A Fast Algorithm for Near-Optimal Static Scheduling of Acyclic Graphs to Multiprocessor Systems

Douglas M. Pase

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-030

July, 1988

A Fast Algorithm for Near-Optimal Static Scheduling of Acyclic Graphs to Multiprocessor Systems

Douglas M. Pase

Oregon Graduate Center

Department of Computer Science and Engineering

19600 NW Von Neumann Drive

Beaverton, Oregon 97006

Abstract

The *Precedence Constrained Scheduling Problem* has long been known to be intractible [9]. We extend the basic problem to account for "practical" considerations, such as non-uniformly weighted tasks and the architectural environment. We then present a fast algorithm which provides a near-optimal solution to the extended problem. Our algorithm is based on the well known Program Evaluation and Review Technique [5]. It is adaptable to many different types of architectures, ranging from shared memory systems with homogeneous processing elements to message passing systems with irregular communication networks and non-homogeneous processing elements. Our algorithm is able to account for both communication latency and channel capacity, as well as irregular resource usage on processing elements. It is also reasonably quick, having time complexity of $O(n^2)$ to $O(n^3)$, depending on the nature of the application graph.

1. Introduction

Many multi-processor scheduling problems are, in general, **NP**-Complete. Here we examine a form of scheduling of which the *Precedence Constrained Scheduling* problem [4,9] is a special case. Precedence constrained scheduling is:

> Let T be a set of tasks, each having length $\ell(t) = 1$ for each $t \in T$. Let $m \in Z^+$ be a number of processors, $\prec$ be a partial order on T, and $D \in Z^+$ be a deadline. Is there an m-processor schedule σ for T that completes before the deadline D and obeys the precedence constraints, i.e., such that $t \prec t'$ implies $\sigma(t)+\ell(t) \leq \sigma(t')$?

The problem is more useful to us if we reword it as:

> What is an m-processor schedule σ for T that minimizes running time and obeys the precedence constraints, i.e., such that $t \prec t'$ implies $\sigma(t)+\ell(t) \leq \sigma(t')$?

Ullman [9] proved this problem to be **NP**-Complete. Intractible though it is, few applications or architectures are so uniform that this would be considered a good description of multiprocessor scheduling on real systems. We extend the problem further to consider a more realistic model of task scheduling on multiprocessor systems, and present a heuristic to solve the extended problem. The heuristic we present has, at worst, complexity of $O(n^3)$, where $n = |T|$. If certain reasonable restrictions are placed on the graph, the complexity may be reduced to $O(n^2)$.

We extend precedence scheduling in three ways.

- We allow non-homogeneous task lengths, i.e. $\ell(t) \in Z^+$. We also allow each task to require more than a single resource.
- The partial order is induced by a communication pattern, which we represent by $M \subseteq T \times T$. M might be considered as a set of messages, where $(p,c) \in M$ implies that task p is a parent task to c, and that p must communicate with c before c may begin. The size of the message may also vary for different elements of M. The partial order $\prec$ would then be defined as:

$$(\forall t_1, t_2 \in T)\ t_1 \prec t_2 \text{ iff } (t_1, t_2) \in M \text{ or } (\exists t \in T) \text{ s.t. } (t_1, t) \in M \text{ and } t \prec t_2$$

- The architecture which executes the *set of tasks* (*refered to as a job, or application graph*) need not be uniform. Each processor may have a different set of resources which it is able to use with differing efficiencies. Communication between two processors may require different resources and have different efficiencies depending upon the set of processors involved in the information transfer. This may include, but is not limited to, the usage of CPU resources and communication co-processors.

Although we have included many things in our model, there are a number of restrictions which remain. Most notably,

- Task preemption is not allowed — tasks, once started, must run to completion. There is no technical reason why our heuristic could not permit task preemption, but we do not pursue that question here.

- M defines a partial order; the graph it represents is acyclic. Loops, recursion, and conditional execution are not allowed.
- The graph is statically defined, that is, all necessary information about task execution lengths and communication requirements is known in advance.

We emphasize that the solution we present is a heuristic. In many cases a complete solution is not needed if a good approximation is available, especially if the complete solution may only be obtained at great cost. It is not our purpose here to completely solve our extended scheduling problem, but only to explore a fast algorithm which yields near-optimal schedules for it.

2. Application Graph Scheduling

Throughout this paper we assume that the scheduling occurs in reverse order of execution, although for the most part is could just as easily have proceeded forward. The general algorithm we propose is:

```
thread the application graph
while ( some task remains unscheduled )
        analyze the application graph
        select a task for scheduling ($t$)
        find the "best" processor ($p$) for $t$
        schedule $t$ on $p$
end while
```

2.1. Application Graph Threading

Graph threading connects each node in the application graph with bidirectional links to its successor and predecessor in a topological ordering. The ordering guarantees that a traversal of the graph will proceed in a topological order, i.e., in such a way that no node is visited until all its predecessors [successors] have been visited. This decreases the time required for the graph analysis, which must traverse the graph in a topological order. Efficient topological sorts are well known (see [6]).

2.2. Application Graph Analysis

We base our application graph analysis on the well known Program Evaluation and Review Technique (PERT) analysis method [5]. PERT analysis is a simple two-pass analysis which finds for each task in the graph a) the earliest starting time (EST), b) the latest starting time (LST), and c) the slack, or leeway one has in scheduling the task. Slack is simply the difference between LST and EST. Tasks whose slack is zero (i.e. LST = EST) are said to be on the critical path. The EST and LST are both calculated assuming the best possible schedule, so as long as every task is started at some time between its EST and its LST, the job will be completed as quickly as possible. Clearly, within a given graph there may be more than one critical path. It is also relatively simple to show that all critical paths within a graph must have the same length.

PERT analysis ignores a number of factors for which we must compensate. The first is that each task is representable by a single value which is referred to as its weight, or length. Weight is often thought of as an estimate of the total time required to complete the task. If each task only requires a single type of service, this is not an unreasonable assumption. If tasks require multiple services (e.g. integer and floating point operations), the meaning of a single weight value becomes less clear, especially when the relative speeds of the different servers (e.g. integer ALU or floating point co-processor) vary from processor to processor.

PERT analysis ignores all available information about the environment in which the task is performed. It assumes that a task may begin the instant all of its immediate predecessors have completed. If there is any delay between tasks, it must somehow be accounted for in one or more of the task weights. PERT offers no facility to account for weights which vary from server to server — either because of varying distances in communication or non-homogeneous servers.

In order to satisfy PERT's need for a single task weight, all the different resources required by a task must be combined to form a single "aggregate" weight, or execution time.

This is the one area which permits "tuning". For example, if communication time varies as a function of the distance between sender and receiver, some weighting function must be chosen to reflect the cost of communication between tasks which have not yet been assigned to any processor. If the weighting function assigns weights which are too heavy relative to the computation, the perceived cost of communication will be heavier than in reality, and tasks will tend to be packed serially on processors when greater speed could have been attained through greater distribution. On the other hand, assigning weights which are too light would cause the tasks to be distributed excessively, and the communication system would become the limiting factor in the computation. The correct weighting function would balance the communication and computation systems in such a way that, on the average, both were equally utilized. Note that the correct weighting would not wholly depend upon the architecture characteristics, but also upon the locality obtained by the scheduling algorithm.

2.3. Task Selection for Scheduling

To understand the intuition behind this heuristic, it is important to look at the objectives which motivate processor scheduling, and examine the information PERT analysis provides in that light. Our algorithm attempts to find schedules for which the total time from start to finish is at or near a minimum. As mentioned earlier, the algorithm starts from the bottom of the graph and schedules tasks in reverse order of execution. We compare tasks to determine which has the highest priority of those that remain unscheduled.

The EST, LST, and slack each give a different view of a task's priority. The EST measures the earliest time that a task may begin without increasing the job running time. If a task is started before the EST, each of its parent tasks must be started earlier also. Because some of the parent tasks are on the critical path, the job execution time will be increased. The LST measures the latest time at which a task may begin without affecting job running time. Starting a task after its LST increases the computation length in the same way that starting a task before its EST does. Slack measures the freedom available in scheduling a task — greater slack means greater flexibility.

In the process of scheduling an application graph, we iteratively fix the starting time and processor mapping of each task in the graph. By scheduling in reverse order of execution, we fix the termination time of the computation and seek the latest start time possible. Because the scheduling takes place in reverse order, the EST is the critical value, since delaying the scheduling of a task will push its start time backwards. If the start time is pushed backwards beyond its EST, the job execution time will take longer than if it is not. The task with the largest EST is selected as the next task to be scheduled, since *it has the least flexibility in when it can be scheduled without delaying the job execution time*. Of all unscheduled tasks, delaying the task with the largest EST (by scheduling another task first) will cause the greatest known impact on the job execution time.

We recognize that this may not always yield the best results possible. Delaying one task which would otherwise be selected could leave holes in the schedule for other tasks later on which would fit better in the schedule. This could in turn provide an overall reduction in the total job execution time. However, finding an algorithm which would always compute the optimal schedule in polynomial time is not possible unless **P** = **NP**, which at the time of this writing is still an open problem [4].

2.4. Processor Selection

Processor selection for a given task is performed by the function *select_PE*. *Select_PE* estimates the task's start time for each processor $p \in P$, and selects the processor with the best time. The estimated start time is the latest possible start time which the task could assume without altering the schedule which has been previously determined. Because the schedule is determined in reverse order, the "best" start time is the latest (largest) start time.

```
select_PE(task) =
        select processor s.t. $(\forall p \in P)$
        $max\_start\_time(task,p) \leq max\_start\_time(task,processor)$

        return processor
```

2.4.1. Task Starting Time

Max_start_time finds the latest possible start time for task t on processor p. It does so by finding the latest possible finish time (f) for t, then finding a slot in the schedule which allows t to finish on p no later than f.

```
max_start_time(task,processor) =
        finish ← min_latest_finish(task,processor)
         start ← start_time(task,processor,finish)

        return start
```

Once the latest possible finish time is found for t on p, the schedule for p, σ_p, is searched for the latest available slot in which the completion time for t does not exceed the latest completion time. This is done by the function *start_time*. We assume the slot must be large enough to run task t to completion without preemption, but nothing about the algorithm requires this to be so. Preemption could conceivably improve the schedule further but we do not pursue that possibility here. The latest start time for t on p is determined when the slot is found. The implementation for *start_time* depends heavily on the representation used for σ, and on the architecture of the network N. A simple approach to representing the schedule is to use a list; searching for an empty slot would simply be a list traversal.

A general representation for N, however, is not quite so easy. There is so much variation between system architectures that a discussion of N's representation would be tantamount to a discussion of all of the different architectures. However it is represented, a function must be available to *start_time* which compares a task with a slot and returns a) whether the task will fit within the slot, and b) what its starting time would be if it were scheduled within the slot. The function must account for all the resources within each processor if it is to provide an accurate schedule. It may even need to allow for separate schedules for each resource if the resources are sufficiently independent. The function is defined as:

```
start_time(task,processor,finish) =
    for each resource $r$ used by task
        find the duration of $r$'s usage on processor
        search $\sigma_r$ for the latest slot which would
            allow task to complete before time finish

    return the earliest start time of all slots used by task
```

Two examples of resources which are independent on some machines are floating point arithmetic co-processors and vector units.

2.4.2. Task Completion Time

Min_latest_finish finds what the latest completion time would be, including communication, for task t if it were scheduled on processor p. It assumes a set of children $C_t = \{c : (t,c) \in M\}$, a schedule σ, and a processor assignment map π. Precedence constraints on the program require that t finish before any of its children begin execution. Additionally, each child must receive its data from t before it begins. Thus t's completion time must occur before any child's start time ($\sigma(c)$), including enough time for t to send its message to c. The function is defined as:

```
min_latest_finish(task,processor) =
    finish $\leftarrow \min_{child \in C_t}$ latest_finish(task,processor,child)

    return finish
```

Latest_finish finds the latest completion time allowed for a task t on processor p, considering only the fact that child c has been assigned to processor $\pi(c)$. It finds the start time $\sigma(c)$ of the child and computes the latest time that t may finish so t's message to c will be available before c begins execution. This depends upon the overhead and delay involved in sending the message from t to c, and the schedules for each of the resources used in the transfer.

```
latest_finish(task,processor,child) =
        message ← size(task,child)
          finish ← mesg_start(message,processor,$\pi$(child),$\sigma$(child))

        return finish
```

Mesg_start is another function which, like *start_time*, is architecture dependent. *Start_time* models the computational resources and *mesg_start* models the communication subsystem. It figures the time at which the message *mesg* must be sent from processor *source* to processor *destination*, so the message is received on *destination* before time *finish*. We present an example function for *mesg_start*. This example function models a completely connected network where some CPU time is required on each processor to read and write the message, and messages are *not* pipelined — that is, only one message at a time may be sent between the two processors. Figure 1 illustrates a message transfer on this architecture.

```
mesg_start(mesg,source,destination,finish) =
        r ← read_ovhd(mesg,destination)
       $s_r$ ← start_time(r,destination,finish)
        x ← mesg_xfer(mesg,source,destination)
       $s_x$ ← start_time(x,channel(source,destination),$s_r$)
       w ← write_ovhd(mesg,source)
       $s_w$ ← start_time(w,source,$s_x$)

        return $s_w$
```

Because even the message passing operations are scheduled, messages will arrive at or before the time they are expected. The scheduling algorithm does not overcommit or otherwise ignore the communication resources. Overcommitting or ignoring communication resources appears to be a problem shared by most, if not all, other static scheduling algorithms. DSH, which was intended to consider the problem of communication in scheduling [7,8], does not consider the effect of overscheduling the communication system. As a result, DSH works as if communication channels between processing elements may simultaneously pass more than one message (for example see [8] figure 4). If such messages were serialized, the performance of the schedule would be substantially reduced. In effect, the architectural model for DSH is unrealistic in at least one important area, since it considers the latency induced by communication, but not the capacity of the communication system. Even so, DSH assumes that the latency is independent of the message source and destination, which is rarely the case in real systems [3].

2.4.3. Communication Costs

Communication costs can be modeled in many different ways, depending on the architecture involved. Perhaps the most commonly sited characteristics of communication are latency and contention. Latency is easily modeled, usually being some linear function of the message size and distance the message must travel. Contention is not so easily modeled, often because the interactions which cause contention are very complex and not well understood. We choose here to avoid contention rather than model it.

Contention may occur wherever communication resources may be shared. This includes the bus in bus-based systems like the Sequent Balance®, multistage switches like the butterfly switch in the BBN Butterfly™, or communication channels in multi-hop networks like the Intel iPSC™. Contention is difficult to avoid in the first two types of systems. It can be avoided for multi-hop networks by maintaining schedules for each of the communication channels. Where resources may service a limited number of users at any given moment, they should be explicitly scheduled.

Consider as an example a 16-node binary n-cube, or hypercube. Several machines of this type are manufactured, by Intel, N-Cube, and Floating Point Systems [3]. A 16-node hypercube is illustrated in Figure 2. In each of these machines communication may occur in parallel over different edges, but only one transmission may occur at a time on any given edge. Therefore each edge must be treated as a resource, the same as each node. In addition, communication may require more than one edge, requiring that a message be scheduled on more than one channel. Dynamic programming [5] could be used to search for the fastest route between source and destination nodes since the particulars of local traffic patterns are known at the time the message is scheduled.

Explicitly scheduling the communication allows optimal message routing without fear of deadlock, or incurring the expense of deadlock prevention. It also enjoys greater throughput and flexibility than static routing methods. In addition, higher performance network topologies for which deadlock-free static routing methods are not yet known, such as the star-graph [1,2], could be used.

Our previous example (Figure 1) also illustrates additional elements which may come into play. When a message is sent on an Intel or N-Cube machine, the node CPU first copies the message into a second buffer, then hands the message over to a co-processor which handles the transmission. The important part to note here is that the CPU must first do some work before transmission may take place. Depending on the magnitude of the work compared to the program tasks, that work must be scheduled on the CPU. A similar effect occurs when the message is copied on the receiving node from system buffers into program memory.

2.5. Individual Task Scheduling

Task scheduling inserts into the appropriate sub-schedules all the work which must be done to complete the task on the selected processor. Task scheduling is completely a matter of bookkeeping, especially since all the required information was generated while selecting the processor. It is only a matter of updating data structures with information previously obtained.

3. Algorithm Complexity

In this section we consider the complexity of our algorithm. We refer once again to the algorithm presented in an earlier section.

```
thread the application graph
while ( some task remains unscheduled )
        analyze the application graph
        select a task for scheduling ($t$)
        find the "best" processor ($p$) for $t$
        schedule $t$ on $p$
end while
```

The complexity of this algorithm is relatively simple to derive. A topological sort of a DAG is at best $O(m+n)$, where $n = |T|$ is the total number of tasks, or nodes in an application graph, and $m = |M|$ is the number of messages, or edges which connect them. Since m may be proportional to n^2 even in a DAG, the complexity at worst is $O(n^2)$. Threading simply connects nodes in a topological order, so its worst case complexity is also $O(n^2)$. However, application functions and operators typically have a small, fixed arity (number of arguments), so for the average case $m \propto n$ and the complexity is $O(n)$ rather than $O(n^2)$.

The while loop selects exactly one task for scheduling with each iteration, so each component within the loop is multiplied by n. (Testing for unscheduled tasks is an $O(1)$ operation.) The graph analysis phase is a simple two-pass PERT analysis [5], also with complexity $O(m+n)$. Task selection requires at most a single scan through the graph so it is also $O(n)$. Processor selection depends on the number of processors in the system, since it tries the selected task on each processor — but it also depends on the number of children a task has. The communication time between the task and its children is evaluated each time a task is considered on a different processor, in order to determine the latest time the task may finish. The schedule is also examined to find the best slot after the finish time is determined, but the latest task finish time provides an upper bound on the depth d of the schedule search. Unfortunately, the only information we have about d is $1 \leq d \leq n$. Thus the complexity for this step is $O(dpc)$, where $p = |P|$, the number of processors in the system, and c is bounded above by $\frac{|M|}{|T|}$.

Task scheduling is much simpler, requiring only constant time to insert a task into the schedule, or if minimal bookkeeping is done, no greater effort than to select a processor.

Collecting terms, the complexity is $O(n+m+n(n+m+n+dpc+1))$, or simply $O(n^2+dpm)$. For constant p and $m \propto n$, the algorithm complexity reduces to $O(n^2)$. At worst $m \propto n^2$ and $d \propto n$, and the algorithm has complexity $O(n^3)$.

4. Conclusions

We have presented an algorithm which computes near-optimal schedules for the extended Precedence Constrained Scheduling Problem. The algorithm computes the schedules in low-order polynomial time, and may be adapted for virtually any MIMD architecture. It is able to account for communication capacity and multiple resources. Also, it is adaptable to irregular networks and non-homogeneous processing elements. The algorithm has been implemented in "C", and preliminary tests have been run. With small graphs (10 to 20 nodes) for which an optimal schedule is known, the algorithm was always able to select schedules which differed little from optimal.

References

1. Akers, S. B. and Krishnamutrhy, B. Group graphs as interconnection networks, *14th International Conference on Fault Tolerant Computing,* Kissimmee, Florida, 1984.

2. Akers, S. B. and Krishnamurthy, B. A group theoretic model for symmetric interconnection networks, *Proceedings of the 1986 International Conference on Parallel Processing,* 1986.

3. R. G. Babb II, ed., *Programming Parallel Processors,* Addison-Wesley, 1987.

4. Garey, M. R. and Johnson, D. S. *Computers and Intractability — A Guide to the Theory of NP-Completeness,* Freeman, 1979.

5. Hillier, F. S. and Lieberman, G. J. *Operations Research,* Holden-Day, San Francicso, 1974.

6. Horowitz, E. and Sahni, S. *Fundamentals of Data Structures*, Computer Science Press, 1976.

7. Kruatrachue, B. and Lewis, T. Duplication scheduling heuristic (DSH), a new precedence task scheduler for parallel systems, Technical Report 87-60-3, Oregon State University, Corvallis, 1987.

8. Kruatrachue, B. and Lewis, T. Grain-size determination for parallel processing, *IEEE Software* **5**, 1 (January 1988), pp. 23-33.

9. Ullman, J. D. NP-Complete scheduling problems, *Journal of Computer and System Sciences* **10**, 3 (June 1975), pp. 384-393.

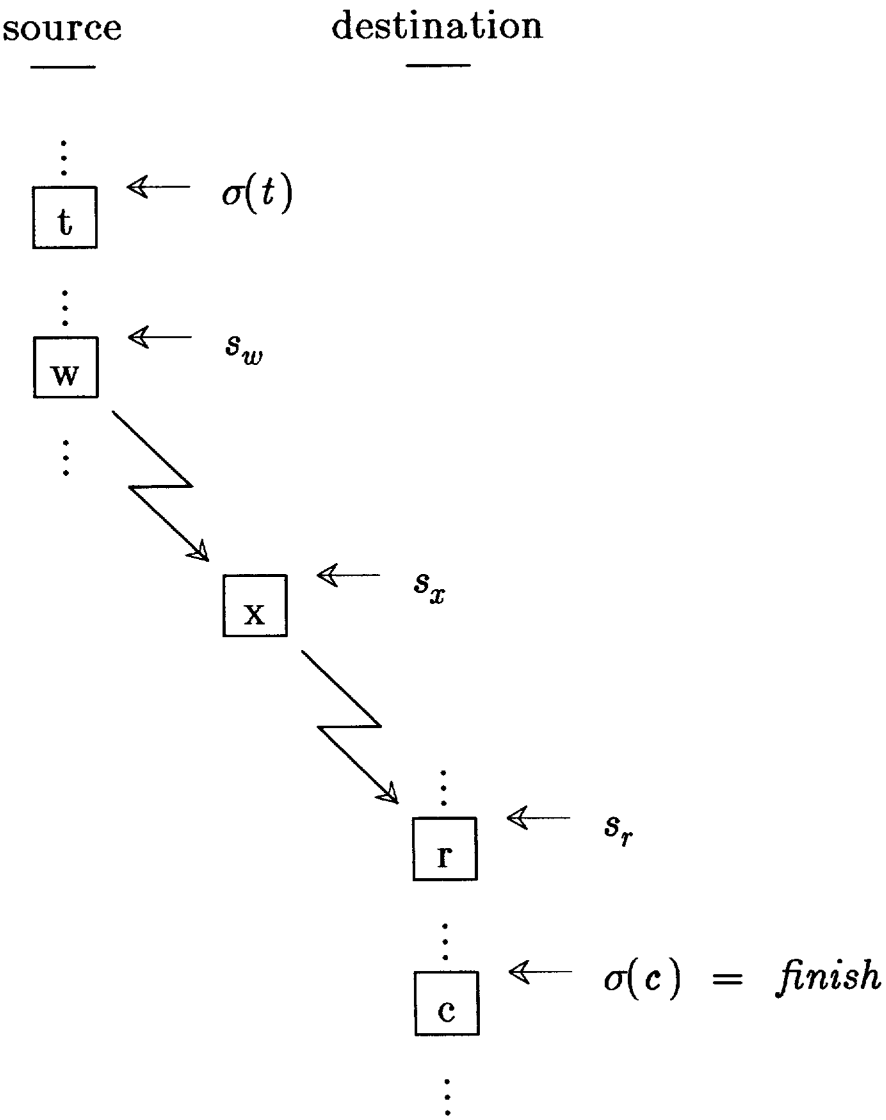

Figure 1. Communication Between Neighboring Nodes

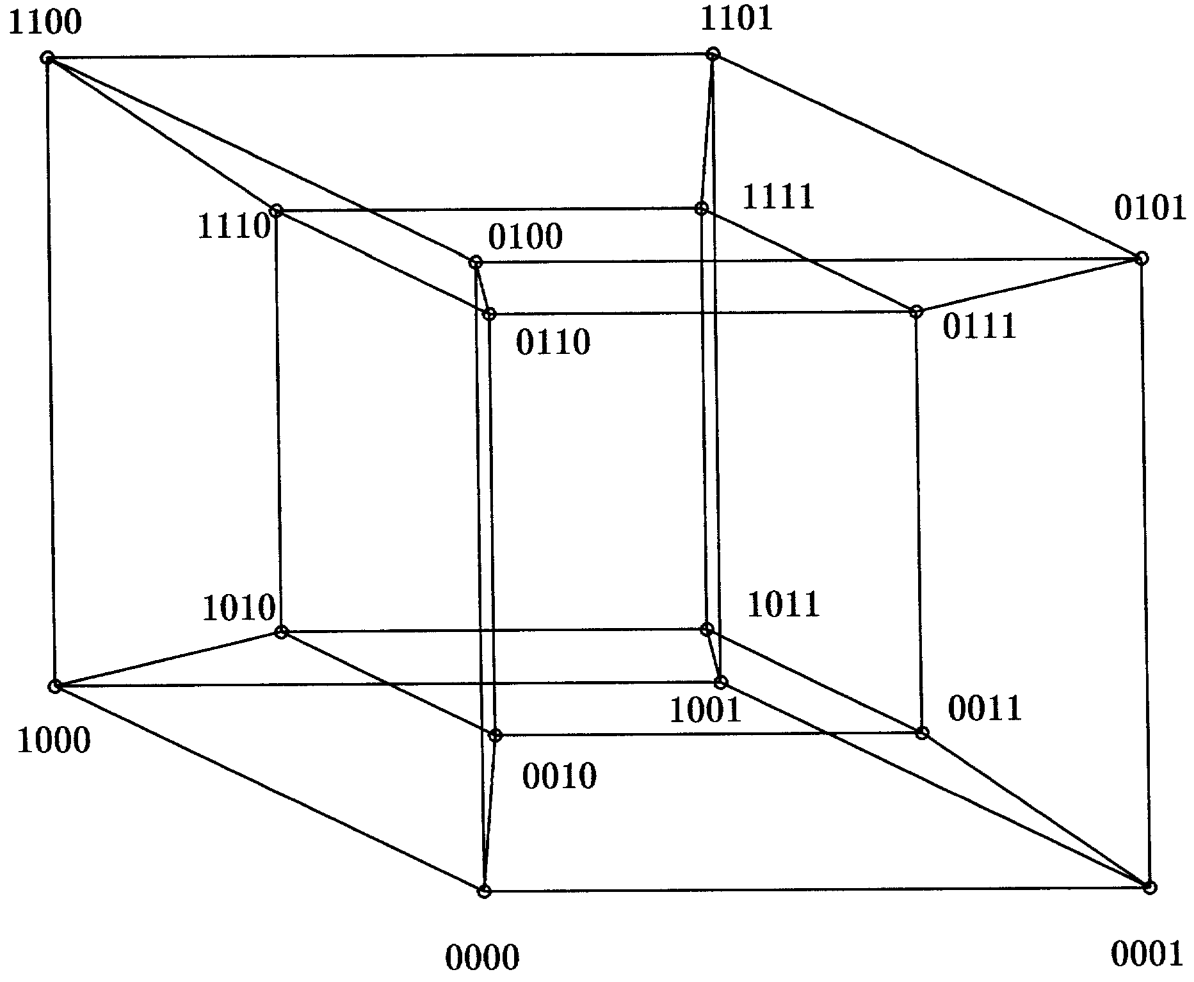

Figure 2. 16-Node Hypercube

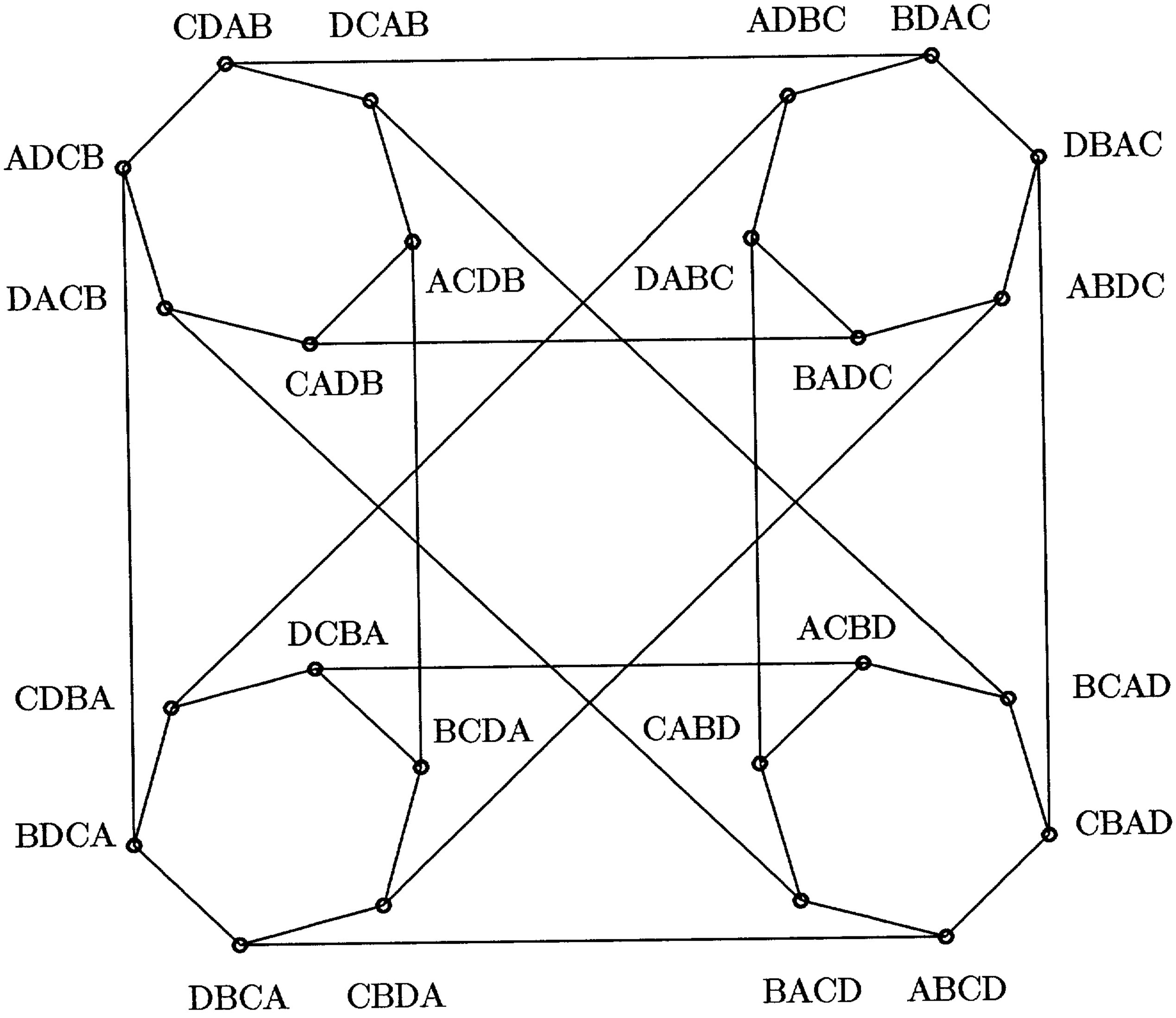

Figure 3. 24-Node Star-Graph