<u>**COMP428 Assignment 1**</u>
Jeremy Pallats – 9619550

1) Comparison of Speedups using MPI.

Notes on executing each of the three implementations are at the top of the files serial.c, parallel.c, spawn_master.c. Read before executing and use the prescribed command with optional arguments if you like. I have optionally allowed the number of darts to be passed as command argument. If not passed in, the default is DEF_DARTS which is 50 million. I found it to be rather small in practice for the fast CPUs. I've only kept 5 decimal places precision rounding on last. All these measurements were taken using the default darts number with the only variation number of tasks started either via busb n or in the case of spawn_master, that parameter is read by argument.

Implementation Note:

In all cases, I've modified the algorithm from the sample pi calculation file on the llnl site. In my implementation, I simply throw n number of darts and calculate one round of pi after all darts are returned to master. In the parallel and spawn programs I divide the workload n darts over the number of spawned processes. In all cases, I measure time in the same way, taking an initial time stamp immediately after MPI_Init and printing the final time immediately before MPI_Finalize. The time is only measured in the master process of the program.

a. Measured time for serial program throwing DEF_DARTS (50000000): 0.33603 seconds.
b. Table below summarizes times for different numbers of children. Program was executed with the command in the parallel.c file, with different n(task) values substituted while keeping DEF_DARTS as the work. I used all even values from 1-16, we aren't allowed more than 16 tasks on the 428 queue it seems.

Tasks	1	2	4	6	8	10	12	14	16
Time	0.34578	0.17390	0.08727	0.05690	0.04308	0.03471	0.02950	0.02597	0.02283

Figure 1: Data for Parallel Measurement with Default Darts

c. In order to test the spawned version, I execute the spawn_master exe with the command written at the top of the respective c file. I have passed only the number of workers directly to spawn_master, this means it will balance the default DEF_DARTS over all workers spawned. Note that I have timed in addition to total time to result, the time only to finish spawning at the master I believe this is relevant to discussion.

Tasks	1	2	4	6	8	10	12	14	16
Time Total	0.82460	1.69225	2.41282	2.60498	3.69162	4.07325	4.28382	3.80600	4.91002
Time Spawn	0.62922	1.56680	2.12659	2.36954	3.37263	3.80806	4.16523	3.51020	4.66241
Time Diff	0.19538	0.12545	0.28623	0.23544	0.31899	0.26519	0.11859	0.29580	0.24761

Figure 2: Data for Spawn Measurement with Default Darts

d. I've used excel to plot the number of workers on the x versus the time taken on the y of the below graph. Due to the spawn having such an impact on the scaling of the y axis, I've made a second graph without it to just show the serial vs parallel improvement in time.

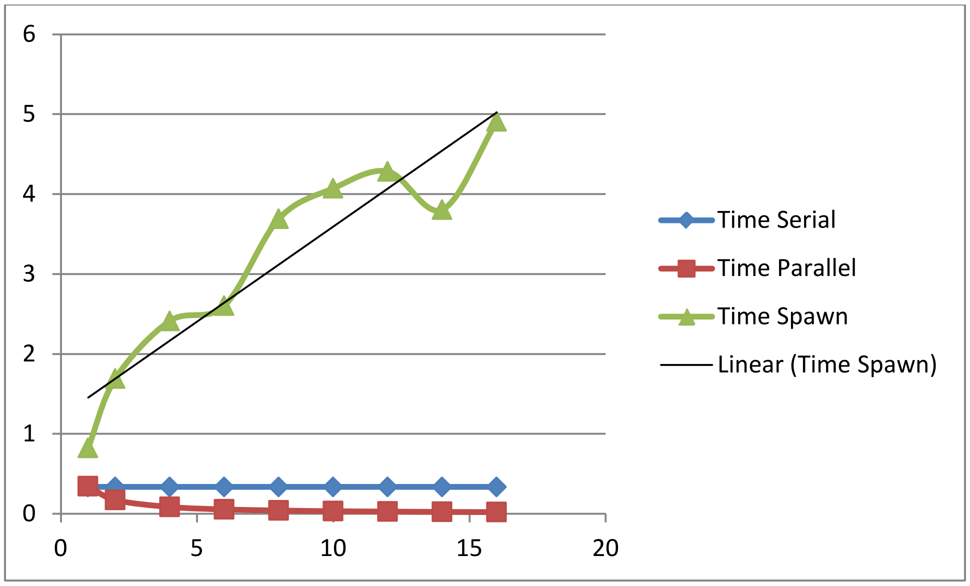

Figure 3: Tasks vs. Time Taken for Default Work

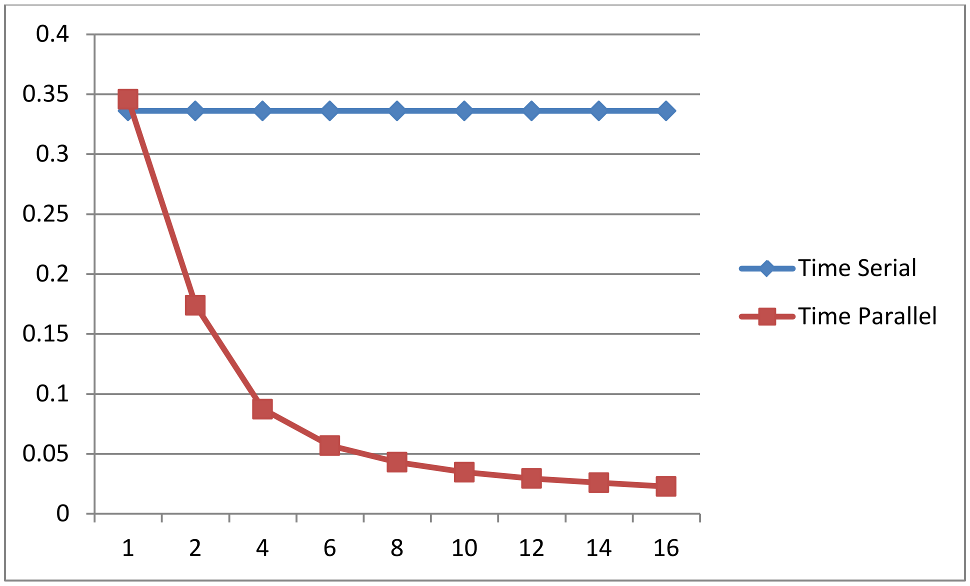

Figure 4 Plot: Tasks vs. Time Taken for Default Work (Serial + Parallel Only)

It is clear that looking at these graphs, the mpi_comm_spawn method fairs poorly. The additional overhead involved in spawning n threads dynamically appears to add a linear amount of time equal to n *Ts, where Ts is the time to spawn 1 thread. On the graph, I've made a black trend line that goes through the points. The fact that this is a collective action means that all threads are blocked until all children have called MPI_Init and then the master continues. There may be times when dynamic spawning can improve speed, but this is not one. We can clearly see from the time difference between spawn and total that it was the only factor changing total time.

It is simple to see why too, this parallel computation of PI (i.e. this dart algorithm) is best suited to static data decomposition where we have a fixed number of tasks and each performs n/t work where n = number of darts and t = num of tasks, then we reduce to the master and calculate. If we look now at the parallel vs serial graph we see that there is a dramatic improvement here. In this implementation we have statically started n tasks via bsub and each task can determine its own amount of work and then reduce. Now we have only one collective block at reduce which is after all tasks have completed work. We see that the efficiency gains drop off over time though they still remain mostly proportional to the tasks. Most importantly, we see that the decrease in time is linearly proportional to the number of tasks we subdivided the computation across this is the expected result as in all normal algorithms (i.e. not super-linear) we are limited in speedup by the number of tasks.

Now in the assignment it was asked to plot the speedup versus workers, so I will do this as well to analyze those graphs. First I will normalize all the data so that times reflect a fraction of the serial execution time Ts = 0.33602. I'll then divide the serial times by each of the parallel wall clock times to get the overall speedup of the program relative to the baseline Ts scenario.

Raw Wall Clock			
Tasks	**Time Serial**	**Time Parallel**	**Time Spawn**
1	0.33603	0.34578	0.8246
2	0.33603	0.1739	1.69225
4	0.33603	0.08727	2.41282
6	0.33603	0.0569	2.60498
8	0.33603	0.04308	3.69162
10	0.33603	0.03471	4.07325
12	0.33603	0.0295	4.28382
14	0.33603	0.02597	3.806
16	0.33603	0.02283	4.91002
Normalized Times where Ts = 1 so we divide all by 0.33603			
Tasks	**Time Serial**	**Time Parallel**	**Time Spawn**
1	1	1.029015266	2.453947564
2	1	0.517513317	5.03600869
4	1	0.259708955	7.1803708
6	1	0.169330119	7.752224504

8	1	0.128202839	10.98598339
10	1	0.103294349	12.12168556
12	1	0.087789781	12.74832604
14	1	0.077284766	11.32636967
16	1	0.067940362	14.61185013
Calculated Speedups Ts/Tp			
	Speedup Serial	**Speedup Parallel**	**Speedup Spawn**
1	1	0.97180288	0.40750667
2	1	1.932317424	0.198569951
4	1	3.850464077	0.139268574
6	1	5.905623902	0.128995232
8	1	7.800139276	0.091025078
10	1	9.681071737	0.082496778
12	1	11.39084746	0.078441671
14	1	12.93916057	0.088289543
16	1	14.71879106	0.068437603

Figure 5: Raw Data and Derived

To finish this question off, I'll plot the speedups. See below. Once again due to the data points the y axis is being pushed from the others so I'll plot another graph without the parallel speedup to more clearly see serial vs. Spawn speedup.

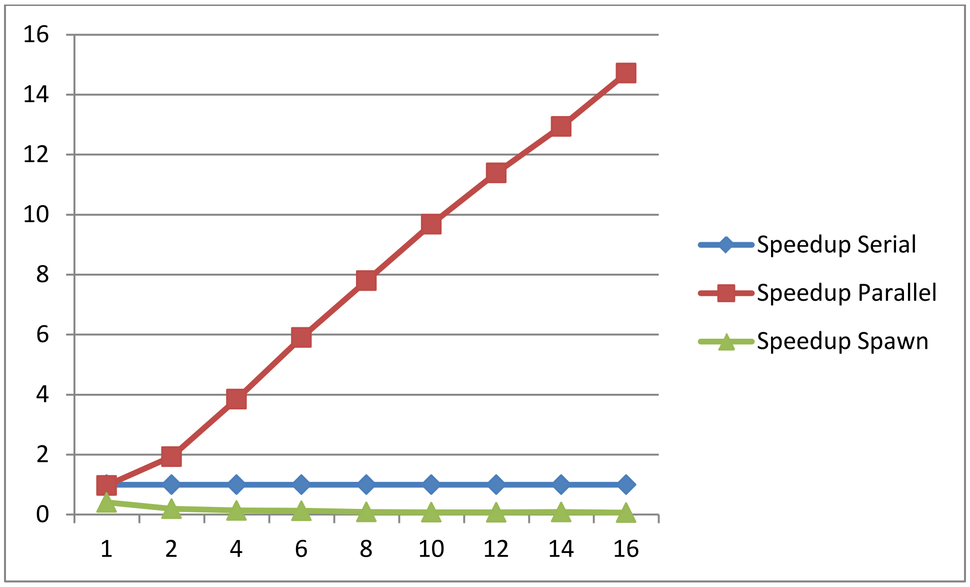

Figure 6: Tasks vs. Speedup of All

COMP428 Assignment 1
Jeremy Pallats – 9619550

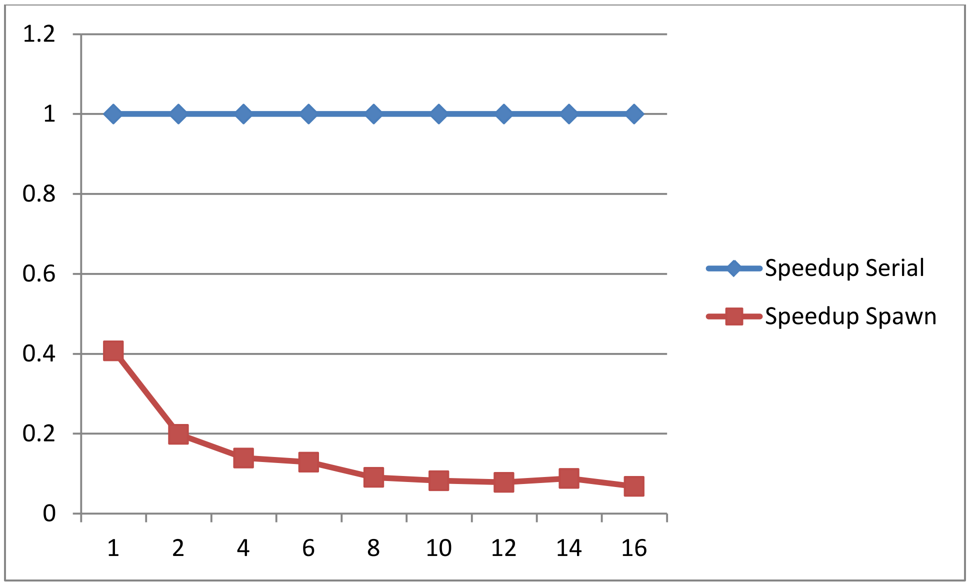

Figure 7: Tasks vs. Speedup (Serial + Spawn Only)

Analysis of Speedup

In Figure 6 it is very easy to see that the speedup of the parallel program is almost perfectly linear. As we increase the number of tasks, we achieve almost perfect linear increase speeding up the program by the same factor as we increased tasks. The slight drop in perfect increase can be accounted for since we have a fixed communication cost in the form of the reduction once all the darts have been thrown as well as other overhead from the MPI library just due to operation. There are also a few serial actions in the code such as prints in the master and calls to MPI functions. This is as expected for a parallel program since each task is doing n/t work (n=work, t = number of tasks), thus time taken decreases with proportion to the decrease in work.

In Figure 7 we see a much different picture and as we saw earlier we already know why. The speedup is dramatically sub linear and we can explain this simply by looking at the Time to Spawn in Figure 2. The majority of time in the spawn program is spent simply spawning the processes and waiting for them all to call init. This wastes a lot of time. In the parallel implementation, all processes are brought up at once by the MPI library minimizing overhead.

2) In order to handle this problem we can dynamically spawn workers with comm_spawn. Assume that each round we spawn 10 workers and ask them to compute pi with some fixed number of darts. They then reduce to the master their results. After the first round we store the result, we then request a second round and keep requesting rounds until the deviation from the running average of all previous rounds and the current round differs by less than some fixed constant. At this point we can say we will no longer have much improvement and print result. As a side note, there are much more efficient algorithms that converge faster and most are infinite series.
    a. Note: In the pseudo code I use a simple deviation formula, let us assume that AVE_PI and RND_PI have non-zero values then the deviation can calculated simply as absolute value of their difference. We compare this to some constant deviation and if it is less than we have a

result. So the calculation for deviation follows where abs is the absolute function:
deviation = abs(AVE_PI - RND_PI)

b. Note, for simplicity sake I have coded it as simply a split between master/worker where master is the original process and worker is any spawned processes. In a real program, these would be two different C programs. We would only spawn one task as the original MASTER and wait for its dynamic result.
c. Pseudo-Code for converging dynamic PI calculation

```
RND_PI =0 // PI Averaged from the round
AVE_PI = 0 // The average PI of all previous rounds
DEV = 0.001 // When less deviation than this, print result.
DARTS = 1000000 // Number of darts to throw.

Determine if I am MASTER or WORKER

If I am MASTER
    do
        MPI_Comm_spawn 16 workers to calculate PI based on darts algorithm
        MPI_Reduce PI calculated from all workers to variable RND_PI using MPI_SUM
        Divide RND_PI by 16
        If AVE_PI = 0
            AVE_PI = RND_PI
            GOTO spawn stage skipping while conditional
        Else
            AVE_PI = (AVE_PI + RND_PI) / 2
    while (absolute deviation of RND_PI from AVE_PI >  DEV)

    Print results of AVE_PI and RND_PI to user.

Else If I am WORKER
    Throw DARTS and calculate the hits.
    Calculate PI by 4.0 * hits/DARTS
    MPI_Reduce to master the result
```

3) No, it is not very likely that a divide-and-conquer strategy will ever reach 100% efficiency. In any parallel algorithm we have numerous overheads as we analyzed in class, these include inter process communication to coordinate, idling time when not working and extra work that may be required due to parallelization. These three things increase the time taken at each process in the algorithm.

In order for the efficiency to be 1, we would have to have the case where Ts = p*TP, where Ts is time serial, Tp is time taken by one process and p is the number of processes. In particular, a divide-and-conquer strategy involves numerous attempts to break down the problem and each subtask is usually delegated to a new worker unless it is very small. Each of these sub tasks will require inter process communication to first send and then receive the response from the workers (alternatively, if dynamically spawned that has a cost too). These overheads will almost always make the Tp be greater than n/p, which means we will never reach 100% efficiency.

To increase efficiency, we must minimize the three kinds of overhead above that means:

- Reduce the number of inter process communications, this likely means subdividing the problem into larger and larger units. The smaller the units the more communication so if we keep the division fairly coarse we will increase the efficiency.
- Reduce the time idling, this requires that the dividing algorithm fairly divide the work amongst the processes. In particular, this means the algorithm dividing the problem should take care to ensure the randomly distributed data is somewhat evenly distributed. If not, it will be lopsided and some processes may finish first.
- Reduce the extra overhead to parallelize. This isn't as clear and depends on the algorithm, in general we must design it so that we minimize the extra computations needed to merge or collect the results of the subtasks together.

4) For this problem, we will have to begin analysis by looking at the serial time taken by this program given this dependency graph by calculating how long it would take to complete all steps sequentially. Given the information in the description we have:
   a. To calculate the efficiency we first need to consider the serial time taken to run this program:
      Ts = inputTime + stage1Time*tasks + stage2Time*tasks + outputTime

      Ts = 1 + 2 * 10 + 5 * 5 + 1

      Ts = 47 units of time

   Now to calculate the parallel time, we must consider how many parallel processors is required at minimum to execute everything that can be done in parallel as soon as it is free to be computed. Looking at the dependency graph we see the following:

   1) Input must be done before all others, map to p0.
   2) Stage 1 can be done in parallel, all 10 tasks can mapped to 10 processes so map them to p0-p10. They should all complete in roughly same time.
   3) Stage 2 can only be done by a maximum of 5 processes so map the outputs of the previous processes to the even processes so: p0 + p1 -> p0, p2 + p3 -> p2 and so on. We have at most 5 processes working here.
   4) Output must combine all previous steps in one sequential task of 1 unit. Map to p0.

   Based on the above observations, we see that to maximize concurrency with the minimum number of processors we need 10 to calculate stage 1 at maximum speed though many processors will idle when not in this stage. Assuming a p = 10, we can calculate Tp as follows.

Tp = inputTime + stage1Time + stage2Time + outputTime

Tp = 1 + 2 + 5 + 1

Tp = 9

We now need to calculate the speedup using amdahl's law, as S = Ts/Tp.

S = 47/9 = 5.22

Now efficiency can be calculated as:

E = S/p = 5.22/10 = 0.522

We see then that we get roughly 52% efficiency for this optimal solution.

b. In order to examine what impacts the efficiency, let us look at the equation for it.
E = S/p = (Ts/Tp)/p =
E= Ts/(p*Tp)

i. Increasing Processors: If we look at the equation, if we increase the number of processors we will scale the denominator of our ratio. This will push our efficiency towards zero as p approaches infinity. We can thus say with certainty that increasing the processors in any case will if other factors remain the same will **decrease the efficiency.**

ii. Decreasing Processors: This statement is simply the inverse of the previous, and by the same logic we can see as p approaches 1 (the logical floor of p, can't compute without a processor) the efficiency will approach 1 as the closer Ts is to Tp. We can thus say as the number of processors decreases in general the efficiency increases. However, there is a caveat, in this program as we decrease the processors below 10, we will bottleneck at stage 1 and if p is less than 5, we will bottleneck at stage 2. When I say bottleneck, I mean that we will have more parallel work waiting than processors free and thus there will be work idling with no free processor to take it. This will further increase the parallel time Tp and thus while decreasing p may increase by some factor efficiency, increasing Tp by a competing factor may counteract the efficiency gain. **Thus, we cannot say that in all cases the Efficiency goes up as we decrease processors, it depends on the factors impacting p and Tp respectively.**

To further illustrate, let us compute time taken above by using only p=8 processors. Assuming no impact on Tp we would expect the new E to be 5.22/8 = .6525. A 25% gain over the old E. However, let us look at the new Tp. Given our previous constraints we calculate Tp as:

Tp = inputTime + outputTime + stage1Time + stage2Time

Given that we only have 8 processors at stage1, we have to do 2 of the tasks after the other 8 are finished. However, while the other 2 are calculating stage1 prerequisites for one of the tasks in stage 2, the other 4 unblocked tasks can execute on other processors and the last stage2 task can start as soon as it is unblocked so we get:

Stage1Time = 2 + 2 = 4

Stage2Time = 5

Tp = 1 + 1 + 4 + 5 = 11

NewE = Ts/(p*Tp) = 47 / (8 * 11) =.534, almost exactly the same since the decrease of p was counter balanced by the stalling of Tp during stage 1. The impact on Tp would only increase dramatically as p goes lower than 5 as there are far too many tasks fighting over processor time.

iii. Faster Processors: Assuming the number of processors p remains the same and we simply change to faster processors there will be no change in E. Firstly because we measure Ts and Tp not in wall time in this case but in units of serial time it would take on any processors. Secondly, even if we did measure in that time, the decrease in times would reduce both Ts and Tp by the same factor, **the efficiency thus remains the same as we increase processor speed.**

iv. Note, none of these are guaranteed to increase efficiency overall I hope these explanations are sufficient.

5) Proof of the statement that given any task dependency graph, the maximum concurrency at any given moment d must be bounded as follows $\lceil t/l \quad \leq d \leq t - l + 1$. In this equation, t = number of tasks, l = longest critical path. In order to prove this is the case, we can look at the upper and lower bounds separately. In order to look at both, we will have to examine the cases of the different extremes of a dependency graph, Figure 8 shows a completely parallel graph with no dependency while Figure 9 shows a completely dependent graph where only the first task is free and every other task is dependent on the previous.

a. $d \leq t - l + 1$

i. We can derive this upper bound of the equation by examining the maximum concurrency possible in our two cases. It is clear that the best possible situation is one where t = d, which happens in case 1 where the graph is completely without dependency.

ii. However, in order to account for other extreme in case 2 we must consider the effect of l (the critical path) on concurrency. We can see that as l increases, it decreases the maximum upper concurrency by the same factor since it results in tasks that cannot complete until prior tasks complete. For example if the critical path is 2, that means that there is one task that cannot be completed in the first round of time. The maximum concurrency is then t -1.

iii. In general we can formulate this upper bound as t – l + 1, the + 1 accounts for the fact that in all cases the first task in l can be completed immediately but we are subtracting it so we must offset it. Thus we can say this proves that d must be less than or equal to t – l + 1.

b. $\lceil t/l \quad \leq d$

i. Now we look at a slightly more difficult case the lower bound, the derivation is a bit less obvious. Here we aim to find the lowest possible concurrency. In any graph, we know that the least amount of concurrency is 1 since the worst graph is case 2 with all tasks dependent on the previous in a line.

ii. In the case where all tasks have no dependency such as in case 1 we can see that d must be equal to t.

iii. Let us look at a third case such as that in Figure 10. Here we have only partial dependency, the critical path is 3, and the number of tasks is 7. Here the minimum concurrency would in fact be 3. This is because in our fully flushed tree,

iv. Using these cases we can see that the lower bound is always found by dividing the total tasks by the tasks in the critical path. We must however take the ceiling of it due to the fact that we can only have an integer number of tasks. We take the ceiling since in all cases the division of t/l is lower than the actual number. When t = l it is exactly one. However, if l = t-1, then t/t-1 is some value slightly greater than 1. However, given l is on less than t we know that at least two tasks can be concurrent at a moment so it must be that d is in fact 2, the ceiling of t/t-1.

c. Based on these observations, we have derived the upper and lower bounds and see how they can predict the maximum concurrency of a graph. By putting them together we get the original formula.

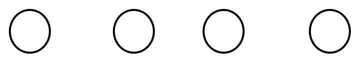
Figure 8: Case 1: No Dependency Among Tasks

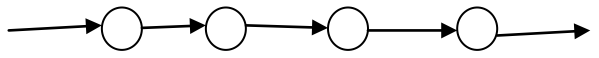
Figure 9: Case 2: Complete Dependency Among Tasks

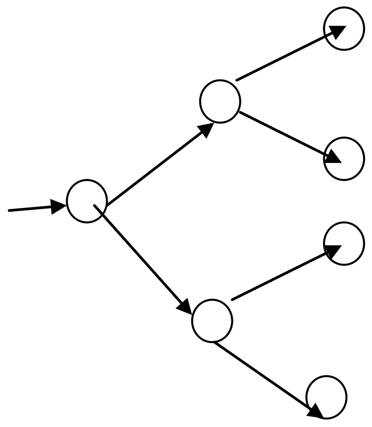

Figure 10: Case 3: Partial Dependency