CSE 4351/5351 Notes 7: Task Scheduling & Load Balancing

Task Scheduling

A *task* is a (sequential) activity that uses a set of inputs to produce a set of outputs. A *task (precedence) graph* is an acyclic, directed graph that uses the inputs and outputs for a set of tasks to indicate *precedence*. In rare cases the relationship between an output and an input may be a data stream, but more commonly the connection is through a file that is available only when the task producing the file as output has completed. (On the other hand, a *task interaction graph* is usually cyclic and features tasks that communicate during execution. For such systems, network flow models are used to achieve a good mapping of tasks to processors.)

The goal of task scheduling is to produce a *schedule* that assigns each task to a processor for a specific time interval such that:

1. A task commences only when its inputs are available at the processor.
2. The time interval for a task is appropriate for the task's requirements and the processor's rating.
3. A processor only runs one task at a time.
4. If two tasks will be assigned to different processors and there is an output-to-input precedence, then there must be sufficient time for the related file to be transferred. (Thus, task scheduling on a shared-memory system may be simpler.)

5. The time for the schedule is minimized.

Heterogeneity, in terms of processor speeds and distances between processors, can greatly complicate the problem. Simplifications, such as assuming a fully-interconnected network with a lower capacity, are often used.

Intractability of Task Scheduling

Even simplified versions of the task scheduling problem are NP-complete:

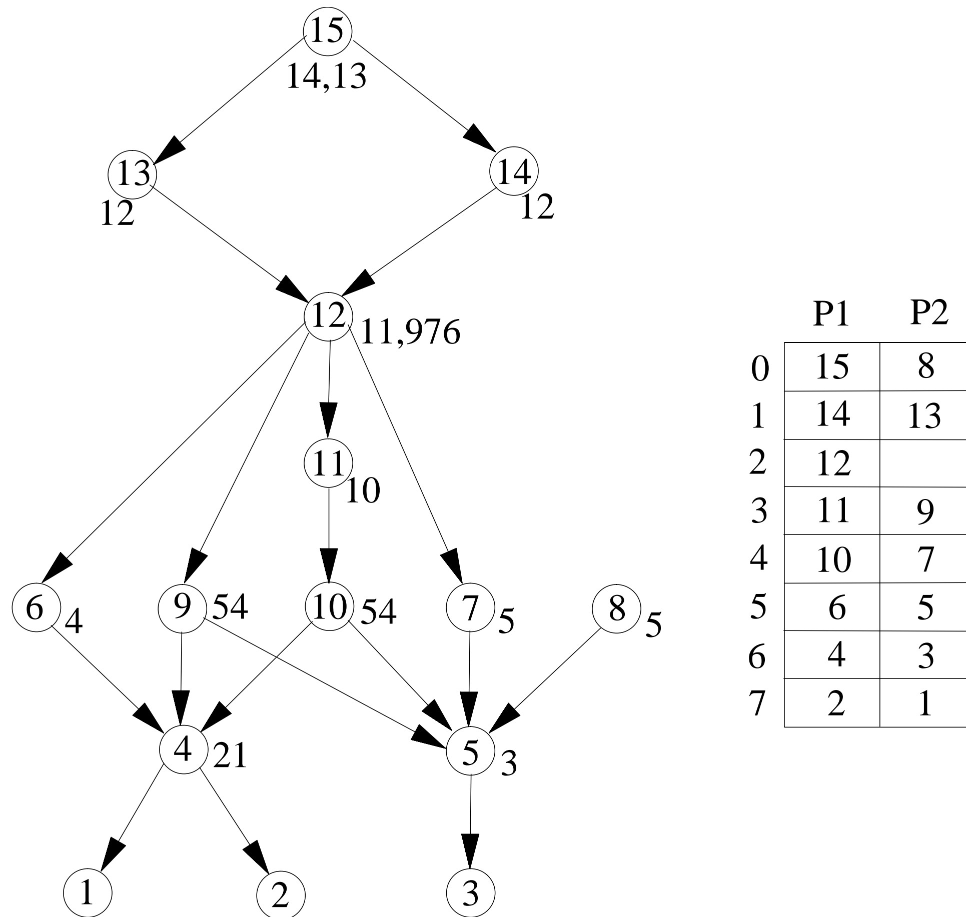
1. All tasks require one unit of time on any processor and no time is required to transfer files. The number of processors is limited, but larger than two.
2. All tasks require one or two units of time on either of two identical processors with no time or contention for transferring files.
3. Variations where the time for executing any task and for transferring any file between arbitrary processors is the same.

Optimal Scheduling in Polynomial Time

Under severe restrictions on the task graph structure (e.g. all tasks have one input or all tasks have one output), task times, and communication, polynomial time algorithms exist, but have narrow application. Coffman and Graham's algorithm for scheduling unit-time tasks on two (SMP) processors is well-known:

1. Assign the label 1 to one of the exit tasks x, so L(x) = 1 (i.e. a task without output). (An *exit task* has no output edges. Likewise, an *entrance task* has no input edges.)
2. Suppose that labels 1, 2, . . ., j - 1 have already been assigned. Let S be the set of tasks without labels that have all of their successors already labeled. For each node x in S define l(x) as follows: Let $y_1, y_2, \ldots, y_k$ be the immediate successors of x; l(x) is the decreasing sequence of integers formed by ordering the set $\{L(y_1), L(y_2), \ldots, L(y_k)\}$. Let x be an element of S such that for all x' in S, l(x) ≤ l(x') (lexicographically). Assign L(x) = j.
3. After using 1. and 2. to label all nodes, schedule tasks in reverse label order. A task should be placed at the earliest possible time that does not violate precedence.

The following example of Coffman/Graham includes the L value for each node, along with the decreasing sequence of successors' L values. Proving the correctness of this algorithm is non-trivial.



	P1	P2
0	15	8
1	14	13
2	12	
3	11	9
4	10	7
5	6	5
6	4	3
7	2	1

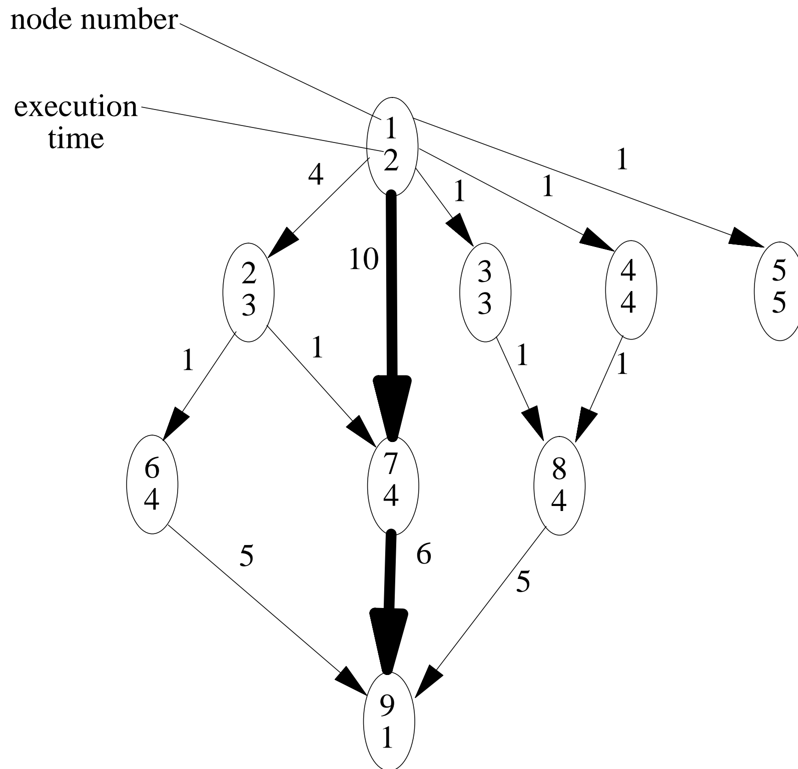
Approximate Scheduling in Polynomial Time

For theoretical investigations, an unbounded number of processors may be assumed. Practical studies emphasize bounds on the number of processors.

List scheduling techniques dominate the research on heuristics for task scheduling. These techniques order (e.g. assign priorities to) the tasks and then apply a priority queue:

1. Place all tasks without predecessors in the PQ.
2. As long as the PQ contains a task do:
    a. Assign the task with highest priority to an available processor (possibly to reduce communication) at the time when all predecessor tasks must have completed.
    b. If the scheduled task is the last immediate predecessor scheduled for any tasks, then place those tasks on the priority queue.

The approach to assigning priorities is what distinguishes the many variations. The following example introduces some graph-based definitions:



node	sl	t-level	b-level	ALAP
1	11	0	23	0
2	8	6	15	8
3	8	3	14	9
4	9	3	15	8
5	5	3	5	18
6	5	10	10	13
7	5	12	11	12
8	5	8	10	13
9	1	22	1	22

sl(i) = maximum execution time of a path including node i through some exit node (static level)
t-level(i) = maximum execution + communication time for a path from an entrance node up to node i (top level)
b-level(i) = maximum execution + communication time for a path including node i through some exit node (bottom level)
ALAP(i) = critical path time - b-level(i), CPT is 23 (As-Late-As-Possible)

The following approaches are representative of the many proposed techniques:

Communication cost of zero (best to worse)

- Highest Level First with Estimated Times (HLFET) - highest priority is given to tasks that are the farthest from a exit node (i.e. use sl(i)), but ties are broken based on the task that has the earliest possible start time.
- Highest Levels First with No Estimated Times (HLFNET) - like HLFET, but assumes tasks have unit time.
- Assign priorities randomly
- Smallest Co-levels First with No Estimated Times (SCFNET) - like SCFET, but assumes tasks have unit time.
- Smallest Co-levels First with Estimated Times (SCFET) - the co-level of each node is the computation time for the longest path from an entry node to the node (i.e. like sl(i), but reversed)

With communication

- HLFET, but with communication costs used in computing distance to exit nodes
- Modified Critical Path (MCP) - tasks with smallest ALAPs are given highest priority

More exotic techniques may use tricks such as 1) replicating tasks to avoid communication, 2) adjusting priorities after some decisons have been made, or 3) allowing tasks to be inserted in the interval between two tasks already in the schedule. Comparing techniques is difficult and usually involves generating random task graphs with specific properties and comparing techniques in terms of:

- Quality of solution - how close is the solution to optimal? How often is the optimal graph produced?
- Ease of computing solution - backtracking and dynamic programming are too slow.
- Good and bad cases.
- How well does technique perform given a large number of processors?

Load Balancing

Static Load Balancing - Processes in fixed set are statically assigned to processors, either at compile-time or at start-up (i.e. partitioning). Avoids the typical 5-20% overhead of load balancing, but is useless when the problem does not divide cleanly such as for problems involving irregularly or unpredictability such as: mesh generation, game playing (chess), and many optimization problems.

Key Issues in Dynamic Load Balancing:

1. Load Measurement - *load index* is a simple measurement usually based on counting ready (and executing) processes on a processor. Other factors (communication, memory requirements, multiple processors at an SMP node) are more difficult to address.

2. Information Exchange - the load at a node is meaningful only when compared to other nodes, often the neighbors. Information exchange may occur in anticipation of load balancing, may be periodic, or may be based on a significant change in a node's load index.

3. Initiation Rule - designed such that benefit exceeds cost. If balancing is initiated by an overloaded node, then designated as *sender-initiated*. If initiated by an underloaded node, then known as *receiver-initiated*. *Symmetrical* policies are also possible.

4. Load Balancing Operation - Defined by having rules for location, distribution, and selection.

   a. Location Rule determines which nodes participate.

   b. Distribution Rule determines the redistribution of load among the participants.

   c. Selection Rule determines the processes to move. A *non-preemptive* rule moves newly spawned processes that have not progressed on the node of the parent processor. A *preemptive* rule can also migrate a process that has progressed.

   Note: For many practical problems it is not necessary to actually migrate processes, especially when a non-preemptive rule is used. Instead, just data that describes a *task* is migrated.

   Note: Even though this sub-area of parallel processing has been called load balancing, practical cases can often emphasize idleness-avoidance over fairness. Having all processors busy between load balancing operations is a reasonable goal.

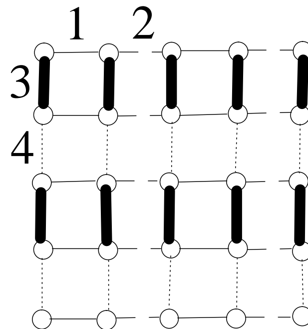
   Note: Since load balancing schemes do not incorporate precedences between tasks explicitly, some scenarios may lead to idle processors later in execution. Dynamic load balancing, however, is usually used in situations where the task graph is not available in advance.

If the information exchange and location rules operate locally, then the technique is called a *nearest-neighbor* algorithm. To move a process several hops, the technique acts as an *iterative* algorithm. Note that the iterations only manipulate load indices; only after these have converged does the redistribution occur.

A *direct* algorithm does not depend on iteration, but depends on having a communication system that supports the increased load. Direct algorithms behave as though the network was a complete graph. Broadcasting and wormhole routing are useful in supporting direct algorithms, but iterative techniques are still often preferable in large systems.

Overview of Techniques:

An *edge coloring* of an undirected graph G assigns each edge a number (color) such that two distinct edges incident to the same vertex will have different colors. Vizing's theorem indicates that the minimum number of colors (the *chromatic index* χ') satisfies degree(G) $\leq \chi' \leq$ degree(G) + 1. (Since deciding whether degree(G) = χ' is NP-complete, degree(G) + 1 is a useful approximation that can be computed (sequentially) in O(|V||E|) time. C code is available on the course web page) For purposes of load balancing on an arbitrary topology, the chromatic index is also known as the *dimension* of the graph. Strategies that address, in turn, a subset of edges assigned a common color are popular. For the 2-d mesh below, four colors are necessary.



Generalized dimension exchange (GDE) is a nearest-neighbor technique that considers each of the χ' edge colors in turn (i.e. each color is considered once in each *sweep*) and rebalances locally from overloaded to underloaded. Whether to split equally (when working with each neighbor) is an important consideration, so an *exchange parameter* λ between 0 and 1 indicates the amount of redistribution being used (theoretically, there could be various exchange parameters, but the extra detail defies practical use). If vertices i and j are connected by an edge with the appropriate color and have loads $w_i < w_j$, then a rebalancing step will change w_i to $(1 - \lambda)w_i + \lambda w_j$ and w_j to $\lambda w_i + (1 - \lambda)w_j$. Dimension exchange was originally proposed as a natural technique (due to the simplicity of the edge coloring) for hypercubes in the late 1980's. For hypercubes, $\lambda = 0.5$ is optimal for convergence (see `GDEcubeSim.c`). For rings with an even number $k > 3$ of vertices, $\lambda = 1/(1 + \sin(2\pi/k))$ is optimal. For a linear array with an even number k of vertices, $\lambda = 1/(1 + \sin(\pi/k))$ is optimal. For tori with even-sized dimensions $k_1 \leq k_2$, $\lambda = 1/(1 + \sin(2\pi/k_2))$ is optimal (see `GDE2dtorusSim.c`). For meshes with even-sized dimensions $k_1 \leq k_2$, $\lambda = 1/(1 + \sin(\pi/k_2))$ is optimal (see `GDE2dmeshSim.c`). (Reference: G. Cybenko, ''Load Balancing for Distributed Memory Multiprocessors'', *Journal of Parallel and Distributed Computing*, 1989.)

Diffusion assumes an ''all port'' communication model and allows an overloaded node to move (diffuse) load to all neighboring underloaded nodes simultaneously. A node that is sending may also receive from some of its neighbors that have an even greater load, typically in an averaging fashion. If N(i) represents the set of neighbors for node i, then a rebalancing step for the *averaging diffusion algorithm* changes w_i to:

$$\frac{w_i + \sum_{j \in N(i)} w_j}{\mathrm{degree}(i) + 1}$$

In the general case for diffusion, each pair of connected processors have a *diffusion parameter* $\alpha_{i,j}$ such that w_i is changed to:

$$w_i + \sum_{j \in N(i)} \alpha_{i,j}\left(w_j - w_i\right)$$

Assuming that all diffusion parameters are set identically, optimal values (in terms of convergence) have been determined for various topologies. For hypercubes, the averaging diffusion algorithm is optimal for convergence. For rings with an even number k of vertices, $\alpha = 1/(3 - \cos(2\pi/k))$ is optimal. For a linear array with k vertices, $\alpha = 1/2$ is optimal. For tori with even-sized dimensions $k_1 \le k_2$, $\alpha = 1/(5 - \cos(2\pi/k_2))$ is optimal. For meshes with even-sized dimensions $k_1 \le k_2$, $\alpha = 1/4$ is optimal.

The term *gradient model* describes a number of global techniques that move load away from overloaded nodes toward the underloaded nodes. In a simple version, the *pressure* of lightly-loaded nodes is set to zero. Other nodes have their pressures set to 1 + minimum pressure among all neighbors. As an example, the following 2-d grid has pressures assigned:

1	2	3	2	3
0	1	2	1	2
1	2	1	0	1
2	3	2	1	2
3	2	1	0	1

Load is now redistributed by moving tasks along the shortest path from each high-pressure node to the nearest zero-pressure node. Conceptually, the flow on a path stops when the low-pressure node will have received a sufficient number of tasks such that its load index should be comparable to its neighbors.

Randomized allocation only addresses new processes by randomly assigning them to a neighboring node. If that node is overloaded, then random assignment is repeated, possibly several times as limited by the *transfer limit*.

Physical optimization takes a fixed set of processes and maps them to a topology in an attempt to reduce communication. If reassignment (due to changes in the set of processes) is done (at less-frequent intervals than other techniques), then process migration may occur. Techniques involving partitioning (a generalization of bisection width) of the set of processes and the set of processors may be applied (*Chaco*, Sandia Nat'l Labs and *Metis*, U. Minnesota are popular partitioners).

Work-stealing is an asynchronous approach in which a processor with an empty run-queue steals work from another processor. Simple approaches include having each processor maintain a *target* variable that cycles work requests around to all other processors. It is also possible to have a global target variable to avoid collisions in stealing or, even more simply, to use randomization for stealing.

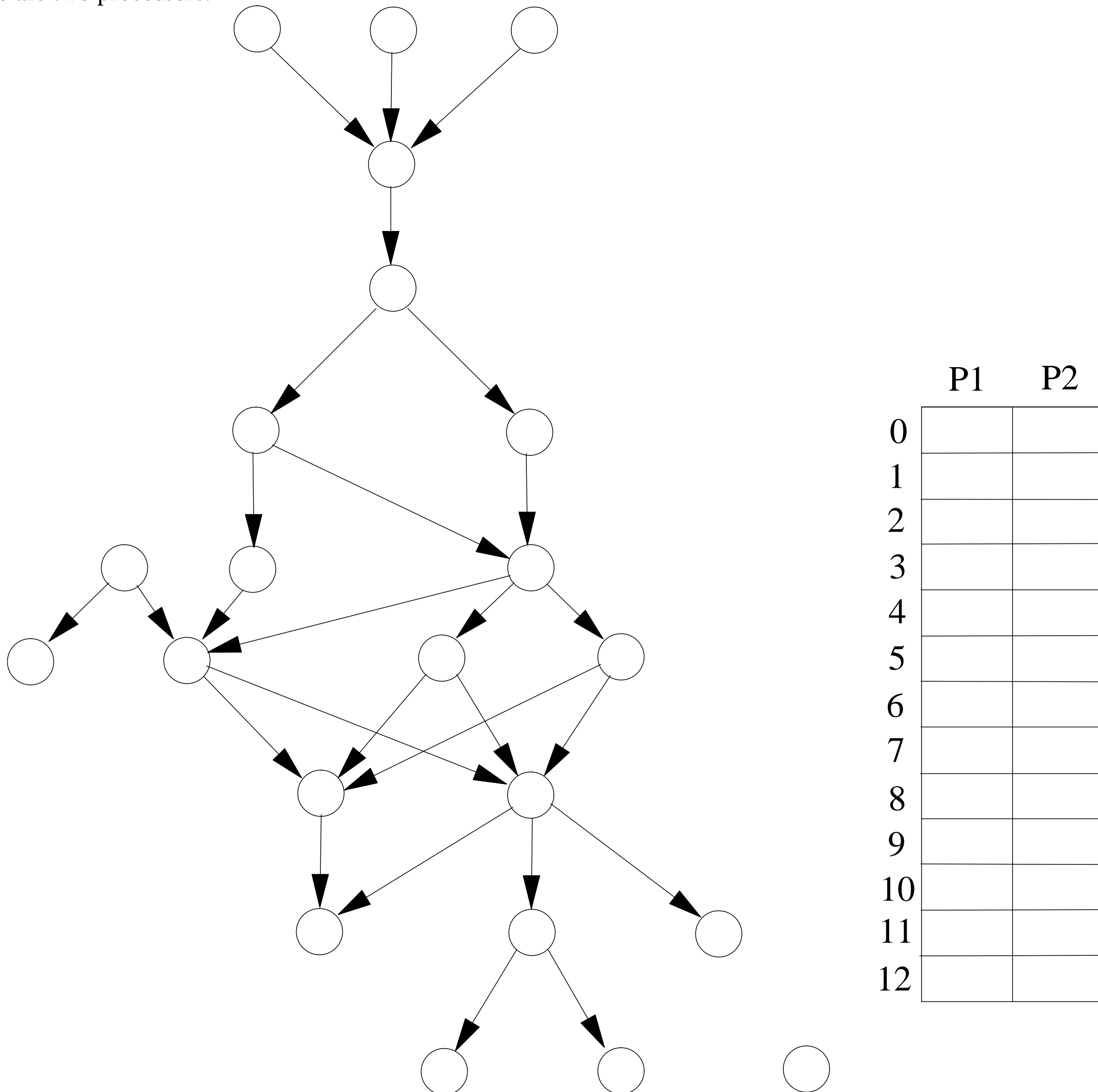
Termination Detection:

Even though the expected number of iterations for GDE and diffusion are predictable, it is important to detect when convergence has occurred for a particular instance. For example, suppose that the Dijkstra-Scholten technique is used. The

initial diffusion tree could be set up artificially before information exchange commences. In this case, some processor acts as the root and all other processors determine an initial parent in the tree. As the information exchange proceeds, a processor is taken as being passive if its load index remains unchanged for a number of exchanges. Of course, if the load index eventually changes as the result of an exchange, then the processor becomes active again (and a parent in the tree must be established). Global termination will be detected when the diffusion tree collapses all the way to the root

Problems: (Provided code may be adapted for the last two problems.)

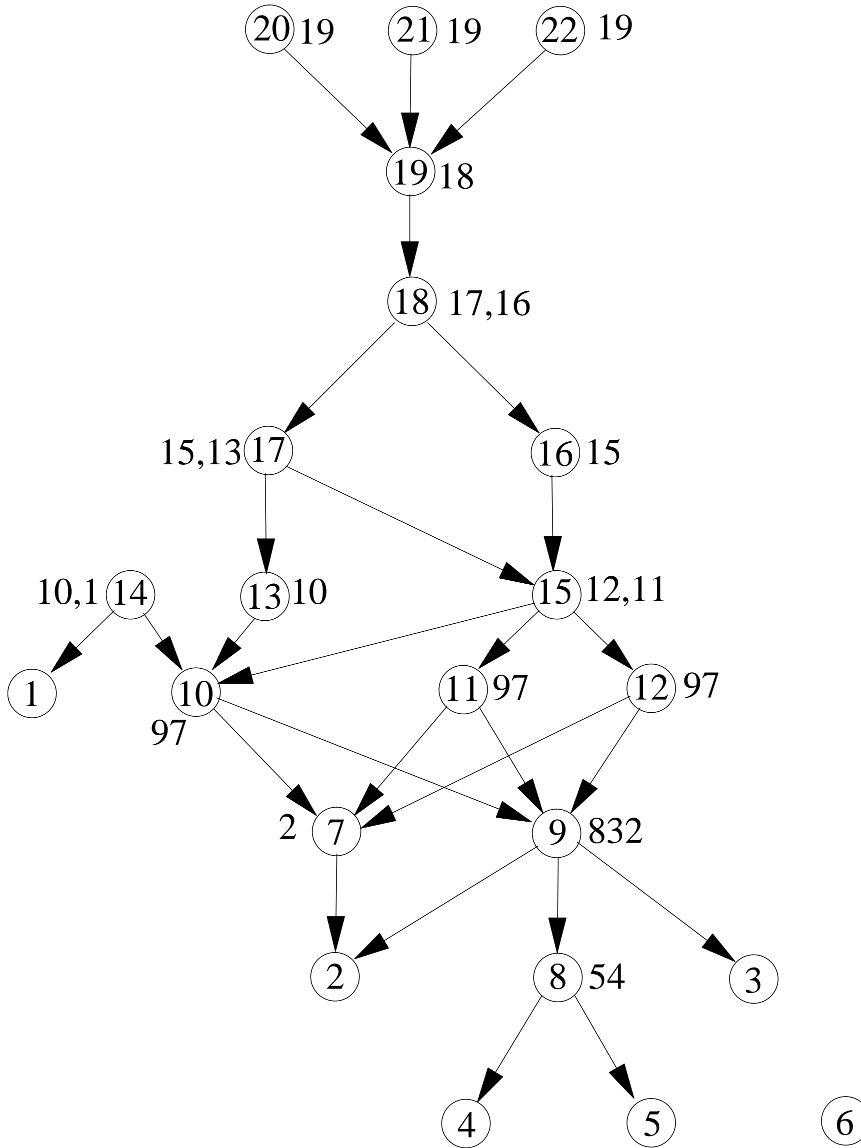
1. Give an optimal schedule for the following task graph assuming that all tasks take unit time, no communication is needed, and there are two processors.



2. Consider a 4-d hypercube that assigns each processor i, $0 \leq i \leq 15$, a load index of 1000i. Apply averaging dimension exchange (i.e. $\lambda = 0.5$) to one sweep and observe that the load becomes balanced.

3. Consider the edge-coloring for the 5 x 5 mesh. Suppose that the rows are numbered top-to-bottom from 0 to 4. Likewise, suppose that the columns are numbered left-to-right from 0 to 4. Now suppose that processor (i, j) is assigned load index

1000(i + j). If the color classes are processed cyclically (as indicated in the diagram) by GDE, how many sweeps are needed?



	P1	P2
0	22	21
1	20	14
2	19	6
3	18	1
4	17	16
5	15	13
6	12	11
7	10	
8	9	7
9	8	3
10	5	4
11	2	
12		