Lecture 4: Mapping and Scheduling

Lecture Outline

(Reading: Chapter 5 of textbook)

1. Introduction: four problems related to implementing algorithms on parallel computers
2. Mapping data to processors or processor arrays and multicomputers
3. Dynamic load balancing on multicomputers
4. Static scheduling on UMA multiprocessors
5. Deadlocks

1. Introduction: four problems related to implementing algorithms on parallel computers

(1) Static mapping of processes to processors, processor arrays or multicomputers

(2) Dynamic load balancing on multicomputers

(3) Task scheduling on multiprocessors

(4) Deadlocks among parallel processes

2. Mapping data to processors or processor arrays and multicomputers

(1) Processor arrays and multicomputers have nonuniform memory structure: access to local memory are much faster than to non-local memory

   a. An important task of parallel algorithm design: maximize number of local memory accesses

   b. Distribution of parallel data structures often dictates which processor is responsible for performing which operation at what time.

   c. An algorithm's data manipulation can often be represented by a graph:

      (a) Each vertex represents a data subset allocated to the same local memory;

      (b) Each edge represents an operation involving data from two data sets;

      (c) These graphs are often regular.

   d. Mapping the algorithm graph into the corresponding graph of the target machine's procesor organization, an important task.

   e. Performance may suffer if the algorithm graph is not a subgraph of the parallel architecture's processor organization.

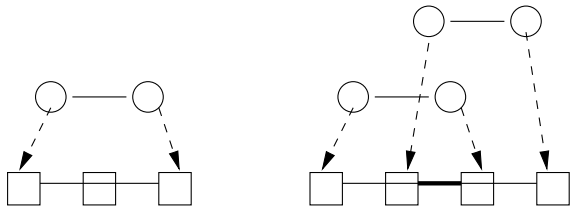


Figure 49: If the algorithm graph is not a subgraph of the target architecture's processor organization, performance of the parallel algorithm may suffer. (a) Dilation: on a multicomputer with store-and-forward message passing, communication time is roughly proportional to the length of the path between the processors. (b) Congestion: on any distributed memory system, every communication link has a fixed bandwidth. Mapping multiple edges of an algorithm graph onto a single communication link can increase communication time (Fig.5-1,p.133).

f. Example 1 (Fig.6-1(a),p.133).
   - Multicomputer using store-and-forward routing, two connected vertices in the algorithm graph map to pair of vertices distance two apart on the target machine.
   - Passing a message from one processor to other will requires twice of the time from one to an adjacent processor.

g. Example 2 (continued, Fig.6-1(b),p.133).
   - Multicomputer using circuit-switching routing, four vertices in the algorithm graph map to four of vertices as shown in Fig.6-1(b).
   - The link between the middle two processors may become a bottleneck.

h. Mapping for popular algorithm graphs (complete binary trees, binomial trees, rings, and meshes) into two popular processor array organizations (2-D mesh and hypercube) are considered.

(2) Definitions

a. **Definition 1**. An **embedding** of a graph $G = (V, E)$ into a graph $G' = (V', E')$ is a function ϕ from V to V'.

b. **Definition 2**. Let ϕ be the embedding function that embeds graph $G = (V, E)$ into graph $G' = (V', E')$. The **dilation** of the embedding is defined as follows:

$$dil(\phi) = \max\{dist(\phi(u), \phi(v))|(u, v) \in E\}$$

where $dist(a, b)$ is the distance between vertices a and b in G'.

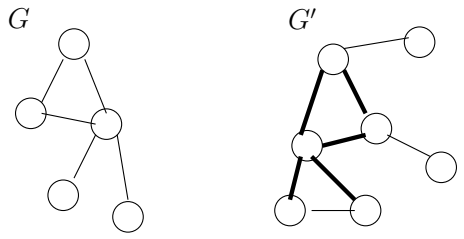


Figure 50: If graph G is a subgraph of G', there exists a dilation-1 embedding of G into G' (Fig.5-2,p.133).

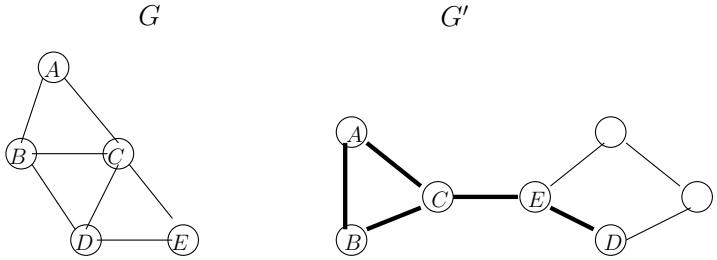


Figure 51: A dilation-3 embedding of graph G into graph G'. The edge (B, D) in G becomes a path of length 3 in G' (Fig.5-3,p.134).

c. Dilation-1 embedding

   * There exists a dilation-1 embedding of G into G' if G is a subgraph of G'.
   * An example dilation-1 embedding: Fig.5-2, p.133.
   * An example dilation-3 embedding: Fig.5-3, p.134.

d. **load** of an embedding: is the maximum number of vertices of G that are mapped to a single vertex of G'.
   Only load 1 embeddings are considered here.

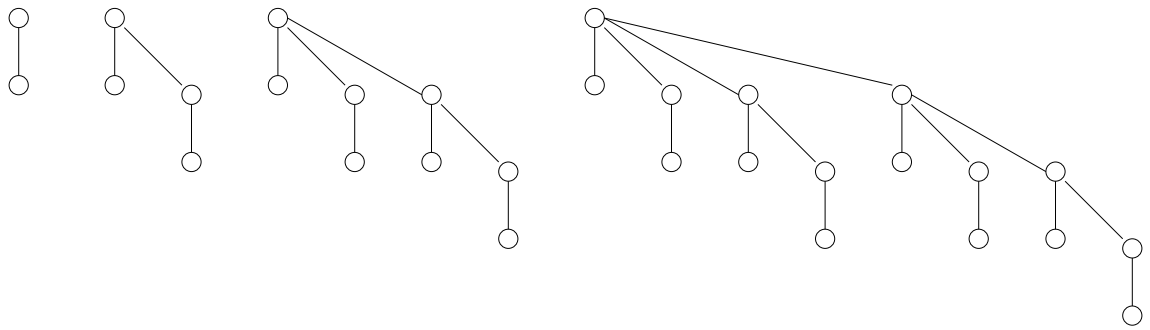


Figure 52: Binomial trees of height one through four (Fig.A-7,p.370).

e. **Binomial trees**: (p.370 & Fig.A-7)

* A binomial tree of height 0 is a single node.
* For $i > 0$, a binomial tree of height i is a tree constructed by connecting the roots of two binomial trees of height $i - 1$ with an edge and designating one of these roots to be the root of the new tree.
* A binomial tree of height n has 2^n nodes.

(3) **Ring into 2-D mesh**

a. Assume that the ring and the mesh have the same number of vertices.

b. A dilation-1 embedding exists if the mesh has an even number of rows and/or columns (A mesh with an odd number of rows and columns has no Hamiltonian circle. Hence dilation-1 embedding is not possible).

c. Example: Fig.5-4,p.134

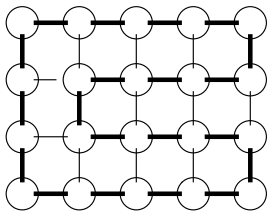


Figure 53: A dilation-1 embedding of a ring into a 2-D mesh having the same number of vertices exists if only if the number of vertices is even (Fig.5-4,p.134).

(4) **2-D mesh into 2-D mesh**

a. Dilation-1 embedding a 2-D mesh into another 2-D mesh (assume both having no wrap-around): conditions?

b. Condition: $A_r \leq M_r$ and $A_c \leq M_c$; or
$A_c \leq M_r$ and $A_r \leq M_c$;

c. Example: Fig.5-5,p.135

(5) **Complete binary tree into 2-D mesh**

a. Example: Fig.5-6,p.135

b. **Theorem 5.1** A complete binary tree of height greater than 4 cannot be embedded in a 2-D mesh without increasing the dilation beyond 1.
Proof: The total # of mesh points k or fewer jumps away from an arbitrary point in a 2-D mesh is $2k^2 + 2k + 1$. The total # of nodes in a binary tree of depth k is $2^{k+1} - 1$. For all $k > 4$, $2^{k+1} - 1 > 2k^2 + 2k + 1$.

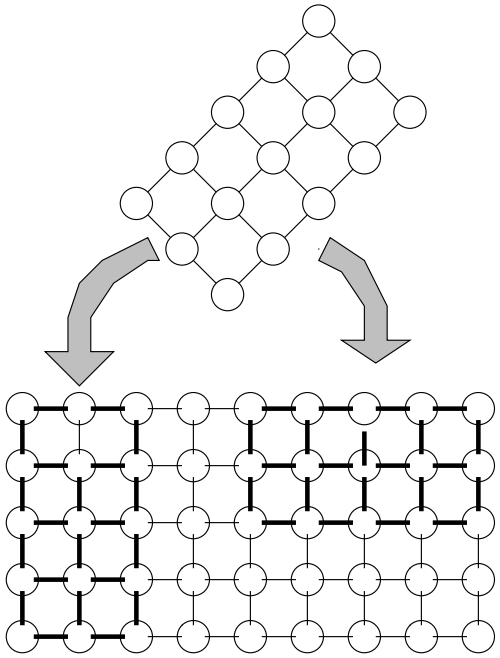


Figure 54: Two dilation-1 embedding of a 2-D mesh into another 2-D mesh (Fig.5-5,p.135).

c. H-trees: Fig.5-7, a common way of embedding a complete binary tree into a 2-D mesh.

d. **Theorem 5.2** A complete binary tree of height n has a dilation $n/2$ embedding in a 2-D mesh.
Proof: Use H-tree. The longest edges in an H-tree are the edges from the root to its two children. These edges have the same length $n/2$ in an H-tree of height n.

e. Example: Fig.5-7,p.136

(6) **Binomial tree into 2-D mesh**

a. Example: Fig.5-8,p.136

b. **Theorem 5.3** A binomial tree of height greater than 4 cannot be embedded in a 2-D mesh without increasing the dilation beyond 1.
Proof: The root node of a binomial tree of height d is connected to d other nodes. On the other hand, no node in a 2-D has more than 4 neighbor nodes.

c. **Theorem 5.4** A binomial tree of height n has a dilation $n/2$ embedding in a 2-D mesh.

d. Example: Fig.5-9,p.137.

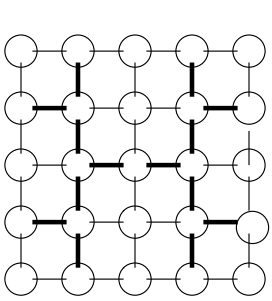


Figure 55: A dilation-1 embedding of a binary tree of height 3 into a 2-D mesh (Fig.5-6,p.135).

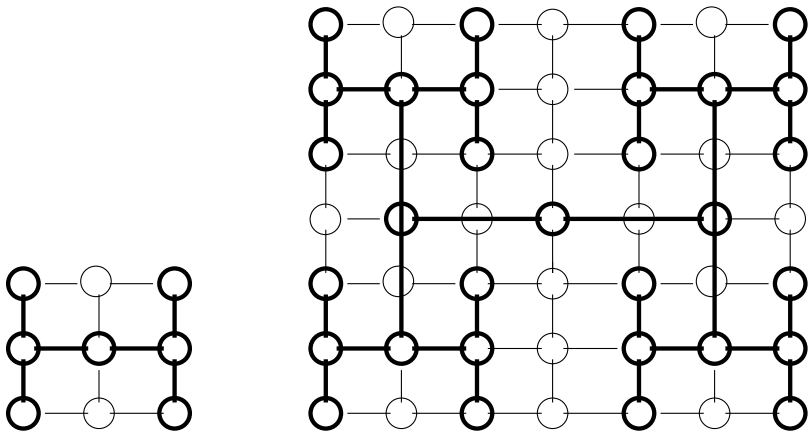


Figure 56: H-trees of height 2 and 4 (Fig.5-7,p.136).

(7) **Embedding graphs into hypercubes**

a. **Definition**. (Cubical graph). A graph G is **cubical** if there exists a dilation-1 embedding of G into a hypercube.

b. **Theorem**. The problem of determining whether an arbitrary graph G is cubical is **NP**-complete (Afrati et al. 1985, Cybenko et al. 1986).

c. **Theorem** (Harel and Moravek, 1972; Livingston and Stout, 1987). A dilation-1 embedding of a connected graph G into a hypercube with n nodes exists if and only if it is possible to label the edges of G with the integers $\{1, \cdots, n\}$ such that:

* Edges incident with a common vertex have different labels.
* In every path of G at least one label appears an odd number of times.
* In every cycle of G no label appears an odd number of times.

Fig.5-10 illusrates the three cases.

(8) **Complete binary trees into hypercube**

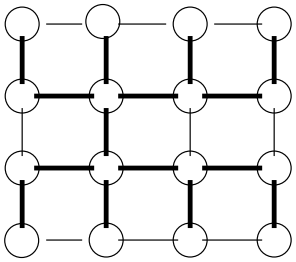


Figure 57: A binomial tree of height 4 embedded in a 2-D mesh. The dilation is 1 (Fig.5-8,p.136).

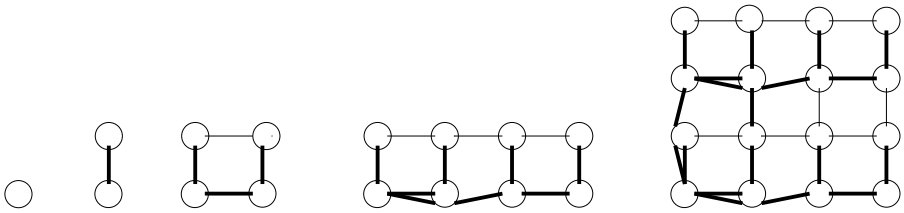


Figure 58: Embeddings of binomial trees into a 2-D mesh. In this scheme an embedding of a binomial tree of height n has dilation $\lceil n/2 \rceil$ (Fig.5-9,p.137).

a. **Theorem 5.7** A dilation-1 embedding of a complete binary tree of height n into hypercube of dimension $n+1$ does not exist if $n > 1$.
Proof: A complete binary tree of height n has $2^{n+1}-1$ nodes. A hypercube of dimension $n+1$ has 2^{n+1} nodes.
   * To embed, the root node of the tree must be mapped onto a node X of the hypercube.
   * Hypercubes are bipartite graphs. Hence half of the hypercube nodes can only be reached from X by following an even number of edges.
   * Similarly, half of the nodes can only be reached from X by following an odd number of edges.
   * If n is odd, then more than half of the nodes of the binary tree are an even distance away from the root of the binary tree. If n is even, more than half of the nodes are an odd distance away from the root.
   * In either case, there is no way to embed the binary tree into the hypercube and keep the dilation at 1, because there are not enough hypercube nodes to accommodate the leaves of the tree while maintaining parity with the interior nodes.

b. **Theorem 5.8** (Nebesky 1974) A balanced binary tree of height n has a dilation-1 embedding into a hypercube of dimension $n+2$.

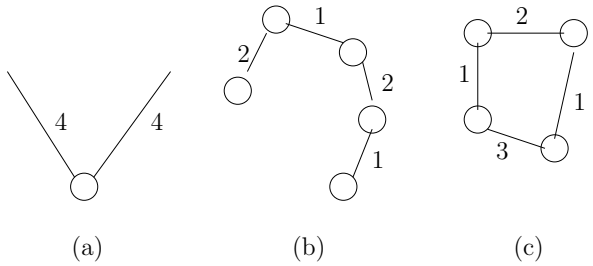


Figure 59: A dilation-1 embedding of a graph G into a hypercube with n nodes does not exist if every possible labeling of edges of G with the integers $1, \cdots, n$ leads to at least one of the following conditions. (a) Two edges incident on the same vertex have the same label. (b) In some path of G no label appears an odd number of times. (c) In a cycle of G at least one label appears an odd number of times (Fig.5-10,p.138).

c. **Theorem 5.9** (Leighton 1992) A complete binary tree of height n has a dilation-2 embedding into a hypercube of dimension $n + 1$, for all $n > 1$.

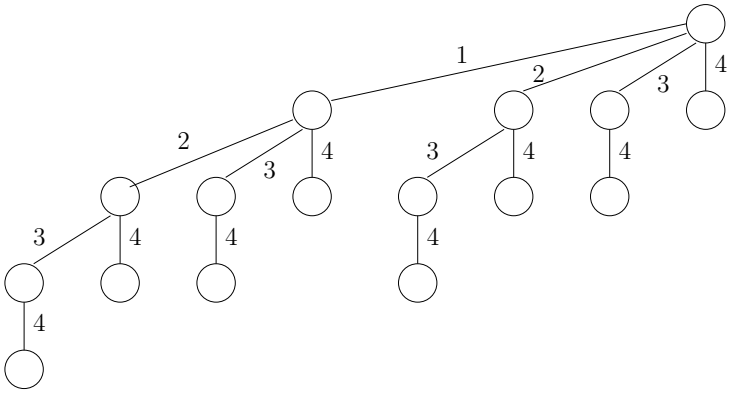


Figure 60: A numbering of the edges of a binomial tree of height 4 that proves the tree can be embedded with dilation 1 into a hypercube of dimension 4 (Fig.5-11,p.139).

(9) **Binomial trees into hypercube** (Fig.5-11,p.139)

a. **Theorem 5.10**. There is a dilation-1 embedding from a binomial tree of height n into a hypecube of dimension n.

b. **Proof**: Organize the subtree such that the nodes that are roots of larger subtrees appear to the left of nodes that are roots of smaller subtrees (Fig.5-11).

* Given the edge to the leftmost child of the root the label 1, the edge to the 2nd child of the root label 2, and so on. The rightmost child is then labeled n.
* For the remaining interior nodes of the tree, if the edge above the node is label i, then the k children will be labeled $i+1, i+2, \cdots, i+k$ from left to right.
* Must show all three conditions in Theorem 5.6 holds.
  - First, show no two edges incident on the same vertex have the same label. Clear by the construction
  - Second, show every path in the graph has at least one edge whose label appears an odd number of times.
* Two kinds of paths: paths from ancestors to decendents and paths between vertices that share a common ancestor.
* Every edge on a path from a node to one of its decendents has a unique label, so there is at least one edge whose label appears an odd number of times.
* On a path between two vertices that share a common ancestor, the edges work their way up the tree to the common ancestor, then back down the tree. Two edges on the path are incident to the common ancestor. The label assigned to the left edge is smaller than any other edge labels on that path.
* Hence this label appears only once, the second condition is satisfied.
* The third condition is trivially satisfied.

(10) **Rings and meshs into hypercube**

3. Dynamic load balancing on multicomputers

(1) Processors' workloads may be unbalanced.

(2) Dynamic load balancing: the process of making changes to the distribution of work among the processors at run time.

a. Successfulness of a dynamic load balancing algorithm is measured by the net reduction of execuction time.

b. May actually increase the execution time due to the extra overhead.

(3) Classification of dynamic load balancing algorithms

a. Load balancing can be either *sender initiated* or *receiver initiated*.

* In sender initiated algo, a processor with too much work to do sends some work to other processors.
* In receiver initiated algo, a processor with too little to do takes some work from some other processors.
* Experiments showed that sender initiated algo perform better for light to medium workload per processor, while the receiver initiated perform better for heavy workload per processor.

b. *Centralized*. A particular node maintains the global state information and directs load balancing.
May not scale well (information increases dramatically).

c. *Fully distributed*. Each processor maintains a partial information.
Lower scheduling overhead.
However workload may not actaually balance.

d. *Semi-distributed*. Processors are divided into regions. Each region executes a centralized algorithm.

4. Static scheduling on UMA multiprocessors

(1) Typical UMA multiprocessor scheduling:

a. Processes in need of CPU cycles are maintained in a single ready queue.

b. Each physical processor accesses this queue to run the next process.

c. Binding of processes to processors are not tight – a single process may receive CPU cycles from many processors

d. This type of dynamic schedulings are provided by OS.

(2) Static scheduling

a. Static scheduling can sometimes result in more efficient algorithms

b. Static scheduling can generate only one process per processor, reducing process creation, synchronization, and termination overhead.

c. Static scheduling can be used to predict speedup (assume preemption of processes is not allowed).

(3) Deterministic models

a. The precedence relations among the tasks and the execution time needed by each task are fixed and known before the schedule is devised (usually at compiler time).

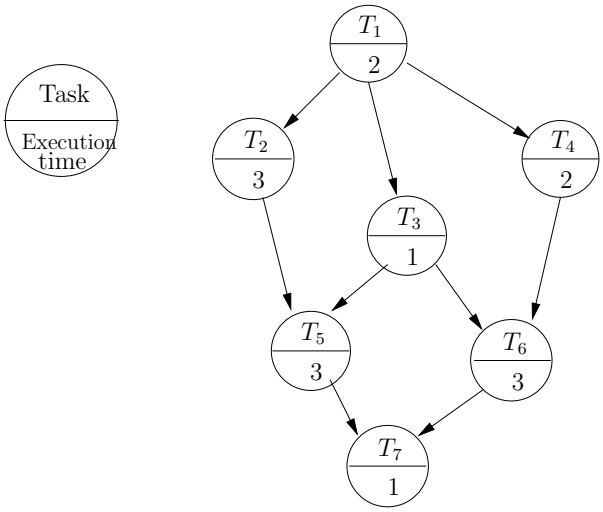


Figure 61: A task graph. Each node represents a task to be performed. A directed arc from T_i to T_j indicates that task T_i must complete before task T_j begins (Fig.5-15,p.144).

b. Not realistic (variations in execution time caused by interrupts, contention for shared memory and etc. are ignored); simple.

c. Tasks and task graphs.

(a) A task graph consists of a set of tasks $T_1, T_2, \cdots, T_n$, each with a known execution time $t_1, \cdots, t_n$.

- A partial ordering $<$ exists among the task names. T_i is a *predecessor* of T_j and T_j is a successor of T_i if $T_i < T_j$. Tasks with no predecessors are called *initial tasks*, and tasks with no successors are called *final tasks*.
- A set of tasks are *independent* if for every pair of tasks T_i and T_j, neither is the predecessor of the other.
- The *width* of the task graph is the size of the maximal set of independent tasks. A *chain* is a totally ordered task graph.
- The *length* of a chain is the number of tasks in the chain.
- The *level* of a task T in a task graph is the maximum chain length in G from an initial task to T.
- The *depth* of a task graph G is the maximal level of any task in G.

(b) **Example.** Fig. 5-15, p.144.

d. *schedules* and *Optimal schedules*

* A *schedule* is an allocation of tasks to processors.
* *Optimal schedule*: the one with minimum exec time.

e. *Gantt Chart* for determinitic models.

* It indicates the time each tasks spends in execution as well as the processor on which it executes.
* It graphically illustrates the untilization of each processor
* **Example.** Fig. 5-16, p145.

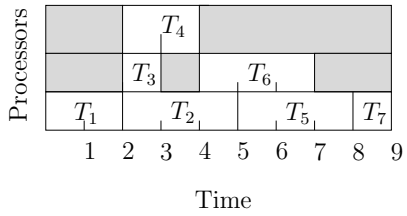


Figure 62: A Gantt chart illustrating a schedule for the task graph of Fig.5-15 (Fig.5-16,p.145).

f. The scheduling problem for deterministic model is *NP-Hard*, if there are more than two processors.

g. Graham's list scheduling algorithm

(a) **Theorem.** (Graham [1972]) Given a set of p identical processors and a deterministic model, we can always schedule the tasks to processors in polynomial time such that the schedule is not more than twice that required by the optimal schedule.

(b) The algorithm. Let $T = \{T_1, T_2, \cdots, T_n\}$ be a set of tasks. Let $\mu : T \to (0, \infty)$ be a function that associates an execution time with each task. The tasks T are associated with a partial ordering $\prec$. Let L be a list of tasks in T.

i. Wheneven a processor has no work to do, it instantaneously removes from L the first task ready; that is, an unscheduled task whose predecessors under $\prec$ have all completed execution.

ii. If two or more processors simultaneously attempt to execute the same tasks, the processor with lowest index succeeds, and the other processors look for another suitable task.

(c) Fig.5-16 is actually the Ghantt graph of applying the Graham's list-scheduling algorithm. The given list L is $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$.

(d) Contrary to intuition, increasing the number of processors, decreasing the execution times of one or more tasks, or eliminating some of the precedence constraints *can actually increase* the length of the schedule produced by Graham's list schedulign algorithm.

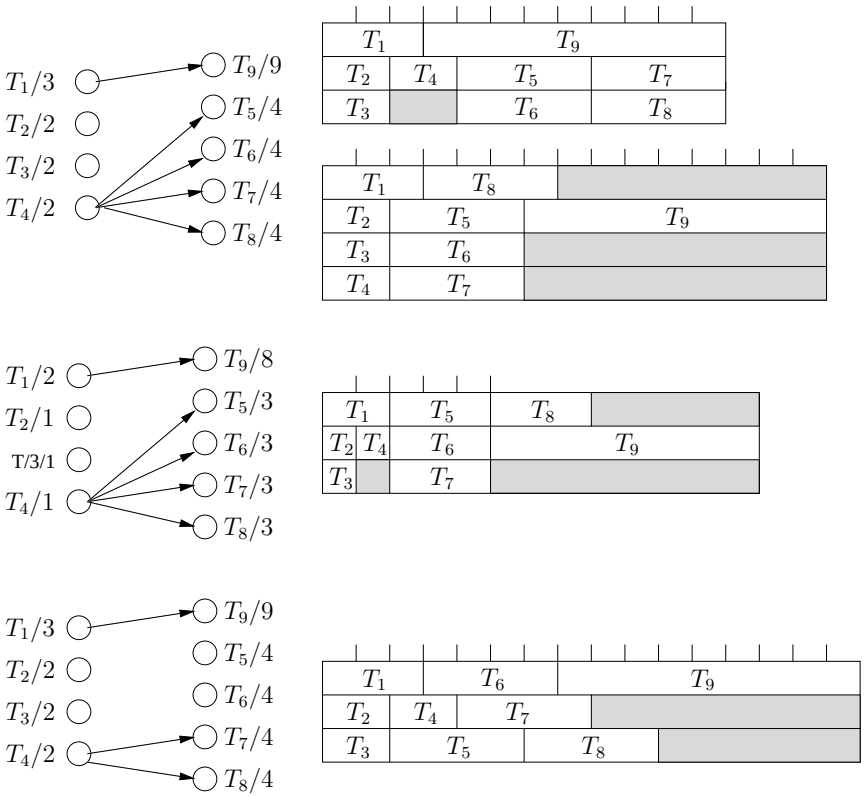


Figure 63: A series of examples illusratting that increasing the number of processors, decreasing the execution time of one or more tasks, or elimilating some of precedence constraints can actually increase the length of the schedule generated using Graham's heuristics (Graham 1972). In each case the priority task list is the same $L = \{T_1, T_2, \cdots, T_9\}$ (Fig.5-17,p.146).

(e) Fig.5-17 shows three different schedules of applying Graham's algorithm to a set of nine tasks.

i. The first schedule is generated from a list of nine tasks

h. Coffman-Graham scheduling algorithm

(a) Bases of Graham's algorithm: a list of tasks with priority associated with the list. How to construct this list is critical.

(b) Coffman-Graham's scheduling algorithm provides a method to construct such lists under the assumption that all tasks take the same amount of time. The algorithm first constructs a list and then applies Graham's list scheduling algorithm to the list.

(c) Assumptions and notations.

· Let $T = T_1, T_2, \cdots, T_n$ be a set of unit time tasks. Assume that the number of processors is p and $\prec$ be a partial ordering on T that specifies which tasks must be executed before other tasks begin.

- If $T_i \prec T_j$, we call that T_i is an *predecessor* of T_j and the latter an *immediate successor* of the former. Let $S(T_i)$ denote the set of immediate successors of task T_i.
- Let $\alpha(T)$ be an integer label assigned to T. $N(T)$ denotes the decreasing sequence of integers formed by ordering the set $\{\alpha(T')|T' \in S(T)\}$.

(d) The algorithm. Let $T = T_1, T_2, \cdots, T_n$ be

i. Choose an arbitrary task T_k from T such that $S(T_k) = \emptyset$, and define $\alpha(T_k)$ to be 1.

ii. for $i \leftarrow 2$ to n do

a. R be the set of unlabeled tasks with no unlabeled successors.

b. Let $T*$ be the task in R such that $N(T*)$ is lexicographically smaller than $N(T)$ for all T in R (breaking ties arbitrarily).

c. Let $\alpha(T*) \leftarrow i$.

endfor

iii. Construct a list of tasks $L = (U_n, U_{n-1}, \cdots, U_2, U_1)$ such that $\alpha(U_i) = i$ for all i where $1 \leq i \leq n$.

iv. Given $(T, \prec, L)$, use Graham's list scheduling algorithm to schedule the tasks in T.

(e) Example: Fig.5-18, p.148.

(4) Nondeterministic models

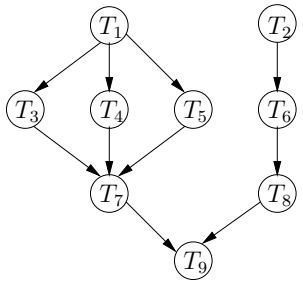
a. The execution time is not predefined. Instead, the execution time for each task is represented by a random variable. Scheduling problem becomes more difficult.

b. Robinson's algorithm for estimating the time needed by a *simple* parallel algorithm.

(a) **Definition.** (Simple task graph) Given a task graph G on n tasks $T_1, \cdots, T_n$, let $x_1, \cdots, x_n$ be n variables, and $C_1, C_2, \cdots, C_m$ be all the chains from the initial to final tasks in G. For any chain C_i consisting tasks $T_{i_1}, T_{i_2}, \cdots, T_{i_j}$ (in that order), let $X_i = x_{i_1} x_{i_2} \cdots x_{i_j}$. G is a **simple task graph** if the polynomial $X_1 + X_2 + \cdots X_m$ can be factored so that every variable x_i appears exactly once.

(b) **Example**. Fig.5-19, p.149.

(c) **Theorem 5.13** (Robinson, [1979]) Given a simple task graph G, if the (a) number of processors exceeds the width of G; (b) the tasks are independent; (c) the depth of G is L; (d) the execution time of all



T_2	T_6	T_4	T_7	T_9
T_1	T_3	T_8		
	T_5			

Step 1 of Algorithm:
Task T_9 is the only task with no immediate successor. Assign 1 to $\alpha(T_9)$.

Step 2 of Algorithm:
$i = 2$. $R = \{T_7, T_8\}$. $N(T_7) = \{1\}$ and $N(T_8) = \{1\}$, trying for lexicographically smallest. Arbitrarily choose task T_7 and assign 2 to $\alpha(T_7)$.

$i = 3$. $R = \{T_3, T_4, T_5, T_8\}$, $N(T_3) = \{2\}, N(T_4) = \{2\}, N(T_5) = \{2\}$ and $N(T_8) = \{1\}$. Task T_8 is lexicographically smallest. Choose it and assign 3 to $\alpha(T_8)$.

$i = 4$. $R = \{T_3, T_4, T_5, T_6\}$, $N(T_3) = \{2\}, N(T_4) = \{2\}, N(T_5) = \{2\}$ and $N(T_6) = \{3\}$. Tasks T_3, T_4, and T_5 all tie for lexicographically smallest. Arbitrarily choose T_4 and and assign 4 to $\alpha(T_4)$.

$i = 5$. $R = \{T_3, T_5, T_6\}$, $N(T_3) = \{2\}, N(T_5) = \{2\}$ and $N(T_6) = \{3\}$. Tasks T_3 and T_5 tie for lexicographically smallest. Arbitrarily choose T_5 and and assign 5 to $\alpha(T_5)$.

$i = 6$. $R = \{T_3, T_6\}$, $N(T_3) = \{2\}$ and $N(T_6) = \{3\}$. Set $N(T_3)$ is lexicographically smaller; Assign 6 to $\alpha(T_3)$.

$i = 7$. $R = \{T_1, T_6\}$, $N(T_1) = \{6, 5, 4\}$ and $N(T_6) = \{3\}$. Set $N(T_6)$ is lexicographically smaller; Assign 7 to $\alpha(T_6)$.

$i = 8$. $R = \{T_1, T_2\}$, $N(T_1) = \{6, 5, 4\}$ and $N(T_2) = \{7\}$. Set $N(T_1)$ is lexicographically smaller; Assign 8 to $\alpha(T_1)$.

$i = 9$. $R = \{T_2\}$. Choose task T_2 and assign 9 to $\alpha(T_2)$.

Step 3 of Algorithm:
$L = \{T_2, T_1, T_6, T_3, T_5, T_4, T_8, T_7, T_9\}$.

Step 4 of Algorithm:
Schedule is result of applying Graham's list-scheduling algorithm to task graph T and L.

Figure 64: Example of the Coffman -Graham scheduling algorithm. The Gantt chart is the result of applying the Coffman-Graham algorithm to the task graph.

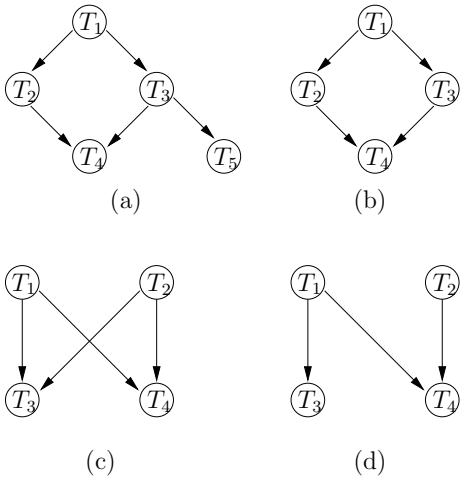


Figure 65: Simple and nonsimple task graphs. (a) Simple task graph: $x_1x_2+x_1x_3x_4+x_1x_3x_5 = x_1[x_2 + x_3(x_4 + x_5)]$. (b) Simple task graph: $x_1x_2x_4 + x_1x_3x_4 = x_1(x_2 + x_3)x_4$. (c) Simple task graph: $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 = (x_1 + x_2)(x_3 + x_4)$. (d) Nonsimple task graph (Fig.5-19,p.149).

m_j tasks on level j is a random variable with mean μ_j and standard derivation σ_j, then

$$\Sigma_{1\le j\le L}\mu_j \le E(t_G) \le \Sigma_{1\le j\le L}(\mu_j + \frac{m_j - 1}{\sqrt{2m_j - 1}}\sigma_j)$$

(4) Nondeterministic models

(d) **Example.** (p.150)

i. A sorting algorithm sorting $n = kp$ values on a p processor machine.

ii. The algorithm is partitioned and has two phases. In the first phase each of the p processors sorts $k = n/p$ values. After all the processors synchronize, each processor merges $k = n/p$ values.

iii. Assume that the sort used in the first phase is the best knonw sequential sorting algorithm, with an expected execution time $3k \log k+200$ and a standard derivation of $2\sqrt{k}$ to sort k values.

iv. Furthermore, assume that the merge algorithm merges k values with an expected time $4k \log k + 100$ and a standard derivation of $\sqrt{k}$. According to the theorem,

$$E(t_G) \ge 3\frac{n}{p}\log\frac{n}{p} + 200 + 4\frac{n}{p}\log\frac{n}{p} + 100$$

$$7\frac{n}{p}\log np + 300$$

$$E(G) \leq 7\frac{n}{p}\log np + 300 + \frac{(p-1)3\sqrt{n}}{\sqrt{2p-1}}$$

v. The expected speedup can be obtained by dividing the expected execution time of the sequential algorithm ($3n \log n + 200$) by the expected execution time of the parallel algorithm.

vi. Fig.5-20,p.150 shows the expected speedup

5. Deadlocks

- **Definition.** A set of processes are **deadlocked** (or in a deadlock state) if each process is waiting for an event that can only be caused by one of the waiting processes.
- **Example.**
  Process 1: $\cdots$; lock(A); $\cdots$; lock(B); $\cdots$
  Process 1: $\cdots$; lock(B); $\cdots$; lock(A); $\cdots$
- **Necessary conditions for deadlocks.** (a) Mutual exclusion; (b) Nonpreemption; (c) Holding and waiting; (d) Circular waiting;
- (a) Deadlock detection and resolution; (b) Deadlock avoidance; (c) Deadlock prevention.