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 n2 and computation increases with n3. Repeat e) for memory bounded speedup.

Solution

For this we must know about Amdahl's Law ,

Amdahl's Law: it states that, if x is the fraction of the application that can be parallalized then, then the maximum speedup for that application is given by 1 / ((1-x) + (x/N))

whrere N is the number of processors .

so now we just need to apply the formula for the different portions given .

here N is 8 since it is 8 core processor .

(a) here x = 0.4 since 40% is parallelized.

speedup = 1 / ((1 - 0.4) + (0.4/8)) = 1.54

(b) x = 0.99

speedup = 1 / ((1 - 0.99) + (0.99/8)) = 7.47

(C) Speedup = 1 / [0.2 + 0.8 * (0.6 + 0.4/8]

= 1.38

(d) speedup = 1/ [0.8 + 0.2 * (0.01 + 0.99/8)]

= 1.21

(e) fixed-time Speedup $= f + P(1-f)$

(a). 0.6 + 0.4* 8 = 3.8
(b) 0.01 + 0.99 * 8 = 7.93

(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. According to the " Scalable problems and memory bound speed up", the workload on each processor is

$$Speedup_{MB} = \frac{Work(p)/Time(p)}{Work(1)/Time(1)} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)G(p)/p}$$

Here $G(p) = p^{3/2} = 8^{3/2} = 22.62$

(a) [0.6 + 0.4 * 22.62] / [0.6 + 0.4 * 22.62/8]

= [0.61] / [1.73] = 0.35

(b) [0.01 + 0.99 * 22.62] / [0.01 + 0.99 * 22.62/8]

= [22.40] / [2.80] = 8

2. Prove Amdahl's law (i.e., when p goes to infinite, under Amdahl's law the speedup is ..?).

Amdahl's law states that in parallelization, if α is the proportion that remains serial , and 1-α is the proportion of a system or program that can be made parallel then, the maximum speedup that can be achieved using N number of processors is 1 / [(α)+(1-α/N)].

- Amdahl's law gives a limit on speedup in terms of α .
- Execution time of any code has two portions
- Portion I: not affected by enhancement
- Portion II: affected by enhancement
- Execution $time_{old}$ = execution $time_{p1}$ + execution $time_{p2}$
- Parallel run time is given by

$$\text{execution time}_{new} = (\alpha) * \text{execution time}_{old} + (1-\alpha) * \frac{\text{execution time}_{old}}{p}$$

$$T_p = (\alpha + \frac{1-\alpha}{p}) \bullet T_s$$

- Amdahl's law gives a limit on speedup in terms of α

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- If we assume that the serial fraction is fixed, then the speedup for infinite processors is limited by 1/α

$$\lim_{p->\infty} S_p = \frac{1}{\alpha}$$

- Speedup is limited by the total time needed for the sequential (serial) part of the program. For 10 hours of computing, if we can parallelize 9 hours of computing and 1 hour cannot be parallelized, then our maximum speedup is limited to 10x.

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?

Solution:

(a). To compute the speedup obtained from the fast mode we must work out the execution time without the enhancement. We know that the accelerated execution time consisted of two halves: the unaccelerated phase (50%) and the accelerated phase (50%).

Without the enhancement, the unaccelerated phase would have taken just as long (50%), but the accelerated phase would take 10 times as long, i.e. 500%. So the relative execution time without the enhancement would be 50% + 500% = 550%.

Thus the overall speedup is

execution time unaccelerated / execution time accelerated = 550% / 100% = 5.5

(b). To find the percentage of the original execution time which was accelerated, we plug these figures into Amdahl's Law again:

fraction vectorised = speeduprm overall X speedup accelerated - speedup accelerated / speedup overll X speedup accelerated - speedup overall

= 5.5 X 10 -10 / 5.5 X 10 - 5.5
=45 / 49.5
=90.90%

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)=10.

Given: a=0.2, p=20

a) **Amdahl's law**

1 / ((x) + (1-x/N)) = 1 / ((0.2) + (0.8/20))

= 1/0.24 = 4.16

b) Gustafson's law

Speedup $= f + P(1-f)$
= 0.2 + 20 (1-0.2) = 16.2

c) Sun/Ni's law, assuming G(20)=10.

(as discussed in class took G(20) =10 **)**

= [α +(1 – α) G (p)] / [α + (1 – α) $G(p)$/ p]
= [0.2 + (0.8) *5] / [0.2 + 0.8 * 10 /20]
= [4.2] / [0.6]
= 7

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.

The four steps to design a parallel algorithm are as follows

Step 1: Partitioning
Divide the computation and data into pieces
Step 2: Communication
When primitive tasks are identified, determine the communication pattern between them There are two types of communication.
Local communication: A task needs values from a small number of neighboring tasks
Global communication: A significant number of tasks must contribute data to perform a computation.
Step 3: Agglomeration
The process of grouping tasks into larger tasks. The purpose is to improve performance and simplify programming.
Step 4: Mapping
Assigning tasks to processors

The five methods/patterns of parallel algorithms which are introduced in class are as follows.

The Partition Method,
The PPT Algorithm,
The PDD Algorithm,
The LU Pipelining Algorithm,
The PTH Method and PPD Algorithm

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.

- Amdahl's law gives a limit on speedup in terms of α (FIXED SIZE)

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- Gustafson's Law, (Fixed-time speedup model)

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

- Sun & Ni's Law (memory-bounded speedup)

$$Speedup_{MB} = \frac{Work(p)/Time(p)}{Work(1)/Time(1)} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)G(p)/p}$$

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

- Amdahl's law gives a limit on speedup in terms of α (FIXED SIZE)

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- Amdahl's Law with Overhead, To include overhead will be even worse, the overhead includes parallelism and interaction overheads

$$Speedup_{FS} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p} + T_{overhead}} \rightarrow \frac{1}{\alpha + \frac{T_{overhead}}{T_1}} \; as \; p \rightarrow \infty$$

- Gustafson's Law, (FIXED TIME speedup model)

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

- Gustafson's Law With Overhead (FIXED TIME speedup model)

$$Speedup_{FT} = \frac{Work(p)}{Work(1)} = \frac{\alpha + (1-\alpha)p}{1 + T_{overhead}/T_1}$$

- Sun & Ni's Law (memory-bounded speedup)

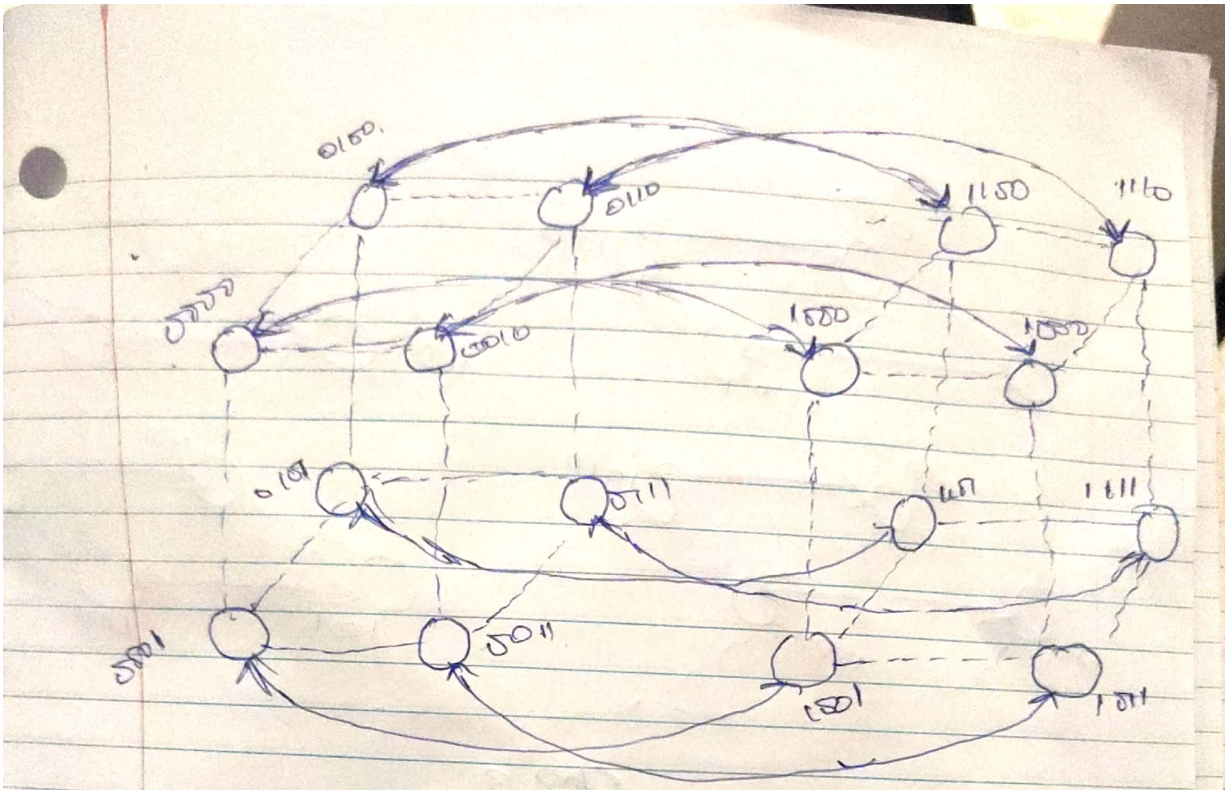
$$Speedup_{MB} = \frac{Work(p)/Time(p)}{Work(1)/Time(1)} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)G(p)/p}$$

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.

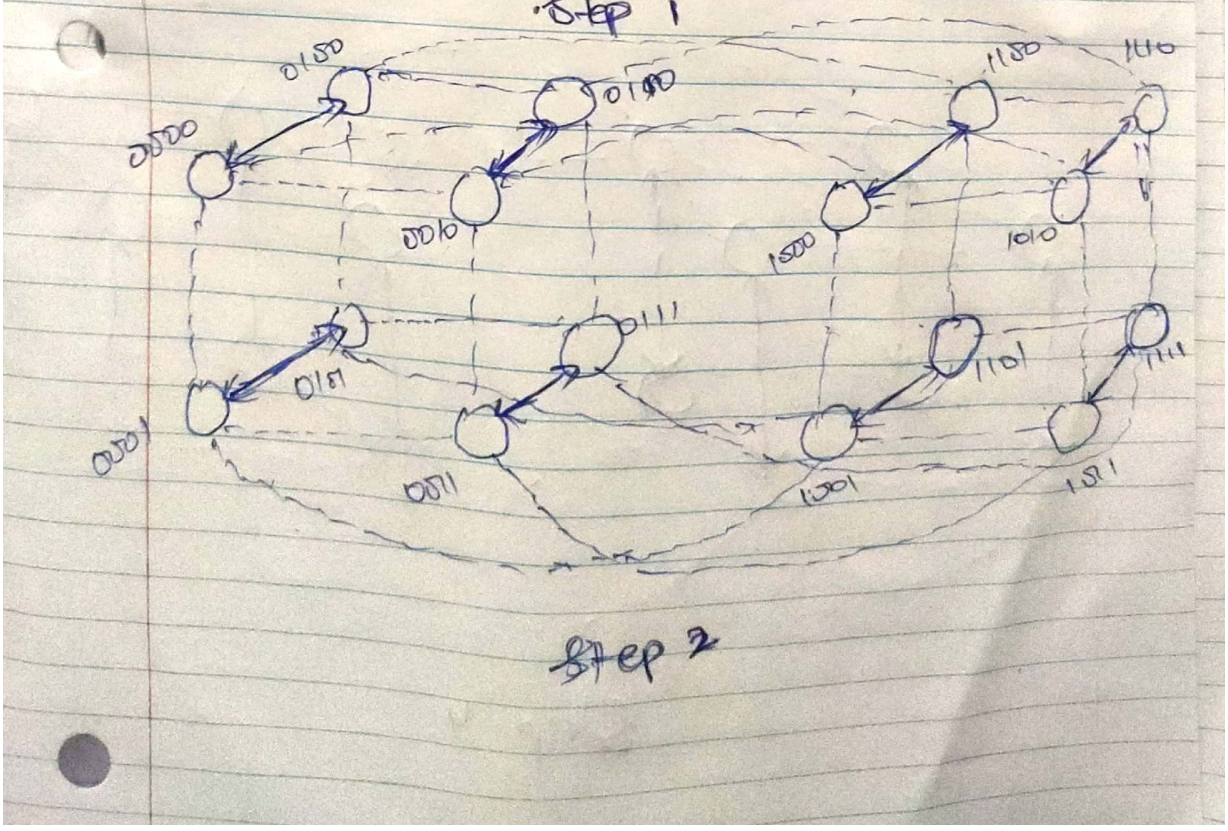
- If there are p processors then we divide the n items into n/p size of p blocks. Here there are only p/2 processors available , if we still divide the items into blocks of n/p size there we only cover (n/p)(p/2) items which is n/2,
- But our requirement is to sort all the n items for that purpose we need to modify the algorithm in such a way that all the elements are covered.
- This can be obtained by dividing the n items into blocks of size (2n/p) so that (2n/p)(p/2) = n items and and now we have to use p/2 instead of p while passing it as arguments. Rest of the things remains the same.

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

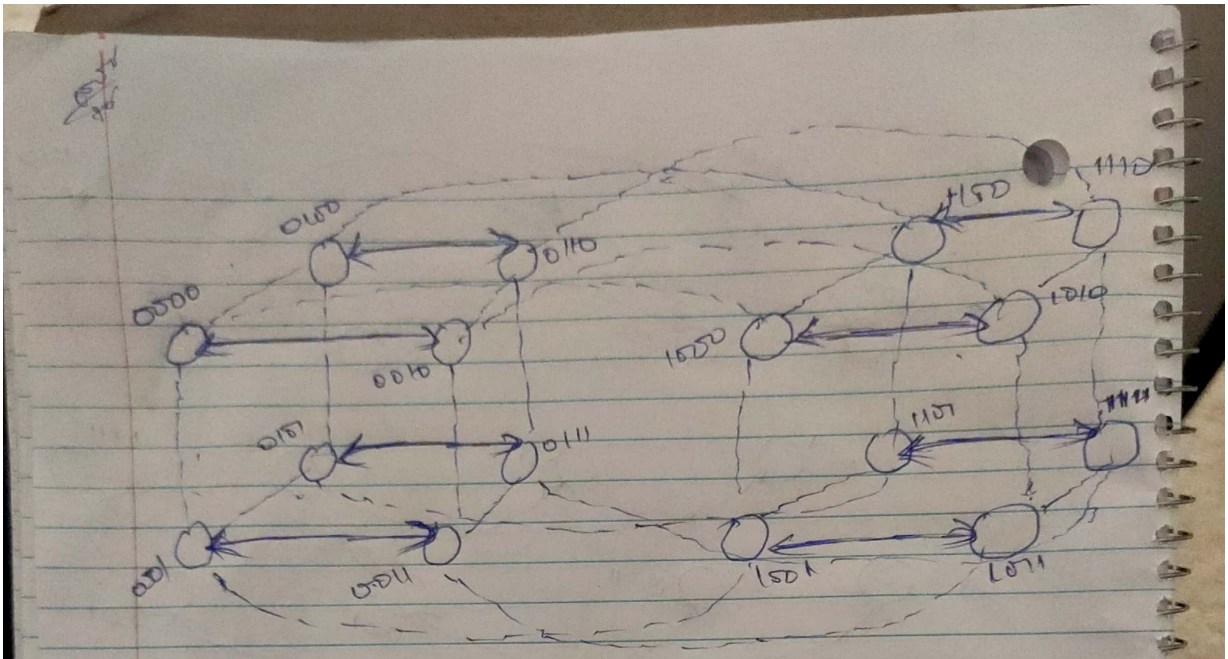
- Hypercube Mapping wires onto the processes of a hypercube-connected parallel computer is straightforward. Compare-exchange operations take place between wires whose labels differ in only one bit.
- In a hypercube, processes whose labels differ in only one bit are neighbors. Thus, an optimal mapping of input wires to hypercube processes is the one that maps an input wire with label l to a process with label l where $l = 0, 1, ..., n - 1$.
- Consider how processes are paired for their compare-exchange steps in a d-dimensional hypercube (that is, $p = 2^d$). In the final stage of bitonic sort, the input has been converted into a bitonic sequence.
- During the first step of this stage, processes that differ only in the d^{th} bit of the binary representation of their labels (that is, the most significant bit) compare-exchange their elements.
- Thus, the compare-exchange operation takes place between processes along the d^{th} dimension. Similarly, during the second step of the algorithm, the compare-exchange operation takes place among the processes along the $(d - 1)^{th}$ dimension
- In general, during the i^{th} step of the final stage, processes communicate along the $(d - (i - 1))^{th}$ dimension. Illustrates the communication during the last stage of the bitonic sort algorithm



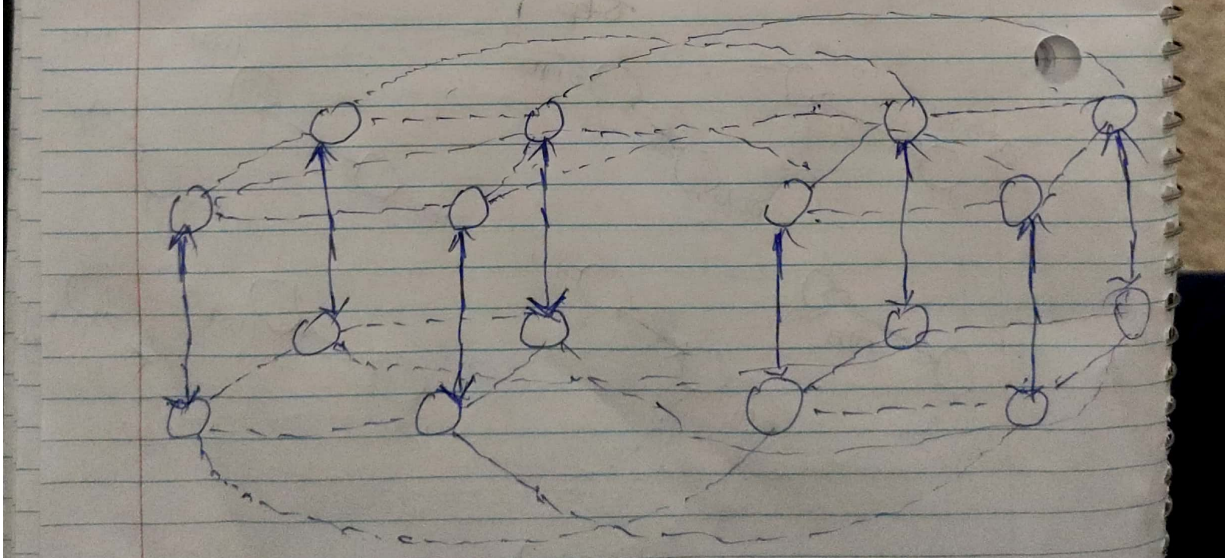
Step 1



Step 2



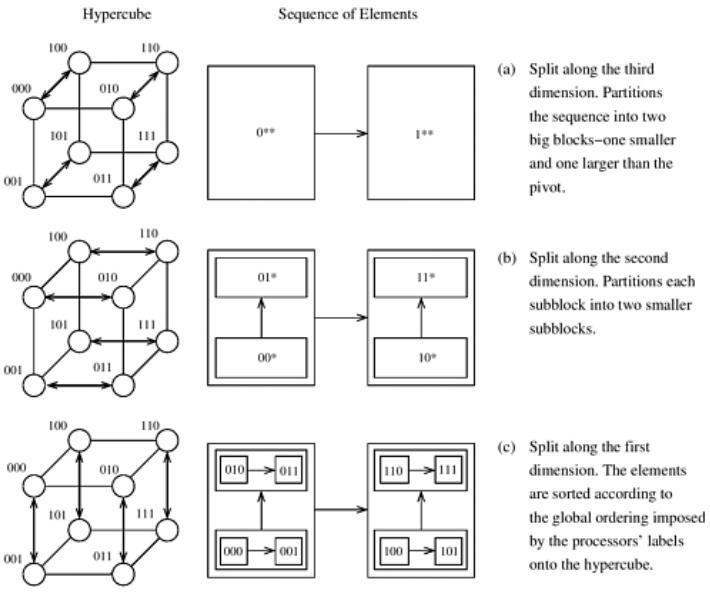
Step 3



Step 4

10. Excise 9.17, Textbook "Introduction of Parallel Computing" (2nd Edition)

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 d^{th} communication link exchange appropriate blocks so that one retains elements smaller than the pivot and the other retains elements larger than the pivot. Specifically, each process with a 0 in the d^{th} 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 d^{th} bit retains the larger elements. After this step, each process in the $(d - 1)$- dimensional hypercube whose d^{th} 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 . Figure 9.21. The execution of the hypercube formulation of quicksort for $d = 3$. The three splits – one along each communication link – are shown in (a), (b), and (c). The second column represents the partitioning of the n -element sequence into subcubes. The arrows between subcubes indicate the movement of larger elements. Each box is marked by the binary representation of the process labels in that subcube. A * denotes that all the binary combinations are included.



Algorithm 9.9 A parallel formulation of quicksort on a d - dimensional hypercube. B is the n/p -element subsequence assigned to each process.

```
1.procedure HYPERCUBE_QUICKSORT (B, n) begin

2.id := process's label; for i := 1 to d do begin

3.x := pivot;

4. partition B into B1 and B2 such that B1 ≤ x < B2; if ith bit is 0 then
begin

5. send B2 to the process along the i th communication link;
6. C := subsequence received along the i th communication link;

7. B:=B1 ∪ C; endif

8. else

9. send B1 to the process along the ith communication link;
10. C := subsequence received along the ith communication link;

11. B:=B2 ∪ C; endelse

12. endfor

13. sort B using sequential quicksort; end HYPERCUBE_QUICKSORT
```

Analyze the complexity of this 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.

(1) Here we need to find time complexity of both sequential and parallel quick sort algorithms and also ratio of sequential time complexity and parallel time complexity.

(2) The block based algorithm for a hypercube with P processes is similar to the one element per process case, but now we have P blocks of size n/P, instead of P elements.

(3) Furthermore, the compare-exchange operations are replaced by compare-split, each taking (n/P) computation time and (n/P) communication time.

(4) Initially the processes sort their n/P elements in time $\Theta((n/P)\log(n/P))$ and then perform $(\log^2 P)$ compare-split steps.

(5) The parallel run time of this formulation is

$$T_P = \Theta\left(\frac{n}{P}\log\frac{n}{P}\right) + \Theta\left(\frac{n}{P}\log^2 P\right) + \Theta\left(\frac{n}{P}\log^2 P\right)$$

Local sort — Comparision — Communication

(6) And also the sequential complexity of the best sorting algorithm is $(n \log n)$, the efficiency is as follows

$$S = \frac{\text{Sequential Complexity}}{\text{Parallel Complexity}}$$

→ (Seq complexity for quick sort algorithm)

$$= \frac{\theta(n \log n)}{\theta((n/p)\log(n/p)) + \theta((n/p)\log^2 p)}$$

↓ (Parallel complexity for quick sort alg)

$$E = \frac{1}{1 - \theta((\log p)/(\log n)) + \theta((\log^2 p)/(\log n))}$$

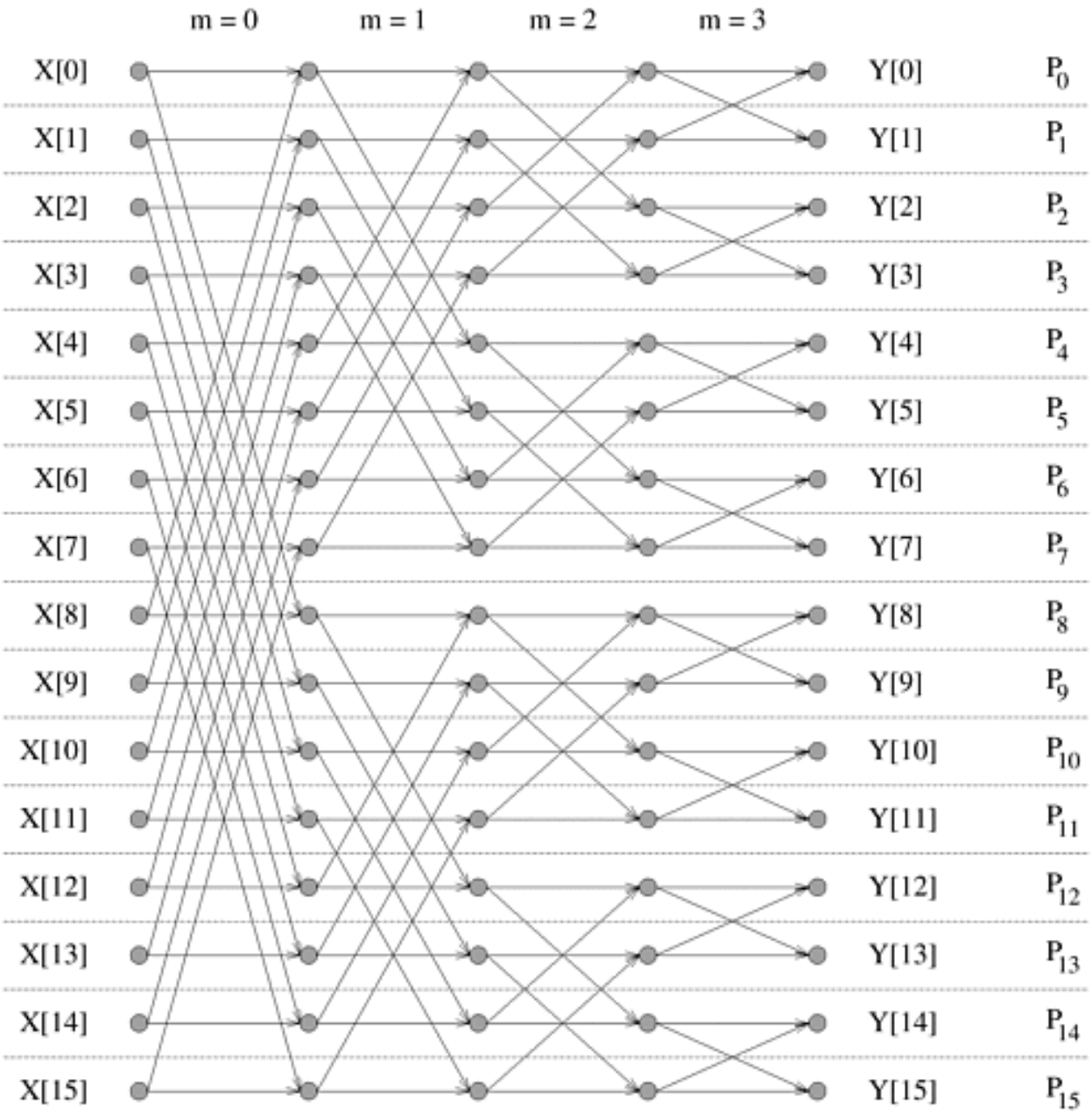
11. In the algorithm shown, assume a decomposition such that each execution of Line 7 is a task. Draw a task-dependency graph and a task-interaction graph.

```
1.   procedure FFT_like_pattern(A, n)
2.   begin
3.       m := log2 n;
4.       for j := 0 to m - 1 do
5.           k := 2^j;
6.           for i := 0 to n - 1 do
7.               A[i] := A[i] + A[i XOR 2^j];
8.       endfor
9.   end FFT_like_pattern
```

The ***binary-exchange algorithm*** for performing FFT on a parallel computer. First, a decomposition is induced by partitioning the input or the output vector. Therefore, each task starts with one element of the input vector and computes the corresponding element of the output. If each task is assigned the same label as the index of its input or output element, then in each of the log *n* iterations of the algorithm, exchange of data takes place between pairs of tasks with labels differing in one bit position.

Task-dependency graph and a task-interaction graph can be drawn from following figure.

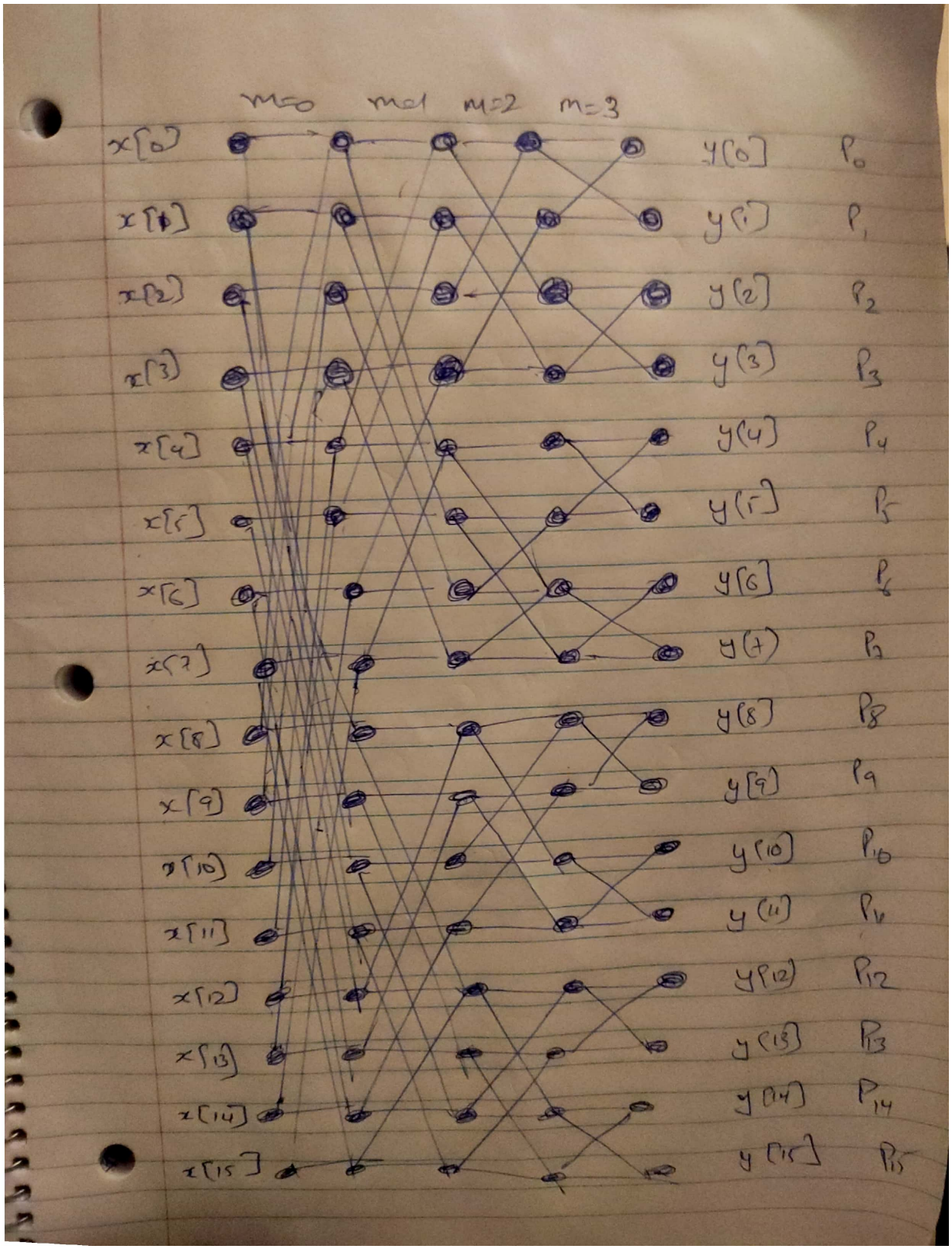
- The graph of tasks and their interactions/data exchange is referred to as a *task interaction graph*.
- *Task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.



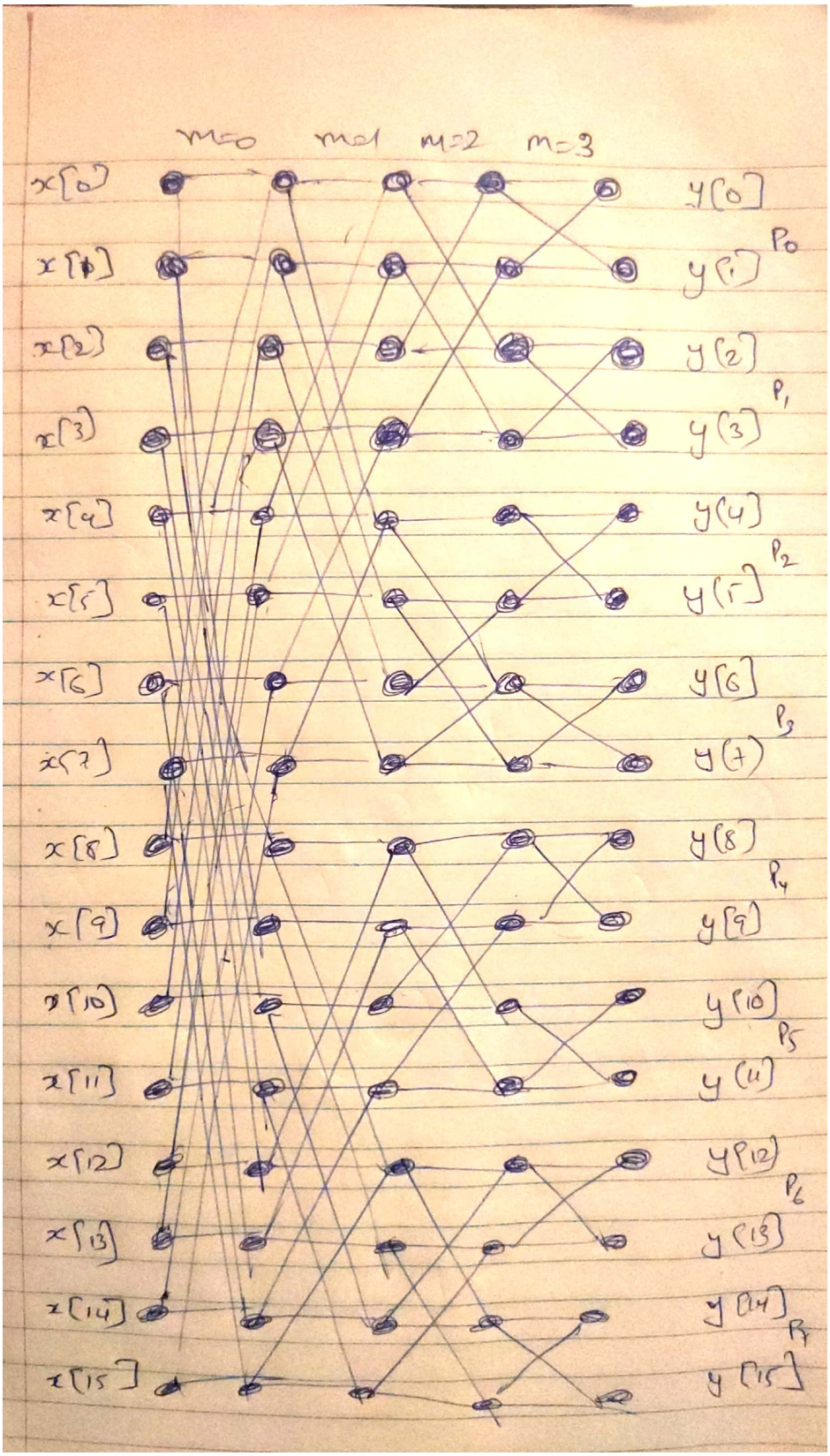
12. In the above algorithm, if n = 16, devise a good mapping for:

1. **a) 16 processes**

We first consider a simple mapping in which one task is assigned to each process the interaction pattern induced by this mapping of the binary-exchange algorithm for $n = 16$ is shown below.



2. **b) 8 processes**



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.

- The time taken to complete all the tasks with single process = 30 Minutes
- Best case speedup will occur when the processes share the load equally
    - Process 1 completes: t1, t2, t3, t4, t5 tasks = 15 minutes
    - Process 2 completes: t6, t7 tasks = 15 minutes
    - It happens when tasks assigned in the following order
    - p1(t1), p2(t6), p1(t2), P1(t3), p2(t7), p1(t4), p1(t5) ,
    - time taken to complete tasks = 15 minutes
    - Best case speedup = 30/15 = 2,
- Worst case speedup will occur when the processes share the load unequally, it will happen in the following case
    - process 1 completes: t1, t2, t3, t4, t7 tasks = 20 minutes
    - process 2 completes: t5, t6 tasks = 10minutes
    - it will happen when the tasks are executed in the following order
    - p1(t1), p2(t5), p1(t2), p1(t3), p2(t6), p1(t4), p1(t7)
    - time taken to complete tasks = 20 minutes
    - worst case speedup = 30/20 = 1.5