A Thesis on

A Clustering Algorithm for Multiprocessor Environment using Communication-Computation Loads of Modules

Submitted for partial fulfillment of award of

MASTER OF TECHNOLOGY

degree

In

Computer Science & Engineerring

By

KAMAL SHEEL MISHRA

Dr. Pramod Kumar Mishra

SUPERVISOR

UTTAR PRADESH TECHNICAL UNIVERSITY, LUCKNOW, INDIA

NOVEMBER, 2009

This page is intentionally left blank.

CERTIFICATE

Certified that **Kamal Sheel Mishra** has cariied out the research work presented in this thesis entitled **"A Clustering Algorithm for Multiprocessor Environment using Communication - Computation Loads of Modules"** for the award of **Master of Technology** from Uttar Pradesh Technical University, Lucknow under my supervision. The thesis embodies result of original work and studies carried out by Student himself and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else.

(Dr. Pramod Kumar Mishra)

Designation: Reader

Address: Department of Computer Science, Banaras Hindu University, Varanasi - 221005.

Date:

ABSTRACT

In this project, a new clustering algorithm is proposed that tries to reduce the parallel execution time by considering the computation-communication loads of modules. Load of a module is defined as a simple linear function of computation time and communication time. This load is used in deciding about the clustering of modules. This algorithm is compared with the optimal algorithm for small sized random task graphs. It is also compared with some existing algorithms and shown to be performing well.

ACKNOWLEDGEMENT

This project entitled 'A CLUSTERING ALGORITHM FOR MULTIPROCESSOR ENVIRONMENT USING COMMUNICATION-COMPUTATION LOADS OF MODULES' carried out by me is a sincere and painstaking work. But no such work is possible without proper guidance. For this I am deeply indebted to my thesis supervisor and guide **Dr. P. K. Mishra (Reader & Head, Department of Computer Science, Banaras Hindu University, Varanasi-221005)** for his invaluable guidance. I express my gratefulness to him. It would have never been possible for me to take this project to completion without his innovative ideas and encouragement. He was a constant force behind my work.

I also wish to thank whole-heartedly Prof P. N. Jha, Director, SMS, Varanasi, and the management of School of Management Sciences for providing me the opportunity to complete this work. I am thankful to all the faculty members of SMS, Varanasi for enhancing my knowledge. I would like to thank whole of the M. Tech. batch for the times I shared with them specially Mr. P. R. Tripathi, Mr. Rajiv Katare, and Mr. T. P. Singh. My special thanks to Mr. Abhishek Mishra (Research Scholar, IT BHU, Varanasi) for helping me whenever I faced problems in developing the tool.

I would like to thank everyone worked with me for providing a nice and challenging work environment.

(KAMAL SHEEL MISHRA)

Contents

List of Figures

Chapter 1

INTRODUCTION

1.1 MULTIPROCESSOR ENVIRONMENT

Parallel architectures are used to solve a variety of large-scale software applications. The availability of multiprocessors helps in concurrent executions of processes. A number of clustering and scheduling algorithms have been developed to extract parallelism. In a multiprocessor architecture, the computation results produced by one processor are sent, over the communication network, to another processor that depends on the newly computed data. The processors in a parallel architecture are generally fully connected, and they exchange information with message passing. Because parallel computer systems are built upon a highly reliable communication backbone, the senders of message generally do not wait for an acknowledgment of message reception by the receiver. Our approach for clustering is to sort the loads of modules in decreasing order. Initially all modules are assumed to be present in a single processor. We take out a module having highest load, and evaluate the

parallel execution time. If the parallel execution time is decreased, then that module is allocated to a different processor. If the parallel execution time is not reduced, then the next module having the highest load is considered. In this project, a new clustering algorithm is proposed that tries to reduce the parallel execution time by considering the computation-communication loads of modules. Load of a module is defined as a simple linear function of computation time and communication time. This load is used in deciding about the clustering of modules. This algorithm is compared with the optimal algorithm for small sized random task graphs.

1.2 CLUSTERING

A parallel software system consists of distributed processes. Each process is a sequentially executable set of modules that are known as clusters. All the modules in a cluster are executed on the same processor. The parallel time is the execution time of the distributed parallel system. One of the objectives during clustering is the reduction of parallel time. One possibility of clustering is to create as many clusters as there are modules, and the other possibility is to create one cluster containing all the modules. The first approach would induce communication delays resulting in an increase in the parallel time, and the second approach would fail to exploit potential concurrency among modules. A clustering heuristic uses both communication costs and execution costs to make a balance between these two extreme cases of

clustering techniques. The multiprocessor's architecture during a clustering phase is assumed to be a completely connected set of processors, where the number of processors is unbounded. A clustering in which independent tasks are clustered together is called nonlinear. In a linear clustering, independent modules are not grouped in the same cluster. To solve the general task allocation problem of allocating a given Directed Acyclic Graph (DAG) of modules to a limited number of processors that have any type of interconnection network. We generally employ a two-step procedure: Step 1 Make a clustering of modules assuming an unlimited supply of identical and fully connected homogeneous network of processors. Step 2 Allocate these clusters to processor using some heuristics so as to minimize the execution time.

1.3 ASSUMPTIONS

In this project, three assumptions are made:

1. The network of processors is fully connected having links between any pair of processors.

2. The network of processors is homogeneous in which all processors, as well as the links connecting them are identical.

3. We have an unlimited supply of processors.

1.4 COMPUTATIONAL MODEL

A parallel software system has distributed processes. Each process contains a sequentially executable set of modules that do computations. Each module executes a set of instructions and outputs results in the form of newly computed data. The computed data is sent to every module that depends on it and the data flows in the form of messages. Now we define the data dependence graph (DDG) and data structure, as well as timing notations which will be used to discuss the clustering algorithm latter.

The data dependence graph (DDG) of a software system gives an ordering of execution of program instructions. When the DDG is applied at the level of software modules, the resulting graph is termed a *module graph* which is defined as shown below. The nodes of the module graph correspond to modules while the edges correspond to dependences. The costs associated with the graph are: 1) a module's computation cost and 2) message cost for every message moving from one module to another. We are using the same computational model as described in (Dinesh Kadamuddi and Jeffrey J.P. Tsai 2000 [1]):

Definition. A Module Graph, G, is represented as $G(V, E, \alpha, \beta)$ where,
V is the set of all the modules in the graph.
E is the set of all the directed edges in the graph. These edges denote data dependences.
For each $v \in V$, α_v denotes the computation cost (time) of module.

For each $(u, v) \in E$, $\beta_{(u,v)}$ denotes the cost (time) of message passing from u to v. When u sends a message to v, it will arrive at v after $\beta_{(u,v)}$ amount of time. It is assumed that these values are derived by testing message transfers between source and destination with varying message sizes. By using such observations, the cost of a message transfer can be derived based on the size of the message.

1.5 MODULE BEHAVIOR

The execution behavior of each module in a module graph can be described by these steps (Dinesh Kadamuddi and Jeffrey J.P. Tsai 2000 [1]): *Receive, Compute, and Send.*

1. Receive all of the required data (messages). The messages are received as they arrive. Consequently, messages can arrive overlapped in time and they are assumed to be handled without delay.

2. Compute the results. The actual work of the module is performed in this step. It is assumed that the all of the required data for computations is available locally before computations can begin.

3. Send the results to all of the dependent modules. Messages are sent out in parallel to all the dependent modules. The sender is a blocked until until the dependent module actually receives the data.

1.6 REPORT ORGANIZATION

The remainder of the report is organized as follows. In chapter 2, some current approaches for solving the clustering problem on a multiprocessor environment is discussed. In chapter 3, a recursive method for generating all possible clustering is discussed. In chapter 4, an algorithm for finding the execution time for a given clustering is discussed. In chapter 5, an optimal algorithm for solving the clustering problem on a multiprocessor environment is discussed. In chapter 6, a suboptimal algorithm using the communication-computation loads is discussed. In chapter 7, the suboptimal algorithm is compared with the CASC algorithm (Dinesh Kadamuddi and Jeffrey J.P. Tsai 2000 [1]). In chapter 8, some experimental results are presented. And finally in chapter 9, the project is concluded. Appendix A shows the code listing for the optimal algorithm, and the suboptimal algorithm based on communication-computation loads of modules.

Chapter 2

CURRENT APPROACHES

2.1 CLUSTERING ALGORITHMS IN THE CURRENT LITERATURE

Sarkar's heuristic (V. Sarkar 1989 [9]) reduces the parallel time at each step by considering the highest cost edge. The approach is summarized by the following steps: 1) sort the edges of the DAG in descending order of edge costs, 2) merge the clusters connected by the edge with the highest cost, if parallel time does not increase, and 3) repeat Step two until all edges are examined. During Step two, if the parallel time increases for an edge, then the two nodes connected by that edge are scheduled on separate processors. Sarkar's algorithm uses a level information to determine parallel time and these levels are computed for each step. The time complexity of Sarkar's algorithm is $O(E(V+E))$. The clustering is nonlinear.

The algorithm of Kim and Browne (S.J. Kim and J.C. Browne 1988 [5]), referred to as KB in this report, uses the critical path information to create

clusters in a parallel system. The approach is summarized by the following steps: 1) mark all edges in the DAG as unexamined, 2) find the critical path composed of unexamined edges only, 3) the nodes in the critical path are clustered and all the edges incident upon the nodes in the newly created cluster are marked as examined, 4) repeat Steps two and three until all edges are examined. The KB algorithm finds a critical path in every step and the time complexity for this step is $O(V+E)$. Since the number of connected components is at most V, the time complexity of the KB algorithm is $O(V(V+E))$. Since all edges incident to nodes in each cluster are marked as examined, the KB algorithm creates linear clusters.

The Dominant Sequence Clustering (DSC) algorithm of Yang and Gerasoulis (T. Yang and A. Gerasoulis 1991 [7]), (T. Yang and A. Gerasoulis 1992 [8]) finds the critical path of the graph. The critical path is called the Dominant Sequence (DS). An edge from the DS is used to merge its adjacent nodes, provided the parallel time reduces. After merging, a new DS is computed and the clustering is tried again. Yang and Gerasoulis have shown the time complexity of DSC algorithm as $O((V+E)log(V))$. The Greedy Dominant Sequence (GDS) algorithm, of Dikaiakos et al. (M.D. Dikaiakos, A. Rogers, and K. Steiglitz 1994 [3]) is a simpler, greedy version of DSC algorithm. In GDS, an edge from DS that reduces the parallel time the most, is selected and the nodes belonging to the edge are merged. The algorithm stops when there is no edge in DS that is able to decrease the parallel time. The time complexity of GDS algorithm is $O(V(V+E))$.

Wu and Gajski (M.Y. Wu and D.D. Gajski 1990 [4]) proposed a Modified Critical Path algorithm for clustering on a parallel system with a bounded number of processors. The MCP algorithm becomes an edge-zeroing clustering algorithm when it is used on a completely connected architecture with unbounded number of processors. The algorithm uses As-Late-As-Possible (ALAP) time of a node that is obtained by ALAP binding created by moving downward through the graph. The MCP algorithm has a time complexity of $O(V^2 log(V))$.

The Dynamic Critical Path (DCP) algorithm of Kwok and Ahmad (Y.K. Kwok and I. Ahmad 1996 [11]) is a clustering algorithm for a parallel system with a bounded number of processors. Similar to the MCP algorithm, assuming an unbounded number of processors, the DCP algorithm reduces to an edge-zeroing clustering algorithm. The algorithm uses the critical path of a graph for clustering and a node belonging to the CP is clustered. Next, the CP is again computed dynamically because the edge zeroing would have a created a new CP in the graph. To differentiate between the original CP of the graph from CPs computed repeatedly, the intermediate CP is called the dynamic critical path (DCP). The time complexity of DCP algorithm is $O(V^3)$.

Chapter 3

GENERATING ALL POSSIBLE CLUSTERINGS

3.1 NOTATION

Let n be the number of modules. Let the modules be denoted by $M_i (0 \leq i \leq n-1)$. Let $\mathcal{M}_n = \{M_i \mid 0 \leq i \leq n-1\}$ be the set of all modules. Let $X \subset \mathcal{M}_n$ be a subset of modules, then a clustering of X having m clusters is denoted as $\{X_i \mid X_i \subset X (1 \leq i \leq m), \bigcup_{i=1}^{m} X_i = X, X_j \bigcap X_k = \emptyset (j \neq k)\}$.

Definition. For $X \subset \mathcal{M}$, the set of all clusterings, $G(X)$ is defined as:

$$G(X) = \{Y \mid Y \text{ is a clustering of } X\} \tag{3.1}$$

Definition. The number of all possible clustering of n modules, $g(n)$ is defined as:

$$g(n) = |G(\mathcal{M}_n)| \tag{3.2}$$

Definition. Let $X \subset \mathcal{M}_n$ and $Y \subset \mathcal{M}_n$ be such that $X \bigcap Y = \emptyset$. Then, the binary operation $\circledast$ between X and $G(Y)$ is defined as:

$$X \circledast G(Y) = \{X \bigcup Z \mid Z \text{ is a clustering of } Y\} \tag{3.3}$$

3.2 EXAMPLES

For $n = 0$, we have $\mathcal{M}_0 = \emptyset$.

$$G(\mathcal{M}_0) = \{\{\emptyset\}\} \tag{3.4}$$

$$g(0) = |G(\mathcal{M}_0)| = |\{\{\emptyset\}\}| = 1 \tag{3.5}$$

For $n = 1$, we have $\mathcal{M}_1 = \{M_0\}$.

$$G(\mathcal{M}_1) = \{\{\{M_0\}\}\} \tag{3.6}$$

$$g(1) = |G(\mathcal{M}_1)| = |\{\{\{M_0\}\}\}| = 1 \tag{3.7}$$

For $n = 2$, we have $\mathcal{M}_2 = \{M_0, M_1\}$. Let X be the cluster containing M_0. Let $Y = \mathcal{M}_2 - X$. Then, we have 2^1 possibilities for X and Y:

$$X = \{M_0\}, \quad Y = \{M_1\} \tag{3.8}$$

$$X = \{M_0, M_1\}, \quad Y = \emptyset \tag{3.9}$$

(3.8) generates the clusters:

$$\begin{aligned} C_1 = X \circledast G(Y) = \{M_0\} \circledast G(\{M_1\}) \\ = \{M_0\} \circledast \{\{\{M_1\}\}\} = \{\{\{M_0\}, \{M_1\}\}\} \end{aligned} \tag{3.10}$$

(3.9) generates the clusters:

$$C_2 = X \circledast G(Y) = \{M_0, M_1\} \circledast G(\emptyset) = \{M_0, M_1\} \circledast \{\{\emptyset\}\} = \{\{\{M_0, M_1\}\}\} \tag{3.11}$$

Therefore, from (3.8) and (3.9), we have:

$$G(\mathcal{M}_2) = C_1 \bigcup C_2 = \{\{\{M_0\}, \{M_1\}\}, \{\{M_0, M_1\}\}\} \tag{3.12}$$

and,

$$g(2) = |G(\mathcal{M}_2)| = |\{\{\{M_0\}, \{M_1\}\}, \{\{M_0, M_1\}\}\}| = 2 \tag{3.13}$$

For $n = 3$, we have $\mathcal{M}_3 = \{M_0, M_1, M_2\}$. Let X be the cluster containing M_0. Let $Y = \mathcal{M}_3 - X$. Then, we have 2^2 possibilities for X and Y:

$$X = \{M_0\}, \quad Y = \{M_1, M_2\} \tag{3.14}$$

$$X = \{M_0, M_1\}, \quad Y = \{M_2\} \tag{3.15}$$

$$X = \{M_0, M_2\}, \quad Y = \{M_1\} \tag{3.16}$$

$$X = \{M_0, M_1, M_2\}, \quad Y = \emptyset \tag{3.17}$$

(3.14) generates the clusters:

$$C_1 = \{M_0\} \circledast G(\{M_1, M_2\}) = \{M_0\} \circledast \{\{\{M_1\}, \{M_2\}\}, \{\{M_1, M_2\}\}\} \tag{3.18}$$

or,

$$C_1 = \{\{\{M_0\}, \{M_1\}, \{M_2\}\}, \{\{M_0\}, \{M_1, M_2\}\}\} \tag{3.19}$$

(3.15) generates the clusters:

$$C_2 = \{M_0, M_1\} \circledast G(\{M_2\}) = \{M_0, M_1\} \circledast \{\{\{M_2\}\}\} \tag{3.20}$$

or,

$$C_2 = \{\{\{M_0, M_1\}, \{M_2\}\}\} \tag{3.21}$$

(3.16) generates the clusters:

$$C_3 = \{M_0, M_2\} \circledast G(\{M_1\}) = \{M_0, M_2\} \circledast \{\{\{M_1\}\}\} \tag{3.22}$$

or,

$$C_3 = \{\{\{M_0, M_2\}, \{M_1\}\}\} \tag{3.23}$$

(3.17) generates the clusters:

$$C_4 = \{M_0, M_1, M_2\} \circledast G(\emptyset) = \{M_0, M_1, M_2\} \circledast \{\{\emptyset\}\} \tag{3.24}$$

or,

$$C_4 = \{\{\{M_0, M_1, M_2\}\}\} \tag{3.25}$$

From (3.19), (3.21), (3.23), and (3.25), we have:

$$G(\mathcal{M}_3) = C_1 \bigcup C_2 \bigcup C_3 \bigcup C_4 \tag{3.26}$$

and,

$$\begin{aligned} g(3) = |G(\mathcal{M}_3)| &= |C_1| + (|C_2| + |C_3|) + |C_4| \\ &= \binom{2}{0} g(2) + \binom{2}{1} g(1) + \binom{2}{2} g(0) \end{aligned} \tag{3.27}$$

or,

$$g(3) = 1(2) + 2(1) + 1(1) = 5 \tag{3.28}$$

3.3 COMPUTING $G(\mathcal{M}_n)$ AND $g(n)$

Generalizing the examples for n, we get:

$$G(\mathcal{M}_n) = \bigcup_{M_0 \in X, Y = \mathcal{M}_n - X} X \circledast G(Y) \tag{3.29}$$

and,

$$g(n) = \sum_{i=0}^{n-1} \binom{n-1}{i} g(n-1-i) \tag{3.30}$$

Chapter 4

FINDING THE EXECUTION TIME FOR A GIVEN CLUSTERING

4.1 THE EVENT QUEUE MODEL

We can calculate the execution time for a given clustering using the event queue model.

Definition. Event Queue Model is defined as follows:

There will be two types of Events:

1. Computation Completion Event.

2. Communication Completion Event.

Let the task graph be given as an adjacency matrix $TG[][]$. We will denote a computation completion event for a module M_i as a 3-tuple (i, i, t), where t is the timestamp at which the module M_i completes its execution.

We will denote a communication completion event from a module M_i to another M_j as a 3-tuple (i, j, t), where M_i is the origin of communication, M_j is the destination of communication, and t is the timestamp at which the communication is completed.

Let $allocation[i] = j$ if the module M_i is allocated to the processor P_j for $(0 \leq i, j \leq V - 1)$ where V is the number of modules in the task graph.

Algorithm CalculateTime$(TG[][], allocation[])$

{

Step 0: Let $\mathcal{E}$ be the event queue. It will be an ascending priority queue based on the value of t. For a given allocation, we will also maintain a *ready queue* of ready to run modules for each processor. A module M_i is ready to run, if it is not having any incoming edges, or, if all of its incoming edges have finished their communication. Initially at least one module will be ready to run for which there is no incoming edge. For each processor, one ready to run module will be added to the event queue (if one exists), setting the t value as the time for execution. Corresponding processor states will be set to $BUSY$.

Step 1: One node will be deleted from the event queue $\mathcal{E}$. There will be a $TIME$ variable initialized to 0. $TIME$ denotes the current time. Whenever any node is deleted from $\mathcal{E}$, $TIME$ will be updated to its timestamp. If the deleted node is a computation completion event, then the following actions will be take place:

Step 2: Each processor can be in two states: $IDLE$ when it is not executing any module. Initially all processors will be in $IDLE$ state. A processor is in $BUSY$ state, when it is executing a module. When a module finishes its execution, the corresponding processor's state is changed to $IDLE$.

Step 3: If there is any other ready to run module on that processor, then its t value will be set to $TIME +$ *computation time of module*, and will be added to $\mathcal{E}$. That processor state will again be changed from $IDLE$ to $BUSY$.

Step 4: For each outgoing edge from the module, a communication completion event will be added to $\mathcal{E}$. For setting the value of t, we have two possibilities (for a communication from Mi to M_j:

Step 4A: M_i and M_j are on the same processor. Then t is set to $TIME$, since there will not be any communication delay.

Step 4B: M_i and M_j are on different processors. Then t is set to $TIME +$ *edge weight*.

Step 5: When the deleted node is a communication completion event, then the following actions will take place:

Let (i, j, t) be the deleted event. Check for M_j if all of its incoming edges have completed their communication. If this is so, add M_j to the *ready queue* of its allocated processor. If that processor is currently $IDLE$, then change its state to $BUSY$, and add a computation completion Event of M_j to $\mathcal{E}$.

Step 6: **Repeat** *Step 1* to *Step 6* untill a total of $V + E$ events are added to, and deleted from E.

Step 7: **Return** $TIME$.

}

4.2 COMPLEXITY ANALYSIS OF THE ALGORITHM CalculateTime()

For the adjacency matrix representation of the module graph, *Step 0* takes $O(V^2)$ time. This is because, for each vertex of the module graph, we have to check for every other vertex, whether there exists an incoming edge from that vertex.

Let V be the number of vertices, and E be the number of edges in the task graph. Then, there will be a total of $(E + V)$ events out of which V events will be computation completion events corresponding to each module, and E events will be communication completion events corresponding to each edge.

Step 1 to *Step 6* are repeated a total of $(E+V)$ times. In each repeatition, the complexity is dominated by the addition and deletion from the event queue E that has a complexity of $O(V + E)$ (Horowitz-Sahni (Eillis Horowitz, Sartaj Sahni 1978 [2]), Tenenbaum (Yedidyah Langsam, Moshe J. Augenstein, and Aaron M. Tenenbaum 1996 [10]), Cormen-Rivest (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein 2001 [6])) if the

priority queue is implemented as a linear linked list. Therefore the complexity of the **CalculateTime()** algorithm is $O((V + E)^2)$.

Chapter 5

AN OPTIMAL CLUSTERING ALGORITHM

Let n be the number of modules. The algorithm **OptimalClustering()** returns the minimum time taken, and the corresponding clustering.

5.1 THE ALGORITHM OptimalClustering()

Algorithm OptimalClustering(TG[][])

{

Step 1: $min \leftarrow INFINITY$

Step 2: Generate all possible $clustering$ using the function $G(\mathcal{M}_n)$ of chapter 3.

Step 3: **for each** $clustering \in G(\mathcal{M}_n)$

Step 4: **do** $time \leftarrow CalculateTime(TG[][], clustering[])$

Step 5: **if** $time < min$

Step 6: **then** $min \leftarrow time$

Step 7: $minClustering \leftarrow clustering$

Step 8: **return** $min, minClustering$

}

5.2 COMPLEXITY ANALYSIS OF THE ALGORITHM OptimalClustering()

The function $G(\mathcal{M}_n)$ is exponential in n:

$$|G(\mathcal{M}_n)| = g(n) = \sum_{i=0}^{n-1} \binom{n-1}{i} g(n-i) \geq \sum_{i=0}^{n-1} \binom{n-1}{i} = 2^{n-1} \quad (5.1)$$

Therefore, the algorithm **OptimalClustering()** has exponential time complexity.

5.3 AN EXAMPLE OF OPTIMAL CLUSTERING

Consider the task graph in figure 5.1:

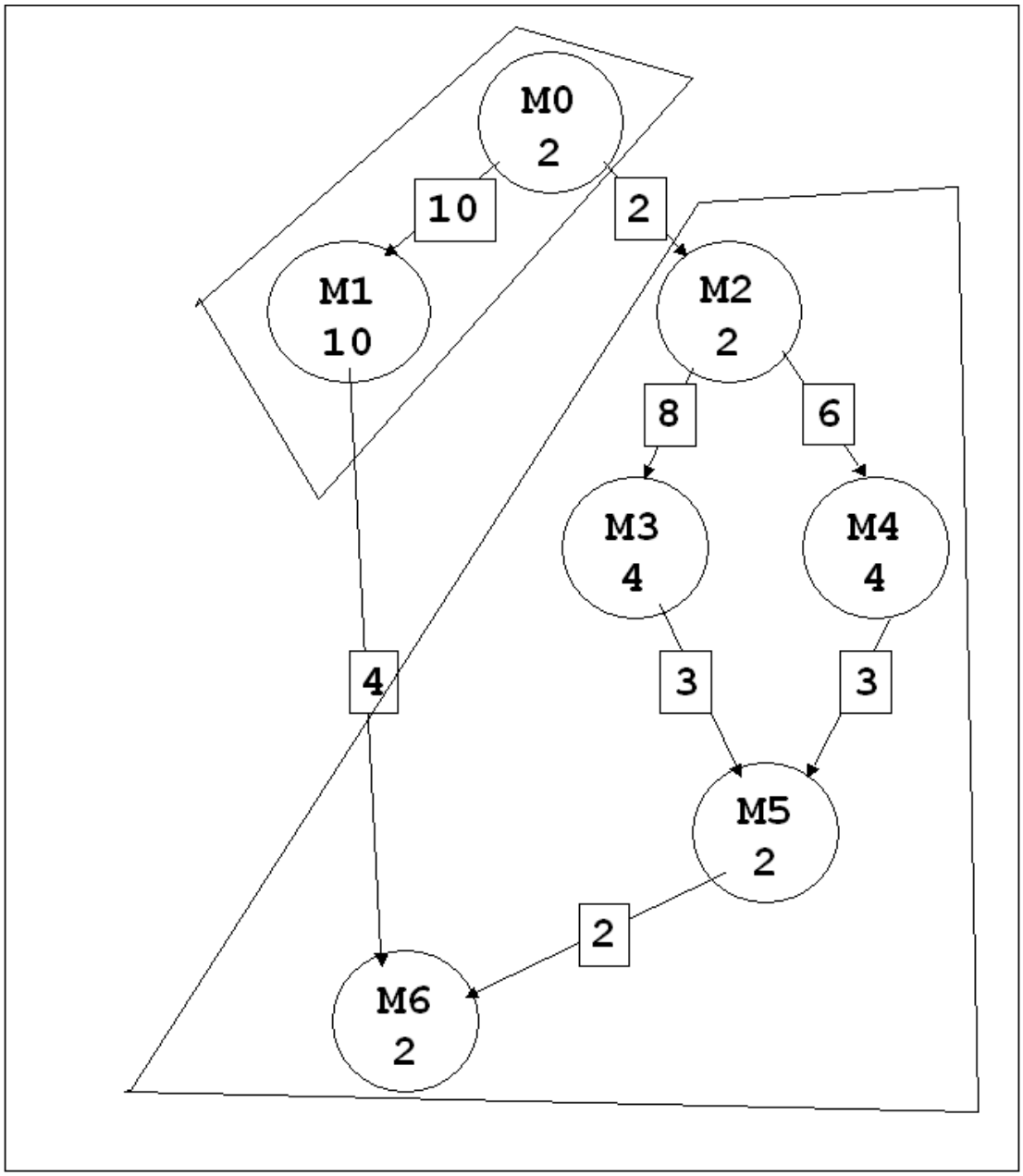

FIGURE 5.1: An example task graph. optimal clustering $= \{\{M_0, M_1\}, \{M_2, M_3, M_4, M_5, M_6\}\}$. $TIME = 18$.

In the optimal clustering (using the algorithm **OptimalClustering()**), modules M_0 and M_1 are clustered together, and the modules M_2, M_3, M_4, M_5, and M_6 are clustered together. Parallel execution time comes out to be 18.

Chapter 6

A SUBOPTIMAL CLUSTERING ALGORITHM USING COMMUNICATION - COMPUTATION LOADS OF MODULES

6.1 COMMUNICATION-COMPUTATION LOAD OF A MODULE

A module is communication intensive, if it spends more time in communication as compared to its time spent in computation. Similarly, a module is computation intensive, if it spends more time in computation as compared to its time spent in communication. To measure the relative communication-computation load of a module, we define a value called $load$ of a module as follows:

$$load_i = time_i - MAX(\{weight_{ji} \mid 0 \leq j \leq V-1\}) \\ -MAX(\{weight_{ik} \mid 0 \leq k \leq V-1\}) \tag{6.1}$$

Load of a module M_i ($load_i$) is defined as its execution time ($time_i$) subtracted by maximum weight of incoming edge ($MAX(\{weight_{ji} \mid 0 \leq j \leq V-1\})$) subtracted by maximum weight of outgoing edge ($MAX(\{weight_{ik} \mid 0 \leq k \leq V-1\})$).

6.2 AN EXAMPLE OF COMMUNICATION-COMPUTATION LOAD

In figure 6.1, $load$ of modules are shown along with the module execution times. For example, for module M_2, $time_2 = 2$, maximum weight of incoming edge is $weight_{02} = 2$, maximum weight of outgoing edge is $weight_{23} = 8$. Therefore,

$$load_2 = time_2 - weight_{02} - weight_{23} = 2 - 2 - 8 = -8 \tag{6.2}$$

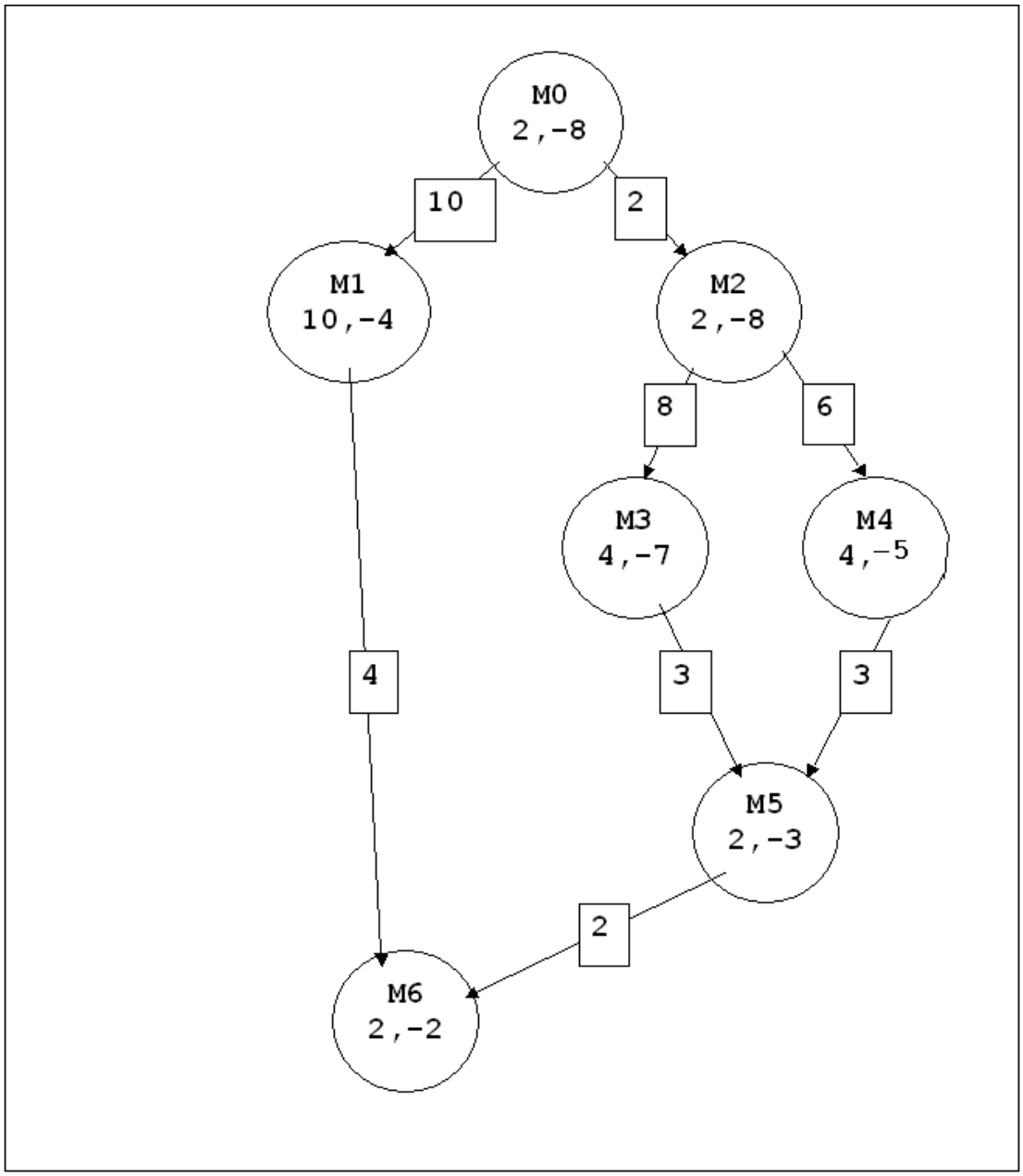

FIGURE 6.1: An example task graph showing *load* of modules.

6.3 ALGORITHM FOR COMPUTING *load* OF MODULES

Algorithm CalculateLoad(TG[][])

{

Step 1: **for** $j \leftarrow 0$ to $V-1$

Step 2: **do** $max_{in} \leftarrow 0$

Step 3: $max_{out} \leftarrow 0$

Step 4: **for** $k \leftarrow 0$ to $V-1$

Step 5: **do if** $k \neq j$

Step 6: **then if** $TG[k][j] > max_{in}$

Step 7: **then** $max_{in} = TG[k][j]$

Step 8: **if** $TG[j][k] > max_{out}$

Step 9: **then** $max_{out} = TG[j][k]$

Step 10: $load_j = time_j - max_{in} - max_{out}$

}

6.4 COMPLEXITY ANALYSIS OF CalculateLoad()

CalculateLoad() has two nested **for** loops of length V each. Therefore, the time complexity of the algorithm **CalculateLoad()** is $O(V^2)$.

6.5 A CLUSTERING ALGORITHM USING $load$ OF MODULES

Algorithm CCLoadClustering(TG[][])

{

Step 1: $load[V] \leftarrow CalculateLoad(TG[][])$

Step 2: Sort the array $load[]$ in decreasing order. Let $load[i].index$ be the

module, and $load[i].value$ be the $load$ of the module for $(0 \leq i \leq V-1)$.

Step 3: Initially let each module be allocated to the processor P_0. Let the corresponding allocation array of modules be $allocation[V]$.

Step 4: $min \leftarrow CalculateTime(TG[][], allocation[])$

Step 5: $min\_allocation \leftarrow allocation$

Step 6: **for** $i \leftarrow 0$ to $V-1$

Step 7: **do** Remove the module $load[i].index$ from the cluster on the processor P_0.

Step 8: **for** each possible way of putting the module $load[i].index$ on a separate cluster

Step 9: **do** $time \leftarrow CalculateTime(TG[][], allocation[])$

Step 10: **if** $time < min$

Step 11: **then** $min = time$

Step 12: $min\_allocation \leftarrow allocation$

Step 13: **return** $min, min\_allocation$

}

The algorithm **CCLoadClustering()** is based on the heuristic that the computation intensive modules should be kept separate, and that the communication intensive modules should be kept on the same cluster to take advantage of edge zeroing.

6.6 COMPLEXITY ANALYSIS OF THE ALGORITHM CCLoadClustering()

Step 1 takes $O(V^2)$ time. *Step 2* takes $O(Vlog(V))$ time. *Step 3* takes $O(V)$ time. *Step 4* takes $O((V+E)^2)$ time. *Step 5* takes $O(V)$ time. The **for** loop of *Step 6* iterates V times. The inner **for** loop of *Step 8* can iterate a maximum of V times. Complexity of each iteration of the inner **for** loop is dominated by *Step 9* that has complexity $O((V+E)^2)$. Therefore, the **for** loop of *Step 6* has complexity $O(V^2(V+E)^2)$ that is also the complexity of the algorithm **CCLoadClustering()**.

6.7 AN EXAMPLE OF THE ALGORITHM CCLoadClustering()

In FIGURE 6.2, initially all the modules are kept on the same cluster. Parallel execution time comes out to be 26. Module M_6 has the greatest $load$ of -2. Therefore, in FIGURE 6.3, M_6 is taken out from the initial cluster. Parallel execution time comes out to be 28. This is greater than as compared to FIGURE 6.2. Therefore, the module M_6 is put back into the initial cluster, and the next module having the greatest $load$ of -3 (M_5) is examined.

In FIGURE 6.4, M_5 is taken out from the initial cluster. Parallel execution time comes out to be 31. This is greater than as compared to FIGURE 6.2. Therefore, the module M_5 is put back into the initial cluster, and the next module having the greatest $load$ of -4 (M_1) is examined.

In FIGURE 6.5, M_1 is taken out from the initial cluster. Parallel execution time comes out to be 28. This is greater than as compared to FIGURE 6.2. Therefore, the module M_1 is put back into the initial cluster, and the next module having the greatest $load$ of -5 (M_4) is examined.

In FIGURE 6.6, M_4 is taken out from the initial cluster. Parallel execution time comes out to be 31. This is greater than as compared to FIGURE 6.2. Therefore, the module M_4 is put back into the initial cluster, and the next module having the greatest $load$ of -7 (M_3) is examined.

In FIGURE 6.7, M_3 is taken out from the initial cluster. Parallel execution time comes out to be 33. This is greater than as compared to FIGURE 6.2. Therefore, the module M_3 is put back into the initial cluster, and the next module having the greatest $load$ of -8 (M_0) is examined.

In FIGURE 6.8, M_0 is taken out from the initial cluster. Parallel execution time comes out to be 28. This is greater than as compared to FIGURE 6.2. Therefore, the module M_0 is put back into the initial cluster, and the next module having the greatest $load$ of -8 (M_2) is examined.

In FIGURE 6.9, M_2 is taken out from the initial cluster. Parallel execution time comes out to be 24. This is less than as compared to FIGURE 6.2. Now, all the modules have been examined. Therefore, the clustering generated by the algorithm **CCLoadClustering()** is $\{\{M_0, M_1, M_3, M_4, M_5, M_6\}, \{M_2\}\}$ having the parallel execution time of 24.

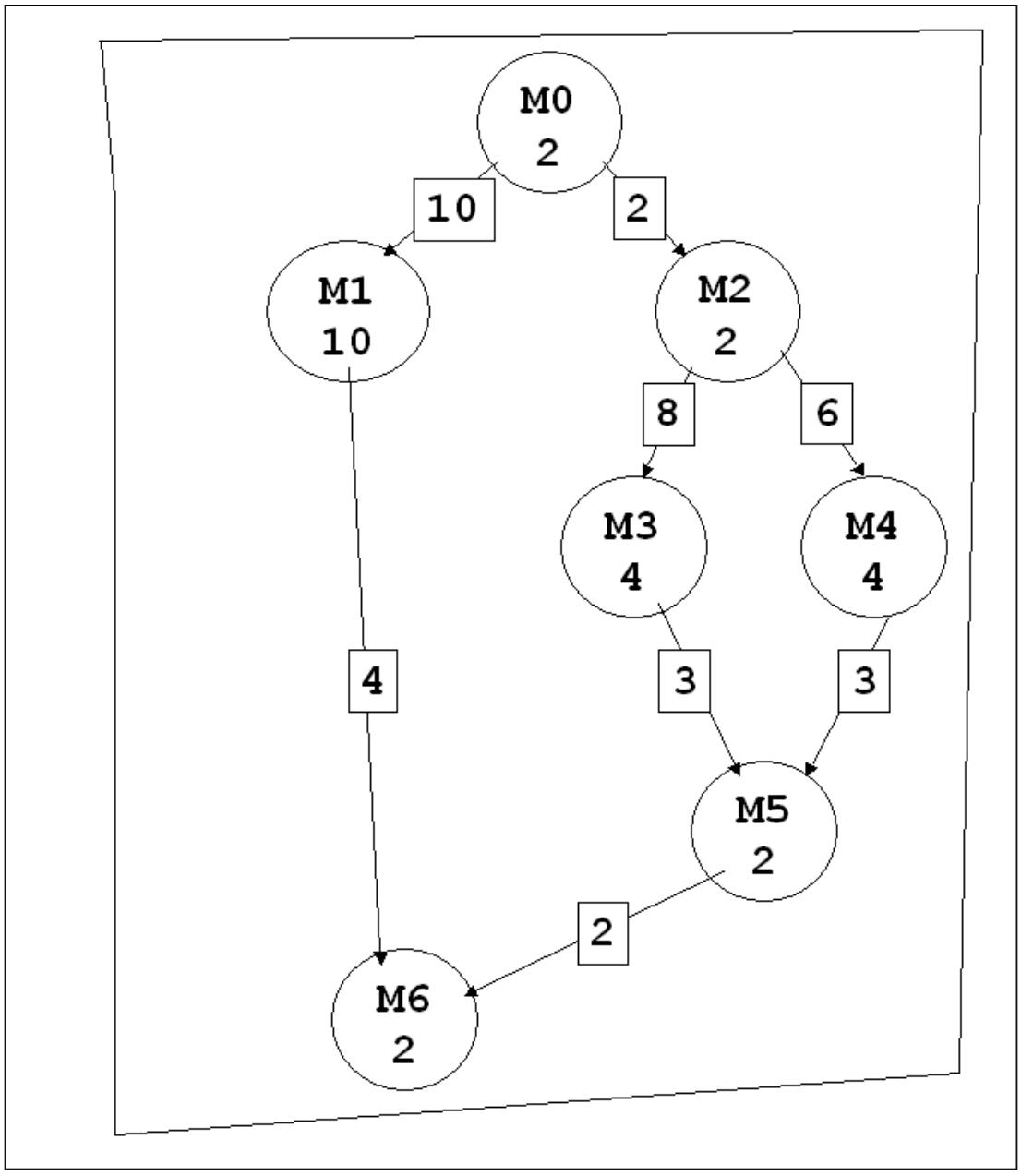

FIGURE 6.2: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 26$, $allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

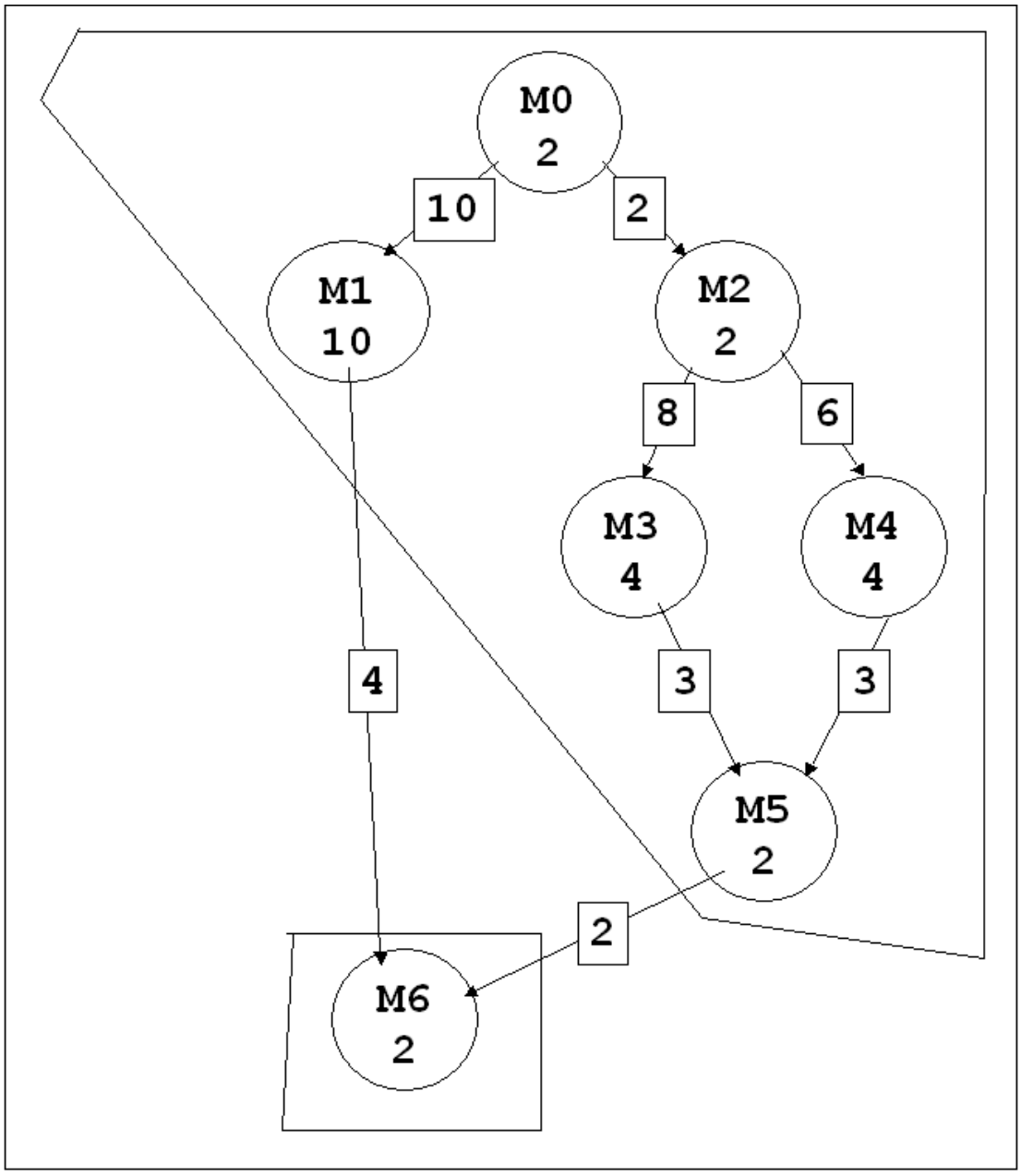

FIGURE 6.3: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 28$, $allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5\}, \{M_6\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

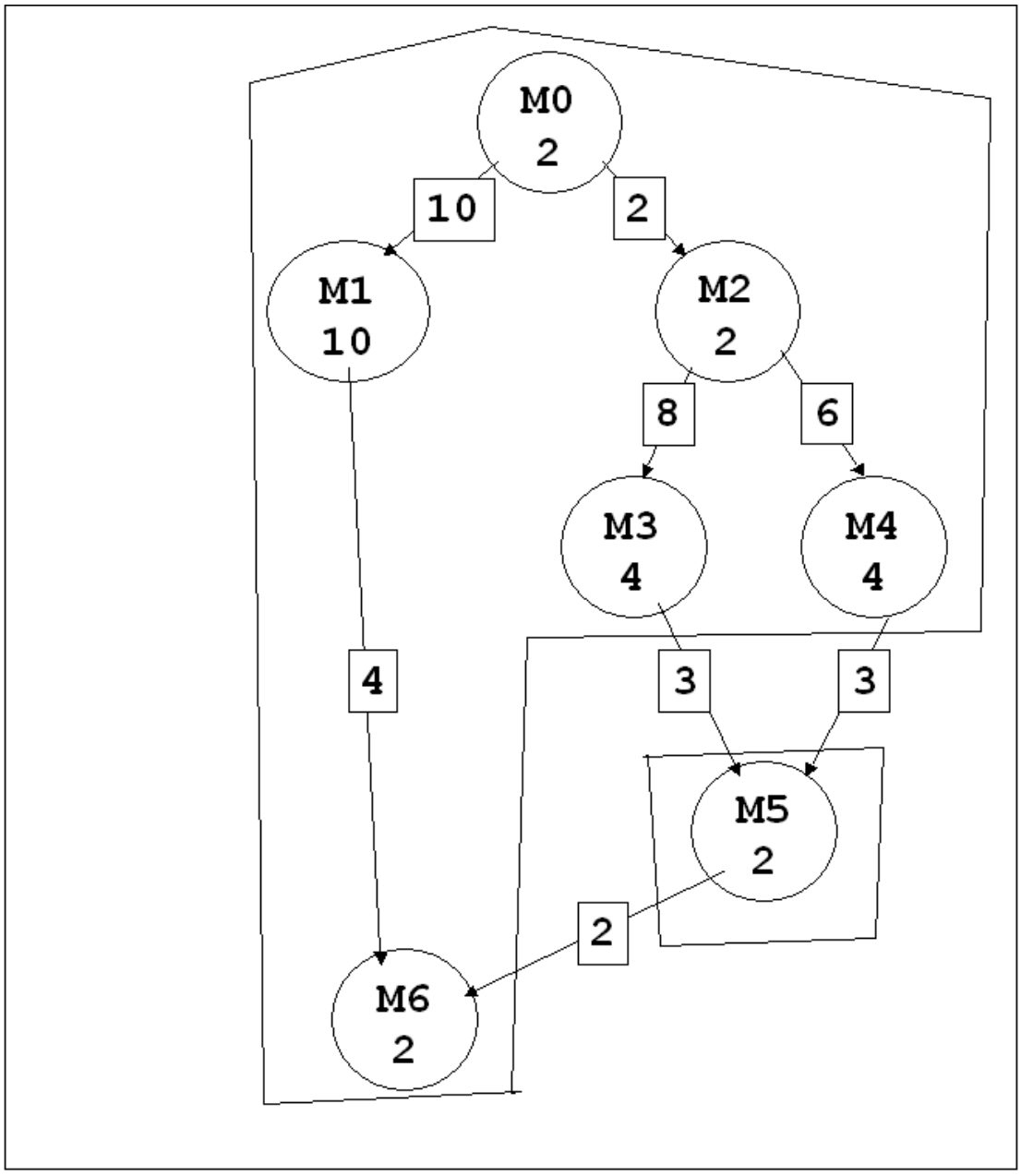

FIGURE 6.4: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 31$, $allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_6\}, \{M_5\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

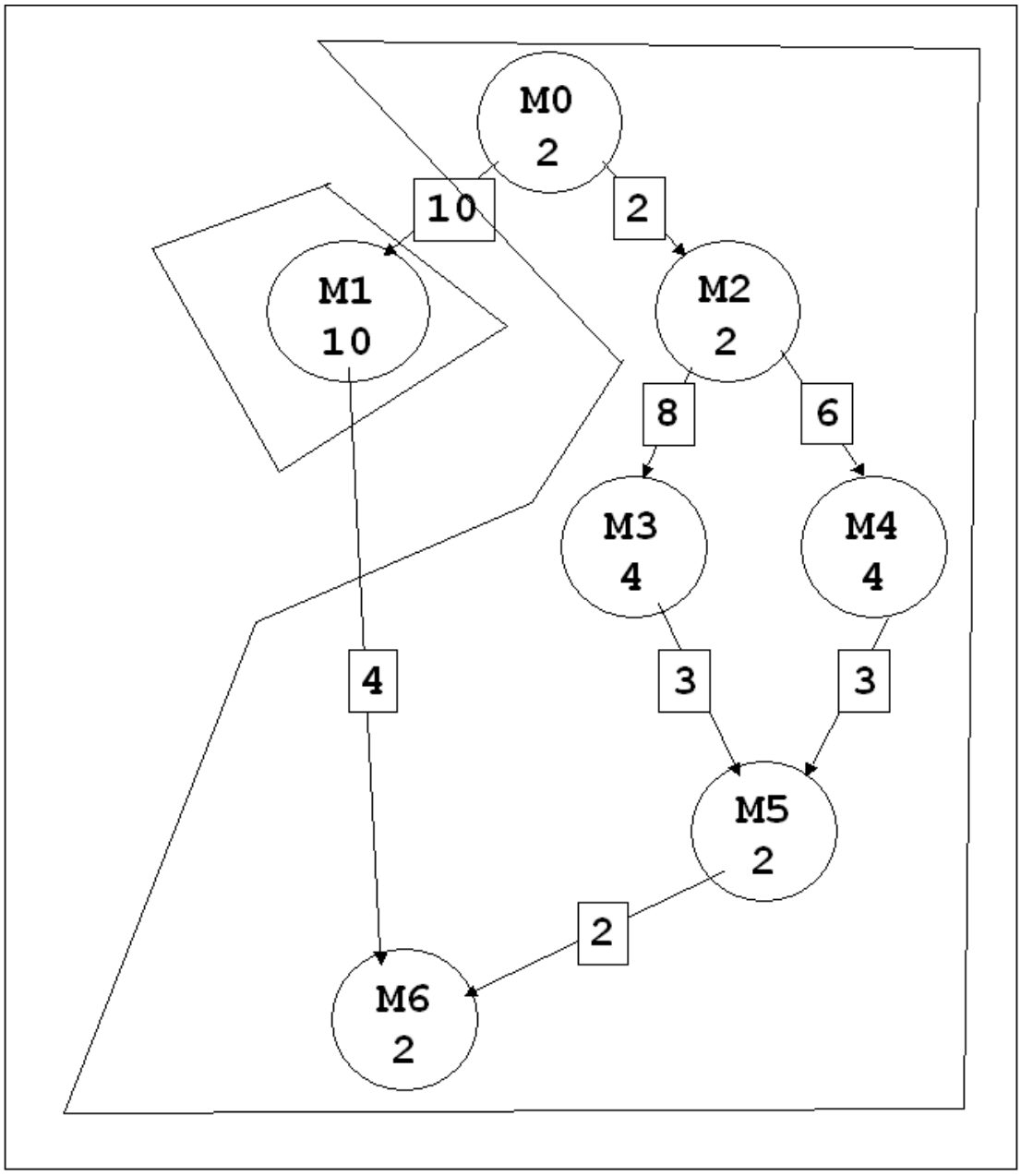

FIGURE 6.5: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 28$, $allocation = \{\{M_0, M_2, M_3, M_4, M_5, M_6\}, \{M_1\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

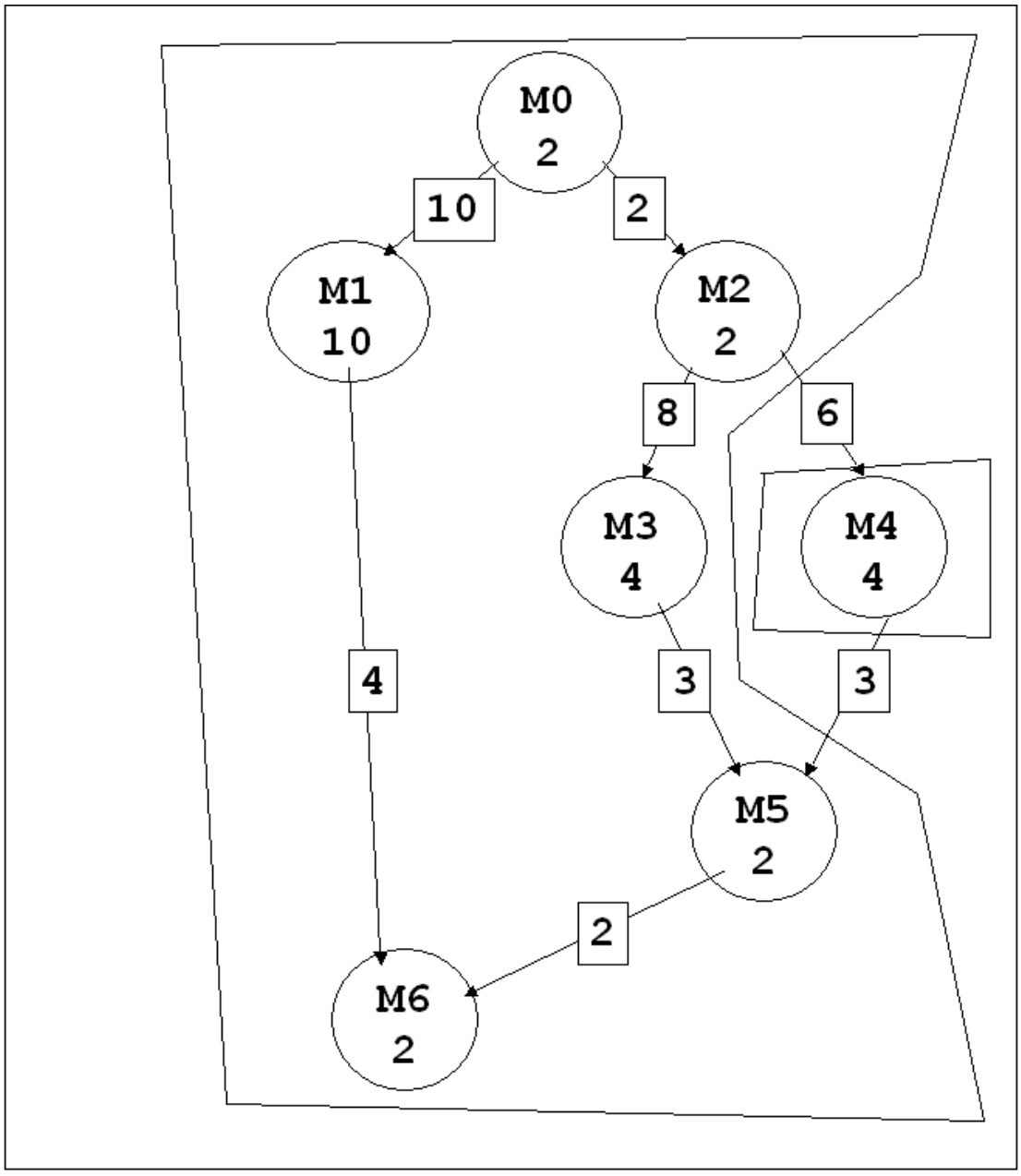

FIGURE 6.6: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 31$, $allocation = \{\{M_0, M_1, M_2, M_3, M_5, M_6\}, \{M_4\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

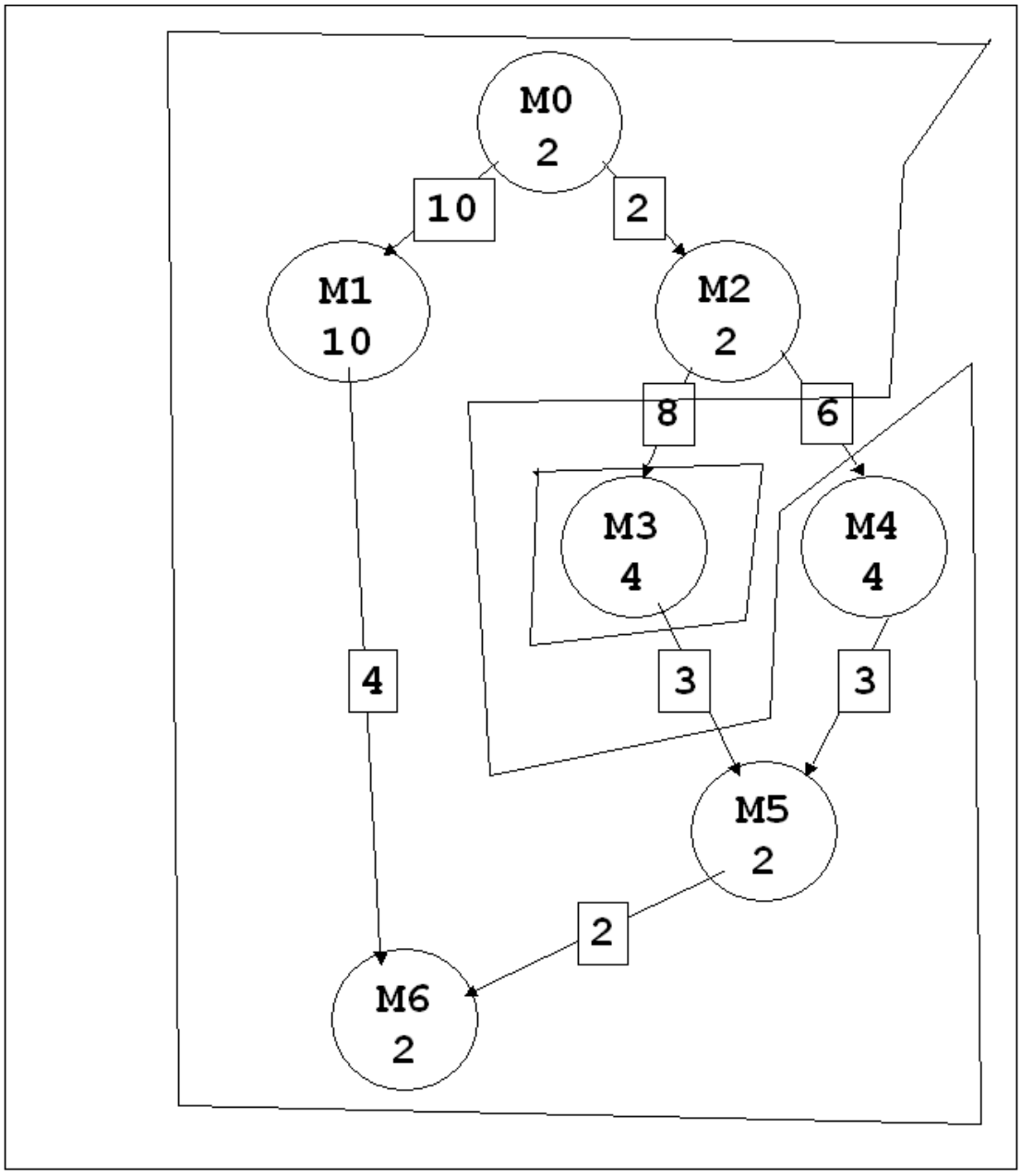

FIGURE 6.7: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 33$, $allocation = \{\{M_0, M_1, M_2, M_4, M_5, M_6\}, \{M_3\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

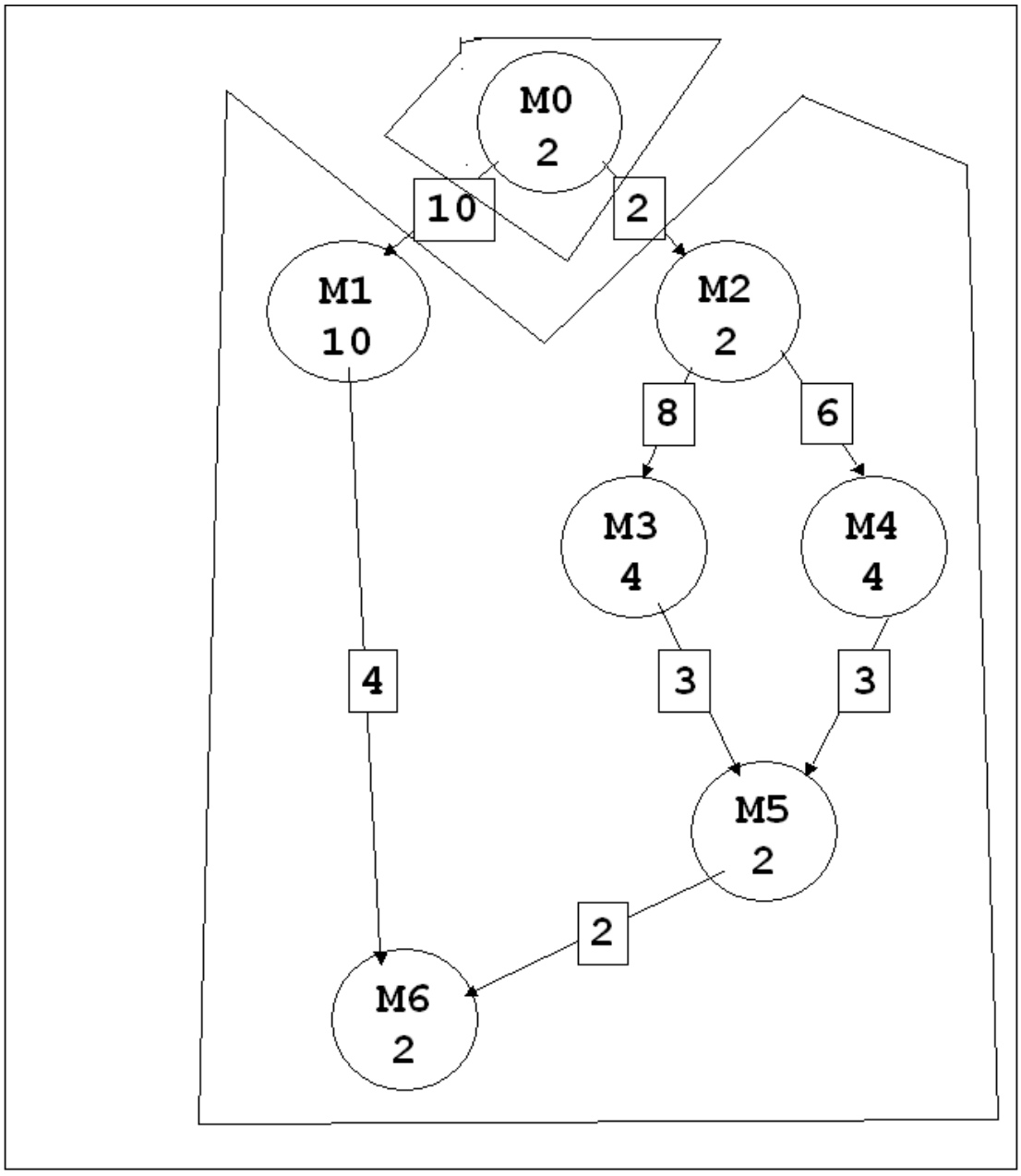

FIGURE 6.8: $min = 26$, $min\_allocation = \{\{M_0, M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $time = 28$, $allocation = \{\{M_0\}, \{M_1, M_2, M_3, M_4, M_5, M_6\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

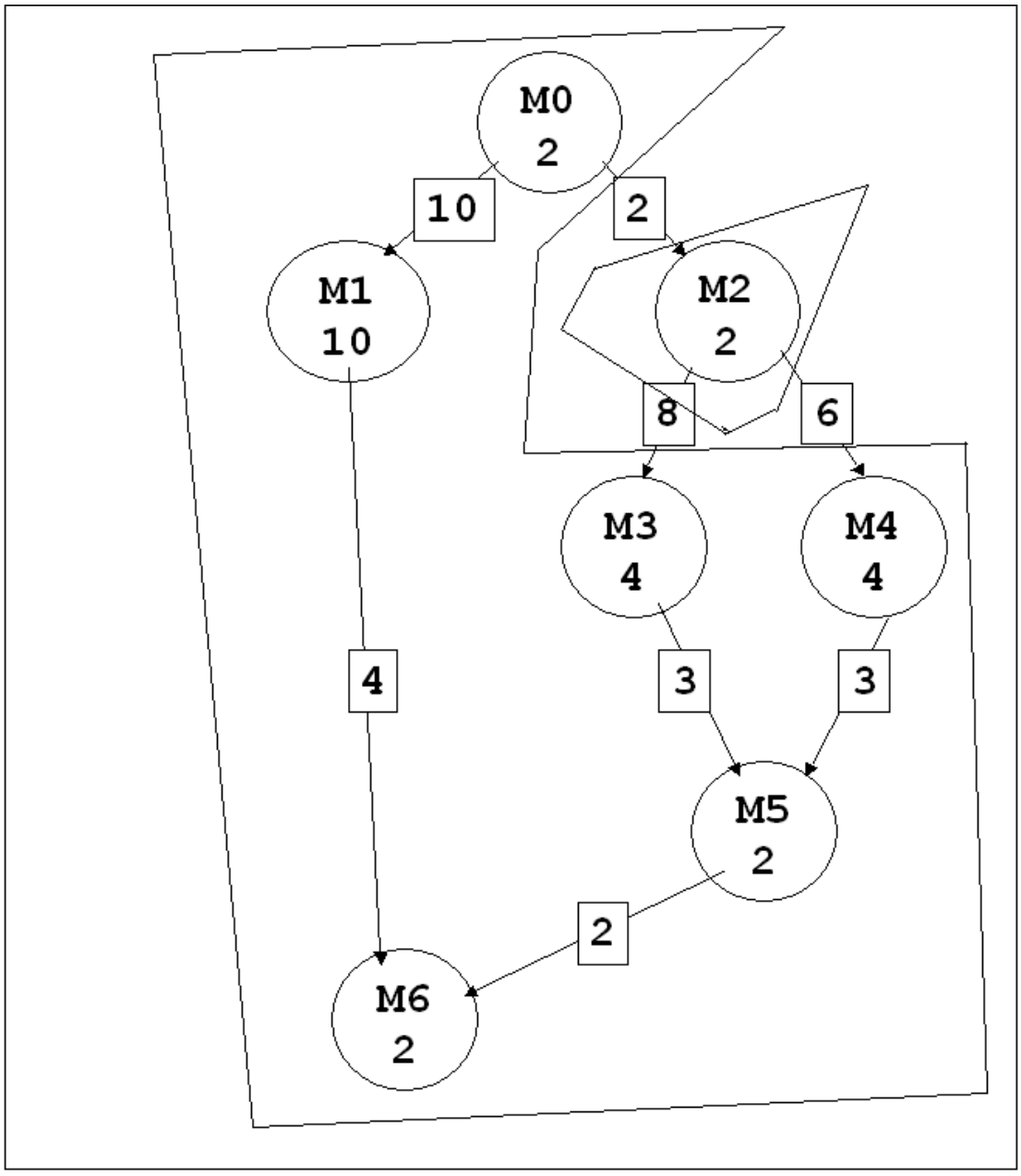

FIGURE 6.9: $min = 24$, $min\_allocation = \{\{M_0, M_1, M_3, M_4, M_5, M_6\}, \{M_2\}\}$. $time = 24$, $allocation = \{\{M_0, M_1, M_3, M_4, M_5, M_6\}, \{M_2\}\}$. $load(\{M_6,\ M_5,\ M_1,\ M_4,\ M_3,\ M_0,\ M_2\}) = \{-2, -3, -4, -5, -7, -8, -8\}$.

Chapter 7

A COMPARISON OF THE CASC ALGORITHM WITH THE CCLoadClustering ALGORITHM

7.1 A COMPARISON OF CASC WITH CCLoadClustering

For a comparison, we are using the **CASC** algorithm (Dinesh Kadamuddi and Jeffrey J.P. Tsai 2000 [1]) on the task graph given in the same paper. The **CASC** algorithm is giving a parallel execution time of 30 (FIGURE 7.1) with the clustering $\{\{M_0, M_1, M_4\}, \{M_2, M_6, M_{10}\}, \{M_3, M_7, M_{11}\}, \{M_5, M_8\}, \{M_9\}, \{M_{12}, M_{13}, M_{14}\}\}$.

The **CCLoadClustering** algorithm is giving a parallel execution time of 31 (FIGURE 7.2) with the clustering $\{\{M_0, M_1, M_4, M_5, M_7, M_8, M_9, M_{11}, M_{12}, M_{13}, M_{14}\}, \{M_2, M_3, M_6, M_{10}\}\}$. It is giving a comparable time with less number of clusters.

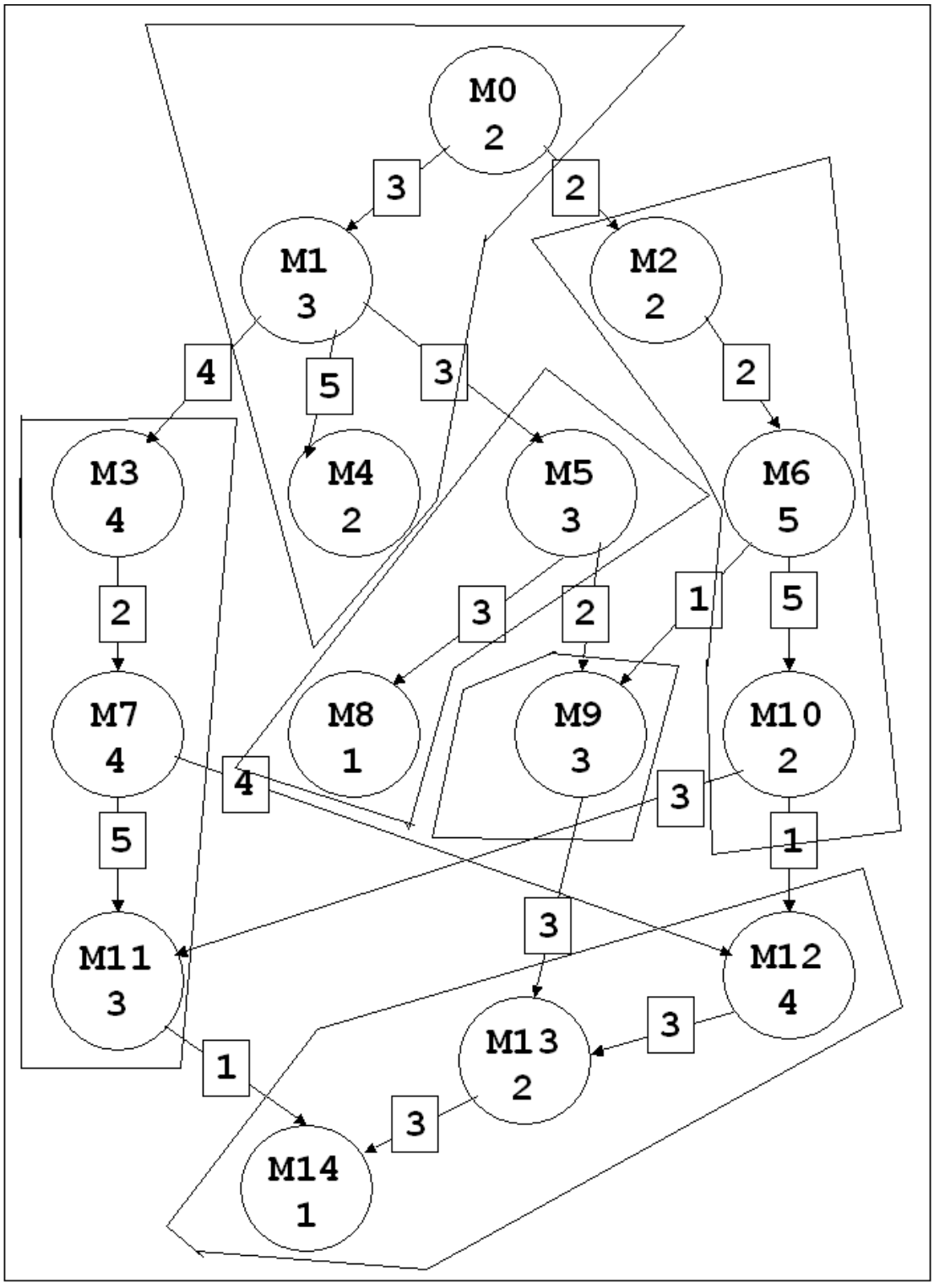

FIGURE 7.1: Clustering using the **CASC** Algorithm. $time$ = 30, $allocation$ = {{M_0, M_1, M_4}, {M_2, M_6, M_{10}}, {M_3, M_7, M_{11}}, {M_5, M_8}, {M_9}, {M_{12}, M_{13}, M_{14}}}.

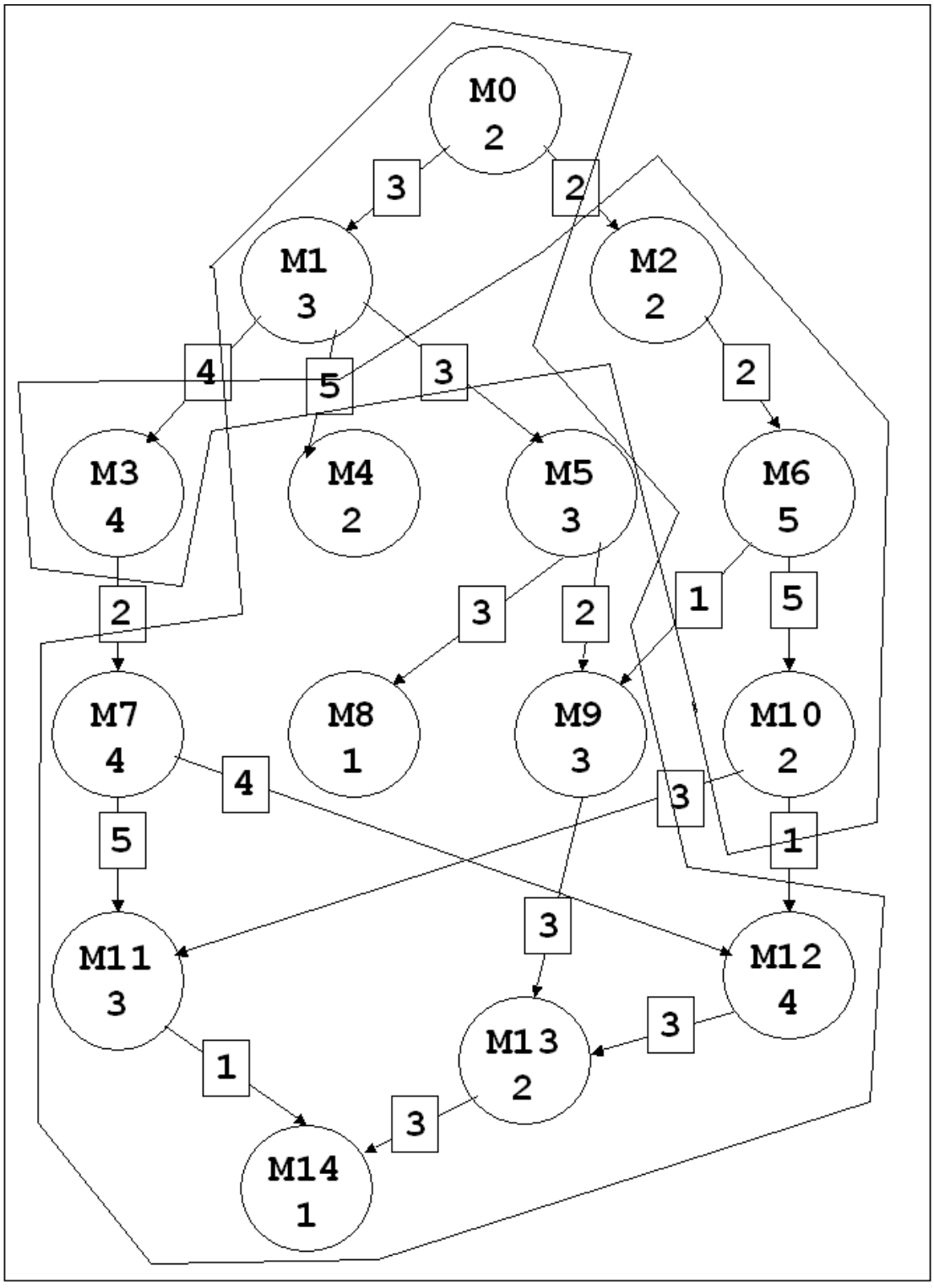

FIGURE 7.2: Clustering using the **CCLoadClustering** Algorithm. $time$ = 31, $allocation$ = {{M_0, M_1, M_4, M_5, M_7, M_8, M_9, M_{11}, M_{12}, M_{13}, M_{14}}, {M_2, M_3, M_6, M_{10}}}.

Chapter 8

EXPERIMENTAL RESULTS

8.1 A COMPARISON BETWEEN THE OptimalClustering AND THE CCLoadClustering ALGORITHMS

The **CCLoadClustering** algorithm was compared with the **OptimalClustering** algorithm for small random task graphs. Parallel execution time of **CCLoadClustering** was nearly optimal. It was only marginally greater than the parallel execution time of **OptimalClustering**. 3 sample task graphs and their results of clustering along with the parallel execution time are shown in FIGURE's 8.1 to 8.6 respectively.

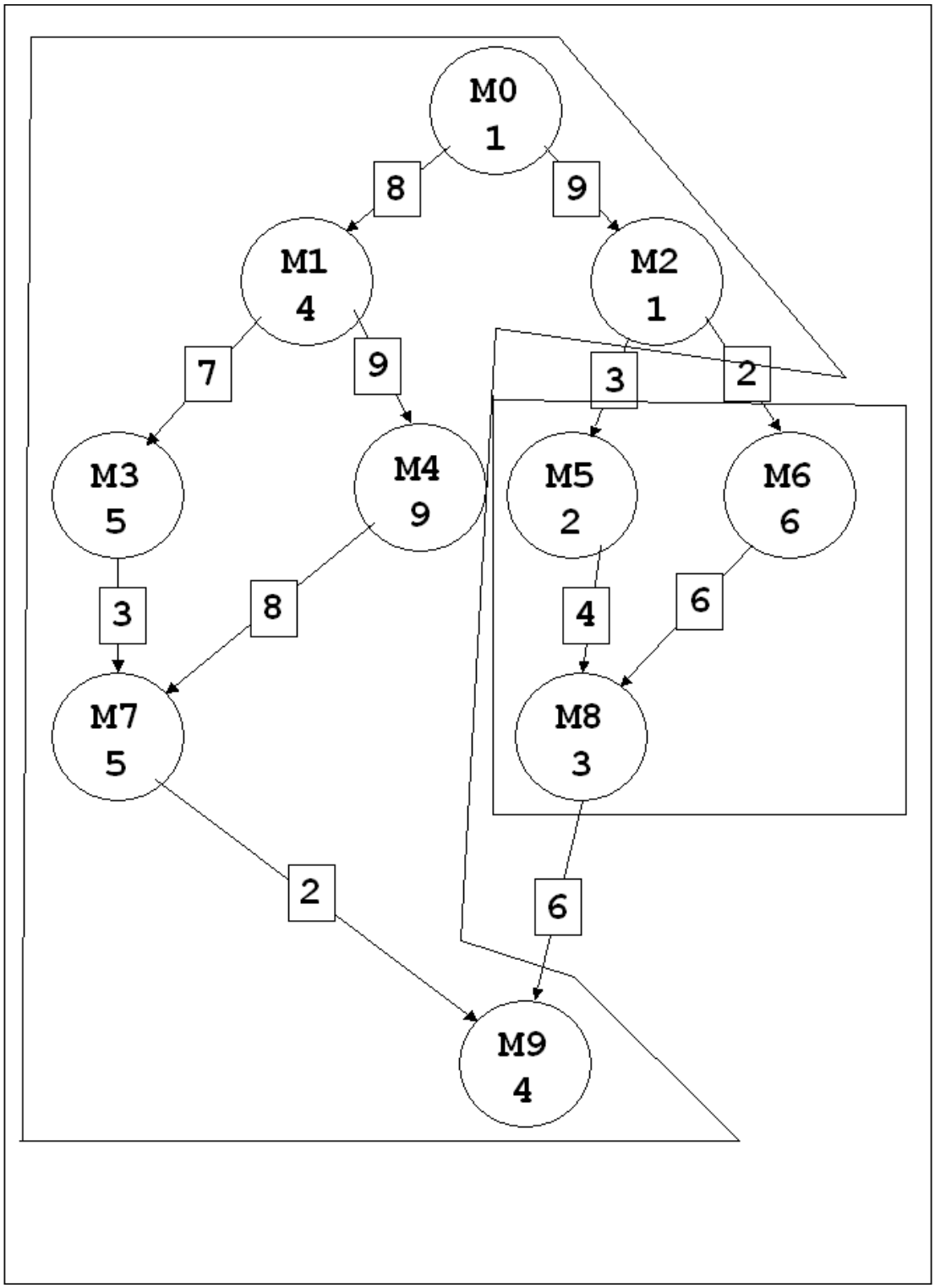

FIGURE 8.1: *Task Graph 1.* Clustering using the **OptimalClustering** Algorithm. $time$ = 30, $allocation$ = {{M_0, M_1, M_2, M_3, M_4, M_7, M_9}, {M_5, M_6, M_8}}.

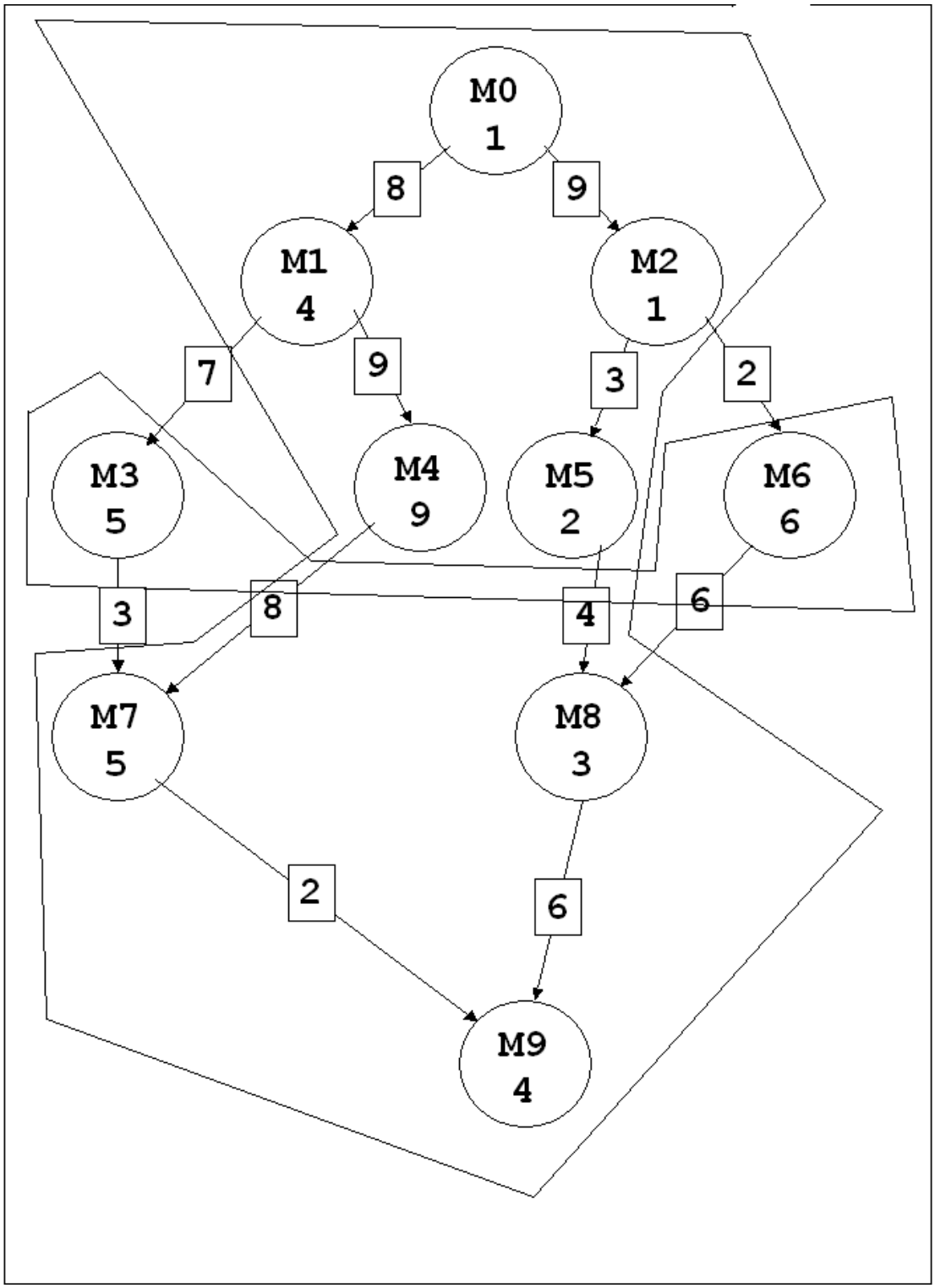

FIGURE 8.2: *Task Graph 1.* Clustering using the **CCLoadClustering** Algorithm. $time$ = 33, $allocation$ = {{M_0, M_1, M_2, M_3, M_4, M_5, M_7, M_8, M_9}, {M_3, M_6}}.

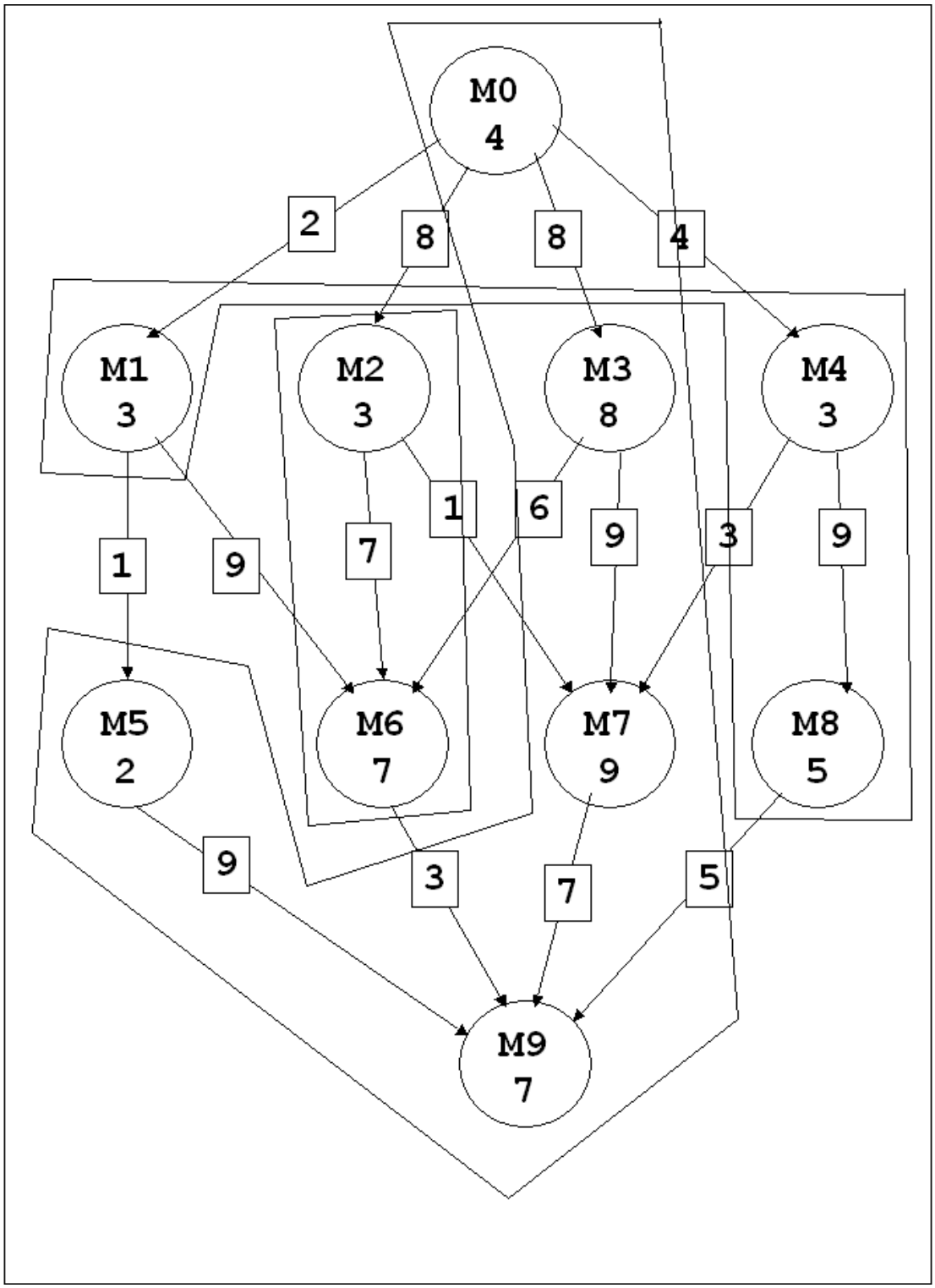

FIGURE 8.3: *Task Graph 2.* Clustering using the **OptimalClustering** Algorithm. $time$ = 35, $allocation$ = {{M_0, M_3, M_5, M_7, M_9}, {M_1, M_4, M_8}, {M_2, M_6}}.

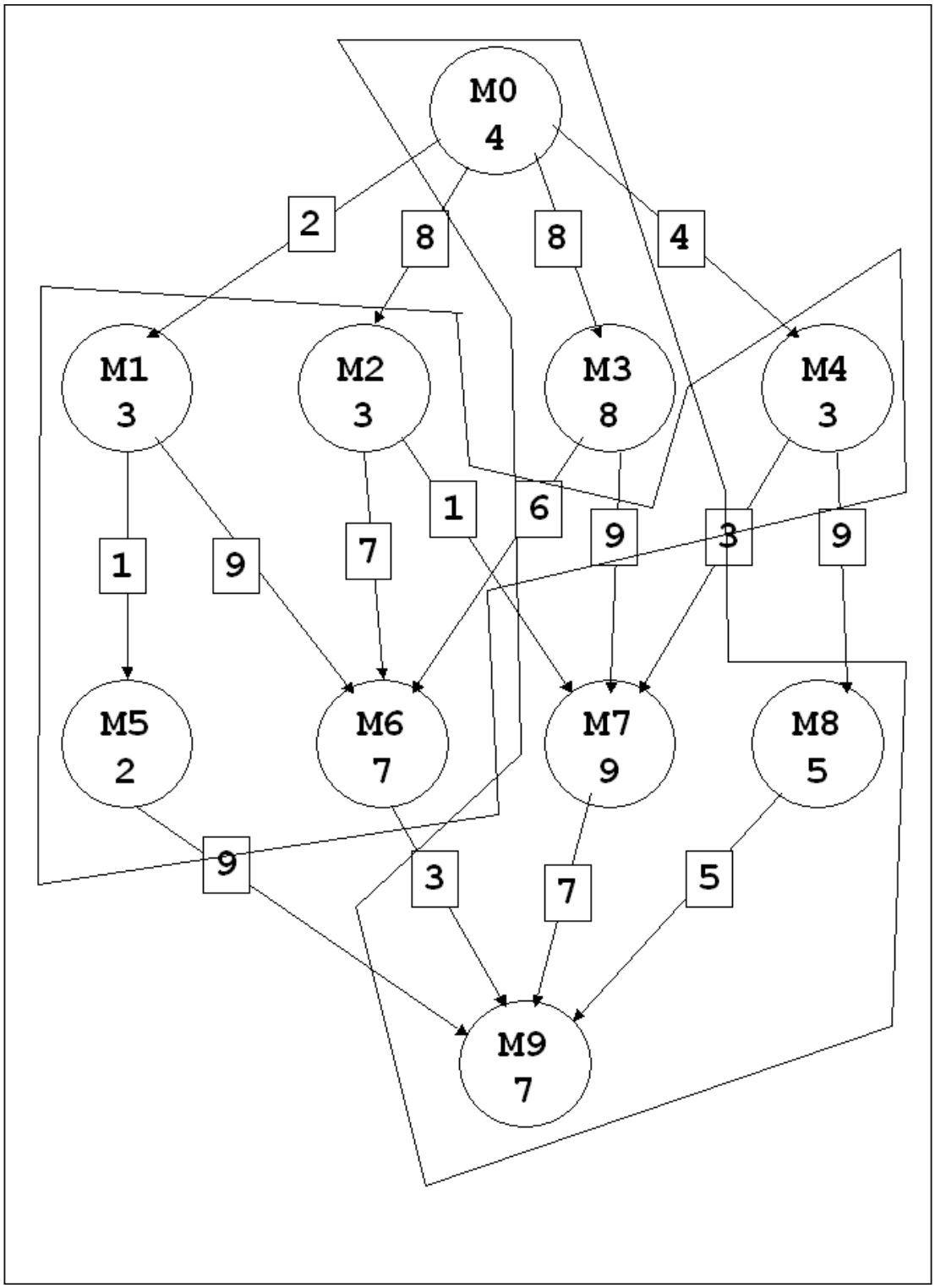

FIGURE 8.4: *Task Graph 2.* Clustering using the **CCLoadClustering** Algorithm. $time$ = 37, $allocation$ = {{M_0, M_3, M_7, M_8, M_9}, {M_1, M_2, M_4, M_5, M_6}}.

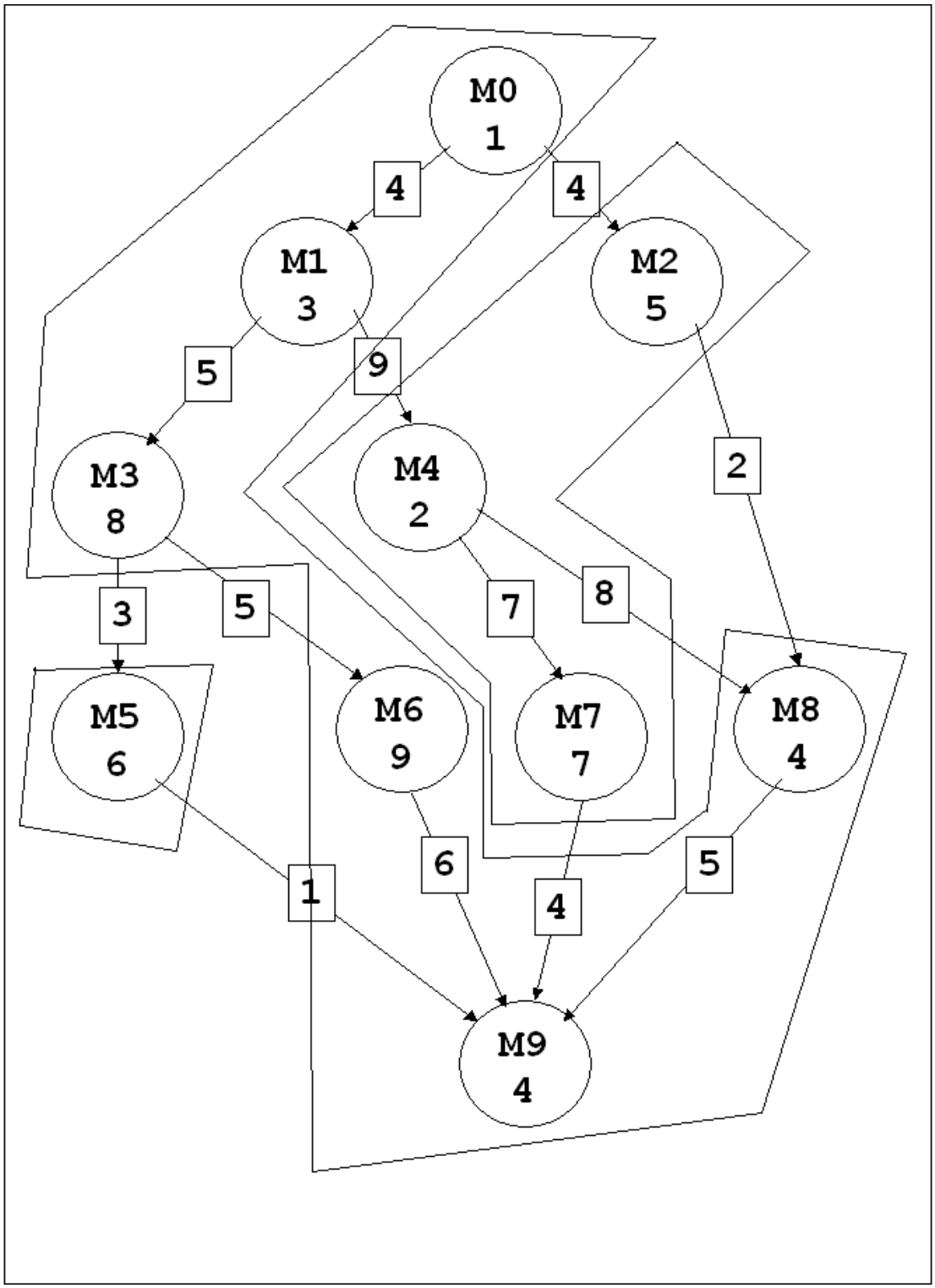

FIGURE 8.5: *Task Graph 3.* Clustering using the **OptimalClustering** Algorithm. $time$ = 36, $allocation$ = {{M_0, M_1, M_3, M_6, M_8, M_9}, {M_2, M_4, M_7}, {M_5}}.

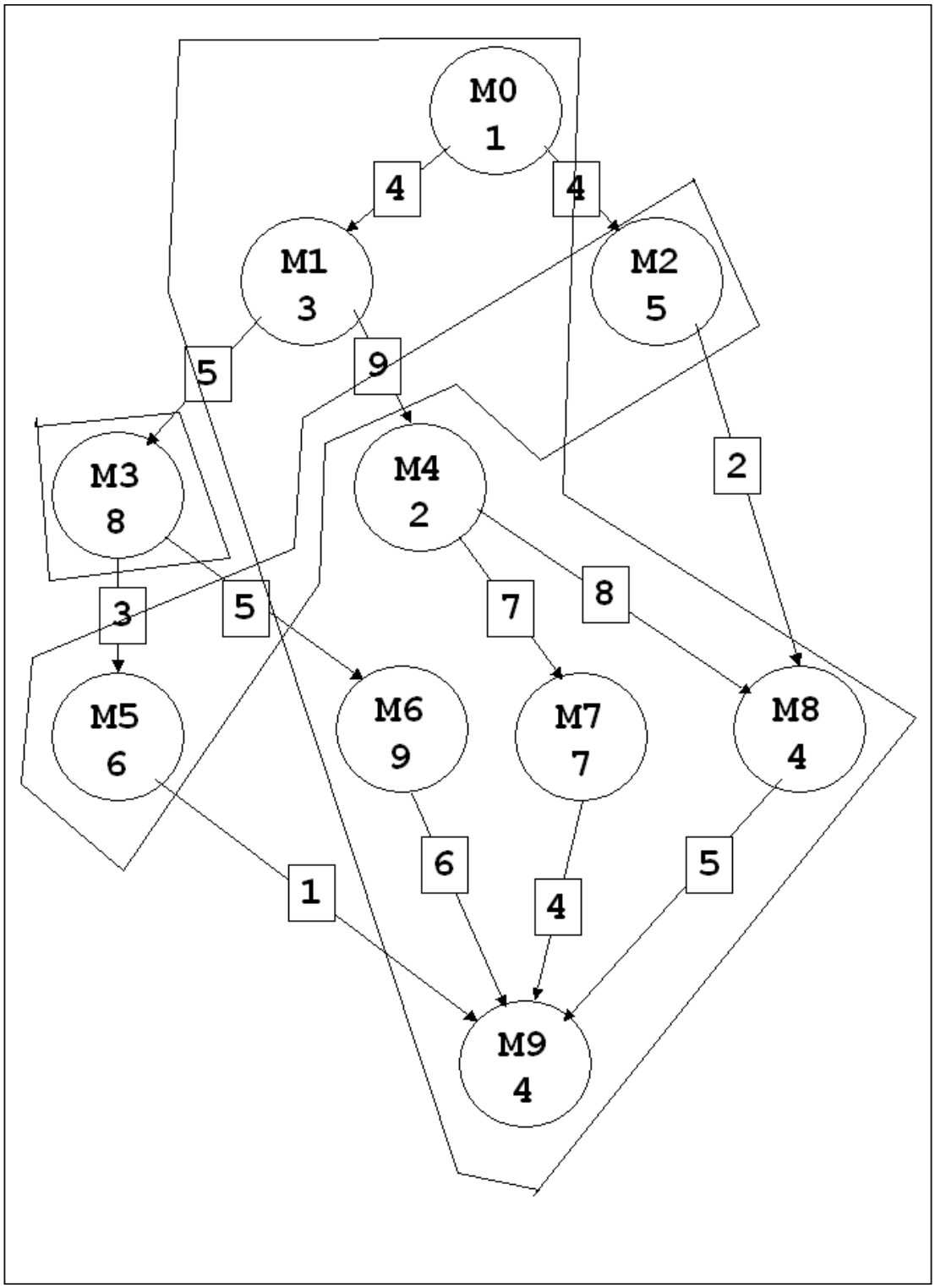

FIGURE 8.6: *Task Graph 3.* Clustering using the **CCLoadClustering** Algorithm. $time$ = 40, $allocation$ = {{M_0, M_1, M_4, M_6, M_7, M_8, M_9}, {M_2, M_5}, {M_3}}.

Chapter 9

CONCLUSION

9.1 CONCLUSION AND POSSIBLE IMPROVEMENTS

In this project a clustering algorithm (**CCLoadClustering** algorithm) for multiprocessor environment was developed using the concept of communication - computation loads of modules. This algorithm was shown to be comparable in performance to some contemporary clustering algorithms (**CASC** algorithm (Dinesh Kadamuddi and Jeffrey J.P. Tsai 2000 [1])). It was also compared with the optimal algorithm (**OptimalClustering** algorithm) for small random task graphs, and was demonstrated to give nearly optimal clusterings. As a possible improvement to the **CCLoadClustering** algorithm, we can think of reducing its time complexity ($O(V^2(V+E)^2)$).