirement increases with n^2 and computation increases with n^3. Repeat e) for memory bounded speedup.

Proof. Below are each seprarte parts.

a. We will utilize Amdhal's Law where we have that $\textit{speed up} = \dfrac{1}{(1 - A_p) + \dfrac{A_p}{N}}$, where N is the number of processors, A_p is the application that is being parallel computed. So we get that.

$$\textit{speed up} = \frac{1}{(1 - 0.4) + \dfrac{0.4}{8}} \approx \boxed{1.54}$$

b. We will do similarly what we did above but have $A_p = 0.99$.

$$\textit{speed up} = \frac{1}{(1 - 0.99) + \dfrac{0.99}{8}} \approx \boxed{7.47}$$

c. Utilizing the same idea, but instead we will have to take into consideration that the other application is utilizing resources in the background. So we get that

$$speed\ up = \frac{1}{0.2 + 0.8((1 - 0.4) + \frac{0.4}{8}} \approx \boxed{1.39}$$

d. Similarly as we did in part c, we can do for part d.

$$speed\ up = \frac{1}{0.8 + 0.2((1 - 0.99) + \frac{0.99}{8}} \approx \boxed{1.21}$$

e. We have two parts in this part e. We will utilize the fixed-time speedup equation which is stated below.

$fixed - time\ speed\ up = f + N(1 - f)$, where f is the fixed part and N is still the number of processors. So knowing this we can solve for our two parts.

e-1. For part a we have that

$$fixed - time\ speed\ up = 0.6 + 8(1 - 0.6) = \boxed{3.8}$$

e-2. For part b we have that

$$fixed - time\ speed\ up = 0.01 + 8(1 - 0.01) = \boxed{7.93}$$

f. So we're asked to repeat part e for memory bounded speed up. So for memory bound speed up we can have that

$speed\ up = \frac{f + N^{3/2}(1 - f)}{f + \frac{N^{3/2}}{N}(1 - f)}$, so we can now calculate for both our parts.

f-1. For part a we have that

$$speed\ up = \frac{0.6 + 8^{3/2}(1 - 0.6)}{0.6 + \frac{8^{3/2}}{8}(1 - 0.6)} \approx \boxed{5.57}$$

f-2. For part b we have that

$$speed\ up = \frac{0.01 + 8^{3/2}(1 - 0.01)}{0.01 + \frac{8^{3/2}}{8}(1 - 0.01)} \approx \boxed{7.98}$$

□

Exercise 2. Prove Amdahl's law.

Amdahl's law state that in a parallel computing that if f is the proportion that remains serial, and $1 - f$ is the part of a system that is made parallel then the maximum speed up we can achieve give N number of processors is

$$speed\ up = \frac{1}{f + (1 - \frac{f}{N})}$$

Amdahl's law also gives us a limit on speed in terms of f such that

$$S_p = \frac{T_S}{T_p} = \frac{T_S}{f \cdot T_S + \frac{(1-f)T_S}{p}} = \frac{1}{f + \frac{1-f}{p}}$$

So if we assume that the serial fraction is fixed, then the speed up for infinite amount of processors p will go off to $1/\infty$ which will be 0 so we are left with that

$$\boxed{\lim_{p \to \infty} S_p = \frac{1}{f}}$$

Proof. □

Exercise 3. Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as percentage of the execution time when the enhanced mode is in use. Recall that Amdahl's Law depends on the fractions of the original, non-enhanced execution time that could make use of enhanced mode. Thus, we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.

a) What is the speedup we have obtained from fast mode?
b) What percentage of the original execution time has been converted to fast mode?

Proof. The two parts are below.

a. To computer the speed up we must first find the execution time with the enhancement involved. So we know the two parts have execution times of each 50%, so the half that doesn't have the enhancement would still take 50% but the half with the enhancement would be 550%. So we would get below the overall speed up.

$$speed\ up = \frac{execution\ time\ enhanced}{execution\ time\ not\ enhanced} = \frac{550\%}{100\%} = \boxed{5.5}$$

b. To find the percentage of the original execution time that was converted to fast mode we will utilize Amdahl's law again

$$Percentage = \frac{(speed\ up\ overall \cdot factor\ speed\ up) - factor\ speed\ up}{(speed\ up\ overall \cdot factor\ speed\ up) - speed\ up\ overall} = \frac{(5.5 \cdot 10) - 10}{(5.5 \cdot 10) - 5.5} = \boxed{90.90\%}$$

□

Exercise 4. Calculate the speedup of the three models of parallel speedup assuming the fraction that cannot be parallelized is a = 0.2, and the number processors is p = 20.

a) Amdahl's Law
b) Gustafson's Law
c) Sun/Ni's law, assuming $G(20) = 20^{3/2}$

Proof. We are given that a = 0.2 and p = 20

a. Using Amdahl's Law we get that

$$\frac{1}{x + (\frac{1-x}{N})} = \frac{1}{0.2 + (\frac{1-0.2}{20})} = \boxed{4.17}$$

b. Using Gustafson's Law we get that

$$f + N(1-f) = 0.2 + 20(1-0.2) = \boxed{16.2}$$

c. Sun/Ni's law, assuming $G(20) = 20^{3/2}$

$$\frac{f + N^{3/2}(1-f)}{f + \frac{N^{3/2}}{N}(1-f)} = \frac{0.2 + 20^{3/2}(1-0.2)}{0.2 + \frac{20^{3/2}}{20}(1-0.2)} \approx \boxed{18.99}$$

□

Exercise 5. What are the four steps to design a parallel algorithm? What are the five methods/patterns of parallel algorithms which are introduced in class.

Proof. The four steps to designing a parallel algorithm are below

1. Partitioning
2. Communication
3. Agglomeration
4. Mapping

The five methods of parallel algorithms are below

1. Partition Method

2. PPT Algorithm
3. PDD Algorithm
4. LU Pipelining Algorithm
5. PTH Method and PPD Algorithm

□

Exercise 6. Please give the fixed-size, fixed-time, and memory-bounded speedup formula for the general PPT (non-pivoting) tridiagonal solver where the communication overhead is considered. Here general means that the algorithm, so the formula, is general for any given number of processors and for any n by n matrix.

Proof. We are asked to formulate for fixed-size, fixed-time, and memory-bounded speedup for a PPT tridiagonal solver.

Below is a fixed-sized formula that leverages Amdahl's law that limited on speed up in terms of α

$$S_{fs} = \frac{T_S}{T_p} = \frac{T_S}{\alpha \cdot T_S + \frac{(1-\alpha) \cdot T_S}{p}} = \boxed{\frac{1}{\alpha + \frac{1-\alpha}{p}}}$$

Below is the fixed-time speed up or Gustafson's Law

$$S_{ft} = \frac{Work(p)}{Work(1)} = \frac{\alpha W + (1-\alpha)pW}{W} = \boxed{\alpha + (1-\alpha)p}$$

Below is memory-bounded speed up or Sun and Ni's Law

$$S_{mb} = \frac{Work(p)/Time(p)}{Work(1)/Time(1)} = \boxed{\frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)G(p)/p}}$$

□

Exercise 7. Please give the fixed-size, fixed-time, and memory-bounded speedup formula for the general PDD tridiagonal solver where the communication overhead is considered. Here general means that the algorithm, so the formula, is general for any given number of processors and for any n by n matrix.

Proof. We are asked to formulate for fixed-size, fixed-time, and memory-bounded speedup for a PDD tridiagonal solver.

Below is a fixed-sized formula that leverages Amdahl's law that limited on speed up in terms of α

$$S_{fs} = \frac{T_S}{T_p} = \frac{T_S}{\alpha \cdot T_S + \frac{(1-\alpha) \cdot T_S}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Now we are asked with overhead so this would be even worse so with overhead we have that

$$S_{fs} = \frac{T_1}{\alpha \cdot T_1 + \frac{(1-\alpha)\cdot T_1}{p} + T_{overhead}} = \boxed{\frac{1}{\alpha + \frac{T_{overhead}}{T_1}} \text{ as } p \to \infty}$$

Below is the fixed-time speed up or Gustafson's Law

$$S_{ft} = \frac{Work(p)}{Work(1)} = \frac{\alpha W + (1-\alpha)pW}{W} = \alpha + (1-\alpha)p$$

But again we are interested in the overhead so this would also be worse so with overhead we have that

$$S_{ft} = \frac{Work(p)}{Work(1)} = \boxed{\frac{\alpha + (1-\alpha)p}{1 + \frac{T_{overhead}}{T_1}}}$$

Below is memory-bounded speed up or Sun and Ni's Law. In this case I don't think overhead would apply.

$$S_{mb} = \frac{Work(p)/Time(p)}{Work(1)/Time(1)} = \boxed{\frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)G(p)/p}}$$

□

Exercise 8. In the parallel formulations of bitonic sort, we assumed that we had p processors available to sort n items. Show how the algorithm needs to be modified when only p/2 processors are available.

Proof. Assuming that we have p processors than we have to dived the n items into n/p size of p blocks, but we are given only $p/2$ or half the processors that would be available. Hence, if we still divided the items into blocks of n/p we would only cover $n/2$ items which is not what we want because we would need to effectively sort all n items. This can be done by diving the n items into blocks of size $2n/p$ so effective we would have that

$$\frac{2n}{p} \cdot \frac{p}{2} = n$$

So this way we have $p/2$ processors but we are still able to cover all the items. Hence result.

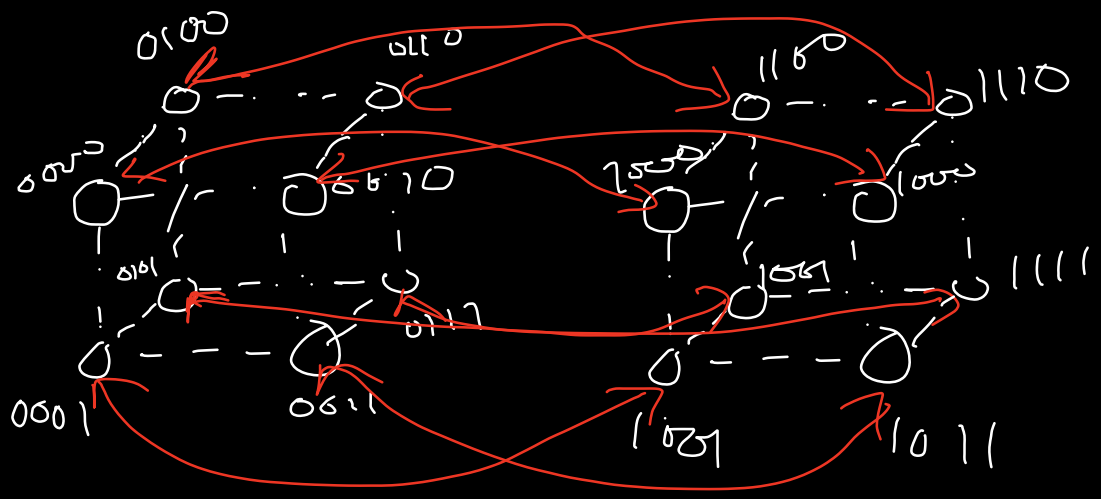
□

Exercise 9. Show that, in the hypercube formulation of bitonic sort, each bitonic merge of sequences of size 2^k is performed on a k -dimensional hypercube and each sequence is assigned to a separate hypercube.

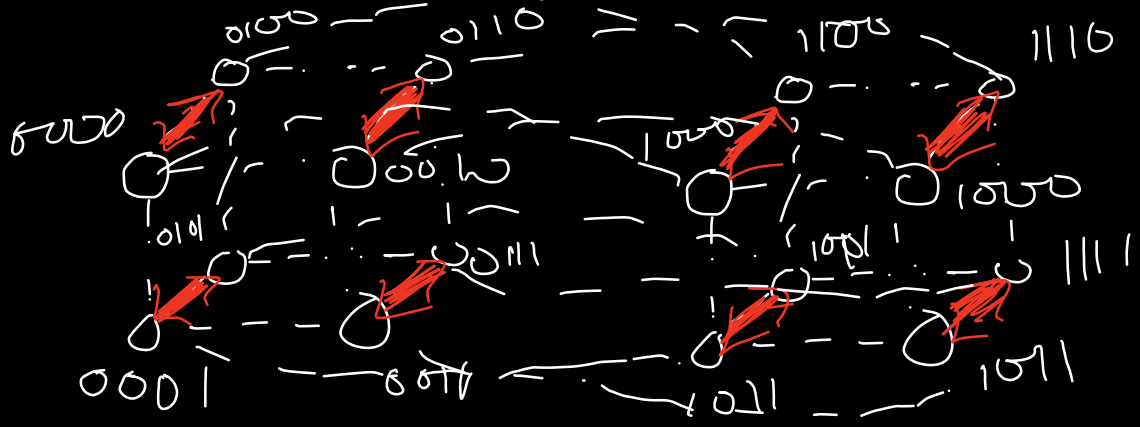
Proof. This will be shown in the following page in a written fashion, but I will also explain it here. So in a hypercube formulation the connection is based on a compare and exchange operations where the nodes differ only in one bit. This means that any nodes or processors that differ in one bit are neighbors, so we have a sequence of size 2^k such that the hybercube is k-dimensional, so during the first step the processors will only differ by a kth bit in a binary fashion. Utilizing the compare and exhange operation would take place between each processer along the kth dimension, in the second step the operation would take place along the (k-1)th dimension. So effectively we would get the sequence in i steps such that (d-(i-1))th dimension. A construction can be shown in the next few pages.

□

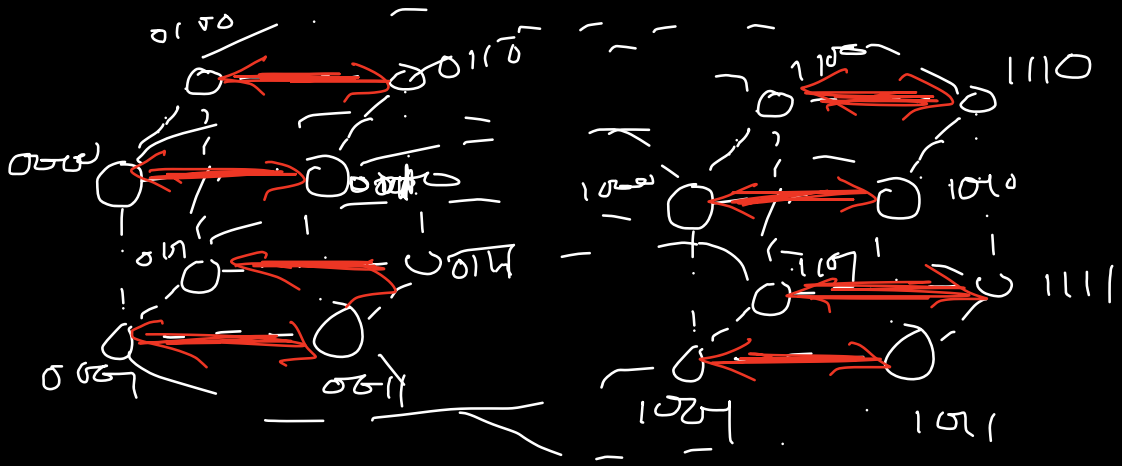
9/ Step 1



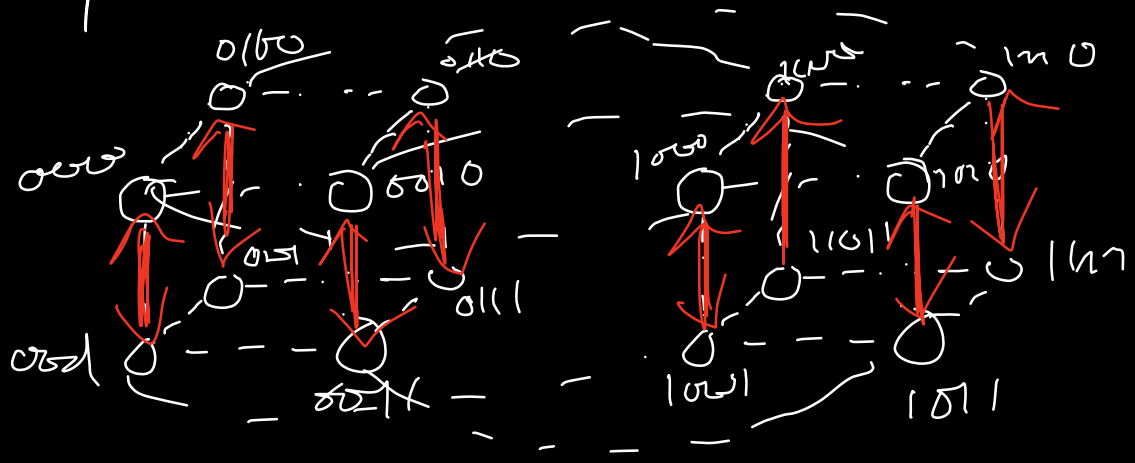
Step 2



Step 3


0100
0110
0000
0010
0101
0001
0011
1100
1110
1000
1010
1001
1011

Step 4


0100
0110
0000
0010
0101
0111
0001
1000
1010
1101
1011

Exercise 10. Consider the following parallel quicksort algorithm that takes advantage of the topology of a p-process hypercube connected parallel computer. Let n be the number of elements to be sorted and $p = 2^d$ be the number of processes in a d-dimensional hypercube. Each process is assigned a block of n/p elements, and the labels of the processes define the global order of the sorted sequence. The algorithm starts by selecting a pivot element, which is broadcast to all processes. Each process, upon receiving the pivot, partitions its local elements into two blocks, one with elements smaller than the pivot and one with elements larger than the pivot. Then the processes connected along the dth communication link exchange appropriate blocks so that one retains elements smaller than the pivot and other retains elements larger than the pivot. Specifically, each process with a 0 in the dth bit (the most significant bit) position of the binary representation of its process label retains the smaller elements, and each process with a 1 in the dth bit retains the larger elements. After this step, each process in the (d-1)-dimensional hypercube whose dth label bit is 0 will have elements smaller than the pivot, and each process in the other (d-1)-dimensional hypercube will have elements larger than the pivot. This procedure is performed recursively in each subcube, splitting the subsequences further. After d such splits – one along each dimension – the sequence is sorted with respect to the global ordering imposed on the processes. This does not mean that the elements at each process are sorted. Therefore, each process sorts its local elements by using sequential quicksort. This hypercube formulation of quicksort is shown in Algorithm 9.9. The execution of the algorithm is illustrated in Figure 9.21. Analyze the complexity of the hypercube-based parallel quicksort algorithm. Derive expressions for the parallel runtime, speedup, and efficiency. Perform this analysis assuming that the elements that were initially assigned at each process are distributed uniformly.

Proof. Here we need to find the time complexity of both the sequential and parallel quick sort algorithms and effective get a ratio. The blocked based algorithm explained in the problem for a hypercube with p processes is similar to the one element process case but instead we have p blocks of size n/p instead of having a fixed number of p elements. We would also need to replace the compare and exchange operation that we have been utilizing with the compare and split with each taking n/p computation time and n/p communication time. Knowing this we can see that the initially the processors with their n/p elements in time would be $((n/p)log(n/p))$ and the nthe performance would be $(log^2 p)$ compare and split steps. So the parallel timing of this formulation is

$$T_p = O(\frac{n}{p}log(\frac{n}{p})) + O(\frac{n}{p}log^2(p)) + O(\frac{n}{p}log^2(p))$$

The above equation is the local sort, comparison and communication all added up in complexity space. Now we also must include the sequential complexity which would be $nlogn$. So we would end up with the following ratio below.

$$O_{total} = \frac{Sequential\,Complexity}{Parallel\,Complexity}$$

So utilizing the above ratio with our sequential and parallel complexity counterpart we would get that

$$O_{total} = \frac{O(nlog(n))}{O(\frac{n}{p}log(\frac{n}{p})) + O(\frac{n}{p}log^2(p)) + O(\frac{n}{p}log^2(p))}$$

Simplifying this by rules of logarithmic functions we get that the overall total complexity would be

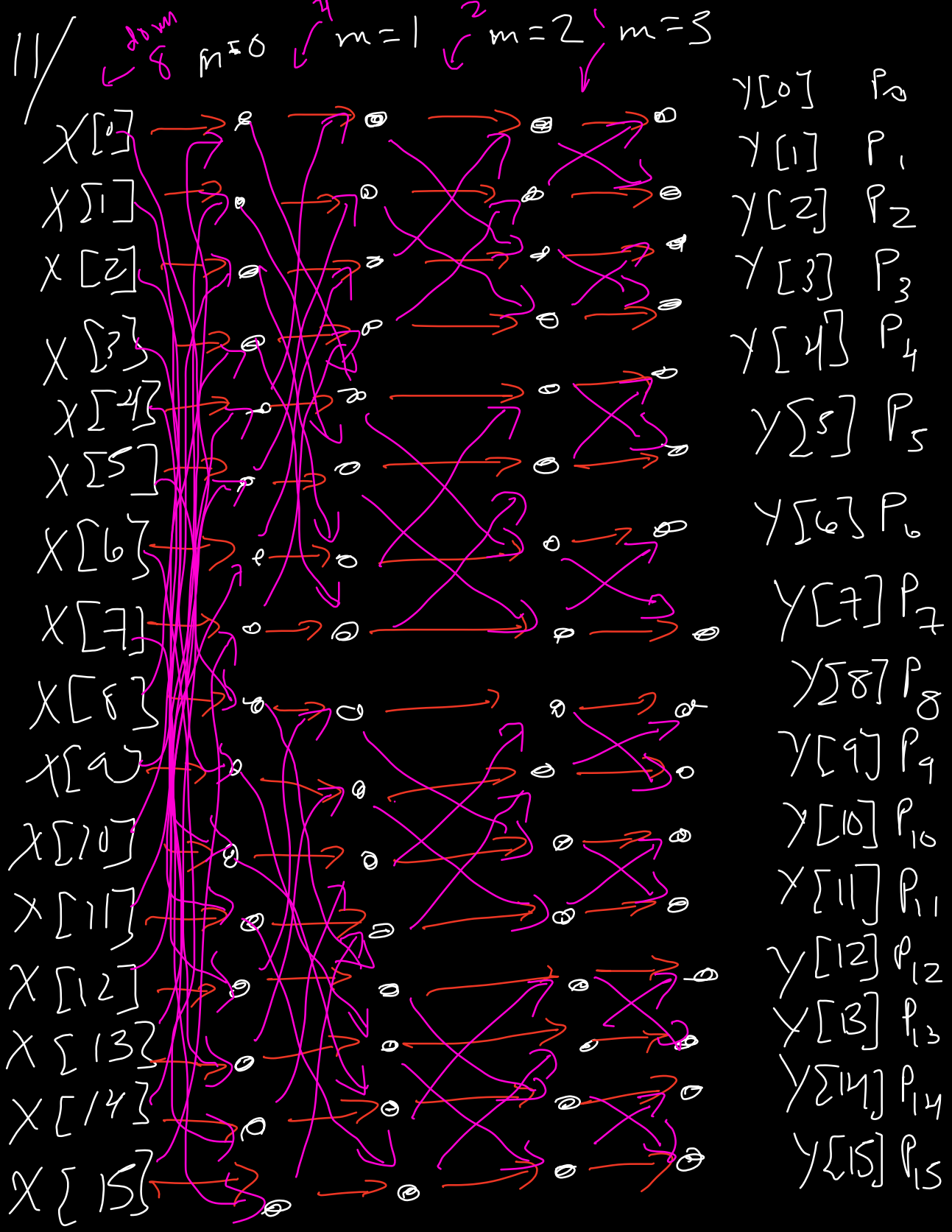
$$\boxed{O_{total} = \frac{1}{1 - O(\frac{log(p)}{log(n)}) + O(\frac{log^2(p)}{log(n)})}}$$

□

Exercise 11. In the algorithm shown, assume a decomposition such that each execution of Line 7 is a task. Draw a task-dependency graph and a task-interaction graph.

Proof. This is drawn on the ipad notebook that I have attached on the next few pages. We are assuming n=16 so this will effectively be the same answer for 12a

□

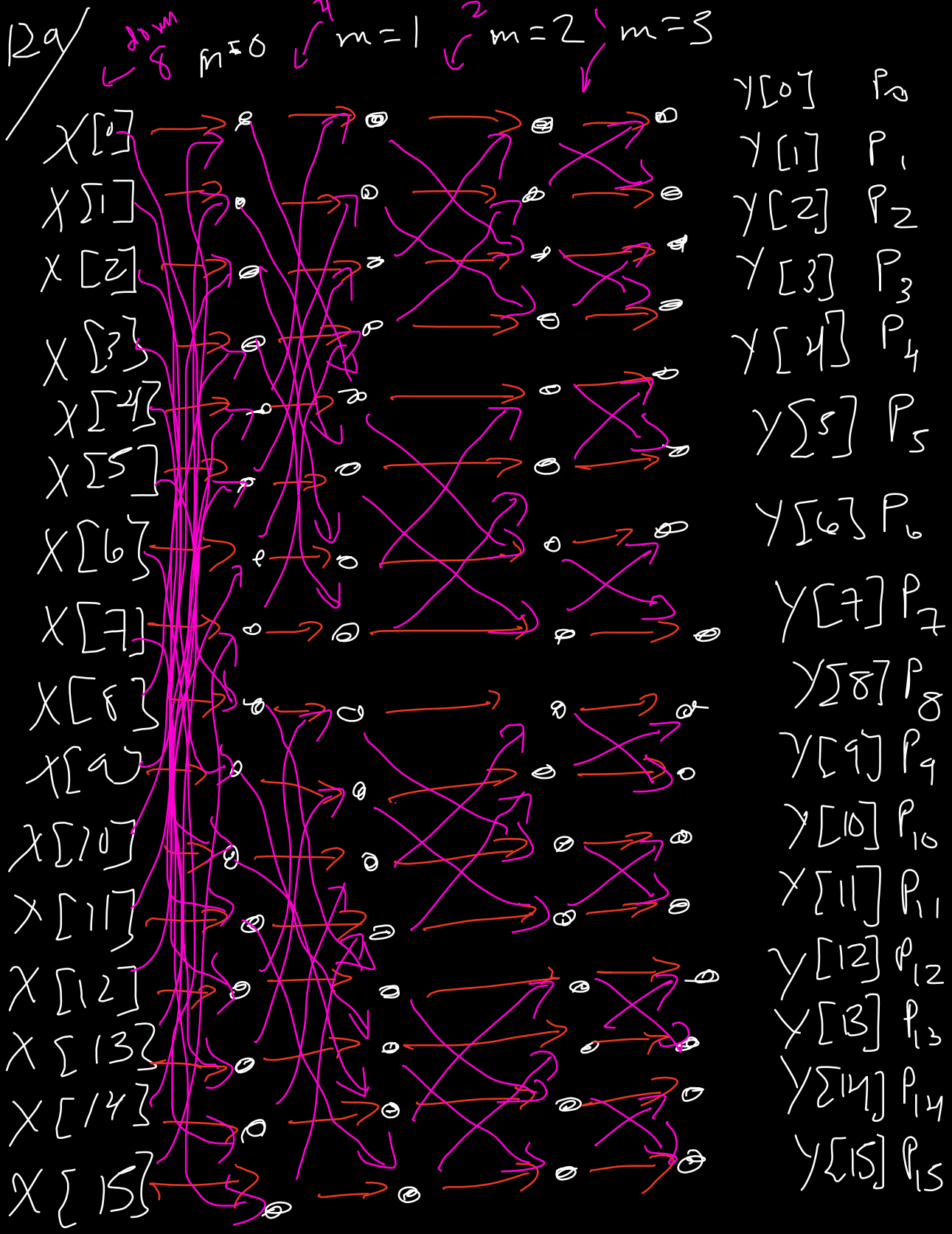

11/
down
m=0
m=1
m=2
m=3
x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
x[6]
x[7]
x[8]
x[9]
x[10]
x[11]
x[12]
x[13]
x[14]
x[15]
y[0] P0
y[1] P1
y[2] P2
y[3] P3
y[4] P4
y[5] P5
y[6] P6
y[7] P7
y[8] P8
y[9] P9
y[10] P10
x[11] P11
y[12] P12
y[13] P13
y[14] P14
y[15] P15

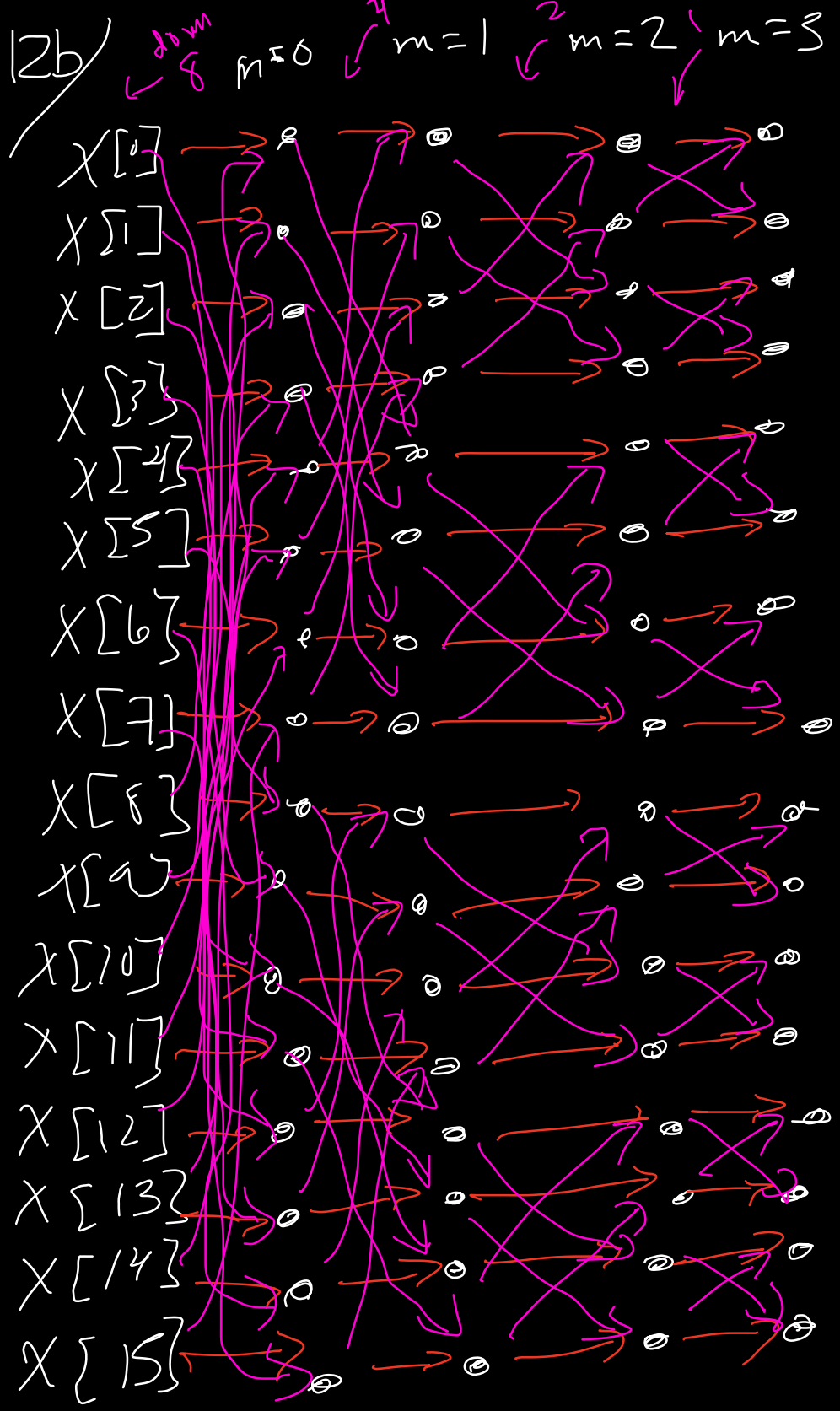
Exercise 12. In the above algorithm, if n = 16, devise a good mapping for:

a) 16 processes
b) 8 processes

Proof. Both of these are shown in the next pages, 12a is the same answer as 11 but I added the work just in case. For 12b, it's also effectively the same answer but now you have 8 processors processing 2 different Y values at the end of the mapping.

□


Q9
down
m=0
m=1
m=2
m=3
x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
x[6]
x[7]
x[8]
x[9]
x[10]
x[11]
x[12]
x[13]
x[14]
x[15]
y[0] P0
y[1] P1
y[2] P2
y[3] P3
y[4] P4
y[5] P5
y[6] P6
y[7] P7
y[8] P8
y[9] P9
y[10] P10
x[11] P11
y[12] P12
y[13] P13
y[14] P14
y[15] P15


12b
m=0
m=1
m=2
m=3
X[0]
X[1]
X[2]
X[3]
X[4]
X[5]
X[6]
X[7]
X[8]
X[9]
X[10]
X[11]
X[12]
X[13]
X[14]
X[15]
Y[0]
Y[1]
Y[2]
Y[3]
Y[4]
Y[5]
Y[6]
Y[7]
Y[8]
Y[9]
Y[10]
X[11]
Y[12]
Y[13]
Y[14]
Y[15]
P0
P1
P2
P3
P4
P5
P6
P7

Exercise 13. Consider seven tasks with running times of 1, 2, 3, 4, 5, 5, and 10 units, respectively. Assuming assign a work to a process does not take any time, compute the best- and worst-case speedup for a centralized scheme for dynamic mapping with two processes.

Proof. So the time taken to complete all the tasks with a single process is 30 units. Now let's show the best case speed, which will occur when the processors share the load equally. The best case would be that when processor 1 completes tasks 1, 2, 3, 4, and 5, while processor 2 will complete tasks 5 and 10. This would be done in the following order, where P is processor and T is task.

$$P_1(T1), P_2(T6), P_1(T2), P_1(T3), P_2(T7), P_1(T4), P_1(T5)$$

So in essence it takes processor 1 and 2 an equal amount of time to complete the tasks which is 15 units. Hence the best case speed up would be $\frac{30}{15} = 2$

Now let's take a look at the worse case, here processor 1 would complete tasks 1,2,3, and 7 while processor 2 would complete task 5 and 6. This would be done in the following where P is processor and T is task.

$$P_1(T1), P_2(T5), P_1(T2), P_1(T3), P_2(T6), P_1(T4), P_1(T7)$$

So here processor 1 would take 20 units to complete while processor 2 would only take 10 units to complete it's tasks. Hence the worst case speed up would be $\frac{30}{20} = 1.5$

Best case speed up = 2
Worst case speed up = 1.5 □