School of Computing

Design and Performance Evaluation of a Flexible Clustering and Allocation Scheme for Parallel Processing

Soontorn Chingchit

This thesis is presented as part of the requirements for the award of the Degree of Doctor of Philosophy of the

Curtin University of Technology

December 1999

Acknowledgements

I would like to thank my supervisor, Dr. Mohan Kumar, for his guidance, patience, persistence and encouragement. He has provided valuable advice and has been very patient throughout the years.

My thanks also go to A/Prof. Dennis. Moore, and Prof. Svetha Venkatesh, the past and current Head of the School of Computing, Curtin University, for their encouragement and support. I would also like to thank Ken Whitbread of the Center of International English, for his proof reading.

I gratefully acknowledge the support of the AusAID funding of the scholarship, without whose help this thesis would not have been possible. I would also like to thank the President of Rajabhat Institute Suandusit, A/Prof. Dr. Sirote Pholpuntin, for his support.

Finally, I would like to thank my wife Outaiwan, for her loyal support and love that helped me through the years.

Abstract

Parallel processing is an important and popular aspect of computing and has been developed to meet the demands of high-performance computing applications. In terms of hardware, a large number of processors connected with high speed networks are put together to solve large scale computationally intensive applications. The computer performance improvements made so far have been based on technological developments. In terms of software, many algorithms are developed for application problem execution on parallel systems to achieve required performance. Clustering and scheduling of tasks for parallel implementation is a well researched problem. Several techniques have been studied to improve performance and reduce problem execution times. In this thesis, a new clustering and scheduling scheme, called flexible clustering and scheduling (FCS) algorithm is proposed. It is a novel approach where clustering and scheduling of tasks can be tuned to achieve maximal speedup or efficiency. The proposed scheme is based on the relation between the costs of computation and communication of task clusters. Vital system parameters such as processor speed, number of processors, and communication bandwidth affect speedup and efficiency. Processor speed and communication bandwidth vary from system to system. Most clustering and scheduling strategies do not take into account the system parameters. The low complexity FCS algorithm can adapt itself to suit different parallel computing platforms and it can also be tuned to suit bounded or unbounded number of processors. The analytical, simulation and experimental studies presented in this thesis validate the claims.

Contents

List of Figures

List of Tables

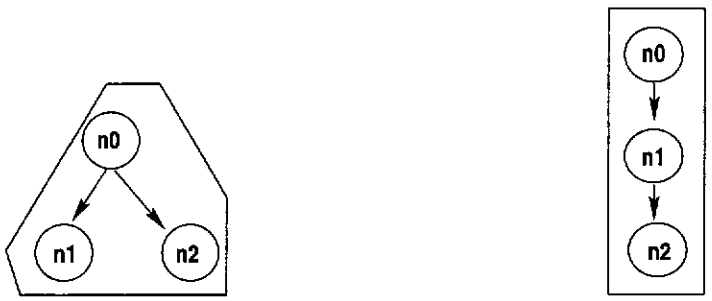
Chapter 1

Introduction

Multiple processors are used in parallel computers to improve performance. In parallel computing two or more processors are employed so that the given application problem is distributed among them to complete the execution within a given time.

The multiple instruction stream, multiple data stream (MIMD) architecture is the most popular architecture employed by parallel computers. In MIMD architectures, each processor has its own control unit and local memory unit. They execute independently and transfer information via interconnection networks by using message passing paradigm[1]. Data communication is carried out through messages. Each message consists of a number of fixed-size packets but the length of each message can be varied. The time to deliver fixed-size messages between any pair of processors is constant. In recent years, network of workstations (NOWs) are used for implementing application problems. The NOWs belong to the multiple program multiple data (MPMD) mode of parallel computers. Many software tools are available to support parallel processing on NOWs, for example, parallel virtual machine (PVM), message passing interface (MPI), local parallel multicomputers (LAM).

Clustering and scheduling techniques are used to enhance the performance of a parallel



(a) nonlinear clustering (b) linear clustering

Figure 1.1: Linear and nonlinear clustering

implementation. Before an application problem is implemented on a parallel processing system, the problem is analyzed to assess its suitability for execution on the system. Task graph is used to represent an application program, each node represents a set of instructions and an arc represents the communication cost between two nodes.

1.1 Clustering and Scheduling

The efficient execution of an application on a parallel system depends on many factors, for instance, the instructions representing a task, the size of cluster and the scheduling of tasks on processors. When an application problem is divided into several small tasks, the size and complexity of the tasks affect the total parallel execution time [2]. If the tasks are independent from each other, they can be distributed to several processors and the parallel execution time is small because there is no interference among processors. On the other hand, an application may comprise several tasks with communication links among tasks. The execution time of such an application may be high due to the costs of communication.

Clustering is a method to collect small tasks together and put them in the same cluster. The problem of finding the optimal clustering is NP-complete when the objective function is to minimize the parallel time, with unbounded number of processors [3]. Clustering can be divided into two types, linear and nonlinear [4]. In nonlinear, clus-

tering, there are two or more independent tasks in a cluster. As shown in Figure 1.1(a), nodes $n1$ and $n2$ are independent nodes and collected into the same cluster. Thus by using nonlinear clustering strategy, the parallelism of the program is reduced. From Figure 1.1(a), if $n0$ and $n1$ are executed on a processor and $n2$ is on another processor, then two processors can be employed to exploit parallelism. However, communication cost between $n0$ and $n2$ is considered to determine whether $n2$ should be executed on a new processor. For example, if the computation of all tasks is 4 and the communication cost between $n0$ and $n2$ is 2, $n2$ should be on another cluster. The parallel time for two processors is 10 while the serial time is 12. On the other hand, if the communication between $n0$ and $n1$, $n2$ should be on the same cluster as $n0$ because the parallel time with two processors is 14. Thus the communication cost among tasks affects the clustering decision. In linear clustering, dependent tasks are gathered into a cluster, as shown in Figure 1.1(b). Dependent nodes that are in a precedence path of the task graph are gathered into a cluster. From the Figure 1.1(b), nodes $n0, n1$ and $n2$ are executed sequentially, and collected into a cluster. In this case, the only way to get the best result is by gathering the three nodes into the same cluster. If any node is seperated to another cluster, the increase in parallel time is dependent on the cost of communication costs among the nodes. Therefore, gathering nodes into a cluster depends on the computation times of individual tasks and the communication costs among the tasks. Typically, all tasks in the same cluster are scheduled for execution on the same processor.

Tasks are optimally placed on processors of a particular parallel machine by using a scheduling mechanism. As shown in Figure 1.2, given a task graph consisting of four tasks and a target machine consisting of two processors, a scheduler attempts to obtain minimum total execution time[5, 6]. A scheduling algorithm determines node allocation to processors. Suppose that the scheduler allocates nodes of the critical path, which consists of $n0, n1$ and $n3$ onto a processor PE0 and $n2$ onto PE1. Gantt chart [3] is used to present the time each task spends in execution as well as the processor on which

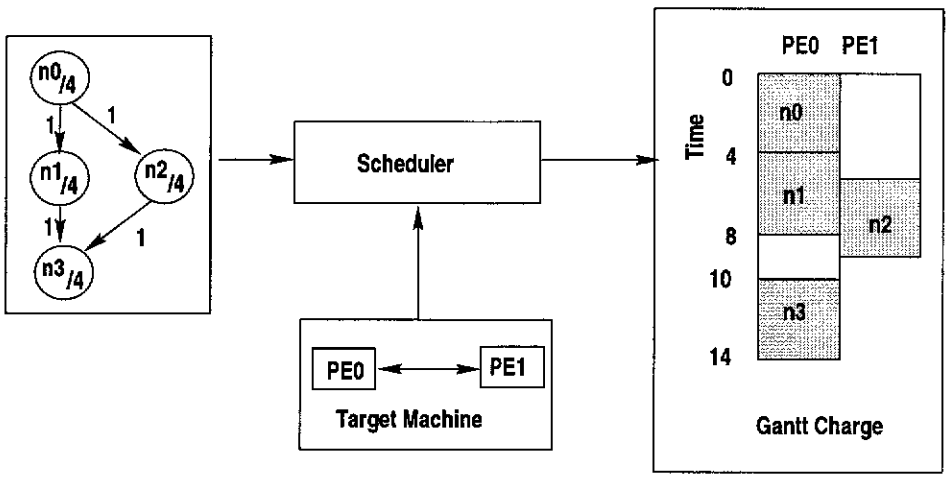


Figure 1.2: Scheduling process

it executes as shown in the Figure 1.2. Scheduling tasks for execution on processors is achieved by one of two approaches, static or dynamic. In static scheduling, tasks are assigned to processors before runtime either by programmers or compilers[1, 2, 3, 5, 6, 7, 8]. The static scheduling approach utilizes the knowledge of problem characteristics to reach a well balanced load[9].

In dynamic allocation approach, tasks are assigned to processors at runtime and dynamic scheduling performs scheduling activities concurrently[1, 9]. In contrast to static allocation, dynamic allocation algorithms employ complex strategies to balance the load among processors. However, using dynamic allocation approach, results in better utilization of processors than static scheduling in some problems.

The total execution time of an application problem consists of computation and communication times. The computation time of an application running on a processor can be reduced by distributing tasks to many processors for parallel execution. The communication time increases when tasks are distributed to many processors because of communication costs. To achieve low execution time, communication cost should be minimized. The parallel execution time is minimal if communication costs are minimized. The communication can be reduced by collecting nodes using linear clustering. When an application problem is executed on different systems, the parallel execution

time might be different because of varying communication bandwidths and processor speeds.

1.2 Performance

Analyzing performance of a multiprocessor system executing an application program is very complex [10]. Many factors, for instance, grain size of execution or cluster size, the processor speed and the communication bandwidth of the system, affect system performance. Parallelism inherent in an application problem is one of the factors affecting speedup and efficiency. After an application problem is divided into small subproblems or tasks, they are distributed to execute on many processors. In clustering schemes, small tasks are collected into clusters and allocated processors. If the size of each cluster is small, clusters are distributed to many processors with high degree of parallelism. However, overall execution time may increase due to overheads associated with context switching, scheduling time, and communication delays [5]. The total communication overhead among clusters can be reduced by increasing the cluster size by gathering many tasks into a cluster. However, this reduces degree of parallelism. The overall execution time increases because less number of processors are used and each processor executes many tasks. The computation speed of processors and communication bandwidth also affect the cluster size, speedup and efficiency.

When an application problem is executing on a system with high speed processor, it can complete earlier than it would on a low speed processor system. In other words, a high speed processor can execute more instructions than a low speed processor in a given time. Thus, for a system with a high speed processor, large number of tasks should be gathered into a cluster for allocation to a processor. Then the number of processors can be reduced and efficiency improved. Communication bandwidth is another factor that affects speedup and efficiency. In a system with high communication bandwidth, data can be transferred among processors rapidly. Thus when tasks are distributed to

many processors, the communication overheads are less than those in systems with low communication bandwidth. Then, the number of tasks in each cluster can be less and many clusters can be distributed to many processors. When tasks are distributed to many processors in a network with low communication bandwidth, computation time may be dominated by communication overheads. The computation speed of processors and communication bandwidth, hereafter referred to as system parameters, vary from system to system. The flexible clustering and scheduling (FCS) algorithm takes these factors into account for considering the cluster size and the number of processors used. The size of each cluster is determined by comparing with the ratio of computation and communication costs of the system on which an application problem is executed. Therefore, by adapting the cluster size to suit the number of processors used and the system parameters, high speedup and/or high efficiency can be achieved. Furthermore, the complexity of the FCS algorithm is lower than those of most existing algorithms.

1.3 Structure of the Thesis

Chapter 2 provides necessary background related to partitioning, clustering, and mapping. Before an application program is distributed to processors, the program is analyzed. The analysis breaks down the program into small tasks. Each task represents a set of commands which is normally unbreakable when allocated to a processor. When a task finishes, some outputs are sent for execution to another task: resulting in communication among tasks. The communication might be point-to-point, broadcast or scatter. Task graphs are used to display the tasks of an application problem and the relationship among the tasks.

Chapter 3 reviews the research work related to scheduling, clustering, and the relation between communication and computation costs. Tasks can be scheduled in two different ways: static and dynamic. In the case of static scheduling, tasks are assigned to processors before runtime. The processors know *a priori* which tasks to execute and

when. On the other hand, if tasks are scheduled dynamically, they are distributed among processors during the execution time. In this chapter, many existing static scheduling algorithms are discussed. Critical path, the shortest path from an entry node to an exit node, is used for scheduling in many of these algorithms. The critical path consists of nodes whose as-soon-as-possible (ASAP) and as-late-as possible (ALAP) times are equal. Thus these nodes are normally considered for scheduling and allocation first. Some algorithms allocate a single node to a processor at a time. After a node has been allocated, a new node is selected and allocated to an available processor. Some other algorithms collect nodes as a cluster and allocate all the nodes in the cluster to a processor. Flexible Clustering and Scheduling (FCS) algorithm considers the size of each cluster relating to the ratio of computation and communication costs.

Chapter 4 describes the FCS algorithm in detail. Most scheduling algorithms do not consider the type of computer system on which nodes are allocated. The capacity of processors and the communication speeds of different systems are not the same. Thus scheduling for different systems should be different. The FCS algorithm takes into account the capacity of each system, referred to as the system parameter. The size of each cluster is determined by comparing the ratio of computation to communication of each cluster to the system parameter. As we know the total time consists of computation and communication times. When tasks are distributed to many processors, communication among processors increases. Minimum parallel time occurs when tasks are allocated to many processor with minimum communication time. Therefore, the scheduling algorithm should minimize the communication cost as much as possible. The FCS algorithm allocates nodes that relate to each other, for example, parent, child, and sibling nodes, to the same processor. If such nodes are on different processors, the communication cost increases. To reduce the communication, they are gathered into the same cluster. Nodes that are in the same cluster are indivisible and allocated to the same processor. When they are in the same processor, the communication cost among them is reduced to zero. By doing so, the communication of the whole program

is reduced and the parallel time is minimized.

Processor speed affects number of nodes allocated to each processor. If processor speed is high, clusters with effectively high computation cost should be allocated to one processor. On the other hand, clusters with low computation costs are more suitable for low-speed processors. FCS algorithm can tune clustering mechanism according to the system to achieve maximal efficiency.

Chapter 5 presents the simulation results of many application problems, for instance, Gaussian elimination, Laplace equation, Mean value analysis, and Floyd Walshall's shortest path problem. The results show the relationships among system parameters, speedup, efficiency, and number of processors. Using FCS algorithm, the speedup and efficiency of the application problems executed on bounded and unbounded number of processors is better than that using other algorithms for most cases. By using the FCS algorithm, the number of processors used is minimal depending on the system parameter of the system for the unbounded number of processors. The number of processors is also dependent on the processor speed of the system. The number of processors used is less for a system with high speed processors.

Chapter 6 presents the experimental results for executing application problems: Gaussian elimination and Floyd Walshall's shortest path. The experimental results validate the simulation results presented in the previous chapters.

1.4 Contributions of this Thesis

In this thesis, flexible clustering and scheduling (FCS) is introduced. Application task graphs are clustered by considering many factors: the structure of application task graph, processor speed, communication bandwidths of the system on which application is executed, and the number of processors available. To reduce communication costs among tasks, tasks are gathered into a cluster by selecting a task and its relatives

together. Then the overall communication cost within a cluster is reduced. The size of each cluster is related to the processor speed and communication bandwidth of the system. The FCS algorithm determines the size of each cluster by comparing with the ratio of computation and communication bandwidth on which the application is executed. The size of each cluster is small if the application is executed on the system with low speed processors and many processors are used. On the other hand, a large cluster size is created when they are executed on high speed processors. When the size of each cluster is small, clusters are distributed to many processors to achieve the high speedup. Application problems, for example Gaussian elimination, Laplace equation, Mean value analysis, LU decomposition are used as example problems to assess the FCS algorithm. By using FCS algorithm, clustering and scheduling of these problem are studied with the different varying processor speeds and communication bandwidths. The results are compared with other algorithms and it is clear, that in most cases the FCS algorithm performs better. To support the simulation results, implementation on a closed network consisting of four processors is carried out to demonstrate the feasibility of the proposed scheme.

Chapter 2

Preliminaries

In this chapter, some terminology and basic concepts related to clustering, allocation, and scheduling and the relationship between computation and communication costs are introduced.

Before an application program is executed on a parallel processor system, it is divided into small tasks, distributed among processors, and executed concurrently. Directed acyclic graphs (DAGs) are used to represent the various tasks and interrelations among tasks in a program. In such a DAG, a node represents a task and an edge represents the communication between two tasks. The development of effective techniques for the distribution of tasks among processing elements (PEs) is one of the most challenging issues for researchers. The tasks may be scheduled before execution time, as in static scheduling or during execution time, as in dynamic scheduling.

2.1 Introduction

When an application problem is considered for execution on a multiprocessor system, users have to determine how to divide the problem into a set of small tasks, distribute

and execute them on many processors in parallel. According to Foster [11], the process to design and build parallel programs consists of four stages; partitioning, communication, agglomeration, and mapping. During partition, an application problem is divided into small tasks which include both the computation associated with the problem and the data on which the computation operates. There are two different approaches for partitioning at the first stage: *functional decomposition* and *domain decomposition*[11, 5]. If *functional decomposition* is selected, the initial focus is on the computation that is to be performed, rather than on the data manipulated by the computation. Initially, the application program is decomposed into small tasks based on functionality. After decomposition, the application data set is divided into small sets based on data associated with each task. The data may be disjoined or overlapped with data associated with other tasks. In the *domain decomposition* approach, data associated with the application is considered for partition first, divided into small pieces of approximately equal size, then, the computation associated with each set of the divided data forms one small task or each set of data is associated with one small task.

Tasks generated by the first stage are intended to be executed concurrently but this may not be possible because the data required may be on other processors. The computation to be performed by one task might require data from other tasks executing on another processor. Data must then be transferred between the tasks so as to allow computation to proceed. If tasks are divided into small pieces, many communication links will be required. A task graph can be employed to illustrate the relations and the communication among tasks [4, 12, 13, 14, 15, 16, 17]. A detailed investigation of the effect of communication costs on overall performance is discussed in Chapter 3.

The next stage is agglomeration or clustering. Small tasks are gathered into clusters for allocation and mapping onto processors. The resulting algorithm may be highly inefficient if the number of tasks is more than the number of processors on the target computer [11] and the computer has not been designed for efficient execution of small tasks. One critical issue influencing parallel performance is the cost of communication.

Clearly, performance improvement can be achieved by reducing communication overheads. Each communication incurs not only a cost proportional to the amount of data transferred, but also a fixed startup cost.

A mapping algorithm is developed in order to minimize total execution time. Tasks are mapped onto appropriate processors to reduce communication and also to increase efficiency. Two mapping strategies with contrasting objectives are used.

i) Tasks that are able to execute concurrently are mapped onto different processors so as to enhance concurrency.

ii) Tasks that communicate frequently are placed on the same processor, so as to increase locality.

According to Lo [18] the major question is how to distribute tasks among processing elements (PEs), to achieve the following goals:

i) Minimize execution time by reducing communication delays and interprocessor communication costs.

ii) Achieve good load balance among the processors

iii) Achieve a high degree of parallelism

iv) Achieve efficient utilization of system resources in general.

Many algorithms have been presented in order to get high performance or efficiency. Some are reviewed in Chapter 3.

2.2 Partitioning

Partitioning consists of two stages; in the first stage the given problem is broken into independent subproblems or tasks of almost equal size and during the second stage the subproblems are solved concurrently [5, 19]. The issues to be concerned in scheduling are the granularity of tasks and the physical distribution of ready tasks throughout the system [20]. The efficient execution of parallel programs depends on the effective partitioning of problems and the scheduling of tasks on a set of processors. Factors that affect performance are number of clusters, number of processors used, and the amount of communication among tasks.

2.2.1 Number of Clusters

The size and complexity of the tasks affect the parallel execution time of application programs. When an application program is partitioned into tasks, the important question is whether the amount of work packaged within a task is large enough to justify it being allocated to a processor. The granularity of a task is defined as the ratio of the task's computation time over its communication time[21]. A task is a fine-grain task if the ratio is relatively low which means that the computation is low when compared with the communication. Fine-grain task implies high degree of parallelism and high amount of overhead. On the other hand, a task is a coarse-grain task if the ratio is relatively high. The computation of the coarse-grain task is high compared with the communication, which implies low parallelism and less communication overhead. According to Cornard and Trystram [8] the term *fine granularity* is used if the number of processors is greater than or equal to the number of data items, otherwise *coarse granularity* is the term used. The relationship between the number of processors and the number of clusters is also important.

2.2.2 Number of Processors and Execution Time

The number of processors used is also an important issue. Maximum performance will be obtained if all PEs are busy all the time or there is no idle processor during the execution process. The addition of more PEs may not improve the performance because of overheads. A given problem may be solved by using different numbers of processors. Large number of processors is required to solve a problem in optimal time. Efficiency increases when the number of processors used is minimal and the execution time is also minimal. The question is how can a problem be solved with an optimal number of processors and in minimal time. The problem can be executed in minimal time if we reduce communication overheads and if all processors are busy executing useful tasks all the time.

2.3 Overheads

When an application program is executed on a multiprocessor system, the parallel execution time might not be better than that executed on a single processor system because of increased communication overheads. Some factors that affect the performance or efficiency of the system are communication, synchronization, and operating system task management overheads.

2.3.1 Communication Overhead

Communication overhead occurs when a task on a processor requires data from a task which is on another processor. In the case of parallel and distributed processor systems, the data or message is transferred from a sending to a receiving processor. The time required for sending and receiving data is known as communication overhead or communication cost. Many patterns of communication are considered, for instance,

point-to-point, broadcasting, and scattering.

Point-to-point communication

This is the most basic communication operation when a processor sends a single message to another processor. The transmitting time is defined as $t_s + t_w ml$ where t_s is the startup time, t_w is per-word transfer time or transmission rate, m is the number of words in a message, and l is the number of links traversed by the message [22].

The startup time is defined as the time required to handle a message at the sending processor. Before a message is transmitted, some information is added, for instance, header, trailer and error correction information. The startup time includes the time needed to prepare the message, to execute the routing algorithm, and to establish an interface between the local processor and the router. This delay incurs only once for a single message transfer. If m is large or predominant as in the case of high-speed machines such as vector multi-processors, or machines on which communication is managed by software, t_s, can be negligible. Per-word transfer or transmission rate is defined as the time required to traverse a word over a link[8]. If the channel bandwidth is r words per second, each word takes time $t_w = 1/r$ to traverse the link. The way processors are connected affects the number of links (l) [22]. If p number of processors are connected as a ring, the number of links is $\lfloor p/2 \rfloor$ for the shortest path between two processors, $2\lfloor \sqrt{p}/2 \rfloor$ for a wraparound square mesh, and $log\ p$ for a hypercube. A factor that can have a significant impact on communication performance is competition for bandwidth. Two processors may need to send data over the same wire at the same time. Typically, only one message can be transmitted at a time, while the other message is delayed [11].

Broadcasting and Scattering

Broadcasting is used in a variety of linear algebraic algorithms such as matrix-vetor multiplication, matrix-matrix multiplication, Gaussian elimination and LU-factorization.

Many types of broadcasting techniques are employed in parallel processing such as one-to-all broadcasting, one-to-all personalized communication, all-to-all broadcasting, and all-to-all personalized communication [23]. One-to-all or Single-node broadcasting occurs when a single node or processor is requireed to send identical data to all other nodes or to a subset. Data which is required by other processors is on the source or sending processor. It will be copied and sent to target or receiving processors. One-to-all personalized communication or *scattering* occurs when a single node sends distinct data to distinct nodes. Conversely, when all distinct nodes send distinct data to a single node it is called *gathering*. All-to-all broadcasting or *expand* occurs where all nodes perform one-to-all broadcasting. All-to-all personalized communication is when all nodes perform one-to-all personalized communication. Two communication models are involved in broadcasting, communication on one port at a time (one-port communication) and concurrent communication on all ports (all-port communication). The communication overheads that affect clustering and scheduling are discussed in Chapter 3.

2.3.2 Synchronization Overhead

Interaction among parallel tasks takes the forms of cooperation and competition [1]. This leads to the need for coordination or synchronization in order to facilitate sequence control and to manage shared resources by providing access control. A process has to wait for an input from another process finishing its task. In shared memory parallel systems, when two processes share the same variable, synchronization will take care of the order to access the data. A process writes or changes the data before another takes the data as an input. Barriers are used to synchronize the execution of a group of processes as they are blocked at the barrier synchronization point until every member's process reaches this point. A barrier is a simple way to ensure that messages generated in the two phases do not get mixed. If many barrier points are used during the execution of a program employing many processors, the number of barrier points affects the

speedup and efficiency of the program. Some processors stay idle at a barrier point waiting for *ready state* to execute the next process.

2.4 Directed Acyclic Task Graph

A graph is used to represent a set of tasks, nodes of a graph represent separated tasks of an application program, edges represent the communication among tasks, and the direction of edges describes data movement [13, 14, 15, 16, 17, 24]. A task is an indivisible unit of computation which may be an assignment statement, a subroutine or even an entire program [15]. A *task graph* can be defined as a tuple $G = (V, E)$ where $V = \{n_j, j = 1 : v\}$ is the set of nodes, $v = |V|$ is the number of nodes, and E is the set of communication edges. An edge between nodes u and v is represented by $e(u, v)$. Task scheduling can be considered as preemptive and nonpreemptive schedulings [10, 25]. In the case of nonpreemptive scheduling once a task starts execution on a processor, it cannot interrupted by another task. Another task can start on the same processor when the previous one finishes. On the other hand, if a task can be interrupted during execution and resumed back to the same state, it is called preemptive scheduling. In this thesis, the nonpreemptive is applied. In the following subsection, we describe creation of a task graph from a given parallel program for the Gaussian elimination algorithm. Gaussian elimination is a standard method for solving the linear equation system of $Ax = b$ [22, 26, 27, 28].

2.4.1 Gaussian Elimination Algorithm and its Task Graph

An algorithm can be divided into small tasks in many different patterns. In Figure 2.1 , the sequential algorithm for the Gaussian elimination problem is shown [22]. Statements 7 and 8 of the algorithm can be considered as one task. However, if we want to have coarse tasks, statements $6, 7, 8, 9$, and 10 can be grouped into a task. A task T_i^k

```
Procedure Gaussian elimination Algorithm{
1   for(k = 0; k < n; k++){
2    for(j = k + 1; j < n; j++)
3      A[k][j] = A[k][j]/A[k][k];
4    y[k] = b[k]/A[k][k];
5    A[k][k] = 1;
6    for(i = k + 1; i < n; i++) {
7      for(j = k + 1; j < n; j++)
8        A[i][j] = A[i][j] − A[i][k] * A[k][j];
9      b[i] = b[i] − A[i][k] * y[k];
10     A[i, k] = 0;
       }
     }
}
```

Figure 2.1: Gaussian elimination algorithm

corresponds to,

$$\begin{aligned}
&\textbf{for}(j = k + 1; j < n; j++)\\
&\quad A[k][j] = A[k][j]/A[k][k];\\
&y[k] = b[k]/A[k][k];\\
&A[k][k] = 1;
\end{aligned}$$

The tasks of the first set, $T_1^1, T_2^2, T_3^3, T_4^4$ as shown in Figure 2.2, compute a single loop of j between $k+1$ and n where n is the size of the problem. Outputs from statements 3, 4 and 5 are respectively the resultant value for $A[k][j]$, $y[k]$ and $A[k][k]$. Then $A[k][j]$ is transferred to nodes of another set, (T_{ij}^k), that computes these statements;

$$\begin{aligned}
&\textbf{for}(j = k + 1; j < n; j++)\\
&\quad A[i][j] = A[i][j] - A[i][k] * A[k][j];\\
&A[i][k] = 0;
\end{aligned}$$

Each node of this set computes a single loop of j between $k+1$ and n by using data

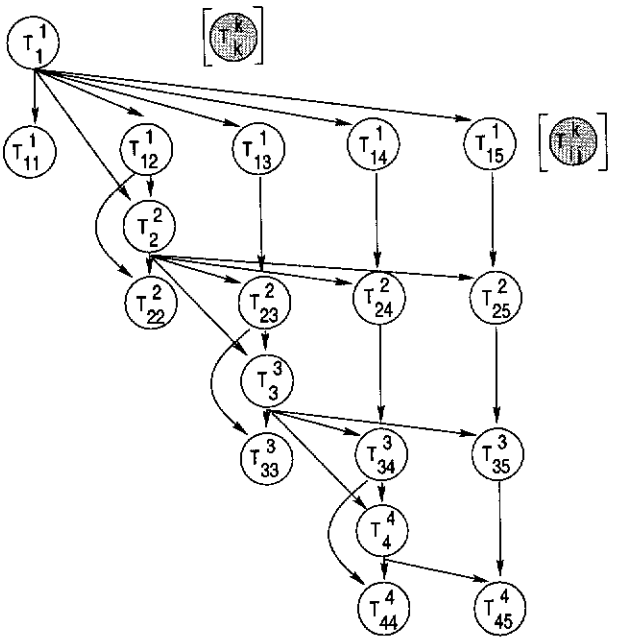


Figure 2.2: Gaussian elimination task graph

$A[i][k]$, received from the previous set of tasks and sets $A[i][k]$ to zero. There is a set of nodes in this group that computes only one statement, $b[i] = b[i] - A[i][k] * y[k]$. The final Gaussian elimination task graph for a problem of size 4 is shown in Figure 2.2.

In our model, once started, tasks run to completion without interruption. The size of a task depends on the number of statements or instructions making up the task. The computation load of a task is represented by r time units. A task begins execution upon the arrival of all data from its predecessors in the DAG. The communication overhead associated with the arrival of input data is defined as the communication load of the task and is represented by c time units. The value of c depends on the indegree of the task and the weight of each incoming edge. The computation to communication ratio of the task is represented by $\alpha = r/c$. We represent the cost of computation of node x by $r(x)$ and the cost of communication associated with that node by $c(x)$. A high value of α implies coarse granularity and a low value of α implies fine granularity.

The symbols used in this thesis are listed below.

r = number of program statements (or instructions) in a basic task

c = number of message packets to be transmitted for a basic task

$\alpha = \frac{r}{c}$ = ratio of computation to communication of a task

R = number of program statements (or instructions) in a cluster

C = number of message packets to be transmitted for a basic cluster

$\beta = \frac{R}{C}$ = ratio of computation to communication of a cluster

ρ = a function of the processor throughput

γ = a function of channel bandwidth for message passing systems

$\tau = k \cdot \rho/\gamma$, τ is the system parameter and k is a constant

$p(x)$ = parent node of x

$p^i(x)$ = the i^{th} parent of x that has not been clustered

$s(x)$ = the sibling node of x

$s^i(x)$ = i^{th} sibling of x

$h(x)$ = the child node of x

$h^i(x)$ = i^{th} successor of $h(x)$

$p(s(x))$ = sibling's parent

$p^i(s(x))$ = sibling's i^{th} parent

$n(x)$ = next node in the lexicographic list

$cl(n_1, n_2, ..., n_v)$ = a cluster consisting of nodes $n_1, n_2, ..., n_v$.

$st(x)$ = starting time of node x

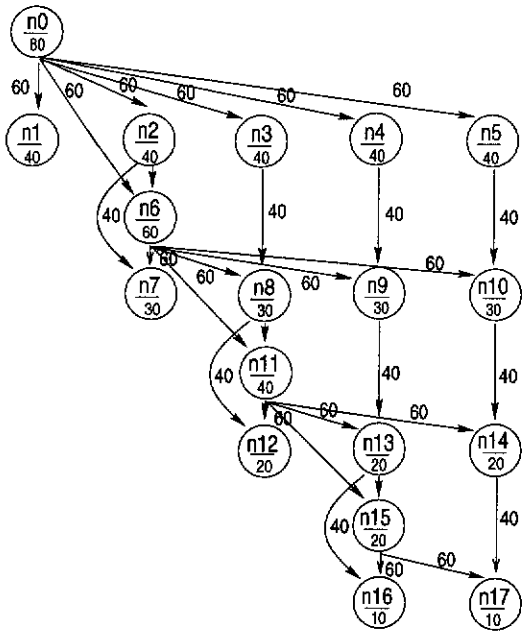


Figure 2.3: Gaussian elimination task graph

$st(cl(x))$ = starting time of cluster $cl(x)$

$L(x)$ = List of relatives of x in order of $p(x), h(x), h^2(x), ..., h^u(x), s(x), s^2(x), ..., s^v(x)$, $p(s(x)), p^2(s(x)), ..., p^w(s(x))$

2.5 Critical Path

A task graph's *critical path* is defined as the path from an entry node to an exit node whose sum of computation costs and communication costs is maximum [1, 3, 7, 8, 13, 14]. A task graph might have more than one *critical path.* When nodes in the *critical path* are allocated on a processor, the communication costs among the nodes are reduced to zero and a new *critical path* of the task graph is generated [13, 29]. The length of the *critical path* is the minimum time required for program execution [1, 13].

Nodes which are in the *critical path* are selected by computing their *As-Soon-As-Possible* (ASAP) and *As-Late-As-Possible* (ALAP) times. ASAP and ALAP terms were introduced in mobility directed (MD) scheduling algorithms [13]. *ASAP* is the

earliest possible execution time which is created by moving forward through the task graph. The $ASAP$ of each node is computed by adding the $ASAP$ of the immediate predecessor nodes and the edge between the two nodes. $ASAP$ is formally described as,

$$Maximum_{1 \leq i \leq v}\{ASAP(p^i(x)) + r(p^i(x)) + c(p^i(x), x)\}$$

where $r(p^i(x))$ is computation cost of i^{th} parent node of x, $c(p^i(x), x)$ is the communication cost between nodes x and $p(x)$, and v is the number of parent nodes of x. For example, the ASAP of node $n0$ in Figure 2.3 is zero because $p(n0)$ does not exist, therefore, $ASAP(p^i(n0))$ is zero and so are $r(p^i(n0))$ and $c(p^i(n0), n0)$. Node $n0$ can start execution when the program begins execution. Node $n1$ starts execution after $n0$ has finished its execution, a time of $r(n0)$, plus the time for sending data to $n1$ with the communication time of $c(n0, n1)$. In case a node, x, has more than one input, its $ASAP$ is given by $Maximum_{1 \leq i \leq 3} ASAP(p^i(x)) + r(p^i(x)) + c(p^i(x), x)$ where $p^i(x))$ is a parent. For example, node $n6$ of the task graph in Figure 2.3, has two parent nodes, $n0$ and $n2$. The result of $ASAP(n0)+c(n0, n6)$ is 140 and that of $ASAP(n2)+c(n2, n6)$ is 220. Therefore the $ASAP$ of $n6$ is 220. $ASAP$ times for all the nodes of the task graph are shown in Table 2.1.

The Critical path length (CPL) of the DAG is calculated prior to computation of the $ALAP$ of nodes. The CPL is formally described as, $Max(ASAP(x) + r(x))$, where x is an exit node in the task graph. The node whose $ASAP$ + *computation cost* is maximum is considered as the last node in the critical path. The $ALAP$ of an exit node x, of a task graph is defined as $CPL - r(x)$. Conversely, the $ALAP$ is obtained by moving backward through the graph from exit nodes to the entry nodes. $ALAP$ is formally described as,

$$Min_{1 \leq i \leq u}\{ALAP(h^i(x)) - c(x, h^i(x)) - r(x))\}$$

where, $c(x, h^i(x))$ is the communication cost between nodes x and $h^i(x)$, and u is the

Table 2.1: ASAP and ALAP of nodes in GE task graph

Nodes	*ASAP*	*ALAP*	*Difference*	*Nodes*	*ASAP*	*ALAP*	*Difference*
n0	0	0	0	*n9*	340	440	100
n1	140	620	480	*n10*	340	520	180
n2	140	140	0	*n11*	410	410	0
n3	140	260	120	*n12*	510	640	130
n4	140	360	220	*n13*	510	510	0
n5	140	440	300	*n14*	510	590	80
n6	220	220	0	*n15*	570	570	0
n7	340	630	290	*n16*	650	650	0
n8	340	340	0	*n17*	650	650	0

number of child nodes of x. The computation and communication costs of Gaussian elimination task graph are assigned as shown in Figure 2.3 [13]. The $ASAP$ of nodes in the task graph are computed first starting from node $n0$. After all nodes are assigned $ASAP$, the CPL of the graph is computed by selecting the node whose ($ASAP$ + *computation cost*) is maximum. In this case the nodes $n16$ and $n17$ have a $CPL = 660$.

Nodes for which $ASAP = ALAP$ are in the *critical path*. For example, node $n0$, $n2$, $n6$, $n8$, $n11$, $n13$, $n15$, $n16$ and $n17$ are nodes in the critical path. However, a given task graph may have more than one *critical path*.

The *critical path* is used for clustering and scheduling in many algorithms, for instance, KB/L [29, 15, 4], MCP [13], MD [13] and DCP [14]. Details of these algorithms are reviewed in Chapter 3.

Many researchers are working on determining how to distribute tasks to processors effectively with high efficiency and low communication cost. The flexible clustering and allocation algorithm uses the critical path for finding the first cluster and dynamically gathers tasks into clusters based on the relation between computation and communication cost of clusters and the system capacity.

2.6 Conclusion

In this chapter, general topics that relate to the scheduling and clustering were reviewed, including partitioning, communication overheads, critical path, and terminology. In the next chapter, a review of research work that relates to scheduling and clustering and a discussion of computation and communication cost affecting the performance and efficiency of parallel computation, will be discussed.

Chapter 3

Related Work

The last chapter, discussed the background and preliminaries related to *partitioning, critical path,* and *overheads* that affect the execution time of a program using a parallel processor system. This chapter presents the research work carried out in the area of scheduling, clustering, and the relation between communication and computation costs.

Scheduling and *Clustering* are critical issues in parallel processing. An application program is divided into small tasks or partitions before it is executed using parallel processors. A task graph is used to represent the processes and communication of an application program; nodes in the graph represent tasks or partitions and arcs represent communication among the tasks. The distribution of tasks among processors is called task allocation. When the communication overheads among processors are very high, for instance, in network of workstations (NOWs), small tasks are collected forming a cluster in order to reduce communication cost among tasks. Typically, all tasks which are on the same cluster are allocated to the same processor. The communication cost among tasks on the same processor is zero because tasks write and retrieve messages from the same memory unit. Reviews of several scheduling and clustering algorithms will be presented in this chapter.

3.1 Scheduling

Scheduling is the process of assigning tasks or nodes to execute on an appropriate processor in a multiprocessor system [1, 2, 3, 5, 6]. The goals of scheduling are to spread tasks to all processors as evenly as possible in order to obtain processor efficiency and to minimize communication among processors, which leads to minimal processing time [1, 2, 30]. Task distribution is important, not only for the execution of application programs on multiprocessor systems, but also to exploit the computer architecture.[31]. The terms *scheduling* and *allocation* are used interchangeably. They are used to describe the same general mechanism, but from different viewpoints. According to Casavant et al.[7, 32], there are two points of view, resource management and consumers. The *allocation* is used in the view of resource management. Tasks are allocated to resources provided by a system. The term *scheduling* is used from a consumers or user's point of view when tasks are allocated to resources. The scheduling methods can be classified into two major groups: static and dynamic scheduling. They are different in terms of time at which the scheduling or assignment decisions are made. In static scheduling, tasks are assigned to processors before the application program execution begins or at compile time and remain fixed throughout the execution process [32, 2, 3, 20]. For dynamic scheduling, the decision as to which tasks are scheduled is made at execution time [2, 20]. Static scheduling can be used to predict the speedup that can be achieved by a particular parallel algorithm on a target machine, assuming no preemption of processes.

3.1.1 Static Scheduling

In static scheduling, tasks are assigned to processors before run time. A scheduling algorithm should consider the task graph structure, task granularity, and arbitrary computation and communication costs [2]. To describe how tasks are scheduled, the program is described in terms of tasks and is represented by a DAG [13, 14, 15, 24, 31,

33, 34], or network flow model. In this thesis, DAGs are used to represent tasks and their communications.

Tasks are assigned priorities and whenever they contend for processors, the one with the highest priority is always scheduled first. Wu et al. [13] use *mobility* assigned to tasks as the priority. Yang and Gerasoulis [4, 15, 24] use the sum of *tlevel* and *blevel* as the priority of each node, where *tlevel* of node x is the length of the longest path from an entry node to x, excluding the weight of x, and *blevel* is the length of the longest path from x to an exit node. In terms of tasks, what we need to consider for a schedule are their task execution times, task dependencies, task communication times and synchronization [14]. In static scheduling, there are two basic methods; in the first, tasks are allowed to duplicate among processors and in the other, task duplication is not allowed.

Tasks are duplicated and allocated to the same processor as relative nodes in order to reduce communication costs between a duplicated task and their relative nodes[35, 36]. When the sending node is duplicated and allocated to the same processor as the receiving node, the communication cost between the nodes is zero. Darbha and Agrawal [35] present an algorithm called Task Duplication based Scheduling (TDS) in which the predecessor of a node x can be duplicated and allocated to the same processor as x. The drawbacks of duplication occur when the problem size is large. A large number of resources are required, for instance, more memory addresses are used to accommodate data related to duplicated tasks. Then the execution times are required more for duplication process of both task and data. Some scheduling algorithms use the critical path as a key to select nodes for scheduling and some select nodes and form clusters by using the relationship among nodes to reduce communication overheads.

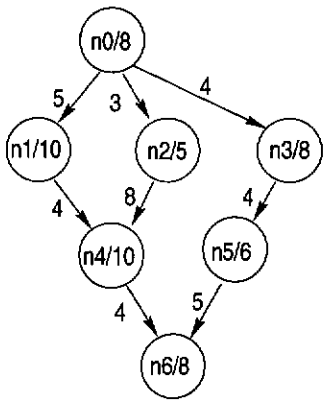


Figure 3.1: A directed acyclic task graph

Table 3.1: ASAP and ALAP of nodes in Figure 3.1

Nodes	*ASAP*	*ALAP*	*Relative mobility*	*Nodes*	*ASAP*	*ALAP*	*Relative mobility*
n0	0	0	0	*n4*	27	27	0
n1	13	13	0	*n5*	24	30	1
n2	11	14	3/5	*n6*	41	41	0
n3	12	18	6/8				

3.1.1.1 Algorithms using Critical Path for Scheduling

Many algorithms use the critical path of task graphs to select nodes for scheduling, for instance, Kim and Brown (KB/L)[4, 15, 29], Modified Critical Path (MCP) [13], Mobility Directed (MD) [13] and Dynamic Critical Path (DCP) [14] algorithms. The method for computing the critical path was discussed in Chapter 2. Kim and Browne used the critical path for a clustering algorithm called the KB/L algorithm [4, 15, 29]. The KB/L algorithm determines the critical path from *unexamined nodes* and forms a cluster by using the nodes in the critical path. Then the nodes are marked as *examined nodes* and the algorithm executes in a recursive fashion to determine the CP of the unexamined nodes.

Wu et al.[13] introduced Modified Critical-Path (MCP) and Mobility Directed (MD) scheduling algorithms using the critical path as the key for scheduling. The MCP

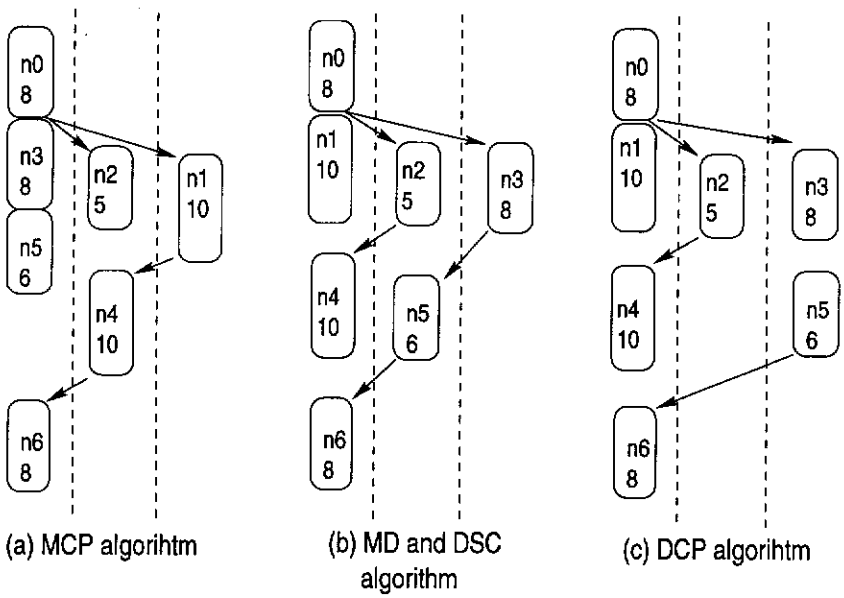


Figure 3.2: A comparison of scheduling algorithms

algorithm schedules nodes in a task graph based on $ALAP$ time. The $ASAP$ and $ALAP$ of every node are computed. Each node, ni, forms a list consisting of $ALAP$'s of ni and its descendants in decreasing order. These lists are sorted in decreasing order lexicographically and a new list is generated according to previous sorted lists. Nodes of the new list are removed and scheduled to the processor that allows for the earliest starting time. For example, the $ASAP$ and $ALAP$ of nodes in Figure 3.1 are computed and displayed in Table 3.1. Each node forms a list consisting of $ASAP$'s of the node itself and its descendants, for instance, node $n1$ forms a list consisting of $41(n6), 27(n4), 13(n1)$. Finally, a list is created by sorting the previous generated list. The members of the list are in the order $n6, n5, n4, n3, n2, n1, n0$. After each node is allocated, the final result of scheduling using MCP algorithm is shown in Figure 3.2(a). Nodes in a task graph scheduled by using this algorithm are pulled down by using the value of $ALAP$ as the index and are selected for scheduling from the bottom to the top of the task graph. The communication cost among nodes after a node is scheduled is not considered as the task graph is not modified during the scheduling process. The complexity of the MCP algorithm is $O(v^2 logv)$, where v is the number of nodes in the task graph[14].

The MD algorithm is also based on the critical path. It computes $ASAP$, $ALAP$ and an attribute called *relative mobility*. The *relative mobility* of a node, x, is defined as $(ALAP(x) - ASAP(x))/r(x)$. The algorithm starts by computing $ASAP$, $ALAP$ and *relative mobility* for all nodes. Nodes whose *relative mobility* is minimal are placed in the list L. A selected node for scheduling is the node in the list L with the constraint that the predecessor of the selected node is not in L. Actually, the selected node is a node in the critical path because the difference of its $ALAP$ and $ASAP$ are zero and the first node of list L is an entry node. However, when a node is scheduled and allocated to a processor, the task graph is modified because the communication among nodes on the same processor is set to zero. The critical path of the graph might be changed. For example, from Table 3.1, and Figure 3.1, nodes for which *relative mobilities* are minimum and form list L are $n0, n1$, $n4$ and $n6$. Node $n0$ is selected for scheduling first, because it is an entry node and it has no precedent on the list L. The selected node is scheduled to the first processor which has a large enough time slot to accommodate it without considering the minimization of the node' s start time. After the selected node is allocated to a processor, the task graph is modified because communication costs among tasks that are on the same processor are set to zero. The selected node's $ASAP$, $ALAP$ and *relative mobilities* of remaining nodes are recomputed. For example, when $n0$ and $n1$ are scheduled, the remaining nodes' $ASAP$, $ALAP$ and *relative mobilities* are recomputed as shown in Table 3.2. The critical path of the modified graph is changed. Node $n2$ is changed to be in the critical path and selected for scheduling in the next step. The final result of scheduling using MD algorithm is shown in Figure 3.2(b). In the MD algorithm, communication cost after a node is scheduled is considered. When a node is scheduled on a processor, the communication costs between the node and other nodes that are on the same processor are set to zero.

Table 3.2: ASAP and ALAP of nodes for MD algorithm

Nodes	*ASAP*	*ALAP*	*Relative mobility*	*Nodes*	*ASAP*	*ALAP*	*Relative mobility*
n0	0	-	0	*n4*	24	24	0
n1	8	-	-	*n5*	24	27	3/6
n2	11	11	0	*n6*	38	38	0
n3	12	15	3/8				

Kwok and Ahmad present Dynamic Critical-Path (DCP) [14] which also uses the critical path to describe the algorithm. The way to find the critical path of this algorithm is the same as that in MCP and MD algorithms. The DCP algorithm uses the attributes *Absolute Earliest Possible Start Time*($AEST$), *Absolute Latest Possible Start Time*($ALST$) instead of $ASAP$ and $ALAP$ respectively. The algorithm starts with calculating the $AEST$, $ALST$ and the difference between $AEST$ and $ALST$ for every node in the task graph. A node that has the smallest difference between $AEST$ and $ALST$ is selected for scheduling first under the constraint that the unscheduled parents of the selected node are not on the critical path. Normally, the selected node is one of the nodes which is on the critical path because the values of $AEST$ and $ALST$ are equal. The critical path changes dynamically whenever the cost of communication between a previous unscheduled node and other nodes allocated to the same processor is reduced. Hence Kwok and Ahmad call it the *dynamic critical path.* If the selected node, x, is on the *dynamic critical path*, processor hosting x's parents is considered for allocation first. If there is a large enough time slot for a node x with minimum starting time then that node x is scheduled on the processor. Otherwise, processor hosting x's child nodes or a new processor is considered. After the communication cost is set to zero, the *dynamic critical path* of the modified task graph is changed. Then an unscheduled node is selected from the new *dynamic critical path* for scheduling. This process continues until all nodes are scheduled.

When DCP is applied for scheduling a task graph, the first step is computing the AEST and ALST for all nodes as shown in Table 3.3. A node selected for scheduling first is

Table 3.3: AEST and ALST of nodes in Figure 3.1

Nodes	*AEST*	*ALST*	*Difference*	*Nodes*	*AEST*	*ALST*	*Difference*
n0	0	0	0	*n4*	27	27	0
n1	13	13	0	*n5*	24	30	6
n2	11	14	3	*n6*	41	41	0
n3	12	18	6				

Table 3.4: AEST and ALST of nodes after $n0$ and $n1$ are scheduled

Nodes	*AEST*	*ALST*	*Difference*	*Nodes*	*AEST*	*ALST*	*Difference*
n0	0	-	-	*n4*	24	24	0
n1	8	-	-	*n5*	24	27	3
n2	11	11	0	*n6*	38	38	0
n3	12	15	3				

$n0$ whose difference is zero and there is no parent on the critical path. After $n0$ is scheduled to a processor, the *AEST* and *ALST* of all nodes are recalculated. Node $n1$ is selected for scheduling in the next step. Being a node on the critical path, node $n1$ selects a processor that its parent, $n0$, allocated first. In this case, node $n1$ can be scheduled on the same processor as $n0$. After the communication cost between $n0$ and $n1$ is set to zero because they are on the same processor, the task graph is modified. *AEST* and *ALST* are recomputed as shown in Table 3.4. The final scheduling of nodes in Figure 3.1 using DCP algorithm is shown in Figure 3.2(c).

3.1.2 Dynamic Scheduling

In dynamic task allocation, tasks are distributed among processors during the execution time, with the aim of balancing the load among idle and busy processors [31]. There are two approaches of dynamic allocation, distributed and centralized [1]. In the case of distributed allocation, tasks are in a pool and any free processor can take these tasks. On the other case, tasks are allocated by central control.

Sih and Lee [37] introduced the Dynamic Level Scheduling (DLS) algorithm, which uses a parameter called Dynamic Level (DL) as the key to allocate nodes to processors.

The DL is computed from the difference between static level (SL) and the start time (ST(n_i,J)) of node n_i on processor J. SL(n_i) is defined as the maximum sum of computation costs along a path from node n_i to an exit node. ST(n_i,J) is the start time of n_i on processor J. Therefore, the DL is defined as SL(n_i) - ST(n_i,J). A node is allocated to the processor that constitutes the largest DL. This method performs exhaustive pair matching of nodes to processors at each step to find the highest priority node.

3.2 Clustering

Clustering can be defined as a process of gathering nodes of a given task graph onto a labeling cluster or a process of mapping of the tasks onto clusters [38, 39, 24, 15, 4, 40, 41, 38]. In a distributed memory parallel system, tasks mapped on different processors communicate solely via message-passing [38]. In existing parallel machines, for instance, network of workstation (NOWs), the cost of message-passing overhead is quite high. To avoid performance degradation, one solution is obtained by clustering fine grain tasks or small single tasks into single coarse grain tasks in order to reduce communication overheads. A task in each cluster is permitted to communicate with other tasks of other clusters immediately after finishing its execution. All the tasks that belong to the same cluster execute on the same processor. We assume that the communication costs among tasks which are in the same cluster are zero because all tasks write and retrieve data from the same memory unit. Many researchers have studied and presented clustering algorithms.

Dominant Sequence Clustering (DSC) algorithm was presented by Yang and Gerasoulis [24, 15, 4]. The algorithm uses attributes called *tlevel* and *blevel* assigned to every node of a task graph. According to Yang and Gerasoulis, *tlevel* of node x is defined as the length of the longest path from an entry node to x, excluding the weight of x or $tlevel(x) = maxtlevel(p(x)) + r(p(x)) + c(x, p(x))$. Symmetrically, the *blevel* of x is defined as the length of the longest path from x to an exit node. For instance, the *tlevel*

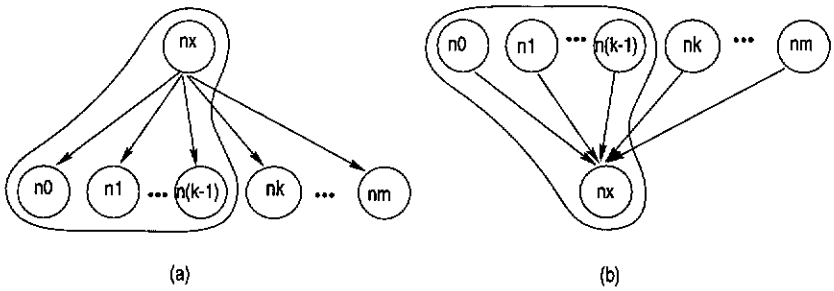


Figure 3.3: Clustering intree and outtree task graph using *DSC* algorithm

Table 3.5: *tlevel* and *blevel* of nodes for DSC algorithm

Nodes	*tlevel*	*blevel*	*Priority*	*Nodes*	*tlevel*	*blevel*	*Priority*
n0	0	49	49	*n4*	27	22	49
n1	13	36	49	*n5*	24	19	43
n2	11	35	46	*n6*	41	8	49
n3	12	31	43				

of $n4$ of the task graph in Figure 3.1 is 27 which is similar to the *ASAP* of MCP and MD algorithms or *AEST* of DCP algorithm. The *blevel* of $n4$ is 22. The *tlevel* and *blevel* of all nodes of task graph in Figure 3.1 is shown in Table 3.5.

The *priority* of a node x is computed from the sum of *tlevel* and *blevel.* The node whose *priority* is maximum and status free is selected for clustering. The status of a node is marked free if its parent nodes are all clustered and *partial free* if there are some parent nodes still unselected for clustering. For instance, if $n0$ and $n1$ in Figure 3.1 are clustered, node $n4$'s status is *partial free* because $n2$ has not been clustered. The selected node x is placed in the same cluster as that of its parent node which has been allocated before if this allocation reduces parallel time. Otherwise, the selected node forms a unit cluster consisting of only one node. For example, in the first step, node $n0$ is selected for clustering as it has the highest *priority* to form a unit cluster. Next, $n1$ is selected and joined with $n0$ in the same cluster. By doing this, the communication cost between $n0$ and $n1$ is set to zero and the parallel time is reduced. Although, node $n4$ and $n6$ have the same value of *priority* as $n1$, they are not selected for clustering because they are not free nodes.

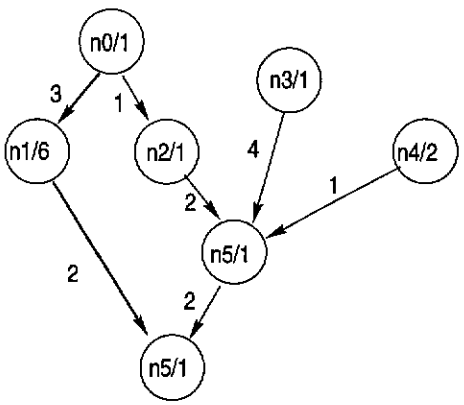


Figure 3.4: Dominant sequence and critical path

According to Yang and Gerasoulis, *dominant sequence* is not the same as *critical path.* Nodes which are on a *dominant sequence* might be the same as nodes on a *critical path* in some cases. For instance, in Figure 3.4 if the weight of $n4$ is 2, nodes on the *dominant sequence* are the same as nodes on the *critical path*, which are $n0, n1, n6$. If the weight of $n4$ is changed from 2 to 6, nodes on the *critical path* are $n0, n1, n6$ but *dominant sequence* consists of nodes of $n4, n2, n3, n5, n6$. This happens because nodes $n2, n3, n4$ are intree nodes and are sorted by considering the weight and communication cost of the nodes. If the weight of $n4$ is 6, it is chosen for clustering first and the sequence for considering intree clustering is $n4, n2$, and $n3$.

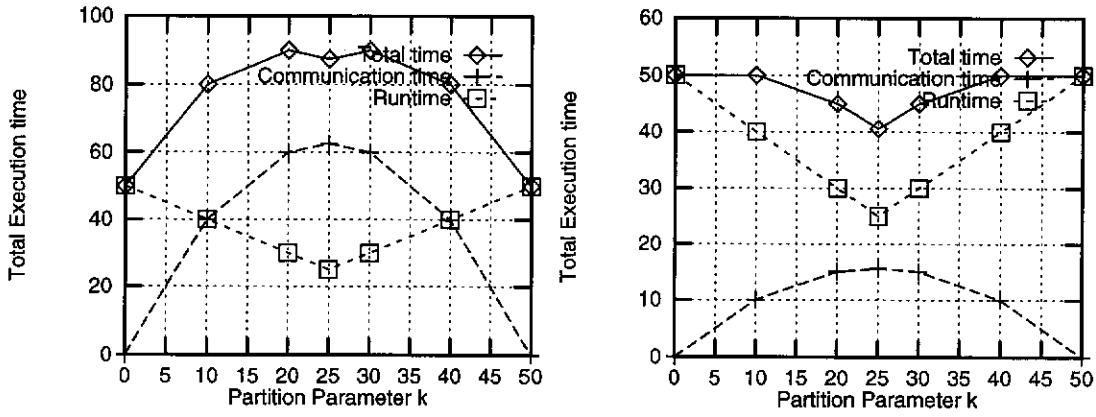
In the case of the out-tree task graph, child nodes of x are merged with x until $\sum_{i=0}^{k-1} r(ni) \leq c(nx, nk))$, where $r(ni)$ is the computation cost of ni and $c(nx, ni)$ is the communication cost between nx and ni. In Figure 3.3(a), node nx has $n0, n1, n2, n3, ..., nm$ as child nodes which are sorted by computation cost in descending order. Therefore, the computation cost of node $n0$ is greater than or equal to that of node $n1$. Node $n0$ is collected with nx first then the communication between nx and $n0$ is set to zero. The result of $r(nx) + r(n0)$ is compared with $c(nx, n2)$. If $r(nx) + r(n0)$ is less than $c(nx, n2)$, the clustering is acceptable. Node $n0$ is clustered with nx resulting in a computation cost of $r(nx) + r(n0)$. Node $n1$ is considered for clustering with nx next. Child nodes of nx are collected to cluster with nx until the total computation of

$nx + n0 + n1 + ... + n(k-1)$ is less than the communication cost $c(nx, nk)$. The parent node of nx is clustered with x until computation cost of the cluster is less than the communication between nx and next parent node. As shown in Figure 3.3(b), when node $n0$ is clustered with nx, the communication between $n0$ and nx is set to zero. If $r(n0) + r(nx)$ is less than $c(nx, n1)$, node $n0$ is clustered with nx. Then node $n1$ is considered for clustering. The computation cost $r(nx) + r(n0) + r(n1)$ is compared with the communication between nx and $n2$, $c(nx, n2)$. If the computation cost is less than the communication, node $n1$ is included in the cluster. This procedure continues until $\sum_{i=0}^{k-1} r(ni) \leq c(nx, nk)$. The results of clustering nodes of Figure 3.1 are shown in Figure 3.2(b) and are the same as those obtained with the MD algorithm. The DSC algorithm is better if there are many entry nodes or many exit nodes in a task graph. These nodes are collected and formed into a clusters as shown in Figure 3.4.

3.3 The Ratio of Computation and Communication Costs

Indurkhya et al.[42] proposed two policies for efficient partitioning of tasks: i) even distribution of tasks among processors and ii) an efficient number of processors among which to distribute tasks evenly.

According to Stone [43] and Indurkhya et al.[42], performance due to the execution of an application using multiprocessors depends strongly on the ratio r/c, where r is the execution time of a single task on a processor and c is the communication cost between the single task and other tasks on other processors. The ratio shows how much overhead is incurred per unit of computation. If many tasks are gathered as a single cluster, R and C are used to represent the execution time and communication cost of the cluster. When the ratio of R/C is very low, C is very high compared with R, hence it becomes unprofitable to use parallelism. In contrast, when the ratio is very high, the communication cost is low, parallelism is potentially profitable. The ratio R/C can be used as a measure of task granularity. Assuming that the total communication



(a) $R/C = 10$ (b) $R/C = 40$

Figure 3.5: The execution time of a program using two processors

bandwidth among processors is a fixed constant, the time attributed to communication delays is proportional to the total amount of data exchanged among processors. A large data size takes more time to be transferred between two processors than a small one. In *coarse grain* parallelism, R/C is relatively high, each unit of computation or a task communicates with others with a relatively small amount of communication. In *fine grain* parallelism, R/C is low, each set of tasks produces a large amount of communication cost. Stone [43] defined the execution time of an application program consisting of M tasks executed on two processors by,

$$Execution\ time = r * Max(M-k, k) + c * (M-k)k, \tag{3.1}$$

where k tasks are assigned to one processor and $M-k$ are assigned to the other. The execution time is varied depending on the value of k, r and c. When k is zero, all tasks are allocated only to a single processor, the execution time is $r * M$ which is the same as when M is equal to k. Then the question is, what is the value of k that produces the smallest execution time?

From Equation 3.1, the total execution time consists of two parts, computation and communication costs. As shown in Figure 3.5, if an application problem consisting of 50 tasks is executed by two processors, the execution time depends on the number of

tasks allocated on each processor and the ratio between computation to communication costs (R/C). Figure 3.5(a) shows the costs of computation, communication, and total execution time, when R/C is set to 10 and the number of tasks on a processor, k, is $0 \leq k \leq 50$. When k tasks are allocated to one processor, $50 - k$ tasks are allocated to the other. When $R/C = 10$, minimum execution time occurs when all tasks are allocated to a single processor. This is due to high communication cost between the two processors. On the other hand, when $R/C = 40$, it is clear from Figure 3.5(b), that equal distribution of tasks among the two processors yields better results. This is because communication costs are not as high compared to the cost of computation of each task. Stone[44] and Indurkhya etal. [42] conclude that,

- if $R/C \geq N/2$ then distribute the N tasks among all processors.
- if $R/C < N/2$ then assign all tasks to one of the processors.

If there are more than two processors, the execution time is calculated using Equation 3.2 [43].

$$Execution\ time = r * Max(k_i) + \frac{c}{2} * (M^2 - \sum_i k_i^2), \tag{3.2}$$

where k_i is the number of tasks that are assigned to the ith processor.

Indurkhya et al. [42, 44] studied task assignments where the processors were not identical. The degradation factor among processors is identified by the value of ρ. A small value of ρ means that the processing speed degrades rapidly, hence a large degradation factor and vice versa. Two systems are compared; system with M and $M + 1$ number of processors. The result of the comparison between the two systems shows that $T(M + 1) \leq T(M)$ iff $N \leq (R/C)(1 + \rho)$, or assigning tasks among $M + 1$ processors gets lower execution time than assigning them among M processors when the number

of tasks is less than $(R/C)(1+\rho)$. In the case that $N \leq R/C)(1+\rho)$, tasks are assigned among all P processors. On the other hand, if $N > R/C)(1+\rho)$, all tasks are executed on one processor only.

The optimal policy of Indurkhya et al.[42, 44], assigns tasks to processors depending on the relative amount of computation and communication. If the processors are identical, tasks are distributed evenly among them. If the processors have different processing speeds, the tasks are assigned inversely proportional to their speed. The processor that is able to execute at high speed gets more tasks for execution than the slower processor. In terms of communication among processors, tasks are distributed in a way that maximizes the use of all processors in the system bandwidth. In contrast, with the system that has a low bandwidth, tasks tend to be executed only on one machine.

According to Stone, Indurkhya et al.[42, 44], tasks are distributed only two cases, distributed evenly to processors or executed only one processor. In the real problem, sometimes it is difficult to divide tasks evenly in terms of computation cost because of different tasks has different amount of computation cost and each task is indivisible. For instance, two tasks size of 5 and 10, cannot be divided into two tasks evenly and distributed to two processors. The communication cost is another important issue for considering. In theoretical manner some costs are ignored to reduce the complicated computation, for instance, sending and receiving data of a processor are assumed that they can be operated at the same time.

In Chingchit et al. [21, 45], tasks are collected into a cluster by determining the ratio of computation and communication costs. The size of clusters depends on the computation and communication cost. When tasks are clustered, the ratio of costs of computation and communication of the cluster is probed so as to keep the overall ratio of cost of computation and communication of each *cluster* within predetermined limits. These limits are based on such factors as processor architecture, number of processors, and desired execution time. Clusters are formed by gathering individual nodes of a given

problem graph into clusters based on costs of computation, communication, precedence relations among tasks/clusters, and the capacity of the underlying architecture. Details of the algorithm are described in Chapter 4.

Processor elements (PEs) of distributed processor systems are connected via communication links. Data and information are transferred via the link. The bandwidth of communication links affect the performance or efficiency of parallel results and the way to schedule nodes across processors. If the communication bandwidth is very low or the transfer rate is very low compared to the computation speed, the given set of tasks should be executed by using less number of processors. In the worst case when the communication cost is very high, one processor might be enough to execute all tasks and more efficient than using many. On the other hand, when the communication bandwidth of the system is very high, the transfer rate is high so it is better to have distributed tasks across many processors. Linear speedup generally does not occur in multicomputer systems because of communication overhead. Whenever the number of processors increases the interprocessor communication cost also increases. In order to achieve greater speedups in such systems, the communication cost should be taken into consideration in making scheduling decisions. The complexity of this problem further increases when communication is considered. Therefore to improve the performance of a distributed computing system, two goals need to be met, interprocessor communication has to be minimized, and the execution cost needs to be balanced among different processors.

3.4 Computation and Communication Costs

Computation and communication costs are important factors which affect clustering and scheduling models[4, 13, 14, 15, 24, 42, 46]. The last section discussed how the ratio of computation and communication cost influences the way tasks are distributed to processors.

Many computation and communication models have been proposed but most have been considered impractical because they have not been validated by experimental results on multiprocessor systems[33]. According to Huang et al.[33], the computation cost of each task is given by $t = n\omega$ and the communication cost for each edge is given by $c = \alpha + n\beta$, where ω is the time for computing, α is the setup time, and β is the transmission time of an element. The execution time obtained from the above formulae differ from experimental ones for the following reasons

i) processors may not be able to send and receive data in parallel

ii) some operations such as 'if' statement may be ignored in calculating cost

iii) the cost of communication depends on various external factors

iv) bidirectional communication over a link may be more expensive than anticipated.

Therefore, when the computation cost is calculated for each task, the time spent for all computation operation should be included. In some multiprocessor systems, each processor is connected to a router, the processor itself can receive and send data in serial. Parallel send and receive can be performed by the router. Receiving time of many nodes from the same source is also important. Generally when a node broadcasts data to others which are not on the same processor, receiving time at different nodes is assumed to be the same. According to Huang et al.[33], receiving time of a task depends on the order in which the sending task transmits the data. For accurate consideration of communication costs, the order in which data is sent and received is critical.

3.5 Conclusion

In this chapter, many algorithms have been reviewed and discussed. None of them take the characteristics of the parallel system into account. The execution time of given

application programs is also driven by the characteristics of the target system: the parallel computer on which the application program is executed. If the clustering is transparent to the parallel computer, then the performance on one parallel computer may be completely different from that on another. For example, execution on a network of workstations with powerful processors connected by a relatively low bandwidth communication network demands large sized clusters whereas execution on multiprocessors required smaller medium sized clusters. The Flexible clustering algorithm addresses these issues. We present the FCS algorithm in the next chapter.

Chapter 4

The Flexible Clustering and Scheduling Scheme

In the last chapter, we presented an overview of existing literature related to scheduling, clustering, communication and computation costs that affect performance of multiple processor systems. In this chapter, the flexible clustering and scheduling scheme (FCS) will be presented. The FCS algorithm can be adapted to suit a particular system and/or satisfy desired goals.

Scheduling algorithms need to address several issues: 1) availability of processors, 2)task granularity of the tasks in the given problem, 3)processor speed, 4) network bandwidth, 5)execution time, 6) efficiency, and 7) load balance. In addition, the complexity of the clustering and scheduling algorithm should be within reasonable bounds in order to minimise overheads. The main objective of clustering is to combine interdependent tasks for execution on the same processor in order to reduce communication costs. The objective of scheduling is to execute tasks in different processors such that the parallel execution time is reduced. The FCS scheme is motivated by the work done by Indurkhya et al. [42, 43] on efficient partitioning of programs and the work of several researchers

on clustering and scheduling techniques [4, 13, 24, 14, 40]. In particular, Gerasoulis and Yang [15] have done pioneering work in task scheduling and allocation, which has been discussed in Chapter 3. Parallel algorithms can be implemented on various types of parallel computing systems : SIMD machines with very primitive processors, MIMD computers with *state of the art* microprocessors and very high channel bandwidths, and NOWs with high performance computers and high channel bandwidths. Clearly, the actual execution times of given algorithms depend not only on the clustering and scheduling technique, but also on the nature of the parallel computing platform. None of the algorithms described in the literature take into account system parameters.

4.1 Introduction

The flexible clustering and scheduling algorithm relates to the ratio of computation and communication cost of clusters and multiprocessor system capacities. If the parallel computing system possesses a high communication bandwidth, then a given application program's tasks are clustered and distributed among many processors in order to achieve high parallelism. However, usage of too many processors may lead to poor efficiency. On the other hand, when the communication bandwidth of a system is low, tasks are grouped into larger clusters in order to reduce communication overheads. In the latter case, the task granularity is high, parallelism may be low, and the efficiency may be better than the former case. With flexible clustering, users have the option of choosing one of the above cases. From another point of view, on a multiprocessing system consisting of high-speed processors, a large number of tasks can be executed on each processor within a short period. Therefore, each cluster allocated to a processor can be large consisting of many tasks. In contrast, if the system has low-speed processors, each processor should execute a smaller cluster. Obviously in this case, many processors are used for executing the tasks of an application program.

4.1.1 System Parameters

When an application program is implemented on a multiprocessor system, many factors affect the speedup or efficiency of the system. The capabilities of the system, for instance, the execution speed of processors or processor throughput (ρ), the transmission rate or communication bandwidth (γ), and the inter-processor communication network are factors that affect clustering, scheduling, and parallel execution time. The values of ρ and γ directly affect clustering and scheduling. The execution speed (ρ) is the ability of a processor to execute a set of instructions within a unit of time. Processors with high execution speed can execute more instructions than those with lower execution speeds in a given unit time. The speed affects cluster size. As we know, a cluster consists of tasks collected and allocated for execution on a processor. Therefore, ideally, we want a high-speed processors to execute a large clusters and low-speed processors to execute small clusters. The transmission rate or communication bandwidth (γ) is another factor that affects clustering and scheduling. With high speed transmission or high communication bandwidth, the system can transfer data among processors rapidly. High speed transmission supports the system that consists of low-speed execution processors because many clusters are divided and distributed to many processors with less communication cost. The flexible clustering and scheduling algorithm (FCS) uses the ratio of processor execution speed to communication speed or bandwidth, $\tau = k \cdot \rho/\gamma$, as a key to determine the size of each cluster, k is a constant greater than zero. For systems that consist of low-speed execution processors and high communication bandwidth, the value of τ is low. Small numbers of tasks are collected in a cluster and distributed to many processors with less communication cost. On the other hand, if each processor has a high throughput and the system has low communication bandwidth, the value of τ is high. Many tasks are gathered in one cluster and executed on a number of processors. Thus, the FCS algorithm can be tuned to suit different system characteristics depending on the value of τ to achieve high speedup or high efficiency.

4.1.2 Clustering

The clustering and scheduling techniques proposed thus far do not take into consideration the relative costs of computation and communication. The relationship between clustering and the variety of system characteristics is ignored. Clustering and scheduling should consider the underlying architecture for practical implementations. Most existing scheduling techniques are based on 'edge zeroing' [13, 24, 14, 47, 48]. When two nodes are gathered into the same cluster, the communication cost between the nodes is set to zero. The algorithms differ in the way in which tasks are gathered into clusters.

The FCS scheme takes the relative costs of computation and communication into consideration. The ratio of costs of computation and communication of clusters are computed in a continuous manner so as to keep the overall ratio of cost of computation and communication of each *cluster* within predetermined limits. These limits are based on such factors as processor architecture, number of processors, and desired execution time. This approach is novel because it is flexible and tuned to yield desirable results for all types of application problems and/or architectures. The FCS scheme compares cluster granularity with processor characteristics and the network communication cost. Clusters are formed by gathering individual nodes of a given problem graph into clusters based on costs of computation, communication, precedence relations among tasks/clusters, and the capacity of the underlying architecture.

The FCS scheme can be effectively employed to achieve one or more of the following requirements,

i) efficient clustering and scheduling on bounded number of processors

ii) maximal speedup and efficiency

iii) minimal execution time on a given architecture

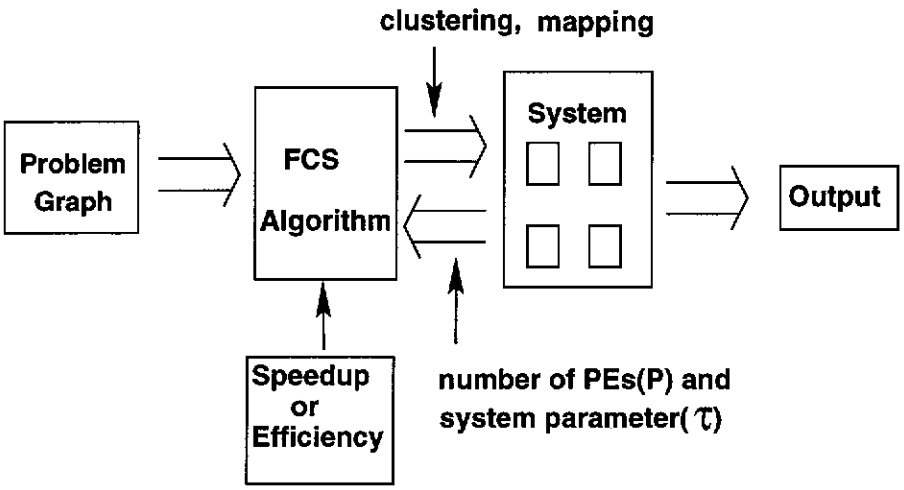


Figure 4.1: The FCS mechanism

iv) efficient execution on heterogeneous systems : MIMD, SIMD, and heterogeneous network of workstations and

v) equitable distribution of load among the processors

Figure 4.1 is a schematic diagram of the FCS algorithm. As we discussed before, a task graph is used to represent an application problem and each task represents a set of instructions. The problem graph is the main input to the FCS algorithm, providing details of communication links among tasks, the computation cost of each task and precedence relations. System parameters are received by the FCS algorithm from the system target. This input includes the number of processors available and the system parameter τ.

Each cluster's β is matched with the corresponding τ of the processor. When a task is added to a cluster, the values of R/C are compared to the value of τ. In the case of unbounded number of processors, the FCS algorithm compares the β of the cluster with τ of the processor in a continuos fashion. When β exceeds τ, the cluster is mapped to an available processor. The process continues by creating a new cluster and comparing the β with τ and maps the cluster as before to the next processors in the lists. The

most innovative feature of the scheme is that the number of processor used is never greater than the required number. Use of more processors does not improve speedup but reduces efficiency instead.

On the other hand, when the number of processors is bounded by P, FCS adapts itself in the following way. The mapping process begins as in the case of unbounded number of processors. When there are no more processors available, the FCS allocates the new cluster to the processor whose cluster has the lowest β. In the case of heterogeneous systems, because τ of each processor may be different, we pick the processor which has the least load. In other word, a processor i such that $|\tau_i - \beta_i|$ is least for $0 \leq i \leq P-1$ is picked.

Speedup and efficiency are other inputs to the FCS algorithm. When an application program is implemented on a system, the objective is to achieve the smallest execution time. The *execution time* of a given problem is the total time taken to execute the problem. In many cases, execution of an application using one processor is faster than many processors due to communication costs. The ratio of the execution time (T_1) using a single processor of the given problem to the execution time (T_p) of the same problem on p processors is called *speedup*($speedup = \frac{T_1}{T_p}$). The number of processors (p) used for executing a given application affects the efficiency of the implementation. The $efficiency$ of parallel implementation is given by, $E = \frac{speedup}{p}$. The main goals of parallel implementation are to increase speedup and efficiency. However, in many cases an increase in speedup results in a decrease in efficiency because of the increase in the number of processors used.

4.2 The Scheme

In this section a novel clustering and scheduling algorithm that takes into account the cost of computation and communication of the underlying tasks is introduced.

The proposed algorithm gathers tasks into clusters such that the ratio of the cost of computation and communication is suited for execution on the underlying architecture. One of the novel features of this method is the use of optimal number of processors for execution of the given problem.

4.2.1 Clustering

The clustering and scheduling algorithm consists of two phases, i)*clustering* during which tasks of the graph are gathered into clusters, and ii)*scheduling* during which clusters are allocated to available processors. Clustering is based on the computation and communication costs of individual tasks and precedence relations. In the following section, sample precedence relations and calculation of R and C of clusters for sample graph segments are illustrated.

4.2.2 Computation of R and C

Nodes of a DAG can be gathered into clusters in many ways depending on computation and communication costs. The two nodes of Figure 4.2(a), r_1 and r_2, are sequentially executed. The total cost of computation of this cluster is given by $R = r_1 + r_2$ and total communication cost C, is given by $(c1 + c3 + c4)$. Tasks r_1 and r_2 are executed in the same cluster which means that these tasks are allocated to the same PE. As was discussed in Chapter 3, when tasks that are on the same processor write and retrieve data from the same memory, there is no communication cost between the tasks. Therefore, communication cost of c_2 is zero. The communication costs that affect this cluster, cl(n1, n2), are $c1, c3$, and $c4$. The communication cost of $c5$ does not affect the starting or finishing time. Communication cost is considered only for input costs because the communication cost affects the starting time only of the receiving node. For example, node A transmits data with the cost of $c(A, B)$ to B, node B starts execution with the starting time of $st(A) + r(A) + c(A, B)$ where $st(A)$ is the starting

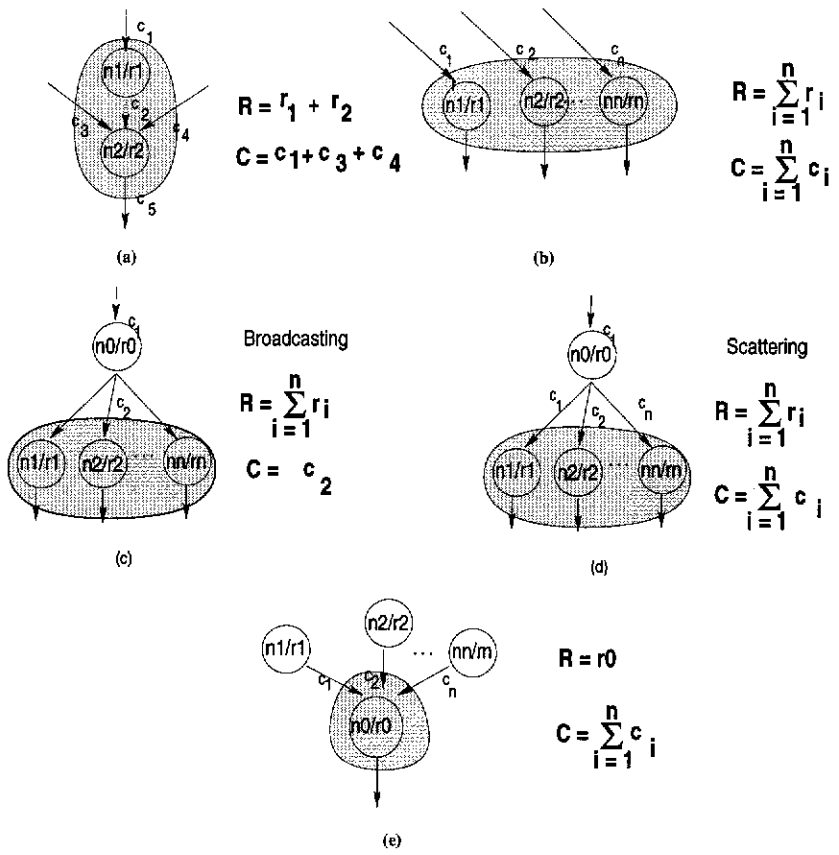


Figure 4.2: Computation of R and C for different clusters

time of A, $r(A)$ is the computation time of A. The communication cost, $c(A, B)$ does not affect starting time, finishing time or computation cost of A at all. In addition, communication is only one way, B receives data from A, therefore $c(A, B)$ is considered for B only. The effective β value for the DAG of Figure 4.2(a) is $\frac{r_1+r_2}{c_1+c_3+c_4}$, when nodes in a cluster have precedent nodes as shown in Figure 4.2(b), the order of execution is dependent on the arrival of inputs from precedent nodes. In other words, order of execution depends on input communication time, $c_1, c_2, ..., c_n$. The effective β is given by,

$$\frac{\sum_{i=1}^{n} r_i}{\sum_{i=1}^{n} c_i}.$$

In the case of broadcasting as shown in Figure 4.2(c), node n_0 broadcasts the same data to nodes $n_1, n_2, ...n_n$ which are in the same cluster. The effective β of the cluster comprising nodes $n_1, n_2, ...n_n$ is $\frac{r_1+r_2+...r_n}{c_2}$. However, in the case of scattering shown in Figure 4.2(d), node n_0 sends different data to different receiving nodes. Therefore, the

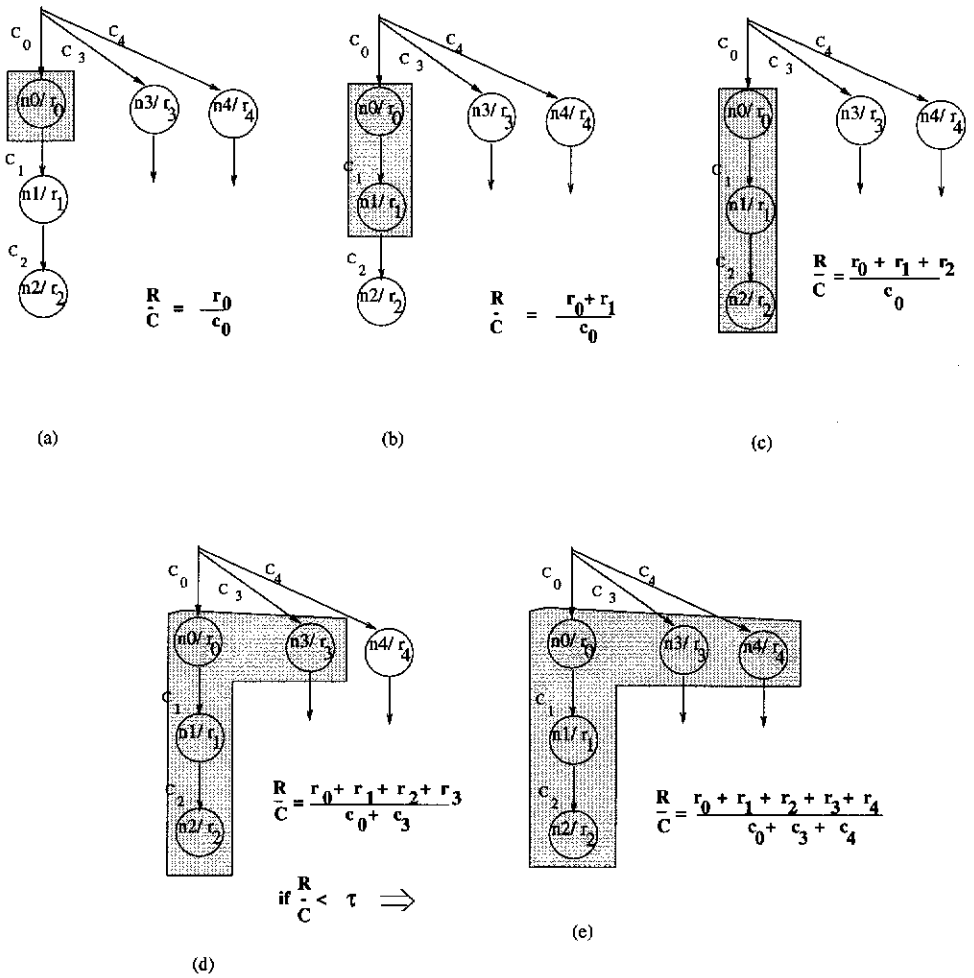


Figure 4.3: The clustering mechanism

communication cost for scattering is $\sum_{i=1}^{n} c(n_0, n_i)$. Although the data can be packed and sent at the same time, the transmission time is reduced only for some sets of data, for instance, header, tailer, and error correction.

In Figure 4.2(e), a node in the cluster receives data from many sources, $n_1, n_2, ...n_n$, the effective β of the cluster $cl(n_0)$ is

$$\frac{r_{n_0}}{\sum_{i=1}^{n} c(n_i, n_0)}.$$

As can be seen in all the above cases, β depends on r and c, and the clustering mechanism.

Figure 4.3 illustrates the process of cluster creation using the FCS algorithm. Nodes that execute in a serial fashion are collected first because some of these nodes are dependent nodes. A node cannot start execution until its precedent nodes finishes execution and data is receieved from them. Nodes n_0, n_1 and n_2 are serial nodes, after collection of their serial nodes, sibling nodes of n_0 are considered for clustering. As shown in Figure 4.3(d), node n_3 is considered for clustering, which results in a β of $\frac{r_0+r_1+r_2+r_3}{c_0+c_3}$. The communication cost of c_3 is reduced to zero in the case of broadcasting, node n_0 and n_3 execute their tasks using the same set of input data which is transmitted by $p(n_0)$. At this stage, the value of β is given by $\frac{r_0+r_1+r_2+r_3}{c_0+c_3}$ for scattering and by $\frac{r_0+r_1+r_2+r_3}{c_0}$ for broadcasting. After the β is compared to τ, if it is still less than τ node n_4 will be collected as shown in Figure 4.3(e). If the value of β of the cluster at any stage is greater than or equal to the value of τ, the cluster is assigned to an available PE. However, if $\beta < \tau$, the remaining nodes in the list L are considered for clustering.

4.2.3 The Algorithm

The flexible clustering and scheduling algorithm is shown in Figure 4.4 and has three main components. Firstly, we determine the critical path (CP) of the given task graph G(V,E) by computing the $ASAP$ and $ALAP$ times for all nodes in G. The nodes of the CP are grouped into one cluster and assigned to the first PE ($PE0$ in this case). The nodes of the graph are ordered lexicographically.

The second part of the algorithm is the most important contribution of the proposed FCS scheme. During this part, clusters are created from the remaining nodes of G. Every time a cluster is formed, its size in terms of β is compared to the system parameters, τ. The nodes of the cluster are removed from the lexicographic list, L. To form a new cluster, the first node x in L is picked for clustering. The β of $cl(x)$ is given by $\frac{r_x}{c_x}$ and is compared with τ. If $\beta \geq \tau$ then the single node x forms a cluster and allocated to a processor. However, if β is less than τ, then other nodes in L are clustered with x

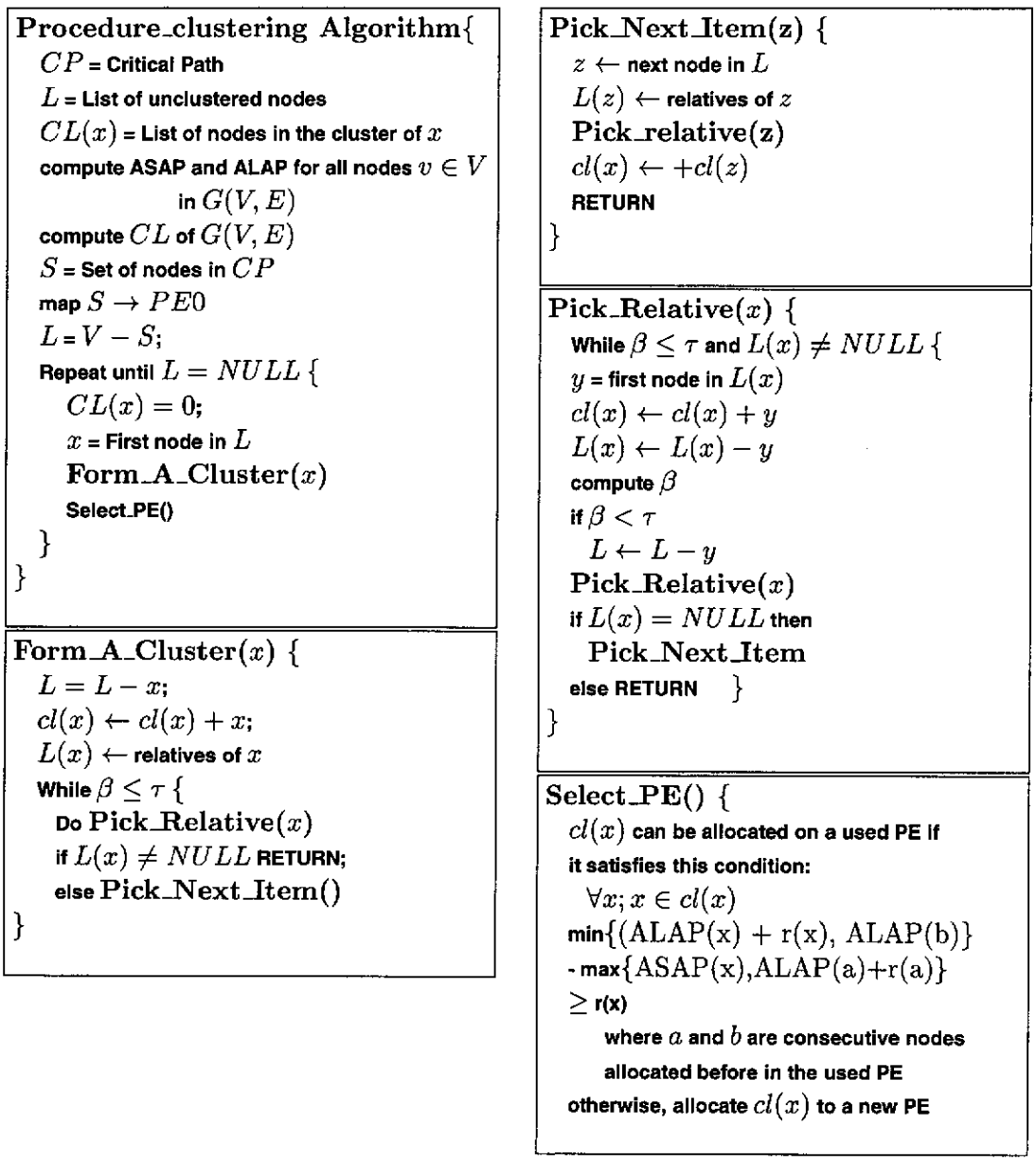


Figure 4.4: The FCS algorithm

until $\beta \geq \tau$ or $L = 0$. The order in which nodes are picked to form clusters is as follows. After gathering each node in the cluster, β is recomputed and goes to the next node if and only if β is less than τ. Lastly, the cluster whose β is less than τ is allocated to an available processor. In the case of an unbounded number of processors, each cluster is allocated to a new processor. On the other hand, if the number of processors is limited, after all processors are used, a new cluster is allocated to the processor whose β is the

least.

The objective of the scheme is to reduce the cost of communication among nodes. The nodes are gathered based on precedence relations. The main idea behind our algorithm is to cluster nodes (tasks) which are the neighbours of a given node x into the same cluster with the intention of reducing the communication cost. We assume that the communication cost between two nodes in the same cluster is zero. The order in which neighbour nodes are considered for clustering with $cl(x)$ is as follows:

i) $p(x)$	iii) $h(x)$	v) $s(x)$	vii) $p(s(x))$
ii) $p^i(x)$	iv) $h^i(x)$	vi) $s^i(x)$	viii) $n(x)$.

Initially, the cluster $cl(x)$ consists of only one node, say x. Now if β of the cluster $cl(x)$ is less than τ, then we consider inclusion of other nodes in the order given above in the cluster. After each inclusion we compare β with τ, the cluster is allocated to an available PE if $\beta \geq \tau$. By maintaining the sequence, we minimize the communication cost of the cluster.

4.2.3.1 Clustering with Neighbours

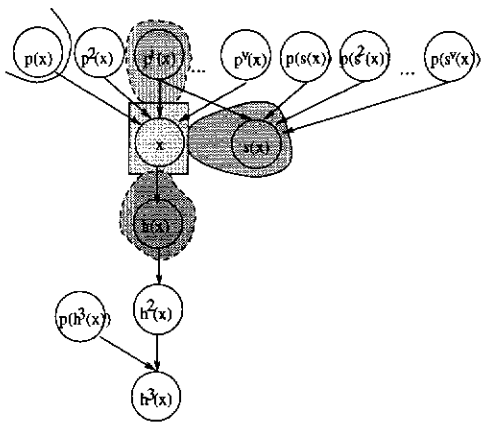


Figure 4.5: Cost of clustering

When a neighbour of node x is considered for clustering with x, the communication cost between them will be reduced to zero if they are in the same cluster. Then β of the cluster will increase due to higher cost of computation and possibly lower cost of communication. By doing this, we increase the granularity of the cluster. This scenario is shown in Figure 4.5

First, let us consider node $p^i(x)$ for clustering with $cl(x)$, where $x \leq i \leq v$. As shown in Figure 4.5, $p(x)$ is assumed to be in another cluster possibly in the critical path. The computation cost of cluster $cl(x, p^i(x))$ is

$$R(cl(x, p^i(x))) = r(x) + r(p^i(x)) \tag{4.1}$$

After clustering, the communication cost of $c(p^i(x), x)$ is set to zero; $c(p^i(x), x) = 0$. The effective cost of communication of the cluster $cl(x, p^i(x))$ is given by

$$C(cl(x, p^i(x))) = \sum_{j=2}^{v} c(p^j(x), x) - c(p^i(x), x), \tag{4.2}$$

where v is the number of parent of x; $p(x)$.

Secondly, node $h(x)$ is considered for clustering with $cl(x)$, the computation cost of the cluster is

$$R(cl(x, h(x))) = r(x) + r(h(x)). \tag{4.3}$$

The communication cost between $h(x)$ and x is zero; $c(x, h(x)) = 0$. Then the communication cost of the cluster $cl(x, h(x))$ is

$$C(cl(x, h(x))) = \sum_{j=2}^{v} c(p^j(x), x). \tag{4.4}$$

Thirdly, node $s(x)$ is considered for clustering with $cl(x)$, the computation cost of the

cluster is

$$R(cl(x, s(x))) = r(x) + r(s(x)). \tag{4.5}$$

Nodes $s(x)$ and x are connected to the same parent, $p^i(x)$. Then communication cost when $s(x)$ is clustered to $cl(x)$ is considered for two cases, broadcast and scatter.

Case 1 : $p^i(x)$ broadcasts data to x and $s(x)$

When $s(x)$ is clustered to x, the communication cost of the cluster is given by,

$$C(cl(x, s(x))) = \sum_{j=2}^{v} c(x, p^j(x)) + \sum_{j=1}^{u} c(p^j(s(x)), s(x)) - c(p^i(s(x)), s(x)), \tag{4.6}$$

where u is the number of parents of $s(x)$ and $1 \leq i \leq u$. In the case of broadcasting, x and $s(x)$ receive the same data from the same node, $p(x)$. Node $p(x)$ sends data to x and $s(x)$ only once. Therefore, $c(p^i(s(x)), s(x))$ is set to zero or there is no communication between $p^i(s(x))$ and $s(x)$.

Case 2 : $p^i(x)$ scatters data to x and $s(x)$.

The communication cost of $cl(x, s(x))$ is

$$C(cl(x, s(x))) = \sum_{j=2}^{v} c(x, p^j(x)) + \sum_{j=1}^{u} c(p^j(s(x)), s(x)). \tag{4.7}$$

For simplicity, we assume that all tasks are of equal complexity. Similarly, the cost of communication of all links is the same.

Theorem 1 *For every unclustered parent node $p^i(x)$ of x, the effective cost of clustering $p^i(x)$ with x is less than the effective cost of clustering $h(x)$ with x.*

Proof: From equations (4.1) and (4.3) we have,

$$R(cl(x, p^i(x))) = R(cl(x, h(x))).$$

The computation costs of $cl(x, p^i(x))$ and $cl(x, h(x))$ are also the same. From equations (4.2) and (4.4), we have

$$C(x, cl(p^i(x))) < C(cl(x, h(x))).$$

Therefore, β of $cl(x, p^i(x))$ is less than that of $cl(x, h(x))$ or the effective cost of $cl(x, p^i(x))$ is less than the effective cost of $cl(x, h(x))$. Another reason for clustering $p^i(x)$ with $cl(x)$ first is due to the starting time of $cl(x, p^i(x))$ with respect to those of other possible clusters. Cluster $cl(x, h(x))$ cannot start execution unless $p^i(x)$ and x finish their tasks.

Now, the effective cost of $cl(x, p^i(x))$ compares with that of $cl(x, s(x))$. From (4.1) and (4.5),

$$R(cl(x, p^i(x))) = R(cl(x, s(x))) \tag{4.8}$$

and from (4.2), (4.6) and (4.7)

$$C(cl(x, p^i(x))) < C(cl(x, s(x))) \tag{4.9}$$

for both broadcast and scatter. From (4.8) and (4.9)

$$\beta(cl(x, p^i(x))) < \beta(cl(x, s(x))) \tag{4.10}$$

Therefore the cost of clustering $p^i(x)$ with x is less than the cost of clustering $h(x)$ with x. It is clear that $p^i(x)$ should be clustered with $cl(x)$ prior to the other neighbours of x.

All parents of x are selected for clustering prior to considering any other neighbours of x. After all remaining parents $p^i(x)$ are clustered with x and β is less than τ, the neighbours of the cluster $cl(x, p(x), ...p^v(x))$ are considered for clustering.

If there is a child of x, $h(x)$, then there exists an edge from x to $h(x)$. Before child of x is selected for clustering, β of $cl(x, p(x), ...p^v(x))$ is $r(x) + \sum_{j=2}^{v} r(p^j(x))$, if all parents are entry nodes. By clustering x and $h(x)$ together, the cost of $c(x, h(x))$ is reduced to zero. The β of the new cluster is changed to $r(x) + h(x) + \sum_{j=2}^{v} r(p^j(x))$.

4.2.3.2 Clustering with Parent Nodes

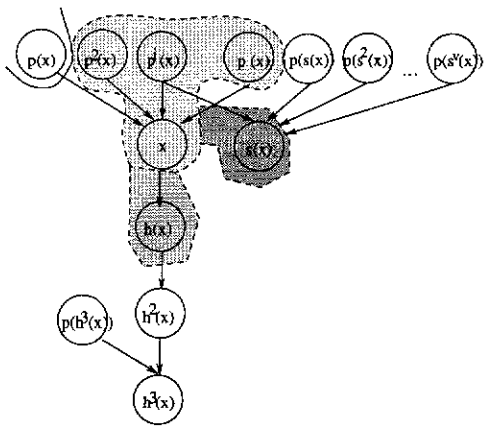


Figure 4.6: Clustering x with $P(x)$

Let $P(x)$ be the set of parent nodes, $p(x), p^2(x), ..., p^v(x)$, that are in the same cluster as x as shown in Figure 4.6. It is trivial to see that the cost of clustering all parents in $P(x)$ with x is lower than the cost of clustering other neighbours of x. Nodes which are likely to be clustered with $cl(x, P(x))$ include $h(x)$, $s(x)$, and other nodes in the lexicographic list, $n(x)$.

Firstly, let us consider node $h(x)$ for clustering with $cl(x, P(x))$. The computation cost of the cluster $cl(x, P(x))$ is $R(cl(x, P(x))) + r(h(x))$. The communication cost of

$cl(x, P(x), h(x))$ is

$$C(cl(x, P(x)), h(x)) = 0. \tag{4.11}$$

When node $s(x)$ is considered for clustering with $cl(x, P(x))$, the computation cost of $cl(x, P(x), s(x))$ is $R(cl(x, P(x))+r(s(x))$ which is the same as that of $cl(x, P(x)), h(x))$. The communication cost of the cluster $cl(x, P(x), s(x))$ is

$$C(cl(x, P(x), s(x))) = \sum_{j=2}^{v} c(p^j(s(x)), s(x)). \tag{4.12}$$

Equation (4.12) gives the cost of communication for both broadcast and scatter as x and $s(x)$ are in the same cluster. If cluster $cl(x, P(x))$ has two neighbouring nodes, a descendent, $h(x)$, and a sibling $s(x)$, we consider clustering $cl(x, P(x))$ with the descendent node first. By doing so we reduce the cost of communication between $cl(x, P(x))$ and $h(x)$ and maintain the parallelism between $cl(x, P(x))$ and $s(x)$.

Theorem 2 *The effective cost of clustering $h(x)$ with $cl(x, P(x))$ is less than the effective cost of clustering $s(x)$ with $cl(x, P(x))$.*

Proof: From equations (4.11) and (4.12),

$$C(cl(x, P(x), s(x))) \geq C(cl(x, P(x), h(x))) \tag{4.13}$$

and

$$R(cl(x, P(x), s(x))) = R(cl(x, P(x), h(x))). \tag{4.14}$$

Therefore,

$$\beta(cl(x, P(x), s(x))) \geq \beta(cl(x, P(x), h(x))). \tag{4.15}$$

Moreover,

$$st(s(x)) \geq st(x) \tag{4.16}$$

whereas,

$$st(h(x)) > st(x). \tag{4.17}$$

From (4.16) and (4.17) it is clear that clustering $s(x)$ with $cl(x, P(x))$ is more expensive than clustering $h(x)$ with $cl(x, P(x))$.

A child of $h(x)$ is represented by $h(h(x))$ or $h^2(x)$, and a child of $h^2(x)$ by $h^3(x)$ and so on. If there is a $h^k(x)$, then there exists an edge from $h^{k-1}(x)$ to $h^k(x)$. By clustering $h(x)$ with $h^2(x)$ the cost of $c(h(x), h^2(x))$ is reduced to zero. In general, by clustering $h^{k-1}(x)$ and $h^k(x)$ together, the cost of $c(h^{k-1}(x), h^k(x))$ is reduced to zero. By doing so the computation cost of the cluster is increased but the communication cost is still the same. Therefore, the granularity of the cluster is increased.

4.2.3.3 Clustering with Child Nodes

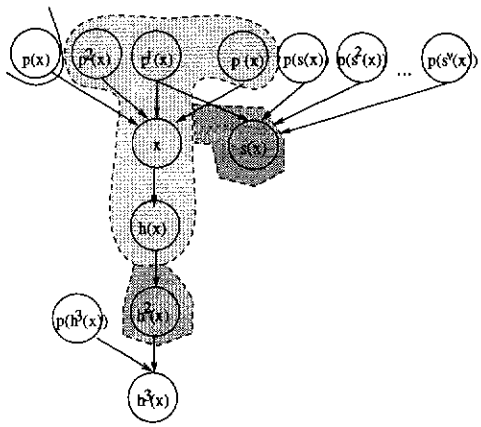


Figure 4.7: Clustering $cl(x, P(x))$ with $h(x)$

The pattern of neighbours of $cl(x, P(x), h(x))$ is similar to the pattern encountered in the previous subsection. Two neighbour nodes of the cluster are $h^2(x)$ and $s(x)$ as shown in Figure 4.7. If $h^2(x)$ is clustered with $cl(x, P(x), h(x))$, the computation cost of the cluster $cl(x, P(x), h(x), h^2(x))$ is

$$R(cl(x, P(x), h(x), h^2(x)) = r(x) + \sum_{j=2}^{w} r(p^j(x)) + r(h(x)) + r(h^2(x)), \tag{4.18}$$

where w is the number of parents of x that are clustered with x. The communication cost of the cluster is

$$C(cl(x, P(x), h(x), h^2(x)) = 0. \tag{4.19}$$

Now, we consider clustering $s(x)$ with $cl(x, P(x), h(x))$. The computation cost of the cluster $cl(x, P(x), h(x), s(x))$ is

$$R(cl(x, P(x), h(x), s(x)) = r(x) + \sum_{j=2}^{w} r(p^j(x)) + r(h(x)) + r(s(x)) \tag{4.20}$$

and the communication cost of the cluster is

$$C(cl(x, P(x), h(x), s(x)) = \sum_{j=2}^{u} c(p^j(s(x)), s(x)). \tag{4.21}$$

Theorem 3 *For every $h^i(x)$, the effective cost of clustering $h^i(x)$ with $cl(x, P(x), h(x), ..., h^{i-1}(x))$ is less than the effective cost of clustering $s(x)$ with $cl(x, P(x), h(x), ..., h^{i-1}(x))$.*

Proof: Node $h^i(x)$, where $2 \leq i \leq k$ and k is the number of successors of x, is dependent on $h^{i-1}(x)$. From Equations (4.13) and (4.14),

$$C(cl(x, P(x), s(x))) \geq C(cl(x, P(x), h(x), h^2(x), ..., h^{i-1}(x), h^i(x))) \tag{4.22}$$

and

$$R(cl(x, P(x), s(x))) < R(cl(x, P(x), h(x), h^2(x), ..., h^{i-1}(x), h^i(x))). \tag{4.23}$$

Therefore,

$$\beta(cl(x, P(x), s(x))) < \beta(cl(x, P(x), h(x), h^2(x), ..., h^{i-1}(x), h^i(x))). \tag{4.24}$$

Moreover

$$st(s(x)) \geq st(x) \tag{4.25}$$

whereas

$$st(h^i(x)) > st(x). \tag{4.26}$$

Although $\beta(cl(x, P(x), h(x), h^2(x), ..., h^{i-1}(x), h^i(x)))$ is greater than $\beta(cl(x, P(x), s(x)))$, the communication cost between $h^{i-1}(x)$ and $h^i(x)$ is reduced to zero and it is less than τ. By clustering $h^i(x)$ with $cl(x, P(x), h(x), h^2(x), ..., h^{i-1}(x))$, we maintain the parallelism between $cl(x, P(x), h(x), h^2(x), ..., h^{i-1}(x), h^i(x))$ and $s(x)$.

Theorem 4 *For every $p(h^i(x))$ the effective cost of clustering $p(h^i(x))$ with $cl(x, P(x), h(x), ..., h^i(x))$ is less than the effective cost of clustering $s(x)$ with $cl(x, P(x), h(x), ..., h^i(x))$.*

Proof: The node, $h^i(x)$ is dependent on $h^{i-1}(x)$ as well as $P(h^i(x)) = x$. If $p(h^i(x))$ is not clustered with $cl(x, P(x), h(x), ..., h^i(x))$ the cluster will wait for input from $p(h^i(x))$ with a communication cost of $c(cl(x, P(x), h(x), ..., h^i(x)), p(h^i(x)))$. When $p(h^i(x))$ is clustered with $cl(x, P(x), h(x), ..., h^i(x))$ the communication cost of $c(cl(x, P(x), h(x), ..., h^i(x)), p(h^i(x))$ is reduced to zero. Node $s(x)$ can start execution at the same time as x which means that $cl(x, P(x), h(x), ..., h^i(x))$ and $s(x)$ can be executed in parallel. Therefore clustering $p(h^i(x))$ with $cl(x, P(x), h(x), ..., h^i(x))$ results in less effective cost than that of clustering $s(x)$ with $cl(x, P(x), h(x), ..., h^i(x))$.

4.2.4 Clustering with Descendent Nodes

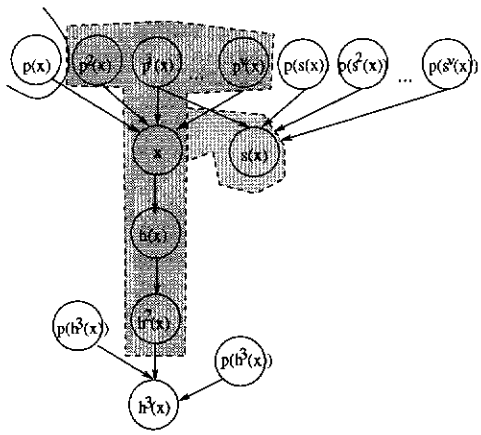


Figure 4.8: Clustering $cl(x, P(x)$ with $h^i(x)$

Let $H(x)$ represent the set $h(x), h^2(x), ..., h^k(x)$ as shown in Figure 4.8. The neighbours of $cl(x, P(x), H(x))$ are $s(x)$ and $n(x)$. As we do not have information of $n(x)$ at this stage, node $s(x)$ is selected to cluster with $cl(x, P(x), H(x))$. By clustering $s(x)$ with $cl(x, P(x), H(x))$, the communication cost of $c(cl(x, P(x), H(x)), s(x))$ is reduced to

zero. It is the same for both broadcasting and scattering because x, $p^i(x)$,and $s(x)$ are all in the same cluster therefore the communication among them is reduced to zero.

The sibling of x is denoted by $s(x)$ and their common parent node $p(x)$ has an edge to $s(x)$ and an associated cost $c(p(x), s(x))$. By clustering $s(x)$ with x, the cost of the edge from $p(x)$ to $s(x)$ is set to zero. Clustering of $s(x)$ results in additional reduction in communication costs associated with the clustering of $s(x)$ with $cl(x, P(x), H(x))$. The computation cost of the cluster $cl(x, P(x), H(x), s(x))$ is

$$R(cl(x, P(x), H(x), s(x))) = r(x) + \sum_{j=2}^{w} r(p^j(x)) + \sum_{j=1}^{v} r(h^j(x)) + r(s(x)). \tag{4.27}$$

The communication cost of the cluster is given by

$$C(cl(x, P(x), H(x), s(x))) = \sum_{j=1}^{u} c(p^j(s(x)), s(x)) - c(p(x), s(x)). \tag{4.28}$$

Theorem 5 *For every $s^i(x)$, the effective cost of clustering $s^i(x)$ with $cl(x, P(x), H(x))$ is less than the effective cost of clustering $n(x)$ with $cl(x, P(x), H(x))$.*

Proof: For every $s^i(x)$ clustered with $cl(x, P(x), H(x))$ the communication cost $c(p^i(x), s^i(x)) = 0$. In the absence of prior knowledge of node $n(x)$ which is a node in the list, L is not likely to be related to x. So we cannot guarantee that clustering $n(x)$ with $cl(x, P(x), H(x))$ will be less effective compared to clustering $s^i(x)$ with $cl(x, P(x), H(x))$.

4.2.5 Clustering with Siblings

Let $S(x)$ represent the set $s(x)$, $s^2(x)$, ..., $s^k(x)$ as shown in Figure 4.9. The neighbours of $cl(x, P(x), H(x), S(x))$ are $p(s(x))$ and $n(x)$. When a sibling of x, $s(x)$, is gathered into the cluster, parents of the sibling node might be unscheduled. Communication costs between the sibling node and its parents can be reduced if the parents are collected to

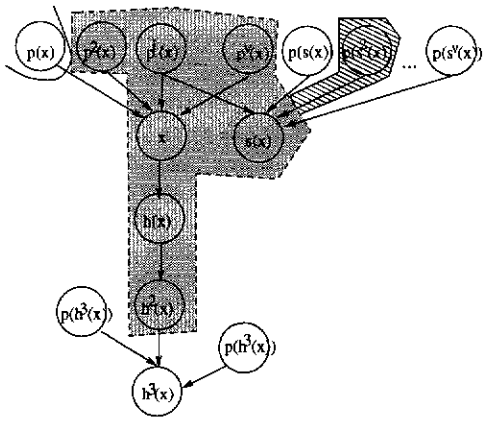


Figure 4.9: Clustering $cl(x, P(x), H(x))$ with $s^i(x)$

the same cluster as that of $s(x)$. When a parent is clustered with $s(x)$, the value of β is recalculated and compared with τ. If $\beta \geq \tau$, the cluster is allocated to an available processor. Otherwise, the next parent is selected to be in the same cluster as $s(x)$. The communication cost of $c(p(s^k(x)), s(x))$ is set to zero when they are in the same cluster.

Theorem 6 *For every $p^i(s(x))$, the effective cost of clustering $p^i(s(x))$ with $cl(x, P(x), H(x), S(x))$ is less than the effective cost of clustering $n(x)$ with $cl(x, P(x), H(x), S(x))$.*

Proof: Due to similar arguments as in the previous subsection, we know that every time we cluster $p^i(s(x))$ with $cl(x, P(x), H(x), S(x))$ the communication cost of $c(p^i(s(x)), s^i(x))$ will be reduced to zero. If we cluster $n(x)$ with $cl(x, P(x), H(x), S(x))$, we cannot guarantee that the communication cost will be reduced. Therefore, clustering $n(x)$ with $cl(x, P(x), H(x), S(x))$ cannot guarantee that the effective cost of clustering is less than that of clustering $p^i(s(x))$ with $cl(x, P(x), H(x), S(x))$.

After a node is gathered into a cluster, it is removed from L. In case that β of the cluster is still less than τ and the list L is nonzero, a node in L is considered for next clustering. If β is less than τ and there is no available node in L, the cluster is allocated to the processor whose β is the least.

Finally, the algorithm allocates processors to clusters in the third part. On completion of a cluster, if the cluster's β is greater than or equal to τ, the cluster is allocated to an available processor. To minimize the number of processors, we try to fit the nodes of such cluster into one of the used processors with sufficient idle time to improve efficiency.

To fit a new node x between the nodes a and b of an already allocated cluster, we employ the following formula,

$$min\{ALAP(x) + r(x), ALAP(b)\} - max\{ASAP(x), ASAP(a) + r(a)\} \geq r(x), \tag{4.29}$$

where a and b are the consecutive nodes which are allocated in a processor before. From Equation 4.29, the possible starting and finishing time of x, *stime(x)* and *ftime(x)* respectively, are considered when x is allocated between a and b on the same processor. Node a can possibly start execution at the earliest $ASAP(a)$ and finish at $ASAP(a) + r(a)$. If x is executed after a, the *stime(x)* must be greater than or equal to the *ftime(a)*. Therefore, the starting time of x is given by,

$$stime(x) = max\{ASAP(x), ASAP(a) + r(a)\}. \tag{4.30}$$

Now, the possible finishing time of x is considered for x which need to be executed before b on the same processor. Node b can start execution at the latest time of $ALAP(b)$. If x is executed before b, the *ftime(x)* must be less than or equal to $ALAP(b)$. Thus, the finishing time of x, *ftime(x)*, is given by,

$$ftime(x) = min\{ALAP(x) + r(x), ALAP(b)\}. \tag{4.31}$$

Therefore, it is possible to allocate node x between a and b on the same processor if $ftime(x) - stime(x)$ is greater than $r(x)$. However, if b is a precedent of x, then x cannot be allocated before b.

Table 4.1: Comparing Complexities of different algorithms

Algorithm	*Complexity*
MCP[13]	$O(v^2 log\ v)$
MD [13]	$O(v^3)$
DSC[24]	$O((e+v)log\ v)$
DCP[14]	$O(v^3)$
FCS	$O(v^2)$

If there is no such processor and more processors are available, this cluster is allocated to a new processor. On the other hand if there are no more processors, the cluster is allocated to a processor i such that $|\tau_i - \beta_i|$ is minimum.

4.2.6 Complexity of the algorithm

The time complexity for computation of ASAP and ALAP is given by $O(v^2)$ [13], where v is the number of nodes in the task graph. The procedure FORM_A_CLUSTER takes $O(v - S)$ time. Therefore, the complexity of the clustering algorithm is $O(v^2)$. The complexity of the proposed algorithm is lower than those of other scheduling algorithms. We compare the complexities of scheduling algorithms in Table 4.1.

4.2.7 Gaussian Elimination

To demonstrate the proposed clustering mechanism, The Gaussian elimination application is used. The task graph of GE is shown in Figure 4.10. First, $ASAP$ and $ALAP$ of all nodes are calculated as discussed in Chapter 2. Nodes that have the same $ASAP$ and $ALAP$ times are identified to be in the critical path (CP). In Figure 4.10(a), nodes $n0$, $n4$, $n6$, $n9$, $n11$, $n13$, $n15$, $n16$, and $n17$ are on the CP. After CP nodes are identified, they are allocated to a single processor, called the main processor or P_0. The complexity of finding nodes of the CP is $O(|V|)$ where $|V|$ is the number of nodes

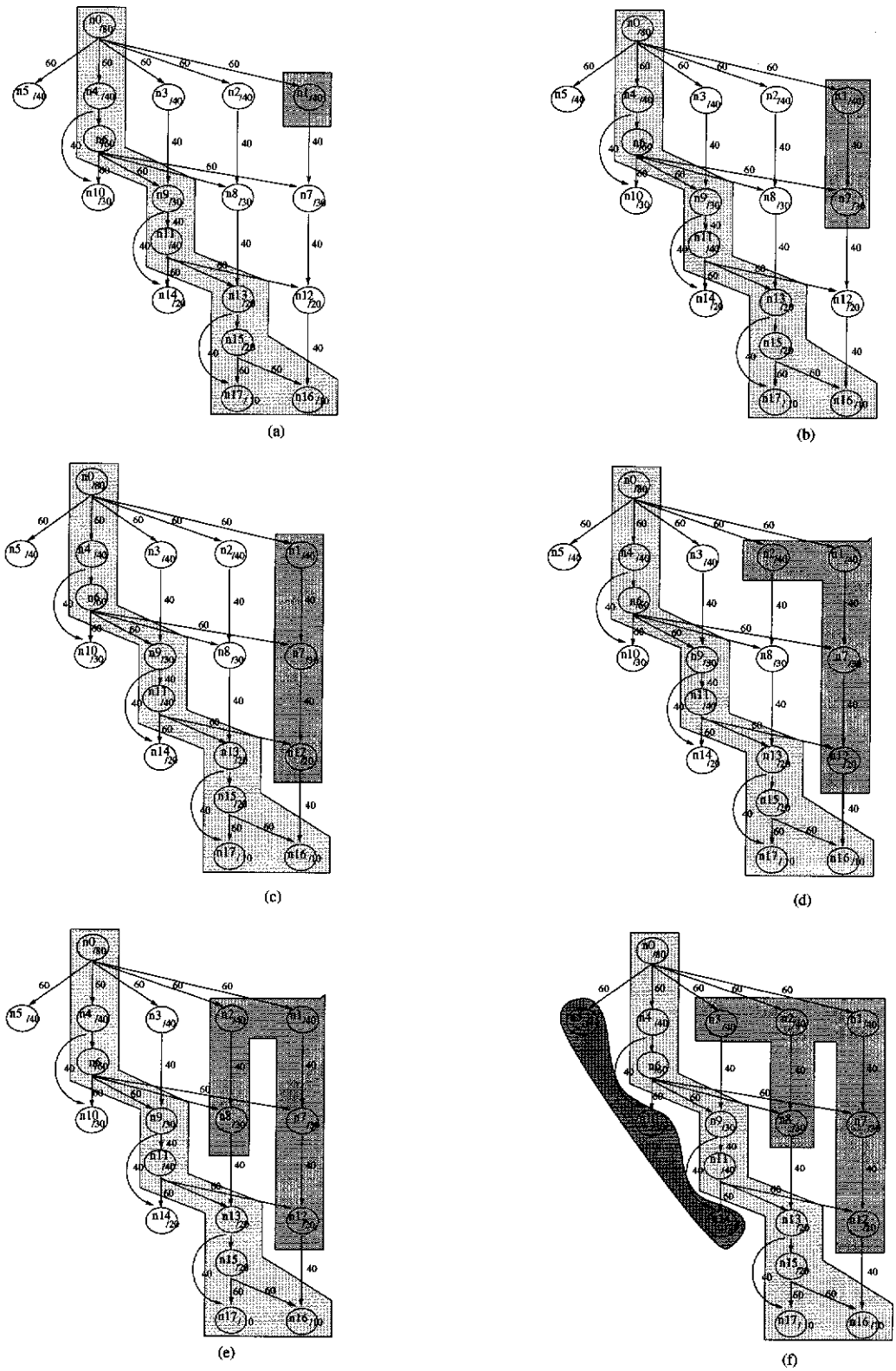


Figure 4.10: Gaussian elimination size of 4

Table 4.2: The clustering procedure of Gaussian elimination

List L	node considered	nodes in the cluster set CL	β	Comments
$\{n_1, n_2, n_3, n_5, n_7, n_8, n_{10}, n_{12}, n_{14}\}$	$x = n_1$	$\{n_1\}$	0.67	
$\{n_2, n_3, n_5, n_7, n_8, n_{10}, n_{12}, n_{14}\}$	$h(n_1) = n_7$	$\{n_1, n_7\}$	0.58	
$\{n_2, n_3, n_5, n_8, n_{10}, n_{12}, n_{14}\}$	$h^2(n_1) = n_{12}$	$\{n_1, n_7, n_{12}\}$	0.50	
$\{n_2, n_3, n_5, n_8, n_{10}, n_{14}\}$	$s(n_1) = n_2$	$\{n_1, n_2, n_7, n_{12}\}$	0.72	
$\{n_3, n_5, n_8, n_{10}, n_{14}\}$	$h(s(n_2)) = n_8$	$\{n_1, n_2, n_7, n_8, n_{12}\}$	0.89	
$\{n_3, n_5, n_{10}, n_{14}\}$	$s(n_1) = n_3$	$\{n_1, n_2, n_3, n_7, n_8, n_{12}\}$	1.11	This Cluster is assigned to PE1
$\{n_5, n_{10}, n_{14}\}$	$x = n_5$	$\{n_5\}$	0.67	
$\{n_{10}, n_{14}\}$	$n(n_5) = n_{10}$	$\{n_5, n_{10}\}$	0.44	
$\{s_{14}\}$	$n(n_5) = n_{14}$	$\{n_5, n_{10}, n_{14}\}$	0.35	This cluster is assigned to PE1

in the task graph. The remaining nodes that are not allocated on CP are considered for allocation to processors. A list which consists of the remaining nodes, $\{L = n1, n2, n3, n5, n7, n8, n10, n12, n14\}$, is generated and ordered lexicographically. The first node of the list, $n1$, is considered for clustering. The value of $\frac{r}{c}$, α, of $n1$ is calculated and compared with τ. If the α is greater than or equal to τ, the node forms a cluster, $cl(n1)$, and is allocated to a processor, otherwise, it is merged with other nodes forming a bigger cluster. For instance, node $n1$ is selected from the list L with its α of $\frac{40}{60}$. Suppose that the value of system parameters, τ, is 1. Then the α is less than τ, another node $n7$ which is a child node of $n1$ is selected for clustering with $n1$. Node $n7$, a child node of $n1$, is selected for clustering with $n1$ with the β of $\frac{7}{12}$. The β of $cl(n1, n7)$ is still less than τ. Therefore, node $n12$ which is a child node of $n7$ or $h^2(n1)$ is selected for clustering next. The process of β evaluation and cluster assignment to processors for the GE graph of Figure 4.10 is shown in Table 4.2.

4.2.8 Processor Selection

In the above example, if $\tau = 1$, nodes n_0, n_1, n_4, n_6 and n_7 are on the critical path and are allocated to PE0. After this allocation, the revised ASAP and ALAP times of the nodes are as shown in Table 4.3. Now, the FCS algorithm clusters n_2 and n_5 into one cluster with $\beta = 8/3 = 2.66$. In order to improve performance and optimise utilization, we examine the scheduling of cl(CP) on PE0 to identify possible idle times or gaps and check whether n_2 and n_5 can be fitted into these gaps. All pairs of allocated nodes are examined, for example, the gap or idle time of PE0 between n_1 and n_4 is

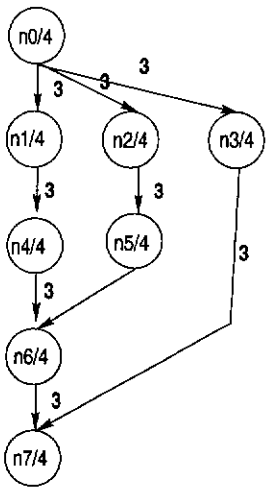


Figure 4.11: An example graph

Table 4.3: The ALAP and ASAP of nodes in Figure 4.11

	Essential		After nodes in CP are allocated		After n_2 is allocated	
Nodes	ASAP	ALAP	ASAP	ALAP	ASAP	ALAP
n0	0	0	0	0	0	0
n1	7	7	4	13	4	4
n2	7	7	7	13	8	8
n3	7	21	7	21	7	13
n4	14	14	8	17	12	12
n5	14	14	14	14	7	16
n6	21	21	21	21	16	16
n7	28	28	28	28	20	20

$min\{(13+4), 17\} - max\{8, (4+4)\} = 17 - 8 = 9$ which is greater than $r(n_2)$. Therefore n_2 can be allocated between n_1 and n_4 on PE0. After n_2 is allocated on PE0, the ASAP and ALAP of nodes are recalculated as shown in Table 4.3. Now n_5 is considered for inserting among nodes on PE0. At this stage, there is no idle time on PE0 large enough to execute node n_5. Therefore, the cluster which consists of n_2 and n_5 are allocated on a new processor, say PE1. In the next step, node n_3 forms a new cluster with its β, 1.33. PE0 is examined if there is an idle time available to fit n_3. From equation 4.31, the idle time of PE0 between n_1 and n_4 is $min\{(21 + 4), 17\} - max\{7, (4 + 4)\} = 9$ which is greater than $r(n_3)$. Therefore, n_3 is allocated to PE0 between n_1 and n_4. By doing this, the number of processors used is two with a parallel time of 26 units.

4.3 Conclusion

Task granularity and clustering are important issues in parallelization. Clustering reduces effective parallelism, but improves performance due to the reduction in communication overheads. In this chapter, the flexible clustering and scheduling algorithm (FCS) to determine efficient cluster size and schedule tasks onto processors was discussed. The algorithm can be adapted on a bounded or unbounded number of processors in order to achieve maximal efficiency or minimal execution time. The technique can be used to tune cluster size, based on such system characteristics as speed of processor and communication bandwidth. The FCS algorithm can be effectively used for implementing application problems on a variety of parallel computers and/or heterogeneous network of workstations. The mapping outcome of existing schemes may perform efficiently on one type of system and very poorly on another. In contrast, the FCS scheme provides a mapping that is most suitable for a given system. Task granularity and clustering are important issues in parallelization. Clustering may reduce effective parallelism, but improve performance due to the reduction in communication overheads. In the next chapter, simulation tests are performed to evaluate the performance of the FCS algorithm on many application problems.

Chapter 5

Simulation Results

In the previous chapter, the investigation of the theoretical aspects of the flexible clustering and scheduling algorithm(FCS) was presented. The FCS algorithm is based on the relation between the costs of computation and communication of clusters. The cluster size can be tuned to match different parallel systems having varying processor speeds and communication channel bandwidths. The communication costs among clusters are minimized and the computational load of each cluster is matched with the capacity of each processor element(PE). This chapter discusses the simulation results of several applications.

The experimental work involves the study of the performance characteristics of problems such as Gaussian elimination(GE), LU decomposition, Mean value analysis, and Floyd-Warshall's shortest path algorithm. Simulation results indicate that mapping, based on our technique, can be tuned to yield higher efficiency when compared with all other methods in the literature for most application problems.

The main objective in all cases is to determine the optimal number of processors to be employed for a given problem size and value of τ in order to minimize the normalized schedule length.

5.1 Introduction

In the experimental work, the directed acyclic task graph is represented by two sets of linked lists, called A and B. List A is used to represent all nodes in a task graph and list B represents relatives of nodes in list A and their computation costs. A single record of list A represents a single task of the graph. A record for each simple task comprises node number, computation cost, *ASAP*, *ALAP*, starting time, finishing time, and links to its parent and/or child nodes in list B. The structure is shown in Figure 5.1.

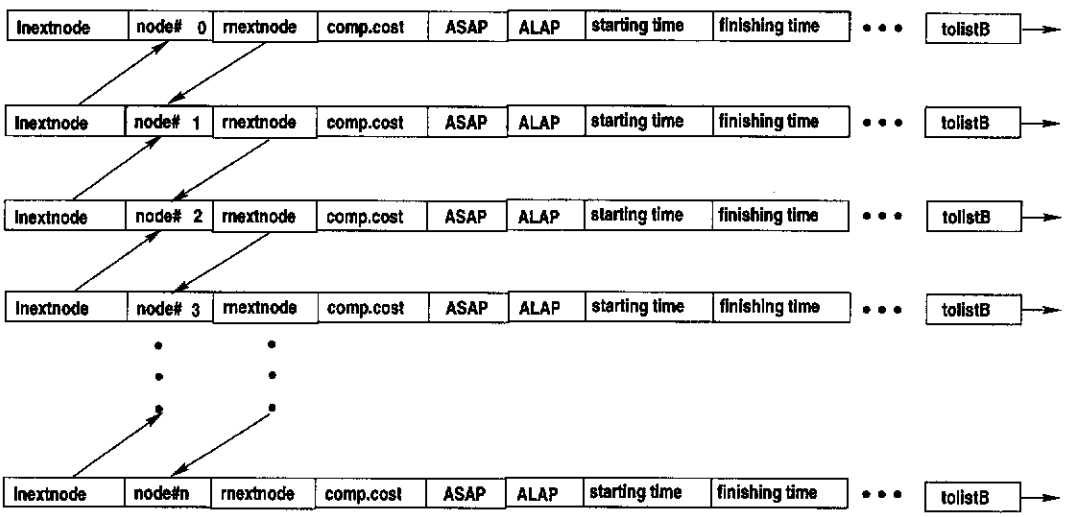


Figure 5.1: Representing nodes of a task graph

All nodes of list A are linked together and they can be traced back and forth by using pointers, *rnextnode* and *lnextnode*. The critical path procedure records the *ASAP* and *ALAP* values in list *A*. In the procedure of finding the critical path of a task graph, *ASAP* and *ALAP* are stored in each record. The node number is the index key field used to identify tasks in a task graph. During the process of scheduling nodes onto a processor, the starting and finishing times of each node are recorded. As shown in Figure 5.2, a node of list A has a set of parent and/or child nodes linked together with the pointers *rnextarc* and *lnextarc*. The *direction* field of each record is used to indicate

the direction of arcs in the task graphs. If node x is a parent of y, the direction of node x to y is positive. On the other hand, the direction of node y to x is negative. The communication cost of the link is recorded in the *comm.cost* field.

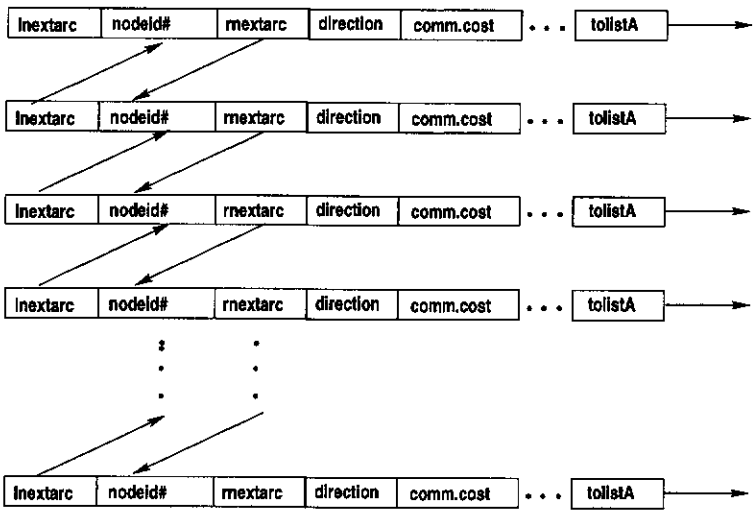


Figure 5.2: Representing communication among nodes

5.2 Gaussian Elimination

The Gaussian elimination algorithm is used to solve a system of linear equations [22, 49, 50, 51], and was discussed in Section 2.4.1. The linear equation is represented by $Ax = b$ where A is a dense $n \times n$ matrix of coefficients such that $A[i,j] = a_{i,j}$, b is an $n \times 1$ vector $[b_0, b_1, \ldots, b_{n-1}]$, and x is the desired solution vector $[x_0, x_1, \ldots, x_{n-1}]$ [22]. The original system, $Ax = b$ is manipulated and reduced to an upper triangular system in the form of $Ux = y$, where U is an upper-triangular matrix. All subdiagonal entries are zero or $U[i,j] = 0$ if $i > j$, and diagonal entries are equal to one or $U[i,j] = 1$ if $i = j$, otherwise, $U[i,j] = u_{i,j}$.

When the FCS algorithm is applied to a Gaussian elimination problem, the results of speedup are shown in Figure 5.3. Three different sizes of Gaussian elimination, 16, 32, and 64 are examined by using ten processors. From the figure, the speedups of all

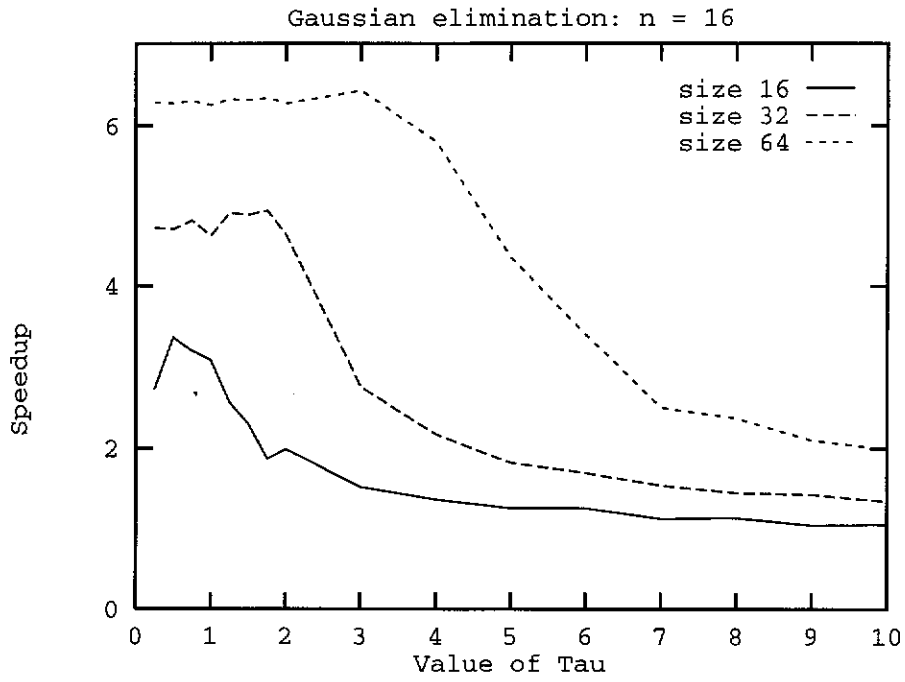


Figure 5.3: Speedup of Gaussian elimination

sizes are high when τ is small. When τ is small, meaning that the system has high communication channel bandwidth, tasks are likely to be distributed to many processors. Therefore tasks are divided into many small clusters and use many processors. As discussed in Chapter 4, the β of each cluster is roughly equal to τ because β is compared with τ when a node is collected into the cluster. When τ is small, many clusters are generated, the parallelism is high and many processors are employed causing high speedup. As already discussed, the emphasis of the FCS algorithm is the reduction in communication costs among tasks. By clustering a node and its relatives together communication costs can be reduced. On the other hand, when τ is high, the communication channel bandwidth of the system is low or the processor speed is high. Many tasks are collected into big clusters with a high value of β and hence numbers of processors used is less. Thus, the speedup is reduced. Although the speedup decreases with increase in τ, the efficiency may increase with less number of processors.

The efficiency results of the Gaussian elimination problem for sizes 16, 32 and 64 are

Table 5.1: Gaussian elimination: n = 16, 32, and 64

τ	Norm.Sched.Length			Efficiency			No. of PEs		
	16	32	64	16	32	64	16	32	64
0.25	1.64	1.69	2.41	0.27	0.47	0.63	10	10	10
0.50	1.33	1.70	2.41	0.34	0.47	0.63	10	10	10
0.75	1.39	1.66	2.40	0.32	0.48	0.63	10	10	10
1.00	1.45	1.73	2.42	0.31	0.46	0.63	10	10	10
1.25	1.74	1.63	2.39	0.26	0.49	0.63	10	10	10
1.50	1.93	1.63	2.39	0.23	0.49	0.63	10	10	10
1.75	2.38	1.62	2.38	0.19	0.49	0.63	10	10	10
2.00	2.24	1.72	2.41	0.25	0.46	0.63	8	10	10
3.00	2.92	2.91	2.35	0.30	0.28	0.64	5	10	10
4.00	3.25	3.68	2.60	0.46	0.22	0.58	3	10	10
5.00	3.53	4.40	3.47	0.63	0.18	0.44	2	10	10
6.00	3.53	4.71	4.46	0.63	0.17	0.34	2	10	10
7.00	3.95	5.18	6.04	0.56	0.15	0.25	2	10	10
8.00	3.91	5.53	6.39	0.57	0.16	0.24	2	9	10
9.00	4.25	5.60	7.19	0.52	0.18	0.21	2	8	10
10.00	4.21	5.92	7.59	0.53	0.19	0.20	2	7	10
11.00	3.79	6.02	8.11	0.59	0.22	0.19	2	6	10
12.00	4.44	6.42	8.57	1.00	0.25	0.18	1	5	10
13.00	4.44	6.57	9.03	1.00	0.24	0.17	1	5	10
14.00	4.44	6.58	9.54	1.00	0.24	0.16	1	5	10
15.00	4.44	6.78	9.76	1.00	0.30	0.15	1	4	10

shown in Table 5.1 and Figure 5.4. All problem instances used 1 to 10 processors based on the value of τ as indicated in Table 5.1. For example, when $\tau = 0.50$, the system uses 10 processors for all the three problem sizes and yields maximum speedup and minimum normalized schedule length for the problem size of 8. The efficiency increases with problem size when τ is between 0.25 and 2.0. On the other hand, when $\tau = 11.00$, the system uses two processors for n = 16, six processors for n = 32, and ten processors for n = 64. The efficiency decreases with problem size. The speedup variation is shown in Figure 5.4. It is clear from Table 5.1 and Figure 5.4, that for a given problem size, schedule length and maximal speedup can be achieved at one value of τ; in this case it is $\tau = 1.75$ for n = 32, and $\tau = 3.0$ for n = 64. After this point, if τ increases, the speedup reduces, but efficiency increases until $\tau = 12.0$, and then the system uses only one processor. This shows that when $\tau = 12.0$, (for n = 16) it is better to execute the whole problem on one processor rather than on multiple processors. When $\tau = 12.0$,

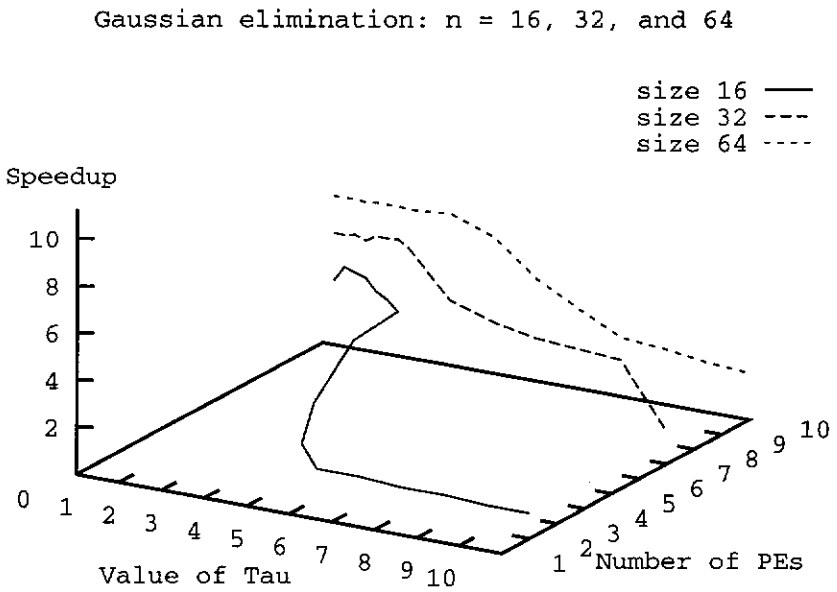


Figure 5.4: Speedup of Gaussian elimination with varying number of PEs

either the processor used is very powerful or the communication channel is very slow. In either case use of multiple processors is not recommended. Figure 5.4 in effect gives a clear indication of the versatility of the FCS scheme. Suppose we are given a τ value, we can easily determine the number of processors required to achieve a required speedup. Furthermore, we can determine the τ and number of PEs required to achieve a certain efficiency. For example if $\tau = 1.25$ and $n = 32$, then P = 10; if efficiency = 0.5 and n = 64, then $\tau < 4$ and P = 10.

Table 5.2: Comparison with other algorithms

Algorithms	Norm.Sched.Length			Efficiency			No. of PEs		
	16	32	64	16	32	64	16	32	64
MCP	1.03	1.41	1.43	0.25	0.17	0.16	16	32	64
MD	3.80	4.87	11.03	0.29	0.41	0.14	3	3	9

Comparing the results in Tables 5.1 and 5.2, it is clear that we can achieve very high efficiency (0.63) when $\tau = 0.25-3.00$ and N = 64 with ten processors. On the other hand with MCP[13], the efficiency is 0.16 when n = P = 64. The speedup with ten processors

```
Procedure_laplace()
  for( i=0; i < row; i++ )
    for( j=0; j < col; j++ )
      t[i][j] = 0.25 * ( t_old[i+1][j] + t_old[i-1][j] +
    t_old[i][j+1] + t_old[i][j-1]);
  dt = 0;
  for( i=0; i < row; i++ )
    for( j=0; j<col; j++ ){
      dt = max( abs(t[i][j]-t_old[i][j]), dt);
      t_old[i][j] = t[i][j];
    }
}
```

Figure 5.5: Laplace equation

employing the FCS algorithm is 6.3, whereas the speedup with 64 processors employing the MCP algorithm is 10.24 for problem size 64. The FCS algorithm, however, can be tuned to achieve greater speedup using more processors. The DCP algorithm for an unbounded number of processors [14] yields very low efficiency. The corresponding speedup and efficiency for the MD algorithm are 1.26 and 0.14 respectively with 9 processors.

5.3 Laplace Equation

Laplace equation is an elliptical partial differential equation of the form:

$$\sum_{i=1}^{N} \frac{\partial^2 \psi}{\partial x_i^2} = 0.$$

The equation occurs in scientific and engineering problems, for instance, in electrostatic potentials, fluid dynamics and heat transfers [26] [52].

The sequential Laplace equation program of two dimensions is shown in Figure 5.5 [26, 22]. The program solves the Laplace's equation over a rectangular grid. The number of iterations can be initialized to control the process of averaging. The final result is a new

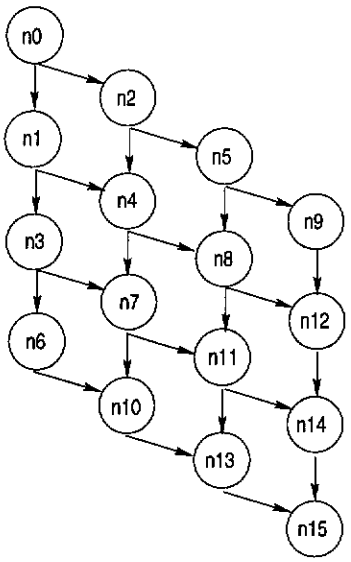


Figure 5.6: Laplace equation's task graph

rectangular grid with the value of averaging over the values of the four nearest neighbors which is called successive over-relaxation. By using the Jacobi iteration method, each grid point is updated from the previous values of its neighbors. The serial program can be divided into many subtasks, for instance, a single task represents a set of tasks which execute the following statements,

for(j=0; j < col; j++) {

t[i][j] = 0.25 * (t_{old}[i+1][j] + t_{old}[i-1][j] + t_{old}[i][j+1] + t_{old}[i][j-1]).

dt = max(abs(t[i][j]-t_{old}[i][j]), dt);

t_{old}[i][j] = t[i][j];

}.

A task graph representing the Laplace equation program of size four is shown in Figure 5.6. In the task graph, node n_0 computes the above task and sends t[0][0], t[0][1], t[0][2],...,t[0][col] to nodes n_1 and n_2. After that, nodes n_1 and n_2 execute loop instructions for *j=0* to *col* and *j=1* to *col* respectively by using data received from n_0. This process continues until *i=row*, at node n_{15}.

The speedup results of the Laplace equation for sizes 8, 16, and 32 are shown in Fig-

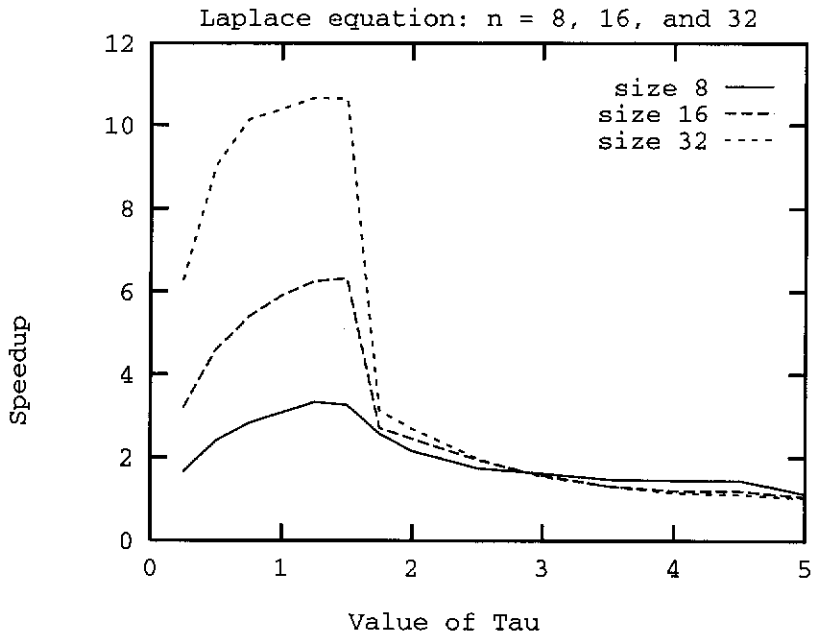


Figure 5.7: Speedup of Laplace equation

ure 5.7. When τ is small, a small number of tasks is gathered into a cluster and allocated to a processor. From the nature of the Laplace equation task graph, it is clear that inherent parallelism is low. Every node, except the entry node, receives data from two parent nodes. The communication costs among the tasks of this application are high. Clusters are distributed to many processors as shown in Table 5.3 with high speedup. When τ is increased which means that the communication channel bandwidth is low, the number of processors is reduced to minimize the communication cost. The complex communication structure of the problem also makes clustering effective in reducing costs. When a node, x, forms a cluster and includes a new node, y, into the cluster, generally a single communication between x and y, $c(x, y)$, is reduced to zero. In this application, a new communication is included every time a node is clustered, except for node at the rim of the graph. For example, in Figure 5.6 if n_2 and n_4 are grouped into a cluster, $c(n_2, n_4)$ is reduced to zero. In this application, only point-to-point communication is used because parent nodes send separate messages to each child. Although

Table 5.3: Laplace equation: n = 8, 16, and 32

τ	Norm.Sched.Length			Efficiency			No. of PEs		
	8	16	32	8	16	32	8	16	32
0.25	2.54	2.57	2.59	0.06	0.11	0.21	30	30	30
0.50	1.76	1.78	1.79	0.08	0.15	0.30	30	30	30
0.75	1.50	1.52	1.60	0.09	0.18	0.34	30	30	30
1.00	1.38	1.39	1.56	0.10	0.20	0.35	30	30	30
1.25	1.27	1.31	1.52	0.11	0.21	0.36	30	30	30
1.50	1.30	1.30	1.52	0.11	0.21	0.35	30	30	30
1.75	1.65	3.04	5.22	0.16	0.09	0.10	16	30	30
2.00	1.97	3.37	6.04	0.15	0.09	0.09	14	26	30
2.50	2.41	4.24	8.21	0.35	0.14	0.07	5	14	30
3.00	2.61	5.22	10.49	0.32	0.17	0.08	5	9	20
3.50	2.86	6.26	12.30	0.37	0.22	0.11	4	6	12
4.00	2.92	6.80	14.00	0.48	0.30	0.17	3	4	7
4.50	2.91	6.83	14.44	0.48	0.40	0.28	3	3	4
5.00	3.74	7.75	15.75	0.56	0.53	0.51	2	2	2
5.50	3.74	7.74	15.75	0.56	0.53	0.51	2	2	2
6.00	3.74	7.74	15.75	0.56	0.53	0.51	2	2	2
6.50	3.74	7.74	15.75	0.56	0.53	0.51	2	2	2

the speedup decreases when τ increases, the efficiency can be high because of using fewer processors to reduce communication costs.

The efficiency results of the Laplace equation implementation for problem sizes 8, 16, and 32 are shown in Table 5.3 and Figure 5.8. All problem instances use 2 to 30 processors based on the value of τ. When $\tau = 1.25$, the system uses 30 processors for all three problem sizes and yields maximum speedup and minimum normalized schedule length. The efficiency is greater at large problem sizes. In contrast, when $\tau = 2.50$, the system uses five processors for n = 8, fourteen processors for n = 16, and thirty processors for n = 32. Efficiency becomes less with problem size. The speedup variation is shown in Figure 5.8. It is also clear from Table 5.3 and Figure 5.8, that for a given problem size, minimal schedule length and maximal speedup can be achieved at one value of τ. In this case, the value of τ is 1.25 for all three problem sizes. As τ increases, the FCS algorithm uses fewer processors and hence increased efficiency is obtained. The use of a large number of processors for large values of τ for the same problem size does not yield higher speedup.

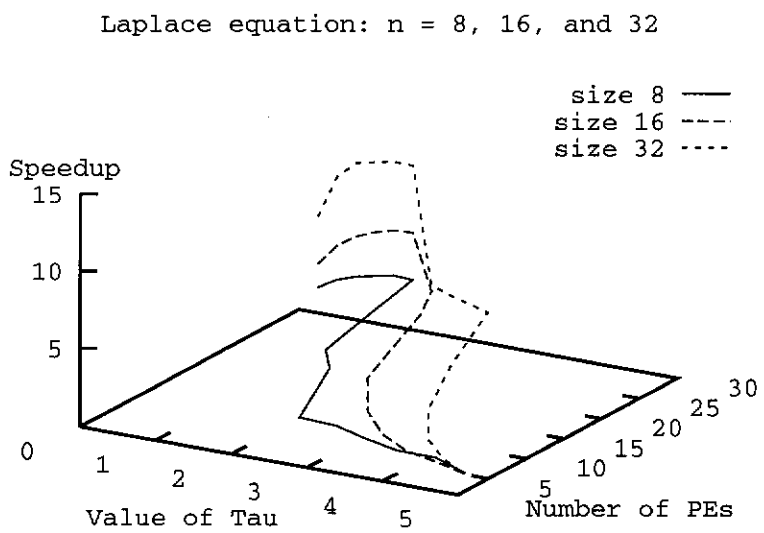


Figure 5.8: Speedup of Laplace equation with varying number of PEs

Table 5.4: Comparison for Laplace equation

Algorithms	Norm.Sched.Length			Efficiency			No. of PEs		
	8	16	32	8	16	32	8	16	32
MCP	1.30	1.50	-	0.41	0.34	-	7	15	-
MD	2.31	4.54	-	0.31	0.61	-	5	2	-

Comparing the results in Tables 5.4 and 5.3, the efficiency of the FCS is similar to that in the MD when the value of $\tau = 2.0 - 2.5$ and N = 8. The efficiency for the FCS algorithm is 0.35 and it is 0.31 for the MD algorithm with an identical number of processors. However, with the FCS algorithm, efficiency can be increased to 0.56 when $\tau = 5.0$ and two processors are used. The efficiency of MCP with n = 8 is 0.41 when 7 processors are used. Comparing FCS with MCP for n = 16, the efficiency of MCP is 0.34 with fifteen processors while FCS uses only four processors with $\tau = 4.0$ to achieve that efficiency. The desired efficiency can be achieved by adjusting the number of processors for a given value of τ. For instance, if $\tau = 5.0$ and n = 16, then efficiency of 0.5 can be achieved with two processors.

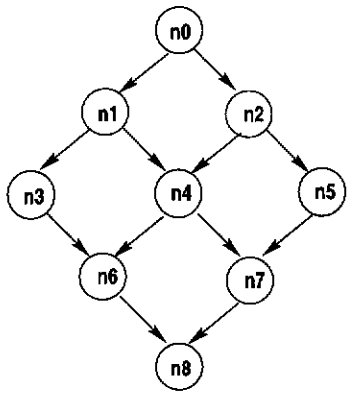


Figure 5.9: Mean value task graph

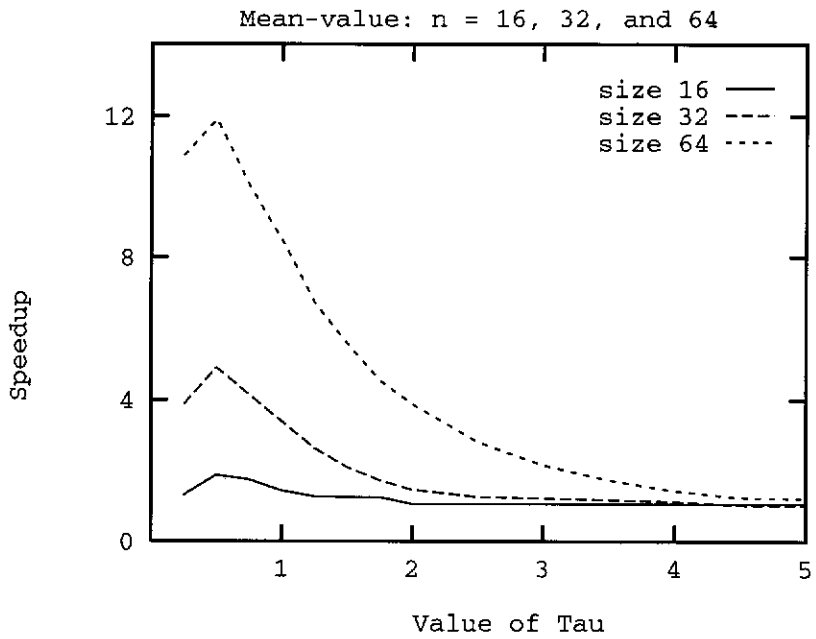


Figure 5.10: Speedup of mean value

5.4 The Mean Value Analysis

The Mean value analysis technique is a recursive algorithm used to analyze computer systems, the performance, mean response times, system throughput, and utilizations [53, 54]. The Mean value task graph is shown in Figure 5.9.

Table 5.5: Mean value: n = 16, 32, and 64

τ	Norm.Sched.Length			Efficiency			No. of PEs		
	16	32	64	16	32	64	16	32	64
0.25	4.52	2.92	2.02	0.04	0.13	0.36	30	30	30
0.50	3.16	2.31	1.85	0.06	0.16	0.40	30	30	30
0.75	3.40	2.73	2.19	0.06	0.14	0.33	30	30	30
1.00	4.13	3.33	2.59	0.10	0.11	0.28	14	30	30
1.25	4.61	4.27	3.27	0.22	0.09	0.22	6	30	30
1.50	4.70	5.33	3.93	0.32	0.07	0.19	4	30	30
1.75	4.69	6.49	4.83	0.32	0.06	0.15	4	30	30
2.00	5.51	7.64	5.67	0.54	0.05	0.13	2	30	30
2.50	5.51	8.82	7.81	0.54	0.11	0.09	2	12	30
3.00	5.51	9.13	10.07	0.54	0.14	0.07	2	9	30
3.50	5.51	9.49	12.58	0.54	0.17	0.06	2	7	30
4.00	5.51	9.84	15.19	0.54	0.29	0.05	2	4	30
4.50	5.51	10.84	17.44	0.54	0.52	0.04	2	2	30
5.00	5.51	10.84	17.83	0.54	0.52	0.05	2	2	26

The speedup results of Mean value analysis for sizes 16, 32, and 64 are shown in Figure 5.10. The speedup increases when the value of τ is less, between 0.25 and 0.75. When τ is small, a cluster consists of only a single node, if β of the node is greater than τ, then many clusters are distributed among a large number of processors. Therefore the speedup results of this stage are high. The speedup size 16 is not changed much through all values of τ because the number of processors used decreases when τ is increasing. When τ increases, many tasks are collected forming big clusters. Although the speedup of sizes 16 and 32 decreases when τ is decreasing, efficiency increases because fewer processors are used, shown in Table 5.5.

The efficiency results of Mean value analysis for problem sizes 16, 32, and 64 are shown in Table 5.5 and Figure 5.11. As shown in Table 5.5, different problem instances used 2 to 30 processors based on the value of τ. When $\tau = 0.5$, the system uses 30 processors for all the three problem sizes and yields maximum speedup and minimum normalized schedule length. Efficiency increases with problem size. On the other hand, when $\tau = 3$, the system uses two processors for n = 16, nine processors for n = 32, and thirty processors for n = 64; the efficiencies are 0.54, 0.14, and 0.09 respectively. The speedup

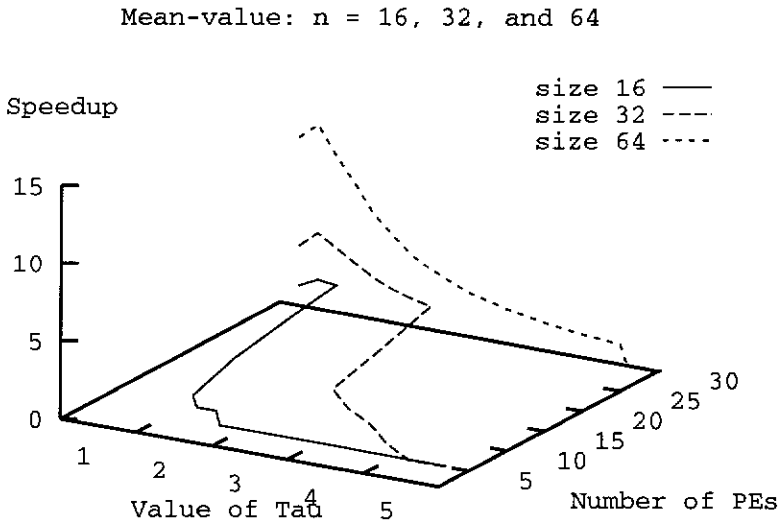


Figure 5.11: Speedup of mean value with varying number of PEs

Table 5.6: Comparison : Mean value

Algorithms	Norm.Sched.Length			Efficiency			No. of PEs		
	16	32	64	16	32	64	16	32	64
MCP	1.80	1.46	-	0.41	0.37	-	7	20	-
MD	2.51	-	-	0.20	-	-	11	-	-

is maximum when τ is low as shown in Figure 5.11. It obviously shows that for a given size problem, schedule length and maximum speedup can be achieved at value of τ; in this case $\tau = 0.5$ for three problem sizes. The maximal efficiencies for sizes n = 16, 32 and 64 are achieved when the value of τ is equal to 2.0, 4.5 and 0.4 respectively.

Comparing the results in Tables 5.5 and 5.6, we can see that the FCS algorithm can achieve high efficiency(0.54) when $\tau = 1.75 - 2.00$ and n = 16 with 2 processors while the efficiency for MCP algorithm is 0.41 with 7 processors. The efficiency of the MD algorithm when n = 16 is 0.20 with 11 processors. The corresponding speedups are 1.08 and 2.87 respectively. As can be seen from Figure 5.11, we can not achieve higher

speedups for small problem sizes with FCS. However, as problem size increases, the speedup increases as well. For example, we can achieve a speedup of 12 when $\tau = 0.5$ and P = 30 for n = 64.

5.5 Floyd-Warshall's Algorithm

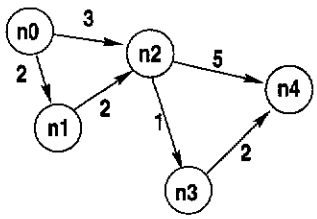


Figure 5.12: Example task graph

Table 5.7: The path between nodes in Figure 5.12

	n0	n1	n2	n3	n4
n0	0	2	3	∞	∞
n1	∞	0	2	∞	∞
n2	∞	∞	0	1	5
n3	∞	∞	∞	0	2
n4	∞	∞	∞	∞	0

(a) node n0 is the pivot

	n0	n1	n2	n3	n4
n0	0	2	3	4	8
n1	∞	0	2	3	7
n2	∞	∞	0	1	5
n3	∞	∞	∞	0	2
n4	∞	∞	∞	∞	0

(b) node n2 is the pivot

	n0	n1	n2	n3	n4
n0	0	2	3	4	6
n1	∞	0	2	3	5
n2	∞	∞	0	1	3
n3	∞	∞	∞	0	2
n4	∞	∞	∞	∞	0

(c) node n3 is the pivot

Floyd-Warshall's algorithm is used to find the shortest path between all pairs of nodes [55, 56]. The shortest path between each pair of nodes is considered by comparing the distance between two nodes and a pivot node. To find the shortest path between two nodes, each node is set as a pivot. Then the shortest path between two nodes is examined via this pivot node. If the old path between two nodes is greater than the path via

the pivot node, the path between two nodes is updated. For example, in Figure 5.12 before node $n2$ is set as a pivot node, the path between $n0$ to $n3$ is infinity, which means that there is no direct path from $n0$ to $n3$. When $n2$ is set as a pivot node, $n0$ can reach $n3$ via $n2$ with the path of 4. Then the shortest path between $n0$ and $n3$ is 4. All results of graphs in Figure 5.12 are displayed in Table 5.7(a),(b) and (c).

```
Procedure_fld()
  for(k = 0; k < N; k++)
    for(i = 0; i < N; i++)
      for(j = 0; j < N; j++)
        if(a[i][j] < a[i][k] + a[k][j])
          a[i][j] = a[i][k] + a[k][j];
      endfor
    endfor
  endfor
```

Figure 5.13: Floyd's algorithm

The serial Floyd-Warshall's application program is shown in Figure 5.13. To represent the Floyd-Warshall's application using a task graph, it is divided into small subprograms. In this case, a subprogram is represented by a single task as shown below.

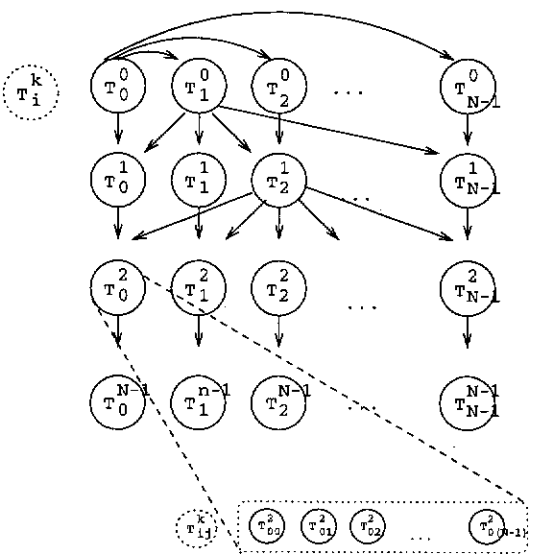


Figure 5.14: Floyd-Warshall's algorithm's taskgraph

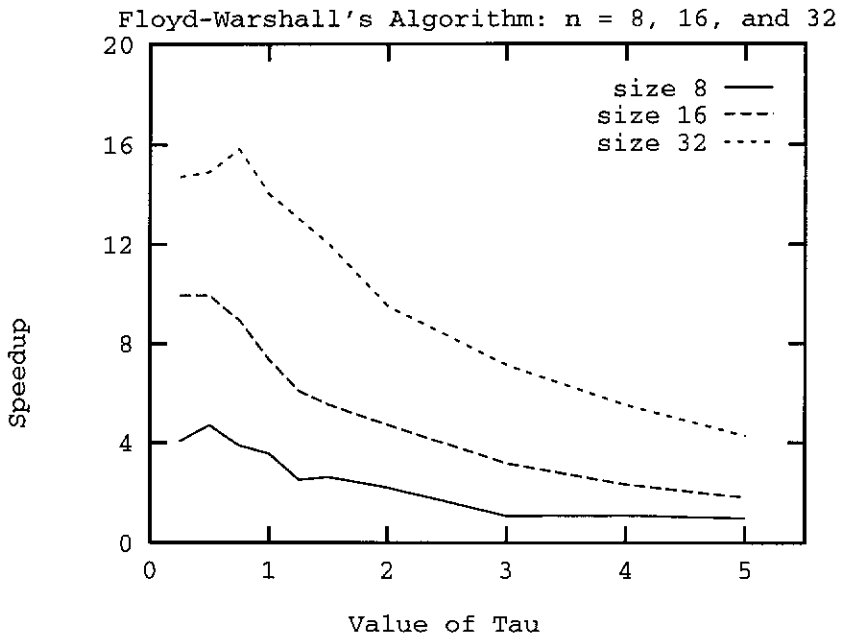


Figure 5.15: Speedup of Floyd's algorithm

$$\text{for}(j = 0; j < N; j++)$$
$$\text{if}(a[i][j] < a[i][k] + a[k][j])$$
$$a[i][j] = a[i][k] + a[k][j];$$

From the subprogram, a single task executes two data sets, $a[i][k]$ and $a[k][j]$. Each node executes a row of data array, $a[i][j]$ for $j = 0$ to N where N is the problem size which is stored in the node. A data set of $a[k][j]$ where $j = 0$ to N is received from a node that the data set is stored. For example, when $k = 1$, a row of $a[1][0]$ to $a[1][N]$ is broadcast to every node from the node that holds the data set. The Floyd-Warshall's problem's taskgraph is shown in Figure 5.14.

The speedup results of Floyd-Warshall's problem for sizes 8, 16 and 32 are shown in Figure 5.15. The speedup is high when τ is between 0.25 and 0.75. Thus each cluster size is small and many clusters are distributed to many processors resulting in high speedup. In the Gaussian elimination problem, the size of data executed in each loop is reduced in a triangular form. In Floyd-Warshall's problem, the size of data executed

Table 5.8: Floyd-Warshall's algorithm: n = 8, 16, and 32

τ	Norm.Sched.Length			Efficiency			No. of PEs		
	8	16	32	8	16	32	8	16	32
0.25	1.60	1.37	1.89	0.14	0.33	0.49	30	30	30
0.50	1.39	1.37	1.86	0.16	0.33	0.50	30	30	30
0.75	1.67	1.52	1.76	0.35	0.30	0.53	11	30	30
1.00	1.83	1.86	1.98	0.40	0.37	0.47	9	20	30
1.25	2.57	2.24	2.14	0.42	0.40	0.43	6	15	30
1.50	2.46	2.47	2.31	0.53	0.46	0.44	5	12	27
2.00	2.92	2.89	2.92	0.56	0.52	0.48	4	9	20
3.00	6.00	4.28	3.91	0.54	0.53	0.55	2	6	13
4.00	5.90	5.79	5.06	0.55	0.59	0.55	2	4	10
5.00	6.50	7.47	6.51	1.00	0.61	0.61	1	3	7
6.00	6.50	9.07	8.09	1.00	0.50	0.57	1	3	6
7.00	6.50	13.00	9.90	1.00	0.52	0.56	1	2	5
8.00	6.50	12.95	11.89	1.00	0.52	0.58	1	2	4
9.00	6.50	12.84	14.52	1.00	0.53	0.48	1	2	4
10.00	6.50	11.65	16.59	1.00	0.58	0.42	1	2	4
11.00	6.50	13.58	14.68	1.00	1.00	0.63	1	1	3
12.00	6.50	13.58	16.09	1.00	1.00	0.58	1	1	3

is always the same.

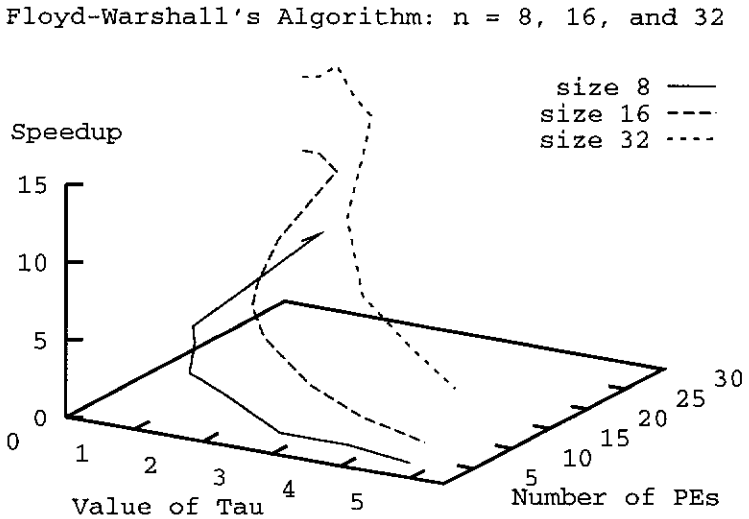


Figure 5.16: Speedup of Floyd's algorithm with varying number of PEs

The efficiency results of Floyd-Warshall's algorithm for sizes 8, 16, and 32 are shown in Table 5.8 and Figure 5.16. When $\tau = 0.5$, the system uses 30 processors and yields minimal normalized schedule length for sizes 8 and 16. For n = 32, the minimal normalized schedule length is achieved at $\tau = 0.75$ as shown in Table 5.8. In contrast when $\tau = 1.5$ and 2.0 the efficiency decreases with problem size. The number of processors used is five for n = 8, and twelve for n = 16, and twenty seven for n = 32. The speedup variation is shown in Figure 5.16. The maximal speedup and schedule length are achieved at $\tau = 2.0$ for n = 8, $\tau = 5$ for n = 16, and $\tau = 11$ for n = 32.

Table 5.9: Comparison of Floyd's algorithm results with other algorithms

Algorithms	Norm.Sched.Length			Efficiency			No. of PEs		
	8	16	32	8	16	32	8	16	32
MCP	1.96	2.01	-	0.47	0.45	-	6	14	-
MD	3.71	6.32	-	0.35	0.13	-	4	16	-

From Tables 5.8 and 5.9 we can see that FCS achieves very high efficiency of 0.61 when $\tau = 5.0$ and n = 16 with three processors, while the efficiency with MCP is 0.45 using 14 processors. The speedup with 14 processors employing the MCP algorithm is 6.30 for problem size of 16. For the same problem, FCS can achieve a speedup of 9.9 with 30 processors and 5.5 with 12 processors. The corresponding speedup and efficiency for the MD algorithm are 2.08 and 0.13 with 16 processors.

5.6 The LU Decomposition Algorithm

The LU decomposition algorithm breaks down a matrix into two separate matrices. These matrices are called the upper and lower triangular matrices [7, 19, 20]. A system with n equations and n unknowns can be represented in matrix notation as a single equation, $Ax = b$. A is an $n-element$ vector of variables x and b is $n-element$ vector constants. A can be factored into two matrices L and U, $A = LU$. After A is broken down into two components, x can be computed. Since $A \cdot x = b$ and $A = L \cdot U$, We have $U \cdot x = y$, $L \cdot y = b$. The serial algorithm of LU decomposition is shown in Figure 5.17.

```
Procedure_LU()
  for k = 0 to size
    for j = k+1 to size
      A(k,j) = A(k,j)/A(k,k)
    endfor
    for i = k+1 to size
      for j = k+1 to size
        A(i,j) = A(i,j) - A(i,k)*A(k,j)
      endfor
    endfor
  endfor
```

Figure 5.17: LU decomposition algorithm

From the LU decomposition algorithm, a task graph is used to illustrate the whole process. The algorithm is analyzed and divided into finer parts. Each of these is represented by a single node in the task graph. The relationships among nodes are shown by using arcs in the task graph. Figure 5.18 displays the LU decomposition task graph size of 4. Node $n0, n4$ and $n7$ represent a set of nodes executing the following statements;

$$for\ j = k + 1\ to\ size$$

$$A(k, j) = A(k, j)/A(k, k).$$

Although the nodes execute the same statements, different values of i, j, and k are assigned and they execute in different stages. After they complete the tasks, data $A(k, j)$ is sent to child nodes for execution in the next step. The second set of nodes $n1, n2, n3, n5, n6$ and $n8$ compute the following statements after they have received all data from their parents;

$$for\ j = k + 1\ to\ size$$

$$A(i, j) = A(i, j) - A(i, k) * A(k, j).$$

The speedup results of LU decomposition problem for sizes 8, 16 and 24 are shown in

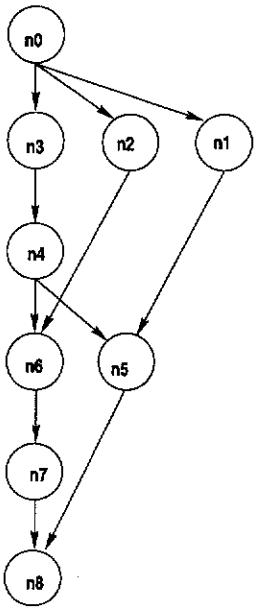


Figure 5.18: LU taskgraph

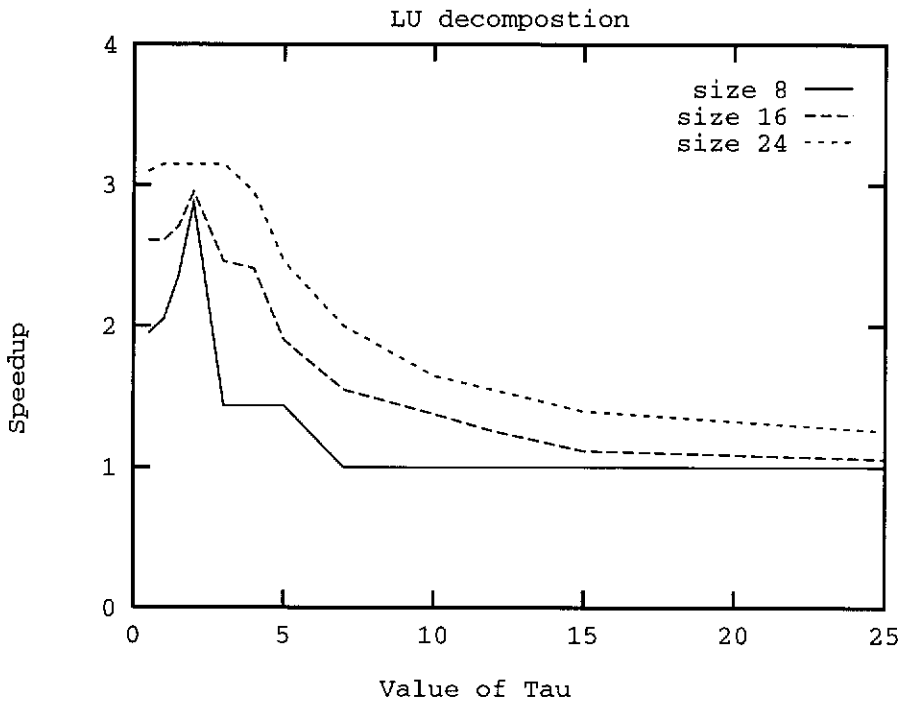


Figure 5.19: Speedup of LU decomposition

Figure 5.19. The speedups for problem sizes 8 and 16 increase for τ between 0.5 and 2. When τ is less, large number of clusters are distributed to many processors. Thus the communication cost of the execution time among processors is high. When the size of

Table 5.10: LU decomposition: n = 8, 16, and 24

τ	Norm.Sched.Length			Efficiency			No. of PEs		
	8	16	24	8	16	24	8	16	24
0.50	1.30	1.66	2.12	0.39	0.52	0.62	5	5	5
1.00	1.26	1.61	2.11	0.41	0.52	0.63	5	5	5
1.50	1.28	1.57	2.09	0.47	0.55	0.63	5	5	5
2.00	1.50	1.47	2.11	0.57	0.59	0.63	5	5	5
3.00	1.77	1.76	2.09	0.72	0.49	0.63	2	5	5
4.00	1.76	2.16	2.25	0.72	0.40	0.59	2	5	5
5.00	1.76	2.31	2.27	0.72	0.38	0.49	2	5	5
7.00	2.56	2.83	3.31	1.00	0.31	0.40	1	5	5
10.00	-	3.11	3.97	-	0.46	0.33	-	3	5
12.00	-	3.42	4.33	-	0.63	0.31	-	2	5
15.00	-	3.87	4.75	-	0.56	0.28	-	2	5
25.00	-	4.08	5.31	-	0.53	0.42	-	2	3

the cluster is increased because of the increasing of τ, the communication cost among tasks is less and the speedup increases. When τ is less than 2, as shown in Table 5.10, the speedup decreases because the number of processors used is fewer. Although the speedup decreases, efficiency increases as shown in Table 5.10 and Figure 5.20 because of fewer processors.

The efficiency results of LU decomposition problem for sizes 8, 16, and 24 are shown in Table 5.10. From the table, different problem instances used 1 to 5 processors based on the value of τ. For example, when $\tau = 0.50$, the system uses 5 processors for all the three problem sizes. Efficiency increases with problem size when τ is between 0.50 and 2.0. On the other hand, when $\tau = 3.0$, the system uses two processors for n = 8 and five processors for n = 16 and 24. It is clear from Table 5.10 that for a given problem size, minimal schedule length and maximal speedup can be achieved at one value of τ. Fewer processors are used if τ increases. When $\tau \geq 7.0$, it is better to execute the whole problem on one processor. For the size 16, the highest efficiency occurs when $\tau = 12$ using only two processors. From the table, it is clear that when the problem size is small, using one or two processors results in high efficiency. In contrast with the large problem size, for instance 24, using many processors if available results in high efficiency.

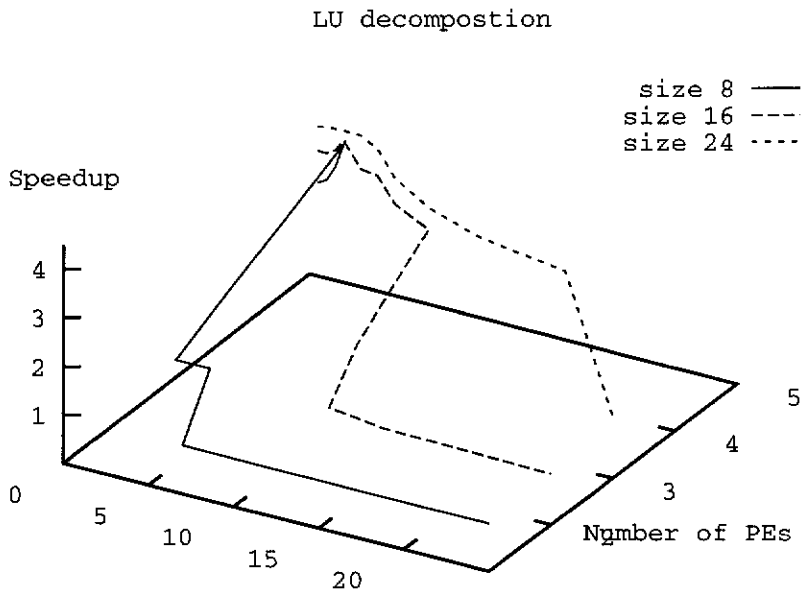


Figure 5.20: Speedup of LU decomposition with varying number of PEs

5.7 Conclusion

This chapter presents the simulation results of implementing different application problems, such as Gaussian elimination, the Laplace equation, Mean value analysis, the Floyd-Warshall's algorithm, and LU decomposition. In each case, the implementation can be tuned to achieve high efficiency or minimal execution time. The results in this chapter show the versatility of the FCS algorithm under various conditions. It can implement application problems on a variety of parallel computers and/or heterogeneous network of workstations. In the next chapter, implementation of the above applications on network of workstations using messages passing interface(MPI) will be discussed.

Chapter 6

Implementatation Results

In the last chapter, the experimental results of applying the FCS algorithm to different application problems such as Gaussian elimination, Laplace equation, Mean value analysis, Floyd-Warshall's algorithm, and LU decomposition, are presented. The implementation results show that by using the FCS algorithm, tasks are gathered into clusters of different sizes. The size of each cluster depends on the system parameters which can vary from system to system. In this chapter, some application problems are implemented on a closed network consisting of four processors for experimental studies. Each of them runs Linux as the operating system and supports the local area multicomputer (LAM) parallel processing software using messages passing interface (MPI).

6.1 Introduction

In our experiments, the parallel system comprises four PCs running the Linux operating system [57, 58, 59, 60] connected as a dedicated cluster. They are connected by using star network topology as shown in Figure 6.1. To support parallel processing, a software called Local Area Multicomputer (LAM) is used. LAM is an MPI

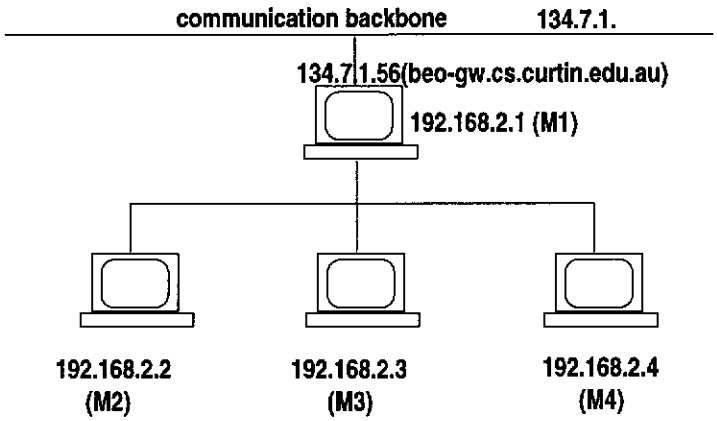


Figure 6.1: The connection of a cluster

Table 6.1: The communication cost of sending a message on the cluster

Sending Pattern	Time in Seconds
512 × 1053	0.444
1024 × 977	0.368
2048 × 488	0.354

programming environment and development system for heterogeneous computers on a network [61, 62, 63]. It features a full implementation of Messages-Passing Interface (MPI) programming standard. It is also a parallel processing environment and development system for a network of independent computers, supporting monitoring and debugging [64]. LAM runs as a UNIX daemon on each computer, which is structured as a nano-kernel and hand-threaded virtual process [61]. LAM commands can be divided into three parts, setting up a system, executing, monitoring. Before running a parallel application software, the LAM daemon is booted by using *lamboot* command. The system can concurrently execute one or more programs.

6.2 System Parameter of the System

As discussed in Section 4.1.1, the system parameters which are the execution speed of processors (ρ) and the communication bandwidth (γ) affect the speedup or efficiency of a system. These parameters vary from system to system, especially, in a network

of workstations (NOWs). In this experimental system, the communication costs are determined experimentally and typical costs are shown in Table 6.1.

6.3 Experimental Results

In this experiment, Gaussian elimination, Floyd-Warshall's shortest path, and Laplace equation applications are implemented. Before each application problem executes on the parallel system, tasks are collected into clusters by using the FCS algorithm as already discussed in Chapter 5. From the results of the clustering and scheduling, tasks are assigned to and queued at each processor waiting for execution.

6.3.1 Gaussian Elimination

As discussed in Sections 2.4.1 and 5.2, Gaussian elimination task graph is generated from its serial program. Then FCS algorithm is applied for clustering and scheduling the task graph. After the clustering and scheduling procedure, each processor receives a task queue. In this case, the experimental system consists of four processors executing in parallel using message passing interface for communications. The results of executing Gaussian elimination for sizes of 16, 64, 96 and 256 are shown in Table 6.2.

Table 6.2: The execution time of Gaussian elimination

Problem size	Execution time in Seconds	
	4 PEs	3 PEs
16	0.026	0.060
64	0.351	0.398
96	0.782	1.029
256	5.419	5.929

It is noticeable that the execution times with 3 or 4 processors are almost similar. This is due to the large size of the problem when each cluster is very large and the parallelism is low. Execution time will be reduced if more processors are employed.

6.3.2 Floyd-Warshall's Algorithm

As already discussed in Section 5.5, Floyd-Warshall's algorithm is used for finding the shortest path of a pair nodes. After the task graph of Floyd-Warshall's algorithm is generated, the FCS algorithm is employed for clustering and scheduling by adapting to suit the system parameters. The results of these experiment are shown in Table 6.3.

Table 6.3: The execution time of Floyd's algorithm

Problem size	Execution time in Seconds	
	4 PEs	3 PEs
8	0.053	0.050
16	0.302	0.329
32	2.807	5.433
48	20.786	21.230

6.3.3 Laplace Equation

In the Laplace equation problem task graph discussed in Section 5.3, nodes are collected using the FCS algorithm into clusters and assigned to processors. The value of τ is set to 0.3 and the number of processors is four. In a Laplace equation, problem of two dimensions, each data in an array is modified by finding the mean of data from four neighbours, for example, $array(i,j) = 0.25 * (array[i-1][j] + array[i+1][j]) + array[i][j-1] + array[i][j+1])$. The results of solving the Laplace equation using four processors with different sizes on the system when $\tau = 0.3$ are shown in Table 6.4.

Table 6.4: The execution time of Laplace equation problem

Problem size	Execution time in Seconds	
	4 PEs	3 PEs
16	0.079	0.144
32	0.100	0.256
64	0.555	0.827

6.4 Discussion

When an application is executed serially, only computation procedure is considered. For example, a single task of Gaussian elimination is represented by following statements discussed in Section 2.4.1,

$$\begin{aligned}
&\textbf{for}(j = k+1; j < n; j++)\\
&\quad A[k][j] = A[k][j]/A[k][k]\\
&y[k] = b[k]/A[k][k]\\
&A[k][k] = 1.
\end{aligned}$$

For serial computation, task graph is not required for consideration of clustering and scheduling. Only an array of data input is required for execution. In contrast with parallel processing, the application is analyzed for clustering and scheduling by using task graph representing the computation and communication among tasks of the application. A task graph of an application problem can be represented by using arrays or linked lists. The procedures executed to prior to the execution of a tasks are as follows.

i) Check if there are any parent tasks. This step is used for checking if the execution tasks requires input data from other tasks. If there is a parent node then the synchronization might be required before execution. The current node has to wait until the parent node finishes its execution.

ii) Location of parent task. Communication is involved when the parent task is on another processor, otherwise, the communication between two nodes is zero.

iii) Packing and unpacking data.

All of the above procedures require time for execution. This is one reason why parallel processing always takes some more extra time for completion than serial processing. For example, in the first step of checking if there are any parent nodes, a searching

procedure is required. The searching time depends on the searching algorithm used and the problem size. If the problem size is large, each node is likely to have many parents.

The limitation of number of processors is another factor related to the results. In the case of large problem size, if the goal is to achieve small parallel time, many processors should be employed. Although LAM software can generate many simulated processors from a real processor, the execution time using a number of real processors is different from that using simulated processors. For example, Laplace equation size 64 is executed on a real four processor system with the time of 0.555 seconds but it takes 1.338 seconds if executed using four simulated processors on one machine. Therefore, although many processors can be generated on the system with real four processors but the execution time results is different from using the same number of real processors.

6.5 Conclusion

In this chapter we presented results of experiments with the FCS algorithm for implementing application problem, Guassian elimination, Laplace equation, and Floyd-Warshall's shortest path. The system used for experiment consists of four processors connected in a closed network using star topology. The local area multicomputer (LAM) parallel software is used for handling communication process using message passing interface (MPI). The value of τ for the system is calculated by timing the computation of processors and communication among them, task graphs of the problem are clustered and scheduled by using the FCS algorithm. Our experimental results validate simulation results presented in Chapter 5

Chapter 7

Conclusions

This thesis proposed a new algorithm for clustering and scheduling application problems onto parallel computers. A directed acyclic task graph is employed to represent the subprogram or tasks and the inter-task relations of a given application problem. The proposed FCS algorithm transforms the resulting task graph into a set of clusters for allocation and scheduling onto parallel computers. The low complexity FCS collects tasks into clusters in order to satisfy certain given criteria. The novel feature of this scheme is the adaptation to any given computing environment. The FCS algorithm performs the clustering and scheduling tasks in a flexible and adaptive fashion and can be used for static as well as dynamic cases.

In the last decade, many clustering and scheduling algorithms are developed in order to achieve high speedup and/or efficiency. Some algorithms consider nodes in critical path as high priority nodes for selection to schedule. If clustering is not employed for scheduling, a single node at a time is selected, scheduled, and allocated to an appropriate processor. After a node is allocated to a processor, the ASAP and ALAP of the remaining nodes are recomputed in order to find nodes in the new critical path. On the other hand, when clustering is used, nodes are collected into a cluster until the size of the cluster approaches the desired granularity, then the cluster is allocated to

a processor. The FCS algorithm differs from all previously proposed schemes because system parameters are taken into account for clustering and scheduling in the FCS algorithm.

The system parameters are the capacities of the system which consist of the computation speed of processors in the system and the communication bandwidths. The computation processor speed affects clustering and scheduling in terms of number of nodes in each cluster. For systems with high speed processors, the number of nodes in each cluster can be large. In other words, each processor can be loaded with a large number of tasks and the processors can be expected to finish the job within a short period of time. On the other hand, a small number of tasks are gathered into a cluster for low speed processors. The communication bandwidth is another factor affecting the clustering and scheduling. In a system with high communication bandwidth, data can be transferred among processors at high speed. Then many processors can be used for executing an application problem to reduce the parallel execution time. In contrast, with a system that has low speed communication bandwidth, if many processors are employed, times for communication are high, then the parallel execution time increases. Because each system has different system parameters, the clustering and scheduling should be different for each system. In other words, clustering and scheduling should be adapted in a flexible fashion to suit the given environment.

The FCS algorithm considers the system parameters for clustering in order to be able to adjust the size of each cluster. By taking the system parameters, the size of each cluster which is the ratio of the computation and communication cost of clusters (β), is compared to the system parameters (τ). Nodes of an application problem are collected into a cluster until the β is comparable to τ. Then the cluster is allocated to an available processor. When a single node is selected for clustering and β is less than τ, the relatives of the node are selected next. The sequence of nodes start with precedents, successors, sibling, precedent of sibling, and so on. When the relative nodes are clustered into the same cluster, the communication costs among them are reduced. As the total execution

time of an application problem consists of computation and communication costs, the total execution time reduces with communication costs. Furthermore, the complexity of the FCS algorithm is lower than those of most existing algorithms.

In our simulation experiments, the FCS algorithm is applied for clustering and scheduling of many application problems such as Gaussian elimination, Laplace equation, LU decomposition, Mean value analysis, and Floyd-Warshall's shortest path problem, for different sizes. FCS performance for varying τ is evaluated in terms of speedup and efficiency with different number of processors. The results show that when τ is less, the number of nodes in each cluster is less. However, it is also dependent on the computation cost of each node and communication among tasks of task graphs. The speedup when τ is less is high when compared with that of the larger τ. As already discussed, when τ is less, the communication cost among processors are low or the communication bandwidth is high and small number of tasks are collected into each cluster. When large number of processors are used for execution, then the speedup is high. In contrast, when τ is high, many tasks are gathered in each cluster with high value of β. However, the efficiency increases with τ when fewer processors are used.

Problem size is another point that affects the speedup and efficiency. The simulation results show that when the problem size increases the speedup also increases, especially, when τ is less. When problem size is large the number of clusters is larger. Which means that when problem size is large, many more processors are used and tasks are distributed to many processors. Therefore, more number of processors should be used when the problem size is large.

The FCS algorithm is tested on a closed network system consisting of four processors connected in the form of a star topology. The local area multicomputer (LAM) parallel software is used for communication among processors by using message passing interface (MPI). Three application problems, Gaussian elimination, Laplace equation, and Floyd-Warshall's shortest path, are executed in parallel by using the FCS algori-

htm for different problem sizes. The results show parallel execution times using three and four processors. When using four processors, the execution time is less than that using three processors. As discussed before, when the problem size is large and the system parameters (τ) is low, many processors must be used for distributing the large number of clusters. Although the LAM parallel software can generate many simulated processors on a real processor, the execution time of an application using simulated processors is obviously greater than that using the same number of real processors.

To summarise, in this thesis we presented the design, development, and performance evaluation aspects of a novel scheme for clustering and scheduling of application problem graphs onto parallel computers. The proposed FCS algorithm is efficient and flexible. It can be tuned to adapt itself to create clusters of different sizes in order to match processor speeds and communication bandwidths. The FCS algorithm can be tuned to achieve high speedup or efficiency on bounded or unbounded number of processors. Most importantly, it can be applied to static or dynamic scheduling schemes, can be easily extended to achieve load balancing. This work can be further extended to assess the suitability of the FCS algorithm for real-time and fault-tolerant computing.

Bibliography

[1] D. I. Moldovan, *Parallel Processing: From Applications to Systems.* California: Morgan Kaufmann Publishess, 1993.

[2] B. Shirazi, A. Hurson, and K. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.* California: IEEE Computer Society Press, 1995.

[3] H. El-Rewini, T. Lewis, and H. Ali, *Task Scheduling in Parallel and Distributed Systems.* New Jersey: PTR Prentice Hall, 1994.

[4] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 686–701, June 1993.

[5] T. Lewis and H. El-Rewini, *Introduction to Parallel Computing.* New Jeysey, USA: Prentice Hall, 1992.

[6] M. Quinn, *Parallel Computing Theory and Practice.* New York: McGraw-Hill, Inc., 1994.

[7] T. Casavant, P. Tvrdik, and F. Plasil, *Parallel Computers : Theory and Practice.* California: IEEE Computer Society Press, 1996.

[8] M. Cosnard and D. Trystram, *Parallel Algorithms and Architectures.* London: International Thomson Computer Press, 1995.

[9] M. Wu, "On runtime parallel schedulling for processor load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, pp. 173–186, February 1997.

[10] S. Cvetanovic, "The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems," *IEEE Transaction on Computers*, vol. C-36, pp. 421–432, April 1987.

[11] I. Foster, *Designing and Building Parallel Program.* New York: Addison-Wesley Publishing Company Inc., 1994.

[12] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, Massachusetts, USA: The MIT Press, 1989.

[13] M. Wu and D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 330–343, July 1990.

[14] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, May 1996.

[15] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 276–291, 1992.

[16] S. Salleh and A.Y.Zomaya, *Scheduling in Parallel Computating Systems : Fuzzy and Annealing Techniques.* Massachusetts, USA: Kluwer Academic Publishers, 1999.

[17] A. Fleischmann, *Distributed Systems Software Design and Implementation.* Berlin, Germany: Springer-verlag, 1994.

[18] V. Lo, "Heuristic algotirhtms for task assignment in distributed systems," *IEEE Transaction on Computers*, vol. 37, pp. 1384–1397, November 1988.

[19] J. JaJa, *An Introduction to Parallel Algorithms.* Massachusetts: Addison-Wesley Publishing Company, 1992.

[20] L. Kronsjo and D.Shumsheruddin, *Advances in Parallel Algorithms.* London: Blackwell Scientific Publications, 1992.

[21] S. Chingchit, M. Kumar, and L.N.Bhuyan, "A flexible clustering and scheduling scheme for efficient parallel computation," in *13th International and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99)*, vol. 1, pp. 500–505, April 1999.

[22] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* California: The Benjamin Cummings, 1994.

[23] F. Ozguner and F.Erçal, *Parallel computing on Distributed Memory Multiprocessors.* New York: Springer-Verlag, 1991.

[24] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, September 1994.

[25] A. Zomaya, M.Clements, and S.Olariu, "A framework for reinforcement-based scheduling in parallel processor systems," *IEEE Transaction on Parallel and Distributed Systems*, vol. 9, pp. 249–260, March 1998.

[26] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems On Concurrent Processors.* New Jersey: Prentice Hall, 1988.

[27] S. G. Aki, *The Design and Analysis of Parallel Algorithms.* New Jersey, USA: Prentice-Hall International, Inc., 1989.

[28] J. Modi, *Parallel Algorithms and Matrix Computation.* Oxford, London: Clarendon Press, 1988.

[29] S. Kim and J.C.Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," in *International Conference on Parallel Processing*, vol. 3, pp. 1–8, 1998.

[30] D. Fernandez-baca, "Allocating modules to processors in a distributed system," *IEEE Transaction on Software Engineer*, vol. 15, pp. 1427–1436, November 1989.

[31] B. Shirazi and M. Wang, "Analysis and evaluation of heuristic methods for static task scheduling," *Parallel and Distributed Computing*, vol. 10, pp. 222–223, 1990.

[32] T. Casavant and J. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineer*, vol. 14, pp. 141–154, February 1988.

[33] W. Huang, X. Chen, L. Bhuyan, and F. Lombardi, "Accurate communication models for task scheduling in multicomputers," in *Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP '95)*, vol. 1, 1995.

[34] A. Hurson, B. Lee, B. Shirazi, and M. Wang, "A program allocation scheme for data flow computers," in *International Conference on Parallel Processing*, vol. 1, pp. I–415–I–423, 1990.

[35] S. Darbha and D. Agrawal, "Optimal scheduling algorithm for distributed-memory machines," *IEEE Transaction on Parallel and Distributed System*, vol. 9, pp. 87–95, January 1998.

[36] B. Shirazi, H. Chen, K. Kavi, J. Marquis, and A. Hurson, "Parsa: A parallel program software development tool," in *Proceeding 1994 Symposium on Assessment of Quality Software Development Tools*, pp. 96–111, IEEE CS Press, 1994.

[37] G. Sih and E. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 175–187, February 1993.

[38] M. Palis, J. Liou, and D. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Transactions on Parallel and Distributed System*, vol. 7, pp. 46–55, January 1996.

[39] R. Paige, J. Reif, and R. Wachter, *parallel Algorithm Derivation and Program Transformation.* Boston: Kluwer Academic Publishers, 1993.

[40] B. kruatrachue and T.lewis, "Grain size determination for parallel processing," *IEEE Software*, vol. 5, pp. 23–32, January 1988.

[41] E. Mohr, D. Kranz, and J. R.H. Halstead, "Laze task creation: A technique for increasing the granularity of parallel programs," *IEEE Transaction on Parallel and Distributed Systems*, vol. 3, pp. 264–280, July 1991.

[42] B. Indurkhya, H. Stone, and L. Cheng, "Optimal partitioning of randomly generated distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. SE-12, pp. 483–495, March 1986.

[43] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Transaction on Software Engineer*, vol. SE-3, pp. 85–93, January 1977.

[44] H. Stone, *Highh-Performance Computer Architecture.* Massachusetts: Addison-Wesley Publishing, 1987.

[45] S. Chingchit and M. Kumar, "Efficient clustering in message passing systems," in *International Conference on Optimization Techniques and Applications (ICOTA'98)*, vol. 1, pp. 277–283, 1998.

[46] J. Dongarra and D. Walker, "Software libraries for linear algebra computations on high performance computers," *SIAM Review*, vol. 37, pp. 151–180, June 1995.

[47] J. H. et al, "Scheduling precedence graphs in systems with interprocessor communication times," *Siam Journal of Computer*, vol. 18, pp. 244–257, April 1989.

[48] M. Al-Mouhamed, "Lower bound on the number of processors and time for scheduling precedence graphs with communication costs," *IEEE Transactions on Software Engineer*, vol. 16, pp. 1390–1401, December 1990.

[49] H. Anton, *Elementart Linear Algebra.* New York: John Wiley and sons, 1987.

[50] B. Noble and J. Daniel, *Applied Linear Algebra.* New Jersey: Prentice Hall, 1988.

[51] J. Dongarra, I. Duff, D. Sorensen, and H. V. der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers.* Philadelphia: SIAM, 1991.

[52] A. Grove, *An Introduction to the Laplace Transform and the Z Transform.* New York: Prentice Hall, 1991.

[53] Center of the New Engineer, George Mason University, USA, *URL:http://cne.gmu.edu/modules/VM/blue/mva.html*, 1999.

[54] P. Denning, "Queueing in networks of computers," *American Scientist 79*, vol. 3, pp. 206–209, May-June 1991.

[55] R. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, p. 345, 1962.

[56] A. Aho and J. Ullman, *Foundations of Computer Science.* New York: Computer Science Press, 1995.

[57] D. Tauber, *The Complete Linux Kit.* Alameda CA: Sybex, 1995.

[58] P. Volkerding, K.Reichard, and E.Foster-Johnson, *Linux: configuration and installation.* New York: MIS press, 1997.

[59] Linux organization, *URL:http://www.linux.org*, 1999.

[60] A. Silberschatz and P. Galvin, *Operating System Concepts.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1998.

[61] University of Notre Dame, USA, *LAM Installation Guide*, 1998.

[62] University of Notre Dame, USA, *LAM Notes/FAQ*, 1998.

[63] University of Tennessee, Knoxville, Tennessee, *MPI: A Messages-Passing Interface Standard*, 1995.

[64] Ohio Supercomputer Center, Ohio, USA, *MPI Primer/Developing with LAM*, 1996.