

TASK ALLOCATION IN PARALLEL AND DISTRIBUTED COMPUTER SYSTEMS

By

JEAN G. MINA

A Thesis

**Submitted in Partial Fulfillment of
The Requirements for the Degree of
Master of Science in Computer Science**

Department of Computer Science

Faculty of Natural and Applied Sciences

Notre Dame University - Louaize

Zouk Mosbeh, Lebanon

July 1998

TASK ALLOCATION IN PARALLEL AND DISTRIBUTED COMPUTER SYSTEMS

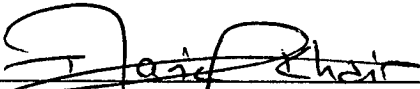
By

Jean G. Mina

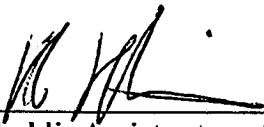
Approved:



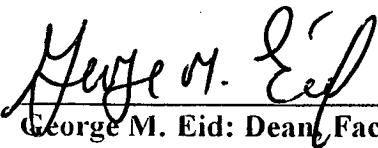
Fouad Chedid: Associate professor of Computer Science and Chairperson.
Advisor.



Marie Khair: Assistant Professor of Computer Science.
Member of Committee.



Khaldoun Khaldi: Assistant professor of Computer Science.
Member of Committee.



George M. Eid: Dean, Faculty of Natural and Applied Sciences.
Member of Committee.

Date of thesis defense: July 3, 1998

Copyright ©1998
by
Jean G. Mina

ACKNOWLEDGMENT

I would like to thank the committee members for reading the draft of the thesis and contributing their helpful comments.

Sincere gratitude goes to my advisor, Dr. Fouad Chedid, whose constant encouragement and motivation made this work possible. The fruitful discussions we had greatly increased my understanding of the topic, helping me to reach these results.

From a very early age, my parents taught me to appreciate the value and the importance of education, and that learning is a continuous process. For that, and for their love and support, I am very grateful.

ABSTRACT

The allocation problem is known to be NP-Hard in the most general case where both the number of modules and the number of processors is arbitrary, and it is also NP-Complete in some of the restricted cases. Also, the general scheduling problem, where no restrictions are imposed on the interconnection structure between modules, on the modules processing times, and on the number of parallel processors, is NP-Hard in the strong sense. Even under some restrictions, the scheduling problem is NP-Hard. In some other restricted cases, it is known to be NP-Complete. In this thesis, we first review fifty algorithms on both the allocation and the scheduling problem. Next, we suggest a reduced layered graph and a variation of Bokhari's [4,6,7] solution to the mapping chains of m tasks onto chains of n heterogeneous processors to achieve better space complexity. We also suggest two heuristic solutions for the same problem in both cases where processors are homogenous or heterogeneous in $O(m)$ and $O(nm)$ running time, respectively. We also adapt Lee and Shin [24] approach, to optimal task assignment in homogenous systems having an n -dimensional array or tree interconnection structure in the presence of attached tasks, on systems having a star graph interconnection structure. We generate the neighborhood tree and solve the problem with $O(Nm^3)$ running time where m is the number of tasks to be assigned, and N is the total number of nodes (processors) in the star graph. We mention that the suggested solution problem can be applied in the case of group graphs on any other type of graphs that can be generated by a permutation of a set of symbols. Finally, we generalize our result for systems having an arbitrary interconnection structure with a run time complexity of $O(Nm^3)$.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES.....	viii
CHAPTER	
I. DEFINING THE PROBLEM	1
II. DEFINITIONS AND NOTATIONS	5
III. THE ASSIGNMENT PROBLEM.....	14
IV. THE SCHEDULING PROBLEM	48
V. A NOTE ON THE ASSIGNMENT PROBLEM OF ARBITRARY PROCESS SYSTEMS TO HETEROGENEOUS DISTRIBUTED COMPUTER SYSTEMS	60
VI. A VARIATION OF BOKHARI'S LAYERED GRAPH ALGORITHM FOR MAPPING CHAINS ONTO CHAINS IN $O(m^2n)$ TIME USING A REDUCED LAYERED GRAPH OF $O(mn)$ NODES	65
VII. A HEURISTIC ALGORITHM FOR MAPPING CHAINS ONTO CHAINS OF A HOMOGENOUS AND A HETEROGENEOUS PROCESSOR SYSTEM IN TIME $O(m)$ AND $O(mn)$ RESPECTIVELY	73
VIII. OPTIMAL TASK ASSIGNMENT IN HOMOGENOUS SYSTEMS IN THE PRESENCE OF ATTACHED TASKS.....	78
IX. A COMPARISON OF ASSIGNMENT AND SCHEDULING ALGORITHMS	91
X. CONCLUSION AND FUTURE RESEARCH	96
REFERENCES	98

LIST OF TABLES

Table 6.1	Execution cost per module on each processor	71
Table 6.2	Summary of results	72
Table 9.1	A comparison of assignment and scheduling algorithms.....	92

LIST OF FIGURES

Fig. 5.1	The clustering algorithm	61
Fig. 5.2	The graph of the worst case example.....	62
Fig. 5.3	Pass 1	63
Fig. 5.4	Pass 2	63
Fig. 5.5	Pass 3	63
Fig. 5.6	Pass 4	63
Fig. 5.7	The cluster tree of the worst case example	64
Fig. 5.8	The cluster tree of the worst case example as shown in Fig. 9 of [8].....	64
Fig. 6.1	A nine-module chain mapped onto a four-processor chain	65
Fig. 6.2	Bokhari's layered graph for the problem of Fig 6.1.....	67
Fig. 6.3	Improved layered graph of the problem of Fig 6.1	69
Fig. 6.4	The reduced layered graph for the problem of Fig 6.1.....	70
Fig. 6.5	The reduced layered graph of the problem in Table 6.1	71
Fig. 7.1	A chain of 8 modules onto a chain of 3 homogenous processors.....	74
Fig. 7.2	A chain of 8 modules onto a chain of 3 heterogeneous processors	76
Fig. 8.1	An example group graphs [1]	79
Fig. 8.2	Star graphs S_3 and S_4	80
Fig. 8.3	The neighborhood tree for S_4	81
Fig. 8.4	Illustrative figures for Lemma 4	86
Fig. 8.5	An example of TIG of 7 modules to be assigned to S_3	87
Fig. 8.6	Iteration 1 of the algorithm on the problem of Fig 8.5.....	88
Fig. 8.7	Iteration 2 of the algorithm on the problem of Fig 8.5.....	88
Fig. 8.8	Iteration 3 of the algorithm on the problem of Fig 8.5.....	88
Fig. 8.9	Iteration 4 of the algorithm on the problem of Fig 8.5.....	89

Fig. 8.10	Iteration 5 of the algorithm on the problem of Fig 8.5.....	89
Fig. 8.11	An optimal task assignment of the problem of Fig 8.5.....	89

CHAPTER I

DEFINING THE PROBLEM

Early research in distributed computing focused on the idea of distributing a computational load, having a well defined interconnection structure, over more than one processor communicating through a well defined interconnection structure. The goal behind this distribution seeks to achieve results such as: minimization of execution cost associated with the computational load and interprocess communication costs, good load balancing, and high degree of parallelism. This latter contributes to the reduction of the overall processing time of the load by efficient utilization of the available resources [25]. For this reason, different algorithms are derived to provide for optimal or near-optimal distributions of the load on the processors. In this thesis, we consider the case where distribution is static, i.e. it remains unchanged until all the requirements of the computational load has been met. Such algorithms require prior knowledge of precise data on the behavior of the computational load and on its interconnection structure as well as the attributes and the interconnection structure of the multiprocessor system. These information will be used by the algorithm at compile time aiming to find a proper distribution. Depending on the nature of the problem and its computational load, a static distribution algorithm may lead to one of two solutions: (i) mapping or matching modules to processors, and (ii) scheduling modules to processors.

Mapping or matching tasks to processors, also known as the assignment problem, deals with the optimal assignment of a serial program or a parallel program to run on more than one processor in order to optimize the running cost [4]. An example of a serial program is a program that have a procedure which deals with floating point computation, and another procedure which have symbol manipulation. In this case, the first procedure can be sent to a powerful floating point processor, while the latter to a processor that handles symbol manipulation. This solution is perfect if the overhead due to interprocessor communication,

caused by transfer of control and parameters between procedures not residing on the same processor, is zero, which is never the case.

In the case of a parallel program, two or more modules may execute concurrently for different periods of time during the lifetime of the program. The purpose behind creating a parallel program is to reduce the total execution time by distributing parts of the program that can run in parallel over different processors. In this case also, the overhead due to interprocessor communication between modules assigned to different processors can be a crucial factor.

Scheduling modules to processors, or simply task scheduling, is the process of mapping each module of a parallel program to a processor and also to a starting time [14]. The scheduling goal is to minimize the total completion time of the parallel program by providing for the shortest possible schedule on the specified interconnected set of processors [14]. The resulting schedule specifies which module is assigned to which processor and the order of execution of each module with respects to the other modules.

On the other hand, two types of distributed computing environments exist. The first is a network of more or less autonomous machines. These machines may be either homogenous or heterogeneous, and may have different computing capabilities. Also, communication links between these machines can be either homogenous or heterogeneous [4,6,7]. Such networks are known as distributed processor systems [4]. The second is an interconnected collection of special-purpose machines, known as parallel processor systems [4]. Moreover, processing is considered parallel if interprocessor communications are very fast; otherwise, it is considered distributed. This distinction may vanish if we consider the use of optical media, and parallel processors with slow communication speed.

In general, a module can be a collection of procedures or subroutines, or one or more data files. Links between different modules will insure transfer of control, and data access

between different modules. A module may also be a segment of a computational domain. In this case, all processors execute the same program, but on different portions of the large computational domain [3,4,6,7,27]. This is also a partitioning problem where each processor is assigned a partition of the data to work on.

The aim of researchers in this area of distributed computing is to find efficient algorithms that provides for optimal or near-optimal distribution of tasks onto processors in the case of the assignment problem and that of the scheduling problem. However, efficient algorithms are unlikely to be found in the general case and even in some of the relaxed or simplified cases. The allocation problem is known to be NP-Hard in the most general case where both the number of modules to be distributed and the number of processors is arbitrary, and it is also NP-Complete in some of the restricted cases [4,6,7,10,25,27]. Also, the general scheduling problem, where no restrictions are imposed on the interconnection structure between modules, on the modules processing times, and on the number of parallel processors, is NP-Hard in the strong sense. Even under some restrictions, the scheduling problem is NP-Hard. In some other restricted cases, it is known to be NP-Complete [14,16,17,22].

Thus, it is important to identify the tractable cases in both problems and try to provide efficient solutions for them. Also, for all intractable cases, it is reasonable to search for efficient algorithms that finds near-optimal solutions and to study the behavior of such algorithms under different inputs.

This research focuses on different aspects of both the assignment and the scheduling problems where the allocation of a load is assumed to be static and duplication of the load on other processors is not allowed.

This thesis is composed of ten chapters. Chapter II includes all the definitions and notations. In chapter III, we review different aspects of the assignment problem. In chapter IV, we review different aspects of the scheduling problem. Chapter V includes a note on the

assignment problem of arbitrary process systems to heterogeneous distributed computer systems. Chapter VI describes a variation of Bokhari's layered graph algorithm for mapping chains onto chains. Chapter VII suggests a heuristic algorithm for mapping chains onto chains of a homogenous and a heterogeneous processor system. In chapter VIII, we solve the problem of task assignment in homogenous systems in the presence of attached tasks. Chapter IX includes a comparison of assignment and scheduling algorithms which we reviewed in chapters II and III. The conclusion and topics for future research make up chapter X.

CHAPTER II

DEFINITIONS AND NOTATIONS

Definition 1: A serial program is a program composed of sequential modules. One module is active on one processor at one time, and modules can pass control to each other while the program executes.

Definition 2: In a parallel program, two or more modules may execute concurrently for various periods during the life time of the program.

Definition 3: A chain-structured program is made up of m modules numbered from $1 \cdots m$, and has an intercommunication pattern such that module i is connected to modules $i + 1$ and $i - 1$.

Definition 4: A directed graph or digraph $G = (V, E)$ consists of a set V of vertices (nodes), and a set E of ordered pairs of these nodes called directed edges, or simply edges. There is a direction associated with each edge. The edge $\langle x, y \rangle$ extends from the tail node x to the head node y .

Definition 5: The cardinality n of V is the number of vertices in V . The cardinality e of E is the number of edges in E . The indegree of a node is the number of edges entering that node. The outdegree of a node is the number of edges leaving that node. In a directed graph, $\max(\text{indegree}) = \max(\text{outdegree}) = n - 1$. The maximum number of edges is $n(n - 1)$.

Definition 5: A directed path from node s to node t in a directed graph $G = (V, E)$ is a sequence of edges $\langle s, p \rangle, \langle p, q \rangle, \dots, \langle v, w \rangle, \langle w, t \rangle$, such that the tail of the first edge is s , the head of the last is t , and for all except the last edge, the head of any edge coincides with the tail of the edge immediately after it.

Definition 6: A node must be visited at most once in a directed path. If s coincides with t , the path is called a cycle. A directed graph that does not have a cycle is called an acyclic graph. The length of the path between two nodes is the number of edges in that path.

Definition 7: An undirected graph is a graph $G = (V, E)$ in which the edges have no directions, and at most one edge connects two nodes. V is the set of vertices, while E is the set of unordered pairs of these vertices.

Definition 8: Two connected nodes of an undirected graph are called adjacent. The edge $\langle x, y \rangle$ is incident on the nodes x and y . The degree of a node is the number of incident edges, and $\max(\text{degree}) = n - 1$. The maximum number of edges is $n(n - 1)/2$.

Definition 9: A path from node s to node t in an undirected graph is a sequence of edges $\langle s, p \rangle, \langle p, q \rangle, \dots, \langle v, w \rangle, \langle w, t \rangle$ such that the first edge is incident on s , the last is incident on t , and every pair of successive edges is incident on a common node. Also, repeated nodes are not allowed. If s coincides with t , then the path forms a cycle. The length of a path is the number of edges in the path.

Definition 10: A graph $G_s = (V_s, E_s)$ is called a subgraph of a graph $G = (V, E)$ if V_s is a subset of V and E_s is a subset of E . This notion applies to both directed and undirected graphs.

Definition 11: An underlying graph results from ignoring directions and deleting duplicate edges connecting two nodes of a directed graph.

Definition 12: A graph is called connected if a path exists between every pair of its nodes, otherwise it is called disconnected.

Definition 13: A cutset or cut of a connected graph is a subset of the edges satisfying two conditions: (i) removal of these edges disconnect the graph, and (ii) no proper subset of these edges also satisfies (i). Property (ii) states that a cutset is a minimal subset of edges that must

be removed in order to disconnect the graph. If a graph G has two distinguished nodes s and t , and if a cutset breaks G into G_1 and G_2 such that s is in G_1 and t in G_2 , then the cutset is called an s - t cut. When the edges of an s - t cut are removed from the graph, nodes s and t are said to be disconnected from each other.

Definition 14: A weighted graph is one in which there is a real number associated (weight) associated with each edge. The length or weight of a path in a weighted graph or a weighted digraph is the sum of the weights of the edges in that path. The bottleneck weight is the weight of the heaviest edge in that path. The weight of a cut in a weighted graph is the sum of the weights on all edges in that cut.

Definition 15: The Ford-Fulkerson Maxflow-Mincut theorem states that the minimum cut, denoted by minimum weight s - t cut or simply mincut, is equal to the maximum flow in the network. The network can be a weighted directed or undirected graph. It can be viewed as network transferring some commodity from s to t . The flow through the network must obey the following restrictions:

1. The flow through an edge cannot exceed its capacity. An edge carrying a flow that is equal to its capacity is called saturated.
2. The flow entering a node must equal to the flow leaving a node, except for s and t .
3. Node s has no flow entering it, node t has no flow leaving it, and the flow leaving s must be equal to the flow entering t .
4. In the case of weighted digraphs, the flow in an edge must be in the direction of the edge.

Definition 16: Gomory and Hu showed that in order to obtain maximal flows between all the $n(n-1)/2$ pairs of nodes of a graph we should run the Maxflow-Mincut algorithm only $n-1$ times. The result of the $n-1$ running of the Maxflow-Mincut algorithm is used to build the

so-called Gomory-Hu tree, abbreviated as G-H tree. The G-H tree represents all maximal flows between any pair of nodes in the graph, and all minimal cut separating two nodes.

Definition 17: An n -way cut is a set of edges which partitions the nodes of the network into n disjoint subsets with exactly one processor node in each subset and naturally represent an assignment of tasks to processors. The cost of an n -way cut is the sum of the weights on the edges in the cut.

Definition 18: An undirected tree is a connected undirected graph which contains no cycle. For n nodes, we must always have $n-1$ edges. Nodes with degree 1 are called leaf nodes. A directed tree is a directed acyclic graph whose underlying graph is a tree.

Definition 19: An out-tree is a directed tree with all edges directed consistently outwards from a specially designated node called root node. The root has indegree zero and all other nodes have indegree exactly one. Nodes with outdegree equals to zero are called leaf nodes.

Definition 20: An in-tree is a directed tree with all edges directed consistently inwards to a specially designated node called root node. The root node has outdegree zero and all other nodes have outdegree exactly one. Nodes with indegree equals to zero are called leaf nodes.

Definition 21: A binary tree is an undirected tree where a designated root node has degree no more than two and all other nodes have degree no more than three. The height of the tree is the maximum distance between the root node and any leaf node.

Definition 22: Graphs with no constraints on the number of edges connecting two nodes are called multigraphs.

Definition 23: A series-parallel program is an undirected multigraph which has two distinguished nodes called the source s and the sink t , and which can be transformed into a graph with just these two nodes s and t connected by a single edge, by repeated applications of the following replacement rules.

1. If two nodes have two parallel edges between them, replace these edges by one edge.

2. If a node b with degree 2 is adjacent to two nodes a and c , replace b and the edges $(a,b), (b,c)$ with the single edge (a,c) .

Some graphs can be transformed into a series-parallel graphs by adding suitable dummy nodes and/or dummy edges.

Definition 24: A task graph is an in-forest if each task has at most one immediate successor.

Definition 25: A task graph is an out-forest if each task has at most one immediate predecessor.

Definition 26: An interval order is a task graph in which the nodes can be mapped into intervals on the real line and two nodes are related if their corresponding intervals do not overlap.

Definition 27: A node with no predecessors has a depth of zero. The depth of any other node is defined as the maximum number of edges between that node and any node with depth zero.

Definition 28: The level or bottom level of a node i is the longest path length from an exit node to node i (including i). In a tree, for each node there is only one such path.

Definition 29: The top level of a node i is the longest path length from an entry node to node i (excluding i). In a tree, for each node there is only one such path.

Definition 30: A ready task is a task with no predecessors or with all its predecessors already executed.

Definition 31: A free node is a node with no predecessors or with all its predecessors already scheduled.

Definition 32: Given an in-forest $G = (V, E)$, the set of siblings S_i is the set of all nodes in V having a common child, denoted by $child(S_i)$.

Definition 33: The elapsed time of a module i is defined as the sum of execution cost of that module and the communication costs with all its adjacent modules.

Definition 34: The overall elapsed time is defined as the maximum of the job completion time and the medium access time.

Definition 35: An n -dimensional homogenous array is composed of $N(= n_1 \times n_2 \times \dots \times n_n)$ functionally-identical processors $\{p_{k_1, k_2, \dots, k_n} \mid 1 \leq k_i \leq n_i, \text{ for all } i\}$ with a communication link between each pair of processors $p_{k_1, k_2, \dots, k_n}, p_{l_1, l_2, \dots, l_n}$ if and only if $|k_j - l_j| = 1$ for some j th coordinate and $k_i = l_i$ for the other coordinates $1 \leq i (\neq j) \leq n$. The distance between any two processors p_{k_1, k_2, \dots, k_n} and p_{l_1, l_2, \dots, l_n} becomes $\sum_i |k_i - l_i|$.

Definition 36: A cutset C_{ij} of the two-terminal network graph $G_{ij} = (V_{ij}, E_{ij})$ is a set of edges which when deleted, separate \hat{S}_{ij} from \hat{T}_{ij} such that $\hat{S}_{ij} \cap \hat{T}_{ij} = \emptyset$, $\hat{S}_{ij} \cup \hat{T}_{ij} = V_{ij}$, $S_{ij} \in \hat{S}_{ij}$, and $T_{ij} \in \hat{T}_{ij}$. \hat{S}_{ij} is called the source set and \hat{T}_{ij} the sink set of the cutset. The weight of a cutset is the total weight of the edges in the cutset.

Definition 37: For each i th coordinate, let C_i be a set of $(n_i - 1)$ cutsets C_{ij} each of which is on the corresponding two-terminal graph G_{ij} , i.e., $C_i = \{C_{ij} \mid 1 \leq j \leq n_i\}$. Then C_i is said to be admissible if no two cutsets in C_i cross each other. The weight of C_i , $W(C_i)$, is the total weight of the cutsets in C_i , i.e., $W(C_i) = \sum_j W(C_{ij})$.

Definition 38: Let C_A be the set of all cutsets C_{ij} , i.e., $C_A = \bigcup_i C_i = \{C_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq n_i\}$. Then, C_A is said to be admissible if each C_i is admissible. The

weight of C_A is the total weight of the cutsets in C_A , i.e.,

$$W(C_A) = \sum_i W(C_i) = \sum_i \sum_j W(C_{ij}).$$

Definition 39: Let C_T be a set of $(N - 1)$ cutsets C_i s each of which is on the corresponding two-terminal network graph G_i . Then C_T is said to be admissible if no two cutsets in C_T cross each other.

Definition 40: A component C of a graph G is any subgraph of G for which the following is true: there are no edges among nodes of C and nodes of $G - C$. That is, C is the union of one or more disjoint connected components of G .

Definition 41: The median of an out-forest G , $\mu(G)$, is the height of some n th highest tree of G plus one, where n is the number of available processors.

Definition 42: The high subgraph H_G of a given out-forest G is the subgraph of G that contains all the trees, with the height strictly greater than the median of G , i.e., $\mu(G)$.

Definition 43: The low subgraph L_G of G that contains all the trees of G that are of height less or equal to the median $\mu(G)$.

Definition 44: Suppose that G is a graph whose dependencies are delay dependencies. Then the corresponding delay free graph G^s is defined as the graph that results if we replace the delay dependencies among every node m and its children m_1, m_2, \dots , with two stages of delay free dependencies:

- 1) between m and some child m_j of m (node m_j will be referred to as the favored child), and
- 2) among m_j and the rest of the children of m .

Definition 45: Shortest delay free graph G^s of a given graph G is a delay free graph such that every subgraph of G^s has height less than or equal to the height of the corresponding (i.e., containing the same nodes) subgraph in any other delay graph for G .

Notation 1: d_{ij} is the total amount of data transmitted between module i and module j .

Notation 2: s_{pq} is the cost of transmitting one unit of data over the link connecting processor p to processor q . It is assumed that $s_{pq} = s_{qp}$.

Notation 3: $s_{pq}(d_{ij})$ is the function that gives the total amount of communication between module i running on processor p and module j running on processor q . In general, this function can be simplified to $s_{pq} * d_{ij}$. We note that s_{pp} is the cost of transmitting a unit of data between two modules expected to execute on the same processor. If $s_{pp} = 0$, then intraprocessor communication is neglected. In the case of a multiprocessor system with homogenous links, s_{pq} is the same for all (p, q) pairs of links. In this case, we denote by c_{ij} the total amount of data transmitted between module i and module j .

Definition 46: In this thesis, “interprocessor communication” is used to indicate the communication volume taking place between modules where only the communication between modules assigned to different processors are accounted for. While “intermodule communication” is used to indicate the communication volume taking place between modules assigned to the same or to different processors.

Definition 47: P is the class of all problems that can be solved deterministically in polynomial time.

Definition 48: NP is the class of all problems that can be solved non-deterministically in polynomial time (can be verified deterministically in polynomial time).

Definition 49: A problem X is NP-Complete if

1. $\forall Y \in \text{NP}$, Y is reducible to X , denoted $Y \alpha X$, if \forall instance $I \in Y$, \exists a polynomial time computation such that $f(I)$ is an instance of X , and I iff $f(I)$.
2. $X \in \text{NP}$.

3. If 1 is satisfied but not 2, X is NP-hard.

CHAPTER III

THE ASSIGNMENT PROBLEM

As described earlier, one aspect of distributing a computational load over more than one processor is known as the assignment problem. Different algorithms based on different techniques were used to solve this problem and to provide optimal solutions when possible. Some of these algorithms can be clearly classified under one technique or another, and some others share more than one technique in searching for appropriate solutions. In our classification of the solutions to the different aspects of the assignment problem, we will group together those algorithms that share the same techniques to find an optimal or near-optimal assignment. Sometimes, classifying an algorithm under a technique or another may be a debatable issue for algorithms that rely on a well-known technique to reach a starting solution that might be optimal and continue improving this solution heuristically if it is not. Such cases will be signaled out in the flow of the text. In general, these techniques are classified under five categories: network flow, integer programming, least cost, probe functions, and heuristic algorithms.

3.1. Network flow algorithms

The Basic Dual-Processor Assignment problem is the one of distributing the execution of a serial or a parallel program over a two-processor system in order to minimize the sum of execution costs of all modules and the sum of interprocessor communication costs. To solve the problem, Stone [4] uses a graph of m nodes to model the interconnection of the m different modules of the program. Each node in the graph represents a module and each edge connecting two nodes indicates that the corresponding modules communicate during the execution of the program. Each of these internal edges is labeled with the total time of communication between the two adjacent modules. From this graph, the assignment graph is

built by adding two additional nodes s and t representing the two processors, and $2m$ additional external edges which link s and t to all module nodes. An external edge connecting node i to node s is labeled with the cost of executing module i on the processor represented by node t . This reverse labeling applies also to the edges incident on t . Stone shows that a cut that disconnects s and t corresponds to an assignment of modules to processors and vice versa, and that the weight of a cut in the assignment graph is equal to the total cost of the corresponding module assignment. Therefore, the optimal solution of the problem depends on finding the minimum weight cut or mincut in the assignment graph. A Maxflow-Mincut [4] algorithm applied to the assignment graph with s as source and t sink will find the mincut in $O(m^3)$ time and the cost of the found assignment is equal to the weight of the cut.

A variation of the basic dual assignment problem is the problem of assignment with dynamic relocation. In the basic dual assignment problem, modules assigned to processors are expected to remain there while the characteristics of the computation inside each module are constants. By characteristics of computation Bokhari [4] means the ratios of the time the program spends in its different parts. Suppose that these characteristics change during the lifetime of the program. This change creates a new dimension, into the optimal assignment problem, which is relocation.

Relocation means that a module may be relocated between processors during program execution. Relocation data is collected from the examination of the program activity. The lifetime of the distributed program is divided into phases, and only one module executes during a specific phase. A module is allowed to move from one processor to another only between phases. The following information is associated with each phase :

1. The executing module during this phase.

2. Running cost of this module on each of the two processors.
3. Cost of residence of the remaining idle modules on each of the two processors.
4. Intermodule communication costs if modules are on different processors.
5. Relocation costs for each module.

This information is represented in a graph where the number of nodes is equal to the number of modules multiplied by the number of phases. Each node in the graph represents the residence of a module in a specific phase. The module that executes during a phase is marked with an asterisk. Vertical edges connect successive residence of the same module and are labeled with relocation costs of the module between the respective phases. Horizontal edges connect the executing modules to other modules of the same phase and represent intermodule communication costs between the executing module and the other modules during this phase.

To build the dynamic assignment graph, we add two nodes s and t , each representing a processor, and connect them to all the nodes in the graph. Again, reversed labeling is used. The edge from node A_1 to node s is labeled with the cost of executing module A on t during phase 1. The residence cost of module C on s during phase 2 goes on the edge joining C_2 to t . The mincut obtained by a Maxflow-Mincut algorithm between s and t is the optimal dynamic assignment of the program. The dynamic assignment graph could be reduced, in terms of edges, to the zero residence graph if the goal is to minimize execution time while ignoring the cost of residence of an idle module on a processor. The zero residence graph is obtained by omitting edges labeled with the cost of residence without execution. Bokhari [4] shows that in the case of zero residence graphs, the minimum weight assignment also corresponds to the mincut. Thus, the dual processor assignment with dynamic relocation can be solved in $O(m^3)$ time.

Network flow techniques are also used to solve the problem of assigning modules of a distributed program of size m to an n -dimensional linear array of $N(= n_1 \times n_2 \times \dots \times n_n)$ homogenous processors connected using homogenous links with the constraint that one or more modules are each attached to a specific processor. The objective is to minimize the sum of execution and interprocessor communication time knowing that if two communicating modules are not mapped to directly communicating processors, then communication will take place indirectly through one or more processors which will add to the overall execution cost of the system. Since the processors of the n -dimensional system are homogenous, the problem reduces to minimizing the communication costs.

To solve the problem, Lee and Shin [24] created an $N(= n_1 \times n_2 \times \dots \times n_n)$ -terminal network graph by adding to the problem graph N terminal nodes each representing one processor. Then, for each i th coordinate or dimension n_i of the n -dimensional linear array network, they generate all the $(n_i - 1)$ sets P_{ij} of processors having their i th coordinate less than j , and their corresponding sets \bar{P}_{ij} containing all the remaining processors. For example, for a (2×3) two-dimensional array network, we can generate $P_{11} = \{p_{11}, p_{12}, p_{13}\}$, $P_{21} = \{p_{11}, p_{21}\}$, and $P_{22} = \{p_{11}, p_{12}, p_{21}, p_{22}\}$. To each of the $(n_i - 1)$ pairs of sets P_{ij} and \bar{P}_{ij} corresponds a two-terminal network graph G_{ij} generated from the N -terminal graph as follows: each set P_{ij} is combined with all modules assigned to one of its processors in a source node S_{ij} , and its corresponding set \bar{P}_{ij} is combined in the same manner in a sink node T_{ij} . Let C_{ij} be a cutset in the two-terminal network graph G_{ij} , then C_i is the set of all $(n_i - 1)$ cutsets C_{ij} of the same i th coordinate. The set C_i is said to be admissible if no two cutsets in C_i cross each other. Let C_A be the set of all sets C_i , i.e. the sets of all cutsets C_{ij} ,

then C_A is said to be admissible if each C_i is admissible. The effect of building C_A is to isolate each processor of the n -dimensional array network and try to assign to it modules while obeying the minimization objective.

Lee and Shin showed that the optimal solution to the problem corresponds to the minimum-weight admissible set C_{A_0} of the n -terminal network, which means that all cutsets C_{ij} must be minimum-weight cutsets. The optimal solution can be obtained by the following procedure.

For every i th coordinate or dimension of the n -dimensional array network

1. Build a two-terminal network graph G_{ij} , $1 \leq j \leq (n_i - 1)$ as described above, since there are $(n_i - 1)$ graphs for each dimension.

i) Find its corresponding minimum-weight cutset C_{ij}

ii) Combine with S_{ij} an unattached module to the side of S_{ij} and not yet combined, and set to j the i th coordinate of the processor where the unattached module is to be assigned.

2. Set to n_i the i th coordinate of the processors where the unattached and not yet assigned modules are to be assigned.

The number of G_{ij} graphs generated by the algorithm for each dimension of the linear array is

$\sum_i (n_i - 1)$ graph. Thus, the algorithm requires $\sum_i (n_i - 1)$ applications of the Maxflow-

Mincut algorithm which is $O(m^3)$ in the worst case, therefor the overall run time complexity

of the algorithm is $O\left(\sum_i (n_i - 1)m^3\right)$ time. A special case of the n -dimensional array network

is the hypercube, i.e. where $n_i = 2$ for all i . In this case, the solution to the problem for an

n -dimensional hypercube with $N = 2^n$ processors is in $O(Nm^3)$.

The same technique is used to solve the problem of assigning a distributed program to a tree of n homogenous processors connected with $n - 1$ homogenous links while minimizing communication costs. To solve the problem, Lee and Shin [24] number the processors in post-order. This way each processor p_k is assigned a higher number than its descendants. Let P_k be the set of processor node p_k and all its descendants, and \bar{P}_k the set of the other nodes. Then, they build the n -terminal network graph G_n by adding n nodes, one node per processor, to the problem graph. From the n -terminal network graph G_n , they construct $(n - 1)$ two-terminal network graphs G_i as follows: combine all the processors in the set P_i and all the modules attached to any of its processors in a source node S_i , and all the processors in its corresponding set \bar{P}_i and all the modules attached to any of its processors in a sink node T_i . To each two-terminal network graph G_i corresponds a cutset C_i . The set C_T of all the $(n - 1)$ cutsets C_i is admissible since no cutsets in C_T cross each other. The optimal solution can be obtained by applying the following procedure.

1. Build a two-terminal network graph G_i , $1 \leq i \leq (n - 1)$ as described above, there are $(n - 1)$ graphs.
 - i) Find its corresponding minimum-weight cutset C_i
 - ii) Combine with S_i an unassigned module to the side of S_i , and assigns it to processor p_i .
2. Assign all remaining unassigned module to processor p_n .

The algorithm requires $(n - 1)$ applications of the Maxflow-Mincut algorithm on each two-terminal network with $O(m)$ nodes each. Thus the overall runtime complexity of the algorithm is in $O(nm^3)$.

In all problems discussed above and solved using the Maxflow-Mincut algorithm, communication costs are restricted to be the communication occurring between modules assigned to different processors, and they are referred to in the text as interprocessor communication. The communication costs between modules assigned to the same processors known as intraprocessor communication costs are considered negligible. In the case where both interprocessor and intraprocessor communication costs are considered, we will refer to the global communication costs as intermodule communication costs.

The next problem is one of assigning a distributed program of m modules to a network of homogenous workstations or to a fully connected multiprocessor system. The network or the multiprocessor system is modeled by a virtual clique architecture having homogenous communication links. The objective is to minimize the maximum of the jobs completion time and intermodule communication costs.

The problem is modeled with a doubly weighted graph where each node represents a module and each edge represents communication between two modules. A node in the graph is labeled with the execution cost of its corresponding node, while an edge has two labels. The first label indicates the interprocessor communication costs between the modules represented by the head and tail nodes of the edge and the second label represents the intraprocessor communication costs between the same modules. To reduce the complexity of finding the optimal solution, Hui and Chanson [19] reformulated the labeling on the edges of the problem graph in order to create the preprocessed interaction graph (PTIG) where each edge has only one label. In the PTIG, each node is labeled by the sum of the execution cost of the corresponding module and the sum of the weights indicating interprocessor communication on all edges incident on that node. An edge of PTIG is labeled with twice the difference of the weight of interprocessor communication and the weight of intraprocessor communication. The main idea of the solution is to merge nodes optimally, such that the

resultant elapsed time cannot be reduced further. When merging two or more nodes into a new node, the weight of the new node is the sum of the execution cost of the modules represented by the merged nodes, added to it twice the sum of all intraprocessor communication costs between merged modules, and the sum of all interprocessor communication costs between any of the merged modules and any node connected to it. This node merging operation leads to merging all the edges connecting a node to any node merged into the same group in a single edge. Thus, the weight of the merged edge is the sum of all merged edges taken from PTIG. The proposed algorithm tries to minimize the elapsed time of each node i by computing its m -set, by merging node i with one or more nodes such that i 's elapsed time is minimum. By finding all m -sets, the minimum overall elapsed time which corresponds to the optimal solution can be computed. Hui and Chanson's [19] solution begin by modifying PTIG into the transformed PTIG (TPTIG). TPTIG have the same set of nodes of PTIG with addition of a terminal node t . To each undirected edge in PTIG corresponds two directed edges in TPTIG, and each is labeled with half the weight of the edge on PTIG. An edge connect each node to the terminal node t , and it is labeled with the difference between the weight of the node in PTIG and half the sum of the weights of all edges incident on this node. In other terms the weight of the edge is equal to the sum of the execution cost of the corresponding node and the intraprocessor communication with all its node directly connected to it in the problem graph.

The algorithm starts by sorting all nodes in PTIG in decreasing order of elapsed times into a heap. Then, it selects the node with the largest elapsed time, removes it from the heap, and create its corresponding TPTIG where the selected node is the source node. Next, the m -set of the selected node is computed by applying a Maxflow-Mincut algorithm on TPTIG between the selected node and the sink node t . Nodes of the m -set are merged into a single node, and TPTIG and the heap are update by removing the merged nodes and adding the new

one. These steps are repeated until the elapsed time cannot be reduced further. The algorithm requires at most m applications of a Maxflow-Mincut algorithm. Hui and Chanson suggests a

Maxflow-Mincut algorithm of $O\left(m e \log \frac{m^2}{e}\right)$ [18,19] runtime complexity for a graph with m

nodes and e edges. Since TPTIG is of $O(m)$ nodes and $O(m+e)$ edges, then the overall run

time complexity of the algorithm is in $O\left(m^2(m+e) \log \frac{m^2}{m+e}\right)$.

3.2. Least cost algorithms

The problem of optimally assigning the modules of a distributed program over a multiprocessor system is also attacked using techniques derived from Dijkstra's shortest path algorithm [11]. The application of these techniques depend on two steps: (i) building for each type of problems a specific assignment graph that represents all possible assignments of modules to processors and (ii) formulating the corresponding cost to be minimized. Then a variation of Dijkstra's algorithm, adapted to each specific assignment graph to insure better run time complexity, traverses the graph in order to find a least cost solution among all possible ones.

One of the problems solved using least cost algorithms is the problem of mapping a chain of m modules to a chain of n heterogeneous processors connected with heterogeneous links. The chain of modules may represent a pipelined or a parallel program where modules are connected in a chain-like fashion. The objective is to minimize the load on the heavily loaded processor, known as the bottleneck load. Bokhari's [4,6,7] solution uses a layered graph of n layers each representing a processor. A node $\langle i, j \rangle$, $1 \leq i \leq j \leq m$ in a layer represents the assignment of the subchain of modules i through j to the processor represented

by that layer. A node $\langle i, j \rangle$ is connected to all nodes $\langle j + 1, k \rangle$ in the layer directly below it for all j except for 1 and n . A source node s connects to all nodes in the first layer, and a sink node t connects to all nodes in the last layer. The number of nodes in Bokhari's layered graph is $O(m^2n)$ nodes, and the number of edges is $O(m^3n)$ edges. Bokhari applies a simple labeling procedure to the layered graph in order to find the minimum bottleneck path from s to t . Each node i in the layered graph is given a label $L(i)$. Initially, all nodes are given infinite labels except nodes of the first layer which are given zero label. The algorithm works as follows:

1. Examine each edge e emanating downwards from a layer connecting a node a (above) to a node b (below). Let the weight on this edge be $w(e)$.
2. Replace $L(b)$ by $\min(L(b), \max(w(e), L(a)))$.

Once t is labeled, the path representing the optimal path can be found by tracing backwards from t to s . Both the labeling procedure and finding the optimal path visit each edge of the layered graph exactly once, therefore the overall complexity of Bokhari's algorithm is $O(m^3n)$ time which is the number of edges of the layered graph.

Using an improved layered graph, Nicol and O'Hallaron [27] were able to solve the same problem in $O(m^2n)$ time using $O(m^2n)$ edges and $O(m^2n)$ nodes. To Bokhari's layered graph $n - 2$ new layers were added, one between each layer, except between layers 1 and 2. Each new layer consists of m nodes labeled from 1 to m . A node $\langle j, k \rangle$ in layer i (with respect to Bokhari's layered graph) directs a single edge to node k in the new layer between layers i and $i + 1$. This edge is labeled exactly as the edge leaving node $\langle j, k \rangle$ in Bokhari's solution. A node k in the new layer directs to all nodes $\langle k + 1, l \rangle$ in the layer $i + 1$. Each edge of this type has a zero weight. A path from s to t corresponds to a solution of

this assignment problem. Bokhari's original algorithm works on the improved layered graph and finds the optimal solution in $O(m^2n)$.

In [4], Bokhari solves the problem of assigning an out-tree structured parallel program to a fully connected system with n heterogeneous processors and heterogeneous links. The goal is to minimize the sum of execution costs of all modules and the sum of intermodule communication costs. The out-tree or invocation tree represents the way modules invoke each other through the lifetime of the program. The assignment graph for this problem is a weighted directed graph derived from the invocation tree by adding a source node s , and several terminal nodes t_1, t_2, \dots one for each leaf node of the invocation tree. In addition to the source and terminal nodes, there are $m \times n$ nodes each labeled with a pair of numbers (i, p) representing the assignment of module i to processor p . Each layer of the assignment graph corresponds to a node in the invocation tree. Nodes in the layers corresponding to nodes in the invocation tree having outdegree greater than one are called forknodes. Each layer of forknodes is called a forkset. Let e_{ip} be the cost of executing module i on processor p . Then the edges connecting the source node s to the nodes $(1,1), (1,2), \dots, (1,n)$, representing the assignment of the root node of the invocation tree to each of the n processors, are labeled with $e_{11}, e_{12}, \dots, e_{1n}$. The edges incident on the terminal nodes t_1, t_2, \dots have zero labels. The edge joining node (i, p) to node (j, q) has weight $e_{jq} + s_{pq}(d_{ij})$ or simply the sum of the execution cost of the tail node and the cost of communication between the head node and the tail node of the corresponding edge in the assignment graph. Dijkstra's [11] shortest path algorithm applied on the assignment graph finds an optimal solution to the problem in $O(m^2n^2)$ time. A feasible solution to the problem is an assignment tree that can be generated from the assignment graph by removing from each layer all nodes except one

representing the corresponding assignment. The assignment tree having the shortest path from s to t represents the optimal solution. Bokhari [4] provided a faster solution due to the layered structure of the assignment graph.

1. For each terminal node

i) Find the shortest path to the nearest forkset, and leave a pointer on the node of the forkset and the next node in the shortest path to the terminal node.

ii) When a forkset f is exposed, i.e. the shortest path to all reachable terminal node has been calculated, temporarily remove all its outgoing edges, create a pseudoterminal node t_f , and connect all nodes in f to it.

iii) Label each edge with the sum of all shortest paths to the temporarily removed terminal nodes. The above three steps remove all forksets from the assignment tree until we reach a graph with one terminal node.

2. Find the shortest path from this terminal node to the source node s . Since we reached node s , we can reconnect all disconnected edges and traverse the graph from s to all terminal nodes using pointers set in a). Nodes with pointers are those of the shortest assignment tree.

The runtime complexity of the shortest assignment tree algorithm is $O(mn^2)$ time.

Towsley [34] generalized Bokhari's [4] results on out-tree structured distributed programs to distributed programs having a series-parallel structure and containing branches and loops. In [4], Bokhari showed how an out-tree can be transformed into a series-parallel graph through the addition of dummy edges. Towsley's [34] solution to the problem suggests a set of series, parallel, or tree transformation to the allocation graph thus reducing it to a two node / one edge graph where the edge weight represents the minimum cost assignment. A parallel replacement consists of replacing two parallel edges, i.e. having the same head and tail node, by one edge, while a series replacement consists of replacing a three nodes

connected by two series edges, where the tail node of the first edge is the head node of the second edge, by two nodes connected with a single edge. The tree replacement is used to remove tree structures by merging leaf nodes of the same forknode. The assignment graph is derived from the problem graph. It is an undirected weighted graph of $m \times n$ nodes having one source node s with zero weight, and a terminal node for each terminal node of the problem graph. Each node $\langle i, j \rangle$ is labeled with the cost of assigning module i to processor j . To each node i in the problem graph corresponds a layer of nodes $\langle i, 1 \rangle, \langle i, 2 \rangle, \dots, \langle i, n \rangle$ in the assignment graph representing all possible assignments of module i to the n processors. An edge connects node $\langle i, n_1 \rangle$ to node $\langle j, n_2 \rangle$ if modules represented by nodes i and j communicate in the program. This edge is labeled with cost of communication between module i on processor n_1 and module j on processor n_2 . All edges connecting to terminal nodes have zero labels. Source node s serves as one entry node for the assignment graph. For each feasible assignment of the m modules to the n processors corresponds a subgraph of the assignment, and one of these subgraphs corresponds to the optimal assignment. Towsley's [34] algorithm performs each type of replacement until no more replacement of that type is possible. These replacement operations are accompanied with necessary computation that finds the weight of the shortest path in the subgraph to be removed and assigns its value to the replacing edges. Both series or parallel replacement operations are in $O(v^3)$ for a graph of v nodes, while tree replacement operations are in $O(v^2)$. Since the number of nodes in the assignment graph is $O(mn^3)$ nodes, finding the optimal assignment for a problem is in $O(mn^3)$.

3.3. Probe functions algorithms

Another technique used in searching for optimal solutions for assignment problems relies on a probe function that searches for an optimal solution subject to a constraint. Instead of searching for the least cost among all possible solutions, a limited number of repeated probes with varying value of the constraint is used to find the solution. Each of the problems considered in this section seeks to minimize either the maximum bottleneck weight or the maximum of a sum weight and a bottleneck weight, referred to by Bokhari [6,7] as SB-weight. The problems that seeks minimizing the bottleneck weight have the property that given any trial weight w , a probe function determines if there is a feasible, optimal or near-optimal, solution to the problem whose bottleneck is less or equal to w . For the problems that seeks minimizing the SB-weight, the probe function accepts the trial weight w , and finds a mapping that minimizes the maximum sum weight among all mappings whose bottleneck is no greater than w . An appropriate probe function must be designed to solve each specific problem. Different probing functions may be designed for the same problem to provide more accurate or faster solutions.

The problem of mapping a chain of m modules onto a chain of n homogenous processors connected with homogenous links was solved by Iqbal [20,27] using probing functions. The use of a probe function in this context implies calling the function with a possible bottleneck weight w chosen using a binary search from an ordered set of possible values. The job of the probe function is to test if an assignment of modules to processors can be achieved taking into consideration that the load on the heavily loaded processor does not exceed the bottleneck value w . The test must be repeated with different bottleneck values until an optimal or near-optimal solution is reached. In Iqbal's solution, the possible bottleneck values used as parameters for the probe functions are discrete points in the range $[W_A, W_T]$ separated by ϵ , where W_A is the average execution cost of all modules, W_T their

total weight, and ε is the distance of the solution for the optimal one. The probe function iteratively chooses a feasible subchain of modules to be assigned to the first available processor by examining all modules, therefore each invocation of this function is $O(mn)$ time. Due to the binary search over the ordered set of possible bottleneck weights, the probe function is repeated $O\left(\log\left(\frac{W_T}{\varepsilon}\right)\right)$ times. Thus the algorithm provides an approximate solution with ε distance from the optimal solution in $O\left(mn \log\left(\frac{W_T}{\varepsilon}\right)\right)$ time.

Nicol and O'Hallaron [27] imposed more restrictions on the same problem. They required that the execution cost w_i of a module i be lower-bounded by a constant W , and the communication cost c_{ij} between two modules i and j be upper-bounded by a constant C . They used an improved probe function which uses the property that when searching for a subchain with total load less than w , we do not need to search for the last module of the subchain among all modules. Instead, this module can be found using a binary search, in $O(\log m)$ time, over a monotonic interval of possible candidate modules. Thus, the runtime complexity of the probe function is reduced to $O(n \log m)$ time.

The set of all possible bottleneck values submitted to the probing function is chosen from a sorted dominance matrix. An element Ω_{ij} of the dominance matrix is equal to the sum of the execution costs of modules i through j , as if they are assigned to the same processor, added to it the communication costs with modules $i-1$ and $j+1$. A two-dimensional binary search over the sorted dominance matrix selects possible bottleneck values which are at most $4m$ values, i.e. in $O(m)$. Thus, the optimal assignment can be found by $O(m)$ calls to the probe function. The overall complexity of the algorithm is $O(mn \log m)$ time.

Iqbal and Bokhari [21] provided optimal solution to the same problem with no assumptions about the magnitude of the costs. They transformed the chain of modules into a monotonic chain by merging two adjacent nodes if the communication cost between them is greater than the weight of any of the two nodes added to it the communication cost with its other adjacent node. Next, the resulting monotonic chain is used to build the lattice that stores the set of possible bottleneck values. An element Ω_{ijk} of the lattice is the sum of the weights of all modules j through k when assigned to processor i . Using two-dimensional binary search over the lattice leaves us with $O(m)$ possible bottleneck values. The probe function used in [21] is $O(n \log m)$ time. Thus, the overall runtime complexity of the algorithm is similar to the previous one, but the difference is that it provides an exact solution with no constraints on the magnitudes of the weights.

The second problem is of partitioning multiple chains each of m modules over a host-satellite system having n heterogeneous satellites. This problem occurs when several satellite computers, connected to a large host with higher computational power, receive from a real time environment data streams that must be processed in a pipelined fashion. The program running on a satellite can be partitioned between the satellite and a more powerful host. For each module i of satellite j , e_{ij} is the execution time on the host. For each pair of modules i and $i + 1$ of satellite j , c_{ij} is the communication cost if i is assigned to the host and j to the satellite. Since all processing is done in a pipelined fashion, the times for execution and communication are the time to process one frame of data. The problem is to minimize the time determined by the greater of : 1) the individual load on the most heavily loaded satellite, i.e. the bottleneck satellite, and 2) the sum of the loads assigned to the host. To solve this problem, Bokhari [4,6,7] uses a doubly weighted layered graph with n layers, one for each satellite, and m nodes per layer, one for each module. An edge connects each node in layer k

to each node in layer $k + 1$. The start (terminating) node $s (t)$ connects to all nodes of the first (last) layer. Each edge leaving node j in layer k is given a σ weight equal to the cost of executing on the host modules 1 through j of chain k , and a β weight equal to the cost of executing on the satellite k modules $j + 1$ through m of chain k . To both weights is added the communication cost between j and $j + 1$ over the link that connects satellite k to the host. Edges leaving node s have zero weights. The optimal SB path corresponds to the optimal assignment. To find the optimal SB-path, Bokhari applies the optimal sum-bottleneck algorithm [4,6,7] between s and t . In this algorithm, all the bottleneck values are sorted in ascending order. Then a modified binary search is used to select the bottleneck value w to be submitted to the probe function. The role of the probing function in this case is to find the shortest path from s to t with sum weight less or equal to w . The search over the list of bottleneck values continues until this path is found. The shortest path found is the optimal SB-path, and it corresponds to the optimal solution to the problem. In general, i.e. when the problem graph is arbitrary, the number of distinct values of w is no more than e the number of edges in the graph, and the optimal SB-path algorithm uses Dijkstra's algorithm to find the shortest path in $O(m^2)$ time for a graph of m nodes. Thus the complexity of the optimal SB Path algorithm is $O(m^2 \log e)$ time. In this specific assignment problem, the number of nodes is $O(mn)$, the number of edges is $O(m^2n)$ edges, and due to the layered structure of the graph we can find the shortest path using a simple labeling procedure in $O(m^2n)$ time which is the number of edges. Thus, the overall runtime complexity for finding the optimal solution when $m > n$ is $O(m^2n \log m)$ time.

Nicol and O'Hallaron [27] use their improved layered graph on this problem which reduces the number of edges to $O(mn)$ edges. This leads to an overall reduction in the

runtime complexity when finding the optimal solution using Bokhari's [4,6,7] SB-path algorithm to $O(mn \log m)$ time.

Iqbal [20,27] provided an approximate solution for this problem in the case of a host-satellite system with n homogenous satellites. He uses the probe function derived for the chain onto chain problem and applies it to each satellite chain. For each satellite, the probe function selects a total load on the satellites less than the bottleneck value w , such that the work assigned to the satellite is minimum. Then, if the total work assigned to the host is less than w , then the solution is optimal. Otherwise, the probe function must be called with a new bottleneck value w . As in Iqbal's chain to chain solution, the bottleneck value w is the result of a binary search over $[W_A, W_T]$ with the difference that W_T is the smaller of the total processing time if all modules are assigned to the host and the total processing time if no module is assigned to the host. For each value w , the probe function examines each module at least once for each satellite before a possible assignment is formulated, therefore the probe function is $O(mn)$. Iqbal's approximate solution is by ϵ far from the optimal solution with an overall runtime complexity of $O\left(mn \log\left(\frac{W_T}{\epsilon}\right)\right)$ time.

Nicol and O'Hallaron [27], by introducing weight restrictions to the host-satellite problem of homogeneous satellite similar to those with chain onto chain problem provided a faster optimal solution than all previously discussed solutions. They modified the improved probe function to deal with the difference in speed between the host and the satellites without affecting its complexity which is $O(n \log m)$ time. The values of all possible bottlenecks are sorted for each satellite and merged in $O(mn \log m)$ time. These values are selected by a binary search in $O(\log m)$ time leading to a total probing time in $O(n \log^2 m)$. For $m > n$,

the overall complexity to find the solution is bounded by the time of sorting and merging possible bottleneck value which is $O(mn \log m)$ time.

The third problem is that of partitioning a chain of m modules over a shared memory or bus interconnected system of n heterogeneous processors. The objective is to minimize the maximum of (i) the largest execution time on any processor, and (ii) the total interprocessor communication cost of the system. Bokhari's [7] solution uses the same layered graph used in the second problem. The communication costs are represented by σ weights, and the execution costs by β weights. Application of the optimal SB-path algorithm to the layered graph results in the optimal assignment in $O(m^2 n \log m)$.

In this problem also Nicol and O'Hallaron [27] used their improved layered graph in order to reduce the number of edges of the graph representing the problem. Due to this reduction, the application of Bokhari's [6,7] optimal SB-path algorithm on this graph solves the problem in $O(mn \log m)$ time.

Iqbal [20] provided approximate solution to the same problem where the processors are homogenous. The job of the probe function is to find if a partition of the chain-like program exists in which the load on any processor is less or equal to possible bottleneck value w . Then, the function must check if the sum of the communication costs resulting from the corresponding partition is also less or equal to w . In this case, the partition corresponds to a feasible assignment, and it can be obtained in $O(m^2)$ time. Bottleneck values w are of ϵ distance from each other, and they are chosen using a binary search over $[W_A, W_T]$. The overall run time complexity is $O\left(m^2 \log\left(\frac{W_T}{\epsilon}\right)\right)$ time [27].

In another solution to this problem, Nicol and O'Hallaron [27] imposed the usual restrictions on the magnitudes of the execution and communication costs. To solve the problem they solve the dynamic programming equations :

$$V(0, w) = 0$$

$$V(j, w) = c_j + \min_{L(j, w) \leq i \leq j} \{V(i-1, w)\}, \text{ for } j = 1, 2, \dots, m$$

where $V(j, w)$ is the minimal cost of partitioning module 1 through j , including the communication cost of separating module j from $j+1$, under the constraint that no subchain has execution weight greater than w . Therefore, $V(m, w) = S(w)$ is the minimal sum weight among all mappings which assign no more than w load on a processor. In this equation, $L(j, w)$ is the least index $i \leq j$ such that $S_{ij} \leq w$. For fixed w , $L(j, w)$ is a monotone non-decreasing function of j , and can be computed in $O(m)$ steps for all $j = 1, \dots, m$. By exploiting these facts, this equation can be computed in $O(m \log m)$ time. The set of possible bottleneck value w is chosen from all S_{ij} values with $i \leq j$ using the same technique used in Nicol and O'Hallaron's solution for the chain problem mentioned earlier. Also, they made the search procedure remembers the bottleneck and the cost of the least cost partition it ever computes where the smallest is the optimal bottleneck. All in all $O(m)$ probe calls are made each with $O(m \log m)$ time. Thus, the overall time complexity of finding the optimal solution is $O(m^2 \log m)$ time.

The fourth problem is that of partitioning n arbitrary distributed program of m modules on a host-satellite system with n heterogeneous satellites. Bokhari's [4,6,7] solution benefits from Stone's nesting theorem which states that as the load increases on the host, modules move away from the host to the satellites and never in the reverse order. This means that successive optimal assignments are nested inside each other. The load values that causes

the load transfer are called critical load values and can be found in no more than m applications of the network flow algorithms developed by Eisner and Severence [13]. Using the nesting property, each of the arbitrary programs can be viewed as having a chain like structure by grouping all modules lying between two adjacent cuts, representing two critical load factors, in one single node. By transferring all the n arbitrary programs into n chains, the assignment graph is a layered graph similar to the one used by Bokhari in his solution to the chains assignment on a host-satellite system. A node i at layer j of the assignment graph may represent one or more modules of the distributed program. The graph is doubly weighted and an edge from a node i at layer k to a node j at layer $k + 1$ has σ weight equal to the cost of assigning modules represented by the chain 1 through i to the host added to it the communication cost with the modules of the chain not assigned to the host, and a β weight equal to the σ weight added to it the cost of assigning nodes $i + 1$ through the last node of the chain to the satellite. The optimal assignment can be found by applying the optimal SB path algorithm on the layered graph in $O(m^2 n \log m)$ time. But the overall run time complexity of the algorithm is dominated by transforming the n arbitrary programs into chains which is $O(m^4 n)$.

Iqbal [20] was able to search, the layered graph representing the n transformed chains, for a near optimal solution with ϵ distance from the optimal solution in the case where the satellites are homogenous in $O\left(mn \log\left(\frac{W_T}{\epsilon}\right)\right)$ time using the same technique used in his solution to the chain onto host-satellite problem. However, the overall complexity is still $O(m^4 n)$.

The fifth problem is that of partitioning a tree structured program of m modules, representing parallel or pipelined computation, over a single-host, multiple-satellite system of

n homogenous satellites under three constraints: (i) the root is always on the host, (ii) once a node is assigned to a satellite all its children are assigned to the same satellite, and (iii) if two nodes are assigned to a satellite, their lowest common ancestor is also assigned to the satellite. It is also assumed that there are as many satellites as the number of leaf nodes of the program tree although the optimal solution may choose not to use all of them. In [4,6,7], Bokhari creates the assignment graph by adding a dummy Δ node below the program tree and connecting all the leaf nodes to it which divides the resulting graph into regions. Assignment graph nodes are inserted in each region with left to right ordering (A,B,...). Connecting with a directed edge all pairs of assignment nodes belonging to regions having a common edge creates a directed dual graph where the directions of the edges are from lower ordered nodes to the higher ordered ones. Each edge of the dual graph separates a subtree from the program tree, and it has two labels. A β label is the cost of running the subtree on the satellite added to it the cost of communication between modules on the host and modules on the satellite. The σ label weight is formulated such that the sum of these weights on a directed path from A to the last label represents the running cost of all modules assigned to the host and the communication cost with the modules assigned to the satellites. Each path from A to the last used label corresponds to an assignment where the SB weight of the path is the time required for the corresponding assignment. The optimal solution can be obtained using the Optimal SB Path Algorithm. For a program tree of m nodes and f leaf nodes, the assignment graph is a multigraph with $f + 1$ nodes and m edges. By adding dummy nodes and edges, the multigraph can be transformed into a conventional graph with no more than $2m$ nodes and m edges. Thus, the solution can be obtained in $O(m^2 \log m)$ time.

For the same problem, Iqbal [20] provided an approximate solution with ϵ distance from the optimal solution using a probe function. The function checks if it is possible to

partition the tree structured program over the host-satellite system such that the load on any satellite and the load on the host is less or equal to a bottleneck value w . Initially, all m nodes are assigned to the host with total load equal W_T . Then, the probe function, from bottom to top, assigns a subtree to a satellite if the computational load of the subtree and the communication costs between its root node and its corresponding parent node is less than or equal to w ; otherwise it merges the root of the subtree with its parent node by removing the edge in between. Possible bottleneck values w are selected using a binary search over $\left[\frac{W_T}{m}, W_T \right]$. The probe function is $O(m)$ time since it examines each node only once to decide whether to assign the node and its children to a satellite or to merge it with its parent.

The overall complexity for finding the approximate assignment is $O\left(m \log\left(\frac{W_T}{\epsilon}\right)\right)$ time.

The same problem was solved also by Iqbal and Bokhari [21]. The algorithm first creates the condensed tree from the problem tree by merging a child node with its parent node such that the tree is monotonic. A monotonic condensed tree ensures that the load caused by the subtree cannot exceed the load caused by a containing subtree. A probing function traverses the condensed tree upwards from the leave nodes and stops each time it identifies a maximal subtree that has weight less than a possible bottleneck value w . When all subtrees are calculated, the load on the host can be calculated; if it is less than w then the assignment is feasible. The probe function looks at each node only once, thus it is $O(m)$ time. The possible bottleneck values w are selected using a binary search from m weights of the subtrees that can be evaluated in $O(m)$ time and sorted in $O(m \log m)$ time. The choice of w takes $O(\log m)$. Thus the overall run time complexity of the algorithm is $O(m \log m)$.

3.4. Integer programming algorithms

The integer programming method is based on implicit enumeration of all the possible cases subject to additional constraints. For example, for a program of m tasks to be assigned to a system of n processors, each task can be assigned to any of the n available processors. This method implies the use of branch-and-bound algorithms [23] which searches for an optimal solution in a set of feasible solutions which is a subset of all possible solutions. Branch-and-bound algorithms are expected to be more efficient than complete enumeration algorithms, which are exponential in the number of inputs, due to the search restrictions. The search occurs in a carefully chosen subset of feasible solutions such that it contains the optimal solution. Kohler and Steiglitz [23] characterized Branch-and-bound algorithms in terms of sextuple (B_p, S, E, D, L, U) , where B_p is the branching rule, S is the selection rule, E is set of elimination rules, D is the node dominance function, L is the node lower-bound cost function, and U is an upper-bound solution cost. They also designed a general algorithm based on their characterization and investigated the computational requirements with respect to the choice of the parameters $S, E, D, L,$ and U .

In [26], a task allocation model for distributed computing systems is defined using a variation of Kohler and Steiglitz model [23]. Solutions enumeration is represented by a tree where each node represents a task and each edge or branch represents a processor. A path in the tree from the root node to a leaf node consists of one possible assignment, and the number of all possible unconstrained solutions is n^m solutions, which means that the runtime complexity of an algorithm that searches for an optimal solution among all enumerated ones is $O(n^m)$ time. Various constraints can be added to reduce the enumeration process to generating only those solutions that satisfy the application requirements. For example, in [26] a task preference matrix indicates that certain tasks can only be executed on a specified processor, and the task exclusive matrix defines tasks that cannot be assigned to the same

processor. Also, elimination rules are included to insure that a path which may not lead to an improvement in the solution in hand will not be generated. All these rules impose a reduction on the number of enumerated cases which leads to an optimal solution in less time.

3.5. Heuristic algorithms

Rao et al. [29] provided feasible solutions for another variation of the dual assignment problem where one of the processors has limited memory. A minimum weight feasible solution is the one that minimizes the sum of execution of all modules and the sum of interprocessor communication and do not exceed the memory requirements of the processor with limited memory. Their work provided two approaches to solve the problem. In both approaches, they started by using the same graph representation used by Stone in order to build the assignment graph with a small addition. Every node of the assignment graph is labeled with the memory requirements of the corresponding module. Let t be the processor with limited memory and s the other processor. Rao et al. show that, in finding the minimum weight feasible solution of the problem, it is sufficient to produce the minimal cut between s and t , and then reassign some subset of modules, if need be, from the processor with limited memory t to s . To reduce the runtime complexity, they considered reducing the size of the assignment graph. In the first approach, they constructed the Gomory-Hu tree [29] or G-H tree which reduces the number of edges of the assignment graph to $O(m)$. The G-H tree represents all maximal flows between any pair of nodes of the assignment graph, and all minimal cuts separating them. Rao et al. show that the G-H tree indicates some subset of nodes that cannot be separated by a minimum weight cut. This property is used to build the reduced G-H tree by condensing those nodes of the G-H tree that cannot be separated. This node condensation may be sufficient to reduce the assignment graphs into trivial graphs where the weight of the condensed node is the sum of the weights of its constituent nodes.

The solution to the problem can be found by enumeration using the labels on the nodes. The runtime complexity of this algorithm is bounded by the computation of the G-H tree which is $O(m^4)$.

The second approach describes a more efficient reduction technique to solve the cases where little or no condensation occurs. This reduction technique can be applied directly to the assignment graph as well as to the G-H tree. It implies the creation of the inclusive cut graph whose cuts are possible minimum feasible cuts and which leads to a reduction at least as powerful as the reduced G-H tree.

1. For each node N in $\min(t)$, find the minimal cut $N(s)$ assigning N to s . This step can be done by running Maxflow-Mincut on the graph to be reduced after setting to ∞ the capacity of the edge connecting N to s . Let $N[s]$ be the set of nodes on the same side as N and s .
2. Condense in a single node all nodes that have equal minimal cuts resulting in step 1. Repeat this until no further reduction is possible. Condense all nodes in $\min(s)$ with s into s^+ .
3. Label nodes in the condensed graph by M_1, \dots, s^+ and T . Each node M_i of the condensed graph represents a subset of the assignment graph which cannot be separated by a minimal weight feasible assignment. The condensed graph is a directed graph where a directed edge from M_i to M_j implies that $M_i[s]$ is included into $M_j[s]$. The last node in the condensed graph is s^+ . Each node in the condensed graph is labeled with a weight equal to the cut $N_i(s)$.
4. A node T is added to the graph with an arc from T to each node with no arc directed to it.

In this case also the weight of the condensed node is the sum of the weights of its constituent nodes. Rao et al. show that the minimum weight feasible cut of the assignment

graph corresponds to some cut in the inclusive cut graph which can be found by enumeration.

This problem can be solved in $O(m^4)$ time which is the time required to build the inclusive cut graph.

To solve the problem of assigning a distributed program of m modules to a fully connected heterogeneous system of n processors, Lo [25] generalized Stone's model [4], that solves the basic dual assignment problem using network flow techniques, to reach a partial (possibly complete) assignment of modules to processors. In the case of complete assignment the solution is optimal. Lo uses two heuristic algorithms in order to improve the partial assignment and sometimes reach the optimal solution. This problem is known as the general task assignment problem where the objective is to minimize the sum of execution and interprocessor communication costs. The solution uses an undirected weighted graph to represent the interconnection structure of the program. The weights on the edges connecting two nodes represent the communication cost between them. The assignment graph is derived from the program graph by adding n nodes representing the n processors. An edge connecting each module i to each processor q is labeled with $w_{iq} = \frac{1}{n-1} \sum_{r \neq q} x_{ir} - \frac{n-2}{n-1} x_{iq}$ (1). In the case where $n=2$, (1) leads to the reverse labeling described earlier in the dual assignment problem and its variation. A n -way cut in such graph is defined to be a set of edges which partition the graph into n disjoint subsets with exactly one processor node in each subset and thus corresponds to an assignment of modules to processors. The cost of an n -way cut is defined to be the sum of the weights on the edges in the cut. Thus, the use of (1) makes the cost of the n -way cut equal the total sum of execution and communication cost of the corresponding assignment. Finding the optimal n -way cut is NP-hard for $n > 2$ [25], therefore Lo uses a Maxflow-Mincut algorithm to find a partial assignment that is to be improved using two heuristic algorithms to reach better solutions.

Lo's solution is divided into three algorithms: (i) Grab, (ii) Lump, and (iii) Simple Greedy. If the first algorithm does not allocate all the modules, the second is used and so on. In Grab, the assignment graph is converted such as we can run a Maxflow-Mincut algorithm between each node P_i representing a processor and a supernode \bar{P}_i representing all other $n-1$ processors. The weights on the edges connecting each module node to the supernode \bar{P}_i is equal to the sum of all weights on the edges connecting that module node to the $n-1$ processor nodes. After applying the Maxflow-Mincut algorithm, the n -processors assignment graph is modified by eliminating the module nodes already assigned and by recalculating edge weights to reflect the partial assignment. Thus, the nodes representing assigned modules are removed from the graph with all their incident edges. Then, the execution cost of each of the unassigned modules is redefined as follows. The execution cost x_{iq} of module i assigned to processor q is augmented by the sum of communication costs between module i and all modules assigned to processors other than q . In mathematical terms, $x_{iq} = x_{iq} + \sum_{r \neq q} \sum_{j \in S_r} c_{ij}$ (2). Finally, the weight of the edge connecting each unassigned module node to a processor is calculated using equation (1) by replacing x_{iq} using equation (2). Grab continues iteratively until no further assignment of modules to processors occurs. If all the modules are assigned after k iterations Grab the solution is optimal.

The algorithm Lump deals with the remaining unassigned nodes. It tests the possibilities of assigning all the remaining modules to one processor. It works on a reduced graph which is derived from the last graph of Grab by eliminating processor nodes and the corresponding edges. Lump computes a lower bound L on the cost of the optimal n -cut from the reduced graph under the constraint that more than one processor be utilized in the corresponding assignment. The lower bound is calculated as $L = \sum_i \min(x_{ip}) + \min_{i \neq r} c(r, i)$

where the first term is the execution costs incurred if each of the remaining modules is assigned to its cheapest processor, and the second term is the minimum of all mincuts between module node i and any other module node. These mincuts represent the communication costs between modules executing on different processors. Based on this lowerbound, the algorithm Lump checks if it is cheaper to assign all remaining modules to one processor yielding a minimum total execution cost for these modules. In this case, the resulting assignment in combination with that of Grab is optimal.

The third algorithm Simple Greedy is used if Lump fails to allocate all the remaining modules. Simple Greedy begins by calculating C which is the average communication costs between all pairs of modules in the reduced graph. Initially, each module is a cluster by itself. Then all edges with $c_{ij} \leq C$ are marked visited. For all unvisited edges, Simple Greedy merges clusters on both sides of each edge into a single cluster until all edges are visited. Then each cluster is assigned to the processor which minimizes its total execution cost.

The runtime complexity of Grab is $O(nm^2 e \log m)$ using a Maxflow-Mincut algorithm of complexity $O(me \log m)$ [15,25] applied at most for m iterations to find n mincuts at each. Lump is $O(m^2 e \log m)$ since the computation of the lower bound L involves finding $n-1$ mincuts in the reduced graph. Simple Greedy examines each edge of the reduced graph exactly once which is $O(e)$. Thus, the overall runtime complexity of the algorithm is upper bounded by that of Grab which is $O(nm^2 e \log m)$ time.

When trying to minimize the total execution and communication costs, one may face the problem that even when the optimal solution is found, one or more processors may be assigned more jobs than others, which causes less concurrency in the system.. Lo [25] showed that by including interference costs between two modules assigned to the same processor, we can still reach the same optimal solution with more concurrency in the system.

Interference costs serves as repulsive forces between modules assigned to the same processor, in the same way communication costs serves as attraction costs. The interference cost between two modules i and j on processor k can be stored in $O(nm^2)$. To account for interference costs, Lo modifies equation (1) by considering that interference costs I_{ij} , between two modules i and j , are independent of the processors to which they might be assigned. The weight of an edge connecting module node i to processor node q is calculated

as $w_{iq} = \frac{1}{n-1} \sum_{r \neq q} x_{ir} - \frac{n-2}{n-1} x_{iq} + \frac{1}{2(n-1)} \sum_{1 \leq l \leq m} I_{il}$. Also, the cost connecting two module nodes i and j , is weighted by $c'_{ij} = c_{ij} - I_{ij}$. If $I_{ij} \leq c_{ij}$, for all c_{ij} , then all weights c'_{ij} are positive, and then we can apply the same sequence of algorithms to the assignment graph. If there exist an edge with negative c'_{ij} , then we can directly apply Simple Greedy to find suboptimal solutions.

Hui and Chanson [19] also use the preprocessed interaction graph (PTIG) to find a heuristic solution for the problem they solved optimally (above) in both cases where the processors are homogenous or heterogeneous. In [19], they reformulated the labeling on the edges of the problem graph in order to create the preprocessed interaction graph (PTIG) where each edge has only one label. The heuristic algorithm uses bin-packing to compute the allocation, where the lower bound for the bin size is equal to zero, the upper bound is the elapsed time if all the tasks are allocated to the fastest processor, and the bin size is selected using a binary search over that range. Initially, a profit function is computed for each edge. This function represents the reduction in elapsed time if the head and the tail nodes of the edge are merged. In each iteration, an expected bin size that represents the overall elapsed time is chosen for all processors. First, the edges are sorted in descending order by profit, and nodes are tested for merging in descending order by profit and merged if they satisfy the bin

size until no more profit could be achieved. Second, the algorithm considers merging nodes with small elapsed time if the overall elapsed time does not increase in order to reduce communication costs and the number of processors needed while obeying the bin size. Thus, it sorts edges in descending order by merge gain in communication and merges pair of nodes until no merge gain could be achieved. Third, the nodes with little or no communication are tested for merging in order to further reduce the number of needed processors while obeying the bin size. Thus, nodes are sorted in order of elapsed time and the nodes with largest and smallest elapsed time are merged when a reduction in overall elapsed time is possible, and the merged node is moved to the front of the list. If no merging occurs, the largest node is discarded. In both cases, the process is repeated until all nodes are tested. If after the three merging steps the allocation is successful, the bin size is reduced to check the possibility of achieving a better overall elapsed time; otherwise, the bin size is increased to get a valid allocation. When all the possible bin size values are tested, the second and the third steps are repeated with taking the total elapsed time achieved till now as a bin size aiming for a better allocation.

For a system with homogenous processors, the runtime complexity of the algorithm is $O((m + e) \log^2 m)$ time, while for a system with heterogeneous processors the runtime complexity is $O((m + e)m \log m)$ time. The difference results from the cost of updating and retrieving the elapsed time of each node which is used in computing the overall elapsed time after each iteration, since a heap is used for the first case, while an array is used in the second.

The problem of assigning an arbitrary parallel program of m modules having unit execution time onto a fully connected heterogeneous system with n processors communicating via homogenous communication links was solved heuristically by Bowen et al. [8]. The objective is to find an assignment that minimizes the interprocessor

communication while observing lower and upper bounds utilization for each processor. Their solution is divided into two parts: (i) heavily communicating tasks are hierarchically clustered to form a cluster tree, then (ii) clustering information is used to allocate tasks to processors. The program is modeled with an undirected graph where each module is represented by a node and each edge connecting two nodes represents the communication between them and is weighted with the cost of this communication. The clustering algorithm selects first as a pivot node the one having the largest adjacent edge. Ties are broken first by greatest number of edges and second by selecting the lowest numbered node. Next, the neighbors of the pivot node which are not yet clustered are sorted in descending order by the weight of the edge which connect them to the pivot node. The nodes whose adjacent edge weights pass a threshold test are clustered with the pivot node and the edges are updated accordingly. This second step, is called recursively to cluster neighbors of the neighbors up to a depth k . These two steps are repeated with a new non clustered node as pivot until all nodes are clustered which consists one pass of the algorithm. The algorithm is repeated until all nodes are clustered into one node which represents the root of the cluster tree. Once a cluster tree is generated, the allocation algorithm attempts to allocate the task tree on the multiprocessor system represented by a cluster tree. At each node, starting from the root node, the module tree is altered so that to have the same number of children as the processor tree while obeying to the lower and upper bound constraints of each processor cluster. The algorithm first runs on the root nodes of the module and processor trees as parameters. It considers the processor which is farthest from meeting its minimum workload constraint until all the children of the root of the module tree are allocated or all processors have met their minimum workload constraint. If all processors have met their minimum workload requirements and there are still module which are not yet assigned to processors, the processor closest to the minimum workload is allocated first. When all modules are assigned, the module tree is modified such

that it has the same number of children at the root node as that of the processor tree. The allocation algorithm is recursively repeated at the children of the root nodes of both trees until both trees become similar. The module to be assigned is selected as the child node having the highest weight. If the assignment of that node deprives the other processors from meeting their minimum workload requirements, the node, which represents a cluster, is popped from the tree, replaced by its children, and the selection process is repeated which enables an allocation at finer grain. The runtime complexity of the clustering algorithm is $O((3e + (d + 1)m) \log m)$ time with d being the node degree. For a small fixed d and a sparse problem graph, the complexity reduces to $O(m \log m)$ time. The allocation algorithm is a linear algorithm which visits each node of the module tree exactly once and tries to map its children to the children of a similar node of the processor tree, thus its runtime complexity is $O(m)$ time.

Another aspect of the assignment problem is assigning a set of similar programs or modules that communicate together to an array of similar processors. Both the module and the processor interconnection structures are represented using graphs. The goal is to place, as much as possible, two directly connected modules on two directly connected processors in order to minimize interprocessor communication. This problem is known as the mapping problem.

The problem graph is denoted by $G_p = \langle V_p, E_p \rangle$ and the array processor graph by $G_a = \langle V_a, E_a \rangle$. The quality of the mapping is measured by the number of edges in G_p that are mapped onto G_a , and is called the cardinality of the mapping. Bokhari [4,5] shows that the mapping problem is computationally equivalent to the graph isomorphism problem. In [4,5] the graph of the array processor is that of the Finite Element Machine, FEM, with

$N(= n \times n)$ processors. Each processor is interconnected to its 'eight-nearest neighbors'. A time shared global bus connect all processors that are not directly connected. A heuristic algorithm, which in most case derives a near optimal solution, is implemented to solve the mapping problem.

Initially, the problem graph G_p , represented by an adjacency matrix, is mapped onto the array processor graph G_a which is also represented by an adjacency matrix. The algorithm attempts to improve this initial mapping by applying a series of pairwise interchange. For each node of the problem graph, a pairwise interchange with all other nodes is considered, and the one that leads to the higher gain in cardinality is kept. When no more improvement could be done, an interchange of n randomly selected pairs of nodes is done and the pairwise interchange is applied again to the resulting mapping. Tests show that the random interchange will not directly lead to better mapping, but the repeated application of pairwise interchange improves the mapping in most cases. Bokhari proves the validity of the algorithm by trying to map random permutations of the array processor graph on the array processor graph. The mapping algorithm takes $O(N^3)$ where N is the total number of processors in the FEM and is equal to n^2 processors.

CHAPTER IV

THE SCHEDULING PROBLEM

The second aspect of distributed computing is the scheduling problem where modules of a distributed or a parallel program are to be partitioned over a multiprocessor system and the order of execution of each module must be unambiguously specified. The general scheduling problem, where precedence relations are of the general type (i.e. transitive edges are allowed, the task processing times are different, and the number of parallel processors is arbitrary) is NP-Hard in the strong sense [22], and is also NP-complete in many restricted cases [14,16,17,22]. Unless $P=NP$, it is impossible not only to find a polynomial time optimization scheduling algorithm, but also a fully polynomial time approximation algorithm [22]. We mention, however, that few optimal solutions exist for some restricted cases, for this reasons, many researchers focus on heuristic solutions for finding near optimal solutions for this problem in short time. In this chapter, we divide these heuristic solutions into two major categories: algorithms providing optimal solutions and algorithms providing near optimal solutions. Under each of these categories we discuss the problem for the cases where (i) the communication costs are ignored and (ii) the communication costs are considered.

4.1. Optimal scheduling algorithms

In this section, we review scheduling problems to which an optimal solution is derived. Most of these algorithms assign a certain priority factor to each module based on its position with respect to its successor and predecessor modules. Some other algorithms use dynamic programming or least cost algorithms to find an optimal solution.

4.1.1. Without communication costs

In [14], Hu solves the problem of scheduling a tree structured program of m modules on a homogeneous fully connected machine with n processors. Modules are assumed to have equal execution times, and the goal is to minimize the sum of execution time of all modules. The algorithm first computes the level of each node in the task graph which is used as each node priority. Then, whenever a processor becomes available, the unexecuted ready task with the highest priority is assigned to it. Hu's algorithm, known as the level algorithm, provides an optimal solution for an in-forest or an out-forest task graph of m tasks in $O(m)$ time, where each task has the same weight.

The second algorithm [14] solves the problem of scheduling m interval-ordered tasks having identical weights on a homogenous fully connected machine of n processors. The objective is to minimize the sum of execution time of all modules. In this case, the number of successors of each node is used as its priority. Then, whenever a processor becomes available, it is assigned the unexecuted ready task with the highest priority. This algorithm solves the problem in $O(e + m)$ time.

Also, in [14] Coffman and Graham consider the problem of scheduling an arbitrary task graph on two homogenous processors, where the m tasks have identical weights. The goal is to minimize the sum of execution time of all modules. The algorithm starts by assigning labels $1, 2, \dots, i$ to the i terminal nodes. Next, from the set of unlabeled nodes with no unlabeled successors, we assign label $i + 1$ to the node having the smallest decreasing sequence of integers, in lexicographical order, formed by ordering the set of the labels (priorities) of its immediate successors. After labeling all tasks, whenever a processor becomes available, assign it the unexecuted ready task with the highest priority. The runtime complexity of finding the optimal solution is $O(m^2)$ which is the time required to assign node priorities.

4.1.2. With communication costs

The first three algorithms described next attempt to assign two communicating tasks to the same processor in order to minimize communication costs, since intraprocessor communication is considered to be negligible. The fourth algorithm schedule tasks to processors in well defined time phases where the execution and communication expenses of the assignment are minimized.

The first problem is that of scheduling a tree structured program of m modules on two homogenous processors, where module execution costs and intermodule communication costs are equal. The main idea of the algorithm [14] is to augment the task graph with precedence relations that compensate for communication costs. Scheduling the augmented task graph with no communication cost is the same as scheduling the original task graph with communication. The algorithm presented works on an in-forest, however, with minor modifications, the algorithm performs equally on out-forests. The algorithm first identifies the sets of siblings S_1, S_2, \dots, S_k in the in-forest. Then, for every set S_i , let u be the node with maximum depth. Disconnect all edges connecting all nodes in S_i to $child(S_i)$ except for node u , and connect these nodes to node u . Next, obtain a schedule by applying Hu's level algorithm [2] on the augmented in-forest. Finally, for every set S_i of the original in-forest, if node u with the maximum depth is scheduled in the time slot immediately before $child(S_i)$, but on a different processor, then exchange $child(S_i)$ with the task scheduled in the time slot immediately after u on the same processor. Using this algorithm, the optimal solution can be found in time $O(m^2)$.

Next, Ali and El-Rewini [14] solves the problem of scheduling m interval-ordered tasks on a homogenous fully connected machine having n processors. Their algorithm assumes that all tasks have equal execution costs which is also identical to the

communication delay. The algorithm works as follows. First, the priority of each node is computed as the number of its successors, where ties are broken arbitrarily. Then, the node with the highest priority is scheduled first, and it is assigned to the processor which ensures its earliest starting time. If for a node i , more than one processor ensure the same earliest starting time t , then node i is scheduled to run on the processor which in time slot $t - 1$ was assigned the task with lowest priority. We note that a processor which is not assigned at time slot $t - 1$ has priority zero. The runtime complexity for finding the optimal solution for a graph of m tasks and e edges is $O(me)$.

Third, Varvarigou et al. [36] solved the problem of scheduling an out-forest structured program of m modules over a fully connected processor system having homogenous communication links. The execution cost of a module of the program and the cost of communication between two modules scheduled to run on different processors are assumed to take each one unit time. The number of processors is bounded by a constant C , and the objective is to minimize the sum of execution and communication time. The main idea of the solution is to transfer the problem graph into an equivalent free delay graph that can be scheduled without considering communication costs. First, the algorithm recursively builds the shortest delay free graph of the problem graph which is an out-forest. For a node i , select the child node j with the highest weight. Ties are broken arbitrarily. Disconnect all edges connecting i to all its children except for j , and connect all disconnected nodes to node j . Thus, in $O(m)$ time the shortest free delay trees for all subtrees can be calculated. Then, the algorithm uses the theorem, by Dolev and Warmuth [35]: given an out-forest precedence graph G , and an optimum schedule for the high subgraph of G , there is an $O(m)$ time algorithm that computes an optimal schedule for the whole graph G . So, the algorithm finds the median, of the shortest delay free graph corresponding to the out-forest, which divides

that graph into a high subgraph and a low subgraph. In order to be able to apply Dolev and Warmuth theorem, the algorithm uses dynamic programming to compute the length of the optimal schedule for all the high subgraphs corresponding to the high shortest delay subgraphs which is equal to computing the optimal schedule of the graph that contains at most $n - 1$ initial components. Then, the complete optimal schedule is computed using Dolev and Warmuth theorem. The resulting schedule divides the nodes into sets that correspond to different time slots, for example the set $S(k)$ is assigned to time slot k . These sets are allocated to processors such that the communication delay constraint is not violated. All nodes of $S(1)$ are assigned to processors at random, then for every node x in $S(k)$ if the parent of x is scheduled in time slot $k - 1$, then x is assigned to the same processor, otherwise it is assigned at random. The runtime complexity of the algorithm is bounded by the time to find the optimal schedule for all high subgraphs which is $O(m^{2n-2})$. Although this algorithm is designed for out-forests structured programs, it can be used to solve the same problem for in-forests, since an in-forest reduces to an out forest by reversing the edge directions and then inverting the resulting optimal schedule.

Fourth, Bokhari [4] provides an optimal solution for the problem of scheduling an out-tree structured parallel program to a fully connected system with n heterogeneous processors and heterogeneous links where the costs (expenses) over the distributed system are assumed to vary with time, that is the cost of processing a task on the system is processor and time dependent. Moreover, the interprocessor communication costs (expenses) between two processors depend on their common link and on the time during which the communication took place. Thus, processing and communication costs are considered with respect to well predefined phases of time. The goal is to minimize the cost (expense) of executing the program, and the penalty for not meeting deadlines by trying to assign tasks to phases where

all costs could be minimized. The optimal solution is obtained using the shortest tree algorithm on an assignment tree of m tasks, n processors, and ϕ phases in $O(mn^2\phi^2)$ time.

4.2. Near optimal scheduling algorithms

In this section, we review heuristic algorithms which provide a suboptimal solutions for the scheduling problem. Some of these algorithms assign a certain priority factor to each module based on its position, or its weight with respect to its successor and predecessor modules, or even based on more complex priority factors. Some other algorithms use clustering based on the edge zeroing technique to find a suboptimal solution.

4.2.1. Without communication costs

The problem is to find the schedule that minimize the total execution time of an parallel or distributed program of m modules on an homogenous fully connected machine of n processors. Communication costs among tasks are negligible, and thus they are not accounted for in the problem which is modeled by a directed acyclic graph (DAG) of $O(m)$ nodes.

The first algorithm, by Shirazi et al. [31], is the Heavy Node First algorithm (HNF). The DAG modeling the problem must have no redundant edges. In HNF, all entry nodes have depth equal to zero. From all the ready nodes with the same depth, HNF assigns first the heaviest node to the processor that insures its earliest execution time. Whenever all nodes of the same depth are assigned, the algorithm works on the ready nodes of the next depth until no more nodes are to be assigned. By using a heap to store the ready nodes at each depth, HNF can provide a schedule which is no worse than twice the optimal schedule in $O(m \log m)$ time.

The second algorithm is the Critical Path Method algorithm CPM [31], which uses the critical path method on a DAG, a generalization of Hu's optimal level algorithm for trees [14]. CPM finds a schedule which is no worse than twice the optimal schedule in $O(m^2)$ time, which is the time to compute node levels.

Next, the Weighted Length (WL) algorithm, also by Shirazi et al. [31], uses an approach similar to the one used in CPM on DAG with no redundant transitive edges. Instead of using the level of each node as its priority, WL computes the weighted length for each node. The weighted length of a node i is the sum of: the weight i , the maximum weighted length of the children of i , and the summation of the weighted length of the children of i normalized over the number of children of i . The weighted length is computed bottom up in $O(m^2)$ time. The algorithm has the same performance as CPM.

Also, Kasahara and Narita [22] presented two algorithms to solve the same problem. Their problem graph must have one entry and one exit node. The first one is the critical path/most immediate successors first algorithm (CP/MISF) [22] which is a modification of the critical path method. In CP/MISF, the level of each node is computed first, and the priority list is ordered in descending order of level and the number of immediately successive tasks. Next, list scheduling is executed on the basis of this priority list. The ready task having the highest level and the highest number of immediately successive tasks will execute first. The worst case performance of CP/MISF results in a schedule length twice as much as the optimal schedule. The runtime complexity of CP/MISF is $O(m^2 + mn)$.

The second algorithm, by Kasahara and Narita [22], is the depth-first/implicit heuristic search algorithm (DF/IHS). DF/IHS method is divided into two parts: (i) the preprocessing part which consists of assigning priorities heuristically to the nodes during search, and (ii) the depth-first enumeration of all possible assignments. In the preprocessing stage, the level of

each node is computed in $O(m^2)$ time. Then the tasks are renumbered using two stage sorting, like in CP/MISF, which takes $O(m \log m)$ time. Thus, the complexity of the preprocessing part is $O(m^2)$. In the depth first search part, a tree that enumerates all possible assignments is created in depth first manner. A path in the tree from the root node to a leaf node consists of a solution of the assignment problem. Because of the priorities assigned in the preprocessing part, the left most path from the root node to a leaf node in the enumeration tree represents the same solution that can be achieved by CP/MISF. Any other path from a root node to leaf node may or may not contain an improved solution. The use of elimination rules within DF/IHS reduces the number of paths to be generated in the case a better solution is not likely to be obtained along that path or a satisfactory approximation is reached. The runtime complexity of the algorithm is $O(n^m)$ which is the time required to enumerate all possible solutions in the case where the elimination rule are set such that all the possible solutions are to be explored.

4.2.2. With communication costs

Varvarigou et al. [35] show that for every out-forest with communication delay, there exists a delay-free out-forest such that their schedule is optimal. They also derived the shortest delay-free out-forest which optimal schedule may or may not correspond to the optimal schedule of the original out-forest and proved that its optimal schedule at most exceeds that of the original out-forest by $(n - 2)$ time units. To solve the problem, the algorithm applies level scheduling on the shortest delay-free graph which provides an optimal schedule for any delay free out-forest. This optimal schedule is a near optimal schedule for the original out-forest with no more than $n - 2$ time units from the optimal schedule. Thus, the level algorithm finds the near optimal solution in $O(m)$ time.

The following four algorithms address the problem of assigning an arbitrary parallel or distributed program of m modules on a completely connected graph with an unbounded number of homogenous processors. A program is modeled by a directed acyclic graph (DAG) where each node represents a module and is labeled with corresponding execution cost, and an edge connecting two nodes is labeled with communication cost between the modules they represent. All four algorithms try to reach the shortest possible schedule by defining a specific goal to minimize. These algorithms use clustering which consists of mapping the tasks of the DAG onto clusters. A clustering is called non linear, if two independent tasks are mapped to the same cluster; otherwise it is called linear.

The first algorithm is Kim and Browne's linear clustering algorithm known as KB/L [16]. The goal is to reduce the length of the longest path determined by a cost function. Initially, all edges are marked unexamined. The first step, determines the longest path composed only of unexamined edges by using a weighted cost function. The nodes in this path constitute a cluster and their edges are set to zero. In the second step, all edges incident to nodes in the longest path are marked examined. Both steps are applied recursively until all edges are examined. Instead of computing the longest path as the sum of node computation costs and of edge communication costs to find the longest path, a weighted cost function is used : $Cost_function = w_1 * \sum t_i + (1 - w_1)(w_2 * \sum c_{i,j} + (1 - w_2) * \sum c_{i,j}^{adj})$. In this cost function, w_1 and w_2 are normalization factors, while $\sum c_{i,j}^{adj}$ represents the sum of the communication costs of all edges adjacent to a node in the path. If $w_1 = \frac{1}{2}$ and $w_2 = 1$, then the cost function will represent the sum of node computation costs and of edge communication costs. Finding the longest path at each node takes $O(m + e)$ time. Therefore,

the complexity of KB/L is $O(m(m+e))$ time. For a dense graph $e = m^2$, the complexity becomes $O(m^3)$.

The second algorithm is Sarkar's [16] algorithm which zeros the highest edge if the parallel time does not increase. Initially, each node is considered to form a single cluster. First, all edges of the DAG are sorted in descending order of edge costs. Then, the highest edge is zeroed if the parallel time does not increase, and this step is repeated until all edges are scanned. In this algorithm, when two clusters are merged, the task within the new cluster having the highest bottom level is scheduled to execute first. Thus, at each clustering step, bottom levels of all tasks must be computed in order to decide which task to schedule first. This computation is done at most e times and it costs $O(m+e)$. Therefore, the complexity of the algorithm is $O(e(m+e))$ time.

The third algorithm is the dominant sequence clustering algorithm, by Yang and Gerasoulis [16], known as the DSC algorithm, and its goal is to minimize the dominant sequence (DS) in the problem graph. At the beginning, all edges are marked unexamined. An edge considered for zeroing is marked visited, and its head node is scheduled. At the completion of a clustering step, two sets of nodes are updated, the scheduled set, and the unscheduled set which initially contains all nodes to be scheduled. The algorithm works as follows :

1. Suspend zeroing an unexamined edge (m,y) in DS until the head node y becomes free, i.e. all its predecessor nodes are scheduled, which insure a breadth first traversal of the graph.
2. Choose a free node x which belongs to the longest path going through any of the free nodes and zero its incoming edge(s) provided the following two conditions are satisfied :
 - i) (CT1) If the starting time of node x decreases.

ii) (CT2) Zeroing incoming edges of node x to minimize the top level of x should not affect the strict reduction of the top level of y at some future step j , $i \leq j$, which means that incoming edges of x can be zeroed only if a strict reduction in the parallel time can be obtained.

3. If all edges in a DS have been examined and this DS continues to dominate in the next step, then recursively apply the above steps on the next longest path (SubDS) to reduce the number of unnecessary processors.

The free node chosen in step 2 is the free node having the highest priority where the node priority is the sum of the top level and the bottom level of that node. Ties are broken using the most immediate successor first (MISF) strategy. The free node with the highest priority will be scheduled on the processor that allows its earliest execution. If no such processor exists, then it is scheduled on a new processor.

Re-computing top levels takes $O(v + e)$ time per step. More reduction in complexity can be obtained by computing the start bound of a node instead of its top level which can be achieved in $O(e \log v)$. Updating priority lists takes $O(\log v)$ time, and since there are v steps the cost is $O(v \log v)$. Traversing the graph takes $O(v + e)$. Thus, the total time complexity of DSC is $O((v + e) \log v)$, and the space complexity is $O(v + e)$. For linear clustering the cost reduces to $O(v \log v + e)$.

The fourth algorithm is the modified critical path algorithm, by Wu and Gajski [16], known as the MCP algorithm. The goal is to schedule at the earliest possible time the tasks with the highest priorities in the critical path. First, the algorithm determines a priority list based on the highest bottom level first ordering. Ties are broken by using the highest level of its successor task, the successor of its successors and so on. Then, while there exists an unscheduled task, it finds an unscheduled free node with the highest priority in the priority

list, then schedule this task to a processor (cluster) that allows its earliest execution. The worst time complexity of the MCP algorithm is $O(m^2 \log m)$ because of the cost in the tie breaking. If there are no ties the complexity is similar to DSC.

4.3. Some other problems and techniques

Many other variations of the allocation and scheduling problem, and many other solutions techniques are derived to find optimal or near optimal solutions for both problems. We briefly mention some of these algorithms since they do not fall into our classification. For example, the problem of scheduling parallel programs on distributed memory parallel architecture where duplication of tasks on more than one processor is allowed [28] and the problem of scheduling compute-intensive tasks in the idle time of a network of workstations [12] are solved. The A* algorithm in artificial intelligence is used to solve the task assignment problem based on the minimax criterion [30], and a new mapping heuristic is derived based on mean field annealing to solve the task allocation problem [9].

CHAPTER V

A NOTE ON THE ASSIGNMENT PROBLEM OF ARBITRARY PROCESS SYSTEMS TO HETEROGENEOUS DISTRIBUTED COMPUTER SYSTEMS

In [8] the authors proposed a clustering algorithm which creates clusters of frequently communicating nodes in a distributed computer system. The system is modeled as a graph where each node represents a module of a parallel program and each edge represents the cost of communication between the corresponding nodes. The purpose of the algorithm is to hierarchically create the kind of clusters which minimize the communication cost.

In the clustering algorithm, two nested while loops insure that the graph is completely hierarchically clustered. The outer loop insures that the graph is reduced to a tree of clusters where the root node represents the entire graph as single cluster. Each iteration of the inner loop deals with the creation of a single cluster which will be represented by a single node in the above mentioned tree. This single cluster is denoted by C in the algorithm and is cleared only before entering the inner loop (Fig. 5.1).

Since each iteration of the inner loop deals with the creation of one single cluster C , the value of C must be initialized at the beginning of each. If not, as written in [8], the nodes of the graph will be grouped into a single cluster and the algorithm will fail. Therefore, we make our first correction which initializes C immediately after entering the inner loop:

```
DO while  $c(v') = 0$  for some  $v' \in V'$  /* Nodes not clustered */  
  Clear  $C$  /*  $C$  will hold all nodes of a single cluster */.
```

Our second correction is related to Fig. 8 of [8]. That figure represents a worst case example of the clustering algorithm. The example describes a graph of 10 nodes as shown in Fig. 5.2. We recall that to create a cluster the algorithm starts by selecting a pivot node as follows :

- A pivot is a node of the graph that is not yet clustered.

Given a graph

$G = (V, F)$ where V is a set of nodes and F is a set of edges:

$V = \{v_i, \dots, v_j\}$, $F \subseteq V \times V$ and:

If $(v_i, v_j) \in F$ then $e_{ij} \in E$ is the weight.

$G' = G$

DO while $|V'| > 1$ /* more than 1 node in V' */

$c(v') = 0 \forall v' \in V'$

Clear C /* C will hold all nodes of a single cluster */

DO while $c(v') = 0$ for some $v' \in V'$ /* Nodes not clustered */

Select a PIVOT ($v_p \in V' \wedge c(v_p) = 0$):

Select node with greatest $e_{pj} \ni v_j \in V' \wedge c(v_j) = 0$

Break ties by greatest number of edges

Break further ties by lowest numbered node

Rank_Neighbors(k)

Update G' :

Mark pivot as clustered ($c(v_p) = 1$)

$\forall v_i \in C, v_i \neq v_p$ remove v_i from V'

If $(v_i, v_j) \in F' \wedge v_i, v_j \in C$ then remove (v_i, v_j) from F'

$\forall v_i, v_j \ni v_i \in C \wedge v_j \notin C \wedge (v_i, v_j) \in F' \wedge v_i \neq v_p$

Remove (v_i, v_j) from F'

If $(v_p, v_j) \notin F'$ then add (v_p, v_j) to F' ; $e_{pj} = e_{ij}$

If $(v_p, v_j) \in F'$ then $e_{pj} = e_{pj} + e_{ij}$

Record v_p and members of C as belonging to the same cluster

END /* nodes in V' not clustered */

END /* more than one node in V' */

Rank_Neighbors(k)

$Q = \{e_{pj} | v_p, v_j \in F' \wedge c(v_j) = 0\}$

Sort Q into descending order

Drop on threshold:

$$Q' = \{Q(1), \dots, Q(t)\} \ni \frac{Q(i) - Q(i+1)}{Q(i)} < T, i = 1 \dots t \wedge \frac{Q(i) - Q(t+1)}{Q(t)} \geq T$$

$C = C \cup V_q$

where $V_q = [p | e_{pj} \in Q']$

If $k > 1$ then Rank_Neighbors($k-1$) for each $v_j \in C$

Fig. 5.1 The clustering algorithm.

- (a) It must be adjacent to a non clustered node having the greatest connecting edge weight.
- (b) If more that one node satisfy (a), the node having the greatest number of incident edges is chosen.
- (iii) If more that one node satisfy (b), the node which has the lowest number is chosen.

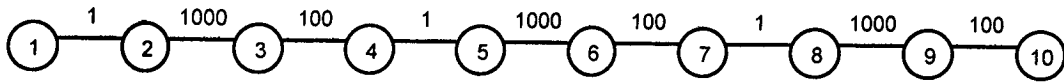


Fig. 5.2 The graph of the worst case example.

Next, the algorithm consider for clustering neighbors of neighbors of the pivot node up to a depth k . If $k = 1$, then only the immediate neighbors of the pivot node are candidates for clustering. Then, all candidate nodes are subject to a threshold test, and only nodes passing the test are clustered.

In the worst case, at most two nodes merge together at each pass, therefore the algorithm must run with $k = 1$, that is, the algorithm will not consider for clustering the neighbors of the neighbors of a pivot node, it will only cluster the direct neighbors which pass the threshold test.

By applying the clustering algorithm, with $k = 1$ and with a threshold $t = 0.25$ as set in [8], on the example of Fig. 5.2, we first find that nodes 2, 4, and 5 are pivot candidates. Node 2 is selected to be the pivot of the first cluster. Although node 2 has neighbors nodes 1 and 3, node 1 is discarded for failing the threshold test. This is how nodes 2 and 3 form a single cluster represented by node 2. Similarly, nodes 5 and 6, and nodes 8 and 9 are clustered and represented by nodes 5 and 8 respectively (Fig. 5.3). This completes one pass of the outer loop.

On the second pass, nodes 2 and 4, nodes 5 and 7, and nodes 8 and 10 are clustered and represented by nodes 2, 5, and 8 respectively (Fig. 5.4). On the third pass, 2 and 5 are

candidates to be selected as pivots. They have the same number of incident edges, but node 2 is selected because it has the lowest number. Nodes 1 and 5 are direct neighbors of node 2

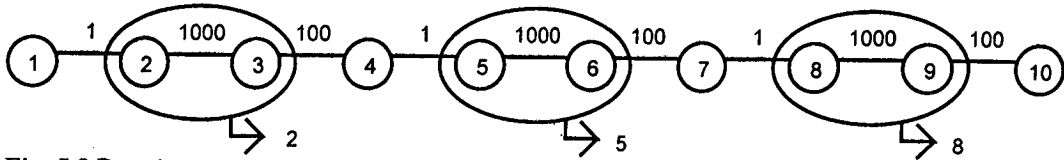


Fig. 5.3 Pass 1.

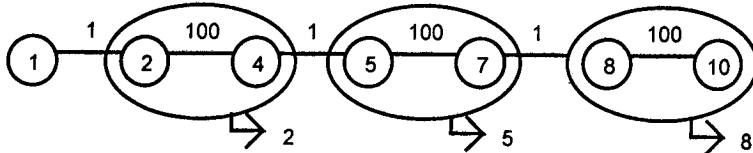


Fig. 5.4 Pass 2.

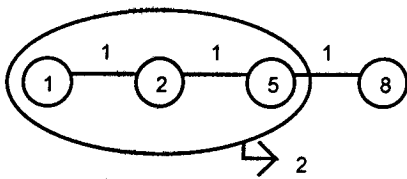


Fig. 5.5 Pass 3.

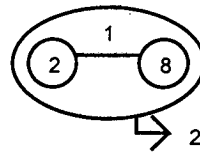


Fig. 5.6 Pass 4.

and they both pass the threshold test. Therefore nodes 2, 1, and 5 are clustered into the same cluster and represented by node 2 (Fig. 5.5). Nodes 2 and 8 are clustered on the last pass (Fig. 5.6). The passes of the hierarchical clustering are represented by a tree (Fig. 5.7).

We note that our final graph is different from the one presented in [8] (Fig. 5.8). We believe that this graph cannot be reached as a result of running the allocation algorithm proposed in [8]. We think our correction is in order.

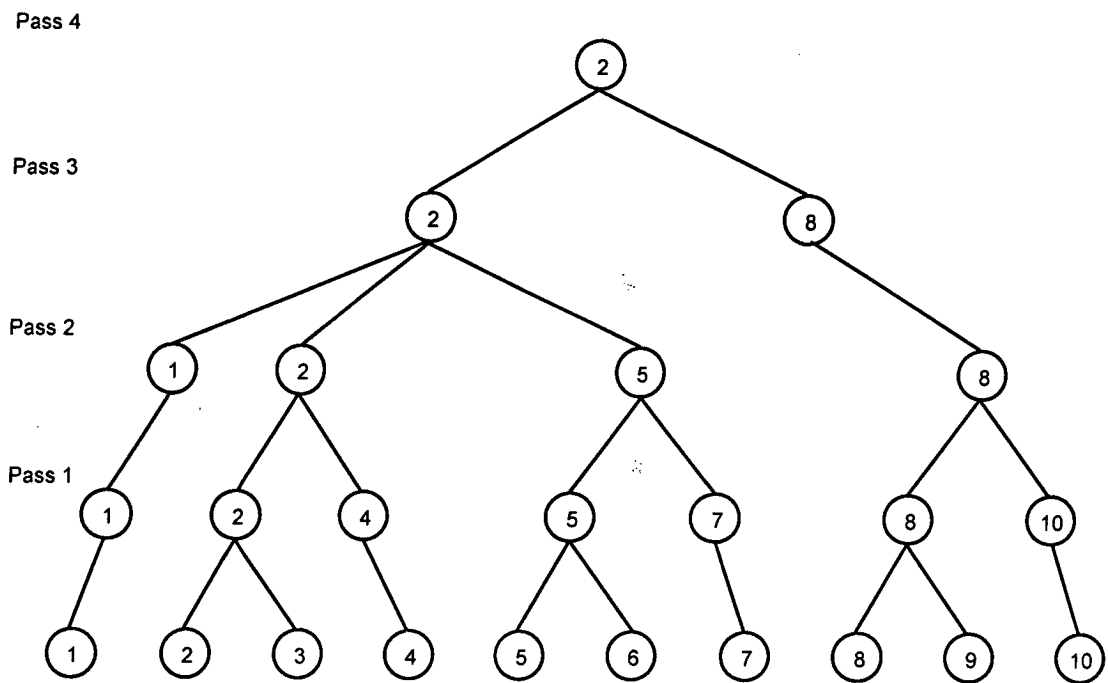


Fig. 5.7 The cluster tree of the worst case example.

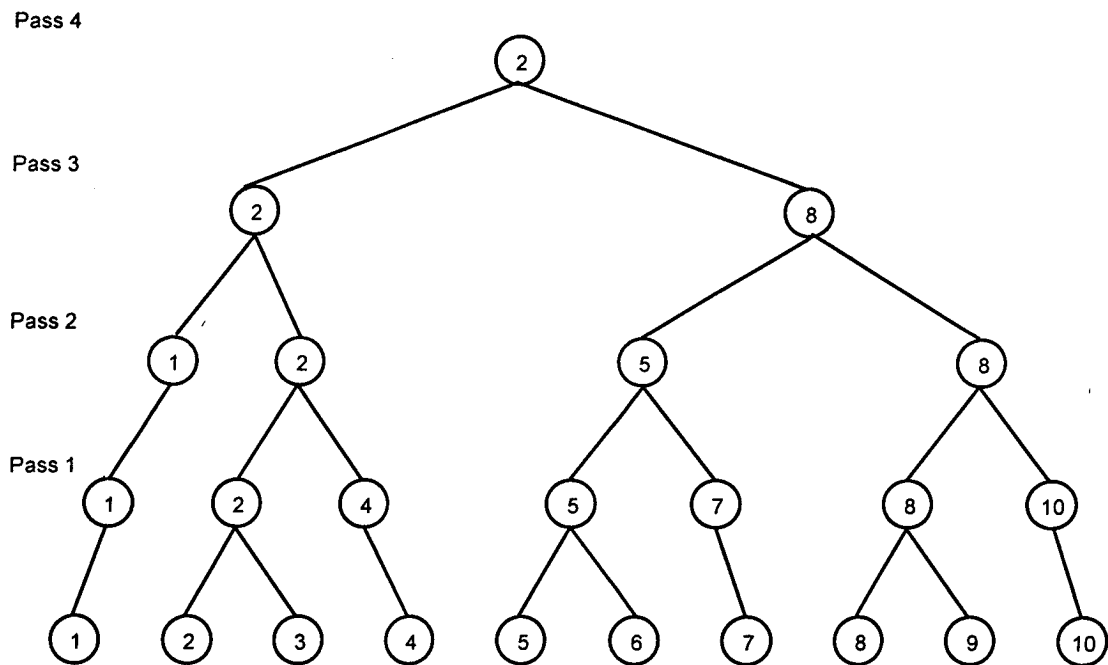


Fig. 5.8 The cluster tree of the worst case example as shown in Fig. 9 of [23].

CHAPTER VI

A VARIATION OF BOKHARI'S LAYERED GRAPH ALGORITHM FOR MAPPING CHAINS ONTO CHAINS IN $O(m^2n)$ TIME USING A REDUCED LAYERED GRAPH OF $O(mn)$ NODES

The problem of mapping chains of modules onto chains of processors, referred to thereafter as MCC, occurs when a packet of data must be subject to a set of operations. For example, the packet of data may have to be Fourier transformed, multiplied by a fixed frequency, filtered ...etc. This kind of operations has a serial chain-like structure. Instead of executing such serial operations on all packets of data using a single processor, it would make a better sense to think of a multiprocessor system having a chain-like structure and try to map the chain of modules to the chain of processors. An example of such assignment is shown in Fig. 6.1 below [4,6,7]. In that figure, each packet of data moves from processor 1 to processor 4 in a unidirectional pipelined fashion. While processor 3 executes operations 4, 5, and 6 on a packet i , processor 2 executes operations 2 and 3 on packet $i + 1$.

Also, this problem occurs when a set of chain-like packets of different sizes must be subject to parallel operations. The problem is to find an optimal assignment of these data packets on the chain-like multiprocessor system where processors can communicate in both directions.

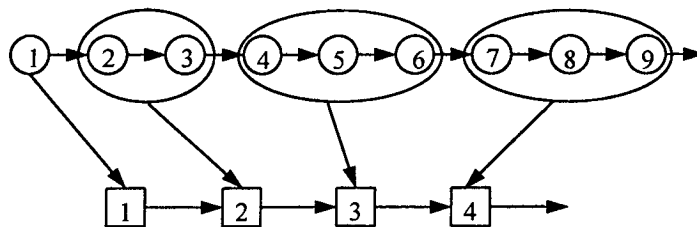


Fig. 6.1 A nine-module chain mapped onto a four-processor chain.

Both problems can be modeled in the same manner as follows. Given a set of m modules connected in a chain-like fashion, and a multiprocessor chain of size $n < m$, we need to find the assignment of subchains of modules to processors that minimize the load on the

heavily loaded processor. The contiguity constraint ensures that two modules that communicate together are assigned to the same or to adjacent processors. The above description of the MCC problem is also used in the next chapter.

The problem of assigning chains of modules onto a chains of processors, where processors are restricted to be homogenous, was studied by Iqbal [20], by Nicol and O'Hallaron [27], and by Iqbal and Bokhari [21], under many assumptions and in the general case. In this chapter, we focus on the general case where the processors are heterogeneous and the communication links between processors are also heterogeneous. This case is solved by Bokhari [4,6,7] and by Nicol and O'Hallaron [27]. We present a new variation of Bokhari's algorithm which runs in $O(m^2n)$ time. More specifically, our algorithm uses a reduced layered graph of $O(mn)$ nodes and $O(m^2n)$ edges. We note that all solutions presented in this chapter assume that all processors are to be utilized. In the remaining figures of this paper, many nodes and edges are omitted to avoid congested diagrams.

6.1. Bokhari's layered graph algorithm

To solve the MCC problem of Fig. 6.1, Bokhari's algorithm [4,6,7] constructs the layered graph of Fig. 6.2 where each layer in the graph corresponds to a processor. A node $\langle i, j \rangle$, $1 \leq i \leq j \leq m$, corresponds to an assignment of the subchain of modules i through j to the processor in that layer. A node $\langle i, j \rangle$ is connected to all nodes $\langle j+1, k \rangle$ in the layer directly below it for all j except 1 and n . All nodes $\langle 1, j \rangle$ ($\langle i, m \rangle$) in the first (last) layer are connected to node s (t). A path from s to t corresponds to an assignment of subchains to processors under the contiguity constraint.

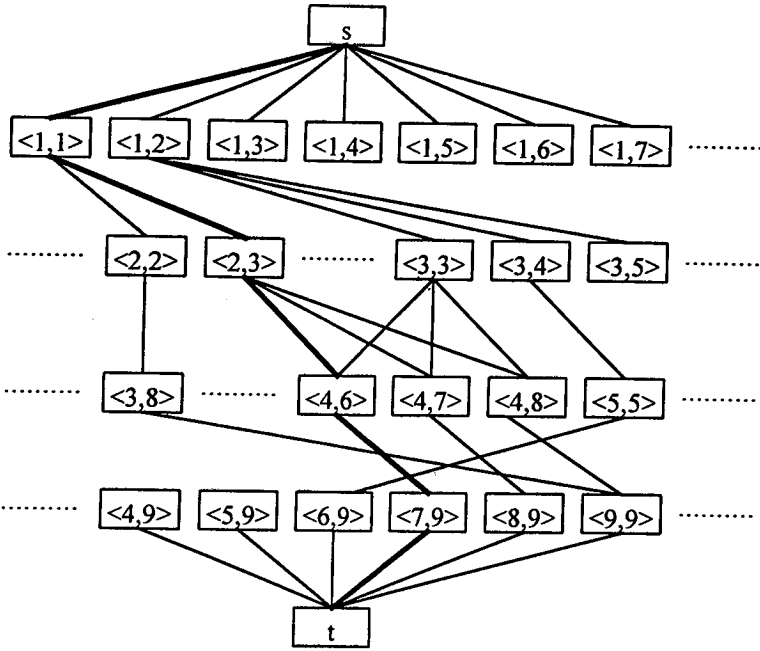


Fig. 6.2 Bokhari's layered graph for the problem of Fig. 6.1.

Edges of the layered graph are labeled as follows. The weight of the edge connecting node $\langle i, j \rangle$ in layer k to any node in the layer below is equal to the cost of executing model i through j on processor k . To account for communication cost between subchains assigned to adjacent nodes, weights may be added as follows. To the weights of each edge connecting node $\langle a, b \rangle$ in layer k to node $\langle b+1, d \rangle$ in layer $k+1$, we add the communication cost between nodes b and $b+1$ over the link connecting processors k to $k+1$.

Using this technique, it is clear that the number of nodes in each layer is in $O(m^2)$. Since the graph is of n layers, the total number of nodes is in $O(m^2n)$, and the total number of edges is in $O(m^3n)$. We note that a variation of Dijkstra's shortest path algorithm [11] solves the problem in $O(m^4n^2)$ time. Due to the layered structure of the graph, Bokhari's solution provides an improved running time; the idea is to find the minimum bottleneck path from node s to node t . Each node i in the layered graph is given a label $L(i)$. Initially, all

nodes are given infinite labels except nodes of the first layer which are given zero label. The algorithm works as follows:

1. Examine each edge e emanating downwards from a layer connecting a node a (above) to a node b (below). Let the weight on this edge be $w(e)$.
2. Replace $L(b)$ by $\min(L(b), \max(w(e), L(a)))$.

Once t is labeled, the path representing the optimal path can be found by tracing backwards from t to s . Both the labeling procedure and finding the optimal path visit each edge of the layered graph exactly once, therefore the overall complexity of Bokhari's algorithm is $O(m^3n)$.

6.2. Nicol and O'Hallaron's variation using an improved layered graph

Using a variation of the layered graph, Nicol and O'Hallaron [27] were able to solve the same problem in $O(m^2n)$ time using $O(m^2n)$ edges. For example, in Fig. 6.2, $n-2$ new layers were added, one between each layer, except between layers 1 and 2. Each new layer consists of m nodes labeled from 1 to m . A node $\langle j, k \rangle$ in layer i (with respect to Bokhari's layered graph) directs a single edge to node k in the new layer between layers i and $i+1$. This edge is labeled exactly as the edge leaving node $\langle j, k \rangle$ in Bokhari's solution. A node k in the new layer directs to all nodes $\langle k+1, l \rangle$ in the layer $i+1$ (see Fig. 6.3). Each edge of this type has a zero weight. A path from s to t corresponds to a solution of this assignment problem.

By adding $m(n-2)$ nodes to Bokhari's layered graph of Fig. 6.2, Nicol and O'Hallaron were able to reduce the number of edges of the layered graph as follows. In the

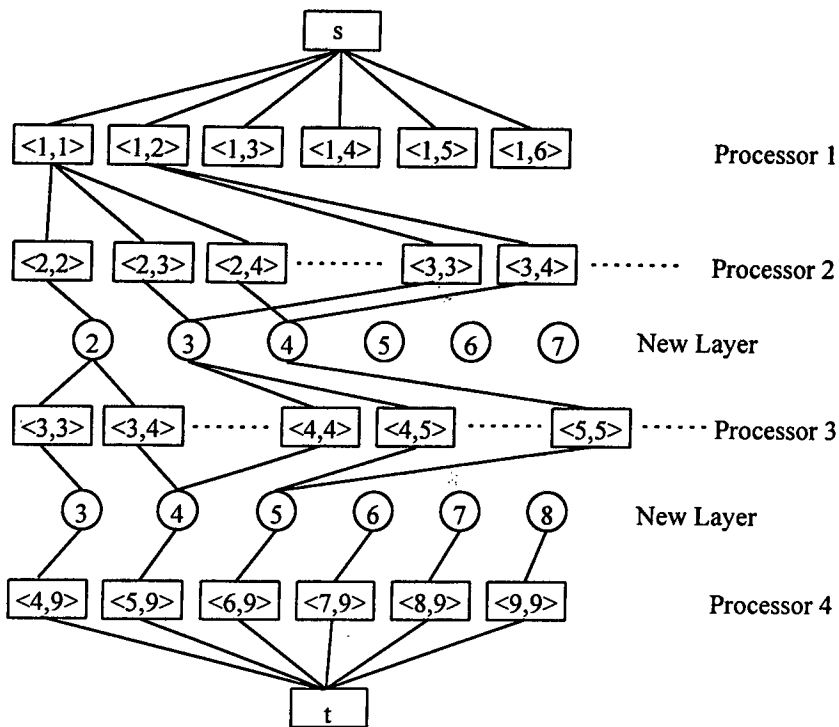


Fig. 6.3 Improved layered graph of the problem of Fig. 6.1.

old layers, each of the m^2 nodes directs one single edge to the layer below, therefore the total number of edges leaving the n old layers is in $O(m^2n)$.

In the new layers, each of the m nodes directs m edges to the layer below, therefore the total number of edges leaving the $n-2$ new layers is in $O(m^2(n-2))$. Thus the total number of edges of the improved graph is in $O(m^2n + m^2(n-2))$ which is $O(m^2n)$. Thus, by using Bokhari's original algorithm on the improved layered graph of Nicol and O'Hallaron, the minimum bottleneck path from s to t can be found in $O(m^2n)$.

6.3. Our variation

Our variation has two parts: (i) a variation of Bokhari's layered graph, and (ii) a variation of Bokhari's algorithm. We begin by defining our reduced layered graph first.

Definition 53: A reduced layered graph, is a graph with n layers and m nodes. Each layer has m nodes labeled from 1 to m . Each node i at layer k is connected to all nodes j of layer

$k+1$, $i+1 \leq j \leq m$ (Fig. 6.4). A node s connects to all nodes of layer 1, while node m at layer n serves also as a terminal node.

A path from node s to node m at layer n in the reduced layered graph consists of a feasible assignment. The optimal assignment corresponds to the path having the minimum bottleneck weight, and it is computed using a variation of Bokhari's algorithm:

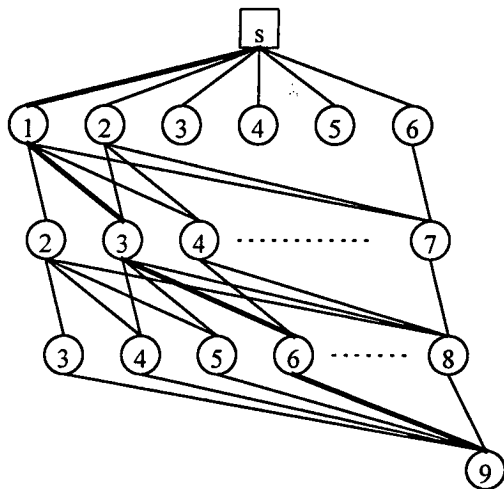


Fig. 6.4 The reduced layered graph for the problem of Fig. 6.1.

Instead of labeling edges, as in the algorithms described earlier, we label nodes as follows:

1. Let $L(a)$ be the label of node a , then $L(s) = 0$.
2. For layer 1 ($k = 1$), each node i is labeled by the cost of executing module 1 through i on processor 1.
3. For layer k , $2 \leq k \leq n$, each node i is labeled by

$$\min \left(\max \left(L(\text{adjacent node } j \text{ in the layer } k-1), \right. \right. \\ \left. \left. \text{cost of executing modules } j+1 \text{ through } i \text{ on processor } k \right) \right).$$

In the reduced layered graph, the number of nodes is reduced to $O(mn)$ nodes. The number of edges is in $O(m^2n)$. Labeling nodes from s to m at layer n takes $O(m^2n)$ time. The path representing the optimal assignment can be traced backwards from node m at layer n to node s . The overall complexity of this algorithm is $O(m^2n)$.

We have traced below a sample run of our algorithm. We assume a chain of four modules to be assigned to a chain of three processors. Table 6.1 shows the execution cost per module on each of the three processors.

	M 1	M 2	M 3	M 4
P 1	2	1	3	2
P 2	4	3	3	4
P 3	3	2	2	4

Table 6.1 Execution cost per module on each processor.

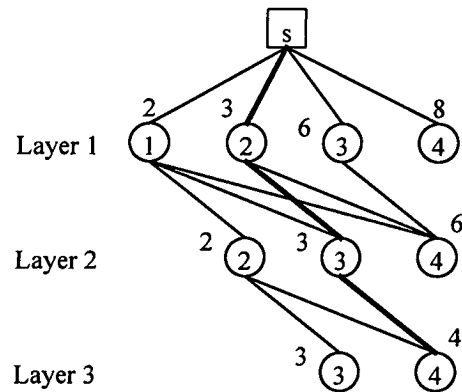


Fig. 6.5 The reduced layered graph of the problem in Table 6.1.

As an example of the labeling procedure of the problem in Table 6.1, we choose to label node 4 of layer 2. In the following, $C(i, j)$ is the cost of executing node i through j on processor k , and $L(a)$ is the label of node a in the layer above. The label of node 4 on layer 2 can be calculated as follows:

$$\min \left\{ \begin{array}{l} \max\{L(1), C(2,4)\} \\ \max\{L(2), C(3,4)\} \\ \max\{L(3), C(4,4)\} \end{array} \right\} = \min \left\{ \begin{array}{l} \max\{2,10\} \\ \max\{3,7\} \\ \max\{6,4\} \end{array} \right\} = \min\{10,7,6\} = 6.$$

A path from node s to node 4 in layer 3 consists of a feasible assignment. The optimal assignment is shown in Fig. 6.5 in bold lines. It represents the assignment of

modules 1 and 2 to processor 1, module 3 to processor 2, and module 4 to processor 3. The label of node 4 in layer 3 is the bottleneck weight of the optimal path, i.e. the weight on the heavily loaded processors corresponding to the optimal assignment.

6.4. Results

In this chapter, we have reviewed the problem of assigning chains of modules onto chains of processors. In particular, we reviewed Bokhari's and Nicol and O'Hallaron's algorithms. Our contribution consists of a variation of Bokhari's work which runs in $O(m^2n)$ time using a reduced layered graph of $O(m^2n)$ edges and $O(mn)$ nodes. A summary of all results is included in Table 6.2.

Problem	Nodes	Edges	Run time
Bokhari	$O(m^2n)$	$O(m^3n)$	$O(m^3n)$
Nicol & O'Hallaron	$O(m^2n)$	$O(m^2n)$	$O(m^2n)$
A new variation	$O(mn)$	$O(m^2n)$	$O(m^2n)$

Table 6.2 Summary of results.

CHAPTER VII

A HEURISTIC ALGORITHM FOR MAPPING CHAINS ONTO CHAINS OF A HOMOGENOUS AND A HETEROGENEOUS PROCESSOR SYSTEM IN TIME $O(m)$ AND $O(mn)$ RESPECTIVELY

In this chapter, we suggest a simple heuristic solution for the MCC problem in the cases where the processors are homogenous and heterogeneous. In the case where the processors are homogenous, a module execution cost is the same on all processors. In the case of heterogeneous processors, each module has a different execution cost on each of the processors. Links connecting the processors are considered to be homogeneous in both cases.

7.1. Homogenous chain of processors

The first algorithm works on a chain system where the processors are homogenous and the communication links between two adjacent processors are also homogenous. Also, the communication load between any two modules must not exceed the sum of the weights of all modules divided by the number of processors, i.e., the average load per processor. The weights of the m modules is represented with a one-dimensional array of size m . Each element i of the array stores the execution cost of the corresponding module and the communication cost with module $i + 1$. The algorithm works as follows:

1. Let $i = 1$.
2. Let AVG be the sum of all module weights divided by the number of processors, i.e. the average load per processor.
3. Traverse a list representing the chain of m modules in the first-to-last node direction. At each node j compute S_j as the sum of module weights from node i to node j until S_j exceeds AVG .

4. If S_j is nearer to AVG than S_{j-1} , i.e., $(S_j - AVG \leq AVG - S_{j-1})$, then assign modules from i to j to the first available processor in the chain and set $i = j + 1$; otherwise, assign modules i to $j - 1$ to the first available processor and set $i = j$.
5. Steps 3 and 4 are repeated until all modules are assigned.

We note that when the number of processors is much smaller than that of the modules, steps 3 and 4 are repeated until $m - 1$ processors are assigned and the remaining modules are assigned to the last processor m . But in the case where this number is greater than the number needed in the assignment, the last subchain of modules, which weights sum does not exceed AVG , will be assigned to the first available processor leaving a number of unused processor at the end of the processor chain.

This algorithm traverses the chain of m modules twice. In the first time, it computes the average load per processor, and in the second time, it provides for a heuristic assignment based on the average calculated. Since both steps are $O(m)$ time, the total run time complexity of the algorithm is $O(m)$ time.

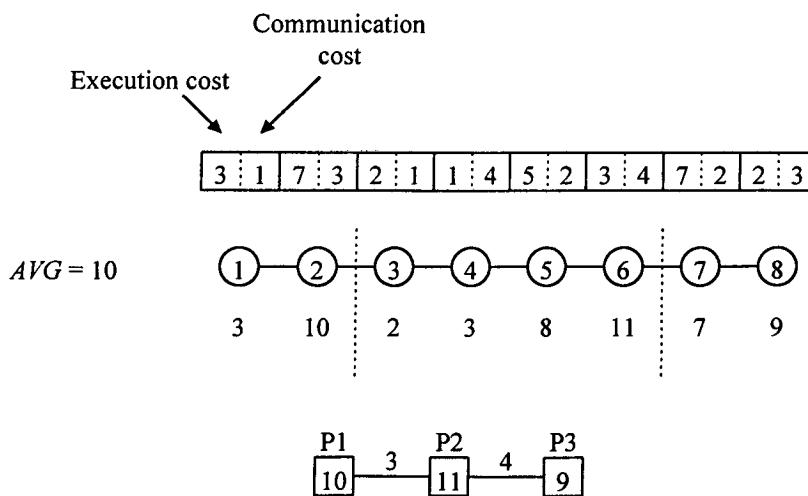


Fig. 7.1 A chain of 8 modules onto a chain of 3 homogenous processors.

In Fig. 7.1, a chain of eight modules is assigned to a chain of three homogenous processors. The sum of module weights starting from module 1 exceeds AVG , which is equal to 10, at node 3. Since S_2 is nearer to AVG than S_3 , modules 1 and 2 are assigned to processor P1 and $i = 3$. This time the sum of module weights starting from module 3 exceeds AVG at node 6, and S_6 is nearer to AVG than S_5 , thus modules 3 to 6 are assigned to P2 and $i = 7$. Since only one processor is not assigned, the remaining subchain of modules 7 and 8 are assigned to P3. This is one of the cases where the heuristic solution is an optimal solution with bottleneck weight equal to 11.

7.2. Heterogeneous chain of processors

For the case where the processors are heterogeneous, each module may have a different execution cost on any of the processors. The weights of the m modules are represented with a two-dimensional array of size $m \times n$. Each element i, j of the array stores the execution cost of module i on processor j . The communication costs between two modules are stored in a one-dimensional array of size m where the content of element i represents the communication of module i with module $i + 1$.

To deal with this issue, we modify step 2 of the previous algorithm by computing AVG as follows:

2. Let AVG be the sum of all module costs on all processors divided by the square of the number of processors.

We also modify step 3 since calculating the weight of the subchain i to j depends on the processor to which it will be assigned. So step 3 becomes:

3. Traverse a list representing the chain of m modules in the first-to-last node direction. At each node j compute S_j as the sum of module weights from nodes i to j on the first available processor until S_j exceeds AVG .

Calculating AVG in this case is $O(mn)$ time which is the time to read all $m \times n$ module execution costs per processor. Thus, the run time complexity of the algorithm is $O(mn)$.

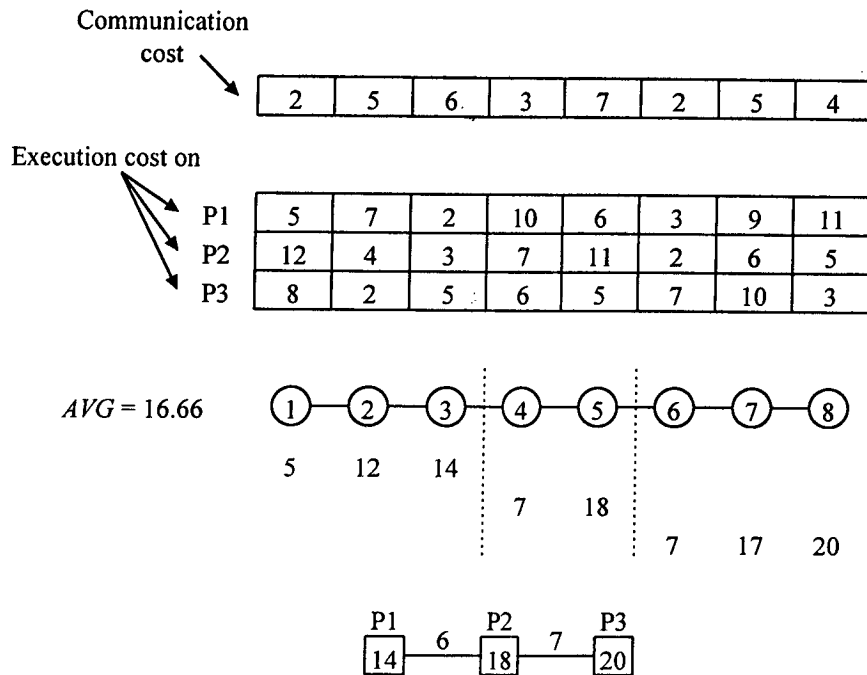


Fig. 7.2 A chain of 8 modules onto a chain of 3 heterogeneous processors.

In Fig. 7.2, a chain of eight modules is assigned to a chain of three heterogeneous processors. The sum of module execution costs on P1 starting module 1 exceeds AVG , which is equal to 16.66, at node 4. Since S_3 is nearer to AVG than S_4 , modules 1, 2 and 3 are assigned to processor P1 and $i = 4$. Since the second assignment corresponds to processor P2, the sum of execution cost will be computed with respect to P2. AVG is exceeded at node 5, and modules 4 and 5 are assigned to P2 because S_5 is nearer to AVG than S_4 . The remaining subchain of modules 6, 7 and 8 are assigned to P3. In this case the heuristic algorithm provides an optimal solution with bottleneck weight equal to 20. Another optimal solution to the same problem assigns modules 1, 2 and 3 to P1, modules 4, 5 and 6 to P2, and modules 7 and 8 to P3.

7.3. Unbounded communication costs

As mentioned above, both algorithms suggest that the communication between any two modules is less than AVG , because if it happens that two modules have the communication cost between them greater than AVG and the first is assigned to the processor i and the second to the processor $i + 1$, then the communication cost between them may dominate the size of the load on the heavily loaded processor. To avoid such situation, all the communication costs between two modules are inspected before step 3, and two nodes are merged if this cost exceeds AVG . Merging two nodes i and $i + 1$ involves adding the weights of module $i + 1$ to those of node i , and removing the merged node from the list representing the chain. Step 3 will work on the newly generated chain which is smaller than the original one. Inspection of the communication links takes $O(m)$ time which does not increase the run time complexity of any of the previous algorithms.

CHAPTER VIII

OPTIMAL TASK ASSIGNMENT IN HOMOGENOUS NETWORKS IN THE PRESENCE OF ATTACHED TASKS

The main idea behind Lee and Shin algorithm [24] which provides an optimal solution to the problem of assigning an arbitrary problem to an n -dimensional array or a set of homogenous processors in the presence of attached tasks, is first to find a cutset that separates a designated processor from all other processors on an appropriately defined network flow graph. Such cutset represents the communication costs occurring on the links connecting the isolated processor and the task assigned to it to the tasks assigned to the remaining processors. Then, it continues by finding another cutset that separates a group of processors, formed by the designated processor and one of its nearest neighbors from all other processors. This process is repeated for all neighbors of the designated processor by adding one processor at a time to the group. Next, for each processor in the group, its nearest neighbors are added one by one and the corresponding cutset is derived separating the group from all other remaining processors. All generated cutsets represent the sum of communication costs incurred by a specific assignment of tasks to processors. An assignment is considered feasible if all attached tasks are properly assigned.

Lee and Shin proved that for n -dimensional array and tree interconnection structures, the choice of the cutsets must be done in a way that they do not cross each other and that all feasible solutions to the problem one-to-one correspond to the set of these cutsets. In the remaining part of this chapter, we will show how to adapt Lee and Shin idea on a homogenous system having a star graph interconnection structure as a representative of group graphs, and then we generalize our results to homogenous systems with arbitrary interconnection structure.

8.1. Group graphs

Akers and Krishnamurthy [1] describe group graphs as follows:

“a group graph results when we select a set of symbols (say A,B,C) and a set of rules (transformations) by which one permutation of the symbols may be changed into another.... The corresponding graph results when we assign one node to each of the resulting permutations and then connect two nodes by a branch if a transformation takes one corresponding node into the other.”

Definition 54: If we choose a set of n symbols and one specific permutation which we call I (the identity element) and a set of transformations which may be applied to I , then the resulting set of permutations is called a group. A group graph results when we assign one node to each permutation in the group and then draw a branch from node P_1 to node P_2 , if there exist a transformation which takes permutation P_1 into permutation P_2 [1].

The set of transformations is further restricted such that (i) it does not generate the identity, (ii) all are different and (iii) all are closed under inverse. These restrictions imply that group graphs are loop-free, do not have multiple edges, and can be represented by undirected graphs. An example [1] of such permutations rules on 4 symbols ABCD that generate the group graph of Fig. 8.1 are:

1. Swap the inner symbols (acbd),
2. Swap the outer pairs of symbols (badc), and
3. reverse the permutation (dcba).

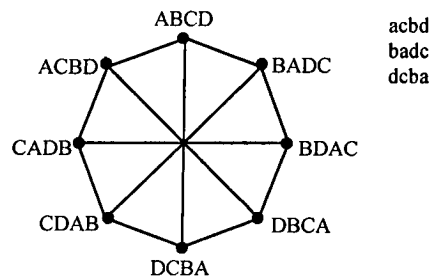


Fig. 8.1 An example group graphs [1].

We note that group graphs are node symmetric, i.e.

from any node in the graph “the rest of the graph

looks exactly the same” [1]. The proof of group graphs node symmetry property can be found in [2].

Another example of group graphs is the star graph.

Definition 55: A star graph S_n , of order n , is defined to be a symmetric graph $G = (V, E)$

where V is the set of $n!$ vertices, each representing a distinct permutation of n elements and E

is the set of symmetric edges such that two permutations (nodes) are connected by an edge iff one can be reached from the other by interchanging its first symbol with any other symbol [33].

For example, in S_3 we take the identity to be ABC and suppose that it is the starting node in the graph. According to the definition, ABC connects to BAC and CBA (Fig. 8.2(a)). We can also use a different set of symbols, for example in S_4 , the identity 1234 can be transformed into 2134, 3214, and 4231. By applying the definition on the generated permutations, we can build the 24 node graph shown in Fig. 8.2(b).

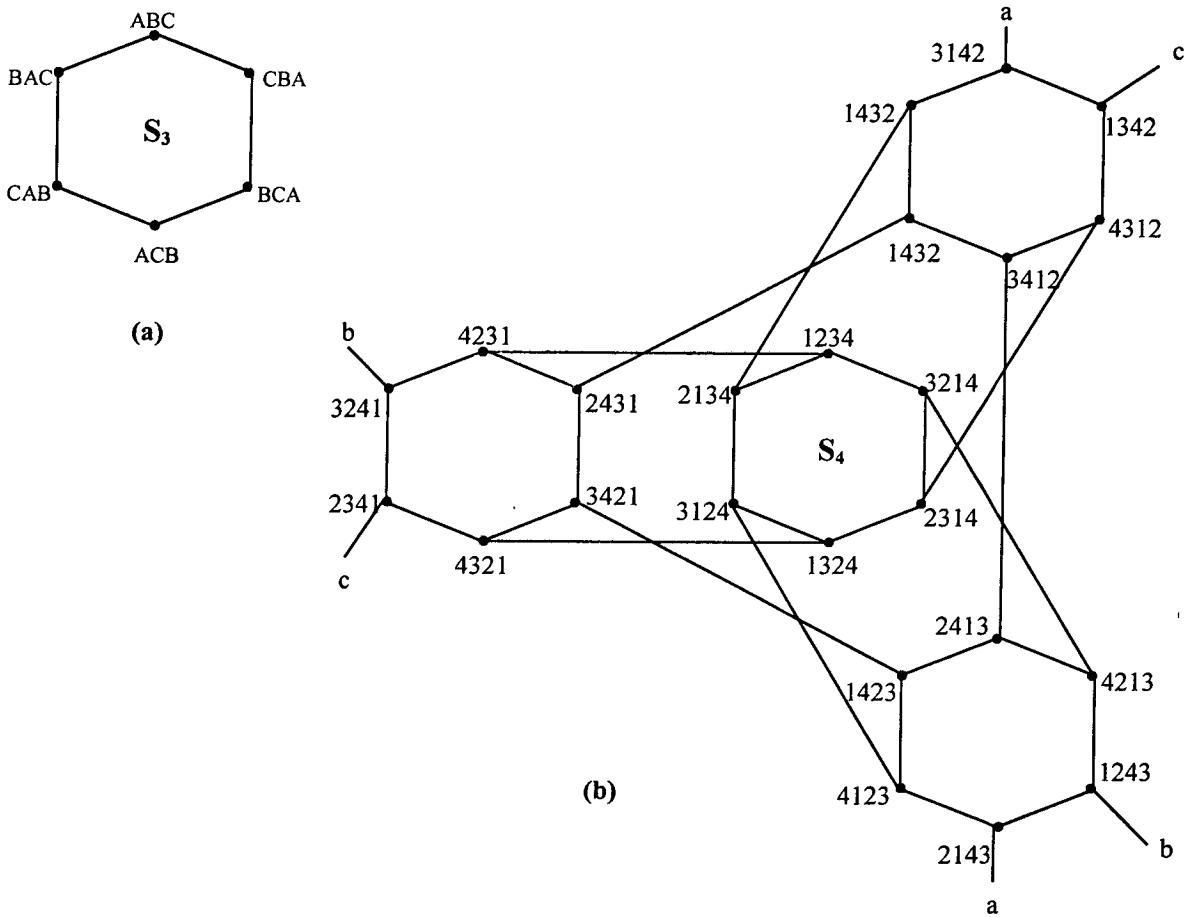


Fig. 8.2 Star graphs S_3 and S_4 .

8.2. Optimal task assignment in star graph networks in the presence of attached tasks

To apply Lee and Shin idea on a homogenous system having a star graph interconnection structure, we need to construct a network-flow graph that represents this assignment problem and find all the set of cutsets, that represent all feasible solutions, such that the total weight of each set of cutsets is equal to the communication cost incurred by its corresponding assignment.

Definition 56: Given a star graph S_n and an identity permutation I , the neighborhood tree is the tree that generates all the neighbors of the identity I and the neighbors of the neighbors such that each of the $n!$ nodes of S_n is represented only once in the tree. Furthermore, nodes of the neighborhood tree are numbered from 1 to $n!$ in breadth first search order.

The neighborhood tree for S_4 is shown in Fig. 8.3. In the remaining part of this chapter, we will denote by N the total number of processors in the network which is equal to $n!$ in the case of star graph.

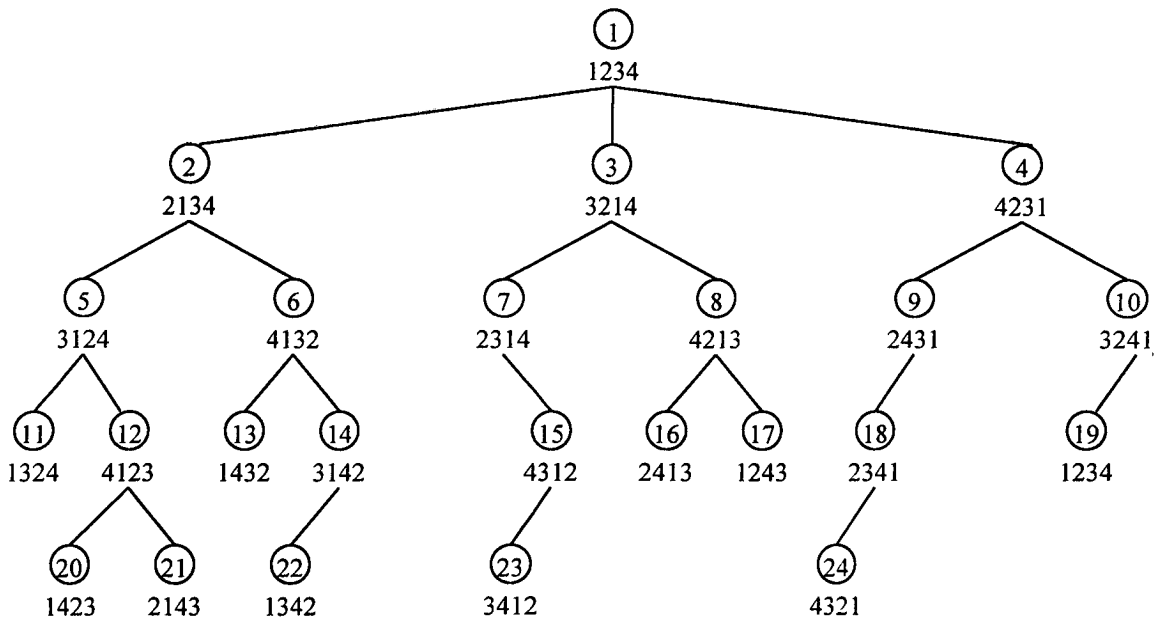


Fig. 8.3 The neighborhood tree for S_4

By traversing the neighborhood tree in a breadth first order, at each node k , representing processor p_k , we can find the sets P_k and \bar{P}_k such that $P_k = \{p_1, \dots, p_k\}$ and \bar{P}_k all the

remaining processors. We then proceed using Lee and Shin approach, so we apply their modeling technique to describe the solution of the problem and we follow their steps to prove its correctness.

Given a $TIG = (V, E)$ of a distributed application of m modules submitted to an N processors star graph, we first make a corresponding N -terminal network $G_N = (V_N, E_N)$ by adding the N processors nodes to the TIG . Then, we construct $(N - 1)$ two-terminal networks graph G_i s, $1 \leq i \leq N$, from the N -terminal network G_N as follows:

1. Generate a source node S_i by combining all the processors nodes in P_i and all the task nodes which are attached to one of these processors.
2. Generate a sink node T_i by combining all the processor node in \bar{P}_i and all the tasks nodes which are attached to one of these processors.

Definition 57: Let C_S be a set of $(N - 1)$ cutsets C_i each of which is on the corresponding two-terminal network graph G_i . Then C_S is said to be admissible if no two cutsets in C_S cross each other. The weight of C_S is the total weight of the cutsets in C_S , i.e.

$$w(C_S) = \sum_i w(C_i).$$

Lemma 1: Each admissible set C_S one-to-one corresponds to a feasible task assignment.

Proof. Each admissible set C_S of a graph G_N partitions the graph into N subsets A_k s each of which has exactly one processor node p_k . Then we can associate C_S with the assignment that every task in A_k is assigned to p_k , and vice versa.

Lemma 2: Let an admissible set C_S corresponds to a feasible assignment X . Then, the weight of C_S is equal to the total communication cost of X , $COMM(X)$.

Proof. The communication costs incurred between two parts of the star graph, nodes 1 to i from one side and nodes $i + 1$ to N from the other side with respect to the breadth first order

enumeration, under the assignment X is equal to the weight of the corresponding cutset C_i .

Thus, the total communication cost is the sum of all the cutsets C_i , $1 \leq i \leq N$, i.e.

$$COMM(X) = \sum_i W(C_i) = W(C_s).$$

From lemmas 1 and 2, we can deduce that, also in the case of a parallel system of homogenous processors having a star graph interconnection structure, the task assignment problem in the presence of attached tasks can be solved by finding the minimum weight admissible set C_{A_0} on the corresponding N -terminal network graph. We propose the following procedure as a solution to this assignment problem.

1. Generate the neighborhood tree of the star graph starting from an identity node I , and label this tree in a breadth first search order.
2. Traverse the tree in breadth first search order until node $N - 1$ is reached.
 - i) At each node labeled i in the neighborhood tree, group all nodes from 1 to i in a source node S along with all attached task to any of the processors 1 to i , and all the remaining nodes $i + 1$ to N in a sink node T along with their attached tasks.
 - ii) Find a minimum weight cutset C_i of the two-terminal network graph $G_i = (V_i, E_i)$ between source S and sink T , which divide the network into two parts.
 - iii) For every unassigned task to the side of S , assigns it to the processor labeled i .
3. All remaining unassigned tasks are assigned to the processor labeled N .

The neighborhood tree can be represented by an adjacency list of $O(N)$ nodes, and can be created, enumerated and traversed in a breadth first search order in $O(N)$ time. The algorithm requires $N - 1$ applications of a network flow algorithm of $O(m^3)$ time each. Thus, the overall complexity of finding the optimal solution is $O(Nm^3)$ time.

Lemma 3: Let a set $C_S = \{C_i \mid 1 \leq i < N\}$ found in the algorithm described above correspond to a task assignment X . Then the assignment X is feasible; that is, C_S is admissible.

Proof. At each node labeled i , every unassigned task to the side of the source node S is assigned to the processor labeled i at step 2.c. Any cutset C_j , $i < j$, cannot partition nodes to the side of S any more, since these nodes form the source node S when finding the cutset C_j . Therefore, any two cutsets found by this algorithm do not cross each other.

Lemma 4: Let a set $C_S = \{C_i \mid 1 \leq i < N\}$ found in the algorithm described above correspond to a feasible assignment X . Then, for any feasible assignment X' , the following inequality holds for each node labeled i : $W(C_i) < W(C'_i)$, for all i .

Proof. Each cutset C_i in C_S is a minimum-weight cutset of G_i and separates the graph into two parts. Every task to the side of source S is assigned to one of the processors p_k 's, $1 \leq k \leq i$, and every task to the side of sink node T is assigned to one of the processors p_l 's, $i + 1 \leq l \leq N$ by the above procedure. Then the weight of the cutset C_i is $W(C_i)$ and it is equal to the sum of all communication costs between tasks assigned to processors on the side of S and tasks assigned to the remaining processors on the side of T . We prove the inequality by induction on i .

1. The result is true for $i = 1$, since C_1 is a minimum-weight cutsets.
2. Suppose the inequality is true for $i = k - 1$. Without loss of generalities, assume that the task nodes are partitioned into two subsets A and B by the minimum cutset C_{k-1} (Fig. 8.4(a)). Let $c(A, B)$ denote the sum of the weights of all edges between two sets A and B . Then $W(C_{k-1}) = c(A, B)$. By the procedure described above, the next cutset C_k cannot partition the tasks in A any more, since every task in A is already included into the source node S . Let C_k partition the task graph into two subsets $A \cup B_2$ and B'_2 (Fig. 8.4(b)), then

$$W(C_k) = c(A \cup B_2, B'_2) = c(A, B'_2) + c(B_2, B'_2) = c(A_1, B'_2) + c(A'_2, B'_2) + c(B_2, B'_2), \quad \text{where}$$

$A_1 \cup A'_1 = A$ and $B_2 \cup B'_2 = B$. (Each of the subsets may be empty, but this will not alter

the proof.) We prove this by contradiction. Suppose the inequality does not hold for $i = k$.

Then there exists another feasible assignment X' which partitions task nodes into two subsets $A_1 \cup B_1$, and $A'_1 \cup B'_1$, where $B_1 \cup B'_1 = B$ (Fig. 8.4(c)) such that

$$c(A_1, A'_1) + c(A_1, B'_1) + c(B_1, A'_1) + c(B_1, B'_1) < c(A_1, B'_2) + c(A'_1, B'_2) + c(B_2, B'_2). \quad (1)$$

Every task in A'_1 is executable on at least one of the processors p_l , $k+1 \leq l \leq N$, since X' is feasible. Thus, the cutset C'_{k-1} (Fig. 8.4(d)) is a cutset of a feasible assignment which assigns every task in A_1 to one of the processors p_l , $1 \leq l \leq k-1$, and every task in $A'_1 \cup B_1 \cup B'_2$ to one of the processors p_l , $k \leq l \leq N$. Then $W(C_{k-1}) \leq W(C'_{k-1})$ by the assumption for $j = k-1$, i.e.,

$$c(A_1, B_1) + c(A'_1, B_1) + c(A_1, B'_1) + c(A'_1, B'_1) \leq c(A_1, A'_1) + c(A_1, B_1) + c(A_1, B'_1). \quad (2)$$

Every task in B_1 is executable on at least one of the processors p_l , $k \leq l \leq N$, since X is feasible. Also, every task in B_1 is executable on at least one of the processors p_l , $1 \leq l \leq k$, since X' is feasible. Thus, every task in B_1 is executable on processor p_l , $l = k$. The cutset C'_k in (figure 5d.) is a possible cutset in G_{ik} which assigns every task in B_1 to processor p'_k (we assumed that every task in $A_1 \cup A'_1$ has already been assigned to one of the processors p_l , $1 \leq l \leq k-1$, by the cutsets C_1, C_2, \dots, C_{k-1}). Then $W(C_k) \leq W(C'_k)$ since C_k is a minimum-weight cutset in G_k , i.e.,

$$c(A_1, B'_2) + c(A'_1, B'_2) + c(B_2, B'_2) \leq c(A_1, B'_1) + c(A'_1, B'_1) + c(B_1, B'_1). \quad (3)$$

By combining the above three inequalities (1), (2), and (3), we obtain the following inequality: $c(A'_1, B_1) < 0$. This contradicts the fact that the weight between any two subsets cannot be negative. Thus, the inequality holds for $i = k$.

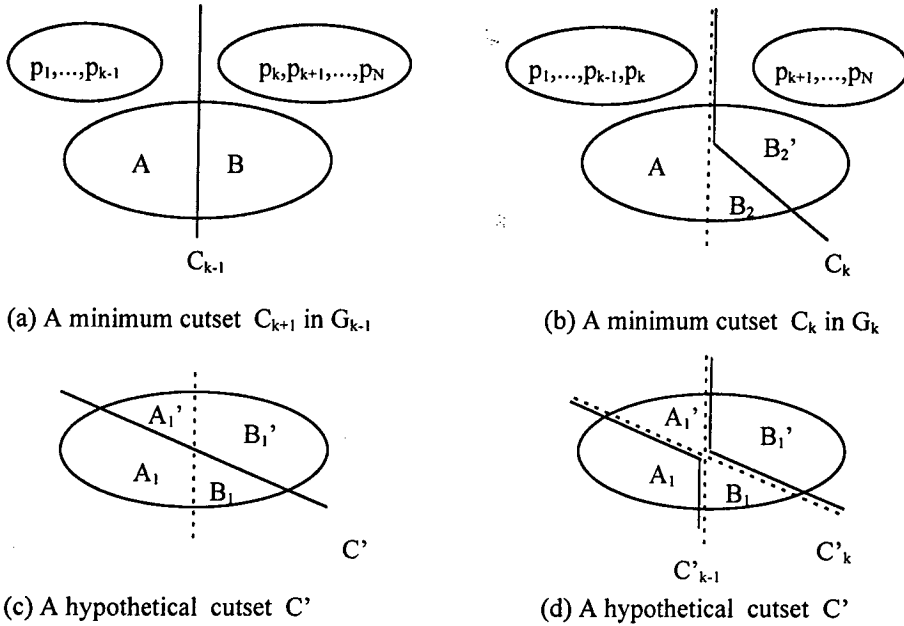


Fig. 8.4 Illustrative figures for Lemma 4.

Theorem 1: Let $C_S = \{C_i \mid 1 \leq i < N\}$ found in the algorithm described above correspond to a feasible assignment X . Then, the assignment X is an optimal assignment with the total communication cost of $\sum_i W(C_i)$.

Proof. By contradiction, assume that X is not an optimal assignment. Let another feasible assignment X' be a optimal assignment, i.e., $COMM(X') < COMM(X)$. Then there exists at least one i , $1 \leq i \leq N - 1$, such that $W(C'_i) < W(C_i)$. This is contradictory to the result of Lemma 4, and thus, X is an optimal assignment. The total communication cost of X is $\sum_i W(C_i)$ by Lemma 2.

Fig. 8.5 shows an example that we have traced below as a sample run of our algorithm. A parallel program of seven modules is to be assigned onto S_3 which neighborhood tree is also shown in Fig. 8.5. In the first iteration (Fig. 8.6), node 1

representing processor 123 is selected to form the source node S along with the task T1 which is attached to 123. All the remaining processors and their attached tasks are grouped into the sink node T . The cutset $C_1 = 25$ is a result of applying step 2.b. This cut does not assign new tasks to S , and its value is equal to the communication costs of T1 on 123 and its adjacent tasks on the remaining processors, i.e. the cost of assigning attached task T1 to 123.

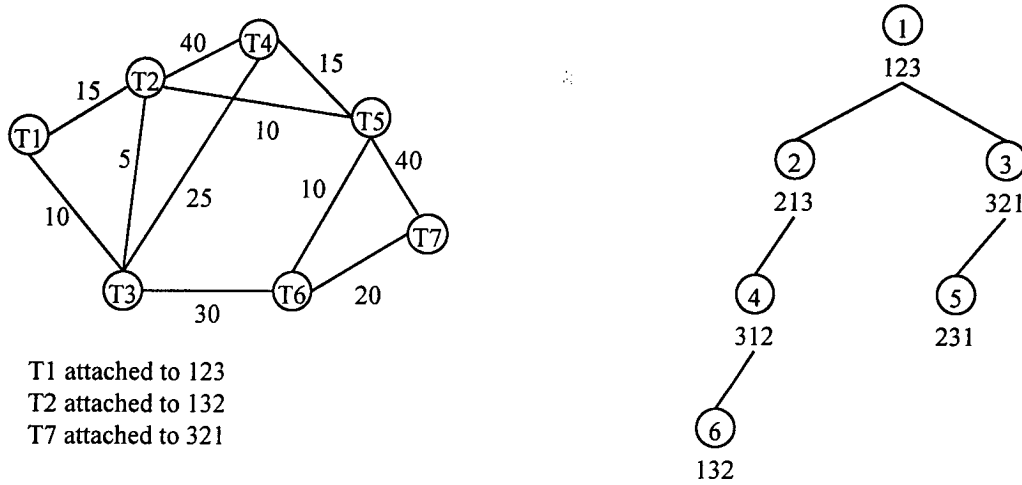


Fig. 8.5 An example of TIG of 7 modules to be assigned to S_3 .

In the second iteration (Fig. 8.7), where processors 123 and 213 are group into S with their attached tasks, cutset C_2 does not assign to S any new tasks, i.e. processor 213 will not execute any task. Cutset $C_2 = 25$ and its value will not be considered in the computing the total communication cost since it does not represent an assignment of tasks to 213. In the third iteration (Fig. 8.8), processors 123, 213 and 321 are grouped with tasks T1 and T7 to into source S , and all the remaining processors with their attached tasks into sink T . The cutset $C_3 = 70$, and it assigns tasks T3, T5 and T6 to processor 321. The fourth (Fig. 8.9) and the fifth (Fig. 8.10) iterations does not assign any tasks to 312 and 231 respectively. The remaining task T4 will be assigned to 132 in sink node T with a cost $C_5 = 70$ since C_5 is the last cutset. C_4 value will not be used in computing the total communication costs because it does not represent any assignment. Thus, the cost of the assignment is

$C_S = C_1 + C_3 + C_5 = 165$. Fig. 8.11 shows an optimal assignment of the problem in Fig. 8.5 with details on the communication costs between any two tasks. We note that this problem admits another optimal solution by assigning task T4 to processor 321, which can be verified in Fig. 8.8 by computing the cutset assigning all tasks to S which is also equal to 70. Moreover, in Fig. 8.11 the communication between task T1 on 123 and task T4 on 132 may take place through processors 213 and 312 with the same cost.

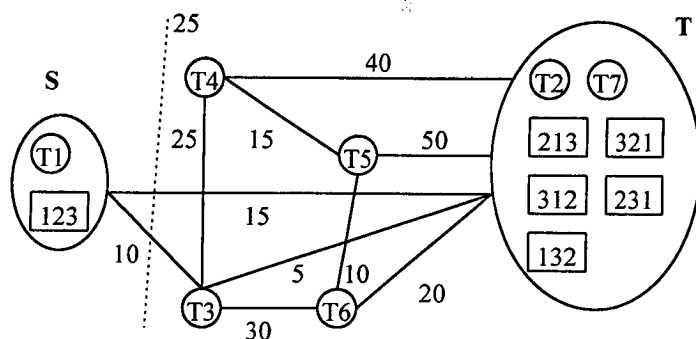


Fig. 8.6 Iteration 1 of the algorithm on the problem of Fig. 8.5.

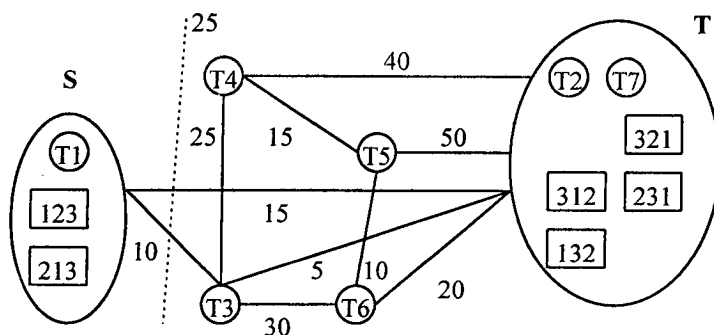


Fig. 8.7 Iteration 2 of the algorithm on the problem of Fig. 8.5.

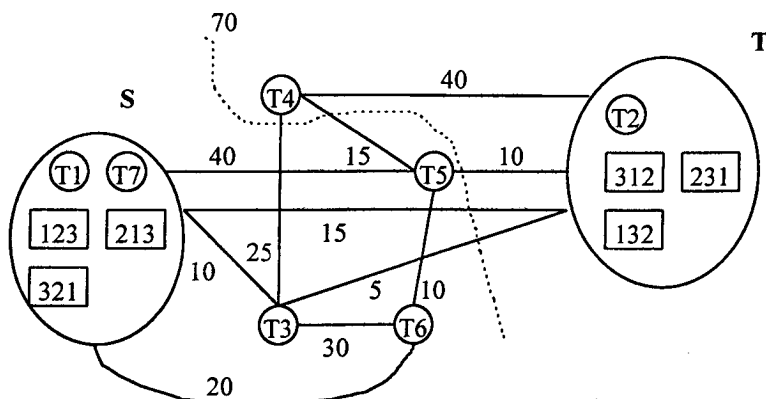


Fig. 8.8 Iteration 3 of the algorithm on the problem of Fig. 8.5.

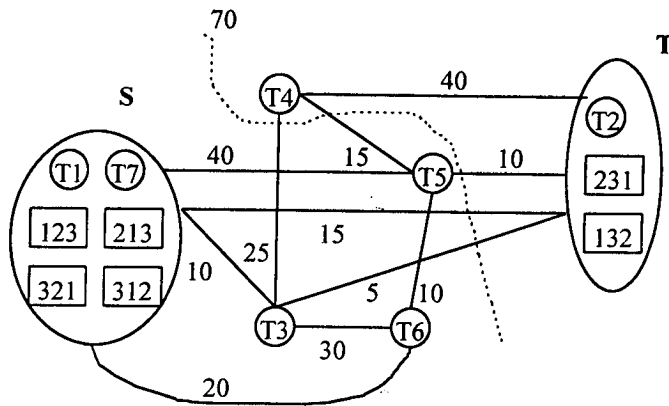


Fig. 8.9 Iteration 4 of the algorithm on the problem of Fig. 8.5.

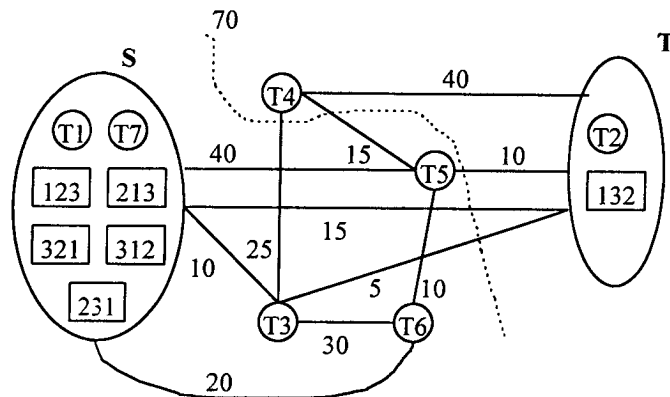


Fig. 8.10 Iteration 5 of the algorithm on the problem of Fig. 8.5.

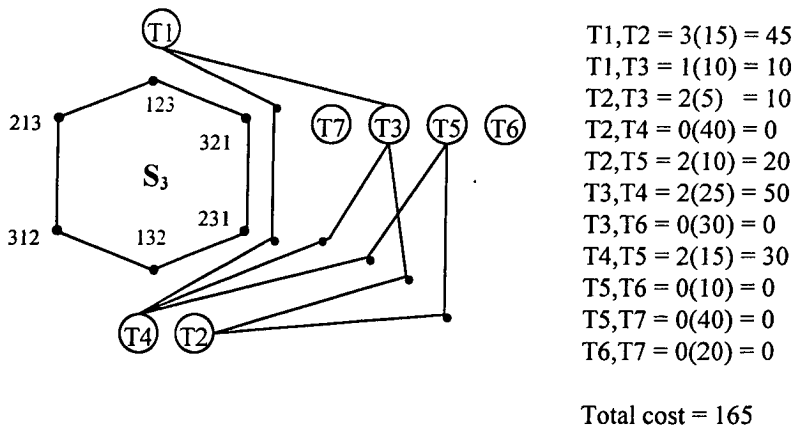


Fig. 8.11. An optimal task assignment of the problem of Fig. 8.5.

8.3. Optimal task assignment in homogenous arbitrary networks in the presence of attached tasks

From the definition of the neighborhood tree, we realize that for any group graph this tree can be generated. Also, this tree can be generated for any interconnection structure that can be described by symbol permutations. Since, the neighborhood tree is nothing but a breadth first order traversal of the interconnection structure of the homogenous network starting from a specific node, which can be verified in Fig. 8.3 for the star graph S_4 , we can generalize our results on star graphs as follows. For a network of homogenous processors connected in an arbitrary form, it is sufficient to create the breadth first search tree for the network and use this tree for creating all the P_k and \bar{P}_k sets of nodes at each k th node of the arbitrary graph representing a processor. Then, we can apply the steps 2 and 3 of our algorithm to find the optimal solution. Creating the breadth first search tree for an arbitrary graph of n nodes is $O(n^2)$ time. Thus, the overall complexity of assigning an arbitrary structured program of m tasks to an arbitrary network of N homogenous processors is $O(Nm^3)$.

CHAPTER IX

A COMPARISON OF ASSIGNMENT AND SCHEDULING ALGORITHMS

In this chapter, we include a table of comparison for the fifty algorithms reviewed from the literature (Table 9.1). In the first column of the table, the problem is briefly defined, and in the second column its type is indicated be it an allocation or a scheduling problem. The third column shows whether communication costs are considered or not in the solution of the problem and indicates whether these costs are homogenous or heterogeneous among all tasks. The fourth column shows the type of costs the algorithm is aiming to minimize. The fifth column indicates the machine structure to which tasks will be allocated or scheduled by the algorithm and the seventh column indicates the number of processors of that machine. The eighth column briefly describes the techniques used in solving the problem and the ninth column indicates whether the solution is optimal or not. The tenth and eleventh columns show the space complexity of the problem in terms of nodes and edges respectively, and the twelfth column indicates the runtime complexity of the algorithm. Column thirteen is reserved for explanatory remarks.

Problem	T	Problem Structure	Co	Cost to Minimize	Machine Structure	No PE	Algorithm / Technique	Op	No. of Nodes	No. of Edges	Run Time Complexity	Remarks
Dual processor assignment [4]	A	Undirected arbitrary	Ho	Σ execution & interprocessor communication	-	2	Maxflow Mincut	✓	$O(m)$	$O(m^2)$	$O(m^3)$	Works also on a serial program.
Dual processor assignment / one with limited memory [14]	A	Undirected arbitrary	Ho	Σ execution & interprocessor communication	-	2	Multiterminal Maxflow, G-H tree, inclusive cut graph & heuristic enumeration	-	$O(m)$	$O(m^2)$	$O(m^3)$	NP-Hard. Works also on a serial program.
Dual processor assignment with dynamic relocation [4]	A	Undirected arbitrary	Ho	Σ execution, interprocessor communication & relocation	-	2	Maxflow Mincut	✓	$O(m)$	$O(m^2)$	$O(m^3)$	Fine grain detailed behavior of the problem is needed. Works also on a serial program.
Assigning trees across space [4]	A	Out-tree	He	Σ execution & intermodule communication	Heterogeneous fully connected	n	Labeling layered graph, Shortest tree	✓	$O(mn)$	$O(mn^2)$	$O(mn^3)$	Dijkstra's Shortest Path Algorithm takes $O(m^2n^2)$.
Mapping a problem graph to FEM $(nxn), N=n^2$ [5]	A	Parallel structured	Ho	Σ interprocessor communication	FEM $N=n^2$	N	Pairwise interchange with probabilistic jumps	-	$O(N)$	$O(N)$	$O(N^3)$	Equivalent to the graph isomorphism problem.
Task allocation model for distributed computer systems based on 0-1 programming [26]	A	Undirected arbitrary	He	Interprocessor communication & load balancing subject to engineering application requirements	-	n	Derivation of Branch and Bound with elimination rules	✓	-	-	$O(n^3)$	In practice, the use of elimination rules in this specific model reduces the runtime complexity to a polynomial factor.
Allocating programs containing branches and loops & having a series-parallel structure [34]	A	Directed series-parallel	He	Σ execution & intermodule communication	Heterogeneous fully connected	n	Shortest path	✓	$O(mn)$	$O(mn^2)$	$O(mn^3)$	
General task assignment problem [25]	A	Undirected arbitrary	Ho	Σ execution & interprocessor communication	Heterogeneous fully connected	n	Network flow algorithm & heuristic	-	$O(m+n)$	$O(m+n)$	$O(mn^2 \log m)$	NP-Complete. Uses $O(m \log m)$ network flow algorithm [15].
General task assignment problem with interference costs [25]	A	Undirected arbitrary	Ho	Σ execution, interprocessor communication & interference	Heterogeneous fully connected	n	Network flow algorithm & heuristic	-	$O(m+n)$	$O(m+n)$	$O(mn^2 \log m)$	NP-Complete. Storing interference table is $O(m^2n)$. Uses $O(m \log m)$ network flow algorithm [15].
Assigning parallel arbitrary programs to a distributed system with upper & lower bounds [8]	A	Undirected arbitrary	Ho	Σ interprocessor communication with load balancing	Heterogeneous fully connected	n	Clustering & Allocation heuristics	-	$O(m)$	$O(m^2)$	$O((3e+(d+l)m) \log m)$	For small fixed and sparse graph runtime is $O(m \log m)$.
Mapping chains onto heterogeneous chains [4, 6, 7]	A	Chain	He	Load on the heavily loaded processor (bottleneck)	Heterogeneous chain	n	Labeling layered graph to find minimum bottleneck	✓	$O(m^2n)$	$O(m^2n)$	$O(m^2n)$	n Contiguous subchains. A variation of Dijkstra's Shortest Path Algorithm [11] takes $O(m^2n^2)$.
Partitioning multiple chains across a heterogeneous host-satellite system [4, 6, 7]	A	Multiple pipelined chains	He	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellites	n	Optimal SB Path algorithm on doubly weighted layered graph	✓	$O(mn)$	$O(mn)$	$O(m^2n \log m)$	2 contiguous subchains of each chain. Heterogeneous links between host and each satellite.
Mapping a chain onto a heterogeneous shared memory machine [7]	A	Chain	He	Σ of all communicating edges exposed by the mapping & the bottleneck load	Heterogeneous shared memory machine	n	Optimal SB Path algorithm on doubly weighted layered graph	-	$O(m^2n)$	$O(m^2n)$	$O(m^2n \log m)$	
Partitioning arbitrary programs in a heterogeneous host-satellite system [4, 6, 7]	A	Undirected arbitrary	He	Σ execution & interprocessor communication	Host-heterogeneous satellites	n	Transform to chain using network flow, Optimal SB Path algorithm on doubly weighted layered graph	✓	$O(mn)$	$O(m^2n)$	$O(m^2n)$	Heterogeneous links between host and each satellite.
Partitioning parallel or pipelined trees in a homogeneous host-satellite system [4, 6, 7]	A	Parallel or pipelined tree	Ho	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellites	n	Directed dual graph, multigraph, Optimal SB-Path algorithm	✓	$O(m)$	$O(m)$	$O(m^2 \log m)$	Maximal subtrees on satellites.

Problem	T	Problem Structure	Co	Cost to Minimize	Machine Structure	No PE	Algorithm / Technique	Op	No. of Nodes	No. of Edges	Run Time Complexity	Remarks
Mapping chains onto heterogeneous chains [27]	A	Chain	He	Load on the heavily loaded processor (bottleneck)	Heterogeneous chain	n	Labeling improved layered graph to find minimum bottleneck	✓	$O(m^2n)$	$O(m^2n)$	$O(m^2n)$	n contiguous subchains. A variation of Dijkstra's Shortest Path Algorithm [11] takes $O(m^2n)$.
Mapping a chain onto a heterogeneous shared memory machine [27]	A	Chain	He	Σ of all communicating edges exposed by the mapping & the bottleneck load	Heterogeneous shared memory machine	n	Optimal SB Path algorithm on doubly weighted improved layered graph	-	$O(m^2n)$	$O(m^2n)$	$O(m^2n \log m)$	
Partitioning multiple chains across a heterogeneous host-satellite system [27]	A	Multiple pipelined chains	He	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellites	n	Optimal SB Path algorithm on doubly weighted improved layered graph	✓	$O(mn)$	$O(mn)$	$O(mn \log m)$	2 contiguous subchains of each chain.
Mapping chains onto homogeneous chains with homogeneous links [20,27]	A	Chain	Ho	Load on the heavily loaded processor (bottleneck)	Homogeneous chain	n	Find minimum bottleneck using probing functions	-	$O(m+n)$	$O(m)$	$O(mn \log(W_T/\epsilon))$	n contiguous subchains. Approximation within ϵ form the optimal solution. W_T is sum of all module weights.
Mapping chains onto homogeneous chains with homogeneous links with $0 < W < w_i$ & $c_{ij} < C$ [27]	A	Chain	Ho	Load on the heavily loaded processor (bottleneck)	Homogeneous chain	n	Minimum bottleneck using probing functions	✓	$O(m)$	$O(m)$	$O(mn \log m)$	n contiguous subchains. Module weight w_i bounded below by W & communication cost between two modules bounded above C .
Mapping a chain onto a shared memory machine with $0 < W < w_i$ & $c_{ij} < C$ [27]	A	Chain	Ho	Σ of all communicating edges exposed by the mapping & the bottleneck load	Homogeneous shared memory machines	n	Minimum sum-bottleneck using probing functions	-	$O(m+n)$	$O(m)$	$O(m^2 \log(W_T/\epsilon))$	Approximation within ϵ from the optimal solution. W_T is sum of all module weights.
Partitioning multiple chains across a homogeneous host-satellite system [20,27]	A	Multiple pipelined chains	Ho	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellites	n	Minimum sum-bottleneck using probing functions	-	$O(mn)$	$O(mn)$	$O(mn \log(W_T/\epsilon))$	Space complexity is $O(m^2)$.
Partitioning multiple chains across a homogeneous host-satellite system with $0 < W < w_i$ & $c_{ij} < C$ [27]	A	Multiple pipelined chains	Ho	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellites	n	Minimum sum-bottleneck using probing functions	✓	$O(mn)$	$O(mn)$	$O(mn \log m)$	2 contiguous subchains of each chain. Module weight w_i bounded below by W & communication cost between two modules bounded above C .
Partitioning arbitrary programs in a heterogeneous host-satellite system [20]	A	Undirected arbitrary	Ho	Σ execution & interprocessor communication	Host-heterogeneous satellites	n	Transform to chain using network flow. Probe function on doubly weighted layered graph	-	$O(mn)$	$O(m^2n)$	$O(m^2n)$	The probe function is $O(mn \log(W_T/\epsilon))$ time. Homogeneous links between host and each satellite.
Partitioning parallel or pipelined trees in a homogeneous host-satellite system [20]	A	Parallel or pipelined tree	Ho	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellites	n	Minimum sum-bottleneck using probing functions	-	$O(m)$	$O(m)$	$O(mn \log(W_T/\epsilon))$	Maximal subtrees on satellites. Approximation within ϵ form the optimal solution. W_T is sum of all module weights.
Mapping chains onto homogeneous chains with homogeneous links [21]	A	Chain	Ho	Load on the heavily loaded processor (bottleneck)	Homogeneous chain & links	n	Find minimum bottleneck using probing functions	✓	$O(m)$	$O(m)$	$O(mn \log m)$	Contiguous subchains.
Partitioning parallel or pipelined trees in a homogeneous host satellite system [21]	A	Parallel or pipelined tree	Ho	The largest of the load on the host (sum) and the worst load on any of the satellites (bottleneck)	Host-heterogeneous satellite	n	Find minimum sum-bottleneck using probing functions	✓	$O(m)$	$O(m)$	$O(m \log m)$	Maximal subtrees on satellites.

Problem	T	Problem Structure	Co	Cost to Minimize	Machine Structure	No PE	Algorithm / Technique	Op	No. of Nodes	No. of Edges	Run Time Complexity	Remarks
Assignment of tasks to an n-dimensional array network $N=n_1 \times n_2 \times \dots \times n_n$ [24]	A	Undirected arbitrary	Ho	Σ execution & interprocessor communication in presence of attached tasks	Homogenous n-dimensional	N	Maxflow Minicut	\checkmark	$O(m+n)$	$O(m^2)$	$O(\sum_{i=1}^n (n_i - 1)m^i)$	For n-dimensional hypercube, $N=2^n$, the runtime complexity is $O(nm^2)$
Assignment of tasks to a tree network [24]	A	Undirected arbitrary	Ho	Σ execution & interprocessor communication in presence of attached tasks	Homogenous tree	n	Maxflow Minicut	\checkmark	$O(mn)$	$O(m^2)$	$O(nm^2)$	
Allocating task interaction graphs in homogenous networks [19]	A	Undirected arbitrary	Ho	Overall job completion time, intermodule communication & number of used processors	Homogenous virtual clique	-	Merging nodes using network flow algorithm on a preprocessed task interaction graph	\checkmark	$O(m)$	$O(m+e)$	$O(m^2(m+e)\log \frac{m^2}{m+e})$	Uses $O(m\log \frac{m^2}{e})$ network flow algorithm [18, 19]
Allocating task interaction graphs in homogenous networks [19]	A	Undirected arbitrary	Ho	Overall job completion time, intermodule communication & number of used processors	Homogenous virtual clique	-	Preprocessed task interaction graph, task merging heuristic	-	$O(m)$	$O(e)$	$O((m+e)\log^2 m)$	
Allocating task interaction graphs in heterogeneous networks [19]	A	Undirected arbitrary	Ho	Overall job completion time, intermodule communication & number of used processors	Heterogeneous virtual clique	-	Preprocessed task interaction graph, task merging heuristic	-	$O(m)$	$O(e)$	$O((m+e)m\log m)$	
Assigning trees across space & time [4]	S	Out-tree	He	Σ execution, intermodule communication & penalties for not meeting deadlines	Heterogeneous fully connected	n	Shortest tree	\checkmark	$O(mn^2)$	$O(mn^2\phi^2)$	$O(mn^2\phi^2)$	Dijkstra's Shortest Path Algorithm [11] takes $O(m^2n^2\phi^2)$.
General task scheduling problem [22]	S	Directed arbitrary	-	Σ execution or schedule length	Homogenous fully connected	n	CP/MISF heuristic	-	$O(m)$	$O(m^2)$	$O(m^2)$	NP-Hard. Worst case schedule at most equals twice the optimal schedule.
General task scheduling problem [22]	S	Directed arbitrary	-	Σ execution or schedule length	Homogenous fully connected	n	DF/IHS heuristic	-	$O(mn)$	$O(mn)$	$O(n^2m)$	NP-Hard. Worst case schedule at most equals twice the optimal schedule.
General task scheduling problem [31]	S	DAG, no redundant transitive edges	-	Σ execution or schedule length	Homogenous fully connected	n	HNF heuristic	-	$O(m)$	$O(m^2)$	$O(m\log m)$	NP-Hard. Worst case schedule at most equals twice the optimal schedule.
General task scheduling problem [31]	S	DAG, no redundant transitive edges	-	Σ execution or schedule length	Homogenous fully connected	n	CPM heuristic	-	$O(m)$	$O(m^2)$	$O(m^2)$	NP-Hard. Worst case schedule at most equals twice the optimal schedule.
General task scheduling problem [31]	S	DAG, no redundant transitive edges	-	Σ execution or schedule length	Homogenous fully connected	n	WL heuristic	-	$O(m)$	$O(m^2)$	$O(m^2)$	NP-Hard. Worst case schedule at most equals twice the optimal schedule.
General Task scheduling problem [16]	S	DAG	Ho	Linear clustering with minimum parallel time	Homogenous virtual clique	-	KB/L clustering heuristic using edge zeroing	-	$O(m)$	$O(m^2)$	$O(e(m+e))$	NP-Hard.
General Task scheduling problem [16]	S	DAG	Ho	Communication volume without increasing parallel time	Homogenous virtual clique	-	edge zeroing	-	$O(m)$	$O(m^2)$	$O(e(m+e))$	NP-Hard.
General Task scheduling problem [16]	S	DAG	Ho	Length of the Dominant Sequence DS	Homogenous virtual clique	-	DSC clustering heuristic using edge zeroing	-	$O(m)$	$O(m^2)$	$O((e+m)\log m)$	NP-Hard. For linear clustering runtime complexity reduces to $O(m\log m+e)$.

Problem	T	Problem Structure	C ₀	Cost to Minimize	Machine Structure	No PE	Algorithm / Technique	Op	No. of Nodes	No. of Edges	Run Time Complexity	Remarks
General Task scheduling problem [16]	S	DAG	Ho	Starting time of the output task	Homogenous virtual clique	-	MCP clustering heuristic using edge zeroing	-	$O(m)$	$O(m^2)$	$O(m^2 \log m)$	NP-Hard.
Scheduling in and out forest like programs [14]	S	In-forest or Out-forest	-	Σ execution or schedule length	Homogenous fully connected	n	Level algorithm. Node priority = level; priority list scheduling	✓	$O(m)$	$O(m)$	$O(m)$	Later, this algorithm is generalized to the Critical Path Method algorithm (CPM)
Scheduling interval-ordered tasks [14]	S	Interval-order	-	Σ execution or schedule length	Homogenous fully connected	n	Node priority = # of successors. priority list scheduling	✓	$O(m)$	$O(e)$	$O(e+m)$	
Scheduling arbitrary task graph on 2 processors with tasks having equal weights [14]	S	Directed arbitrary	-	Σ execution or schedule length	-	2	Node priority = level of successors, priority list scheduling	✓	$O(m)$	$O(m^2)$	$O(m^2)$	
Scheduling in and out forest like programs on 2 processors [14]	S	In-tree or Out-tree	Ho	Σ execution & communication	-	2	Uses CP on an augmented graph, precedence relations are added to compensate for communication	✓	$O(m)$	$O(m^2)$	$O(m^2)$	
Scheduling interval-ordered tasks having equal task and communication weights [14]	S	Interval-order	Ho	Σ execution & communication	Homogenous fully connected	n	Node priority = number of successors, priority list scheduling	✓	$O(m)$	$O(e)$	$O(me)$	
Scheduling in and out forests with task & communication weights take one time unit on a fully connected system with bounded number of processors [35]	S	In-forest or Out-forest	Ho	Σ execution & communication	Bounded homogenous fully connected	$<C$	Transform the graph into free delay graph, allocation	✓	$O(m)$	$O(m^2)$	$O(m^{n-2})$	The algorithm is designed for out forests. An in forest reduces to an out forest by reversing the edge directions and then inverting the resulting optimal schedule.
Scheduling out forests with tasks & communication weights take one time unit on a fully connected system with bounded number of processors [35]	S	Out-forest	Ho	Σ execution & communication	Bounded homogenous fully connected	$<C$	Transform the graph into shortest free delay graph, use critical path scheduling	-	$O(m)$	$O(m)$	$O(m)$	The schedule exceeds the optimal schedule by no more than $(n-2)$ time units.

T: type, A= Assignment, S= Scheduling.
Op: Optimal solution, ✓ =Yes, - = No.
Co: Communication; Ho= Homogenous links.
He= Heterogeneous links.
- = Cost not considered.
m: number of modules.
e : number of edges.
n : number of processors.
 ∞ : number of phases.
d: node degree.

Table 9.1 A comparison of assignment and scheduling algorithms.

CHAPTER X

CONCLUSION AND FUTURE RESEARCH

In this research, we have reviewed fifty algorithms dealing with static assignment and scheduling of tasks with no duplication onto parallel or distributed systems. These algorithms provide solutions for different aspects of this problem, where the nature of the program to be assigned and the interconnection structure of the parallel system vary from one problem to another. We have also presented a comparison of these algorithms based on the nature of the problem, the specific case they solve and the techniques used in deriving the solution.

We have commented on an example related to the paper "On The Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems" [8]. We have investigated the problem of mapping chain of m tasks onto chains of n processors, and derived a variation of Bokhari's algorithm [4,6,7] with a reduced space complexity in the case where the processors are heterogeneous. We have also suggested a heuristic solution for the same problem in the case of homogenous and heterogeneous system of $O(m)$ and $O(mn)$ time respectively.

We have also investigated Lee and Shin [24] optimal assignment of an arbitrary problem to homogenous processor networks having an n -dimensional array structure or a tree structure in the presence of a attached tasks. Based on their approach to the solution, we derived a solution for the optimal assignment of a parallel program of m arbitrary tasks onto a network of homogenous processors having a star graph interconnection structure S_n , with a runtime complexity of $O(Nm^3)$ time, where $N = n!$ is the total number of processors in the star graph. We also generalized our results to homogenous arbitrary network with a runtime complexity of $O(Nm^3)$ time, where N is the total number of processors in the arbitrary network.

For future research, we suggest that both heuristic algorithms described in chapter VII for mapping chains onto chains in the cases of homogenous and heterogeneous processors be compared against optimal solutions in order to define how often these algorithms provide for an optimal solution and how far their solution is from the optimal one in the worst case.

In the case of the optimal task assignment in homogenous networks in the presence of attached tasks, we provided a general algorithm that applies to arbitrary homogenous networks. Our general algorithm when applied to the n -dimensional array problem of $N(=n_1 \times n_2 \times \dots \times n_n)$ homogenous processors finds the optimal solution for a parallel program of m tasks in $O(Nm^3)$ time. On the other hand, Lee and Shin algorithm [24] is $O(\sum_i (n_i - 1)m^3)$ time for the same specific case. Their algorithm benefits from the possibility of isolating each dimension of the n -dimensional program to achieve this relatively better result. More research should be done to investigate the possibility of enumerating the star graph or any group graph in a similar n -dimensional fashion which might lead to the use of Lee and Shin algorithm for the n -dimensional array as is in order to achieve faster results.

References

- [1] S. B. Akers and B. Krishnamurthy, "On group graphs and their fault tolerance," *IEEE Trans. Comput.*, vol. C-36, pp. 885-888, July 1987.
- [2] S. B. Akers and B. Krishnamurthy, "A group theoretic model for symmetric interconnection networks," *IEEE Trans. Comput.*, vol. 38, pp. 555-566, Apr. 1989.
- [3] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Trans. Comput.*, vol. C-36, pp. 570-580, May 1987.
- [4] S. H. Bokhari, *Assignment problems in parallel and distributed computing*. Boston: Kluwer Academic, 1987.
- [5] S. H. Bokhari, "On the mapping problem," *IEEE Trans. Comput.*, vol. C-30, pp. 207-214, Mar. 1981.
- [6] S. H. Bokhari, "Partitioning problems in parallel, pipelined and distributed computing," *IEEE Trans. Comput.*, vol. 37, pp. 48-57, Jan. 1988.
- [7] S. H. Bokhari, "Partitioning problems in parallel, pipelined and distributed computing," Tech. Rep. 85-54, ICASE, Nov. 1985
- [8] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor, "On The Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans. Comput.*, vol. 41, pp. 257-273, Mar. 1992.
- [9] T. Bultan and C. Aykanat, "A new mapping heuristic based on mean field annealing," *J. Parallel Distrib. Comput.*, vol. 16, pp. 292-305, 1992.
- [10] E. G. Coffman, Jr., Ed., *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976.
- [11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269-271, 1959.
- [12] K. Efe and V. Krishnamoorthy, "Optimal scheduling of compute-intensive tasks on a network of workstations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 668-673, June. 1995.
- [13] M. J. Eisner and D. G. Severance, "Mathematical techniques for efficient record segmentation in large databases," *J. Ass. Comput. Mach.*, vol. 23, pp. 619-635, Oct. 1976.
- [14] H. El-Rewini, H. H. Ali, and T. Lewis, "Task scheduling in multiprocessing systems," *IEEE Computer*, pp. 27-37, Dec. 1995.

- [15] Z. Galil, S. Micali, and H. Gabow, "Priority queues with variable priority and an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs," in *Proc. 23rd Annu. Symp. Foundations Comput. Sci.*, pp. 255-261, Nov. 3-5, 1982.
- [16] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," *J. Parallel Distrib. Comput.*, vol. 16, pp. 276-291, Dec. 1992.
- [17] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 686-701, June 1993.
- [18] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," *Proc. 18th Ann. ACM Symp. Theory of Computing*, pp. 136-146, 1986.
- [19] C.-C. Hui and S. T. Chanson, "Allocating task interaction graphs to processors in heterogeneous networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, pp. 908-925, Sept. 1997.
- [20] M. A. Iqbal, "Approximate algorithms for partitioning and assignment problems," Tech. Rep. 86-40, ICASE, June 1986.
- [21] M. A. Iqbal and S. H. Bokhari, "Efficient Algorithms for a class of partitioning problems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 170-175, Feb. 1995.
- [22] H. Kasahra and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, pp. 1023-1029, Nov. 1984.
- [23] W. H. Kohler, K. Steiglitz, "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems," *J. ACM*, vol. 21, pp. 140-156, Jan. 1974.
- [24] C.-H. Lee and K. G. Shin, "Optimal task assignment in homogenous networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, pp. 119-128, Feb. 1997.
- [25] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. Comput.*, vol. 37, pp. 1384-1397, Nov. 1988.
- [26] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Comput.*, vol. C-31, pp. 41-47, Jan. 1982.
- [27] D. M. Nicol and D. R. O'Hallaron, "Improved algorithms for mapping parallel and pipelined computations," *IEEE Trans. Comput.*, vol. 40, pp. 295-306, Mar. 1991.

- [28] M. A. Palis, J.-C. Liou and D. S. L. Wei, "Task clustering and scheduling for distributed memory parallel architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 46-55, Jan. 1996.
- [29] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. Comput.*, vol. C-28, pp. 291-299, Apr. 1979.
- [30] C.-C. Shen and W.-H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. Comput.*, vol. C-34, pp. 197-203, Mar. 1985.
- [31] B. Shirazi and M. Wang, "Analysis and evaluation of heuristic methods for static task scheduling," *J. Parallel Distrib. Comput.*, vol. 10, pp. 222-232, 1990.
- [32] H. S. Stone, "Critical load factors in distributed computer systems," *IEEE Trans. Software Eng.*, vol. 4, pp. 254-258, May 1978.
- [33] S. Sur and P. K. Srimani, "A fault tolerant routing algorithm in star graph interconnection networks," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 267-270, 1991.
- [34] D. F. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. Software Eng.*, vol. 12, pp. 1,018-1,024, Oct. 1986.
- [35] T. A. Varvarigou, V. P. Roychowdhury, T. Kailath, and E. Lawler, "Scheduling in and out forests in the presence of communication delays," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 1065-1074, Oct. 1996.