A GENERAL APPROACH TO MULTIPROCESSOR SCHEDULING

Sung Jo Kim

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-04 February 1988

A GENERAL APPROACH TO MULTIPROCESSOR SCHEDULING

ABSTRACT

As a variety of general-purpose multiprocessor systems have been recently designed and built, multiprocessor scheduling is becoming increasingly important. Multiprocessor scheduling is a technique to exploit the underlying hardware in a multiprocessor system so that parallelism existing in an application program can be fully utilized and interprocessor communication time can be minimized. Traditionally, most research on multiprocessor scheduling has focused on the development of specific scheduling strategies to take advantage of unique characteristics of a specific multiprocessor system or application program. In this thesis, we define and characterize scheduling techniques and related heuristic mapping algorithms which are applicable to a spectrum of multiprocessor systems and a broad class of application programs.

The fundamental idea we use is that multiprocessor scheduling can be regarded as a series of mappings from a computation graph (representing an application program) to a virtual architecture graph (representing an optimal architecture for the program) and eventually to a physical architecture graph (representing a target multiprocessor system). We propose linear clustering and linear cluster merging as effectual heuristics. After linear clustering and merging, the computation graph is transformed into a virtual architecture graph. This graph represents an optimal architecture which compromises between two conflicting goals, minimization of interprocessor communication and maximization of potential parallelism, and satisfies the other goals, throughput enhancement and workload balance, relatively well. Then we develop two efficient scheduling algorithms which map the optimal architecture graph onto a physical architecture graph which may represent either a homogeneous or a heterogeneous multiprocessor system. These algorithms rely not only on local information but also on limited global information. Finally, we present the result of performance evaluation of the mapping algorithms on an Intel iPSC with 32 processors and a Sequent Balance with 10 processors.

Table of Contents

CHAPTER 1

INTRODUCTION

For the past several years, we have witnessed a surge of new general-purpose multiprocessor systems unequaled since the first generation multiprocessors like the CMU C.mmp [MAS82] and the BBN Pluribus [KAT78] were built around the early 1970's. This is a natural approach to solve computationally intensive problems as hardware technologies approach physical limitations.

Depending on the degree of coupling among processors (i.e., the amount of time it takes to move data from one processor to another or to initiate an action on one processor from another), there is a spectrum of multiprocessor systems from loosely-coupled to tightly-coupled systems. On one extreme, a shared-memory architecture like the Sequent Balance [SEQ86] is an example of a tightly-coupled multiprocessor system. On the other extreme, a wide area network like the ARPANET [HEA70] can be regarded as a loosely-coupled multiprocessor system. Between these two extremes lie most architectures like the IBM RP3 [PFI85], the BBN Butterfly [BBN85a], the NYU Ultra [GOT83a], the Flex/32 [MAT85], the Intel iPSC [INT87], the Ncube/*ten* [NCU85], and the Connection Machine [HIL85]. Note that general-purpose multiprocessor systems we deal with are assumed to employ MIMD (multiple-instruction and multiple-data stream) type architectures. Consequently, an array processor like the Illiac IV [BOU72] will not be considered as a multiprocessor system by itself, but as a processor component of a multiprocessor system.

Even though an enormous amount of theoretical work has been done to find optimal solutions of multiprocessor scheduling problems, no practical, generally applicable polynomial-time algorithm has yet been found. The obvious approach then is to concentrate on the development of polynomial-time algorithms that provide sub-optimal solutions in many cases. This thesis reports research on the development of scheduling techniques and related scheduling algorithms based on heuristics which are applicable to a spectrum of general-purpose multiprocessor systems mentioned above.

In this chapter, we overview and outline this thesis. From here on we use the term *task* and *schedulable unit of computation*, which corresponds to a node in a computation graph, interchangeably.

1.1. An Overview of the Thesis

In this section, we address the problem we tackle and our approach to the problem, and summarize our results and contributions.

1.1.1. Problem Statement

Optimal scheduling of parallel computations to multiprocessor systems requires the optimal assignments of processors to computations and communication resources to the implementations of dependency relations between the schedulable units of computation in order to minimize total execution time for the computations. The assignment of computational work to processors must take into account both balancing the workload assigned to processors and minimizing the communication cost among processors.

A parallel computation can be represented by a direct acyclic graph $G_C = (N_C, E_C)$, where $N_C = \{n_1, n_2, \cdots, n_l\}$ is a set of schedulable units of computation to be executed, and E_C specifies scheduling constraints defined on N_C. A multiprocessor system can be represented by an undirected graph $G_P = (N_P, E_P)$, where $N_P = \{p_1, p_2, \cdots, p_m\}$ is a set of processors, and E_P specifies interconnection network among the processors. The basic problem that we are attempting to solve is to find a mapping of G_C onto G_P which minimizes the schedule length (or makespan) defined as:

$$\max_{1 \le k \le s} \sum_{i,j \in \phi_k} (comp_i + comm_{ij}),$$

where $\phi = \{\phi_1, \phi_2, \ldots, \phi_s\}$ represents a set of paths from the root node to the leaf node in G_C, node n_j (assigned to processor $p_y \in N_P$ $(1 \leq y \leq m)$) is a direct descendant of node n_i (assigned to processor $p_x \in N_P$ $(1 \leq x \leq m)$) in G_C, $comp_i$ is computation time of n_i, and $comm_{ij}$ is communication time from n_i to n_j ($comm_{ij} = 0$ if $p_x = p_y$ or n_i has no direct descendants).

An optimal schedule is one which meets the criteria of the minimum schedule length for a single parallel computation structure or the maximum total throughput for a set of simultaneously executing parallel computation structures. It must integrate scheduling of computations and dependency relations to resources. An approach which integrates consideration of all the interacting factors is one which maps a computation graph defining the computation structure (including the resource requirements for execution of each element of the computation structure) onto an architecture graph which defines the capability and capacity of the resource set of the execution environment.

This integrated approach is a substantial advancement in scope over the current state-of-the-art. Some research in multiprocessor scheduling dealt with only one of the interacting factors, usually the computation time [GOT83a, BBN85a], the workload balance [STA84, VAN84, WAN85, EAG86] or the implementation cost of dependency relations between the schedulable units of computation [HAE80, BIA85]. Other research which has considered integrated scheduling was primarily concerned with the development of the concepts rather than applicable methodologies or else were applicable only to specific configurations of resources or specific problems [FOR78, LIU78, SOL79, WIT80, BRY81, CHO82, DEG81, GOT83a, BER84, SHE85, BBN85a, LEE87].

The integrated approach defined and characterized herein requires three elements:

- A graph representation of a computation structure which integrates the specification of dependency relations and resource requirements of each schedulable unit of computation;
- A graph representation of an execution environment which includes the specification of the capabilities and capacities of the resource set;

- Algorithms for mapping from a computation graph to an architecture graph.

The research defined and described in this thesis is a first attempt at such an integrated and generally applicable approach to scheduling of parallel computation structures on multiprocessor systems. The difficulty of attainment of a fully integrated, broadly applicable and truly optimal scheduling methodology is not underestimated. In this research, we attempt to extend the state-of-the-art to include simultaneous consideration of the important elements of the multiprocessor scheduling problem. The critical element of this approach is the establishment of a general framework in which to evaluate integrated scheduling methodologies rather than the development of individual mapping algorithms which are suitable for individual applications and a specific multiprocessor system. We develop a generalized multiprocessor scheduling methodology which is applicable to a spectrum of multiprocessor systems and a broad class of application programs.

The fundamental idea on which this research is based is that multiprocessor scheduling can be regarded as a mapping between graphs. To be specific, one will take a representation of a computation as a computation graph and a representation of a multiprocessor system as an architecture graph and then formulate the problem of scheduling as the mapping of one graph onto another graph subject to specific scheduling constraints [BRO86]. In fact, we consider a series of mappings between a computation graph and an architecture graph including a direct mapping of the former onto the latter as illustrated in Fig. 1-1.

As can be seen in Fig. 1-1, a computation specification of a parallel computation structure and a resource specification of a target multiprocessor system are transformed into a computation graph and a physical architecture graph, respectively. We have designed two interface specification languages (*ISL's*) [KIM86a]: one for computation graph generation and the other for physical architecture graph generation. The most important motivation for designing an interface specification language for computation graphs is to create a language which defines the interface specification of the computation structures and dependency relations, including resource requirements of the structures and the relations. The nodes and directed edges in a computation graph (restructured to acyclic direct graphs, if necessary) represent respectively the bindings of operations to data and the dependency relations between the computations executed at the nodes [BRO85]. Since in general the bindings can occur dynamically at runtime, a computation graph can be dynamic as well. In this research, however, we assume that all the bindings are fixed prior to runtime in the sense that there are no runtime bindings of operations to data objects which modify the structure of the computation graph. Consequently, a computation graph is *static* .

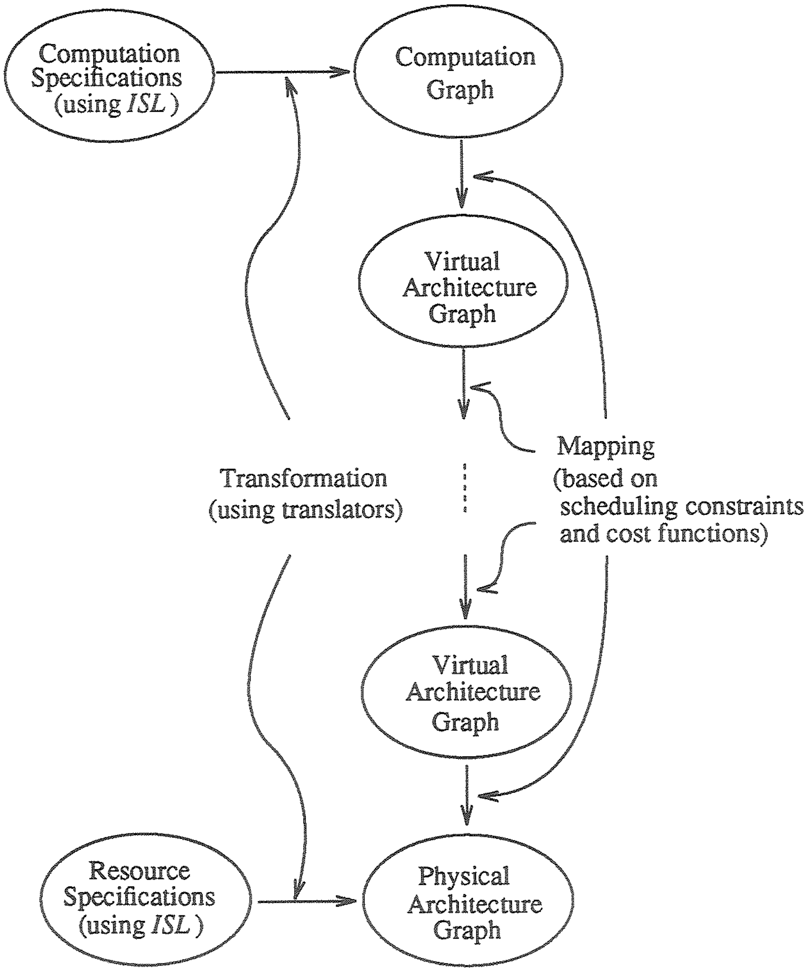

Computation Specifications (using ISL)
Computation Graph
Virtual Architecture Graph
Mapping (based on scheduling constraints and cost functions)
Transformation (using translators)
Virtual Architecture Graph
Resource Specifications (using ISL)
Physical Architecture Graph

Figure 1-1 General Overview of Our Approach

There are two types of architecture graphs: *virtual architecture graphs* and *physical architecture graphs*. Both architecture graphs are undirected. A physical architecture graph represents a real multiprocessor system, and a virtual architecture graph represents an abstract multiprocessor system which is regarded as an optimal architecture for a given computation. Except in the case of direct mapping of a computation graph onto a physical architecture graph, scheduling algorithms make use of virtual architecture graphs for the initial mapping as well as the intermediate mappings. The intermediate mappings, including the initial mapping, are called *logical mappings*. At the final stage of the mapping, one of the virtual architecture graphs is mapped onto the physical architecture graph. This mapping is called *physical mapping*.

A restricted case of the general graph mapping problem is the graph isomorphism problem. It has been shown [BOK81] that the general mapping problem is computationally equivalent to the graph isomorphism problem. Unfortunately, there is no known polynomial time algorithm for solving even this restricted problem [GAR79]. Furthermore, precedence constrained scheduling problem which is a subproblem of the general scheduling problem has been proved to be *NP−complete* [ULL75]. As a result, it is clear that in order to stay in the realm of practicality, we should either rely on efficient heuristics as in [KER70, BAR81] or develop algorithms which give optimal solutions only for restricted cases [KER71, RAM72, LUK75, LLO80, CHI84, LO85]. We concentrate on the development of multiprocessor scheduling techniques and related heuristic algorithms based on graph mapping that supply efficient approximate and computationally feasible solutions. Most importantly, they are applicable to a spectrum of multiprocessor systems from loosely-coupled to tightly-coupled systems.

1.1.2. Approach

One of the contributions of this thesis is to propose new multiprocessor scheduling techniques based on *linear clustering*. A linear cluster is a connected subgraph of a computation graph which is in the form of a linear list of schedulable units of computation. Linear clustering is an effectual heuristic to compromise between two conflicting goals of multiprocessor scheduling, minimization of interprocessor communication and maximization of potential parallelism, and to satisfy the other goals, throughput enhancement and workload balance, relatively well. The underlying idea of linear clustering is that the schedulable units of computation that are sequentially dependent on each other are to be assigned to one processor, while those that are mutually independent are to be allocated to separate processors. The critical restriction of linear clustering is that it expects a computation graph to be acyclic. In order to relieve this restriction, we identify cases in which cyclic computation graphs can be transformed into acyclic graphs in a straightforward manner.

A computation graph is transformed into a virtual architecture graph by linear clustering. The virtual architecture graph in fact represents an optimal multiprocessor system for the computation graph. The optimal multiprocessor system may provide one processor to every linear cluster so that mutually independent tasks belonging to different linear clusters can be executed in parallel as long as possible. Furthermore, direct communication links are always available for any adjacent linear clusters in the optimal multiprocessor system. The virtual architecture graph may be transformed into another virtual architecture graph by *merging* two or more linear clusters into one cluster. Two linear clusters K_1 and K_2 are combined into one if K_2 may start only after K_1 finishes or may be executed only while K_1 is idle. It contributes to further balancing the workload of processors, and further reducing the amount of resources to be utilized and interprocessor communication overhead.

Virtual architecture graphs can be constructed independently of the target multiprocessor system. As a result, the characteristics of the target system matter only during physical mappings. After constructing a virtual architecture graph which represents the optimal multiprocessor system for a given computation graph, we just need an optimal mapping of the virtual architecture graph onto a physical architecture graph which represents the target multiprocessor system. We develop *homogeneous* and *heterogeneous mapping* algorithms for homogeneous and heterogeneous multiprocessor systems, respectively.

These algorithms rely on not only local information but also on limited global information. The key issue is then how to reduce the mapping complexity while sacrificing the optimality as little as possible. A *dominant request tree* is a maximal spanning tree of a virtual architecture graph. It provides limited global information on the virtual architecture graph such as the mapping order of the nodes and the edges whose adjacency should be maintained. Both mapping algorithms utilize dominant request trees, but take quite different approaches to mapping the trees onto physical architecture graphs. Most importantly, in the case of homogeneous mappings, the trees are directly mapped onto physical architecture graphs. On the other hand, in the case of heterogeneous mappings, they are mapped onto dominant service trees. A *dominant service tree* is a maximal spanning tree of a physical architecture graph. For heterogeneous mappings, one of the important issues is how to identify and utilize resources with high performance as much as possible. A dominant service tree provides such information.

1.1.3. Results: Theoretical and Practical

The problem of scheduling of general static computation graphs to general multiprocessor architectures is expressed as a sequence of transformations on computation graphs and architecture graphs, and a series of mappings among the graphs. Explicit consideration is given to both homogeneous and heterogeneous multiprocessor architectures. Algorithms for these transformations and mappings are given and are characterized by their computational complexity. Bounds on the number of processors needed to attain maximum parallelism at a given level of task granularity are derived. It is shown by analysis and by experimental demonstration that the scheduling methodology developed, while known in general to produce sub-optimal schedules, does produce effective schedules with reasonable effort.

New schemes for transformation of the computation graphs of a large class of loop structured programs into a form to which the general scheduling methodology applies are defined and described. In addition, they are integrated into the transformation process.

Experiments are conducted on both a distributed memory architecture, an Intel iPSC with 32 processors and a shared memory architecture, a Sequent Balance with 10 processors. The computation graphs used in the experiments include a regular graph with a cycle, a regular graph without a cycle and an irregular graph. The Intel Hypercube is turned into a heterogeneous multiprocessor system by deliberate manipulation of the computation times and communication times of the nodes and edges of the irregular computation graph, respectively. The results of the experiments reveal substantial performance enhancement after linear clustering and linear cluster merging of the computation graphs through application of the general scheduling methodology and sub-optimal results in some cases where optimal execution times can be derived.

1.2. Organization of the Thesis

We begin in Chapter 2 by taxonomizing general-purpose multiprocessor systems. Next, we discuss various multiprocessor scheduling strategies: *ad-hoc, manual, automatic* and *restricted* schedulings. Traditionally, depending on the degree of coupling among processors, there have been two main multiprocessor scheduling: one for loosely-coupled systems and the other for tightly-coupled systems. After reviewing the previous strategies and cost functions, we discuss their weaknesses.

The graph models described in Chapter 3 are concerned with the properties of graphs we deal with for multiprocessor scheduling. This chapter introduces two types of graphs: computation graphs and architecture graphs. In fact, we make use of the same graph model to represent two different types of architecture graphs: virtual and physical architecture graphs. This chapter also introduces the conceptual model for a generalized multiprocessor scheduling. We use this model to realize the different scheduling strategies mentioned in Chapter 2.

In Chapter 4, we propose a new multiprocessor scheduling technique based on *linear clustering* and *linear cluster merging*. The ideas of linear clustering and merging are explained and justified. Then we identify the cases that cyclic computation graphs may be transformed into acyclic graphs in a straightforward manner.

In this chapter, we also establish some interesting properties of linear clustering and merging.

The main subject of Chapter 5 is to describe efficient heuristic, polynomial-time mapping algorithms for the different types of multiprocessor systems: homogeneous and heterogeneous systems. We explain each mapping algorithm and discuss the time complexity of each. We also propose and justify transformations of virtual architecture graphs and physical architecture graphs into dominant request trees and dominant service trees, respectively.

In Chapter 6, the performance of the proposed mapping algorithms is measured and analyzed. After summarizing our experimental environments, we enumerate the performance metrics being considered in the experiments. Then, we discuss the implementations and the results of performance evaluation in order to show how the proposed mapping algorithms behave on the different types of multiprocessor systems and computation graphs.

Finally, concluding remarks are contained in Chapter 7. In this chapter, we summarize our research and suggest the future research directions.

CHAPTER 2

MULTIPROCESSOR SCHEDULING PROBLEM

General-purpose multiprocessor systems can be classified into four categories based on the level of couplings among processors and memory modules as follows:

- very tightly-coupled systems;
- tightly-coupled systems;
- loosely-coupled systems;
- very loosely-coupled systems.

The first category includes systems like the Sequent Balance 21000 [SEQ86]. It has one global memory shared by all the identical processors, all of which are connected by one global bus. The second category includes the IBM RP3 [PFI85], the NYU Ultracomputer [EDL85a, EDL85b] and the BBN Butterfly [BBN85a]. Each processor in these systems has a local memory which can be shared efficiently by the other processors through a switching network. In particular, the RP3 also supports software-controllable shared-memory. As a result, a single system like the RP3 may realize a spectrum of the coupling among its processors. This category also includes the Flex/32 [MAT85] in which each processor has its own local memory, and all processors can share a global memory through a global bus. The Intel iPSC [INT87] belongs to the third category. The processors, each of which has local memory, are connected by asynchronous communication links. Since there is no shared memory in the iPSC, the processors communicate with one another by message passing. Finally, a wide area network like the ARPANET [HEA70] or a local area network based on the Ethernet [MET76] can be regarded as a very loosely-coupled system.

Scheduling strategies for multiprocessor systems can be discussed in terms of the following factors:

- The time (hardware design time, compile time, load time and runtime) at which mappings are fixed and elements of the computations are bound to resources;
- The degree of automation of mappings and bindings;
- The degree of coupling among resource sets;
- The cost metrics or cost functions used as decision variables.

In this section, we first discuss multiprocessor scheduling strategies in terms of the four factors. After reviewing previous approaches to multiprocessor scheduling and cost functions, we discuss their weaknesses. Finally, we discuss the complexity of multiprocessor scheduling problems being investigated in this thesis.

2.1. Multiprocessor Scheduling Strategies

For a given multiprocessor system having a certain degree of coupling, depending on the degree of automation of mappings and bindings provided for the user, there is a spectrum of multiprocessor scheduling strategies. These range from leaving the user the entire burden of managing processors in the system to simply requiring the user to define tasks and specify dependency relations among tasks. The degree of automation in fact depends on the level of visibility of multiprocessor systems exposed to the user. There are four classes of multiprocessor scheduling strategies in the spectrum:

- *Ad−hoc* multiprocessor scheduling: This is suitable for multiprocessor systems like systolic arrays [KUN82] whose configurations and operational characteristics should be unfolded as low as the hardware gate level to the user. As a matter of fact, there is virtually no resource scheduling by the operating system, because the user is fully responsible for allocation of processors in the system and synchronization between them; it is even required to specify the complete timing of all instructions for all the processors in the system. The binding of processors to tasks is fixed at hardware design time or possibly at compile time [KUN84].
- *Manual* multiprocessor scheduling: This strategy aims at handling multiprocessor systems whose physical configuration of resources and status are partially

visible to the user (e.g., the number of active processors, the connectivity of processors, the communication costs, the characteristics of resources, etc.). The user can make use of such visibility for the actual resource scheduling. For example, he may specify statically process working sets or may designate a specific processor to execute a particular task as in the Medusa operating system [OUS80, OUS82] for Cm* [JON80]. The processors are bound to tasks at compile time or possibly at load time.

- *Automatic* multiprocessor scheduling: The actual allocation of resources is hidden from the user and is done by the operating system of a multiprocessor system. Nothing related to physical resources is visible to the user; however, he is responsible for the specification of dependency relations among tasks as in the Uniform System [BBN85b] for the Butterfly system [BBN85a] or as in Dynix for the Sequent Balance [SEQ86]. In the Uniform System, the binding of processors to the tasks can be done either before or at runtime. Because of the flexible binding time, the tasks may be migrated dynamically to other processors to balance the workload of each processor or to bypass failed processors. The Dynix system allows both static and dynamic schedulings. The latter is useful for workload balancing among processors.
- *Restricted* multiprocessor scheduling: This is a variation of the preceding two strategies. It may restrict utilization of full resources available in a multiprocessor system or the way they can be used. For example, the Cedar system [GAJ83] and the IBM RP3 [PFI85] may be partitioned into subsystems so that each subsystem can be assigned to a different application program to improve the total throughput.

Depending on scheduling objectives, we can define a variety of cost functions which may be expressed in terms of various cost metrics. In fact, those functions are utilized as decision variables for reducing the solution space during mapping. As a result, multiprocessor scheduling strategies (more specifically, mappings) are significantly influenced by the cost functions. The most important and frequently used cost metrics are the computation cost and the interprocessor communication cost. We will discuss them in detail in the following section.

2.2. Review of Previous Multiprocessor Scheduling Strategies

Basically, there have been two main streams of research on multiprocessor scheduling strategies: one for loosely-coupled systems and the other for tightly-coupled systems. In this section, we discuss scheduling strategies for each system. We also discuss cost functions and metrics considered in the previous research. Note that the visibility of a multiprocessor system exposed to the user (which determines the degree of automation of scheduling) has not been considered explicitly in any previous study of multiprocessor scheduling strategies.

2.2.1. Scheduling Strategies for Loosely-Coupled Systems

Several approaches to scheduling strategies in loosely-coupled systems have been suggested in the past. They can be roughly classified into three categories, namely, graph theoretic [STO77, STO78], mathematical programming [LEE77, CHU80, MA82], and heuristic methods [GYL76, BOK81, EFE82, MA82, KAS84, PAT84, STA84, VAN84, CAM85, SHE85, LEE87, SAD87].

The graph theoretic approach uses a computation graph to represent an application program and applies the minimal-cut algorithm [STO77] to the graph. The goal of this scheduling is to minimize the total execution cost, defined as the sum of the computation cost and the interprocessor communication cost. In this approach, in order to minimize the total cost, a computation graph is modified so that each cutset in the modified graph corresponds in a one-to-one fashion to a task allocation, and the weight of the cutset is the total execution cost for that allocation. With this modification, a maximum flow problem [FOR64] is solved on the modified graph. The minimum weight cutset obtained from this determines the task scheduling which is optimal in terms of the total cost.

Even though the min-cut max-flow algorithm adopted by this approach guarantees an optimal solution, it is only practical for finding a minimum cost allocation between two or three processors. Moreover, it does not deal with such issues as workload balancing, resource constraints (e.g., memory size), and dependency relations among tasks. Rao et al. [RAO79], however, have tried to find a minimum feasible allocation of tasks to two processors, one of which has limited memory, as an attempt to include limited scheduling constraints.

In the mathematical programming approach, the scheduling problem is formulated as an optimization problem and solved with linear programming techniques. This approach has been applied with some success to the file allocation problem [CHU69] in a loosely-coupled multiprocessor system. For multiprocessor scheduling purposes, it can be used to minimize the total execution cost subject to some given scheduling constraints. The objective function is the total cost which is also a sum of the computation cost and the interprocessor communication cost. The constraints might be a memory size restriction on each processor or response time, for example.

As in the previous approach, this approach gives us an optimal solution. It is more flexible than the previous one, however, since it allows various scheduling constraints to be introduced into the model. This is difficult or impossible with the graph theoretic approach. Since this approach suffers the problem of increased complexity as the dimension of the problem becomes larger, it is not suitable for time-critical applications. It was reported in [CHU80] that it typically took a CDC 6000 series computer two to three minutes to schedule 25 modules onto 15 processors. As a result, it is not practical to specify more than a couple of constraints. Furthermore, there is no provision for specifying dependency relations.

The heuristic approach, in general, provides an approximate solution for multiprocessor scheduling. Even though the two previous approaches provide us with optimal solutions, it is unlikely to find efficient exact algorithms based on them for the general mapping problem within the reasonable time constraints. The heuristic approach is very useful when an optimal solution is not required, not obtainable, or can not be obtained within time limits. Note that heuristic approaches may be based on a combination of the previously described approaches.

As a result, for more flexible multiprocessor scheduling, it is imperative to rely on heuristic algorithms which are computationally tractable. In general, heuristics will not guarantee optimal solutions, but may allow the specification of much more scheduling constraints than the previous two approaches, since they are less sensitive to the magnitude of the scheduling problem.

2.2.2. Scheduling Strategies for Tightly-Coupled Systems

There are not as many distinct scheduling approaches for tightly-coupled systems as for loosely-coupled systems. Scheduling strategies for tightly-coupled systems may be categorized based on the degree of visibility of underlying multiprocessor architectures; the more visible the architectures are, the more responsible the user is for resource scheduling.

Ousterhout [OUS82] proposes a multiprocessor scheduling strategy for Cm* which is appropriate for a tightly-coupled environment[†] in which some characteristics of the architecture are known to the user. That is, the configuration of the multiprocessor system is partially exposed to the user. As a result, processor scheduling may rely on the user's ingenuity. For example, the user can designate a processor to which he prefers to allocate a task. He also should know the number of active processors in the system (more preferably, in the same cluster) so that all of the tasks contained in a *task force*[††] are guaranteed to be coscheduled (i.e., executed concurrently on different processors). Coscheduling is based on the observation that the duration of a busy-waiting time is usually less than two context switching times. It may prevent the possibility of process thrashing and deadlock.

Recently, several multiprocessor scheduling strategies have been proposed under the assumption that the configurations of multiprocessor systems are almost concealed from the user. Those are the Uniform System approach [BBN85b] in the Butterfly system [BBN85a], the self-scheduling approach [LUS85, TAN85] in the Cedar system [GAJ83], the self-service approach [EDL85b] in the Ultracomputer [GOT83a], and the Dynix approach in the Balance [SEQ86]. The Uniform System treats each processor as able to execute any task. The user is required to supply task generators which generate the next tasks that will be allocated to any available

[†] Even though Cm* has no central, shared memory in the sense of C.mmp [FUL73], local memory of each processor can be shared by other processors with some degraded performance. See [HAY82] for the cost ratio for different types of memory references.
[††] A task force can be defined as a set of heavily interacting tasks.

processor. Self-scheduling is a non-preemptive, distributed scheduling strategy that allows processors to schedule themselves without intervention from the operating system. Self-service is a centralized strategy, in which a single central queue of ready tasks is shared by all processors. Depending on scheduling purposes, Dynix provides three different strategies: prescheduling, static scheduling and dynamic scheduling. In order to run a different task on each processor, we should rely on the first one. On the other hand, the others are useful if we have identical tasks to execute. If a program can be partitioned into tasks in such a way that each has the same computation time, static scheduling is better than dynamic scheduling. No matter which type is chosen, it is not necessary to perceive the underlying architecture. It will be informative (but not imperative) to know the maximum number of available processors. Since scheduling strategies in these systems are basically identical and have similar problems, we will discuss only the Uniform System in detail.

The Butterfly parallel processor is a tightly-coupled, shared-memory, homogeneous system. The Uniform System implements a methodology for task scheduling in the Butterfly system. It achieves parallel processing by utilizing any active processor as soon as it becomes available. A similar approach has been adopted by the HYDRA for C.mmp [MAS82], by Dynix's dynamic scheduling scheme for the Balance [SEQ86] and by the operating system for the Pluribus [KAT78]. While the system automatically allocates tasks to processors, the user is responsible for supplying a task generator procedure to generate next tasks to run. Since the details of machine configuration are hidden from the user, he is not burdened with dynamic machine reconfiguration. Consequently, its configuration can be changed dynamically according to the availability of its processors without interfering with execution of the user's programs.

There are several drawbacks to the approach taken by the Uniform System. In fact, the other approaches also have similar problems. First of all, task generators contain any number of *internal* critical regions through which processors must proceed one by one. For example, the regions usually include the following operation:

$$\text{index} = \text{Atomic\_add}(p_1, 1).$$

This operation[†] atomically fetches and adds 1 to the location pointed to by p_1 to generate the next index value. Since processors must proceed through the critical region in a certain serial order, the region will limit the maximum number of processors that can be utilized simultaneously, no matter how small the region is [BBN85b]. Furthermore, it may force the user not to choose a smaller size of task, because the size may increase the overhead of the critical region in inverse proportion to the task size. In order to reduce the overhead, the notion of chunking [EDL85b] may be useful.

Secondly, the Uniform System approach may increase computation overhead (e.g., the amount of interprocessor communication) unnecessarily. For a given computation, because of data dependencies among tasks, there is always a minimum time span to complete the computation even if an unlimited number of processors is available. Therefore, there is an optimal number of processors required to complete the computation in the minimum time. The Uniform System may utilize more processors than optimal whenever there are available processors and one or more tasks waiting for the processors. This use of additional processors may not increase the performance of the system at all. Instead, it may increase overhead such as amount of interprocessor communication, remote memory access time and task creation time, because of unnecessary (or excessive) creation/distribution of tasks.

The next problem of the Uniform System is that for parallel processing it does not allow multiprogramming; in other words, it allows only a single user program to execute and only one task per processor in the system. This is a serious restriction, since it prevents the user from parallel processing more than one task or application program simultaneously. In spite of the potential for waste in busy-waiting, what we gain from this simple restricted scheduling is that unnecessary context switchings can be eradicated.

[†] The NYU Ultracomputer and the Cedar multiprocessor system have similar synchronization primitives. The former provides *Fetch-and-Add* [GOT83b] and the latter provides more general primitives such as {*variable, test condition, operation on key, operation on data*} [TAN85].

The final problem of the Uniform System is that each processor may be forced to be idle longer than necessary. As soon as a processor becomes available and as long as there are tasks yet to be executed, the task generator residing on the processor generates a next task and executes it. Even after a task is allocated to the processor, however, it may be forced to remain idle further until some sequencing constraints are satisfied. Since the Butterfly system allows each processor to run no more than one task for parallel processing, the power of the processor may be wasted. On the other hand, if we consider a multiprocessor system which allows multiprogramming, we need more coordinated scheduling to prevent premature occupation of a processor and waste of its processing power. For example, allocation of a task which prematurely occupies the processor may be deliberately delayed until all the necessary sequencing constraints are satisfied. During that time, the processor may execute another task which can be executed immediately, rather than being idle.

2.2.3. Cost Functions in Previous Approaches

In the previous research, many cost functions using a variety of cost metrics have been proposed especially for loosely-coupled multiprocessor systems. The most basic cost metrics are the computation cost and the interprocessor communication cost. Assuming that A is an allocation matrix (a_{ij}) of t tasks to p processors, a cost function to evaluate the total execution cost can be formulated as the sum of the two cost metrics [CHU80, MA82, SHE85]:

$$\sum_{i=1}^{t} \sum_{k=1}^{p} (\omega \cdot a_{ik} \cdot T_{comp_{ik}} + \sum_{j>i}^{t} \sum_{l>k}^{p} a_{ik} \cdot a_{jl} \cdot T_{comm_{ij}}).$$

Allocation matrix A is defined as follows:

$$a_{ik} = \begin{cases} 1, \text{ if task } i \text{ is assigned to processor } k; \\ 0, \text{ otherwise.} \end{cases}$$

$T_{comp_{ik}}$ is the computation cost for task i on processor k. Similarly, $T_{comm_{ij}}$ is the communication cost between task i and task j. The scale factor ω is used to normalize different cost metrics. Because the time complexity to find an optimal allocation, A, increases rapidly as the number of processors increases, either various scheduling constraints or heuristic information has typically been employed to reduce complexity. For example, Ma et al. [MA82] proposed a branch and bound technique using allocation constraints; Shen and Tsai [SHE85] made use of the A^* algorithm to reduce the feasible solution space.

Another interesting cost metric proposed in [BOK81] is cardinality, which is the number of edges connecting tasks which are mapped onto adjacent processors. In this case, the optimal scheduling is one that maximizes cardinality. Stankovic and Sidhu's adaptive bidding algorithm [STA84] takes into account various parameters. They propose interesting cost metrics based on memory utilization, processor workload, task distribution/cluster, and others. The wave scheduling strategy proposed by van Tilborg and Wittie [VAN84] makes use of a unique cost metric: total CPU time to schedule a task group of size S successively. Efe [EFE82] proposed a cost metric to measure processor workload: assuming p is the number of processors in a multiprocessor system, the workload q_i is represented by $\frac{l_i}{l_{avg}}$, where l_i is the queue length in processor i and l_{avg} is defined as $\sum_{i=1}^{p} \frac{l_i}{p}$. The other cost metrics proposed include speedup [AE82], turnaround time [NI81] and scheduling length [KAS84].

2.2.4. Discussion on Previous Strategies

Previous approaches have focused mainly on the development of specific scheduling strategies based on the coupling factors of processors in multiprocessor systems. Some of them also attempt to take advantage of the unique hardware characteristics such as interconnection topologies of multiprocessor systems under consideration. Since each strategy is usually an ad-hoc scheme, it is in most cases applicable to some limited class of multiprocessor architectures (e.g., tightly-coupled homogeneous systems [GOT83a, BBN85a], loosely-coupled homogeneous systems [SOL79, WIT80], loosely-coupled heterogeneous systems [FOR78, CHO82], and multicomputers connected in point-to-point fashion [BRY81]).

Moreover, various simplifying assumptions are common. For example, Bokhari [BOK81] studies the assignment of tasks to processors with the restriction that the number of tasks should be less than or equal to the number of processors. Shen and Tsai [SHE85] propose a graph matching approach for solving task assignment to processors, but ignore dependency relations among tasks. Some approaches have limited scheduling objectives; they find the best schedule with respect to either the total computation time [GOT83a, BBN85a] or interprocessor communication time [HAE80, BIA85]. Other approaches are interested in balancing the workload of the total multiprocessor system [STA84, VAN84, WAN85, EAG86].

In most scheduling strategies for tightly-coupled systems, specific interconnection networks are assumed such as the Butterfly switch, the Omega network, the SW-Banyan network or even a composition of them [PFI85]. On the other hand, most of them do not take into account scheduling constraints, resource limitations, and the current workloads of processors in the system. It is also assumed that each processor is identical (i.e., all have the same processing speed and memory capacity). Furthermore, some multiprocessor systems such as the Butterfly allow allocation and execution of only one task on a processor at a time; each task is nonpreemptive and occupies the processor until its execution is completed. Finally, while most scheduling strategies make heavy use of busy-waiting as a synchronization mechanism, there is little attempt to reduce or avoid using it.

All in all, there are a myriad of multiprocessor scheduling strategies which can be applied to specific multiprocessor systems. On the other hand, there is little research which attempts an integrated approach to multiprocessor scheduling which could be applicable to various multiprocessor systems regardless of underlying architectural characteristics.

2.3. Complexity Issues of Multiprocessor Scheduling Problems

In this section, in order to justify the development of heuristic algorithms, we discuss the complexities of three multiprocessor scheduling problems to be investigated in this thesis:

SP1) Given a minimum schedule length L, what is the minimum number of fully-connected identical processors in order to finish a parallel computation within L, which consists of a set $S = \{s_1, s_2, \cdots, s_m\}$ of mutually independent schedulable units of computation with arbitrary computation times ?

SP2) What is the minimum schedule length of a parallel computation consisting of a set of $S = \{s_1, s_2, \cdots, s_m\}$ of mutually dependent schedulable units of computation with arbitrary computation times and zero communication times on the fixed number of p (≥ 2) fully-connected identical processors ?

SP3) What is the minimum schedule length in SP2, if the unlimited number of fully-connected identical processors are available ?

We first prove that the decision problems DP1 and DP2 corresponding to the first two optimization problems SP1 and SP2, respectively, are *NP−complete*. We then discuss the complexity of the third problem.

Theorem 2.1: DP1 is *NP−complete*.

Proof: In order to get a "yes-no" answer for DP1, we just need a nondeterministic Turing machine which makes a guess at a set of partitions of schedulable units of computations in S and checks in polynomial time whether that the length of each partition and the number of the partitions in the set satisfy the given bounds, respectively. So, this problem is in *NP*.

We now show that DP1 is polynomially reducible from Bin-Packing problem. For a finite set $U = \{u_1, u_2, \ldots, u_m\}$ of items, a size $\sigma(u) \in Z^+$ (where Z^+ represents the positive integers) for each $u \in U$, a bin capacity $C \in Z^+$, and $k \in Z^+$, $B = \{U_1, U_2, \ldots, U_k, C\}$ is a given instance of bin packing of U such that $\max\limits_{1 \leq i \leq k} |U_i| \leq C$, where $|U_i| = \sum\limits_{u \in U_i} \sigma(u)$. An instance of DP1 can be trivially constructed in polynomial time from the instance of bin packing as follows: Given $U =$

$\{u_1, u_2, \cdots, u_m\}$, we construct a set $S = \{s_1, s_2, \cdots, s_m\}$ of mutually independent schedulable units of computation by replacing each $u_i \in U$ with a schedulable unit of computation s_i with a length $l(s_i) = \sigma(u_i)$ for $1 \le i \le m$, and create a (dummy) schedulable unit of computation s_{m+1} such as $l(s_{m+1}) = C$, where C is the bin capacity.

We now need to show that there exists a partitioning $\{U_1, U_2, \cdots, U_k\}$ of U into minimum possible k disjoint subsets such that $\max\limits_{1 \le i \le k}(|U_i|) \le C$ if and only if there exits a partitioning $\{S_1, S_2, \cdots, S_k, S_{k+1}\}$ of $S \cup \{s_{m+1}\}$ into minimum possible $k+1$ disjoint subsets such that $\max\limits_{1 \le i \le k}(|S_i|) \le l(s_{m+1}) = L$, where $|S_i| = \sum\limits_{s \in S_i} l(s)$ and $S_{k+1} = \{s_{m+1}\}$. First, suppose that $\{U_1, U_2, \ldots, U_k\}$ is a partitioning of U with the minimum possible k such that $\max\limits_{1 \le i \le k}(|U_i|) \le C$ Since $l(s_i) = \sigma(u_i)$ for $1 \le i \le m$ and $l(s_{m+1}) = C$, $\{s_1, s_2, \cdots, s_m, s_{m+1}\}$ can be partitioned into $\{S_1, S_2, \cdots, S_k, S_{k+1}\}$ such that $S_i = \bigcup\limits_j \{s_j\}$ for $1 \le i \le k$ and $S_{k+1} = \{s_{m+1}\}$, where $U_i = \bigcup\limits_j \{u_j\}$ and $l(s_j) = \sigma(u_j)$, Therefore, $\{S_1, S_2, \cdots, S_{k+1}\}$ is a partitioning of $S \cup \{s_{m+1}\}$ with the minimum possible k such that $\max\limits_{1 \le i \le k}(|S_i|) \le l(s_{m+1})$. Conversely, suppose that $\{S_1, S_2, \cdots, S_k, S_{k+1}\}$ is a partitioning with the minimum possible k such that $\max\limits_{1 \le i \le k+1}(|S_i|) \le L$. Since all s_i $(1 \le i \le m)$ are mutually independent, a minimum schedule length L becomes $\max\limits_{1 \le i \le m+1}(l(s_i)) = C$. Without loss of generality, let $S_{k+1} = \{s_{m+1}\}$. Since $l(s_i) = \sigma(u_i)$ for $1 \le i \le m$ and $L = C$, $\{u_1, u_2, \cdots, u_m\}$ can be partitioned into $\{U_1, U_2, \cdots, U_k\}$ such that $U_i = \bigcup\limits_j \{u_j\}$ for $1 \le i \le k$, where $S_i = \bigcup\limits_j \{s_j\}$ and $\sigma(u_j) = l(s_j)$. Therefore, $\{U_1, U_2, \cdots, U_k\}$ is a partition of U with the minimum possible k such that $\max\limits_{1 \le i \le k}(|U_i|) \le C$.

□

Theorem 2.2: DP2 is *NP–complete*.

Proof: First, it is trivial to see that this problem is in *NP*, since we just need a nondeterministic Turing machine, which makes a guess at a set of partitions of schedulable units of computation in S and checks in polynomial time whether that the length of each partition and the number of the partitions in the set satisfy the given bounds, respectively.

We can easily show the *NP-completeness* of DP2 by showing that DP2 contains as a special case *Precedence Constrained Scheduling* (PCS) which has been known an *NP-complete* problem [ULL75]. To be specific, if we allow only instances of DP2 in which all the schedulable units of computation in S happen to have the unit computation time, then we obtain a problem identical to the PCS problem. As a result, DP2 is an *NP-complete* problem.

□

We now consider a slightly general version of SP2, which allows a nonzero communication time on each edge. Since this is another general case of PCS problem as long as the sum of the communication times does not exceed the unit time, this problem is also *NP-complete*.

Finally, we discuss the complexity of SP3. *General Scheduling Problem* (GSP) is to identify a schedule which minimizes schedule length of a parallel computation of schedulable units of computation with arbitrary computation times and arbitrary scheduling constraints under the assumption that a variable (but limited) number of identical processors are available. It is well know that GSP is *NP-complete* in the strong sense [KAR72, KAS84]. Even in the case that each schedulable unit of computation has the unit time, finding the minimum schedule is an *NP-compete* problem [ULL73, LLO80]. Since SP3 is more general version of GSP, it also appears to be an intractable problem.

CHAPTER 3

THE MODELS FOR MULTIPROCESSOR SCHEDULING

The fundamental idea of our approach is that multiprocessor scheduling is equivalent to mapping of a computation graph onto an architecture graph. In this chapter, we first present the models of computation graph and architecture graph, and we then discuss the conceptual model of a generalized multiprocessor scheduling strategy based on a series of graph mappings.

3.1. The Graph Models for Multiprocessor Scheduling

This section presents the models for computation graphs and architecture graphs. Computation graphs are directed and acyclic, while architecture graphs are undirected. The former describe parallel computation structures, and the latter specify multiprocessor computer systems.

The computation graph model described provides a framework within which various classes of parallel computation structures can be represented. A computation graph itself contains all necessary information to define the sequences of events of a modeled computation. This leads to determinism in computation graph execution. The computation graph is also labeled with information necessary for scheduling the graph on a target multiprocessor system. Finally, we consider only static computation graphs.

On the other hand, an architecture graph represents a target multiprocessor computer system onto which a computation graph is to be mapped. There are two types of architecture graphs: virtual architecture graphs (*VAG's*) and physical architecture graphs (*PAG's*). A *VAG* depicts a desirable abstract multiprocessor system for the execution of a computation graph; A *PAG* depicts a real multiprocessor system on which the computation graph is to be scheduled.

3.1.1. The Model for Computation Graphs

Karp and Miller [KAR66] first proposed a computation graph as the graphical model to specify parallelism existing in computations. Browne [BRO86] also proposed a directed graph as a representation basis of a parallel computation, in which the nodes represent the bindings of operations to data and the edges represent dependency relations between schedulable units of computation executed at the nodes.

Our computation graph model is a triple $(G_c, f_c^{comp}, f_c^{comm})$, whose first component $G_c = (N_c, E_c)$ specifies a parallel computation. Computation graph G_c is a directed acyclic graph and defined as follows:

(i) A node set $N_c = \{n_1, n_2, \cdots, n_l\}$;

(ii) An edge set $E_c = \{e_1, e_2, \cdots, e_t\}$, where any given edge $e_p = (n_i, n_j)$ is directed from node n_i to node n_j.

To be specific, graph G_c defines computation steps by the nodes and sequencing among the steps by the edges. The remaining components provide information necessary for mapping the computation graph onto a target system. The second component f_c^{comp} is a function which maps each node in N_c onto a positive integer which is the expected computation time used by the schedulable unit of computation corresponding to the node. The next function f_c^{comm} maps each edge (n_i, n_j) in E_c onto a nonnegative integer which is the expected amount of interprocessor communication from node n_i to node n_j. For example, if $f_c^{comm}(e_p) = N_{bytes}$ for $e_p = (n_i,n_j)$, then the total length of messages sent from node n_i to node n_j is N_{bytes} bytes. We denote $n_i < n_j$ if $f_c^{comm}(e_p) \neq 0$ where $e_p = (n_i, n_j)$. We call n_i and n_j a *source* node and a *destination* node, respectively.

In general, a computation can be specified in terms of schedulable units of computation which may be realized as arithmetic expressions, arbitrary collections of statements or procedures in a higher-level programming language such as Fortran, ADA or Pascal, and in terms of dependency relations between these schedulable units. The amount of computation at any node represents the granularity of a computation graph which, in turn, determines the structure of the graph. Note that since each schedulable unit of computation is assigned to a unique node we will use the terms node, task and schedulable unit of computation interchangeably.

In order to implement a meaningful computation, units of computation should be executed in some coordinated order. Sequencing establishes the ordering relationships between primitive units of computation to form schedulable units and between schedulable units of computation to form parallel computation graphs [BRO86]. In our model, sequencing inside a schedulable unit of computation is of any form. For example, it may be nondeterministic, and even back edges may be embedded. On the other hand, sequencing between schedulable units of computation is fixed and deterministic. Furthermore, each schedulable unit of computation is not allowed to be embedded in any do-loops. This can be achieved by unrolling every do-loop existing in a computation graph. In this case, we need to adjust the computation and communication times of the unrolled nodes and edges accordingly.

An edge is an abstract representation of a dependency relation between schedulable units of computation; synchronization and communication are realizations of such dependency relations. A directed edge between a pair of nodes not only represents the transfer of information but also carries the sequencing information between the nodes. In our model, however, sequencing is assumed to be subsumed by communication primitives as in a data flow model of computation.

Interprocessor communication is based on the *asynchronous-send/synchronous-receive* communication model. Whenever needed, a node may send messages to its destination nodes in the order specified by its outgoing edges. A *send* primitive will not be blocked whether its destination node actually receives the messages or not. Each time a node sends a message to another node, the message is assumed to be placed in a queue on the edge between the two nodes. On the other hand, when a node requests a message from the other node, it is blocked from further execution until the message actually arrives at the appropriate input channel. The node also expects input messages to arrive in the order specified by its incoming edges.

As alluded to in the preceding paragraph, our graph model provides more flexible interpretation of dependency relationships than the precedence graph model. For two nodes n_i and n_j in a precedence graph, if $n_i < n_j$, n_j can not begin until n_i has been completed. On the other hand, in our model, once n_i sends a message requested by n_j and has no outstanding requests for messages from the other source nodes, n_j may begin execution. For example, node n_i in the computation (sub)graph shown in Fig. 3-1 functions as follows. Upon being assigned to a processor, a schedulable unit of computation represented by n_i may be triggered and be executed until it needs a message from its first input message from n_{h_1}. As soon as it receives the requested message, it immediately resumes its execution (whether or not n_{h_1} has been completed) until it runs into the next *receive* primitive from n_{h_2}. The input edges may be also considered as synchronization points during execution of node n_i. The output edges from n_i to n_{j_1}, n_{j_2}, and n_{j_3} simply specify the fact that there are three *send* primitives from n_i in the order of the outgoing edges.

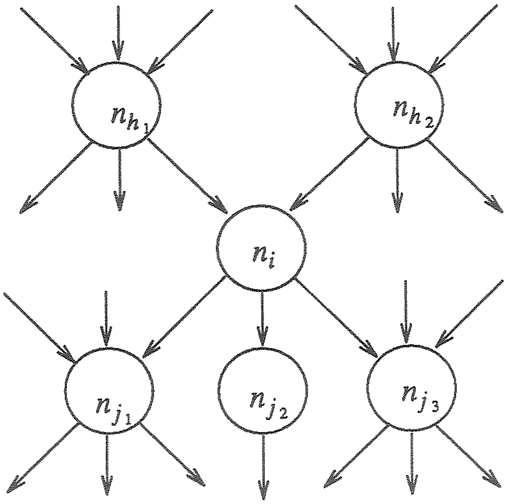


Figure 3-1 A Subgraph of Computation Graph

The exact order of communication primitives (i.e., the *send* and *receive*) issued by a node can not be determined uniquely from a computation graph. As a matter of fact, a given graph may have many feasible orders. More specifically, when a node receives messages from more than one direct ancestor, it expects the messages to arrive in the relative order of its incoming edges (i.e., from left to right). Similarly, when the node sends messages to more than one direct descendant, it sends the messages in the relative order of the outgoing edges (again, from left to right). Any interleaving of the *send/receive* primitives for a node may be regarded as correct as long as the relative orders within the *send* and *receive* primitives are maintained. For example, any number of the *send* primitives may precede any number of *receive* primitives. Table 3-1 shows examples of the *send/receive* orderings which may and may not be represented by node n_i and by its two incoming and three outgoing edges shown in Fig. 3-1. Note that the *send* and *receive* primitives are defined as follows:

- $send(n_j, message)$ - send a message to node n_j;
- $receive(n_h, message)$ - receive a message from node n_h.

The incorrect order in Table 3-1 can not be an issue order specified by node n_j, since it fails to maintain the relative order between the *receive* primitives of node n_i. If only $receive(n_{h_1}, message)$ precedes $receive(n_{h_2}, message)$, the order becomes one of 10 correct orders represented by Fig. 3-1.

A Correct Order	An Incorrect Order
$receive(n_{h_1}, message)$	$receive(n_{h_2}, message)$
$send(n_{j_1}, message)$	$send(n_{j_1}, message)$
$send(n_{j_2}, message)$	$send(n_{j_2}, message)$
$receive(n_{h_2}, message)$	$receive(n_{h_1}, message)$
$send(n_{j_3}, message)$	$send(n_{j_3}, message)$

Table 3-1 Correct/Incorrect Orderings of Communication Primitives

Our computation graph is a restricted model in several ways, some of which have been briefly mentioned above. The critical restriction that makes the model inappropriate for representing some computations is that sets of edges entering and leaving a given node may not be joined by *or* conditions. This prevents us from embedding nondeterminism inside computation graphs. A second restriction is that computation graphs may not have any back edges. This forces us to unroll all the existing cycles so that computation graphs become cycle-free. Also, an edge represents one-way communication channel from one node to the other. That is, it is directed and associated with exactly two nodes, and, as such, there exists exactly one *unidirectional channel* between a pair of communicating nodes (if any). Finally, computation graphs are required to be static; neither new nodes nor new edges can be created during runtime. The main reason for these restrictions is to avoid ambiguity in determining the computation and communication requirements of the nodes and edges in a computation graph. It should be also mentioned here that we assume that a computation graph has one root node and one terminal node without loss of generality.

Some of the restrictions may be relaxed with a little difficulty. For example, the first restriction can be lifted if we enhance algorithm *LinearCluster* (cf. Section 4.1.1.) so that it is able to take into account dynamic information to select paths with the highest probabilities of execution as trace scheduling [FIS84]. From a practical point of view, it may not be always feasible to unroll every do-loop. We discuss how to avoid full expansion of do-loops by removing back edges in some special cases in Section 4.4.

The differences in parallel programming methodologies can be isolated to specifying dependency relations between schedulable units of computation. These methodologies differ mainly in their interprocess communication mechanisms and in the amount of internal parallelism allowed inside a schedulable unit of computation. The dependency relations may be resolved to message or shared memory synchronization operations depending on target multiprocessor systems. In spite of some restrictions, our model can represent a variety of communication model for parallel processing like the message-based communication model (e.g., Hoare's CSP model and ADA's rendezvous model), the shared-memory model and the data flow model among others, if their nondeterminism may be ignored. To be specific, both CSP and rendezvous models establish precedence constraints by suspending either source or destination nodes until the other also executes the corresponding *send/receive* primitives. Our model may establish this kind of communication mode by blocking the sender until a message from the sender has been actually received by the receiver. Communication paradigms based on shared-memory allow communicating nodes to exchange messages through shared variables. A dependency relationship in the shared-memory model is nothing but a synchronization constraint that must be met in order to produce the correct results. Edges in our computation graph model, in fact, specify such synchronization constraints. A static, data-driven data flow model is also a special case of our model. It can be implemented by not triggering (or firing) the execution of a node until all the necessary messages have been received. Finally, it should be mentioned here that even though the model of communication allows sending of messages during execution, the algorithms for linear clustering treat computation graphs as pure precedence graphs for easy identification of linear clusters, where the messages are sent only at the completion of each node.

3.1.2. The Model for Architecture Graphs

The model for architecture graphs provides a representation basis for the structural description of multiprocessor systems. There are three types of resources which are currently considered: processor, communication link and memory. An architecture graph is an undirected graph in that each architecture edge is bidirectional. It is also assumed that an architecture graph is static; the resource configuration of a physical multiprocessor system will not be changed dynamically during runtime. Moreover, it maintains the exact current status of the system. The status includes the information on which processors are currently active/inactive, which communication links are currently available and what is the current memory capacity available in each processor.

Our architecture graph model is also a triple $(G_a, f_a^{comp}, f_a^{comm})$, whose first component $G_a = (N_{a,} E_a)$ is an undirected graph defined as follows:

(i) An architecture node set $N_a = \{an_1, an_2, \cdots, an_l\}$;

(ii) An architecture edge set $E_a = \{ae_1, ae_2, \cdots, ae_t\}$, where any architecture edge $ae_p = (an_i, an_j)$ is undirected.

In an architecture graph, an architecture node represents a processor as well as a memory module, and an architecture edge represents a communication link between two processors. The second component f_a^{comp} is a function which maps each architecture node in N_a onto a pair of positive integers which denote the level of computing power of a processor relative to the others in the system and the current local memory size. A common global memory may be specified by a *dummy* architecture node which is fully-connected with the other architecture nodes. The next function f_a^{comm} maps an architecture edge (an_i, an_j) in E_a onto a positive integer which represents the bandwidth of communication link from architecture node an_i to architecture node an_j and vice versa.

There are two types of architecture graphs: virtual architecture graphs (*VAG's*) and physical architecture graphs (*PAG's*). A *VAG* is an architecture graph which defines a desirable abstract multiprocessor system for the execution of a computation graph, regardless of the operational characteristics of the corresponding real multiprocessor system. A *PAG* is another type of architecture graph corresponding to a real target multiprocessor system on which the computation graph is to be executed.

It is the most desirable for a parallel computation structure if a multiprocessor system is available which has a sufficient number of processors with an unlimited amount of memory and enough communication links so that every schedulable unit of computation can be assigned to a separate processor, and the adjacency between schedulable units of computation can be maintained after the mapping. In fact, it is a goal of silicon compiler research to implement directly such abstract architectures on VLSI chips. While it may be possible to reach this goal, it would not be always feasible to pursue it for every computation graph. A virtual architecture graph representation provides a basis for specifying such an abstract multiprocessor system which is expected to be the best for a given computation graph in terms of performance. Depending on the information available on the target system in the course of a series of mapping, a *VAG* may be transformed into another *VAG*.

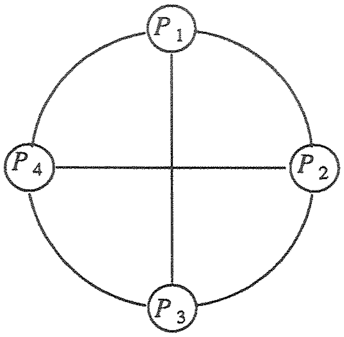


Figure 3-2 Physical Architecture Graph for the Butterfly System with 4 Processors

A physical architecture graph represents an operational view of the actual resource configuration of a multiprocessor system. It may or may not be exactly the same as the real resource configuration of a multiprocessor system. Fig. 3-2 depicts a physical architecture graph for the Butterfly system with four processors. In fact, the *PAG* does not depict exactly the Butterfly system which utilizes the Butterfly switch. The exact representation of physical resources is in fact not important for the scheduling itself as long as those two have the same operational behavior. For example, although the Butterfly switch does not provide dedicated paths between each pair of processor nodes, there is a path through the packet switching network from each processor node to other nodes. That is, the Butterfly switch operates as if there were fully-connected direct communication links between processors. The *PAG* in Fig. 3-2 describes such an operational behavior of the Butterfly system.

3.2. The Conceptual Model for Generalized Multiprocessor Scheduling

This section introduces the model for generalized multiprocessor scheduling and the mapping strategies based on this model. Also explained are logical and physical mappings and their differences.

3.2.1. Generalized Multiprocessor Scheduling Model

We mentioned in Section 2.1 that the degree of automation of the mappings and bindings is determined by the level of visibility of a multiprocessor system exposed to the user. In this section, we introduce a model for generalized multiprocessor scheduling based on the level of visibility.

Various scheduling strategies (from *ad−hoc* to *restricted*) can be accomplished by utilizing virtual architecture graphs and the mappings between them. A *VAG* is mapped onto another *VAG* which represents the physical resource configuration of a multiprocessor system more accurately or gives more detailed resolution of the physical architecture. Given n levels of visibility, assume that a *VAG* graph at level 1 represents a multiprocessor system whose physical resource configuration is not visible to the user at all. Since nothing related to physical resources is visible, it is reasonable to assume that the *VAG* at level 1 represents a multiprocessor system with unlimited amounts of resources. Virtual architecture graphs at intermediate levels represent multiprocessor systems whose configurations are partially visible. At the other end, a *VAG* at the level n represents a multiprocessor system whose configuration is completely visible to the user. Fig. 3-3 describes the conceptual model of scheduling strategies for multiprocessor systems. Assumed here are three levels of visibility among *VAG's*. It makes it easy to illustrate the basic principles of our model; however, there is no reason to limit the number of levels of visibility.

A physical architecture graph represents a real multiprocessor system. Moreover, it maintains the exact current status of the system. The status includes the information on which processors are currently active/inactive, which communication links are currently available, the current workload of each processor and each communication link, the size of each memory module currently in use (conversely, not in use), and others. A *VAG* at level n and a *PAG* are common in that both of them represent a real multiprocessor system. While the former maintains a status of resources which is not necessarily up-to-date, the latter keeps the most up-to-date status of all the resources in the system.

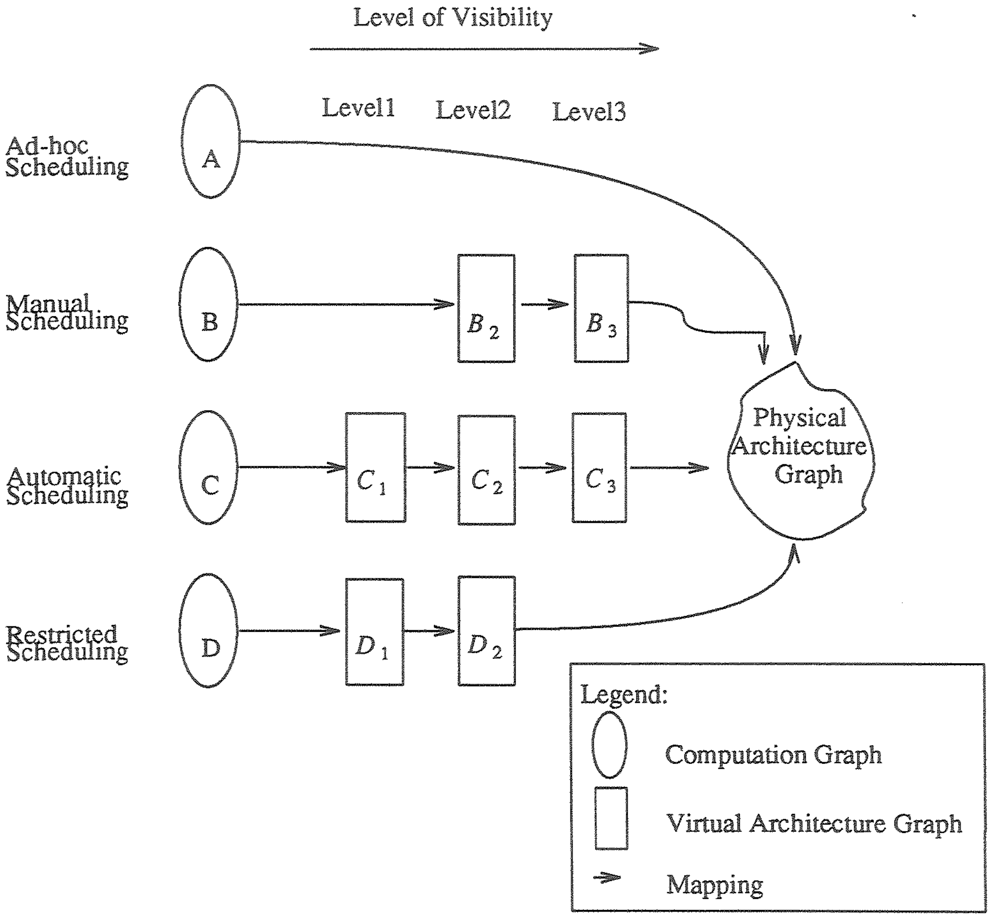


Figure 3-3 Conceptual Model of Multiprocessor Scheduling Strategies

We now explain how different multiprocessor scheduling strategies can be realized in our model depicted in Fig. 3-3. First, in the case of ad-hoc multiprocessor scheduling, there is no intermediate mapping between computation graph A and the PAG; the computation graph can be directly mapped onto the PAG. This implies that the characteristics of physical resources are completely exposed to the user. Therefore, he can take advantage of this visibility to get the maximum performance out of the system. The operations to manipulate all processors in the system may be specified at every single time step. As a result, the user is left with the entire burden of deriving a schedule by which all data dependency constraints are satisfied. The approach taken by systolic arrays [KUN82] is an example of this kind of model.

In the case of manual multiprocessor scheduling, there exist two intermediate mappings before computation graph B is eventually mapped onto the physical architecture graph. At the first stage of mapping, the computation graph is mapped onto virtual architecture graph B_2 at level 2, which represents a partially visible multiprocessor system. Due to the partial visibility of the multiprocessor system, the user may construct parallel computation structure so that resources in the system can be utilized more efficiently than in automatic multiprocessor scheduling (but not as much as in ad-hoc scheduling). This type of scheduling also takes advantage of the partial visibility of the physical multiprocessor system to determine better mappings. Medusa [OUS80, OUS82] is based on this kind of scheduling model.

The next model is automatic multiprocessor scheduling. In this case, since a physical multiprocessor system is completely invisible to the user, computation graph C is mapped onto virtual architecture graph C_1 at level 1 and eventually mapped onto the physical architecture graph after one or more intermediate mappings. During these mappings, more and more accurate information pertaining to the physical system becomes available. Most scheduling strategies [BBN85b, EDL85b, TAN85] for tightly-coupled systems are based on this model. Even though they provide a limited version of manual scheduling strategy as well [BBN85b, PFI85, SEQ86], they try to make the architectural details invisible to the user as far as possible.

A variation on the previous two strategies are restricted automatic scheduling strategies. In this case, even at the final stage of the mapping (e.g., from virtual architecture graph D_2 to the physical architecture graph), the physical multiprocessor system is only partially visible to the user; the actual physical multiprocessor system is masked in such a way that the user is allowed only to make use of a part of the available resources in the system. This makes it possible for physical resources to be partitioned into different clusters (which are not necessarily mutually disjoint). This partitioning may affect the utilization of some resources, reduce the complexity of the scheduling and enforce some specific scheduling constraints. This scheduling model can represent the multiprogramming paradigm in the Cedar system [GAJ83], which is physically partitioned into clusters of one or more processors to run a number of different application programs. The IBM RP3 [PFI85] can also be partitioned into completely independent clusters through bounds registers. Such a partitioning can be easily supported by this restricted scheduling model as well.

3.2.2. Logical and Physical Mappings

A series of mappings from a computation graph to a physical architecture graph can be decomposed into two distinct mappings: *logical mapping* and *physical mapping*. A computation graph is first mapped onto a virtual architecture graph in most cases. Then the *VAG* is successively mapped between pairs of *VAG's* until it is mapped onto the physical architecture graph. These mappings of a computation graph onto a *VAG* and its subsequent mappings onto the other *VAG's* are called logical mappings. During logical mappings, it is assumed that the whole system is available to a single application program.

At the time when one of the virtual architecture graphs is finally mapped onto the physical architecture graph, the actual allocation of tasks to processors occurs. This final mapping is called physical mapping. This mapping takes into account the current status information on physical resources (e.g., active/inactive processors, available memory size, channel connectivity and capacities, etc.) and the workload of each processor in the system. Note that ad-hoc scheduling requires only one physical mapping.

CHAPTER 4

MULTIPROCESSOR SCHEDULING BASED ON LINEAR CLUSTERING AND MERGING

Clustering techniques have been used in a variety of areas in computer science [CHI84, BAN87]. In this chapter, we propose a new multiprocessor scheduling technique based on linear clustering and linear cluster merging. We first characterize and justify linear clustering under the assumption that computation graphs are acyclic. We then discuss linear cluster merging and its properties. We also propose algorithms for iterative refinements of linear clusters (if necessary) for the minimization of schedule length. Finally, we propose a set of schemes to transform cyclic computation graphs into acyclic ones in order to make linear clustering be applicable to a broader class of parallel computation structures. Note that we use the terms *computation graph* and *computation* interchangeably in this chapter.

4.1. Linear Clustering

Linear clustering is a fundamental idea of multiprocessor scheduling algorithms discussed in Chapter 5. In this section, after characterizing linear cluster, we introduce algorithm *LinearCluster* to identify linear clusters from an acyclic computation graph. Then we justify why this idea is efficient for multiprocessor scheduling.

4.1.1. Characterization of Linear Cluster

Given a computation graph $G = (N, E)$ with node set N and edge set E, a clustering of G is defined as a cutting of G into a set of nonempty and disjoint connected subgraphs by removing some edges in G. In other words, a clustering is an assignment function which assigns a weight of either 0 or 1 to each edge of E. A graph is said to be *connected* if there is a path between any of its two nodes (when the directions of edges are ignored).

A cluster of G is called a *linear cluster* K if it satisfies the following conditions:

- K is nonempty;
- K is a connected subgraph of G;
- Both indegree and outdegree of every node in K is less than or equal to 1.

Linear clustering is a special case of general clustering in that a linear cluster is a degenerate tree in which each node has at most one direct ancestor and/or one direct descendant, while a cluster, in general, is an arbitrary graph. Note that the root and leaf nodes of the tree are called the *header* and *trailer* nodes, respectively. Scheduling algorithms based on general clustering are not appropriate for multiprocessor scheduling since they do not take into account potential parallelism among tasks when finding clusters. However, general clustering may be useful for identifying a set of clusters which minimize intercluster communication overhead.

The following algorithm *LinearCluster* illustrates how to identify linear clusters:

```
LinearCluster ($G$, $K$)

/* $G$ is a (cycle-free) computation graph. */
/* $K$ is a set of linear clusters. */

Begin
  Let $K = \emptyset$;
  Find a longest path $P$ from the root to a leaf node in $G$;
  During traversing path $P$ backward from the leaf to the root node,
      cut all the incoming and outgoing edges except the one belonging to $P$;
  For each connected subgraph $S$ of $G$,
    If both indegree and outdegree of each node in $S$ is less than or equal to 1,
    Then
       $K = K \cup S$
    Else Do
          LinearCluster ($S$, $K'$);
          $K = K \cup K'$;
    End Do;

End LinearCluster.
```

A path $(n_1, e_1, n_2, \ldots, e_{l-1}, n_l)$ of graph $G = (N, E)$ such that $n_i \in N$ and $e_i \in E$ is considered the longest path if it maximizes the following objective function:

$$\sum_{i=1}^{l-1}(\omega_1 \cdot T_{comp_i} + (1-\omega_1)\cdot(\omega_2 \cdot T_{comm_i} + (1-\omega_2)\cdot\sum T_{comm^i_{adj}})),$$

where T_{comp_i} is the computation time of node n_i, T_{comm_i} is the communication time of n_i with adjacent node n_{adj}, and $T_{comm^i_{adj}}$ is the communication time of n_{adj} with its neighbors other than n_i. ω_1 and ω_2 are normalization factors. We explain the motivation behind the cost function in detail in Section 5.1. A longest path P may vary with input data and initial state of computation. A path which is expected to be the longest may not even be executed for some input data. In order to determine a longest path uniquely, we restrict ourselves to a graph, all of whose edges are actually traversed. Furthermore, the nodes and edges in the graph are labeled with appropriate information such as computation and communication times as defined in Chapter 3.

We find a longest path based on a simple modification of *Dijkstra's algorithm* [AHO83] for the single source shortest paths problem. The time complexity to identify all linear clusters in a graph at the worst case is $O(l^3)$, since *Dijkstra's algorithm* takes $O(l^2)$ for each subgraph, where l is the number of nodes in the graph.

4.1.2. Justification of Linear Clustering

There are three types of multiprocessor scheduling techniques which have been frequently referenced in the past: list scheduling, critical path scheduling and Coffman-Graham scheduling. Prior to justifying linear clustering, we compare briefly those scheduling techniques to motivate our technique. All in common construct a *priority list* L for tasks in computation graph G in some order. In *list scheduling*, L is constructed in an arbitrary order. Then the tasks are scheduled as follows: As soon as a processor becomes idle, it searches list L from its beginning until it finds the first task T, all of whose direct ancestors have already been completed. After being removed from L, the task is assigned to the processor. If it fails to locate such a task, the processor remains idle.

The other two scheduling techniques are based on level number, which is defined as the length of a longest path from a task to a terminal task (i.e., a task with no descendants). They only differ in how to construct the priority list L. In *critical path scheduling* [COF76], the tasks are included in list L in descending order of their level numbers. This scheduling technique tries to minimize the total execution time by first executing tasks far from terminal tasks. In *Coffman–Graham scheduling* [COF72], the list also contains tasks in descending order of their level numbers as critical path scheduling. If more than one task has the same level number, the scheduling gives priority to the task which has more direct descendants than others rather than choosing one randomly as in critical path scheduling. A similar idea may be used if any scheduling information is not available or very difficult to estimate.

A list constructed by Coffman-Graham scheduling is a special case of critical path scheduling, which in turn is a special case of list scheduling. As a result, critical path schedules are a subclass of list schedules, and Coffman-Graham schedules are a subclass of critical path schedules. Particularly, Coffman-Graham schedules are guaranteed to be optimal in the limited case that each task has unit execution time, and the number of processors is two, while list and critical path schedules do not necessarily yield optimal schedules for any cases.

These simple scheduling techniques make it possible to maximize potential parallelism by utilizing idle processors as long as there are any ready tasks whose precedence constraints are satisfied. On the other hand, they completely ignore interprocessor communication overhead. None of them takes into account the relationship among tasks when assigning them to processors. For example, for two adjacent tasks T_1 and T_2, suppose that T_1 has been assigned to processor P_{T_1}. According to these techniques, T_2 may be assigned to any idle processor whether or not it is adjacent to P_{T_1}. List schedules may be effective for tightly-coupled homogeneous systems (e.g., the Sequent Balance [SEQ86] or the Encore Multimax [ARG86]). On the other hand, they will not work well for loosely-coupled systems (e.g., the Cosmic Cube [SEI85]) unless tasks are mutually independent. If we are only interested in minimizing the total communication overhead, we could rely on general clustering and find clusters which minimize intercluster communication. However, it may prevent tasks executable in parallel from being executed concurrently in most of cases since two mutually independent tasks may be contained in the same cluster.

Therefore, the fundamental problem for multiprocessor scheduling is how to compromise between potential parallelism and interprocessor communication and synchronization overhead so as to minimize the total execution time of a computation graph. This leads to four goals for an optimal multiprocessor scheduling:

- Minimization of interprocessor communication by allocating tasks that are serially dependent on one another to the same processor;
- Maximization of potential parallelism by allocating tasks that can be executable in parallel to separate processors;
- Throughput enhancement by executing as many computation graphs as possible simultaneously;
- Workload balancing among processors.

We claim that linear clustering is an efficient heuristic to accomplish these goals; it compromises between the first two conflicting goals and satisfies the other two goals relatively well. First of all, it attempts to minimize communication overhead by allocating all the tasks in a linear cluster to one processor. By the nature of linear clusters, since all of the tasks are serially dependent, they should be executed in a sequential order. Even though we allocate them to separate processors, only one of the processors is active at a time. As a result, it should be much more efficient to allocate all the tasks in the linear cluster to a single processor than to allocate them to two or more processors (disregarding workload balance). The unnecessary task distribution should incur the extra communication overhead caused by interprocessor communication, which takes many more clock cycles than intraprocessor communication.

Furthermore, regardless of the amount of hardware resources available to a given computation graph, there exists a linear sequence of tasks whose execution time is no less than the total execution time of a computation graph. The sequence is called the *critical path*. Since tasks on the critical path are serially dependent, the only way to reduce the critical path length (i.e., to reduce the total execution time) is to minimize interprocessor communication overhead among the tasks. Theoretically, linear clustering makes it possible to achieve the minimum execution time that is equivalent to the sum of computation times of tasks on the critical path.

After linear clustering, the original computation graph is transformed into a virtual architecture graph. Since the latter usually has many fewer nodes and edges because it is transformed into a simplified graph, we have a better chance to maintain the node adjacency of the latter than the former during mapping onto a physical architecture graph. All these factors contribute significant reduction of the total communication overhead. Note that due to the simplified graph, the complexity of mapping algorithms should be also reduced.

Secondly, linear clustering attempts to maximize potential parallelism by clustering a given computation graph into disjoint linear clusters in such a way that tasks executable in parallel are assigned to different clusters. As long as they are assigned to separate processors, they can be executed in parallel. Since each processor component in a multiprocessor system is assumed to be a sequential engine, it is better not to assign two tasks that potentially can be executed in parallel to the same processor.

In order to identify potential parallelism among tasks, we may rely on a transitive closure algorithm. Two tasks can be executed in parallel unless they are directly connected in the transitive closure of the graph. Warshall developed an $O(n^3)$ transitive closure algorithm [WAR62], which is impractical for large computations. The important observation is that no matter what algorithms we make use of to identify parallel tasks, we can accomplish maximum parallelism if any task can start execution as soon as all of its dependencies are satisfied. To be specific, all direct descendants of a task can be potentially executed in parallel if each descendant has identical communication overhead with the task and any descendant task is neither a direct nor an indirect child of another descendant task. If the direct descendants are allocated to separate processors, we can possibly maximize potential parallelism existing in a computation. Algorithm *LinearCluster* generates clusters of tasks so that direct descendants of a task always belong to different linear clusters. Assuming that the number of processors is greater than or equal to the number of linear clusters, we can allocate different linear clusters to separate processors; consequently, all the descendant can possibly be executed in parallel.

Next, linear clustering attempts to increase the throughput of multiprocessor systems. As briefly mentioned above, because of data dependencies among tasks in a parallel computation, there is always a minimum time span to complete the computation even if an unlimited amount of resources is available. Even if more processors than necessary can be involved in the computation (whenever there are idle processors and ready tasks), it may not reduce the execution time of the computation at all. On the contrary, it may increase overhead caused by interprocessor communications and remote memory accesses due to unnecessary (or excessive) distribution of tasks. It may turn out to be counter-productive if we assign each task to a processor only because the processor is idle. This phenomenon is similar to one of four multiprocessor *abnomalities* observed by Graham [GRA69]: the total execution time of a given parallel computation can increase as the number of processors involved in the computation increases.

Therefore, there should be an optimal number of processors for executing the computation in the smallest time. We prove in Section 4.2.4. that the sufficient number of processors for the computation is always less than or equal to the number of linear clusters. After linear clustering, it is reasonable to expect that we need fewer processors to execute the clustered graph without affecting potential parallelism at all than to execute the original graph as it was. If the number of linear clusters is less than the available processors in a multiprocessor system, since the rest of processors in the system can be utilized for the execution of other computation graphs, the throughput of the system can be improved.

Finally, linear clustering attempts to improve the workload balance of each processor. In some cases, the execution time of two clusters K_1 and K_2 assigned to separate processors are not overlapped at all. To be more specific, K_2 may start only after K_1 finishes, or it is executed only while K_1 is idle. Merging those linear clusters should not affect the total execution time, but should reduce the idle time of processors and the number of processors necessary for a computation.

There are other advantages of linear clustering. Using linear clusters generated by algorithm *LinearCluster*, we can construct a virtual architecture which is the most suitable for a given computation graph. No matter what the target architecture will be, this virtual architecture may be regarded as an optimal one for the computation graph in the sense that the execution time of the graph can be minimized on the virtual architecture. The reason for the minimization is that linear clusters of mutually independent tasks are assigned to separate processors so that the tasks can be executed in parallel as long as possible, and the adjacency among clusters is always to be maintained so that intercluster communication overhead can be minimized. As a matter of fact, it has been observed by experiments in [DEM82] that an almost linear speedup might be obtained if a computation graph corresponds well to an architecture graph.

Above all, linear clustering is very suitable for generalized multiprocessor scheduling. What we are really looking for, at the stage of linear clustering, is which path in a computation graph takes the longest time to complete. Due to the characteristics of the computation graph model defined in Chapter 3, we can uniquely determine the critical path in the computation graph. Furthermore, we always assign all the tasks on the critical path to a single processor. As a result, virtual architecture graphs can be constructed independently of target architectures. If we do not apply linear clustering to the computation graph, then especially in the case of a heterogeneous system, it is impossible to figure out the critical path. This is because it is calculated based on the estimated execution times of tasks and the execution times in fact depend on which processors the tasks are assigned to. Consequently, we do not know which path is the critical path until all the tasks are bound to physical processors.

After identifying an optimal architecture based on linear clusters, we just need to search for an optimal mapping of the architecture onto target architectures. Whether it is homogeneous or heterogeneous, the characteristics of the target architecture actually matter only during physical mappings. Furthermore, we can identify which target architecture is the best for a given computation graph prior to actually running the program on the target architectures by comparing the results of mapping the virtual architecture onto them.

4.2. Linear Cluster Merging and Its Optimality

In this section, we investigate a means to merge two or more linear clusters into one without affecting potential parallelism existing in a computation graph. It may contribute to further balancing the workload of processors. It may also contribute further reducing the amount of resources to be utilized and interprocessor communication overhead. Then we find the sufficient number of processors to exploit all potential parallelism and to minimize the total execution time of a given computation graph. In order to support our claim, we make use of the level number of a node.

4.2.1. Level Number and Related Definitions

The *level number* concept has played an important role in multiprocessor scheduling in the past. Depending on node labeling methods, the level number of a node can be uniquely determined in a computation graph. The level number of a node, however, can not uniquely identify the node; the nodes with the same level number are not necessarily the same node. There are two approaches to assigning a level number to each node: *bottom−up* and *top−down*. In the bottom-up approach, the level number of a node is defined as the length of the longest path from that node to a terminal node:

$$\begin{aligned} level(T) &= 1 \text{ if } T \text{ is a leaf node;} \\ &= [max(level(D) \text{ for each direct descendant } D \text{ of } T)]+1, \\ &\quad \text{otherwise.} \end{aligned}$$

In the top-down approach, the level number of a node is the length of the longest path from a root node to that node:

$$\begin{aligned} level(T) &= 1 \text{ if } T \text{ is a root node;} \\ &= [max(level(A) \text{ for each direct ancestor } A \text{ of } T)]+1, \\ &\quad \text{otherwise.} \end{aligned}$$

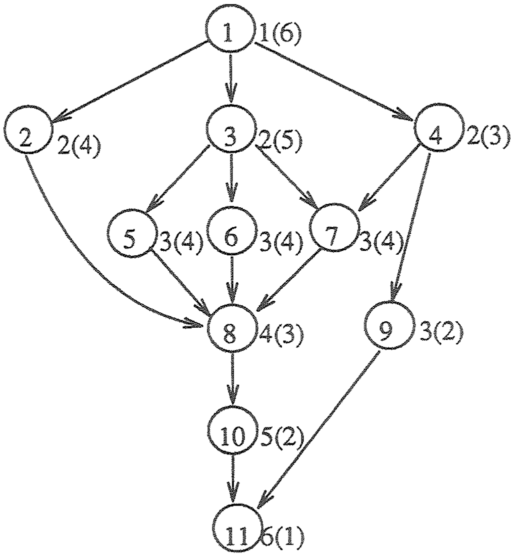


Figure 4-1 Level Numbers (Top-Down *vs* Bottom-Up)

Three multiprocessor scheduling techniques proposed in the past [LLO80] utilized the level numbers to determine the order of task execution based on the bottom-up approach. In our research, we make use of them in a different way. As observed in [PAT84], these level numbers may be used to identify potential parallelism in a computation graph if the top-down approach is adopted. When the level numbers are assigned from top to bottom, the same level number implies mutual independence. To be more specific, if a group of tasks have the same level number, they are mutually independent and may be simultaneously executable. If there exists any data dependency between two tasks, they will have different level numbers. On the other hand, when the bottom-up approach is utilized, tasks which can be executed in parallel may be given different level numbers. As an example, consider Fig. 4-1. Note that the number inside each node represents its node number, while the number beside each node represents the level number based on either the top-down or bottom-up (if in parenthesis) approach. When the level numbers are assigned from bottom to top, the nodes 2, 3 and 4, which are the direct descendants of node 1, have different level numbers, even though they can be executed simultaneously. From here on, we assume that level numbers are assigned using the top-down approach.

If every task in a computation graph has the identical computation time and the same amount of communication overhead with its adjacent tasks, then tasks with the same level number can be always executed in parallel. Otherwise, they may not always be executable in parallel. For example, if they have different communication overheads with their common direct ancestor, they can not be triggered simultaneously. If they have a non-identical set of direct parents (e.g., nodes 6 and 7 in Fig. 4-1), depending on the computation times of their parents, they may or may not be executed in parallel. If that is not the case, their execution may be partially overlapped. It should be noted here that even two tasks with different level numbers can be also computed in parallel if there is no path between them (e.g., node 8 and node 9). It requires exact estimation of computation and communication times and tedious analysis of timing to identify idle periods of linear clusters. In this research, we are not interested in merging which utilizes such idle periods.

We now define some terminology and notation which are frequently referenced in this section. Let L_i represent a set of level numbers assigned to tasks in linear cluster K_i. Two linear clusters K_i and K_j are said to be *sequentially* *strong −dependent* if they satisfy the following conditions:

1) $L_i \cap L_j = \varnothing$;
2) The trailer node of linear cluster K_i precedes the header node of linear cluster K_j.

Condition 2) above implies that the first task in K_j can be triggered only after the last task in K_i is completed. In this case, linear clusters K_i and K_j are called *master* and *slave clusters*, respectively, if the latter depends on the former. More than two linear clusters are said to form a sequentially strong-dependent group if every pair of them are sequentially strong-dependent.

Two linear clusters K_i and K_j are said to be *mutually strong −dependent* if they satisfy the following conditions:

1) $L_i \cap L_j = \varnothing$;
2) For two tasks T_1 and T_2 in K_i, T_1 is a direct ancestor of T_2, where the former is one of direct ancestors of the header node of K_j and has the largest level number among the direct ancestors, and the latter is one of direct descendants of the trailer node of K_j and has the smallest level number among the direct descendants.

More than two linear clusters are said to form a mutually strong-dependent group, if every pair of slave clusters is mutually strong-dependent or sequentially strong-dependent. Note that a linear cluster in a mutually strong-dependent group is called a *master* if it has the smallest and the largest level number among linear clusters in the group. The other clusters are called the *slaves*.

The other notations we need to define are as follows:

- M_i: The i th set of linear clusters which are mutually strong-dependent;
- $|M_i|$: The cardinality of M_i;
- S_i: The i th set of linear clusters which are sequentially strong-dependent;

- $|S_i|$: The cardinality of S_i;
- N_{msd}: The number of sets consisting of linear clusters which are mutually strong-dependent and merged eventually;
- N_{ssd}: The number of sets consisting of linear clusters which are sequential strong-dependent and merged eventually;
- N_{lc}: The number of linear clusters generated by algorithm *LinearCluster*.

4.2.2. Linear Cluster Merging

Algorithm *LinearCluster* (cf. Section 4.1.1.) recursively partitions a computation graph in such a way that direct descendants of a task are assigned to separate linear clusters. Based on the observation made in the previous section, we now introduce another algorithm *MergeCluster* which merges linear clusters if they are not executable in parallel.

```
MergeCluster ($K$)
/* $K$ is a set of linear clusters. */
Begin
  Do
    For each pair of linear clusters $K_i$ and $K_j$ in $K$
        in the order of intercluster communication overhead,
      If $L_i \cap L_j = \varnothing$,
      Then
        If $K_i$ and $K_j$ are mutually strongly-dependent and/or
          sequentially strong-dependent,
        Then
          Merge them into one cluster $K_{ij}$;
  Until no clusters are merged;
End MergeCluster.
```

Since the order of merging is important by Lemma 4.2.3 and Lemma 4.2.4 (cf. Section 4.2.4.), when there are a set of linear clusters, any pair of which can be merged, we first need to choose a pair of clusters which reduces the total communication overhead the most after merging. This selection procedure takes $O(l^2)$ at the worst case, where l represents the number of linear clusters prior to merging. Since the number of linear clusters may be decremented by one at a time during every iteration at the worst case, the complexity of algorithm *MergeCluster* is $O(l^3)$.

4.2.3. Merging Conditions

In this section, we discuss two merging conditions mentioned in algorithm *MergeCluster*. Once assigned to separate processors, linear clusters can be executed potentially in parallel if data dependencies among them are satisfied. In certain cases, however, even when they are assigned to separate processors, they may not be executed simultaneously.

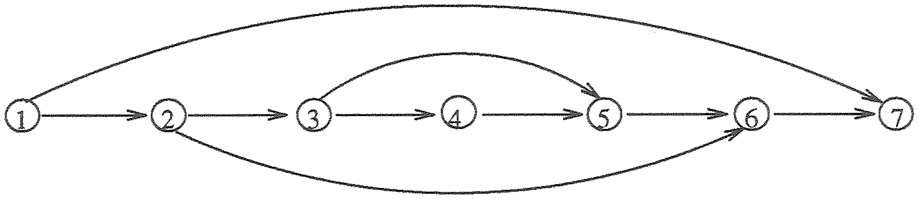


Figure 4-2-a Computation Graph

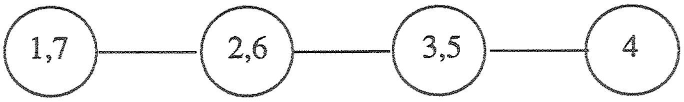


Figure 4-2-b Virtual Architecture Graph after Linear Clustering

As a pathological example, consider the computation graph in Fig. 4-2-a. Suppose that algorithm *LinearCluster* partitions the graph into four linear clusters as shown in Fig. 4-2-b. Notice that no processor can be utilized concurrently with other processors, even after each cluster is assigned to separate processor. The reason is that the linear clusters, which are mutually strong-dependent, are assigned to separate processors.

Suppose that two linear clusters K_i and K_j are mutually strong-dependent. By the definition of mutual strong-dependency of linear clusters, $L_i \cap L_j = \emptyset$, i.e., no tasks in the clusters share the same direct ancestor. Next, as shown in Fig. 4-3, there should be a direct path from T_1 to T_n, where T_1 is a task which has the largest level number among direct ancestors of T_2 in K_i, and T_n is a task which has the smallest level number among direct descendants of T_{n-1} in K_i.

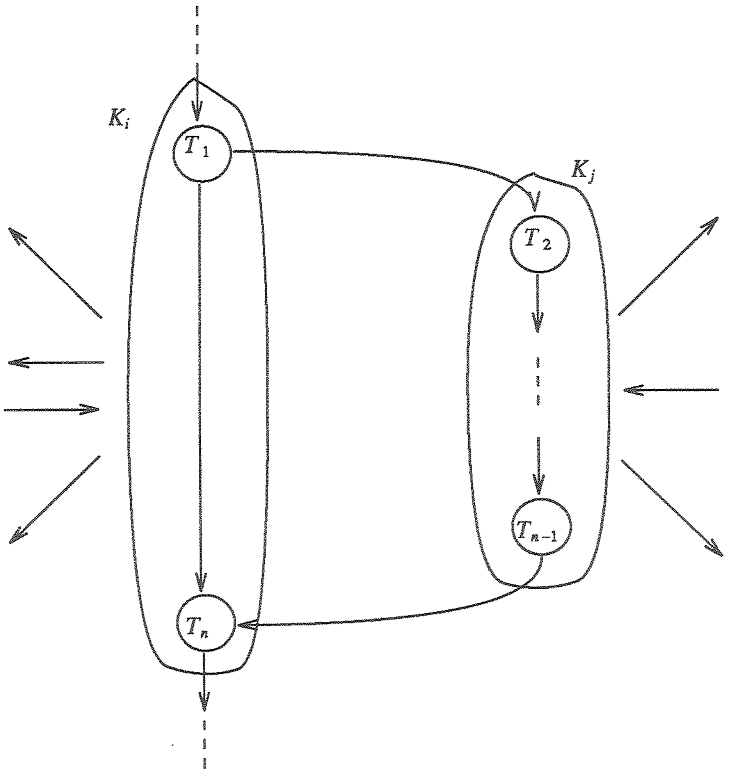


Figure 4-3 Mutually Strong-Dependent Linear Clusters

Furthermore, neither can the execution of T_n be triggered until (at least) all the tasks of linear cluster K_j are completed nor can the execution of T_2 be triggered until (at least) T_1 is completed. There is no reason to assign K_i and K_j to separate processors. As a result, we can merge them into one cluster without sacrificing potential parallelism at all. Note that this is true independent of the data dependencies of these linear clusters on other linear clusters. Assuming there are N_{lc} linear clusters before merging, N_{lc} processors should be more than sufficient when there exists at least one pair of linear clusters which are mutually strong-dependent.

In order to demonstrate how to merge mutually strong-dependent linear clusters, we make use of the computation graph shown in Fig. 4-4-a. It is assumed that the graph has been partitioned into four linear clusters: $\{K_1 = (1 \rightarrow 3 \rightarrow 9 \rightarrow 12), K_2 = (2 \rightarrow 5 \rightarrow 8 \rightarrow 11), K_3 = (4 \rightarrow 7 \rightarrow 10), K_4 = 6\}$.

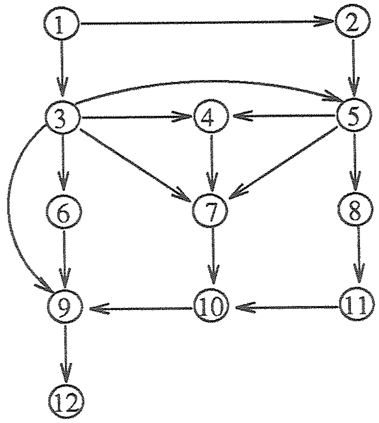


Figure 4-4-a Computation Graph

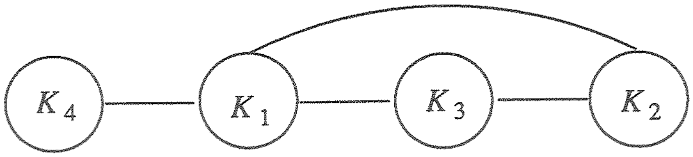


Figure 4-4-b Virtual Architecture Graph before Any Merging

Based on these linear clusters, we can construct a virtual architecture graph which represents an optimal architecture for the computation graph as shown in Fig. 4–4–b. Linear cluster K_1 has two clusters, K_3 and K_4, as its slave clusters; i.e., both of them are mutually strong-dependent on K_1. Assuming that K_1 has greater communication overhead with K_3 than with K_4, K_1 and K_3 can be merged into one cluster (K_{13}) as follows:

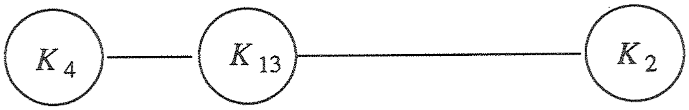


Figure 4-4-c Virtual Architecture Graph after Merging Mutually Strong-dependent Clusters

In Fig. 4-4-c, there are two sets of linear clusters $\{K_{13}, K_2\}$ and $\{K_{13}, K_4\}$ which seem to be mutually strong-dependent. However, neither can we merge K_{13} and K_2, since $L_{13} \cap L_2 \neq \emptyset$, nor can we merge K_{13} and K_4, since there is no direct path between task 3 and 9 any more after K_1 and K_3 have been merged into K_{13} (even though $L_{13} \cap L_4 = \emptyset$). Fig. 4-4-c shows a virtual architecture graph after merging mutually strong-dependent linear clusters.

In general, a master cluster may have more than one slave cluster, as K_1 in Fig. 4-4-b. After the master cluster and one of its slave clusters are merged into one cluster, some of the other slave clusters may not remain as slave clusters any more. Consequently, depending on the order of merging, different virtual architecture graphs may be generated. In order to reduce interprocessor communication overhead as much as possible, the master cluster is merged with the slave clusters in decreasing order of the communication overhead with them. When there are N_{msd} groups of linear clusters which are mutually strong-dependent and merged eventually, the reduction in the total number of linear clusters achieved by merging is $(\sum_{i=1}^{N_{msd}} (|M_i| - 1))$.

We are now concerned with linear cluster merging based on *sequential strong-dependency*. Suppose that two linear clusters K_i and K_j are sequentially strong-dependent. By definition, $L_i \cap L_j = \varnothing$. Next, the trailer task of one linear cluster (say, K_i) precedes the header task of the other linear cluster (say, K_j). Since a processor assigned to K_j will be idle while the tasks in K_i are being executed, we can also merge them into one cluster without sacrificing potential parallelism at all. Suppose that $|S_i|$ linear clusters are sequentially strong-dependent. Then they can be merged into one linear cluster without sacrificing any potential parallelism. Such a merger reduces the number of the clusters by $|S_i| - 1$. When there are N_{ssd} groups of such linear clusters that are merged eventually, the reduction in the total number of linear clusters achieved by merging is $\sum_{i=1}^{N_{ssd}} (|S_i| - 1)$.

4.2.4. Properties of Linear Clustering and Merging

Based on the discussion in the previous section, Lemma 4.2.1 shows how many processors are sufficient to fully exploit potential parallelism available in a given computation graph. In the following lemmas, we assume that the number of available processors is always greater than the number of tasks executable in parallel.

Lemma 4.2.1: It is sufficient (but may not be necessary) to utilize $(N_{lc} - \sum_{i=1}^{N_{msd}} (|D_i| - 1) - \sum_{i=1}^{N_{ssd}} (|S_i| - 1))$ processors to fully utilize potential parallelism available in a computation graph on a given physical architecture graph.

Proof: First we observe that any two tasks (whether or not they are direct descendants of a task) can be executed in parallel when neither direct nor indirect paths exist among them. In order to find out the sufficient number of processors to exploit fully the potential parallelism available, we need to identify a maximal set of tasks which have neither direct nor indirect paths with any other tasks in the set.

As mentioned in the above, there are two cases in which two tasks can be merged without affecting potential parallelism: when they are mutually strong-dependent and/or sequentially strong-dependent. If two tasks satisfy at least one of the two cases, they can not be executed in parallel, even if they are assigned to separate processors. Among N_{lc} linear clusters generated by algorithm *Linear-Cluster*, we can first reduce the number of linear clusters by $\sum_{i=1}^{N_{msd}} (|D_i| - 1)$ by merging mutually strong-dependent linear clusters. It generates a new set of linear clusters. Then we can further reduce the number of linear clusters by $\sum_{i=1}^{m} (|S_i| - 1)$ by merging sequentially strong-dependent linear clusters. The order of merging by mutual strong-dependency and sequential strong-dependency is not important, as proved below in Lemma 4.2.5.

Now that we have merged all the linear clusters which can not be executed in parallel at any time during execution, we can maximize parallelism if we assign the remaining clusters to separate processors. Consequently, the sufficient number of processors is $(N_{lc} - \sum_{i=1}^{N_{msd}} (|D_i| - 1) - \sum_{i=1}^{N_{ssd}} (|S_i| - 1))$.

□

Next, Lemma 4.2.2 gives the sufficient number of processors to guarantee that the total execution time of a computation graph is not longer than the execution time of the critical path length of the graph. In the lemma, we assume an ideal multiprocessor system in which there is no communication overhead other than the actual transmission overhead of messages. To be specific, it is assumed that every message is transmitted to its destination without any queuing delay on communication links.

Lemma 4.2.2: Assume that there are no queuing delays on communication links. It is sufficient (but may not be necessary) to utilize $(N_{lc} - \sum_{i=1}^{N_{msd}} (|D_i| - 1) - \sum_{i=1}^{N_{ssd}} (|S_i| - 1))$ processors to execute a computation graph on a given physical architecture graph, which guarantees that the total execution time is at most the critical path length of the computation graph.

Proof: First assume that we are given a virtual architecture which has exactly the same configuration as a given computation graph. Then there exists a one-to-one onto mapping between the edges and nodes in the two graphs. Suppose that the actual execution time of the computation graph can be longer than the critical path length. This can only happen if a task on the critical path is forced to be idle until its data dependency on its direct ancestor(s) other than on the direct parent task on the critical path are satisfied. Now, we will show that this can not occur.

Let SP_N be the connected subpath of the critical path from a root node to node N in Fig. 4-5. For each node N on the critical path, which has more than one path from a root node, we need to show that the length of subpath SP_N should never be longer than other subpaths.

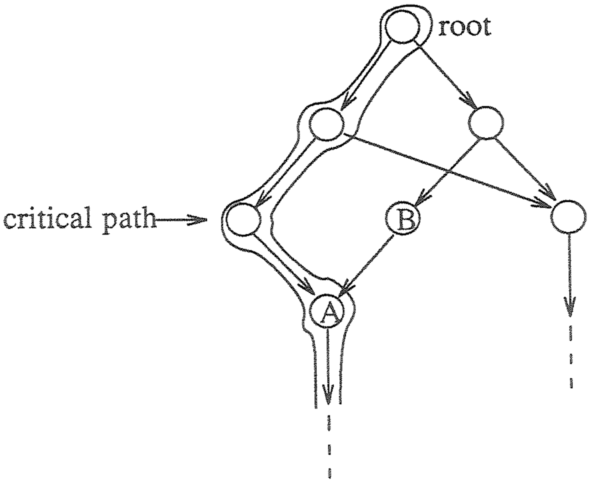


Figure 4-5 Computation Graph

For ease of exposition, assume that node B delays the start of execution of node A in Fig. 4-5. This implies that there exists an alternative path from the root node to node A through node B whose length is longer than that of the subpath SP_A. If so, the alternative path should be a part of the critical path. No matter which path is to be taken from the root to node A, the rest of the critical path is identical from node A to a leaf node. This contradicts the assumption that subpath SP_A is a part of the critical path. The above argument holds for any node on the critical path.

Therefore, there is not any single task on the critical path which is forced into being idle by other tasks on non-critical paths.

In reality, node A in Fig. 4-5 may be forced into waiting for data from node B. Because all the tasks on the critical path are to be assigned to the same processor, their intertask communication overheads can be ignored. The actual waiting time depends on linear clustering of the computation graph; however, regardless of the linear clustering, the waiting time of each task will not be longer than the length of SP_A minus the actual execution time of SP_A (when intertask communication overhead ignored). Consequently, the total execution time will not be longer than the critical path length of the original computation graph.

Furthermore, we have shown in Lemma 4.2.1 that we can partition the graph into $(k - \sum_{i=1}^{n} (|D_i| - 1) - \sum_{i=1}^{m} (|S_i| - 1))$ clusters without affecting the length of the critical path. Therefore, we just need $(k - \sum_{i=1}^{n} (|D_i| - 1) - \sum_{i=1}^{m} (|S_i| - 1))$ processors.

□

We mentioned above that two or more linear clusters may be merged into one cluster, if they are mutually strong-dependent or sequentially strong-dependent. The following three lemmas are concerned with the order of merging of linear clusters.

Lemma 4.2.3: When a linear cluster is mutually strong-dependent with more than two linear clusters, the order of merging is important.

Proof: For three linear clusters K_1, K_2 and K_3, suppose that K_1 is mutually strong-dependent with K_2 as well as with K_3. Then $L_1 \cap L_2 = \emptyset$ and $L_1 \cap L_3 = \emptyset$. Unless $L_2 \cap L_3 = \emptyset$, after L_1 is merged with one of the others (say, L_2), $(L_1 \cup L_2) \cap L_3 \neq \emptyset$. Conversely, if L_1 is first merged with L_3, then $(L_1 \cup L_3) \cap L_2 \neq \emptyset$. Hence, the order of merging is important.

□

Lemma 4.2.4: When a linear cluster is sequentially strong-dependent with more than two linear clusters, the order of merging is important.

Proof: The proof is similar to that for Lemma 4.2.3.

□

Lemma 4.2.5: Assume that linear cluster K_1 is mutually strong-dependent on linear cluster K_2 but not on K_3, and also sequentially strong-dependent on K_3 but not with K_2. Then the order of merging of K_1 with K_2 and K_3 is not important.

Proof: Since K_1 is mutually strong-dependent on K_2 and sequentially strong-dependent on K_3 simultaneously, $L_2 \cap L_3 = \emptyset$. After first merging K_1 with K_2 into cluster K_{12}, K_3 is still strongly in linear order of execution with K_{12} since the trailer node K_1 remains as direct ancestor of the header node of K_3. Similarly, after merging K_1 and K_3 into K_{13}, K_2 is still mutually strong-dependent on K_{13}. Hence, the merging order is not important.

□

As a result, if a linear cluster is mutually strong-dependent or sequentially strong-dependent (but not both) on more than one linear cluster, we should determine the order of merging based on the scheduling objectives. On the other hand, we may disregard the order when we merge linear clusters, one of which is mutually strong-dependent on one cluster and sequentially strong-dependent on the other cluster.

4.3. Iterative Refinement of Linear Cluster

In the previous sections, we discussed how to transform a computation graph G into a virtual architecture graph by linear clustering and merging. It is expected that linear cluster consisting of schedulable units of computation on the critical path of G takes the longest time to finish in the VAG in most cases. In this case, we can make use of the VAG for the mapping onto a physical architecture graph. Otherwise, we may need to identify better (linear) clustering by iterative refinement of linear clusters in the VAG. In this section, we propose new algorithms for iterative transformations of the *VAG* into another *VAG's* so that we can further reduce the total length of schedule prior to mapping. It consists of two steps:

- Linear cluster labeling;
- Linear cluster refinement.

Algorithm *LinearClusterLabeling* labels edges in a computation graph $G = (N, E)$. The level number $level_{edge}$ of edge $e_{ij} = (n_i, n_j)$ may be defined as follows:

$$level_{edge}(e_{ij}) = \omega \cdot comp_j + (1-\omega) \cdot comm_{ij} + level_{node}(n_j),$$

where $level_{node}(n_j)$ is the level number of node n_j, $comp_j$ and $comm_{ij}$ are computation time of n_j and communication time from n_i to n_j, respectively, and ω is a normalization factor. Note that $level_{node}(n_j)$ is defined as $\max_{n_k \in D_j}(level_{edge}(e_{jk}))$ where D_j is a set of direct descendants of node n_j. These edge labels allow us to identify the longest path to be considered for the minimization of the total schedule length in a *VAG*.

```
LinearClusterLabeling ($N$, $level_{node}(N)$)

/* $N$ is a node. */
/* $level_{node}(N)$ is the level number of $N$. */
/* The level numbers of all nodes are initialized to zero. */
/* $\omega$ is a normalization factor. */

Begin

  For each direct ancestor $A$ of $N$,
    If nodes $A$ and $N$ are in the same linear cluster,
    Then
        $comm_{A,N} = 0$;
    $level_{edge}(e_{A,N}) = level_{node}(N) + \omega \cdot comp_N + (1-\omega) \cdot comm_{A,N}$;
    $level_{node}(A) = \max(level_{node}(A), level_{edge}(e_{A,N}))$;
```

```
        If all the edges to direct descendants of $A$ are labeled,
        Then
            LinearClusterLabeling($A$, $level_{node}(A)$);
    End For;

End LinearClusterLabeling.
```

After linear cluster labeling, we can determine if there are paths through a virtual architecture graph, each of whose length is longer than the total computation time of linear cluster corresponding to the critical path of the original computation graph G. If there are such paths, the following algorithm *LinearClusterRefinement* is invoked to further reduce the total schedule length through iterative refinements of linear clusters.

```
LinearClusterRefinement ($G$, $VAG$)

/* $G$ is the original computation graph. */
/* $VAG$ is a virtual architecture graph to be transformed. */
/* $|P|$ denotes the length of path $P$. */

Begin

    Do
        Let $\phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ represent a set of paths
            from the root node to the leaf node in $G$;
        Let $\phi_l$ $(1 \le l \le n)$ be the current longest path in $VAG$;
        Choose a cut edge $(n_{adj_1}, n_{adj_2})$ on $\phi_l$
            which minimizes $\max_{1 \le k \le n} \sum_{i,j \in \phi_k} (\omega \cdot comp_i + (1-\omega) \cdot comm_{ij})$,
            where node $n_j$ is a direct descendant of node $n_i$,
            by comparing all possible refinements of linear clusters
            after temporarily merging nodes $n_{adj_1}$ and $n_{adj_2}$ into one;
        Let $\phi_m$ $(1 \le m \le n)$ be the new longest path;
        If $|\phi_m| < |\phi_l|$,
        Then Do
```

```
            Modify $VAG$ based on the selected refinement;
            LinearClusterLabeling($n_{adj_2}$, $level_{node}(n_{adj_2})$);
        End;
    Until no more reduction in $\max_{1 \le k \le n}(|\phi_k|)$ is possible;
End LinearClusterRefinement.
```

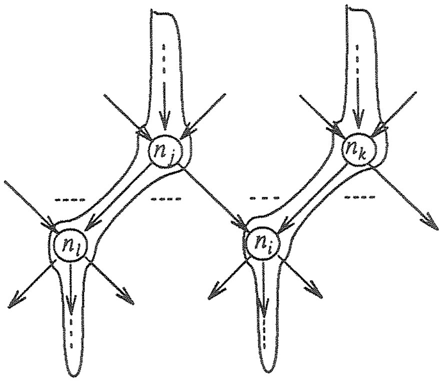
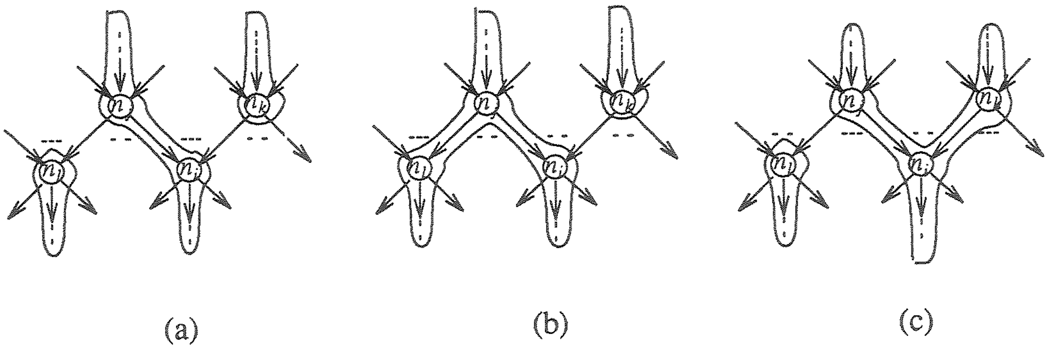


Figure 4-6 Linear Clusters



(a)

(b)

(c)

Figure 4-7 Possible Refinements of Linear Clusters

The basic idea of this algorithm is to locate a *cut* edge (n_j, n_i) on the longest path and to reduce the length by merging nodes n_i and n_j (belonging to separate linear clusters) into one. In Fig. 4-6, let us assume that the longest path is passing through nodes n_j and n_i, i.e., the longest path is $(\cdots, n_j, n_i, \cdots)$. We also assume that nodes n_j and n_i belong to different linear clusters in such a way that the former is in a linear cluster represented by path $(\cdots, n_j, n_l, \cdots)$ and the latter is in another linear cluster represented by path $(\cdots, n_k, n_i, \cdots)$. After the two nodes n_j and n_i are merged, linear clusters shown in Fig. 4-6 can be refined as shown in Fig. 4-7. In Fig. 4-7-a, we merge n_i and n_j into one cluster, and cut the edges like e_{jl} and e_{ki} so that all the clusters remain as linear clusters. In Fig. 4-7-b and Fig. 4-7-c, however, we merge them, but leave one of the edges uncut while we cut the other edge. This type of refinement may force us to sacrifice some potential parallelism since two or more nodes (e.g., n_l and n_i in Fig. 4-7-b) executable in parallel are to be assigned to the same cluster. Nonetheless, it is worthwhile to merge two linear clusters in this way if interprocessor communication overhead from n_j to n_l is larger than the extra computation time overhead caused by sequential execution of tasks (e.g., n_l and n_i in Fig. 4-7-b). Fig. 4-7-c shows another way to merging linear clusters.

As mentioned previously, linear clustering requires three algorithms: *LinearCluster*, *LinearClusterLabeling* and *LinearClusterRefinement*. For a given computation graph $G = (N, E)$ such that $|N| = n$ and $|E| = e$, the first algorithm takes $O(n^3)$ for the initial identification of linear clusters while the next one takes $O(e)$ since we need to visit each edge just once. The complexity of the third algorithm is $O(n \cdot e^3)$. It takes $O(n \cdot e^2)$ to find a cut edge connecting two adjacent nodes on a longest path which minimizes the schedule length after merging the nodes and refining linear clusters. The worst case occurs when we have to check all the edges in G for the refinement. As a result, the overall time complexity of linear clustering is $O(n \cdot e^3)$.

4.4. More Clarification On Linear Clustering

Up to this point, we have not imposed any restrictions other than acyclicity on computation graphs to which we have applied linear clustering. In order to generate an acyclic computation graph, each computation graph generated by an arbitrary program is supposed to be fully expanded whenever necessary. One of the important issues related to linear clustering which we need to scrutinize is how to keep computation graphs from being expanded to impractical size by unrolling do-loops in order to generate acyclic computation graphs. In this section, we investigate how to avoid full expansion of all existing do-loops to allow identification of linear clusters from computation graphs.

For a given cyclic computation graph, Martin and Estrin [MAR67] have proposed a couple of transformations of computation graphs, which replace cyclic graphs by mean-value equivalent acyclic ones. Their basic idea is, using the estimation of computation times and branching probabilities at each node, to remove back edges under certain restrictions after adjusting the total computation time of each node.

In this section, following a similar approach, we propose new schemes to contract cyclic graphs into cycle-free graphs in a straightforward manner, if they meet certain properties, based on the following ideas:

- Hierarchical Expansion;
- Overlapped Nodes and Overlapped Computation Graph;
- Preclustering.

Note that we are only interested in the expansion of nested loops which are executable in parallel. If a loop is not executable in parallel, we regard it as a single schedulable unit of computation.

4.4.1. Definitions

We first define some terminology used in this section. In general, each loop except the innermost one may contain one or more nested loops. The *depth* of a nested loop L is the number of outer loops in which it is embedded. By convention, the depth of the outermost one is 0. Next, we define a maximal computation graph. Assume that $G_i = (N_i, E_i)$ is a computation graph generated during the ith iteration of L, where $1 \leq i \leq l$ and l is the maximum iteration count of a nested loop L. A computation graph $G = (N, E)$ is called a *maximal computation graph*

(*MCG*), when $n \in N_i$ if and only if $n \in N$, and $e \in E_i$ if and only if $e \in E$ for all i. During every iteration, a nested loop may repeat either its maximal computation graph or an arbitrary subgraph of the maximal computation graph. During each iteration, it is assumed that the relative orderings of computation and communication requirements among nodes and edges, respectively, are maintained. For example, for two nodes A and B, if A requires more computation time than B for one iteration, it is supposed to do so for every iteration. A nested loop is called *regular* if there are no conditional exits from the loop, and there is one join point at the end of every iteration. A cyclic graph, representing a nested loop, is also called *regular* if it is regular. We may apply algorithm *LinearCluster* to a subgraph of a computation graph, independent of other portions of the graph. Linear clusters generated from the subgraph are called *preclusters*. Among these preclusters, the one which corresponds to the critical path of the subgraph is called the *Most Dominant Precluster* (*MDP*). A cyclic computation graph can be transformed into an acyclic graph by contracting each cyclic subgraph into a single node (called a *contracted node*). Once all cyclic subgraphs are replaced with single nodes, the graph is called a *contracted graph*. A critical path of a contracted graph is called the *global critical path*, and linear clusters generated from a contracted graph are called *global linear clusters*. Finally, a cyclic computation graph is called a *fully expanded graph* (*FEG*) if all the existing nested loops are unrolled.

4.4.2. Hierarchical Expansion

It is a primary goal of multiprocessor scheduling to fully utilize potential parallelism available in a given computation graph. Whenever there is more than one task which is executable in parallel, they should be assigned to separate processors. The number of linear clusters after applying algorithm *LinearCluster* to a fully expanded graph may be much larger than the number of available processors. Such a phenomenon occurs when the level of granularity of the minimal schedulable unit of computation is not appropriate to a target architecture. It may be an impractical attempt to assign a large number of tasks with fine granularity, such as one-statement tasks to separate processors. If that is the case, algorithm

MergeCluster will normally combine mutually and sequentially strong-dependent linear clusters into one until the number of linear clusters becomes not larger than the number of available processors. The basic motivation of hierarchical expansion is to determine the appropriate level of granularity in order to avoid subsequent merging of linear clusters after identifying linear clusters from a fully expanded graph. Sarkar and Hennessy [SAR86] have proposed graph expansion and internalization techniques to identify schedulable units of computation with the optimal granularity at compile-time. They utilized an objective function $F(\pi)$ for partition π, which expresses the trade-off between parallelism and communication overhead. In our approach, the number of available processors will play an important role in determining the level of granularity of schedulable units of computation.

In order not to generate an impractically large computation graph from an application program (before even trying to find linear clusters), we initially generate a computation graph based on coarse granularity; the subroutine level seems to be appropriate. Hierarchical expansion of a computation graph allows a contracted node to be expanded further into another computation graph with finer granularity and to be replaced by a set of new nodes. This expansion occurs only when a sufficient number of processors are available for the expanded nodes. Since a contracted node representing a nested loop may be expanded into a graph representing different iterations of the loop, we need a scheme to replace the expanded graph into another straightforward occurrence. It is difficult, however, to replace general cyclic computation graphs (for example, generated by pairs of *if* and *goto* statements) with equivalent acyclic computation graphs. We focus our attention on how to unroll regular nested do-loops.

A multiply-nested do-loop structure can be considered as an arbitrary combination of sequential code blocks (S) and parallel code blocks (P). Each embedded block can be defined recursively as another nested loops. Recursive expansion terminates when there are neither any available processors nor any embedded parallel code blocks. There are four types of relationships of a nested loop with its immediately nested loops: $SS-type$, $SP-type$, $PS-type$ and $PP-type$. By convention, $XY-type$ denotes that X and Y are the types of the outer and inner code blocks in a nested loop, respectively. We explain below the rules to unroll the outer loop for each type. By applying these rules to a nested loop recursively, we may unroll as many outer loops of a nested loop with arbitrary depth as needed.

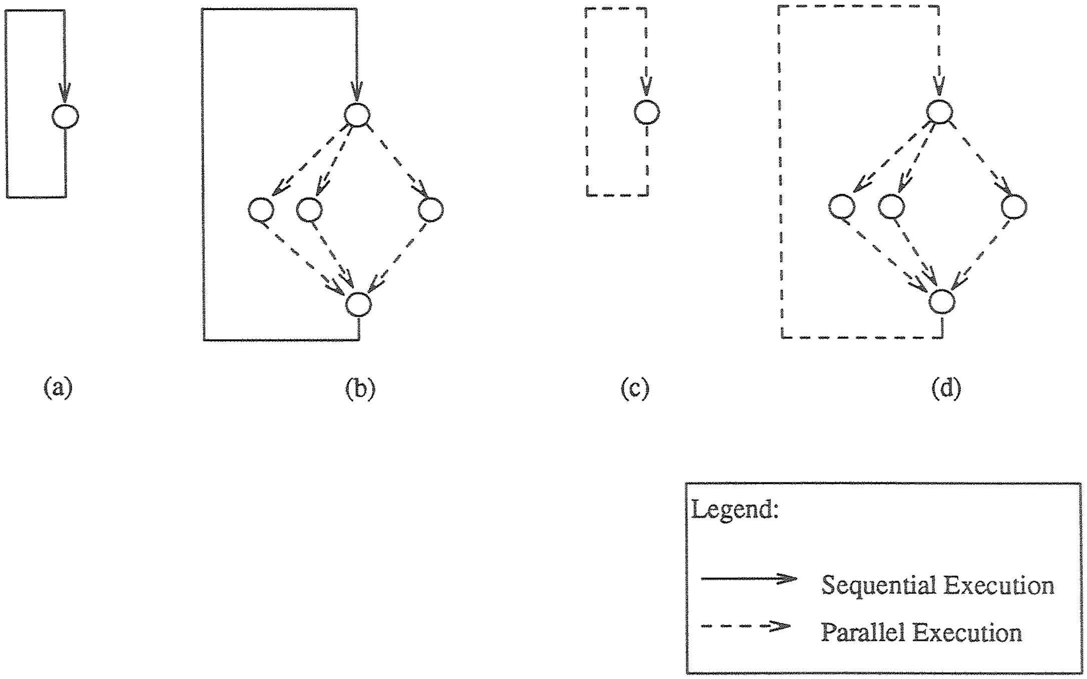


Figure 4-8 Types of Nested Loops

- *SS−type* : This type represents a sequential iteration of a sequential code block. Fig. 4-8-a shows a graphical representation of this type. Since there is no parallelism to be exploited, after merging all the sequential iterations of the inner sequential block into one node, we may remove the back edge from the graph.
- *SP−type* : This type represents a sequential iteration of parallel code blocks. There is a join point at the end of every iteration as shown in Fig. 4-8-b. Because of the join point at the end of each iteration, we can remove the back edge from the graph. We will explain in detail how to remove it in next section. This type may also represent parameterized invocation of inner block as described in Fig. 4-9.

- *PS−type* : This type represents a parallel execution of a sequential code block as shown in Fig. 4-8-c. It can be regarded as a parallel execution of mutually independent sequential code blocks with different parameters like loop indices. The outer loop of this kind of nested loop can be simply unrolled in such a way that sequential code blocks are assigned to separate nodes as long as there are processors available to be utilized.
- *PP−type* : This type represents a parallel execution of parallel code blocks as shown in Fig. 4-8-d. It corresponds to parallel execution of mutually independent do-loops. Depending on the number of available processors, we may unroll the outer loop by assigning independent do-loops or nodes in the do-loops to separate processors.

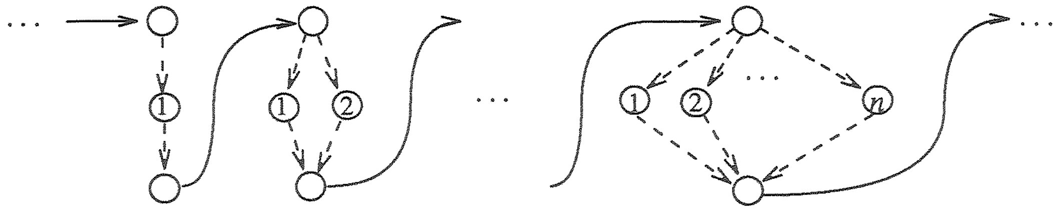
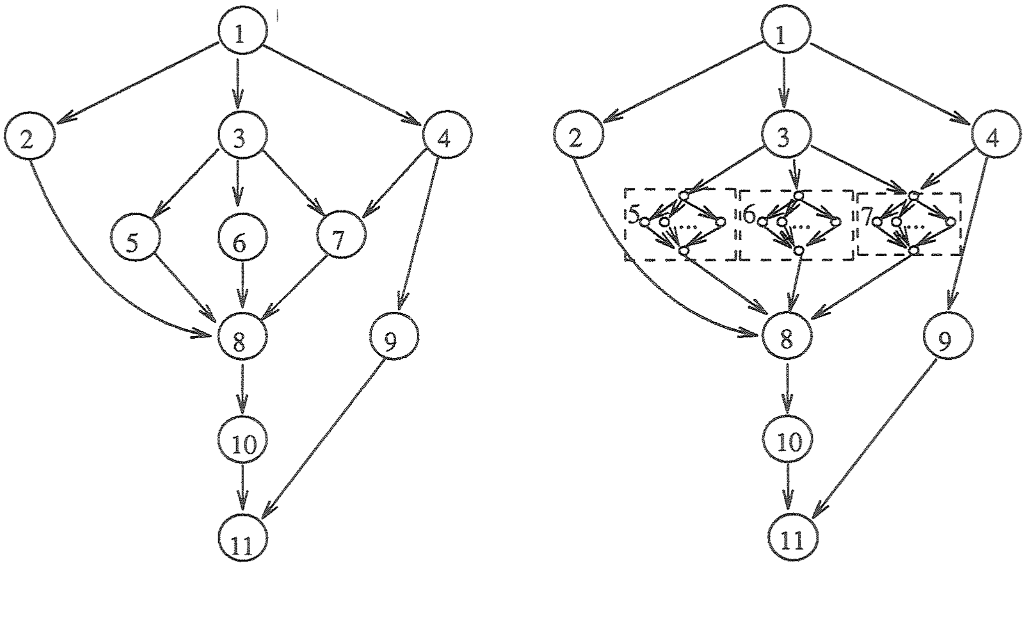


Figure 4-9 Parameterized Invocation of a Nested Loop

As a matter of fact, except for *SP−type* do-loops, we can remove back edges by coalescing all nodes into a single code block (*SS−type*), or by hierarchically expanding it (*PS−type* and *PP−type*) as long as the number of expanded nodes is not greater than the available number of processors.

We now discuss an observation related to hierarchical expansion. A node in a computation graph with coarse granularity may represent a nested loop as a whole; its computation time is the duration for executing all the iterations. On the other hand, when we consider a computation graph with finer granularity, each node may represent a statement, a set of statements or another nested loop which is expected to be executed during each iteration; its computation time is the duration for executing a single iteration of the loop. We observe that the critical path may vary as we unroll nested do-loops. In order to exemplify such a case, we consider the graphs shown in Fig. 4-10.

Depending on the characteristics of the loops (e.g., the number of iterations, the duration of one iteration, etc.), even though the total duration of one loop is longer than that of the others, the longest duration of its single iteration may be shorter than that of the others. In Fig. 4-10-a, we assume that the communication requirements of two paths (e.g., 1-3-5-8 and 1-3-6-8) are equal while the computation requirement of node 6 is larger than that of node 5. In addition, we assume that the critical path is (1-3-6-8-10-11) in Fig. 4-10-a. Fig. 4-10-b shows the computation graph after expanding nodes 5, 6, and 7 in Fig. 4-10-a. Suppose that one of the paths in the expanded graph of node 5 is longer than any path in the expanded graphs of node 6 and 7. Then the critical path becomes (1-5-8-10-11) in Fig. 4–10–b.



(a)

(b)

Figure 4-10 Computation Graphs with Coarse and Finer Granularity

As shown above, with nested do-loops being unrolled, we may get different critical paths. In order to make hierarchical expansion be applicable to loop unrolling, all edges and nodes except the node to be expanded on the critical path should remain as the subpath of the critical path even after expanding. the node. For example, if path (1-3-6-8-10-11) is the critical path in Fig. 4-10-a, it should remain as the critical path after node 6 is expanded. This leads to the *principle of global critical path*. The principle requires that any subpath of the global critical path in a contracted graph remain as a subpath of the critical path after expanding contracted nodes. Suppose that a cyclic computation graph satisfies the principle of global critical path. In order to identify proper linear clusters, we first generate the computation graph based on a coarse granularity. Then we identify linear clusters from the graph. After that, for each node having enough parallelism to be exploited, we may expand it into another graph if there are more processors available.

4.4.3. Overlapped Nodes and Overlapped Computation Graph

Each computation graph corresponding to an iteration of a nested loop can be regarded as a subgraph of its maximal computation graph. Some of nodes and edges in the maximal computation graph may be dummies in the sense that computations and communications represented by the nodes and edges in fact are not invoked during a particular iteration. The expanded graph could be represented by as many repetitions of the identical maximal computation graph as the iteration count. Furthermore, if it is regular, then it can be represented by a single maximal computation graph. Each node in the graph may represent one or more invocations of the same computation mainly with different loop indexes. Such a node is called an *overlapped node*. The computation and communication times of an overlapped node should be the sum of computation and communication times of the different invocations of the node during iteration.

These overlapped nodes make it possible to contract cyclic computation graphs to straightforward occurrences independent of the number of iterations. A computation graph which consists only of overlapped nodes is called an *overlapped computation graph* (*OCG*). The *OCG* in Fig. 4-11 corresponds to the *FEG* in Fig. 4-9. In the following *SP–type* nested do-loop:

```
For index1 = 1, n
  ParFor index2 = 1, index1
      .
      .
      .
  End ParFor;
End For;
```

an arbitrary node i will be executed $(n-i+1)$ times. Based on the number of iterations of each node, we can estimate the total computation and communication requirements of each node and contract the expanded graph to the overlapped computation graph as in Fig. 4-11. The only difference between the *OCG* and the *FEG* is that the latter represents the exact computation pattern of a nested loop while the former overlaps different invocations of the same computation onto one node because of the sequential nature of the execution order of the outer loop.

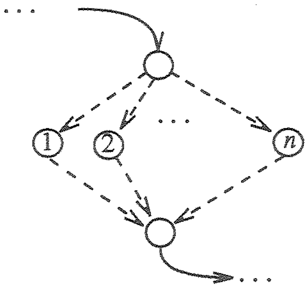


Figure 4-11 Overlapped Computation Graph

Lemma 4.4.1 claims that it is sufficient to replace a cyclic subgraph G_C = (N_C, E_C) representing a do-loop in a computation graph with an overlapped computation graph $G_O = (N_O, E_O)$ if G_C satisfies the following conditions:

- G_C represents a sequential execution of parallel code blocks (i.e., *SP−type* do-loop);
- G_C has a single join point at which all the parallel code blocks should be syn-

chronized at the end of every iteration;

- Any computation subgraph $G_{C_i} = (N_{C_i}, E_{C_i})$ which represents an iteration of the do-loop (represented by G_C) is isomorphic to a subgraph of G_O.

Lemma 4.4.1: Given a cyclic computation graph, which satisfies the previous conditions, linear clusters generated from a consecutive application of algorithms *LinearCluster* and *MergeCluster* to a *FEG* are identical to those generated from only application of algorithm *LinearCluster* to the *OCG* corresponding to the *FEG*.

Proof: We first apply algorithms *LinearCluster* and *MergeCluster* to the *FEG*. Since a cyclic graph to be expanded is regular, there always exists a join point at the end of each iteration where all tasks executable in parallel should be synchronized. Due to the existence of such join points in the *FEG*, the nodes corresponding to simultaneous invocations of a sequential code block with different do-loop indices are assigned to separate clusters by algorithm *LinearCluster*. Since the nodes representing the join points must belong to the critical path, the other nodes which are not on the critical path form different linear clusters. Moreover, based on do-loop indices, we can identify $(l-1)$ groups of sequentially strong-dependent linear clusters when l is the maximum iteration count. Then algorithm *MergeCluster* eventually combines all sequentially strong-dependent linear clusters into one cluster for each loop index.

We now apply algorithm *LinearCluster* to the *OCG*. Since the given computation graph is regular, the critical path of the *OCG* represents all the nodes in the critical path of the *FEG*. Similarly, other clusters also represent the groups of tasks which have the same do-loop index. These clusters trivially corresponds to those generated from the *FEG* by algorithm *MergeCluster*. Consequently, we have the identical set of linear clusters.

□

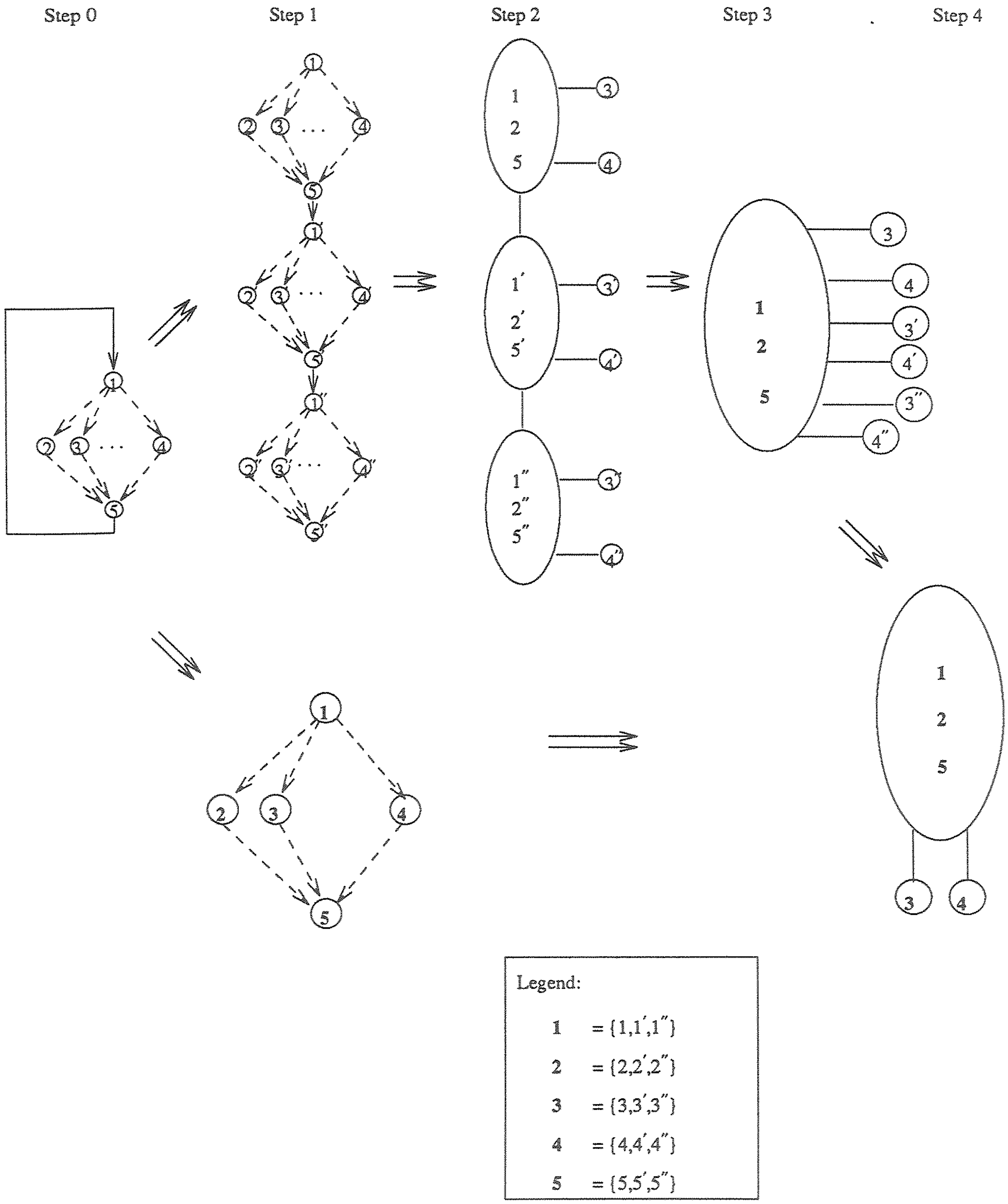


Figure 4-12 Transformations of an Expanded Computation Graph

Using Lemma 4.4.1, we can transform nested do-loops with arbitrary depth into an *OCG* by unrolling outer loops one at a time as long as they are regular. After unrolling the outermost do-loop, we may transform it into an *OCG* by Lemma 4.4.1, leaving the contracted nodes (representing embedded nested loops) intact. We keep unrolling the current outermost do-loop and transforming it into another *OCG* until we have enough processors to support the finer granularity of scheduling units of computation.

Fig. 4-12 describes how we generate the same virtual architecture graphs (shown at Step 4) from the *FEG* and the *OCG* (shown at Step 1) of a *SR−type* regular cyclic computation graph (shown at Step 0). In order to simplify the description, we limit the number of iterations and tasks executable in parallel during each iteration to be 3. We first assume that path (1-2-5) takes the longest time to complete during every iteration. Due to the principle of global critical path and the join points of the cyclic regular graph, we get the global critical path by first preclustering separately subgraphs corresponding to different invocations of a nested loop at Step 2 and then by coalescing the critical paths of each subgraph at Step 3. If we apply algorithm *LinearCluster* to the expanded graph, we may get directly the linear cluster graph shown at Step 3 (i.e., we may skip Step 2). After merging clusters representing the different invocations of the same computation, we can get the final linear clusters shown at Step 4. On the other hand, if we cluster the *OCG* at Step 1, we can directly generate the *VAG* shown at Step 4. Therefore, two different graph representations of a cyclic regular graph generate the identical virtual architecture graph.

4.4.4. Preclustering

A critical restriction to using an *OCG* is that each cyclic graph is required to be regular. There are many types of nested do-loops which can not be represented by cyclic regular graphs. The *preclustering* scheme previously used to precluster a subgraph of a computation graph can be adopted to prevent full expansion of cyclic subgraphs which are not regular but show uniform patterns of execution.

This scheme is useful if we can identify regular patterns of dependencies among nodes when expanding a contracted node representing a cyclic computation graph. For example, as can be seen in graph G shown in Fig. 6-5, the elimination of elements of row i of matrix A should be done in the order of $A_{i1}, A_{i2}, A_{i3}, \ldots,$ $A_{i(i-1)}$ because of sequencing constraints among the nodes. For such a graph, we can find linear clusters based on the constraints rather than unrolling do-loop. Such an attempt is called *preclustering*, since it is done only for the expanded nodes without considering the other nodes prior to identifying the global linear clusters. Interestingly enough, it usually corresponds to unrolling the outermost loop. Note that any *OCG* can be also preclustered if necessary.

A node in a preclustered (sub)graph is different from the other nodes in a computation graph which contains the subgraph. Each edge does not represent a sequential order of execution, but a communication path between clusters. As a result, each node in the preclustered subgraph may be assigned to a separate processor so that each computation can be done in parallel. That does not mean that all tasks in a preclustered node are independent of the other tasks in the other preclustered nodes, but that there exists at least one task in every preclustered task which is executable in parallel.

We are now concerned with how to identify linear clusters after a contracted node is expanded and clustered into a preclustered graph. All we need after preclustering the expanded nodes is to designate the *MDP* in the preclustered graph so that it can be included as a part of the global critical path which will be identified thereafter. For example, assume that a contracted node is on a global critical path before being expanded and preclustered. After preclustering the expanded nodes, some of the nodes may be excluded from the critical path. To be specific, the *MDP* of the subgraph should remain as a subpath of the critical path, while the other clusters should become separate linear clusters. The path corresponding to the *MDP* becomes a subpath of the global critical path. Therefore, there is no difficulty in identifying the critical path of the whole graph even if it is a mixture of computation nodes and linear clusters.

4.4.5. Linear Clustering of Cyclic Computation Graphs

We have defined and described new schemes to find linear clusters from cyclic computation graphs without fully expanding them. These are only applicable to computation graphs with cyclic regular subgraphs or to a do-loop which shows a uniform pattern of execution when being unrolled. The following algorithm *LinearClusterWithCycles* illustrates how to identify linear clusters from such computation graphs.

```
LinearClusterWithCycles ($G$, $K$)

/* $G$ is a cyclic computation graph. */
/* $K$ is a set of linear clusters. */

Begin

  Transform $G$ into contracted graph $G_C$;
  Identify global linear clusters from $G_C$;
  For each cluster in descending order of its path length
  Do
     If there exist enough parallelism
         which can be exploited within the cluster,
     Then
       Unroll each outermost loop by transforming
         each of embedded cyclic regular subgraphs into an OCG
         and then precluster it to generate a preclustered graph if necessary;
     Precluster other types of cyclic subgraphs directly into precluster graph
       by unrolling the outermost loop;
  Until all nodes are examined
     or the number of linear clusters generated becomes not less than
     the number of available processors whichever becomes true first;

End LinearClusterWithCycles.
```

The complexity of this algorithm is dependent on the level of granularity of schedulable unit of computation and the number of processors available. Let us assume that a cyclic computation graph $G = (N, E)$ is transformed into a cycle-free contracted graph $G_c = (N^0_{c,0}, E^0_{c,0})$, where $|N^0_{c,0}| = n_0$. The time complexity for linear clustering $G_{c,0}$ is $O(n_0^3)$. Next, a contracted node $N^i_{c,0}$ $(1 \leq i \leq n_1)$ in $N_{c,0}$ may be expanded into another cyclic graph $G^1_{c,i} = (N^1_{c,i}, E^1_{c,i})$ where $|N^1_{c,i}| = n_1$. Then the time complexity for preclustering $G^1_{c,i}$ is $O(n_1^3)$ for $1 \leq i \leq n_0$. Assuming at most k times of hierarchical expansion of each contracted node in G at the worst case, the total time complexity is:

$$O(n_0^3)+O(n_1^3)+\cdots+O(n_1^3)+\cdots+O(n_{k-1}^3)+\cdots+O(n_{k-1}^3) \leq O(n^3),$$

where $n = n_0 + n_0 \cdot n_1 + n_1 \cdot n_2 + \cdots + n_{k-2} \cdot n_{k-1}$. In fact, n represents the total number of nodes expanded after k-level hierarchical expansion of each contracted node in $G^0_{c,0}$. As a result, the total time complexity of algorithm *LinearClusterWithCycles* is $O(n^3)$, where n is the number of nodes after k-level hierarchical expansion.

CHAPTER 5

PHYSICAL MAPPING

The subject of this chapter is how to map a *virtual architecture graph* (*VAG*) onto a *physical architecture graph* (*PAG*). A *VAG* represents an imaginary multiprocessor system which guarantees the maximum parallelism available and reduces significantly the interprocessor communication overhead for a given computation graph, while a *PAG* represents a real multiprocessor system onto which the *VAG* is to be mapped. This mapping is called a *physical mapping* as it is the final mapping of a computation graph onto a real physical multiprocessor system.

A system is regarded as homogeneous if it consists of identical processors and communication links. Every processor in the system may share one global memory or may have local memory with the same capacity. However, each processor may have a different number of communication links. For example, the Intel Hypercube can be regarded as a homogeneous multiprocessor since every node consists of identical types of resources. Due to the characteristics of homogeneous systems we define here, the number of links is the only factor to be considered during homogeneous mapping. On the other hand, a heterogeneous system is composed of resources with different characteristics and capacities. Even though it has identical processors, it is regarded as a heterogeneous system if it has different sizes of local memories, different speeds of communication links, etc. For example, Cm* can be regarded as a heterogeneous multiprocessor since there are two types of communication links: local and global busses. The Intel Hypercube may become a heterogeneous system if each processor had a local memory with different capacity.

Both *VAG's* and *PAG's* are undirected graphs. For the matter of convenience of mappings, we ignore the directions of communications among linear clusters and physical communication links. The amount of communication between any pair of linear clusters is the sum of the amount of communications in both directions. All communication links are also assumed to be bidirectional.

It is well known that the optimal graph mapping problem is *NP−hard* [BRU74]. Even its sub-problems [ULL73] are *NP−complete*. Although it may not be possible to develop polynomial-time algorithms which are globally optimal unless $NP = P$, it is still feasible to develop polynomial-time algorithms which are globally near-optimal. There are two extreme approaches to the graph mapping problems. One of them is to find globally optimal mapping for some restricted cases. For example, Hu [HU61] proposed a polynomial-time, optimal scheduling algorithm which is applicable if every task has unit execution time, and a given computation graph is in the form of tree. Coffman and Graham [COF72] also found a polynomial-time, optimal scheduling algorithm under the assumption that each task has unit execution time and the number of available processors is two. On the other extreme, Pathak [PAT84] proposed a greedy algorithm which strictly relies on local information. Neither of these approaches is suitable for our purpose. The former can only be applied to such restricted cases that it is not applicable to any practical cases at all. On the other hand, the latter could find globally optimal mapping; however, in most cases, especially when there are many local optima, the result may be far from the optimal one.

The important goal of our proposed algorithms is to compromise between two extreme approaches by reducing the complexity of the mapping algorithms while sacrificing their optimality as little as possible. For physical mapping, we need to take into consideration as much global information as possible during mapping. In this chapter, we propose and justify various transformations of computation and architectures graphs, and heuristic algorithms to fulfill the goal.

5.1. Dominant Request Tree

The basic idea of our algorithm is to find subgraph isomorphisms from a *VAG* to a *PAG* which minimize the total execution time and satisfy given scheduling constraints. A subgraph $G_1 = (N_1, E_1)$ is said to be *isomorphic* to graph $G_2 =$

(N_2, E_2), provided there is a one-to-one, onto mapping $h: N_1 \rightarrow N_2$ such that $(u, v) \in E_1$ if and only if $(h(u), h(v)) \in E_2$. The subgraph isomorphism problem is "Given graphs G_1 and G_2, is G_1 isomorphic to some subgraph of G_2?" We can easily show that the subgraph isomorphism problem is *NP-complete*, making use of the fact that Undirected Hamilton Circuit is *NP-complete*. This fact forces us to rely on heuristics. We map each node of a *VAG* one by one in a sequential order. The key issue is then how to determine the mapping order which leads to the minimization of the total execution time. It is desirable to gather global information on a *VAG* to determine the order of nodes to be mapped. When a node has more than one adjacent node, the order selects the edge whose adjacency should be preserved. For this purpose, we propose another transformation of a *VAG* into a virtual architecture graph called a *Dominant Request Tree* (*DRT*). This transformation can be done independently of a target architecture (i.e., whether it is homogeneous or heterogeneous). A main purpose of this transformation is to gather global information which can be utilized during mapping.

A *DRT* is a maximal spanning tree of a *VAG*. We construct the *DRT* starting from a node called the *Most Dominant Node* (*MDN*) rather than starting from an arbitrary node in the *VAG*. The *MDN* is that node N which maximizes the cost function defined as:

$$\omega \cdot T_{comp} + (1-\omega) \cdot T_{comm},$$

where T_{comp} is the computation time of N, T_{comm} is the total communication time of N with its adjacent node, and ω is a normalization factor. The *MDN* is considered to be the most important node in the *VAG* in the sense that it represents a linear cluster which represents all tasks on the critical path in a given computation graph. It is usually the case that the *MDN* requires the largest amount of computation and communication among nodes in the *VAG*. As a result, we want to start mapping from the *MDN* so that it can be assigned to the most appropriate processor in a *PAG*.

Starting from the *MDN* as the root node of a *DRT*, we select next the node among its adjacent nodes which has the highest binding power. The binding power of node N_{adj} adjacent to node N is determined by:

$$\omega_1 \cdot T_{comp} + (1-\omega_1)\cdot(\omega_2 \cdot T_{comm} + (1-\omega_2)\cdot \sum T_{comp_{adj}}),$$

where T_{comp} is the computation time of N_{adj}, T_{comm} is the communication time of node N with node N_{adj}, and $T_{comm_{adj}}$ is the communication time of N_{adj} with one of its neighbors other than N. ω_1 and ω_2 are again normalization factors. We keep identifying another node with the highest binding power among unassigned nodes incident upon any node which has been already selected until all the nodes are selected.

A *DRT* of a *VAG* has two types of edges: the primary and the secondary edges. The former are edges belonging to the *DRT*, while the latter are edges belonging to the *VAG* but not to the *DRT*. While traversing a *VAG*, we can identify the *most dominant edge* of each node as the one which requires the largest amount of communication among all edges incident to the node. As a result, a *DRT* of a *VAG* always includes the *most dominant path* to a cluster among all the adjacent paths. It does not necessarily mean, however, that the amount of communication required by two clusters connected by a primary edge is always larger than that required by another two clusters connected by a secondary edge. By maintaining the adjacency of the primary edges during mapping, however, we can possibly minimize the total communication overhead.

Another important piece of information acquired after constructing a *DRT* is the *priority list* L. The order in which each cluster is included in the *DRT* determines the order of the actual mapping of linear clusters onto available processors. We need this sequential order to reduce the time complexity of our mapping algorithms. In order to get L, we rely on local information as well as global information. To be specific, when calculating the binding power of node N, we have considered two kinds of communication times: T_{comm} and $T_{comm_{adj}}$. The former provides us with local information. On the other hand, the latter supplies restricted global information. It is an estimation of the total communication overhead of an adjacent node with its adjacent nodes other than N. This global information is very helpful in the situation depicted in Fig. 5-1.

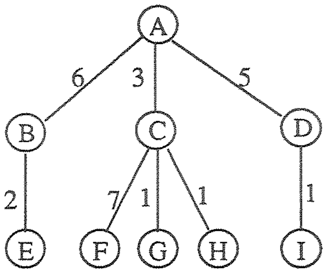


Figure 5-1 Dominant Request Tree

For example, assume that all the nodes have the same computation time, but different communication times as can be seen in Fig. 5-1. Then, node C has the highest binding power among the nodes adjacent to A, even if A has the smallest amount of communication with C. As a result, C will be assigned to a processor right after A. It will greatly increase the chance for node C to maintain the adjacencies with its direct descendants (especially, with F), and eventually decrease the total communication overhead. In this case, unless we take into account C's communication overhead with its descendants in calculating A's binding power with C, F is to be assigned to a processor only after all nodes B, C and D.

To conclude, a *DRT* plays an important role in mapping algorithms explained in the next sections. It provides us with global information on a computation graph. It determines a group of the edges whose adjacency should be satisfied to minimize the total communication overhead, and in what order. In addition, it makes it possible to reduce the complexity of mapping significantly, since we now focus on the mappings of the primary edges. Whenever it is not possible to maintain adjacency of the primary edges, we try to utilize the secondary edges.

5.2. Homogeneous Mapping

This section concerns physical mapping algorithms for homogeneous multiprocessor systems and related heuristics. It also discusses the time complexity of the algorithms. Since all resources in homogeneous systems are identical, there are no scheduling constraints to be applied during mapping. It is enough to attempt to find an optimal mapping which minimizes intercluster communication. For example, we do not have to worry about memory constraints for the mapping. Since each processor has the same amount of memory, any linear cluster can be assigned to any processor.

5.2.1. Homogeneous Mapping Algorithms

The main purpose of homogeneous mapping is to find a subgraph in a *PAG* to which a *DRT* of a *VAG* is isomorphic, relying on various heuristics like full-connectivity, exclusion, perturbation and foster mapping. In this section, we explain homogeneous mapping algorithms as well as the heuristics in detail. Each node of the *VAG* is assigned to a processor in the order determined during transforming the *VAG* into the *DRT*. The basic approach of the homogeneous mapping algorithm is to try to maintain adjacency of each node in the *DRT* with its neighbors as far as possible; whenever there is a direct primary edge from cluster C_1 to C_2, we choose processor P_{C_2} which has a direct link from P_{C_1}. Note that P_C denotes a processor that cluster C is to be mapped.

Algorithm *HomogeneousMapping* below summarizes our basic approach to homogeneous mapping.

```
HomogeneousMapping($DRT$, $PAG$)
Begin
  Perform the initial mapping of the root node of $DTR$ onto $PAG$;
  For each cluster $K$ (except the $MDN$) in the order of priority list $L$,
    ConnectivityMapping($K$, $DRT$, $PAG$);
    If unsuccessful,
    Then
      ExclusionMapping($K$, $DRT$, $PAG$);
    If unsuccessful,
    Then
```

```
        PerturbationMapping($K$, $DRT$, $PAG$);
      If unsuccessful,
      Then
        FosterMapping($K$, $DRT$, $PAG$);
      If unsuccessful,
      Then
        Allocate $K$ to any unassigned node
          with the most appropriate number of links among the nodes
          incident upon any of already assigned nodes;
    End For;
    Improvement($DRT$, $PAG$);

  End HomogeneousMapping.
```

At the beginning of mapping, the MDN is assigned to a processor which has the most appropriate number of communication links. Suppose that the MDN has e edges. If there are processors with e or more adjacent processors, we select any processor with the smallest number of neighbors. Otherwise, we select any processor with the largest number of adjacent nodes. If we have more than one candidate, we take into account the number of links required by the nodes adjacent to the MDN; we choose one with the most sufficient number of links for them. If more than one candidate is still available, we choose one arbitrarily.

We next explain four heuristics in the order they are applied during mapping. For each heuristic, we introduce the algorithm and a related description with examples. As can be seen in the preceding algorithm, we first attempt to maintain full-connectivity among clusters as summarized below.

```
ConnectivityMapping($K$, $DRT$, $PAG$)

/* $K$ is a cluster to be mapped. */

Begin

  Check if there are any free nodes adjacent to $P_{K_{da}}$ in $PAG$
```

```
        onto which $K$'s direct ancestor $K_{da}$ in $DRT$ has been assigned;
    For each free node adjacent to $P_{K_{da}}$,
        Check if there exist clusters $K_n$ in $DRT$
            which has direct path with $K$ as well as $K_{da}$;
        For each cluster $K_n$ in descending order of binding power with $K$,
            Find a node in $PAG$ which has direct links with $P_{K_{da}}$ and $P_{K_n}$;
    End For;

End ConnectivityMapping.
```

Suppose that cluster C is mapped onto a processor in a PAG and assume that direct ancestor C_{da} of C has already been assigned to $P_{C_{da}}$. We first consider the case that at least one processor adjacent to $P_{C_{da}}$ is free (i.e., has not been assigned to any cluster yet). Then, we check if C has an adjacent cluster B which has already been assigned to processor P_B in the PAG. If so, there are two possibilities. First of all, the three nodes form a fully-connected subgraph as below:

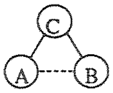


Figure 5-2 Fully-connected Subgraph

In Fig. 5-2, solid and dotted lines represent the primary and secondary edges, respectively. As can be seen in the figure, there exist primary edges between C and A as well as C and B. There also exists a direct secondary edge between A and B. If a cluster to be mapped is a member of a fully-connected subgraph such as C, P_c will be chosen from processors which also form a fully-connected subgraph with processors P_A and P_B.

For example, assume that we want to map cluster C after mapping cluster A, where C and A are connected by a primary edge as shown in Fig. 5-3. First we look for a node in the VAG which is adjacent to C and has been already assigned to a processor. In Fig. 5-3, B_1 and B_2 are such clusters, each of which forms a fully-connected subgraph with A and C. Assume that B_2 has not yet been assigned, while B_1 has been assigned to P_{B_1}. We then map C onto P_C, since it is the processor which forms a fully-connected subgraph with processors P_A and P_{B_1}.

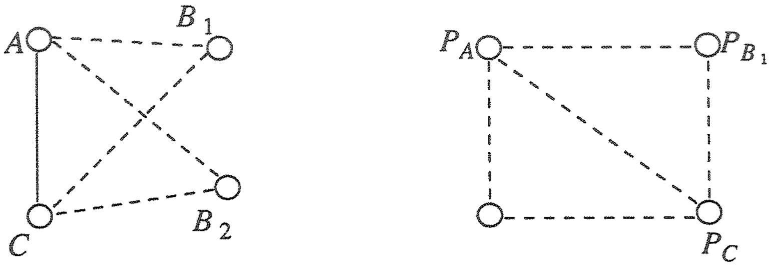


Figure 5-3 Mapping of Fully-connected Component

Secondly, C is adjacent to both A and B, but there is no direct edge between A and B. In this case, we just need to find a processor which is adjacent to both P_A and P_B. Even if we can find a cluster such as C, it may not be possible to locate a processor suitable for the cluster. If that is the case, we try to maintain the adjacency of the primary edge between A and C.

If cluster C fails to form full-connectivity with other clusters, we select one which has the most suitable number of links among the adjacent nodes of P_A, as we do for the initial mapping of the MDN. During the selection, we exclude processors which might be crucial for other clusters yet to be assigned as follows.

```
ExclusionMapping($K$, $DRT$, $PAG$)
/* $K$ is a cluster to be mapped. */
Begin
  Check if any nodes adjacent to $P_{K_{da}}$ are free;
  For each free node in a certain order,
    If the node is not critical to another unassigned cluster
    Then Do
      Allocate $K$ to it;
```

```
            Return;
        End Do;
    End For;

End ExclusionMapping.
```

For example, suppose that we have already allocated clusters A and B to processors P_A and P_B, respectively. In Fig. 5-4, there are three candidate processors (i.e., P_1, P_2 and P_3) to which cluster C may be assigned. We remove P_1 from the candidates, since it will become a crucial processor for the assignment of cluster D adjacent to both A and B.

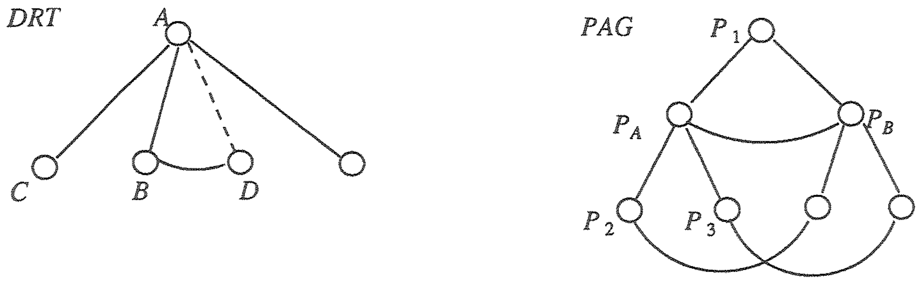


Figure 5-4 Example for Exclusion Mapping

Now we consider the case that we want to map cluster C whose direct ancestor in a DRT has been assigned to $P_{C_{da}}$, but there is no more free processor adjacent to $P_{C_{da}}$. Due to the lack of free adjacent processors, C must be mapped onto a processor which is not adjacent to $P_{C_{da}}$. For this case, we provide two heuristics: perturbation and foster mappings. In both heuristics, we first choose a processor which has the most appropriate number of links among currently unassigned processors. If there is more than one, we choose the one which is the nearest to its direct ancestor $P_{C_{da}}$. Those unassigned processors should be adjacent to at least one processor to which a cluster has already been assigned. We call the selected processor P_{any} for future reference.

In perturbation mapping, we attempt to preempt one of the adjacent nodes of $P_{C_{da}}$ which has been already assigned as follows:

```
PerturbationMapping($K$, $DRT$, $PAG$)
/* $K$ is a linear cluster to be mapped. */
Begin
  Choose processor $P_K$ with the most appropriate number of links
     among currently unassigned processors
     which are adjacent to any of the assigned ones;
  For each already assigned node $P_{K_n}$ adjacent to $P_{K_{da}}$
     in ascending order of the binding power of $K_n$ with $K_{da}$,
     If $K_n$ has no direct primary edge with $K_{da}$,
       Then Do
         Assign $K$ to $P_{K_n}$ and $K_n$ to $P_K$.
         Return;
     End Do;
  For each already assigned node $P_{K_n}$ adjacent to $P_{K_{da}}$
       in ascending order of binding power with $K_n$,
     Swap it temporarily with $P_K$;
     If it results in less communication overhead,
     Then Do
        Swap them permanently;
        Return;
     End Do;
  End For;
End PerturbationMapping.
```

There are two possible cases that a linear cluster can be preempted after being assigned to a processor. First, an adjacent processor (say, $P_{C_{adj}}$) of $P_{C_{da}}$ might be assigned to cluster C_{adj} which is not adjacent to cluster C_{da}. The other possible case is that all the clusters assigned to adjacent processors of $P_{C_{da}}$ are neighbors of C_{da}, but C_{adj} might have less communication overhead with C_{da} than C. If either of the above cases occurs, after swapping temporarily the current assignment (e.g., C to $P_{C_{adj}}$ and C_{adj} to P_{any}), we compare the total communication overheads between before and after swapping. If it successfully reduces the total communication overhead, we make it permanent. As can be seen in Fig. 5-5, assume that we have already assigned clusters A, B, C, E, and F to processors P_A, P_B, P_C, P_E and P_F in PAG, respectively. Since P_A has no more free adjacent processors by the time we need to map cluster D, we temporarily assign it to processor P_D in PAG. Then we try to swap D with one of clusters which have already been assigned to processors (adjacent to P_A). For example, it may be better to assign D to P_F and F to P_D, since D is a common neighbor of A and B as P_F is a common neighbor of P_A and P_B.

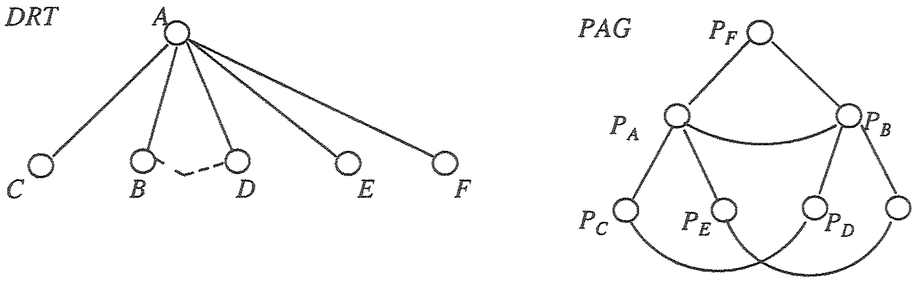


Figure 5-5 Example for Perturbation Mapping

This might be considered as a perturbation in a simulated annealing technique [KIR83]. The reason for the perturbation is to compensate for drawbacks of sequential mapping. As discussed previously, the mapping order of nodes in a VAG is determined largely based on local information: the closer a node is to the MDN, the better chance it has to be mapped onto an appropriate processor. Furthermore, the communication requirement of a secondary edge may be greater than that of a primary edge. As a result, we may get further reduction in the total communication overhead by swapping a node to be assigned yet with another node which has been already assigned to a processor.

As long as perturbation mapping does not make any improvement, it is not possible to maintain adjacency using a primary edge for this particular mapping. That is, cluster C can not communicate directly with cluster C_{da}. In order to lessen the effect of the indirect communication, we first check whether there is any cluster adjacent to C through the primary edge which has already been assigned to a processor. If there is more than one, we choose cluster C_{fa} which has the highest binding power (other than C_{da}) with cluster C. After assuming cluster C_{fa} as a direct ancestor of cluster C, we reiterate the same mapping procedure mentioned above (i.e., finding the best mapping from processor $P_{C_{fa}}$). We call such a mapping *foster mapping*. The only difference is that C_{fa} now becomes the direct ancestor of C for C_{da}.

```
FosterMapping($K$, $DRT$, $PAG$)
/*$K$ is a linear cluster to be mapped. */
Begin
  For each cluster $K_{adj}$ ($\neq K_{da}$) adjacent to $K$ in $DRT$
    in descending order of binding power with $K$,
    If $K_{adj}$ has already been assigned to $P_{K_{adj}}$,
    Then Do
      Find the best mapping $K$ from $P_{K_{adj}}$;
      Return;
    End Do;
  End For;
End FosterMapping.
```

If there does not exist such a primary edge, utilizing the secondary edges, we repeat the same procedure as we do for the primary edge. For example, in Fig. 5-6, we assume that the priority list L is (A, B, C, D, E). By the time we want to assign cluster E, there is no free neighbor of P_A. As a result, P_E is forced to be assigned to one of the free non-neighbor nodes. Suppose that cluster E has more binding power with C than with D. Then P_C acts like the original direct ancestor of E for P_A. Among the remaining four free processors, we choose P_E since it is a common neighbor of P_C and P_D as E is a common neighbor of C and D:

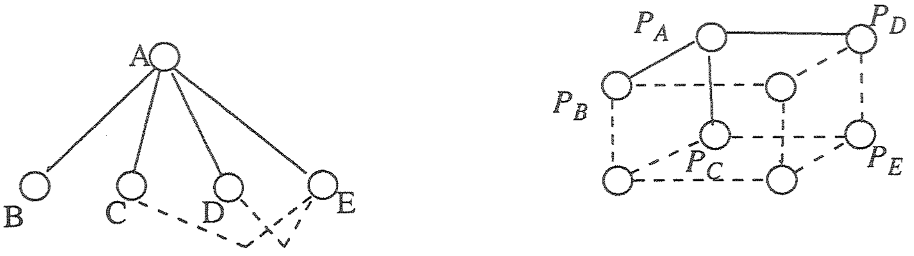
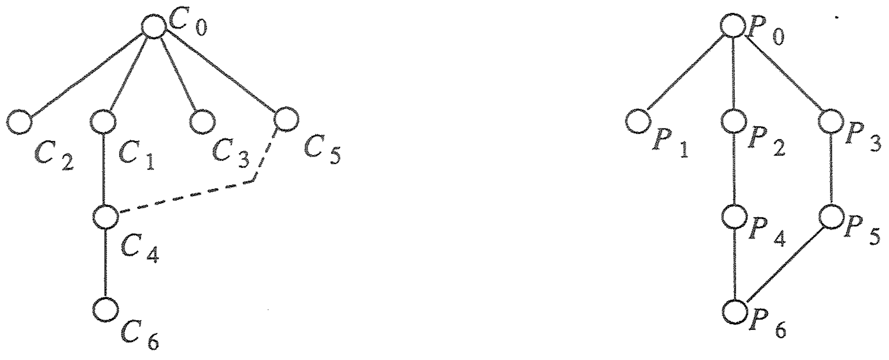


Figure 5-6 Example for Trivial Foster Mapping

On the other hand, it may not be always a good heuristic to utilize secondary edges during foster mapping. In the *VAG* shown in Fig. 5-6-a, suppose that the priority list L is $(C_0, \ldots, C_6)$. It is trivial to map clusters C_0, C_1, C_2, C_3 and C_4 onto processors P_0, P_1, P_2, P_3 and P_4, respectively. Next, we need to map cluster C_5. Since P_0 onto which C_5's direct ancestor C_0 has been mapped has no more free adjacent processors, we try to maintain the adjacency of C_5's other edge (i.e., the secondary edge between C_4 and C_5). Since C_4 has been already assigned to P_4, C_5 will be assigned to P_6. Depending on the amount of communication of C_5 with its neighbors, however, it may be better to map C_5 to P_5 than to P_6. For example, let us assume that the communication requirement between C_4 and C_5 is nominal comparing with the requirement between C_0 to C_5. Then although the latter is not adjacent to P_4, we can reduce communication overhead by reducing the number of hops (from 3 to 2) that a message has to travel between C_0 and C_5. Furthermore, if P_5 is assigned to C_5, then the primary edge from C_4 to C_6 can be maintained. During foster mapping, we need to take into consideration these factors, especially when a processor (e.g., P_6) to be mapped is crucial to another cluster yet to be mapped. If none of the above attempts succeeds, we simply return any unassigned processor which has the most appropriate number of links and is also adjacent to any already assigned processors.



(a) (b)

Figure 5-7 Example of Nontrivial Foster Mapping

Since the previous mapping algorithms do not guarantee an optimal solution, we try to further improve the result by applying restricted pairwise exchange similar to [BOK81] at the final step.

```
Improvement(DRT, PAG)

Begin

  Do
    For each candidate node which may be exchanged,
      Swap it temporarily with other candidate nodes;
      If the total communication overhead can be reduced,
      Then
        Swap them permanently;
    End For;
  Until no improvement is possible;

End Improvement.
```

We do not allow, however, random pairwise exchange of nodes. During mapping, we keep track of how each cluster has been assigned and each cluster is given a code according to the way it has been mapped as follows:

- *code* 0: assigned to the *MDN*;
- *code* 1: assigned to a node if it is a member of a fully-connected subgraph;
- *code* 2: assigned to a node if it is adjacent to more than one cluster;
- *code* 3: assigned to a node if it is assigned using a primary edge;
- *code* 4: assigned to a node if it is assigned in a way not mentioned above.

If code 4 is assigned to a cluster during mapping, it becomes a candidate for pairwise exchange. To be specific, it will be swapped with any free processor or with another processor onto which a cluster with code 4 has been assigned. This filtering allows us to obviate a lot of exchanges. This is quite a simple scheme, but it is expected to give a reasonably good improvement efficiently by throwing away a large unnecessary search space.

Finally, we discuss the time complexity of the mapping algorithms. We assume that l and p are the number of linear clusters derived from a given computation graph by linear clustering and merging, and the number of processors in a target multiprocessor system, respectively. The initial mapping takes $O(l)$, since we just need to traverse each node once in a *VAG*. During mapping steps, it takes the longest time to check full-connectivity among clusters during foster mapping, which takes $O(l^2 \cdot p \cdot (l \cdot p + l + 1))$. The final step takes $O(l^2)$ for pairwise exchange. As a result, the overall time complexity of homogeneous mapping algorithms is $O(l^3 \cdot p^2)$ in the worst case.

5.2.2. Discussion

Our algorithm relies on local as well as some limited global information. First, we look for a maximal spanning tree *DRT* for a *VAG*. Then the nodes in the spanning tree are mapped onto available processors in descending order of their binding powers. The sequential mapping order is expected to give us a sub-optimal mapping result without exhaustively testing all the possible combinations of mapping.

Comparing this new approach with the previous greedy algorithms [PAT84, STA84, VAN84], we expect the results of our mapping algorithms to be much better. Usually, the greedy approach first identifies a task which requires the largest amount of communication, called the heaviest communicator. After finding the most appropriate processor to which to assign the cluster in terms of the number of links, it assigns the neighbors of the cluster to its adjacent processors. The critical factor is that it favors the neighbors of the heaviest communicator not because they have more communication requirements than others but because they happen to be the neighbors of it. Its neighbors, on average, may have more communication requirements than others; however, it is not necessarily true for all of them. Furthermore, there may exist an intercluster communication path which does not belong to the longest path but its adjacency should be maintained for significant reduction of communication overhead. Since this path is considered only after the appropriate processors have been already assigned to its neighbors, it may be difficult to maintain its adjacency. On the other hand, our approach tries to take into account the global communication pattern and does not give any preference to those clusters adjacent to the heaviest communicator. Unfortunately, since linear clusters are to be mapped onto a *PAG* in a (predetermined) sequential order, the mapping algorithms may not be always globally optimal. The optimality of the algorithms may increase as we take into account more global information when constructing *DRT's*.

Finally, prior to the actual implementation of a parallel computation, we may identify which architecture is more appropriate to a given computation graph by applying the algorithm to a variety of multiprocessor systems. This will save us a lot of effort to develop application programs and choose appropriate architectures for the applications. For example, we can easily show that hypercube architecture is more appropriate to molecular dynamics code [EUB86] than hypertree architecture [KIM86b].

5.3. Heterogeneous Mapping

Heterogeneous mapping is a mapping of computation graphs onto architecture graphs which represent heterogeneous multiprocessor systems. In the following section, we characterize resources which are to be considered during heterogeneous mapping. Then we introduce the Dominant Service Tree (*DST*) concept and discuss various issues related to its generation. A *DST* is a subgraph of a *PAG*. Analogous to a dominant request tree, it provides global information on a heterogeneous multiprocessor system. To be specific, the tree implicitly specifies the relative order of computing power of processors and transmission speed of communication links. We then introduce and justify heterogeneous mapping algorithms based on tree-to-tree mapping.

5.3.1. Characterization of Resources

Resources to be considered during heterogeneous mapping are processor, communication link and memory. A processor can be characterized by a variety of parameters. There are basically two types of parameters: architecture and implementation parameters. Architecture parameters include data types (e.g., integer, real, etc.), operation types (e.g., register-to-register, memory-to-memory, etc.) and number of registers supported by a processor. In fact, those are visible to (low-level language) programmers. Implementation parameters specify how various components of a processor have been implemented in reality. A processor may include stack, pipeline or vector processor[†], interleaved memory, etc., in its implementation. Depending on these parameters, each processor may show different performances for a variety of application programs; a processor may be very effective for some applications while not efficient at all for the others. Instruction execution speeds may be used to measure the performance of a processor. Traditionally, it has been one of the most popular metrics for benchmark testing of processors. It may not be sufficient, however, to rely solely on instruction execution

[†] Sometimes, it may be considered as an architecture parameter as in CRAY systems.

speeds of processors to determine their performance.

In general, given an application program, the more parameters we take into account to characterize a processor, the more accurately we determine the effectiveness of the processor. On the other hand, it may be too complicated to consider all feasible parameters simultaneously during mapping. Furthermore, it may not be possible to construct a *DST* independently of the application program. Unless it is determined which task is to be assigned to which processor, it is hardly possible to figure out the performance of processors and in turn to generate a *DST*.

In order to determine a *DST* as little dependent on an application program as possible, we assume that the characteristics of the program are known a priori. We further assume that every submodule of the application program has the identical characteristics, regardless of the level of granularity of the submodules. These assumptions make it possible to determine the relative performance of processors for the application program. Each processor may be given a different *Processor Rating* (*PR*) based on its relative performance with respect to the application program. In our research, we assume that such ratings are also known a priori.

Secondly, a communication link can be characterized by communication latency and bandwidth. The former can be defined as the interval between the initiation of communication and the actual initial transmission. The latter is the maximum transmission rate which can be sustained by the processor and a communication link attached to it. In the case that the distances between processors are long enough to affect their interprocessor communication times, we consider *normalized bandwidth* (*NB*) defined as $k \cdot \frac{B}{D}$, where B and D represent bandwidth and distance, respectively. It can be also characterized by operational modes such as simplex, half duplex and full duplex. The communication overhead can be defined as $\omega \cdot T_{delay} + (1-\omega) \cdot \frac{T_{comm}}{NB}$, where ω is a normalization factor, and T_{delay} and T_{comm} represent communication latency and the amount of communication message to be transmitted, respectively.

Finally, memory can be characterized by memory access time and capacity, among other properties. Memory can be either local or global to a processor. We will consider only memory capacity to make the mapping problem simple. This resource is different from the other resources considered previously in that we will not take into consideration memory constraints when constructing a *DST*. This constraint is to be applied on the fly; i.e., a processor can be assigned to a task or not during mapping based on its memory capacity.

5.3.2. Dominant Service Tree

For heterogeneous mappings, it is important to utilize resources with high performance as far as possible so that the total execution time can be minimized and the workload balance can be achieved. We first need to distinguish resources with higher performance from those with lower performance. A *Dominant Service Tree* (*DST*) provides a limited amount of global information on resources in a heterogeneous multiprocessor lest our mapping algorithms become totally greedy based on local information. We can construct a *DST* by utilizing a maximal spanning tree algorithm. In fact, this may be considered as a transformation of a physical architecture graph into another physical architecture graph. In a sense, the transformation can be regarded as prescanning of architecture graphs prior to physical mapping. During the scanning, we collect information like which processors have more computing power and which communication links have more bandwidth than others.

It also represents connection patterns among those resources with higher performance, which implicitly specify how to utilize resources in what order (if possible). The relative locations of direct descendants of a node in a *DST* determine their overall ratings in the combined capacity of processors and communication links attached to them. For example, a node has usually more computing power than its descendants, and its leftmost direct child has the highest rating among direct descendants. Note that any scheduling constraints, including memory constraints, do not affect *DST* construction.

The construction of a *DST* starts from the *Most Dominant Node* (*MDN*) as the root node rather from an arbitrary node. It is that processor P which maximizes the cost function defined as:

$$\omega \cdot PR + (1-\omega) \cdot NR ,$$

where PR is the rating of processor P, NB is the sum of normalized bandwidths of communication links connected to P and ω is a normalization factor. Then we select next nodes according to their binding powers. The binding power of node P_{adj} from node P in the DST can be defined as follows:

$$\omega_1 \cdot PR + (1-\omega_1) \cdot (\omega_2 \cdot NB + (1-\omega_2) \cdot \sum_i NB_{adj_i}),$$

where PR is the rating of processor P_{adj}, NB is normalized bandwidth of processor P with P_{adj}, and NB_{abj} is normalized bandwidth of P_{adj} with its neighbor other than P. ω_1 and ω_2 are normalization factors. Using this function, we select next the processor node with the highest binding power among unassigned nodes which are adjacent to any assigned node, until all the nodes in a PAG are selected.

After the transformation of a PAG into a DST, the scheduling problems for heterogeneous multiprocessor systems turn into the tree-to-tree mapping problems. The edges in the PAG are to be divided into two different types: the primary and the secondary edges. Analogous to a DRT, the edges belonging to the DST are called the primary edges, while the edges belonging to the PAG but not to the DST are called the secondary edges. Note that the primary edges are essential even when only processors are heterogeneous. In that case, the primary edges identify which processors have more computing power than the others.

5.3.3. Overview of Heterogeneous Mapping

In this section, we overview the basic idea of heterogeneous mapping algorithms presented in Section 5.3.5. We first construct a DRT from a VAG based on the computation and communication requirements of each task. Similarly, from a given PAG, we construct a DST based on the number of links, their normalized bandwidth and processor rating. Then heterogeneous mapping can be considered as a tree-to-tree mapping from the DRT to the DST. The main goal of heterogeneous mapping is to identify the mapping which maintains adjacency of the primary edges of the VAG with those of the PAG. When there are no primary edges available, however, we utilize secondary edges of the *PAG* during mapping. Specific scheduling constraints are also to be applied during the mapping. Fig. 5-8 shows an overview of our approach to heterogeneous mappings.

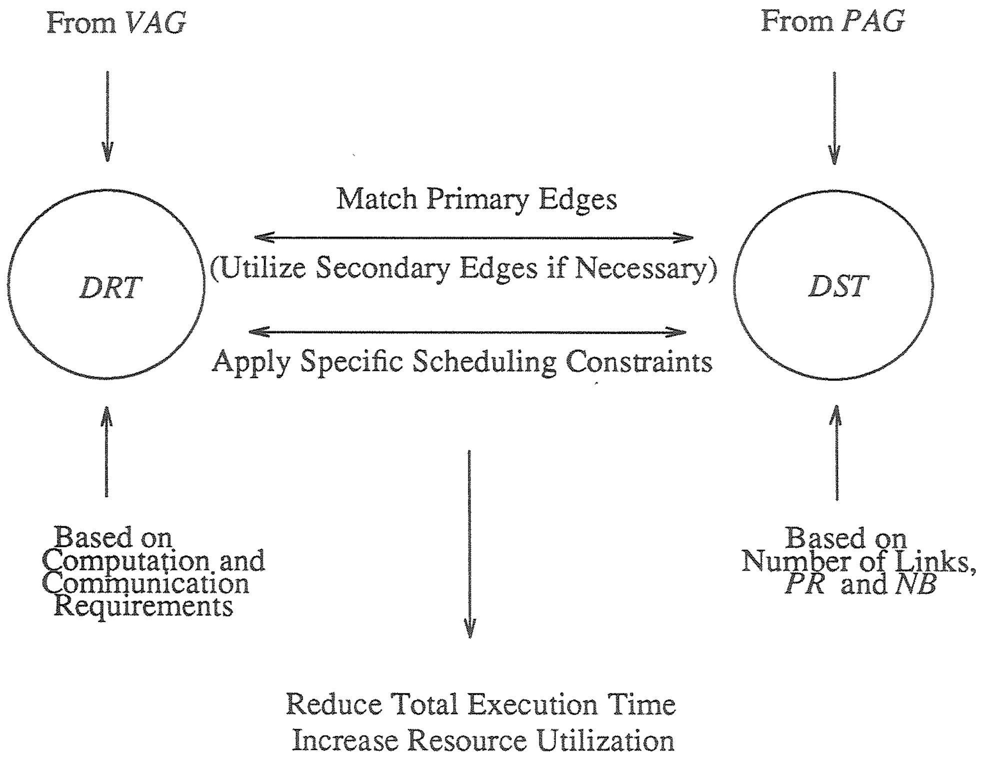

From *VAG*
From *PAG*
DRT
DST
Match Primary Edges
(Utilize Secondary Edges if Necessary)
Apply Specific Scheduling Constraints
Based on Computation and Communication Requirements
Based on Number of Links, *PR* and *NB*
Reduce Total Execution Time
Increase Resource Utilization

Figure 5-8 General Overview of Heterogeneous Mappings

The ideal goal of heterogeneous mapping is to find a subgraph of a *PAG* which has the identical topology with that of a *VAG*. Once we identify subgraphs with the identical topology of the *VAG*, we choose a matching which minimizes the following objective function:

$$\sum_{i=1}^{l}(\omega_1\cdot\frac{T_{comp_i}}{PR_i} + (1-\omega_1)\cdot(\sum_{j=1}^{e_i}(\omega_2\cdot T_{delay_j}+(1-\omega_2)\cdot\frac{T_{comm_j}}{NB_j}))),$$

where T_{comp_i} is the computation requirement of cluster i assigned to a processor whose rating is PR_i, and T_{comm_j} is the amount of communication of cluster i with one of its neighbors assigned to a communication link whose normalized bandwidth and communication delay are NB_j and T_{delay_j}, respectively. In the preceding function, we assume that there are l linear clusters to be mapped, and cluster i has e_i edges. In addition, the matching should satisfy all given scheduling constraints.

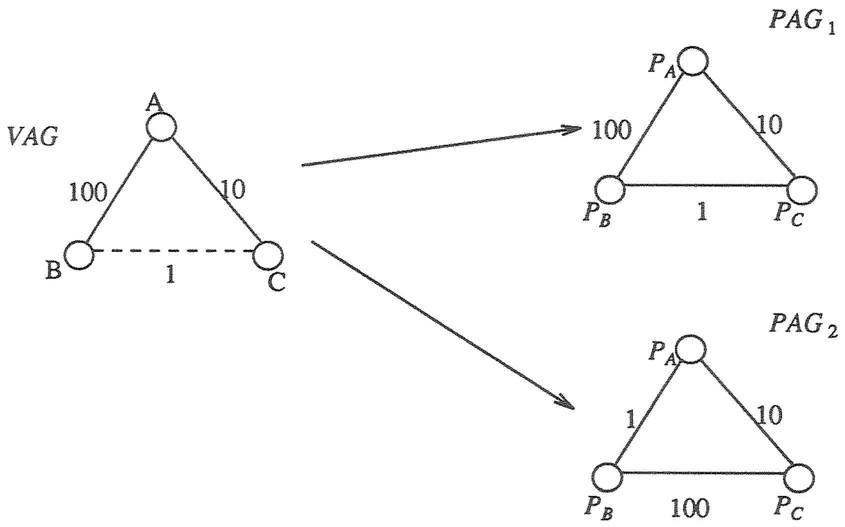


Figure 5-9 Two Mappings onto Identical Topology

Fig. 5-9 exemplifies the role of the objective function in a heterogeneous mapping. We assume that all the clusters in VAG have the same computation time, and all the processors in $PAG's$ have the same processor rating. Then we may let ω_1 be 0 without loss of generality in order to make a comparison between two different matchings. In Fig. 5-9, A, B, and C in VAG are mapped onto P_A, P_B, and P_C in $PAG's$, respectively. The numbers in VAG and $PAG's$ represent the amounts of communication to be transmitted between each pair of clusters and normalized bandwidths of communication links, respectively. Finally, let T_{delay} and ω_2 be 0 and 0.5, respectively. Then the total communication time is 3 when VAG is to be mapped onto PAG_1, while it is 101.001 when mapped onto PAG_2. One main cause for the huge difference in the total communication times is the workload imbalance of communication links in the second matching. The first mapping can balance workload by assigning the link with large bandwidth to the edge with large communication requirement.

5.3.4. Issues on Heterogeneous Mappings

In order to reduce the complexity of mapping, we propose the transformation of the graph-to-graph mapping problem to the tree-to-tree mapping problem. Unfortunately, it is still an *NP−complete* problem to find an optimal mapping from one tree to another. The issue here is how to develop efficient heuristic mapping algorithms between a *DRT* and a *DST*. In following sections, we explain how to determine the mapping order of nodes of a *DRT* and introduce so-called *node information* as a means to avoid exhaustive matching between two trees. The other issue with which we are concerned is how to take into account various scheduling constraints during mapping.

5.3.4.1. Mapping Order

We introduced in Section 5.3.3. objective function to minimize for an optimal mapping. One way to minimize the function is to order linear clusters in descending order of the sum of their computation and communication requirements and to order processors in descending order of their combined capacity of computation and communication links attached to them. Then match them according to their relative locations in the orders so that it minimizes the objective function.

There are two approaches to determine the mapping order. First, we may construct the priority list L in descending order of the binding power of each node alone in a *DRT*. That is exactly the same as the order we have used for homogeneous mappings. In order to show that this approach may not be suitable for a heterogeneous mapping, we consider the following scenario using Fig. 5-10. For two linear clusters L_1 and L_2 at the same level in *DRT*, we assume that the binding power of L_1 is greater than that of L_2 alone but less than the total binding power of L_2 and all its descendants.

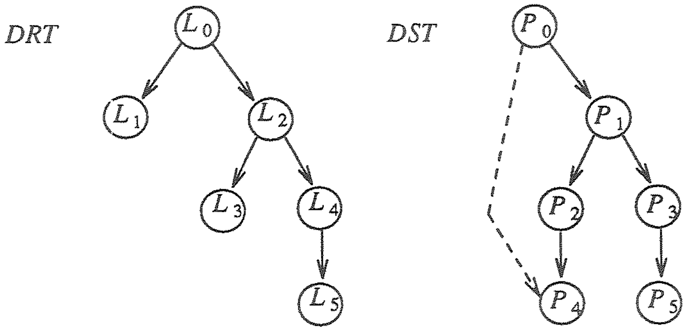


Figure 5-10 The Importance of Mapping Order

According to the first approach, L_1 will be mapped onto P_1; however, this is not a good choice. If L_2 were mapped onto P_1, even though the primary edge from L_0 to L_1 cannot be mapped onto a primary edge in *DST*, all the other primary edges in *DRT* would be mapped onto the primary edges in *DST*. If we follow the mapping order generated by this approach, it may be difficult to fully utilize the primary edges, since connection patterns of those edges may be ignored during mapping. In the case of homogeneous mappings, as there are only the primary edges in *PAG's*, homogeneous mapping algorithms will choose P_4 over P_1 since the algorithms may discover that P_1 is essential to L_2 and to later mappings of other nodes. On the other hand, in the case of heterogeneous mappings, since the primary edges are always first consumed whenever available, and P_1 is the only node connected to P_0 through a primary edge, L_1 will be assigned to P_1.

In order to avoid this situation, we propose a second approach to determine the mapping order in which the priority list L is constructed in descending order of the sum of the binding power of each node and all its descendants in a *DRT*. This order can be regarded as less greedy than the previous order, since it considers the binding power of its descendants as well as its own. It also gives us a better chance to utilize primary edges than the previous one. This is because the number of primary edges in a tree is expected to be approximately proportional to the weight of its root node. Since the only primary descendant of P_0 has been assigned to L_2, L_1 should be assigned to P_4 by following the secondary edge from P_0 to P_4. On the other hand, besides requiring extra overhead for generating a new mapping order, the second approach discriminates against the nodes which request more computation and/or communication for themselves than the others, but have few descendants. Note that it is still possible that L_1 is mapped onto P_1 if the binding power of L_1 itself is greater than the combined binding power of $L_2, \ldots, L_5$.

5.3.4.2. Node Information

It is evident that an optimal mapping can seldom be found if the mapping relies solely on local information. Nonetheless, it is also infeasible to match each node in the trees exhaustively to find a perfect match. The important issue for heterogeneous mappings is how to find an optimal mapping from a *DRT* to a *DST* without doing exhaustive matching. As a compromise, we propose *node information* associated with every node in a *DRT* and a *DST*. It is a septuple (*Depth, AvgDepth, Width, NoOfDirectChildren, NoOfIndirectChildren, AvgBranchFactor, AvgSiblingCnt*). This data provides us with clues to determine how a (sub)tree looks from its root node without traversing it completely; it will supply information useful for finding a perfect match between trees.

We explain here the definition of attributes of the septuple and their functions. *Depth* represents the longest path length from the root to any leaf node in a tree. It allows comparison of the heights of two trees. Similarly, *AvgDepth* represents the average path length from the root to all leaf nodes in the tree. *Width* of a (sub)tree is equivalent to the number of leaf nodes. It makes it possible to determine the width of the (sub)tree. Next, *NoOfDirectChildren* denotes the number of direct descendants of a node in a tree. Since our heterogeneous mapping algorithms are basically greedy, it is one of the important factors in maintaining adjacency of two nodes in a tree even after they are mapped onto the other tree. This attribute makes it easier to find a node in a *DST* which has a sufficient number of adjacent nodes for a node in a *DRT*. The fifth attribute of the septuple is *NoOfIndirectChildren*.

Using this attribute, we may estimate how many nodes other than direct descendants are in a tree.

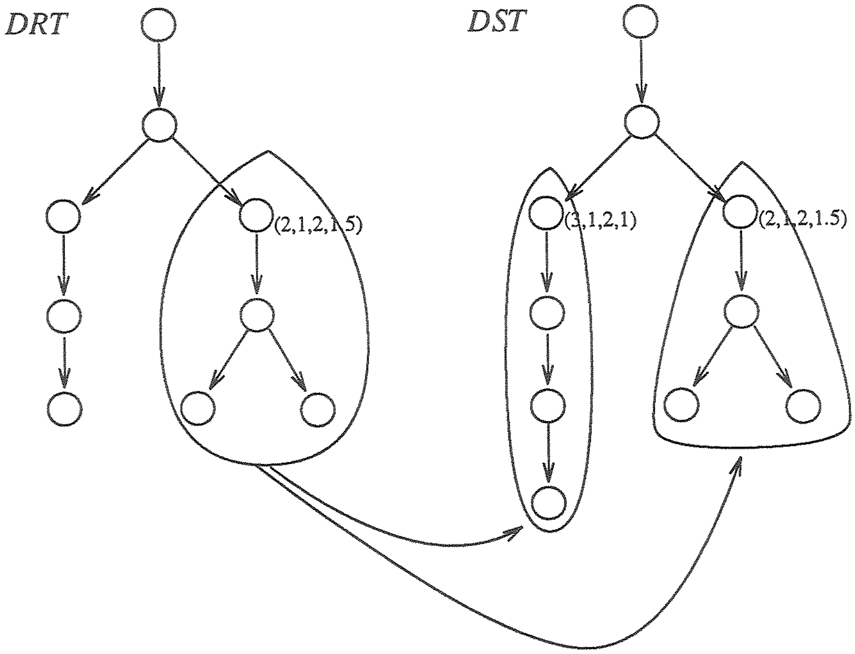


Figure 5-11 Mapping based on Node Information

It would be better if each node had information on all of its indirect descendants. Unfortunately, it would increase drastically the amount of node information to be provided as the depth of a tree increases. *AvgBranchFactor* of node N is defined as $\frac{(N_d + N_{ind})}{N_t}$, where N_d and N_{ind} denote the number of direct and indirect descendants, and N_t denotes the total number of nodes other than leaf nodes in a tree rooted at N. Even though two trees have exactly the same number of descendants, they may form completely different trees depending on how they are connected together. This information gives us the average number of direct descendants of the nodes in the tree. Finally, *AvgSiblingCnt* is defined as $\frac{(N_d + N_{ind})}{Depth}$. This information lets us know how many nodes are on each level on average.

Although two trees have the identical *AvgBranchFactor*, they may have different topologies. *AvgSiblingCnt* makes it possible to differentiate between them. If those attributes were unavailable, it would be difficult to figure out which subtree of a *DST* is better to match with a subtree of a *DRT*.

Assuming node information is a quadruple (*Depth, NoOfDirectChildren, NoOfIndirectChildren, AvgBranchFactor*), Fig. 5-11 shows an example to explain why we need the attributes like *AvgBranchFactor*. Without *AvgBranchFactor* in node information, it is not clear which subtree of the *DST* is better to match, even though the right subtree is the one to match.

5.3.4.3. Scheduling with Resource Constraints

As previously mentioned, if there are any scheduling constraints to be applied, they are considered during the physical mapping. This makes it possible to transform a *PAG* into a *DST* without caring which task will be assigned which processor. Prior to allocating a task to a processor, we check whether the processor satisfies the scheduling constraints of the task; that is, whether the processor has sufficient memory and disk space, a floating point accelerator, etc. This kind of scheduling is simple, but strictly greedy in the sense that the first processor satisfying the scheduling constraints of a task is assigned to the task, no matter how much more it would be efficient for or critical to other tasks which have not been assigned yet.

When being compared with the previous approaches [GAR75, CHU80], this scheme reduces the complexity of mapping significantly while sacrificing optimality. This approach is useful when it is expected that the majority of resources satisfy given scheduling constraints. The main advantage of this approach is to allow us to keep relying on the tree-to-tree mapping idea. On the other hand, a resource critical to a cluster may be preoccupied by another cluster to which the resource may not be essential. If that is the case, it may result in frequent preemptions and remappings of tasks which have been already assigned, due to the greediness of this approach. When only a few processors are expected to satisfy the constraints, it may be useful to preallocate tasks to these processors to prevent such preemptions during mapping.

5.3.5. Heterogeneous Mapping Algorithms

In this section, we describe heterogeneous mapping algorithms and related heuristics in detail. A heterogeneous mapping is basically a tree-to-tree mapping. Starting from initial mapping of the root node of a *DRT*, it allocates a node of the *DRT* onto a node of a *DST* in a serial order determined while building the *DRT*. Primarily, it attempts to maintain adjacency: first of primary edges and then of secondary edges, if necessary. If neither is possible, leaving the node unassigned, it attempts to map next unassigned node. Algorithm *PostMapping* maps those unassigned nodes at the end of mapping.

The following algorithm *HeterogeneousMapping* summarizes our approach to heterogeneous mapping algorithms.

```
**HeterogeneousMapping**($DRT$, $DST$)
Begin
  InitialMapping($DRT$, $DST$, $Assigned$);
  If $Assigned$ is $false$,
  Then
    return;
  For each node $N_{DRT}$ except root node of $DRT$ in predetermined order,
    Let $N_{DRT}^{DA}$ be a direct ancestor of $N_{DRT}$;
    Let $N_{DST}^{DA}$ be a node in $DST$
      on which $N_{DRT}^{DA}$ has been already mapped;
    FindPrimaryMatch($N_{DRT}$, $N_{DST}^{DA}$, $N_{DST}$, $Assigned$);
    If $Assigned$ is $false$,
    Then
      FindSecondaryMatch($N_{DRT}$, $N_{DST}^{DA}$, $N_{DST}$, $Assigned$);
  End For;
  PostMapping($DRT$, $DST$);
End HeterogeneousMapping.
```

Suppose that DST_{root} and DRT_{root} represent the *MDN's* of DST and DRT, respectively. If DST_{root} provides DRT_{root} with sufficient resources, then the initial mapping is very trivial; otherwise, we need to generate another DST again from any node which satisfies the scheduling constraints of DRT_{root} as explained in the next algorithm:

```
InitialMapping($DRT$, $DST$, $Assigned$)
Begin
  Check if $DST_{root}$ satisfies scheduling constraints of $DRT_{root}$;
  Set $Assigned$ to $true$;
  If satisfied,
    Then
      Return;
    Else  Do
      For each node $N$ in $DST$ in breadth-first search order,
        If $N$ satisfies scheduling constraints of $DRT_{root}$
        Then Do
          Set $N$ to $DST_{root}$ as the $MDN$;
          Build a new $DST$ starting from the new $DST_{root}$;
          Assign $DRT_{root}$ to $DST_{root}$;
          Return;
        End Do;
      Set $Assigned$ to $false$;
      Return;
    End Do;
End InitialMapping.
```

The unsuccessful initial mapping implies that a given computation can not be executed on the target architecture, since no processor satisfies the scheduling constraints required by the *MDN* of *DRT*. If that is not the case, we map the remaining nodes in the *DRT* one by one in the predetermined order as follows:

```
FindPrimaryMatch($N_{DRT}$, $N_{DST}^{DA}$, $N_{DST}$, $Assigned$)

/* $N_{DRT}$ is a cluster to be mapped. */
/* $N_{DST}$ is a processor onto which $N_{DRT}$ is to be mapped. */

Begin

  Set $Assigned$ to $false$;
  Set $\{N_{DST}\}$ to be the set of direct primary descendants of $N_{DST}^{DA}$
     which have not been assigned yet;
  Eliminate nodes from $\{N_{DST}\}$
     which do not satisfy scheduling constraints of $N_{DRT}$;
  If the set is empty,
  Then
    Return;
  FindMatch($N_{DRT}$, $\{N_{DST}\}$, $N_{DST}$);
  If the depth of the subtree rooted at $N_{DST}$ is not less than
     that of the subtree rooted at $N_{DRT}$,
  Then
     Return;
  Else
     CheckDescendants($N_{DRT}$, $N_{DST}$);
  Set $Assigned$ to $true$;
  Return;

End FindPrimaryMatch.
```

Let us assume that node N_{DRT} in a *DRT* is to be mapped onto one of the direct descendants of N_{DST}^{DA} in a *DST*. When more than one direct descendant satisfies scheduling constraints of N_{DRT}, we need to select one of them. Selection procedure is based on the node information of candidate nodes in the *DST* as summarized in algorithm *FindMatch*. Whenever we need to break tie, we make use of distance function *DF* defined as follows:

$$DF(\mathbf{s},\mathbf{r}) = \sqrt{\sum_{i=1}^{7} \omega_i \cdot (s_i - r_i)^2},$$

where $\mathbf{s} = (s_1, \ldots, s_7)$ and $\mathbf{r} = (r_1, \ldots, r_7)$ are the node information of one of the candidate nodes and N_{DRT}, respectively, and ω_i's are normalization factors as usual. In fact, distant function *DF* is defined as a yardstick to estimate how well a node in the *DRT* matches with a processor in *DST*.

```
FindMatch($N_{DRT}$, $\{N_{DST}\}$, $N_{DST}$)
Begin
  Choose nodes from $\{N_{DST}\}$ such that $s_i \geq r_i$ for all $i$ such that $1 \leq i \leq 7$,
    where $\mathbf{s} = (s_1, \ldots, s_7)$ and $\mathbf{r} = (r_1, \ldots, r_7)$ are
    the node information of nodes in $\{N_{DST}\}$ and $N_{DRT}$, respectively;
  If there is one,
  Then
    Return;
  If there is more than one such node,
  Then Do
    Choose one which minimizes distance function $DF$;
    Return;
  End Do;
  Choose and return a node from $\{N_{DST}\}$ which has the maximum number
    of attributes such that $s_i \geq r_i$ for $1 \leq i \leq 7$;
  If there is more than one,
  Then
    Choose one such that $s_1 \geq r_1$ and $s_3 \geq r_3$;
    If there is more than one,
        Then  Do
            Choose one which minimizes distance function $DF$;
            Return;
        End Do;
    Choose one which minimizes distance function $DF$;
    Return;
End FindMatch.
```

We now show how to map the *DRT* shown in Fig. 5-12 onto the *DST's* in Fig. 5-13 and Fig. 5-15. The mapping order of the *DRT* will be C_1, C_4, C_7, C_2, C_6, C_5 and C_3. Note that we omit the node information for leaf nodes as it is always (0, . . . , 0).

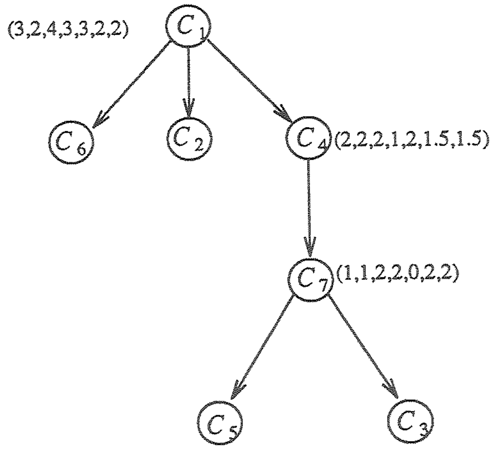


Figure 5-12 Dominant Request Tree with Node Information

The mapping from the *DRT* in Fig. 5-12 to the *DST* in Fig. 5-13 is trivial. Using only the primary edges, we can easily find appropriate matches between nodes in the *DRT* and those in the *DST*. Clusters C_1, C_4, C_7, C_2, C_6, C_5, C_3 are mapped onto nodes A, B, E, C, D, K, and L, respectively.

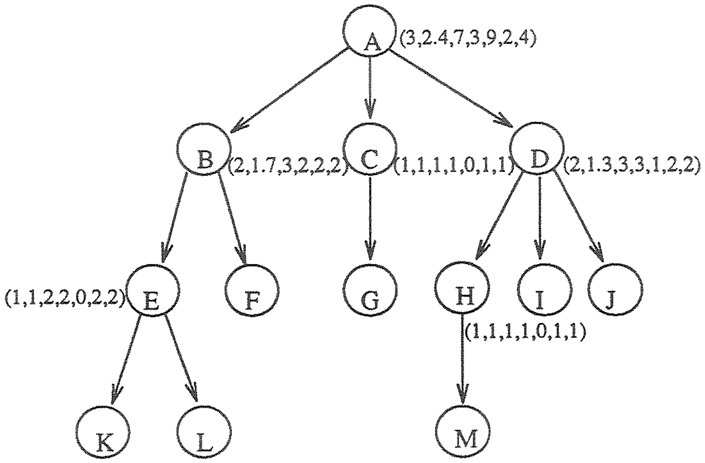


Figure 5-13 Dominant Service Tree with Node Information (Trivial Case)

When algorithm *FindPrimaryMatch* matches subtree S_{DRT} rooted at N_{DRT} to subtree S_{DST} with insufficient depth for node N_{DRT}, it may be beneficial to partition S_{DRT} into smaller subtrees in such a way that the depth of each subtree is always less than that of S_{DRT}. One of the subtrees will include N_{DRT}. Then we may find a better matching for those subtrees in the *DST*. The following algorithm *CheckDescendants* is invoked for the purpose.

```
CheckDescendants($N_{DRT}$, $N_{DST}$)

Begin
  Partition subtree $S_{DRT}$ rooted at $N_{DRT}$ into smaller subtrees in such a way
    that one of them ($S_{DRT_{root}}$) includes $N_{DRT}$ as its root node
    and their depths are less than that of $S_{DRT}$;
  If the depth of $S_{DRT_{root}}$ remains same,
  Then Do                          /* We cannot reduce the depth. */
    Assign $N_{DRT}$ to $N_{DST}$;
    Return;
  End
  Else Do
    Update the node information of $N_{DRT}$;
    FindPrimaryMatch($N_{DRT}$, $N_{DST}^{DA}$, $N_{DST}$, $Assigned$);
    If $Assigned$ is $false$,
    Then
      FindSecondaryMatch($N_{DRT}$, $N_{DST}^{DA}$, $N_{DST}$, $Assigned$);
  End
End CheckDescendants.
```

Fig. 5-14 shows a case where the depth of a subtree in DRT is longer than the depth of any subtree in DST. In that case, we consider the mapping of subtrees of DRT_1 onto subtrees of DST. Since its depth is reduced by one, we are now able to find a perfect match of the right subtree of DRT_1 onto DST_2. The root and the left subtree of DRT_1 are mapped onto DST_1.

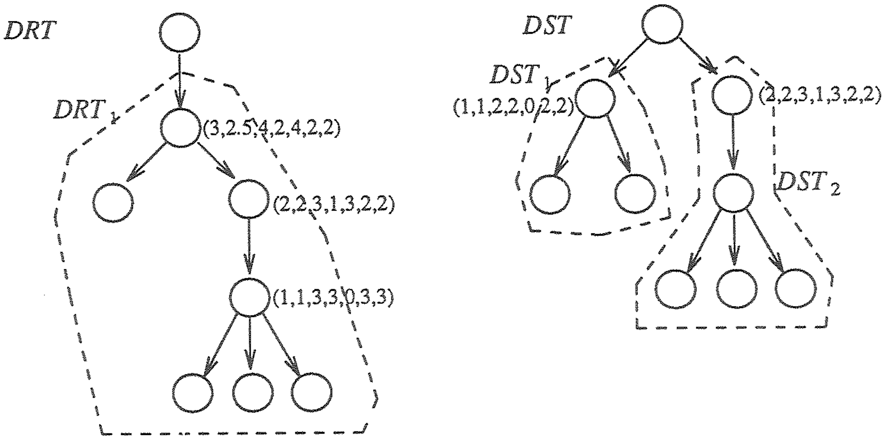


Figure 5-14 Subtree Matching

Algorithm *FindSecondaryMatch* is almost equivalent to algorithm *FindPrimaryMatch* except it attempts to map using the secondary edges. It also utilizes algorithm *FindMatch* to choose subtree S_{DST} rooted at N_{DST} which is expected to have the most similar topology of subtree S_{DRT} rooted at node N_{DRT}, and algorithm

CheckDescendants for better matching in the case where the depth of S_{DRT} is greater than that of S_{DST}.

```
FindSecondaryMatch($N_{DRT}$, $N_{DST}^{DA}$, $N_{DST}$, $Assigned$)

/* $N_{DRT}$ is a cluster to be mapped. */
/* $N_{DST}$ is a processor onto which $N_{DRT}$ is to be mapped. */

Begin
  Set $Assigned$ to $false$;
  Set $\{N_{DST}\}$ to be the set of direct secondary descendants of $N_{DST}^{DA}$
     which have not been assigned yet;
  Eliminate nodes from $\{N_{DST}\}$
     which do not satisfy scheduling constraints of $N_{DRT}$;
  If the set is empty,
  Then
     Return;
  FindMatch($N_{DRT}$, $\{N_{DST}\}$, $N_{DST}$);
  If the depth of the subtree rooted at $N_{DRT}$ is not less than
     that of the subtree rooted at $N_{DRT}$,
  Then
     Return;
  Else
     CheckDescendants($N_{DRT}$, $N_{DST}$);
  Set $Assigned$ to $true$;
  Return;

End FindSecondaryMatch.
```

We need *FindSecondaryMatch* for the mapping of the *DRT* in Fig. 5-12 to the *DST* in Fig. 5-15. For example, we can easily allocate clusters C_1, C_4, C_7 and C_2 to nodes 1, 2, 7, and 4, respectively, according to the mapping order. Then we simply allocate C_6 onto node 8, since there is only one secondary edge available from node 1 to node 8. For more sophisticated mapping, on the other hand, we may not assign C_6 to node 8. Taking into account the fact that C_7 assigned to node 7 has two direct descendants and the fact that node 3 is closer to node 1 than node 8, we may choose node 3 for C_7. Then we allocate clusters C_5 and C_3 to nodes 8 and 6, respectively.

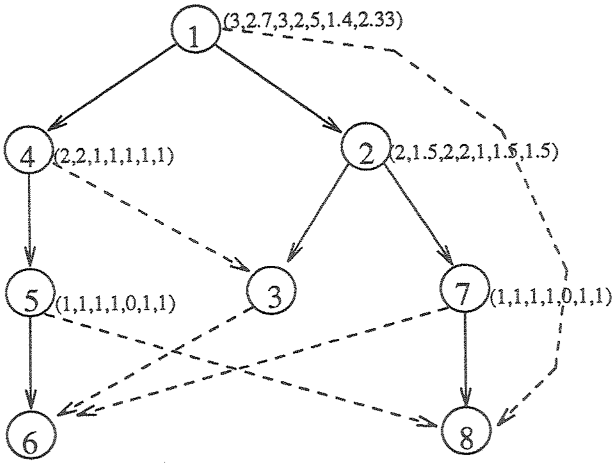


Figure 5-15 Dominant Service Tree with Node Information (Nontrivial Case)

As mentioned before, when we fail to assign a node to a processor because either no more primary or secondary edges are left or scheduling constraints cannot be satisfied, we leave it unassigned. Algorithm *PostMapping* takes care of those unassigned nodes and does pairwise exchanges, if necessary. This delayed matching of some nodes in a *DRT* is quite different from the approach taken for homogeneous mappings, in which the mapping order is strictly enforced. Due to the characteristics of heterogeneous mapping based on tree-to-tree matching, node N_{DRT} is mapped onto node N_{DST}, not only because the latter satisfies scheduling constraints of the former but also because subtrees S_{DRT} rooted at N_{DRT} and S_{DST} rooted at N_{DST} are expected to have a similar topology. As a result, it is very important to match nodes belonging to one subtree to nodes belonging to the other subtree as far as possible. If one of the nodes belonging to S_{DST} were assigned to a node not belonging to S_{DRT}, it may totally mix up the later mapping result. In order to avoid the problem, we temporarily suspend the mapping of such nodes until we complete mapping nodes of the *DRT* onto the *DST* using the primary edges as well as the secondary edges. Using a secondary edge for the mapping may cause a similar problem, but the effect will not be as severe as the previous case.

```
PostMapping($DRT, DST$)
Begin
  Set $\{N_{DRT}\}$ to be the set of nodes in $DRT$ which have not been assigned yet;
  Set $\{N_{DST}\}$ to be the set of nodes in $DST$ which have not been assigned yet;
  For each node $N_{DRT}$ in the predetermined order,
    Choose nodes from $\{N_{DST}\}$
      which satisfy scheduling constraints of $N_{DRT}$;
    If the set becomes empty,
    Then
      Swapping($N_{DRT}, DST$);
    Else
      Choose one which minimizes
        the total computation and communication times of $N_{DRT}$;
    Remove $N_{DST}$ from $\{N_{DST}\}$;
  End For;
End PostMapping.
```

Algorithm *Swapping* is invoked whenever we can not find an appropriate node among unassigned nodes in a *DST* which satisfies the scheduling constraints of node N_{DRT} to be mapped. If that is the case, we may find node $N_{DST'}$ which satisfies the scheduling constraints among the nodes in the *DST* which have been already assigned to some clusters in the *DRT*. In addition, we may find node N_{DST} among unassigned nodes in the *DST* which satisfies the scheduling constraints of cluster $N_{DRT'}$ currently assigned to $N_{DST'}$. Then we may assign N_{DRT} and $N_{DRT'}$ to

$N_{DST'}$ and N_{DST}, respectively. If there is more than one pair of candidates for the swapping, we choose the pair which minimizes the total computation and communication overhead after the swapping.

```
Swapping($N_{DRT}$)
/* $N_{DRT}$ is a cluster to be swapped. */
Begin
  Set {$N_{DST}$} to be the set of nodes among already assigned nodes in $DST$
     which satisfies scheduling constraints of $N_{DRT}$;
  If the set is empty,
  Then
     Return;                /* Scheduling is impossible */
  For each node $N_{DST'}$ in {$N_{DST}$} in the predetermined order,
     Set $N_{DRT'}$ to be the cluster assigned to $N_{DST'}$;
     Find node $N_{DST}$ among unassigned nodes
        which satisfies scheduling constraints of $N_{DRT'}$
        and minimizes the total overhead after being swapped with $N_{DST'}$;
     If found,
     Then Do
        Assign $N_{DRT}$ and $N_{DRT'}$ to $N_{DST'}$ and $N_{DST}$, respectively.
        Return;
     End Do;
  End For;
End Swapping.
```

We briefly discuss the time complexity of heterogeneous mapping algorithms. We assume that there are l linear clusters to be mapped onto p processors. First, the initial mapping takes $O(p^3)$ since we have to generate a new *DST* starting from every processor node in a *PAG* in the worst case. During both *FindPrimaryMatch* and *FindSecondaryMatch*, the worst case occurs when we need to recursively invoke *CheckDescendants*. The complexity of the matching step is $O(l^2 \cdot p \cdot (l \cdot p + 1))$. Finally, algorithm *PostMapping* takes at most $O(l \cdot p \cdot (l + 1))$.

5.3.6. Discussion

For heterogeneous multiprocessor scheduling, we transform the graph-to-graph mapping problem to the tree-to-tree mapping problem. The transformation is expected to reduce the mapping complexity significantly while sacrificing the optimality of the mapping as little as possible. We generate a *DST* based on the number of links, processor ratings and normalized bandwidth. As a result, it is critical to determine properly the processor ratings. In order to make mapping problem simple, it is assumed that the ratings are known a priori.

Heterogeneous mappings may not be so difficult as long as we can identify suitable *DST's*. However, when secondary edges or some scheduling constraints are to be considered during mapping, the result of mapping may be far from the optimal solution. Memory is the only special hardware to be considered during heterogeneous mapping. If there are only a few different capacities of memories available, it may be sufficient to take scheduling constraints into account on the fly during mapping.

CHAPTER 6

PERFORMANCE EVALUATION

The first goal of performance evaluation is to investigate how much further our scheduling algorithms can improve performance of *already-parallelized* computations, relying on less hardware resources if possible. The next goal is to test if they are applicable to a spectrum of architectures from loosely-coupled to tightly-coupled systems and from homogeneous to heterogeneous systems. Computation graphs may be regular or irregular[†], and cyclic or acyclic; they may represent numeric or nonnumeric computations, and the granularity of their nodes may be fine or coarse. The final goal is to check if the algorithms are also applicable to such a broad class of computation graphs.

The previous chapters introduced linear clustering and mapping algorithms as techniques for improving the performance of parallel computations. This chapter presents the performance results of those techniques as applied to a variety of computation graphs and architecture graphs. In the experiments are used three different applications: sieve of prime numbers, forward elimination of square matrices, and a synthetic program. The applications are chosen to illustrate different classes of computation graphs and to exemplify different problems associated with parallel computations. The experiments were performed on an Intel iPSC with 32 processors and on a Sequent Balance 21000 with 10 processors. Note that every measured time is the average of 20 repetitions of each application under consideration.

After summarizing our experimental environments, we explain the performance metrics being considered in the experiments. Then, we discuss the

[†] A computation graph is regarded as *regular* if the same pattern of computation is repeated at each level in a computation graph; however, it does not necessarily have the same number of nodes on each level. On the other hand, it is regarded as *irregular* if there exist any random patterns of computation at any level.

implementations and the results of performance evaluation of the three applications in detail to show the versatility of our scheduling algorithms.

6.1. Experimental Environments

As was mentioned in Chapter 2, there is a spectrum of multiprocessor architectures from loosely-coupled to tightly-coupled system. The Intel iPSC system is a loosely-coupled multiprocessor based on a binary n-cube network. The system has two components: the cube manager and the cube. The cube manager enables us to develop programs, to load them into the cube, and to get or release the cube and others [INT87]. The cube itself consists of 32 identical processors, each provided with 512K bytes of local memory and connected to 5 neighboring processors. Neither shared memory nor a global synchronization mechanism is available in this system. No variables are shared among processes even though they are on the same node. Therefore, both interprocessor and intraprocessor communications can only be achieved by message passing in any circumstances. Synchronization is subsumed by communication in the sense that a parallel computation assigned to a node may be triggered or resumed only after its necessary message (if any) arrives from the other node or the cube manager. Each node may contain up to eight point-to-point, asynchronous, bidirectional communication channels. The eighth channel is a global Ethernet channel that provides direct accesses to and from the cube manager. Each node may be assigned up to 20 processes, and, therefore, the number of processes involved in a single application program may be more than the number of processors in the system.

The Balance 21000 system is a tightly-coupled multiprocessor with 10 identical processors, each of which is connected with a single shared memory of 16M bytes through a common high-speed bus. Unlike other tightly-coupled systems (e.g. the Ultra, the Butterfly, and the RP3), there is no switching network which connects the processors and memory modules. The maximum number of processes which can be created simultaneously is $(n-1)$, where n is the number of processors which is currently on-line. Processor scheduling is done automatically by its operation system Dynix, which provides three types of scheduling algorithms

[SEQ86]: prescheduling, static scheduling and dynamic scheduling. No matter which scheduling algorithm we choose, though, we have no control over processor assignment; we can not assign a process to a specific processor. As a matter of fact, it is trivial to map linear clusters on a shared memory machine like the Balance, since it has nominal interprocessor communication overhead and acts as if it were a fully connected system.

6.2. Performance Metrics

Before presenting the results of the experiments, we introduce six performance metrics: total execution time, communication overhead, total throughput, processor utilization, theoretical communication time, and theoretical execution time. The first four of the metrics are actually measured either on the iPSC or on the Balance, while the rest of them are theoretically calculated to compare them with the measured performance. Once identifying the theoretical lower bounds, we can better evaluate performance of our scheduling algorithms.

The total execution time T_{exec} is simply the sum of the total computation time (T_{comp}) for computing schedulable units of computation on available processors in parallel and the total communication time (T_{comm}) for transferring the required messages among the processors. The communication overhead is the ratio of the communication time excluding pure message transmission time to the total execution time; i.e., $\frac{(T_{comm}-T_{theo.comm})}{T_{exec}}$. The total throughput is defined as the number of application programs completed during a given time span. The processor utilization is a measure of how efficiently each processor in a multiprocessor system has been used by a program assigned to it. It can be determined by $\frac{T_1}{p \times T_p}$, where T_i represents the total execution time to solve an application program on i processors.

According to empirical measurement performed on the iPSC [DUN86], the set-up time to acquire a channel is 860 μsec, while the transmission time to send one byte is 0.18 μsec. Those performance characteristics are used to calculate theoretical communication time $T_{theo.comm}$ (i.e., $(860 + 0.18 \times n)$ μsec), where n is the total number of bytes to be transferred along the critical path. Note that theoretical communication time only counts the pure message transmission time. Since the size of a single message can not be greater than 16K bytes in the iPSC, we considered communication delay per every 16K bytes when the message length is longer than 16K bytes. In order to obtain the theoretical execution time $T_{theo.exec}$, we run a program and measure its total execution time T_1 on the Balance using a single processor. Then, we divide T_1 by p, the number of processors which participated in parallel computation of the program.

6.3. Calculation of Prime Numbers

In this section, we describe a parallel algorithm implemented on the iPSC to find prime numbers and, then, provide the performance results with a related discussion. The computation graph shown in Fig. 6-1 describes how to calculate all the primes from 2 to N in parallel.

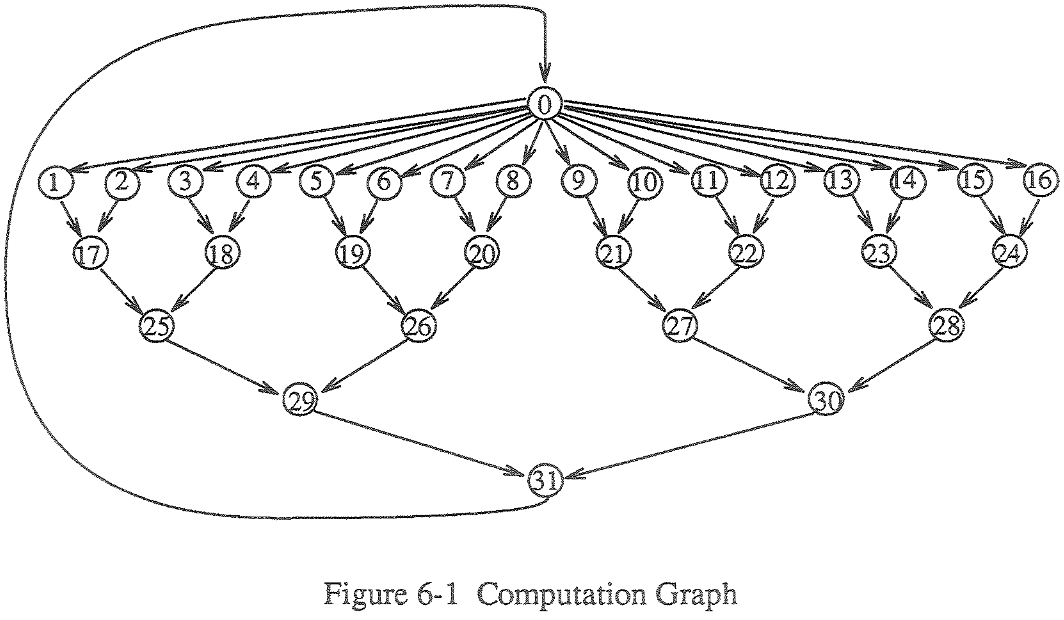


Figure 6-1 Computation Graph

Cluster Name	Node Number	Cluster Name	Node Number
C_0	0,1,17,25,29,31	C_8	9,21,27,30
C_1	2	C_9	10
C_2	3,18	C_{10}	11,22
C_3	4	C_{11}	12
C_4	5,19,26	C_{12}	13,23,28
C_5	6	C_{13}	14
C_6	7,10	C_{14}	15,24
C_7	8	C_{15}	16

Table 6-1 Linear Clusters

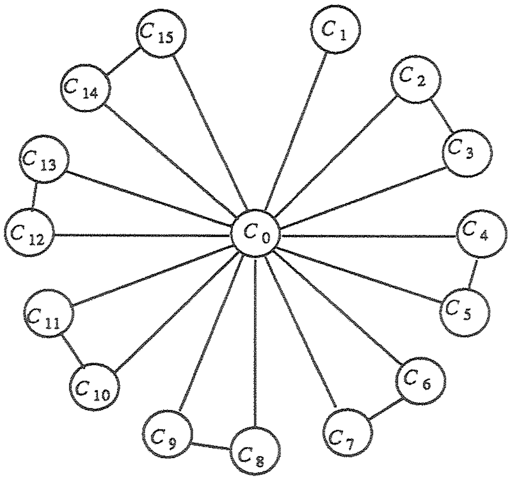


Figure 6-2 Clustered Graph

The operations performed by all the nodes except node 0 are to eliminate the multiples of a given prime from a group of numbers sent from its direct ancestor nodes. Each of these operations may be performed independently and simultaneously, using the different set of data in each node. Node 0 divides evenly the numbers which have not been eliminated yet into groups and distributes them to its direct descendants. The leftmost node (except node 31) on each level sends a prime number (whose multiples are to be eliminated) to the nodes on the next lower level. Node 31 sends the remaining uneliminated numbers to node 0. Coming back again to node 0, we repeat the elimination sweeps until only the primes are left. It is enough, however, to repeat the elimination sweeps only up to $\sqrt{N}$ based on the observation that any number less than N cannot have all its multiples greater than $\sqrt{N}$. Even though our parallel implementation is not the most efficient way to find the prime numbers on the iPSC, it provides us with a communication-intensive computation graph, which is cyclic and regular. Table 6-1 shows how the nodes in Fig. 6-1 are clustered in Fig. 6-2.

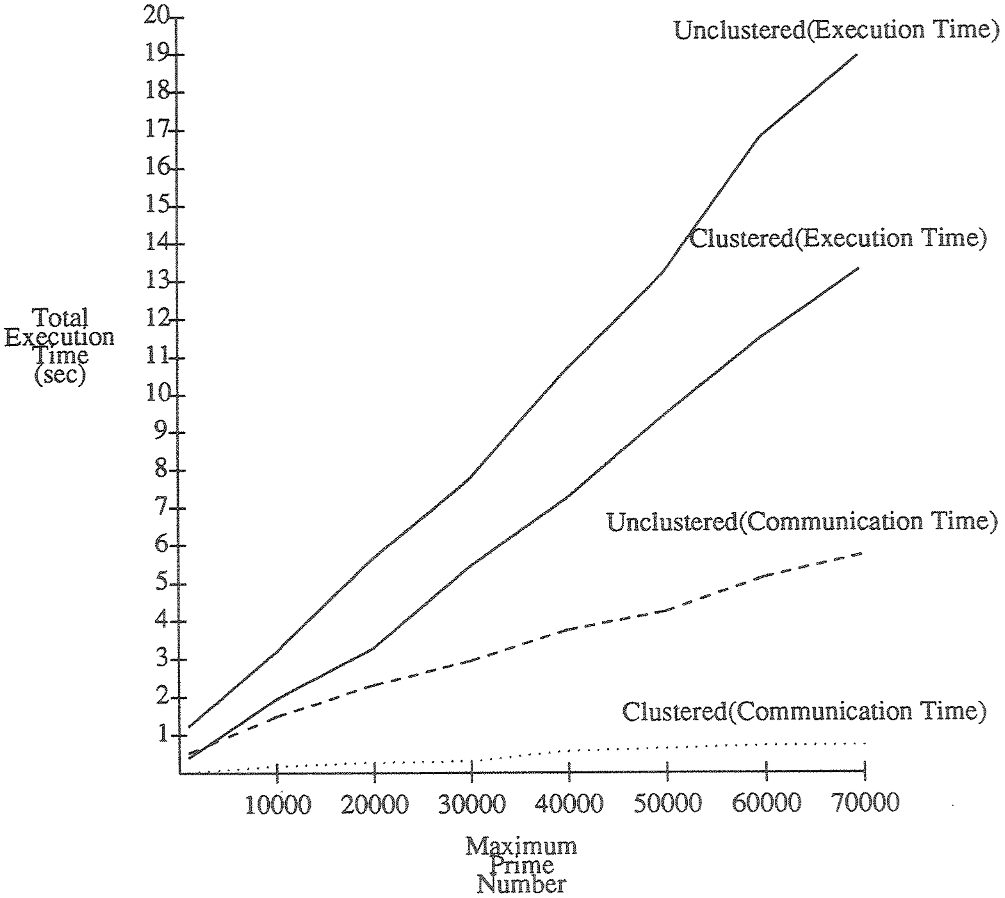


Figure 6-3 Comparison of Total Execution Times and Total Communication Times

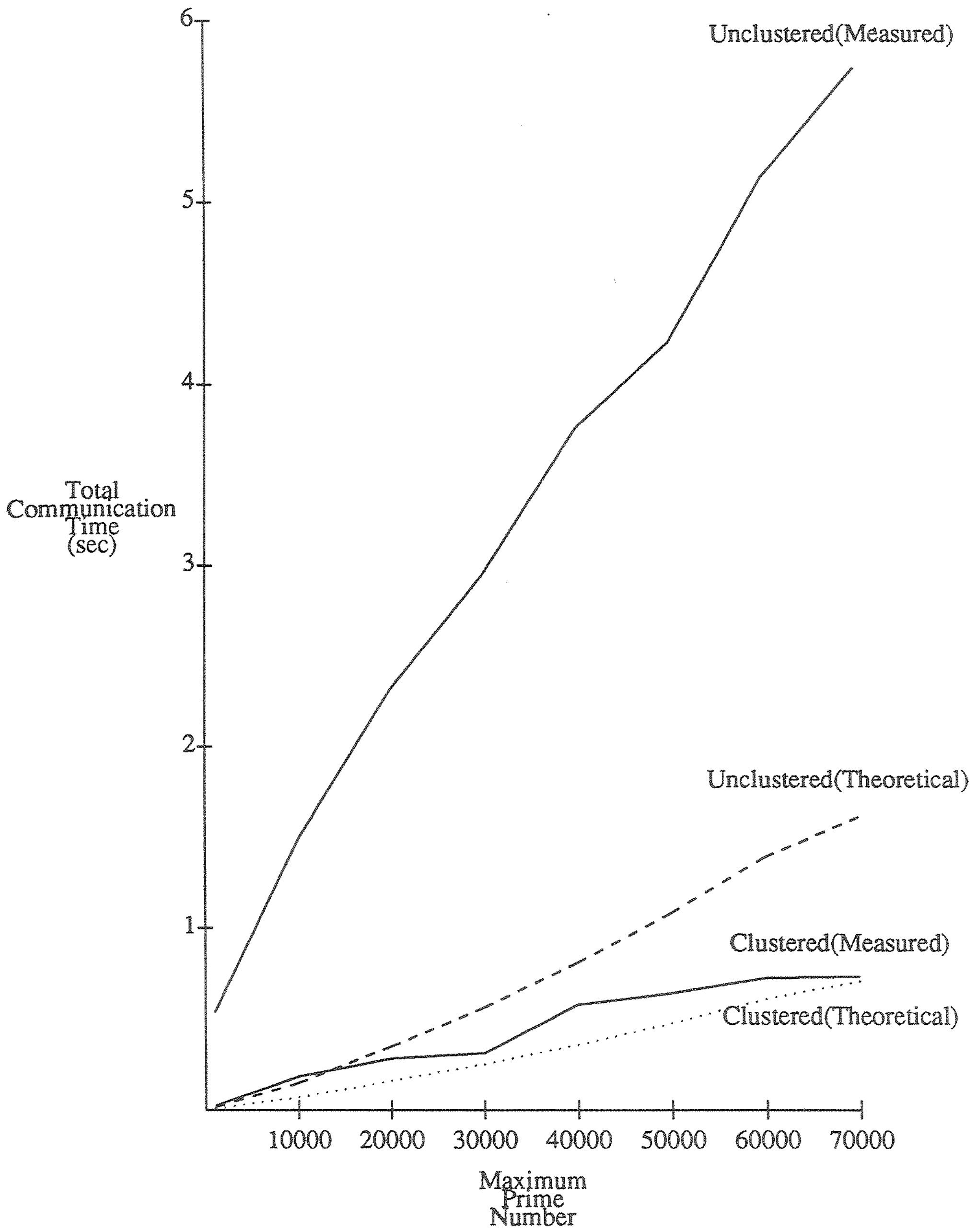


Figure 6-4 Measured vs Theoretical Total Communication Times on the Critical Path

Fig. 6-3 demonstrates how much we can further reduce the total execution time of the computation graph in Fig. 6-1 after linear clustering it. As expected, the improvement of total execution time is achieved mainly by reducing communication time. In Fig. 6-4, we compare the measured communication times with their corresponding theoretical lower bounds for both the unclustered and clustered graphs. The curve for the measured communication time is very close to the curve for its theoretical lower bound in the clustered graph. In fact, the proximity of the two curves shows how efficient our algorithm is.

The total communication time consists of pure message transmission time and communication delay. Communication delay in the iPSC is mostly affected by the following factors: the number of hops a message must travel, waiting time for a message to arrive, and message queuing delay for a channel. We can significantly reduce unnecessary interprocessor communications by clustering inherently sequential processes into one cluster and, as such, reduce message waiting time and message queuing delay. Furthermore, it is expected that the adjacency is better maintained during mapping of the *VAG* shown in Fig. 6-2 onto a *PAG* representing the iPSC than during direct mapping of the computation graph shown in Fig. 6-1 onto the *PAG*, largely due to the simplified graph after linear clustering. For example, the average number of hops each message must travel was reduced from 1.652 to 0.978 after clustering the computation graph. For the unclustered graph, on the other hand, the various overheads keep being accumulated as N gets larger. It results in a large gap between two curves in the unclustered graph as the maximum prime number to be found gets larger.

Max Prime Number	Unclustered	Clustered
10000	0.420	0.058
20000	0.348	0.037
30000	0.308	0.011
40000	0.277	0.030
50000	0.239	0.017
60000	0.223	0.010
70000	0.218	0.002

Table 6-2 Comparison of Communication Overhead to Total Execution Time

Table 6-2 compares the communication overhead of the unclustered graph with that of the clustered graph. While the clustered graph has very little overhead, the unclustered graph wasted a significant portion of execution time for communication overhead. Note, however, that the overhead diminishes as N gets larger. The reason for this can be explained as follows: The ratio of communication time to computation time decreases as N get larger (cf. Table 6-3), since most of the nonprimes are scratched out while calculating the first few primes. As a result, synchronization overhead is expected to be diminished.

Max Prime Number	Unclustered	Clustered
10000	0.872	0.104
20000	0.694	0.094
30000	0.613	0.061
40000	0.546	0.086
50000	0.472	0.073
60000	0.440	0.067
70000	0.435	0.058

Table 6-3 The Ratio of Communication Time to Computation Time

Table 6-4 demonstrates another advantage of linear clustering. As mentioned in Chapter 4, linear clustering enables us to make use of less resources than required by unclustered graphs. For example, when the computation graph shown in Fig. 6-1 requires all 32 nodes for a one-to-one mapping of nodes to processors, the *VAG* shown in Fig. 6-2 requires just 16 processors. Furthermore, the former requires 47 direct communication channels while the latter requires 32 direct channels. It makes it possible to run two *VAG's* simultaneously on the iPSC. We measured the total execution time to complete two clustered graphs and, then, divided it by 2 to estimate the average time to complete one clustered graph. The throughput results indicate the number of application programs which have been completed during the time span required to get all the primes up to 70000 without clustering the computation graph. As can be seen in Table 6-4, when executing two clustered graphs simultaneously in the iPSC, the throughput improvement is always more than twice. This result reflects the fact that in spite of the fact that we run two computation graphs simultaneously, there is not much overhead caused by that. This result also demonstrates the significant reduction in communication overhead and better mapping caused by linear clustering.

Max Prime Number	Number of Tasks Completed		Ratio(B/A)
	Unclustered(A)	Multiple Clustered(B)	
10000	5.930	17.248	2.909
20000	3.348	9.684	2.892
30000	2.439	6.465	2.651
40000	1.779	4.919	2.765
50000	1.434	3.772	2.630
60000	1.128	2.969	2.632
70000	1.000	2.575	2.575

Table 6-4 Comparison of Total Throughput

To conclude, we find that linear clustering makes it possible to decrease the total communication time significantly by getting rid of unnecessary interprocessor communication and reducing the number of hops for a message to travel. When expecting heavy communication among processes, it has been usually the case that we avoid loosely-coupled architecture. We show that linear clustering may reduce such an overhead drastically, and therefore, loosely-coupled architectures may be applicable to such an application.

6.4. Forward Elimination of Matrices

Gaussian elimination is the well-known algorithm for solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$ by successively eliminating the unknowns. We assume that $\mathbf{A}$ is an $n \times n$ real, dense, nonsingular matrix, $\mathbf{x}$ is the column vector of

n unknowns, and $\mathbf{b}$ is a given column vector of n real constants. In this section, we describe a series of experiments related to the algorithm using a Sequent Balance 21000 and then discuss their results. Although Gaussian algorithm is a two step process of forward elimination and back substitution, we are only interested in the forward elimination step represented by the computation graph in Fig. 6-5.

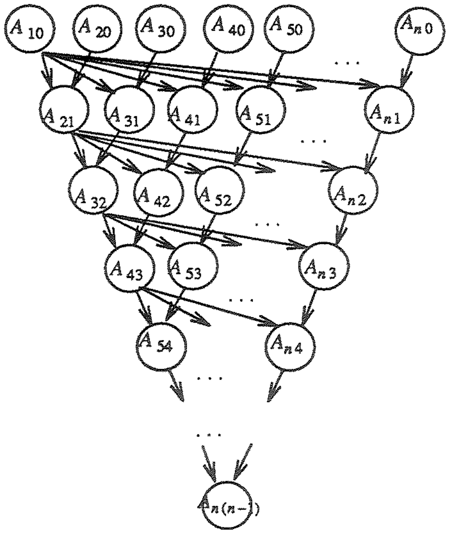


Figure 6-5 Computation Graph

Each node (labeled A_{ij} in Fig. 6-5) represents the row operation for each pivot row k $(1 \le k \le n-1)$ of matrix $\mathbf{A}$ to make the element $a_{l,k}$ $(k+1 \le l \le n)$ zero. This operation also updates the other parts of matrix $\mathbf{A}$ as follows:

$$\text{For } k+1 \le i \le n,\ a_{i,j} = a_{i,j} - a_{i,k} \times \frac{a_{k,j}}{a_{k,k}},\ \text{where } k \le j \le n.$$

The row operations for the pivot rows can be performed simultaneously as long as sequencing constraints are satisfied as described in the computation graph in Fig. 6-5. To be specific, node A_{ij} can be triggered as soon as nodes $A_{(i-1)(j-1)}$ and $A_{i(j-1)}$ have been completed.

The computation graph in Fig. 6-5 was clustered in such a way that the nodes with the same row index (say, i) were put into linear cluster C_{i-1} as shown in Fig.

6-6. It should be mentioned here that it is permitted to create simultaneously up to 9 processes in the Balance with 10 processors. Taking into account this rèstriction, the *VAG* in Fig. 6-6 needs to be transformed into another *VAG* in Fig. 6-7. A merged node M_i in Fig. 6-7 was constructed as follows:

$$\text{For } 0 \le i \le 8, M_i = \{C_j \mid i = j\,(mod\ 9) \text{ and } 0 \le \text{j} \le \text{n}-1\}.$$

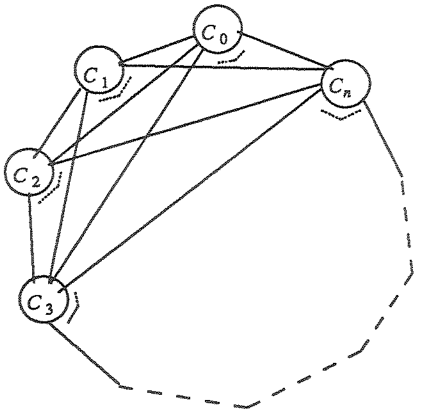


Figure 6-6 Clustered Graph

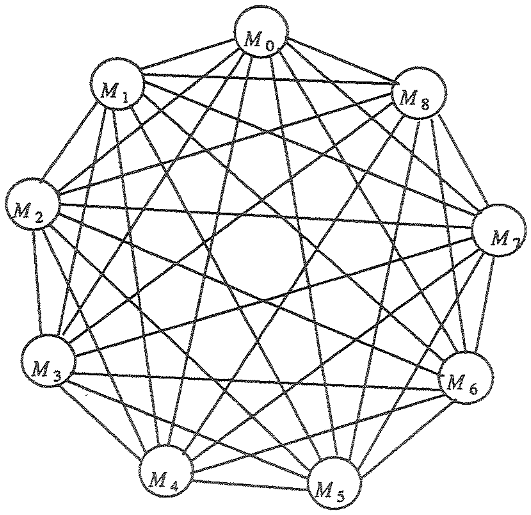


Figure 6-7 Merged Graph

In Fig. 6-8, we can see that how much further cluster merging improves performance of the clustered graph; the result for the merged graph is very close to the theoretical lower bound of the total execution time.

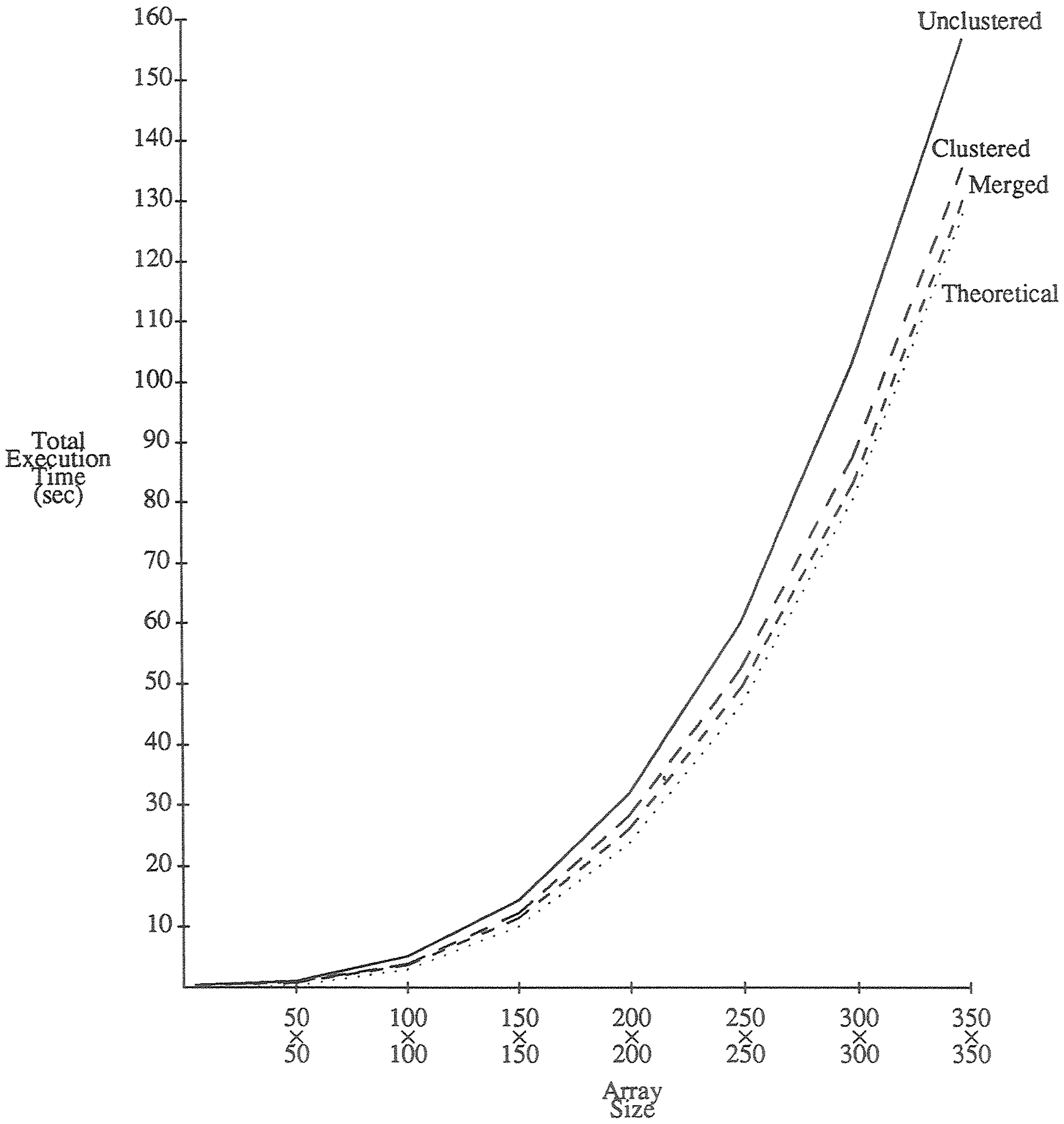


Figure 6-8 Comparison of Total Execution Times using 9 Processors

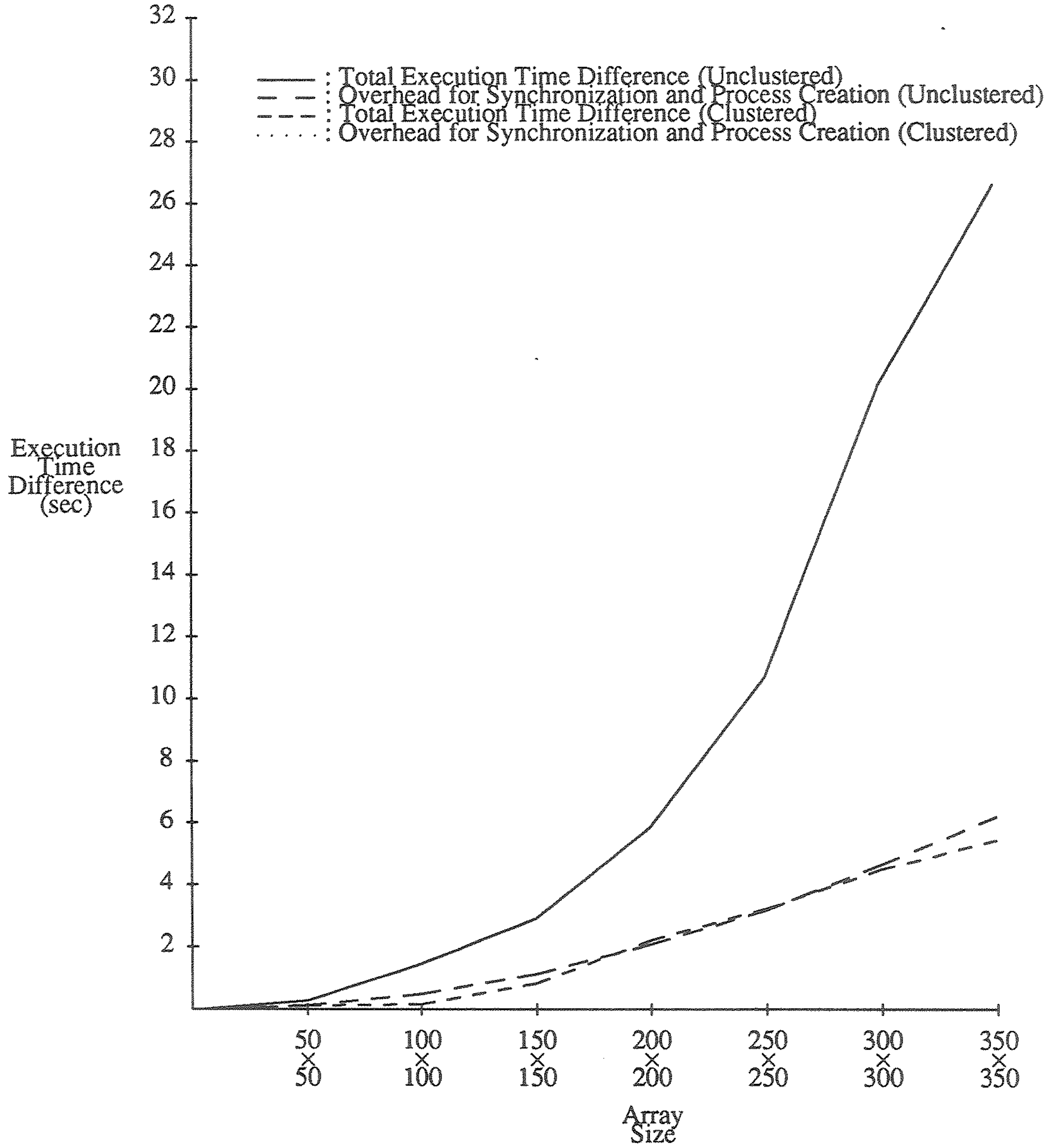


Figure 6-9 Analysis of Scheduling Overhead Relative to Overhead based on Merging

There are basically two factors which affect the communication overhead in the Balance: the overhead in creating multiple processes and the overhead in synchronization and communication among processes. Observe, though, that there is nominal communication overhead between processors because the Balance has shared memory through a common bus. As a result, whether a computation graph is being clustered or not, the sources of performance improvement are to reduce the number of processes to be created and to reduce unnecessary synchronization overheads. The other important source of the improvement is to balance the workloads of processors.

Fig. 6-9 presents an analysis of the total execution time differences shown in Fig. 6-8. The x-axis in Fig. 6-9 represents the total execution time and the communication overhead for synchronization and process creation and other overhead like bus contention in the merged case. The curves in Fig. 6-9 represent the differences among the total execution time and the overhead of the unclustered and clustered graphs relative to the merged one (represented by the x-axis). In the case of the unclustered graph, there is significant reduction in synchronization and process creation overhead; the rest of the reduction is accomplished by load balancing. On the other hand, in the case of the clustered graph, there is very little difference in synchronization and process creation overhead; notice that the curve for the overhead is virtually overlapped with the x-axis. As a result, load balancing should affect the difference of the total execution times.

Array Size	Number of Processors								
	9			6			3		
	Unclustered	Clustered	Merged	Unclustered	Clustered	Merged	Unclustered	Clustered	Merged
10×10	0.99%	1.02%	1.02%	2.26%	2.30%	2.33%	7.97%	8.87%	9.09%
50×50	33.69%	40.00%	45.39%	47.54%	56.38%	64.63%	67.03%	79.74%	86.84%
100×100	58.31%	77.71%	81.24%	76.55%	87.31%	89.59%	83.27%	93.86%	95.44%
150×150	70.74%	82.72%	88.79%	81.55%	92.32%	94.17%	89.56%	96.36%	96.99%
200×200	74.73%	84.35%	91.49%	85.11%	94.50%	95.84%	89.99%	96.52%	97.10%
250×250	77.61%	88.64%	94.42%	86.61%	95.08%	96.03%	90.08%	96.66%	98.05%
300×300	78.09%	92.06%	97.05%	86.67%	95.38%	98.54%	92.22%	98.42%	99.50%
350×350	81.58%	94.34%	98.29%	86.88%	95.43%	98.81%	92.36%	98.29%	99.81%

Table 6-5 Comparison of Processor Utilization

Table 6-5 presents the processor utilization versus the number of processors for a range of numbers using three different graphs. From the table, the following observations may be made: As the number of processors decreases, the processor utilization increases since the workload of each processor becomes more balanced. Secondly, as the array size increases, the processor utilization also increases since the ratio of computation time to the various communication overhead increases.

To conclude, this experiment demonstrated that our scheduling algorithm is also applicable to a tightly-coupled system. This result also suggests that merging plays an important role for balancing workloads as well as reducing resource requirement. We also observed that we may get better processor utilization as the number of processors gets smaller. If the total throughput is important, we would utilize as small a number of processors, each of whose utilization approached 1, as possible. For this specific application, we got the best utilization when the number of processors was three.

6.5. Synthetic Program

Fig. 6-10 represents a synthetic computation graph. It is different from the previous computation graphs in that it does not represent a real application program. We modified the computation graph for molecular dynamics code [EUB86] so that the number of nodes and edges in the graph became larger than those available in a *PAG* representing a heterogeneous system. The system is simulated by an Intel iPSC in such a way that half of the processors and one fifth of the communication channels are twice as fast as the real ones. We deliberately manipulate the computation and communication times of the nodes and edges of the irregular computation graph, respectively. Fig. 6-11 shows the clustered graph based on Table 6-6. Solid and dotted lines represent the primary and secondary edges (cf. Chapter 5), respectively.

In order to utilize the concept of data driven synchronization of the static dataflow machines, a node is assumed to be fired only after all the messages are available and outputs messages only at the end of its computation. The main purpose of the experiments using this synthetic program is to compare the influence of linear clustering between homogeneous and heterogeneous systems.

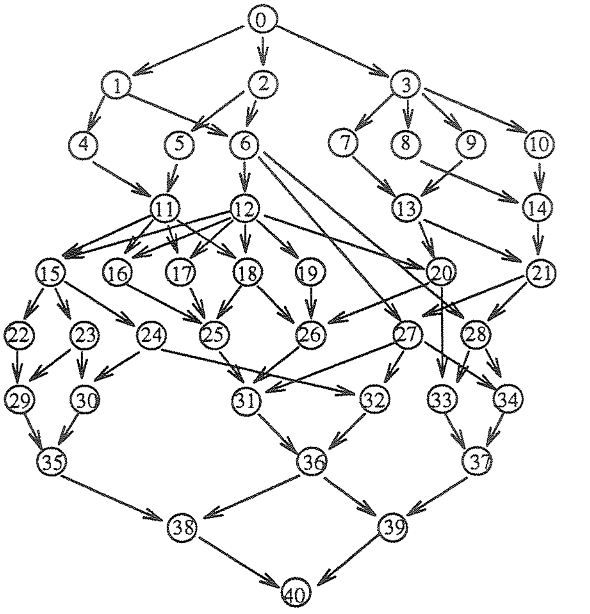


Figure 6-10 Computation Graph

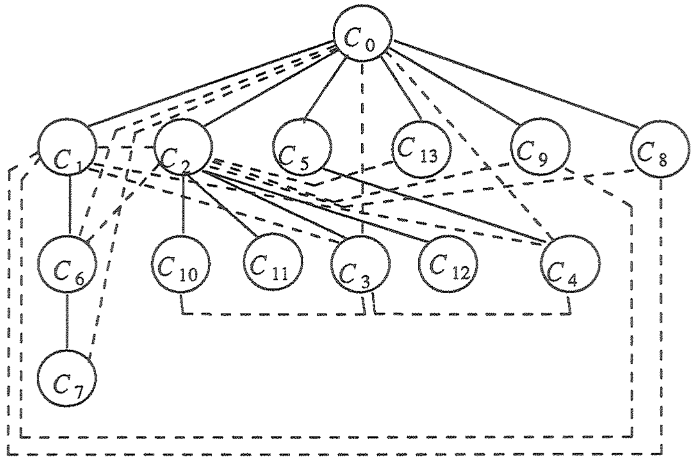


Figure 6-11 Clustered Graph

Cluster Name	Node Number	Cluster Name	Node Number
C_0	0,2,6,12,15,22,29,35,38,40	C_7	20
C_1	5,11,16,25,31,36,39	C_8	19
C_2	3,7,13,21,28,33,37	C_9	17
C_3	27,34	C_{10}	10,14
C_4	24,32	C_{11}	9
C_5	23,30	C_{12}	8
C_6	18,26	C_{13}	1,4

Table 6-6 Linear Clusters

Fig. 6-12 compares the total execution times for four different cases: the measured and the theoretical total execution times for homogeneous and heterogeneous systems. Note that dotted and solid bars represent the total execution times for the unclustered and the clustered graphs, respectively. As can be seen, after linear clustering the computation graph shown in Fig. 6-10, we got a reduction of 58.69% in the measured total execution time in the heterogeneous iPSC, versus 36.36% in the homogeneous iPSC. The total execution time of the unclustered graph in the heterogeneous iPSC is improved by just 12.57% when comparing with that in the homogeneous one. On the other hand, the total execution time of the clustered graph in the heterogeneous iPSC is improved by 47.20%. These comparisons demonstrate that linear clustering allows us to utilize heterogeneity of the target architecture more efficiently than the unclustered graph. They reemphasize the fact that linear clustering makes it easier to take advantage of architectural characteristics during mapping. In the theoretical lower bound, the improvement is not as impressive as the measured case. The reason for that is the disregard of various communication overheads except for pure transmission time when measuring the theoretical bounds. Nevertheless, the clustered graph shows much better performance improvement in the heterogeneous case than the homogeneous case.

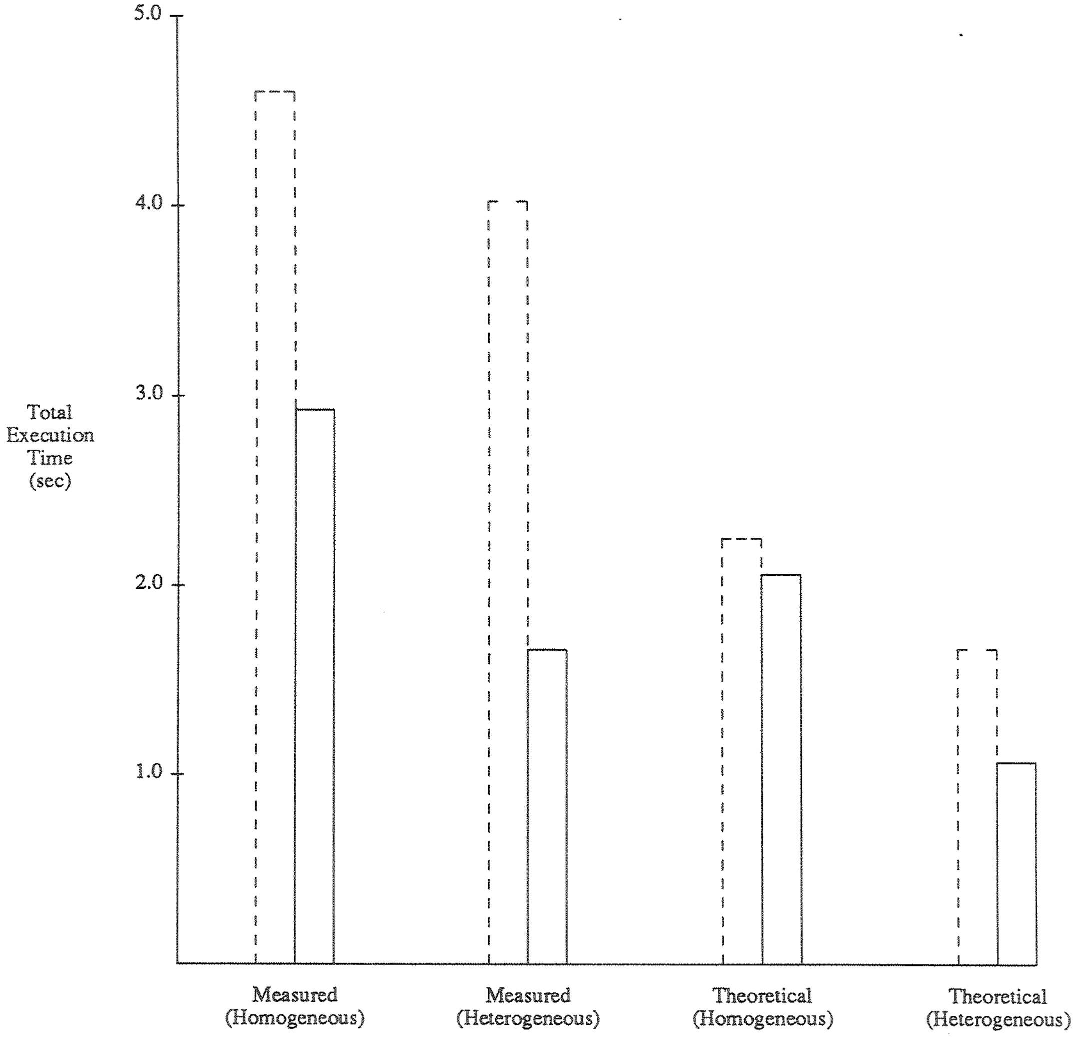


Figure 6-12 Comparison of Total Execution Times (Unclustered vs Clustered)

Tabulated in Table 6-7 is throughput comparison between homogeneous and heterogeneous architectures. As can be seen, since there are 14 clusters in Fig. 6-11, we are able to assign and execute at least two clustered graphs simultaneously in the iPSC using only 28 processors. The throughput improvement in the heterogeneous iPSC is 36.25% better that in the homogeneous iPSC.

Clustering Type	Homogeneous	Heterogeneous
Unclustered	1.000	1.000
Clustered	1.571	2.421
Multiple Clustered	3.030	4.753

Table 6-7 Comparison of Total Throughput

To conclude, linear clustering seems to be more effective for heterogeneous systems for better utilization of their architectural characteristics than for homogeneous systems. It allows us to utilize better the processors with more computing power and the channels with faster speed.

CHAPTER 7

CONCLUDING REMARKS

Technological innovations made it feasible to construct a variety of multiprocessor systems from collection of processors. These processors can either share common memory or have their own local memories, or both. To fully utilize such systems, enormous research efforts have been invested on the development of effective ways to find optimal allocations of application programs to processors so that the total execution time can be minimized. Traditionally, there has been a split between research on multiprocessor scheduling for tightly-coupled and loosely-coupled systems. Most researchers have focused on the development of specific scheduling strategies to take advantage of unique hardware characteristics such as interconnection topologies.

As opposed to the previous approaches, we have investigated and proposed multiprocessor scheduling techniques and heuristic mapping algorithms which are applicable to a spectrum of multiprocessor systems. The fundamental idea we have used is that multiprocessor scheduling can be regarded as a mapping of a computation graph onto an architecture graph. We defined the graph models for computation graphs and architecture graphs. The former, which are acyclic directed graphs, describe parallel computation structures to be executed. The latter, which are undirected graphs, represent target multiprocessor systems onto which the computation graphs are to be mapped.

Based on the models, we proposed new scheduling techniques using linear clusters. A linear cluster is a connected subgraph of a computation graph which is in the form of a linear list of schedulable units of computation. We justified that linear clustering was an effectual heuristic to compromise between two conflicting goals, minimization of interprocessor communication and maximization of potential parallelism, and to satisfy the other goals the throughput enhancement and the workload balance, relatively well. The underlying idea of linear clustering is that the schedulable units of computation that are sequentially dependent on each other are to be assigned to one processor, while those that are mutually independent are to be allocated to separate processors. The critical restriction of linear clustering is that it expects a computation graph to be acyclic. In order to relieve this restriction, we identified the cases that cyclic computation graphs might be transformed into acyclic graphs in a straightforward manner if they meet certain properties. We also proposed a technique for linear cluster merging to balance the workload of processors and to reduce the amount of resources necessary for a parallel computation.

After linear clustering and merging, we can identify an optimal architecture graph for a given computation graph. We developed two efficient heuristic scheduling algorithms which mapped the optimal architecture graph onto a physical architecture graph, which in turn could represent either a homogeneous or a heterogeneous multiprocessor system. Those algorithms rely not only on local information but also on limited global information. Both algorithms utilize dominant request trees to reduce the mapping complexity, but take quite different approaches to mapping the trees onto architecture graphs. Most importantly, in the case of homogeneous mapping, the trees are directly mapped onto a physical architecture graph. On the other hand, in the case of heterogeneous mapping, they are mapped onto dominant service trees. A dominant service tree provides limited global information on a heterogeneous multiprocessor system like which resources have high performance.

We presented performance evaluation of our mapping algorithms on an Intel iPSC with 32 processors and a Sequent Balance with 10 processors. The evaluation results showed that we could get impressive performance improvements of already-parallelized computations on both the loosely-coupled system and the tightly-coupled system, utilizing less hardware resources whenever feasible.

There are many directions for future work. Of immediate interest is to enhance the models for computation and architecture graphs for allowing nondeterminism in the former and for embedding more scheduling constraints in the latter. Next, it will be worthwhile to investigate how to allow dynamic binding in computation graphs. Other future work will involve the extension of our mapping algorithms to include more heuristics. It will be also interesting to compare the performance results of our algorithms with those of other approaches for the same computation graphs. Finally, the important issues that have not been considered in this thesis are how optimal our algorithms are and what their theoretical performance bounds are.

Bibliography

[AE82] Ae, T., and Aibara, R., "Experimentation and Analysis of Multiprocessor Systems," *IEEE 1982 Real-Time System Symp.*, Dec. 1982, pp. 69-80.

[AHO83] Aho, A.V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, 1983.

[ARG86] "Using the Encore Multimax," Argonne National Laboratory, Mathematics and Computer Science Division, Technical Report ANL/MCS-TM-65, Argonne National Lab., Argonne, Ill., 1986.

[BAN87] Banerjee, J., Kim, W., Kim, S. J., and Garza, J. F., "Clustering a DAG for CAD Databases," To appear in *IEEE-SE.*

[BAR81] Barnes, E. R., "An Algorithm for Partitioning the Nodes of a Graph," IBM Research Report RC 8690, Feb. 1981.

[BBN85a] "Butterfly (TM) Parallel Processor Overview," Bolt Beranek and Newman Inc., Cambridge, MA, June 1985.

[BBN85b] "The Uniform System Approach To Programming the Butterfly (TM) Parallel Processor," Bolt Beranek and Newman Inc., Cambridge, MA, Nov. 1985.

[BER84] Berman, F., and Snyder, L., "On Mapping Parallel Algorithms into Parallel Architectures," *Proc. Int'l Conf. on Parallel Processing*, Aug. 1984, pp. 307-309.

[BIA85] Bianchini, R. P., and Shen, J. P., "Automated Compilation of Interprocessor Communication for Multiple Processor Systems," Dept. of ECE, CMU, Nov. 1985.

[BOK81] Bokhari, S. H., "On the Mapping Problem," *IEEE-TC*, Vol. C-30, No. 3, Mar. 1981, pp. 207-214.

[BOU72] Bouknight, W. J., Denenberg, S. A., McIntyre, D. E., Randall, J. M., Sameh, A. H., and Slotnick, D. L., "The Illiac IV system," *Proc. IEEE*, Apr. 1972, pp. 369-379.

[BRO85] Browne, J. C., "Formulation and Programming of Parallel Computations: A Unified Approach," *Proc. of Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 624-631.

[BRO86] Browne, J. C., "Framework for Formulation and Analysis of Parallel Computation Structures," *Parallel Computing* 3, 1986, pp. 1-9.

[BRU74] Bruno, J., Coffman, E. G., Jr., and Sethi, R., "Scheduling Independent Tasks to Reduce Mean Finishing Time," *CACM* 17, 1974, pp. 382-287.

[BRY81] Bryant, R. M., and Finkel, R. A., "A Stable Distributed Scheduling Algorithm," *2nd Int'l. Conf. on Distributed Computing Systems*, 1981, pp. 314-323.

[CAM85] Campbell, M. L., "Static Allocation for a Data Flow Multiprocessor," *Proc. of Int'l Conf. on Parallel Processing*, Aug. 1985, pp. 511-517.

[CHI84] Chiang, W. P., "Optimal Graph Clustering Problems with Applications to Information System Design," Technical Report CRL-TR-30-84, The Univ. of Michigan, June 1984.

[CHO82] Chou, T. C. K., and Abraham, J., "Load Balancing in Distributed Systems," *IEEE-SE*, Vol. SE-8, No. 4, July 1982, pp. 401-412.

[CHU69] Chu, W. W., "Optimal File Allocation in a Multiple Computer System," *IEEE-TC*, Vol. C-18, No. 10, Oct. 1969.

[CHU80] Chu, W. W., Holloway, L. J., Lan, M.-T., and Efe, K., "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, No. 11, Nov. 1980, pp. 57-69.

[COF72] Coffman, E. G., Jr., and Graham, R. L., "Optimal Scheduling for Two-Processor Systems," *Acta Informatica* 1, 1972, pp. 200-213.

[COF76] Coffman, E. G., Jr. (Ed.), *Computer and Job-Shop Scheduling Theory*, John Wiley and Son, N. Y., 1976.

[DEG81] DeGroot, R. D., "Mapping Computation Structures onto SW-Banyan Network", Ph. D. Thesis, The Univ. of Texas at Austin, Dec., 1981.

[DEM82] Deminet, J., "Experience with Multiprocessor Algorithms," *IEEE-TC*, Vol. C-31, No. 4, Apr. 1982, pp. 278-288.

[DUN86] Dunigan, T. H., "Hypercube Performance," in *Hypercube Multiprocessor*, M. T. Heath (Ed.), SIAM, 1986.

[EAG86] Eager, D. L., Lazowska, E. D., and Zahorjan, J., "Dynamic Load Sharing in Homogeneous Distributes Systems," *IEEE-SE*, Vol. SE-12, No. 5, May 1986, pp. 662-675.

[EDL85a] Edler, J., Gottlieb, A., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M., Teller, P. J., and Wilson, J., "Issues Related to MIMD Shared-memory Computers: The NYU Ultracomputer Approach," *IEEE Proc. of the 12th Annual Int'l Symp. on Computer Architecture,* June 1985, pp. 126-135.

[EDL85b] Edler, J., Gottlieb, A., and Lipkis, J., "Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3," Ultracomputer Note No. 91, Courant Institute of Math. Sci., NYU, Dec. 1985.

[EFE82] Efe, K., "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, No. 6, June 1982, pp. 50-56.

[EUB86] Eubank, S., Personal Communication.

[FIS84] Fisher, J. A., and O'Donnel, J. J., "VLIW Machines: Multiprocessors We Can Actually Program," *COMPCON*, Spring, 1984, pp. 299-305.

[FOR64] Ford, L. R., and Fulkerson, D. R., *Flows in Networks*, Princeton Univ. Press, Princeton, N. J., 1964.

[FOR78] Forsdick, H., Schantz, R., and Thomas, R., "Operating Systems for Computer Networks," *IEEE Computer*, Vol. 11, Jan. 1978.

[FUL73] Fuller, S. H., and Siewiorek, D. P., "Some Observations on Semiconductor Technology and the Architecture of Large Digital Modules," *IEEE Computer*, Oct. 1973, pp. 15-21.

[GAJ83] Gajski, D., Kuck, D., Lawrie, D., and Sameh, A.," Cedar - A Large Scale Multiprocessor," *Proc. of Int'l Conf. on Parallel Processing*, Aug.

1983, pp. 524-529.

[GAR75] Garey, M. R., and Graham, R. L., "Bounds for Multiprocessor Scheduling with Resource Constraints," *SIAM J. Comp.*, Vol. 4, No. 2, June 1975, pp. 187-200.

[GAR79] Garey, M. R., and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. M. Freeman and Company, New York, 1979.

[GOT83a] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE-TC*, Vol. C-32, No. 2, Feb. 1983, pp. 175-189.

[GOT83b] Gottlieb, A., Lubachevsky, B., and Rudolph, L., "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM TOPLAS* 5, Apr. 1983, pp. 164-189.

[GRA69] Graham, R. L., "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Appl. Math.,* Vol. 17, No. 2, March 1969, pp. 416-429.

[GYL76] Gylys, V. B., and Edwards, J. A., "Optimal Partitioning of Workload for Distributed Systems," *Digest of Papers, COMPCON*, Fall 1976.

[HAE80] Haessig, K., and Jenny, C. J., "Partitioning and Allocating Computational Objects in Distributed Computing Systems," *IFIP*, 1980, pp. 503-508.

[HAY82] Haynes, B. S., Lau, R. L., Siewiorek, D. P., and Mizell, D. W., "A Survey of Highly Parallel Computing," *IEEE Computer*, Jan. 1982, pp. 9-24.

[HEA70] Heart, F. E., Kahn, R. E., Ornstein, S. M., Crowther, W. R., and Walden, D. C., "The Interface Message Processor for the ARPA Computer Network," *Proc. AFIPS*, SJCC, 1970, pp. 551-567.

[HIL85] Hillis, W. D., *The Connection Machine*, MIT Press, 1985.

[HU61] Hu, T. C., "Parallel Sequencing and Assembly Line Problems," *Operations Research*, 9, 6, 1961, pp. 841-848.

[INT87] "iPSC User's Guide," Intel Corporation, Apr. 1987.

[JON80] Jones, A. K., and Gehringer, E., "The Cm* Multiprocessor Project: A Research Review," Technical Report CMU-CS-80-131, CMU, July 1980.

[KAR66] Karp, R. M., and Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math.*, Vol. 14, No. 6, Nov. 1966, pp. 1390-1411.

[KAR72] Karp, R. M., "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (Eds.), Plenum Press, New York, 1972, pp. 85-103.

[KAS84] Kasahara, H., and Narita, S., "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE-TC*, Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[KAT78] Katsuki, D., Elsam, E. S., Mann, W. F., Roberts, E. S., Robinson, J. G., Skowronski, F. S., and Wolf, E. F., "Pluribus - An Operational Fault-Tolerant Multiprocessor," *Proc. IEEE*, Vol. 66, No. 10, Oct. 1978, pp. 1146-1159.

[KER70] Kernighan, B. W., and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Tech. J.*, Vol. 49, No. 2, Feb. 1970, pp. 291-307.

[KER71] Kernighan, B. W., "Optimal Sequential Partitions of Graphs", *JACM*, Vol. 18, No. 1, Jan. 1971, pp. 34-40.

[KIM86a] Kim, S. J., "Interface Specification Languages for Computation and Resource Graph Generations," Internal Memo, Univ. of Texas at Austin, Jan. 1986.

[KIM86b] Kim, S. J., "On a Physical Mapping for Homogeneous Multiprocessor Systems," Internal Memo, Univ. of Texas at Austin, Nov. 1986.

[KIR83] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671-680.

[KUN82] Kung, H. T., "Why Systolic Architectures ?", *IEEE Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37-46.

[KUN84] Kung, H. T., "Systolic Algorithms for the CMU Warp Processor," Technical Report CMU-CS-84-158, Dept. of Comp. Sci., CMU, Jan. 1984.

[LEE77] Lee, R. P., and Muntz, R. R., "On the Task Assignment Problem for Computer Networks," *Proc. 10th Hawaii Int'l Conf. System Sciences*, Jan. 1977, pp. 5-9.

[LEE87] Lee, S. Y., and Aggarwal, J. K., "A Problem-driven Approach to Parallel Processing: System Design/Scheduling and Task Mapping," Technical Report TR-87-7-39, Computer and Vision Research Center, The Univ. of Texas, June 1987.

[LIU78] Liu, C. L., and Dhall, S. K., "On a Real-Time Scheduling Problem," *Operations Research*, Vol. 26, No. 1, Feb. 1978, pp. 127-140.

[LLO80] Lloyd, E. L., "Scheduling Task Systems with Resources," Ph. D. Thesis, MIT, May 1980.

[LO85] Lo, V. M., "Task Assignment to Minimize Completion Time," *5th Int'l. Conf. on Distributed Computing Systems,* 1985, pp. 329-336.

[LUK75] Lukes, J. A., "Combinatorial Solution to the Partitioning of General Graphs," *IBM J. Res. Develop.*, Vol. 19, 1975, pp. 170-180.

[LUS85] Lusk, E. L., and Overbeek, R. A., "Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors," Technical Report ANL-84-51, Argonne National Lab., Argonne, Ill., June 1985.

[MA82] Ma, P.-Y. R., Lee, E. Y. S., and Tsuchiya, M., "A Task Allocation Model for Distributed Computing Systems," *IEEE-TC*, Vol. C-31, No. 1, Jan. 1982, pp. 41-47.

[MAR67] Martin, D. E., and Estrin, G., "Model of Computational Systems - Cyclic to Acyclic Graph Transformations," *IEEE-TC*, Vol. 16, No. 1, Feb. 1967, pp. 70-79.

[MAS82] Mashburn, H. H., "The C.mmp/Hydra Project: An Architectural Overview," in [SIE82] pp. 350-370.

[MAT85] Matelan, N., "The Flex/32 Multicomputer," *Proc. of the 12th Int'l Conf. on Computer Architecture,"* June 1985, pp. 209-213.

[MET76] Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed Packet Switching fo Local Computer Network," *CACM*, Vol. 19, No. 7, July 1976, pp. 395-404.

[NCU85] "NCUBE/*ten* : An Overview," NCUBE Corp., Nov. 1985.

[NI81] Ni, L. M., and Hwang, K., "Optimal Load Balancing Strategies for a Multiprocessor System," *Proc. Int'l Conf. on Parallel Processing,* Aug. 1981, pp. 352-357.

[OUS80] Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S., "Medusa: An Experimental in Distributed Operating System Structure," *CACM*, Vol. 23, No. 2, Feb. 1980, pp. 92-105.

[OUS82] Ousterhout, J. K., "Scheduling Techniques for Concurrent Systems," *Proc. of the Third Int'l Conf. on Distributed Systems,* Oct. 1982, pp. 22-30.

[PAT84] Pathak, G. C., "Towards Automated Design of Multicomputer System for Real-time Applications," Ph.D. Thesis, North Carolina State Univ., 1984.

[PFI85] Pfister, G. F., "The Architecture of the IBM Research Parallel Processor Prototype (RP3)," IBM Research Report RC 11210, June 1985.

[RAM72] Ramamoorthy, C. V., Chandy, K. M., and Gonzalez, M. J., "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE-TC*, Vol. C-21, No. 2, Feb. 1972, pp. 137-146.

[RAO79] Rao, G. S., Stone, H. S., and Hu, T. C., "Assignment of Tasks in a Distributed Processor System with Limited Memory," *IEEE-TC*, Vol. C-28, No. 4, Apr. 1979, pp. 291-299.

[SAD87] Sadayappan, P., and Ercal, F., "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Trans. Computer*, Vol.

C-36, No. 12, Dec. 1987, pp. 1408-1424.

[SAR86] Sarkar, V., and Hennessy, J., "Compile-time Partitioning and Scheduling of Parallel Programs," *Proc. on Compiler Construction*, 1986, pp. 17-26.

[SEI85] Seitz, C. L. "The Cosmic Cube," *CACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.

[SEQ86] "Balance Guide to Parallel Programming," Sequent Computer Systems, Inc., 1986.

[SHE85] Shen, C.-C., and Tsai, W.-H., "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE-TC*, Vol. C-34, No. 3, Mar. 1985, pp. 197-203.

[SIE82] Siewiorek, D. P., Bell, C. G., and Newell, A., *Computer Structures: Principles and Examples*, McGraw-Hill Company, N. Y., 1982.

[SOL79] Solomon, M. H., and Finkel, R. A., "The Roscoe Distributed Operating System," *Proc. 7th Symp. on Operating Principles*," 1979, pp. 108-114.

[STA84] Stankovic, J. A., and Sidhu, I. S., "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups," *Proc. on the 4th Int'l. Conf. on Distributed Computing Systems*, 1984, pp. 49-59.

[STO77] Stone, H. S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE-SE*, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

[STO78] Stone, H. S., and Bokhari, S. H., "Control of Distributed Processes," *IEEE Computer*, Vol. 11, No. 7, July 1978, pp. 97-106.

[TAN85] Tang, P., Yew, P.-C., and Zhu, C.-Q., "Processor Self-Scheduling in Large Multiprocessor Systems," No. 536, Center for Supercomputer Research and Development, Univ. of Ill., Nov. 1985.

[ULL73] Ullman, J. D., "Polynomial Complete Scheduling Problems," *Operating Systems Review*, Vol. 7, No. 4, 1973, pp. 96-101.

[ULL75] Ullman, J. D., "NP-complete Scheduling Problem," *J. of Computer System Science*, Vol. 10, 1975, pp. 384-393.

[VAN84] Van Tilborg, A. M., and Wittie, L. D., "Wave Scheduling - Decentralized Scheduling of Task Forces in Multicomputers," *IEEE-TC*, Vol. C-33, No. 9, Sep. 1984, pp. 835-843.

[WAN85] Wang, Y.-T., and Morris, R. J. T., "Load Sharing in Distributed Systems," *IEEE-TC*, Vol. C-34, No. 3, Mar. 1985, pp. 204-217.

[WAR62] Warshall, S., "A Theorem on Boolean Matrices," *JACM*, Vol. 9, No. 1, Jan. 1962, pp. 11-12.

[WIT80] Wittie, L. D., and van Tilborg, A. M., "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE-TC*, Vol. C-29, No. 12, Dec. 1980, pp. 1133-1144.