ABSTRACT

IMPLEMENTATION OF AN AUTOMATIC MAPPING TOOL FOR MASSIVELY PARALLEL COMPUTING

by
Ajitha Gadangi

In this thesis, an implementation of a generic technique for fine grain mapping of portable parallel algorithms onto multiprocessor architectures is presented. The implemented mapping algorithm is a component of Cluster-M. Cluster-M is a novel parallel programming tool which facilitates the design and mapping of portable softwares onto various parallel systems. The other components of Cluster-M are the Specifications and the Representations. Using the Specifications, machine independent parallel algorithms are presented in a "clustered" fashion specifying the concurrent computations and communications at every step of the overall execution. The Representations, on the other hand, are a form of clustering the underlying architecture to simplify the mapping process. The mapping algorithm implemented and tested in this thesis is an efficient method for matching the Specification clusters to the Representation clusters.

IMPLEMENTATION OF AN AUTOMATIC MAPPING TOOL FOR MASSIVELY PARALLEL COMPUTING

by
Ajitha Gadangi

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

January 1994

APPROVAL PAGE

IMPLEMENTATION OF AN AUTOMATIC MAPPING TOOL FOR MASSIVELY PARALLEL COMPUTING

Ajitha Gadangi

Dr. Mary M. Eshaghian, Thesis Advisor Assistant Professor of Computer and Information Science, NJIT	Date
Dr. Daniel Y. Chao, Committee Member Assistant Professor of Computer and Information Science, NJIT	Date
Dr. David Wang, Committee Member Assistant Professor of Computer and Information Science, NJIT	Date

BIOGRAPHICAL SKETCH

Author: Ajitha Gadangi

Degree: Master of Science in Computer Science

Date: January 1994

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1994

- Bachelor of Science in Computer Science
  Jawaharlal Nehru Technological University, Hyderabad, India, 1992

Major: Computer Science

This thesis is dedicated to my beloved husband

ACKNOWLEDGMENT

The author wishes to express her sincere gratitude to her supervisor, Dr. Mary M. Eshaghian for her guidance, friendship, and moral support throughout this research.

Special thanks to Dr. Daniel Chao and Dr. David Wang for serving as members of the committee. The author is grateful to the Department of Computer and Information Science for partially funding this research.

The author appreciates the timely help and suggestions from the project group team members Phil Chen, Ying-Chieh Jay Wu ,Geetha Chitti and Javier G. Vasquez.

The author is very much grateful to Ravindra K. Gadangi, without his assistance this thesis would not have been in this format. The author also wishes to thank Annette Damiano for her professional comments in finalizing this thesis.

Last but not the least the author wishes to express her deep appreciation to her husband Rajendra K. Gadangi for all his love and support and for making this day possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Introduction

An efficient parallel algorithm designed for a parallel architecture includes a detailed outline of the accurate assignments of the concurrent computations onto processors, and the data transfers onto communication links, such that the overall execution time is minimized. This process may be complex for many application tasks with respect to the target multiprocessor architecture. Furthermore, this process is to be repeated for every architecture even though the application task may be the same. Consequently, this has a major impact on the ever increasing cost of software development for multiprocessor systems. In this thesis, we concentrate on the design of portable parallel algorithms and present a methodology for fine grain mapping of these algorithms onto various parallel machines.

Cluster-M, a novel parallel programming tool introduced recently, facilitates the design and mapping of portable softwares onto various multiprocessor systems [7]. Portable algorithms are specified in Cluster-M format in a way that represent concurrent computations and communications at every step of the overall execution. To map the Cluster-M Specification onto the target architecture, the processors of the underlying system are clustered in a hierarchical fashion such that all those in the same cluster have efficient communication medium. The mapping methodology outlined previously specifies a direction towards good matching of the concurrent tasks (Specification clusters) and interconnected processors (Representation clusters) [8]. In this thesis, we present an efficient algorithm for fine grain mapping of Specifications onto Representations.

A number of other parallel programming tools have also been developed recently, which provide an environment for design and automatic mapping of

portable algorithms [1, 2, 13, 17]. These tools can be classified into two groups. The first group uses a library of pre-defined routines for mapping [1, 13]. In the second category, the mapping is determined based on a graph matching technique. The mapping problem here is the same as the classic one defined and studied by several researchers over the years [15, 3, 12, 4, 6, 14]. The input to the mapping problem is two graphs. The first graph is called the problem graph which is similar to the data flow representation of the execution process, where each node is a computation task and edges represent dependency and flow of data. The second graph is called the system graph which is a trivial representation of the underlying architecture. The mapping problem is defined as the matching of these two graphs such that the overall execution time is minimized. This problem has been proven to be computationally equivalent to the graph isomorphism problem and hence is an NP-complete optimization problem [3].

To reduce the complexity of the mapping problem, a number of approaches such as graph contraction and clustering have been studied [5, 2, 11, 16, 17, 14]. In graph contraction, a pair of connected nodes are merged into a supernode [2, 14], while in clustering, a set of connected nodes are merged into one new node [5, 11, 16, 17]. This process continues until a graph with desired order and pattern is reached. In all these graph matching based techniques, the entire problem graph is considered against the entire system graph. The important observation that can be made here, is that it is not necessary to map all the steps of an algorithm which is present in a problem graph, onto the system graph all at once. An algorithm represents a step by step procedure for solving a problem. Therefore, it is sufficient to map one step of an algorithm at a time onto available processors. However, this assignment should be made intelligently so that it minimizes the total execution time of the future steps and the overall execution time. For this reason, in this thesis, we present an implementation of a mapping algorithm which not only considers the problem

graph in a layered (clustered) fashion, but also layers the target system graph in a clustered form. This Cluster-M based approach is the process of finding a good matching between the two sets of clusters.

The rest of the thesis is organized as follows. In chapter 2, we introduce the components of Cluster-M. The mapping algorithm being implemented is detailed in chapter 3. An implementation of this algorithm and some experiment results are shown in chapter 4. A brief conclusion is given in chapter 5.

CHAPTER 2

CLUSTER-M MODEL

2.1 Cluster-M Model

Cluster-M has three main components: Cluster-M Specifications, Cluster-M Representations and Cluster-M mapping module [7, 8, 9]. In this chapter, we show how to generate a layered problem graph with Cluster-M Specifications and a layered system graph with Cluster-M Representations. The mapping module generates an efficient mapping of the Specification graph onto the Representation graph.

2.1.1 Cluster-M Specifications

A Cluster-M Specification of a problem is a high level machine independent program that specifies the computation and communication requirements of a solution to a given problem. A Cluster-M Specification can be translated into a graph consisting of multiple levels of clustering. In each level, there is a number of clusters representing concurrent computations. Clusters are merged when there is a need for communication among concurrent tasks. For example, if all n elements of an array are to be squared, each element is placed in a cluster, then the Cluster-M specification would state:

For all n clusters, square the contents.

Figure 2.1 *Cluster-M Specification graph of a unary operation on array of size n*

Note, that since no communication is necessary, there is only one level in the Cluster-M Specification graph as shown in Figure 2.1. The mapping of this Specification to any architecture having n processors would be identical.

The basic operations on the clusters and their contained elements are performed by a set of constructs which form an integral part of the Cluster-M model. For a complete listing and description of these constructs which are essential for writing Cluster-M Specifications, refer to [8, 9]. All these constructs have been implemented in PCN [9, 10]. Below we show an example for computing the associative binary operation $*$ of N elements of vector A, using the constructs implemented in PCN. The resulting Cluster-M specification will be as follows, where $CMAKE$, $CMERGE$ and CBI are Cluster-M specification constructs. The Cluster-M Specification graph of this example is shown in Figure 2.2.

```
ASSOC_BIN(op, N, A, Z) /* op: operation, Z: return value */
int N, A[ ];
{ ; lvl = 0,
   make_tuple(N, cluster),
   { ; i over 0 .. N-1 ::
      { ; CMAKE(lvl, [A[i]], c),
         cluster[i] = c
      }
   } ,
   Binary_Op(cluster, N, op, Z)
}

Binary_Op(X, N, op, B)
int N, n;
{ ? N > 1 -> { ; n := N / 2,
                  make_tuple(n, Y),
```

```
                { ; i over 0 .. n-1 ::
                    { ; BI_MERGE(op, X[2 * i], X[2 * i + 1], Z),
                        Y[i] = Z
                    }
                } ,
                Binary_Op(Y, n, op, B)
            } ,
    default -> B = X
}

BI_MERGE(op, X1, X2, M)
int e;
{ ; CBI(op, X1, 1, X2, 1, e),
    CMERGE(X1, X2, [e], M)
}
```

2.1.2 Cluster-M Representations

For every architecture, at least one corresponding Cluster-M Representation graph can be constructed. Cluster-M Representation of an architecture is a multi-level nested clustering of processors. To construct a Cluster-M Representation, initially, every processor forms a cluster, then clusters which are completely connected are merged to form a new cluster. This is continued until no more merging is possible. In other words, at level LVL of clustering, there are multiple clusters such that each cluster contains a collection of clusters from level $LVL+1$ which form a clique. The highest level consists of only one cluster, if there exists a connecting sequence of communication channels between any two processors of the system. A Cluster-M Representation is said to be *complete* if it contains all the communication channels and all the processors of the underlying architecture. For example, the Cluster-M Representation of the n-cube architecture is as follows: At the lowest level $n+1$,

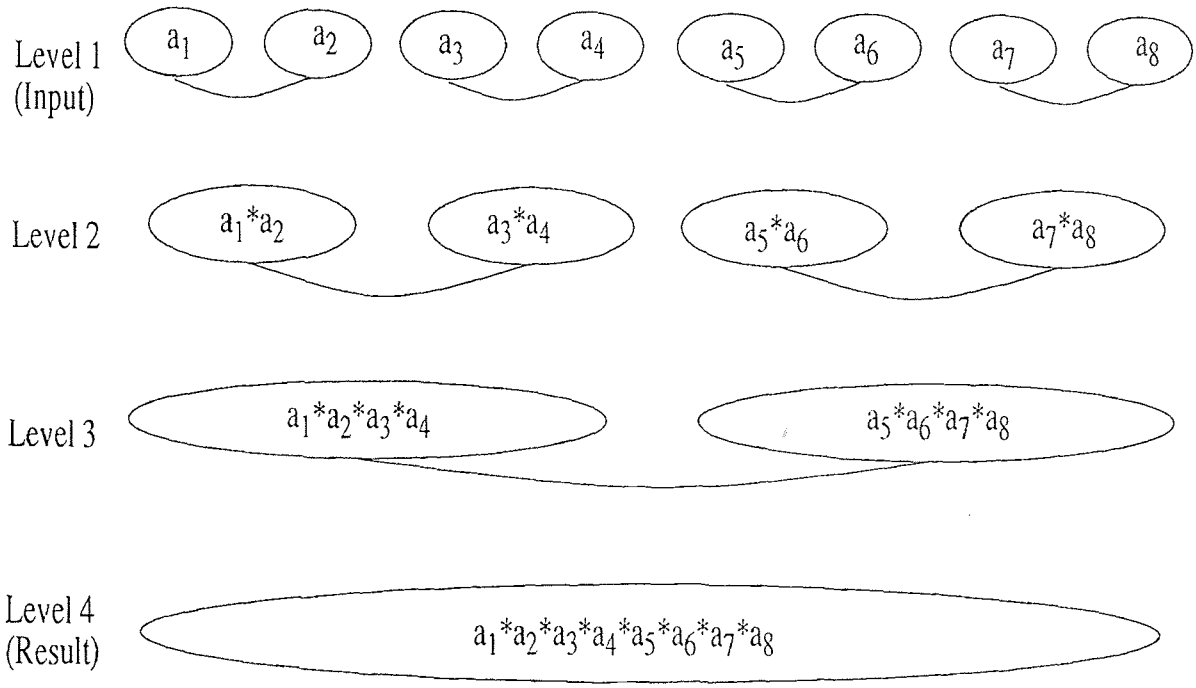

Figure 2.2 *Cluster-M Specification of associative binary operation.*

every processor belongs to a cluster which contains just it self. At level n, every two processors (clusters) which are connected are merged into the same cluster. At level $n-1$, clusters of previous level which are connected belong to the same cluster, and so on until level 1. The complete Cluster-M Representation of a 3-cube, and of a system with arbitrary interconnections are shown in Figures 2.3 and 2.4, respectively.

An algorithm for generating a Cluster-M Representation for any given architecture has been presented and implemented in [9]. The algorithm has a running complexity of $O(N^3)$ where N is the number of processors.

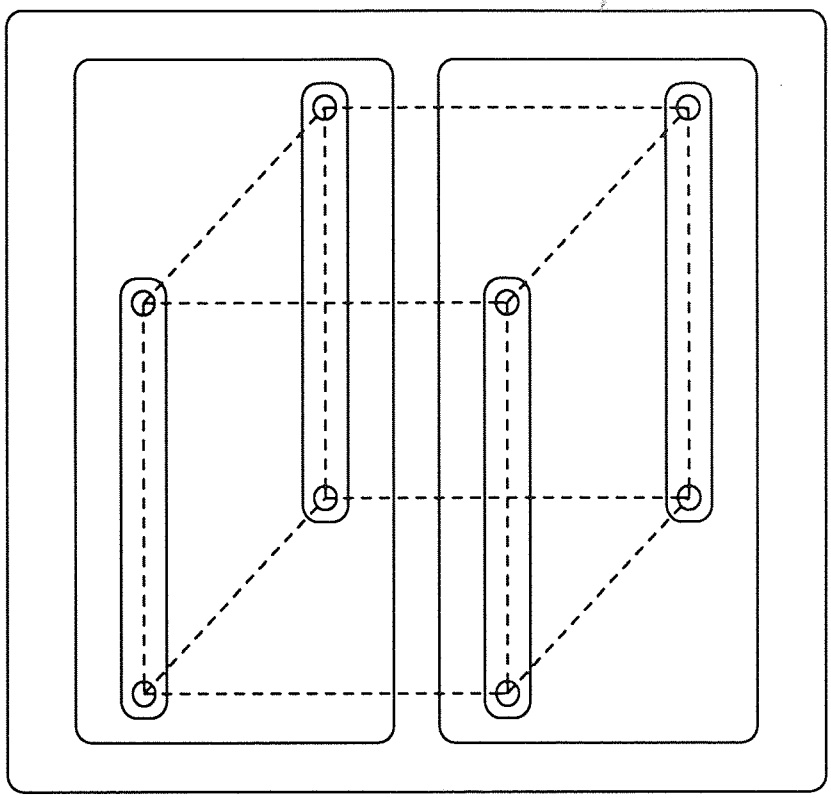

Figure 2.3 *Cluster-M Representation of n-cube of size 8.*

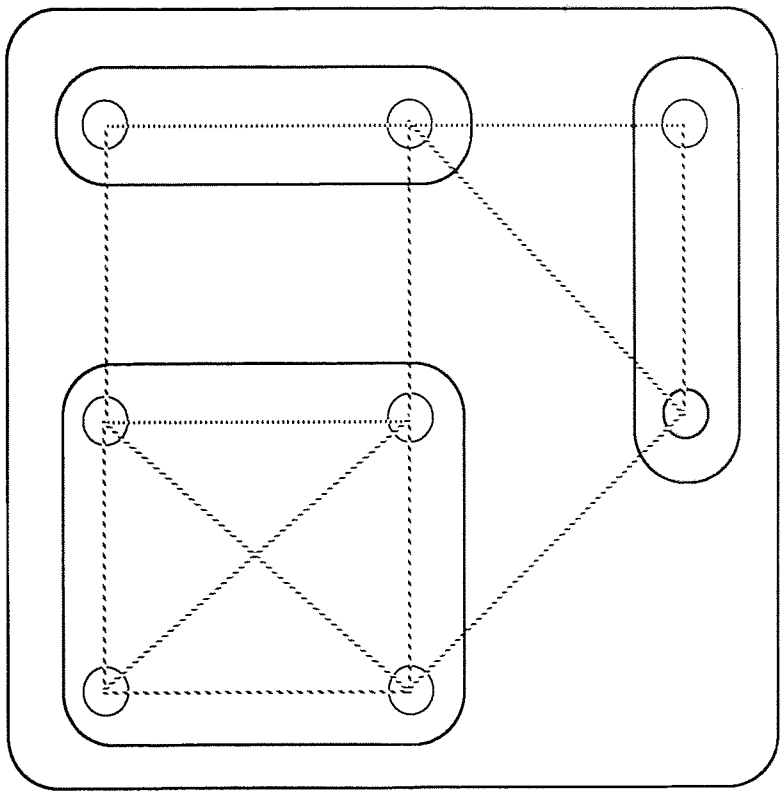

Figure 2.4 *Cluster-M Representation of an arbitrarily connected system of size 8.*

CHAPTER 3

CLUSTER–M MAPPING ALGORITHM

3.1 Cluster–M Mapping Algorithm

Given a Specification graph and a Representation graph as the input to the mapping module, the process continues as explained in this section. The mapping procedure presented in this thesis has a much lower time complexity than the traditional mappings since it contains a graph matching procedure which considers the input graphs level by level. In this chapter , we first present a set of definitions and preliminaries, Then in 3.2 we present a high level description of the mapping algorithm.

3.1.1 Preliminaries

Definition 3.1

- Let a Specification cluster at level LVL be denoted by $\kappa_S[i_1, i_2, \cdots, i_{LVL}]$, where i_{LVL} is the cluster number at level LVL and i_l $(1 \leq l \leq LVL-1)$ is the cluster number of its parent cluster at level l.

- If a Specification cluster $\kappa_S[i_1, i_2, \cdots, i_{LVL}]$ can not be further decomposed into sub-clusters, i.e. if this cluster is corresponding to a fine grain sub-task T_k. let $\kappa_S[i_1, i_2, \cdots, i_{LVL}] = T_k$.

Definition 3.2

- Let a Representation cluster at level LVL be denoted by $\kappa_R[i_1, i_2, \cdots, i_{LVL}]$, where i_{LVL} is the cluster number at level LVL and i_l $(1 \leq l \leq LVL-1)$ is the cluster number of its parent cluster at level l.

- If a Representation cluster $\kappa_R[i_1, i_2, \cdots, i_{LVL}]$ can not be further decomposed into sub-clusters, i.e. if this cluster is corresponding to a processor p_j, let $\kappa_S[i_1, i_2, \cdots, i_{LVL}] = p_j$.

Definition 3.3

- Let the computation requirement of Specification cluster $\kappa_S[i_1, i_2, \cdots, i_{LVL}]$ be denoted by $\sigma_S[i_1, i_2, \cdots, i_{LVL}]$.
- The computation requirement of any fine grain subtask T_k, $\sigma_S[T_k]$, is specified in Problem Specifications.
- For any cluster $\kappa_S[i_1, i_2, \cdots, i_{LVL}]$ which contains sub-clusters at a lower level $LVL+1$, $\sigma_S[i_1, i_2, \cdots, i_{LVL}] = \sum_i \sigma_S[i_1, i_2, \cdots, i_{LVL}, i]$.

Definition 3.4

- Let the computation capacity of Representation cluster $\kappa_R[i_1, i_2, \cdots, i_{LVL}]$ be denoted by $\sigma_R[i_1, i_2, \cdots, i_{LVL}]$.
- The computation capacity of any processor p_j, $\sigma_R[p_j]$, is given.
- For any cluster $\kappa_R[i_1, i_2, \cdots, i_{LVL}]$ which contains sub-clusters at a lower level $LVL+1$, $\sigma_R[i_1, i_2, \cdots, i_{LVL}] = \sum_i \sigma_R[i_1, i_2, \cdots, i_{LVL}, i]$.

Definition 3.5

- Let the clustering degree of Specification cluster $\kappa_S[i_1, i_2, \cdots, i_{LVL}]$ be denoted by $\delta_S[i_1, i_2, \cdots, i_{LVL}]$, and defined to be the the number of levels down to its deepest sub-cluster $\kappa_S[i_1, i_2, \cdots, i_{LVL}, \cdots, i_{LVL^\delta}]$, i.e. $\delta_S[i_1, i_2, \cdots, i_{LVL}] = LVL^\delta - LVL$, where for any sub-clusters $\kappa_S[i_1, i_2, \cdots, i_{LVL}, \cdots, i_{LVL^{\delta'}}]$, $LVL^{\delta'} \leq LVL^\delta$.
- The clustering degree of a fine grain subtask is 0.

Definition 3.6

- Let the clustering degree of Representation cluster $\kappa_R[i_1, i_2, \cdots, i_{LVL}]$ be denoted by $\delta_R[i_1, i_2, \cdots, i_{LVL}]$, and defined to be the the number of levels down to its deepest sub-cluster $\kappa_R[i_1, i_2, \cdots, i_{LVL}, \cdots, i_{LVL^\delta}]$, i.e. $\delta_R[i_1, i_2, \cdots, i_{LVL}] = LVL^\delta - LVL$, where for any sub-clusters $\kappa_R[i_1, i_2, \cdots, i_{LVL}, \cdots, i_{LVL^{\delta'}}]$, $LVL^{\delta'} \leq LVL^\delta$.

- The clustering degree of a processor is 0.

A cluster of less clustering degree has more communication requirement/capacity than a cluster with the same computation requirement/capacity.

According to the above definitions of clusters, we have the following propositions.

Proposition 3.1

- Specification clusters $\kappa_S[i_1, i_2, \cdots, i_{LVL}, i]$ and $\kappa_S[i_1, i_2, \cdots, i_{LVL}, j]$ have communication need.

- Representation clusters $\kappa_R[i_1, i_2, \cdots, i_{LVL}, i]$ and $\kappa_R[i_1, i_2, \cdots, i_{LVL}, j]$ have communication links.

Proposition 3.2

- $\delta_S[i_1, i_2, \cdots, i_l] \geq \delta_S[i_1, i_2, \cdots, i_l, \cdots, i_{l+m}] + m$.

- $\delta_R[i_1, i_2, \cdots, i_l] \geq \delta_R[i_1, i_2, \cdots, i_l, \cdots, i_{l+m}] + m$.

Definition 3.7

- Let S be the total computation requirement of the whole task, i.e. $S = \sum_i \sigma_S[i]$.

- Let R be the total computation capacity of the whole system, i.e. $R = \sum_i \sigma_R[i]$.

- Let f be the reduction factor which indicates how much Specification computation is to be mapped onto a Representation processor. $f = R/S$. Therefore, if $f > 1$, then the computation capacity of the system is greater than what is required to solve the problem as outlined in the Cluster-M Specification. Otherwise, $1/f$ of the computations specified are to be mapped onto each of the computational units represented.

Definition 3.8

The measure of mapping quality we use can be formulated as:

$$|f_m| = F_\sigma * \sum_i |f \times \sigma_S[i] - \sigma_R[f_m(\kappa_S[i])]| + F_\delta * \sum_i g(\delta_S[i] - \delta_R[f_m(\kappa_S[i])]) \quad (3.1)$$

where f_m is the mapping function for Specification cluster at top level, and g is a function defined as $g(x) = 1$ if $x < 0$ and $g(x) = 0$ if $x \geq 0$. F_σ and F_δ are the favor factor for computation and communication matching respectively ($F_\sigma \geq 0$, $F_\delta \geq 0$). The best mapping is the one with the minimum $|f_m|$.

3.1.2 Mapping Algorithm

Given a Specification graph to be mapped onto a Representation graph, the mapping procedure starts at the top layer (level) of the Specification graph. To map every Specification cluster $\kappa_S[i]$ at the top level, onto a Representation cluster, we search for the best matched Representation cluster with a computation capacity closest to $f \times \sigma_S[i]$ and a clustering degree equal to or less than $\delta_S[i]$.

When the mapping at top level is done, for each pair of the mapped Specification and Representation clusters, the same mapping procedure is continued (recursively) at a lower level until the mapping is fine grained to the processor level. A high level description of the mapping algorithm is given below.

1. Sort all $\kappa_R[i_1, \cdots, i_{LVL}]$ in descending order of the value of $\sigma_R[i_1, \cdots, i_{LVL}]$.

2. Sort $\kappa_S[i]$ in descending order of the value of $\sigma_S[i]$.

3. Calculate S, R and f. If $f > 1$, let $f = 1$. Calculate the required computation capacity of $\kappa_S[i]$ to be $f \times \sigma_S[i]$.

4. Find a virtual Representation layer consisting of non-overlapping clusters such that when the Specification clusters at the current lavel are mapped onto these clusters, the measure of quality $|f_m|$ is minimized.

5. For each pair of $\kappa_S[k]$ and $\kappa_R[i_1, i_2, \cdots, i_{LVL_k}]$
   $= f_m(\kappa_S[k])$, if $\kappa_R[i_1, i_2, \cdots, i_{LVL_k}]) = p_j$, then stop. Otherwise
   let $\kappa_S[i_2, \cdots, i_l] = \kappa_S[i, i_2, \cdots, i_l]$, and $\kappa_R[i_2, \cdots, i_{LVL_k}, \cdots, i_l] =$
   $\kappa_R[i_1, i_2, \cdots, i_{LVL_k}, \cdots, i_l]$ for any existing l, and go to step 2.

The total time complexity of this algorithm is analyzed as follows. For each level except step 4, the complexity of step 2 to 5 is dominated by the complexity of sorting which can be done trivially in $O(K^2)$ time sequentially for K inputs, where K is the number of Specification clusters at current level. An optimal solution for step 4 can have an exponential time complexity. However, since the number of clusters being mapped every level is usually constant, it leads to an average linear time performance. The step 2 to 5 may be repeated for J iterations, where J is the number of nested levels in the Cluster-M Specification graph.

CHAPTER 4

IMPLEMENTATIONS AND EXPERIMENTAL RESULTS

We have implemented the algorithm described above in C language under UNIX environment. In our implementation, we have used a heuristic for finding f_m in step 4, which has a time complexity of $O(KN)$, where K is the number of Specification clusters at current level, and the total number of clusters in the Representation graph is $O(N)$, where N is the number of processors. The total time complexity of this entire implementation, steps 1 to 5, is $O(T^2)$, where T is $max\{M, N\}$. The pseudo code of the implementation is shown in below.

```
program mapping;
var Spec: entire problem specification;
    Rep: entire system representation;
    {Spec and Rep are represented by a set of
    multi-level lists}
begin
    sort all the clusters of Spec and Rep at each level;
    map(Spec, Rep)
end.


procedure map(Spec, Rep); {recursive mapping
                                procedure}
var
    S: integer; {total computation capacity of Spec}
    R: integer; {total computation capacity of Rep}
```

```
    f: real; {reduction factor}
begin
    if Spec or Rep null then return
    calculate S of Spec;
    calculate R of Rep;
    f:=R/S;
    if f>1 then f:=1;
    while (Spec not empty) do
        begin
            for each Spec cluster i at top level
                begin
                    {calculate the required computation
                    capacity of the Rep cluster to be
                    mapped onto}
                    R_i := R - (sum_{k:=1}^{i-1} R_k
                            + f x sum_{k:=i+1}^{n} sigma_S[k]);
                    { This is a more accurate version of
                    R_i = f x sigma_S[i] }
                    search for Rep cluster at top level of
                    computation capacity of R_i;
                    if found such Rep cluster j then
                        begin
                            delete cluster i from Spec;
                            delete cluster j from Rep;
                            map(Spec cluster i, Rep cluster j)
                        end {if}
                end; {for}
```

{Now there is no best match between Spec
and Rep clusters at top level}
delete header cluster s_h from Spec list;
initialize Rep cluster h to be empty;
$\sigma_R[h] := 0$;
repeat
delete header cluster r_h from Rep list;
$\sigma_R[h] := \sigma_R[h] + \sigma_R[r_h]$;
merge cluster r_h into cluster h;
until $\sigma_R[h] \geq R_{s_h}$;
if $\sigma_R[h] = R_{s_h}$ then
map(Spec cluster s_h, Rep cluster h)
else
begin
$extra$:=$\sigma_R[h] - R_{s_h}$;
delete the last cluster r_h from h;
split(cluster r_h, $extra$);
{returns the extra part and the
required part of cluster r_h}
merge extra part into Rep;
merge required part into cluster h;
map(Spec cluster s_h, Rep cluster h)
end {else}
end {while}
end; {map}

function split(cl:cluster, $extra$:integer);

{to split cl into two parts and return these two

clusters. One with computation capacity of $extra$.

The other one then has the required capacity.

This function is also a recursive one. }

begin

go to a lower level of cl;

search for a cluster of capacity $extra$;

if found such a cluster ca then

begin

delete ca from cl;

cluster $cb := cl - a$;

return the extra part as ca

and the required part as cb

end

else

begin

initialize a Rep cluster h to be empty;

$\sigma_R[h] := 0$;

repeat

delete header cluster r_h from cl;

$\sigma_R[h] := \sigma_R[h] + \sigma_R[r_h]$;

merge cluster r_h into cluster h;

until $\sigma_R[h] \geq extra$;

if $\sigma_R[h] = extra$ then

return h as extra part

```
            and the rest of cl as required part
        else
            begin
                extra1 := σ_R[h] − extra;
                delete the last cluster r_h from h;
                split(cluster r_h, extra1);
                {returns the extra part and the
                required part of cluster r_h}
                merge extra part into cl;
                merge required part into cluster h;
                return h as extra part
                and the rest of cl as required part
            end {else}
    end {else}
end; {split}
```

As shown in the pseudo code, to implement the step 4 of the algorithm in $O(KN)$ time for every Specification cluster, we consider all Representation clusters and select the best one which minimizes $|f_m|$. We do this for all the Specification clusters at that level. Furthermore, in calculating $|f_m|$, we only consider matching the computation capacity by letting $F_\delta = 0$. The results are good even though F_δ is forced to be 0 due to the effects of clustering. Clustering has two effects. First, it partitions the problem graph vertically to indicate group of computations which have data dependency. Second, it partitions the problem graph horizontally to create independent layers such that all the computations in that layer are to be computed concurrently. As a result of this, the concurrent computations (Specification clusters) are mapped onto concurrent processors (Representation clusters),

such that the clustered data dependent computations are mapped onto group of processors having efficient communication medium.

In the following, we show the results of our experiments in four categories according to the structures of the input problem and system graphs.

Regular problem vs. regular system

The output of mapping of an associative binary operation problem onto a cube architecture, whose Cluster-M specification and representation are illustrated in Section 2, is shown in Figure 4.1. In this example, both the problem and the system are regular, i.e. the problem graph and system graph have uniform structures. Our experiment results show that our algorithm is very efficient in producing close to optimal solutions for these regular mappings.

Irregular problem vs. regular system

Multiprocessor systems are usually constructed with a uniform structure. However problems may have an irregular structure. This may make the mapping process difficult. Figure 4.2 shows how an irregular problem is mapped onto a 2×2 mesh. The problem contains 6 fine grain subtasks. Subtask a, b and c have computation requirement of 2, while the computation requirement of subtask d and f are of 3, and subtask e of 4. The output of our mapping algorithm shown here is actually an optimal solution.

Regular problem vs. irregular system

Figure 4.3 illustrates the output of our algorithm in mapping an associative binary operation onto an irregular system. The problem contains 8 subtasks of computation requirement 1, while the system has 8 processors of computation capacity 1, except processor A has capacity 2 (e.g. A is a master processor of the completely connected

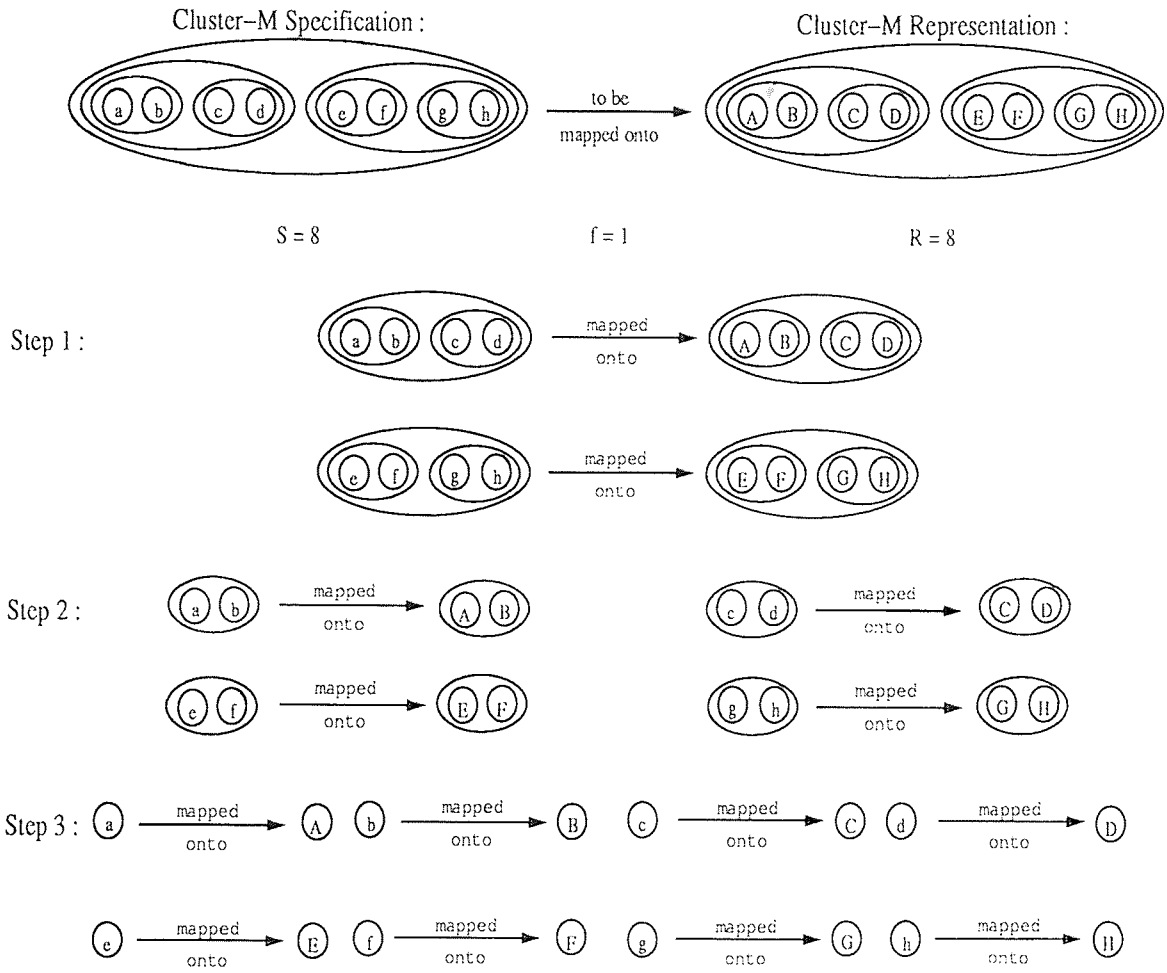

Figure 4.1 *Mapping associative binary operation onto cube.*

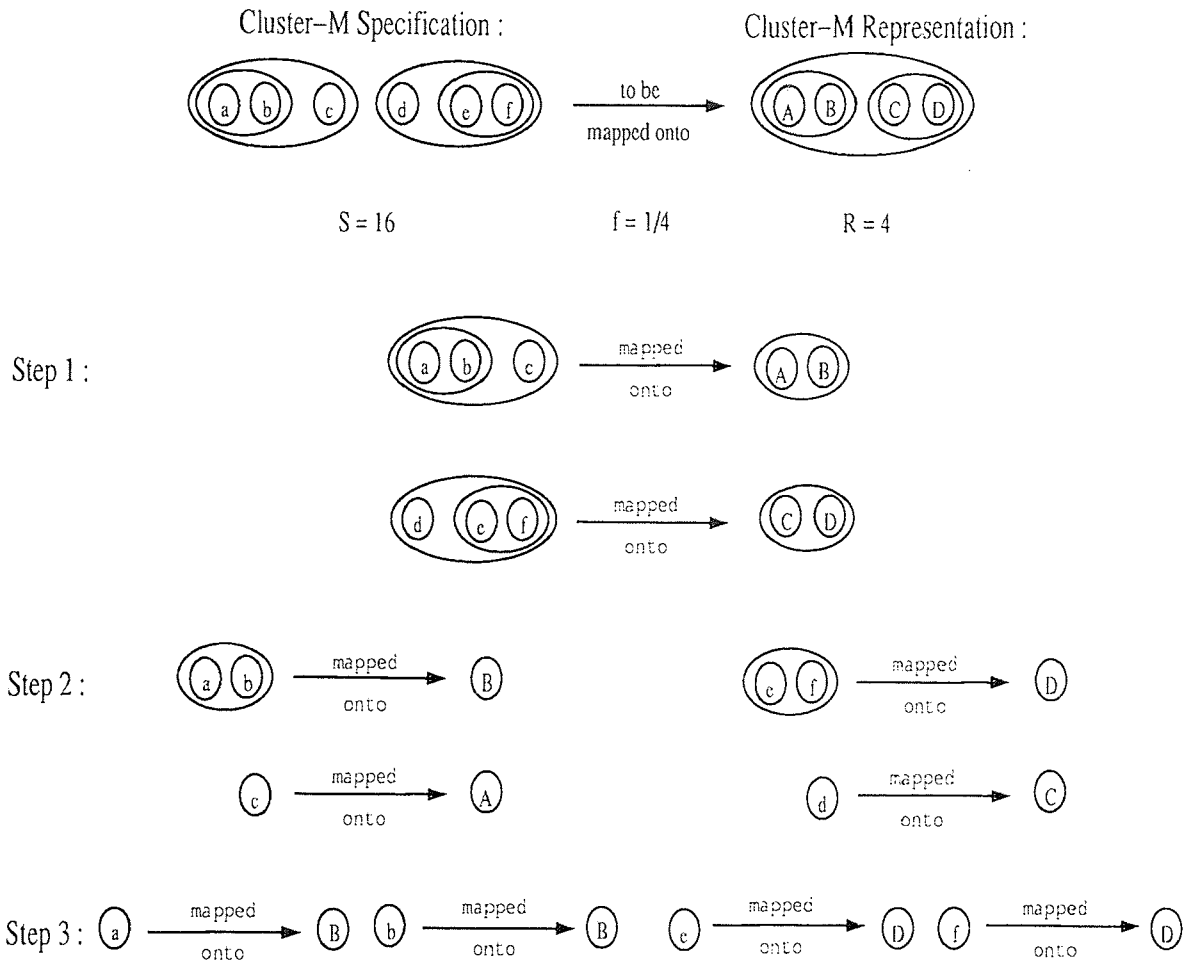

Figure 4.2 *Mapping irregular problem onto mesh.*

subsystem of A, B, C and D). In this particular example, the system provides more computation capacity than the problem requirement while still maintaining the problem communication needs. Therefore, processor D is not used.

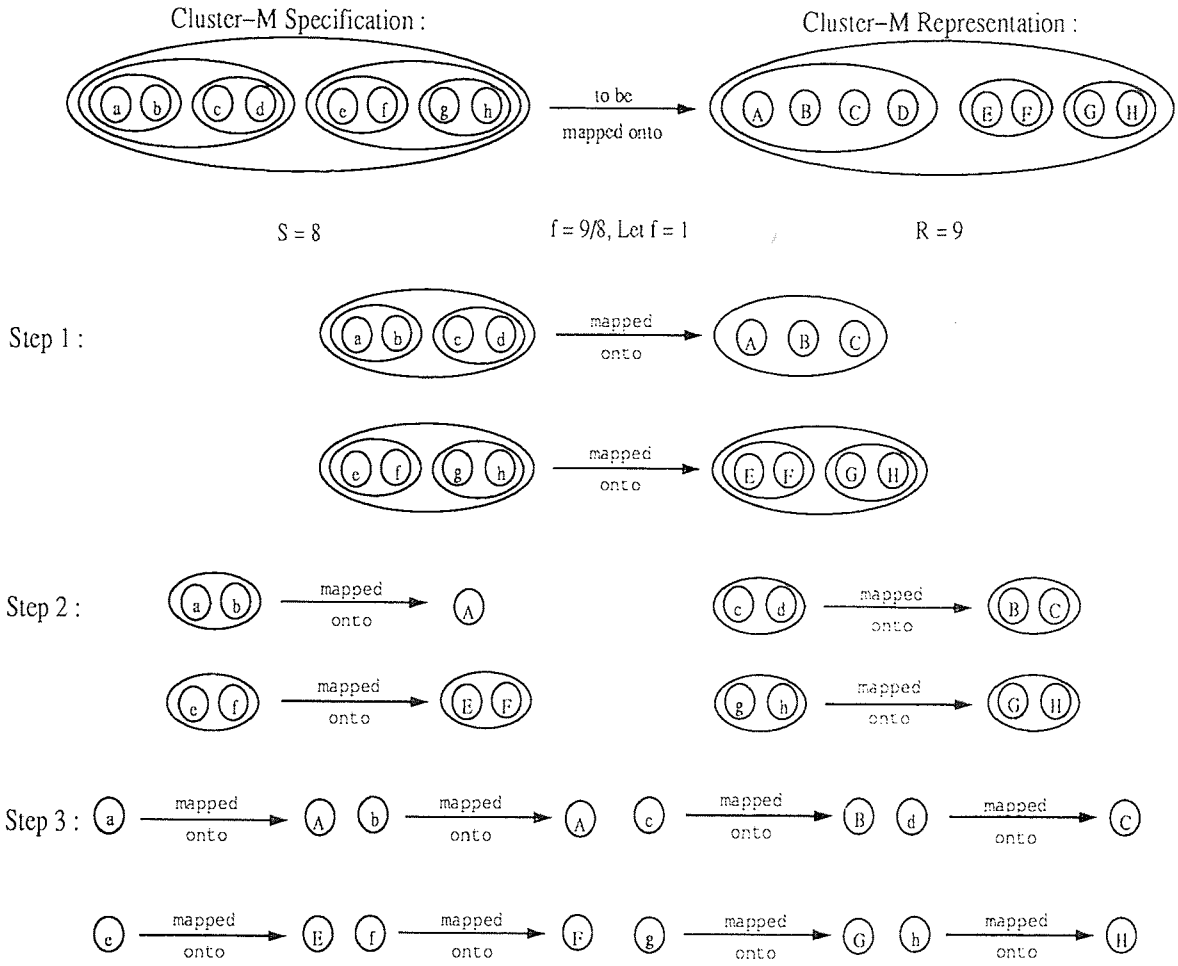

Figure 4.3 *Mapping binary-associative operation onto an irregular system.*

Irregular problem vs. irregular system

Such an example is given in Figure 4.4. In this example, all the fine grain subtasks have the same computation requirement of 1, and all the processors have the computation capacity of 1. However, the total Specification computation requirement and total Representation computation capacity are different, and the communication patterns of the subtasks and processors are different too.

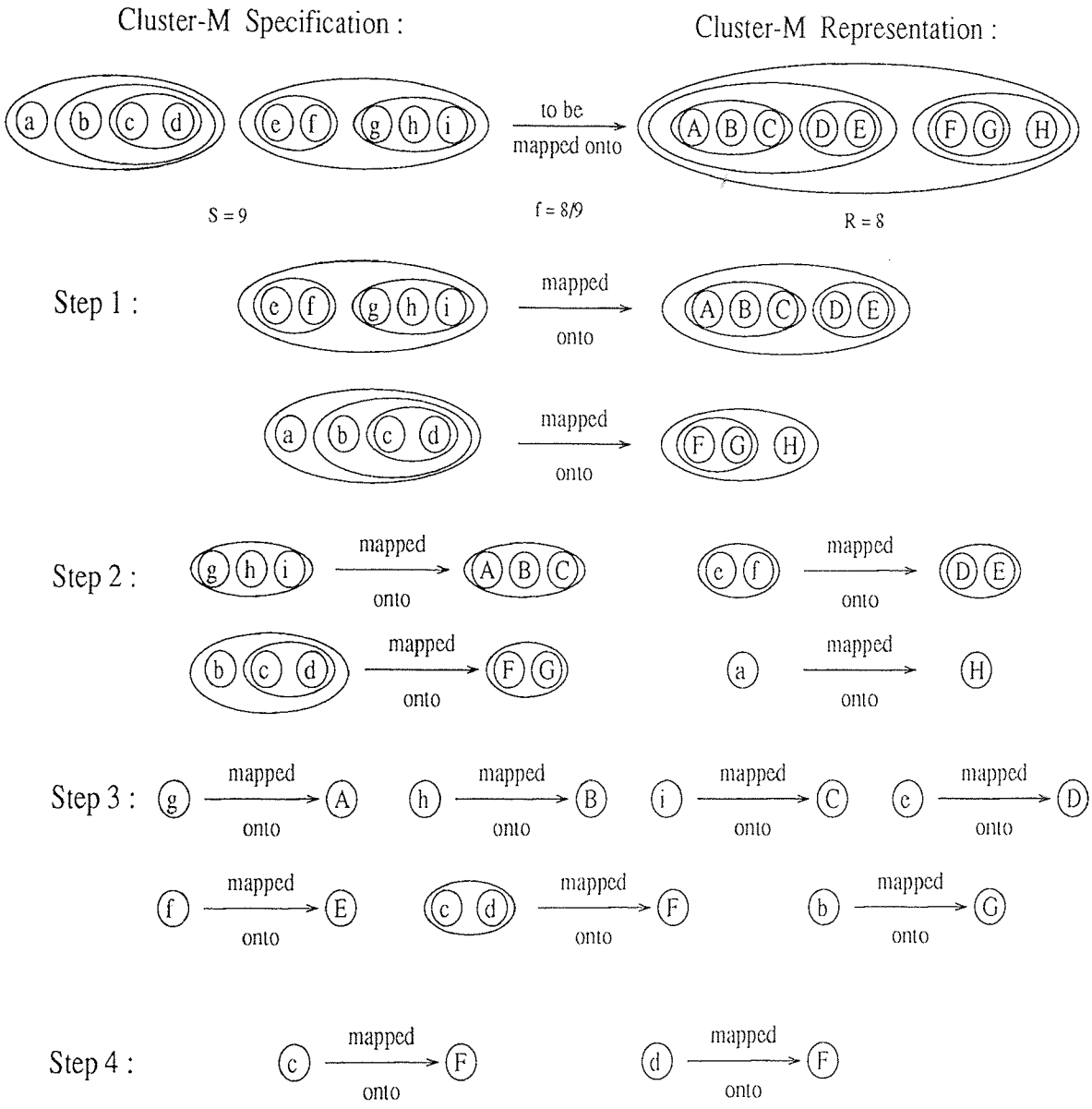

Figure 4.4 *An example for irregular mapping.*

CHAPTER 5

CONCLUSION AND FUTURE RESEARCH

In this thesis we have implemented a novel algorithm for mapping fine grain computations of portable Cluster-M algorithms onto various multiprocessor systems. The input to the mapping algorithm is a Cluster-M Specifications graph which corresponds to a layered problem graph, and a Cluster-M Representations which corresponds to a layered system graph. Unlike other mapping approaches, which map the entire problem graph onto the entire system graph, this algorithm reduces the complexity of the problem by only mapping the corresponding layers of the two graphs. We presented our experimental results in using the implemented algorithm for mapping various types of problems onto systems. Our implementation produces fast and sub-optimal results. Furthermore, these results can be extended for mapping of application tasks onto heterogeneous suite of processors. Also the proposed implementation can be improved by parallelizing various components of the original algorithm.

APPENDIX A

SOURCE CODE MAP.C

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int getnum();
struct cluster {
   int cluster_num;
   int pr_num;
   int ser_num;
   int req_rep;
   int map_flag;
   struct cluster *save_ptr;
   struct cluster *cl_ptr;
   struct cluster   *sub_cl;
   struct cluster   *par_ptr;
};
struct cluster *rep_start,*spec_start;
struct cluster *insert();
struct cluster *st_cluster  ;
struct cluster *start_rep;
struct cluster *start_spec;
struct cluster *spec_save;
struct cluster *start_ins;
struct cluster *insert_clptr();;
struct cluster *map_ptr;

int N,sernum_tag =1,ser_num_tag2,ser_num_tag1;
FILE *fp;

/*  This function reads a whole number from the file fp and returns  */
/*  it. If cflag is one character, ch is taken as part of the        */
/*  number                                                           */


int getnum(cflag,ch)
int cflag,ch;
{
```

```
  int i;
  char ch_array[10];
  int c;
   i = 0;
   /* Retreiving already read character into ch_array        */
   if (cflag)
      ch_array[i++] = ch;
   c = fgetc(fp);
   if (c == EOF)
   {
      return(0);
   }
   /* Reading the whole number                               */
   while (isdigit((char) c))
         {
      ch_array[i++] = (char) c;
      c=fgetc(fp);
         }
   ch_array[i] = '\0';
   c = atoi(ch_array);
   /* Converting characters to number                       */
   return(atoi(ch_array));
}

/*  This function creates and inserts in ascending order, the cluster */
/*  of size "size" in the recursive pointer structure given           */
/*  st_cluster                                                        */

struct cluster *crt_cl(size)
int size;
{

   struct cluster *cl,*cl_prev;
   int c;
   /* Searching for the position of the cluster             */
   for (cl=st_cluster, cl_prev=NULL; ((cl) &&
      (cl->cluster_num > size)); cl_prev=cl,cl=cl->cl_ptr)
         ;
        /* Creating the cluster with the given size              */
   cl= (struct cluster *) malloc(sizeof(struct cluster));
   cl->cluster_num = size;
   cl->ser_num = sernum_tag++;
   cl->map_flag =0;
   cl->cl_ptr = (struct cluster *) NULL;
```

```
  cl->sub_cl = (struct cluster *) NULL;
  cl->save_ptr = NULL;
  /* First cluster in the list st_cluster                    */
  if (st_cluster == (struct cluster *) NULL )
  {
     st_cluster = cl;
     return(cl);
  }
  /* To be inserted after cl_prev                            */
  if (cl_prev)
  {
     cl->cl_ptr = cl_prev->cl_ptr;
     cl_prev->cl_ptr = cl;
  }
  else
  {
     /* To be inserted after st_cluster                      */
     if (st_cluster->cluster_num > cl->cluster_num)
     {
        cl->cl_ptr= st_cluster->cl_ptr;
        st_cluster->cl_ptr = cl;
     }
     /* To be inserted before st_cluster                     */
     else
     {
        cl->cl_ptr = st_cluster;
        st_cluster = cl;
     }
  }
  return(cl);
}

/*  This function outputs the recursive pointer structure pointed */
/*  by start to the output device                                 */

Write_Graph(start)
struct cluster *start;
{
struct cluster *cl;

if (start == NULL)
   return;
else
{
```

```
    /* for all the clusters from start                          */
    for (cl = start ; (cl != NULL) ; cl = cl->cl_ptr)
    {
       if (cl->sub_cl == NULL)
          printf("Processor num : %d \n",cl->pr_num);
       else
           printf("cluster num      : %d \n",cl->cluster_num);
    }
    printf("\n");
    if (start->sub_cl == NULL)
    {
       /* Searching for next parent to start from       */
       while ((start->par_ptr != NULL)&&
          (start->par_ptr->cl_ptr == NULL))
          start =start->par_ptr;
       /* start has no parent                           */
       if (start->par_ptr == NULL)
          Write_Graph(start->cl_ptr);
       else
          Write_Graph(start->par_ptr->cl_ptr);
    }
    else
       Write_Graph(start->sub_cl);
}
}

/*  This function gives the serial number to each cluster in the  */
/*  recursive pointer structure starting at start                 */

giv_sernum(start)
struct cluster *start;
{
struct cluster *cl;

   if (start == NULL)
      return;
   else
   {
   /* Giving serial number to all the clusters                  */
   for (cl = start ; (cl != NULL) ; cl = cl->cl_ptr)
   {
      cl->ser_num = ser_num_tag1++;
   }
```

```
   /* Invoking recursion if necessary                          */
   if (start->sub_cl == NULL)
   {
      while ((start->par_ptr != NULL)&&
         (start->par_ptr->cl_ptr == NULL))
         start =start->par_ptr;
      if (start->par_ptr == NULL)
         giv_sernum(start->cl_ptr);
      else
         giv_sernum(start->par_ptr->cl_ptr);
   }
   else
   giv_sernum(start->sub_cl);
   }
}

/*  This function gets the input representation clusters and  */
/*  specification clusters from the file fp and forms the     */
/*  recursive pointer structure used for mapping              */

Input()
{

  int i;
  int c;
  struct cluster *cl,*cur_cl;
  struct cluster *start;
  struct cluster *cur_par;
  int open, closed;
  int break_flag;
  int num,k;
  int rep_flag,j,l;
  int save;

   /* Reading the given input */
   rep_flag = 1;
   for (j =0; j<2;j++) /* for rep and spec input  */
   {
   num= getnum(0,0);
   for (i=0;i<num;i++) /* Number of clusters times */
   {
      N= getnum(0,0);
      st_cluster = start ;  /* Initialising the start   */
           cur_cl = crt_cl(N);
```

```
cur_cl->par_ptr = NULL;
start = st_cluster;
c = fgetc(fp);
open =0;
closed =0;
break_flag =0;
save = 0;
while ((c != EOF) && (c != '\n'))
{
   if (c == '[') /*Sub cluster is read       */
   {
      open++;
      N= getnum(0,0);
      save = N;
      st_cluster = cur_cl->sub_cl;
      cur_par = cur_cl;
      cur_cl = crt_cl(N);
      cur_cl->par_ptr = cur_par;
      cur_par->sub_cl = st_cluster;
    }
   else {
   if( c == '(')  /* Processor Number is given*/
   {
      N= getnum(0,0);
      cur_cl->cluster_num = save;
      cur_cl->pr_num = N;
      cur_cl->sub_cl = NULL;
   }
   else {
   if ( c == ')')
   {
      c = fgetc(fp);
      if (isdigit(c))
      /* Another sub cluster is read     */
      {
         N = getnum(1,c);
         save = N;
         st_cluster = cur_par->sub_cl;
         cur_cl = crt_cl(N);
         cur_cl->par_ptr = cur_par;
         cur_par->sub_cl = st_cluster;
      }
      else
      {
```

```
   /* All sub clusters of present
    cluster are completely read*/
      if(c == ']')
      {
      do{
         cur_par = cur_par->par_ptr;
         closed++;
         if (open == closed )
         {
            break_flag = 1;
            break;
         }
         c = fgetc(fp);
      }while (c==']');
      if (break_flag)
      {
      /* Complete cluster is read  */
          c = fgetc(fp);
          while ((c!='\n')&&(c!=EOF))
         c=fgetc(fp);
          break;
      }
      /* Get next input            */
      N = getnum(1,c);
      save = N;
      st_cluster = cur_par->sub_cl;
      cur_cl = crt_cl(N);
      cur_cl->par_ptr = cur_par;
      cur_par->sub_cl = st_cluster;
      }
   }
}
else {
   if(c == ']')
   {
   do{
      cur_par = cur_par->par_ptr;
      closed++;
      if (open == closed )
      {
         break_flag = 1;
            break;
      }
      c = fgetc(fp);
```

```
            }while (c==']');
            if (break_flag)
               break;
            N = getnum(1,c);
            save = N;
            st_cluster = cur_par->sub_cl;
            cur_cl = crt_cl(N);;
            cur_cl->par_ptr = cur_par;
            cur_par->sub_cl = st_cluster;
            }
         }
         }
         }
         c = fgetc(fp);
      }
   }
   if (rep_flag )
   {       /* Assigning the Representation start            */
      rep_flag = 0;
      start_rep = start;
      ser_num_tag1 = sernum_tag;
      ser_num_tag2 = sernum_tag;
      start = NULL;
   }
   else
          /* Assigning the Specification start            */
   {
      rep_flag = 0;
      start_spec = start;
   }
   }
   printf("\n\n Representation clusters are : \n");
   Write_Graph(start_rep);
   printf("\n\nSpecification clusters are : \n");
   Write_Graph(start_spec);
   }

/*  This function creates and initializes a given cluster and */
/*  returns it                                                */

struct cluster *initialise()
{
   struct cluster *cl;
   /* Creating and Initialising the cluster cl          */
```

```
   cl= (struct cluster *) malloc(sizeof(struct cluster));
   cl->cluster_num = 0;
   cl->cl_ptr = (struct cluster *) NULL;
   cl->sub_cl = (struct cluster *) NULL;
   cl->save_ptr = NULL;
   cl->par_ptr = NULL;
   return(cl);
}

/*  This function returns the deepest sub cluster of the cluster cl */

struct cluster *cheksubcl(cl)
struct cluster *cl;
{
   struct cluster *save,*save1;
   /* Checks sub clusters of cl and returns the last */
   /* subcluster of the cluster cl                   */

   save = cl->sub_cl;
   while (save)
   {
      save1 = save;
      save = save->sub_cl;
   }
   return(save1);
}

/*  This function inserts a sub_cluster into par_cl cluster  */

struct cluster *insert_sub(par_cl)
struct cluster *par_cl;
{
   struct cluster *cl,*save1,*cl1,*cl1_prev;
        for (cl = par_cl->sub_cl;cl; cl = cl->cl_ptr)
   /* For all the subclusters of par_cl         */
   {
        if (save1=cheksubcl(cl))
         cl = save1;
      /* If there are no subclusters to par_cl   */
      if (par_cl->sub_cl == NULL)
      {
          par_cl->sub_cl = cl;
          par_cl->sub_cl->cl_ptr = NULL;
```

```
    }
    else
    {
    /* Insert cl in the sub clusters of par_cl  */
         for (cl1_prev=NULL,cl1 = par_cl->sub_cl;cl1;
             cl1_prev = cl1,cl1=cl1->cl_ptr)
                ;
        if (cl1_prev)
             cl1_prev->cl_ptr = cl;
        cl1_prev->cl_ptr->cl_ptr = NULL;
    }
    cl->par_ptr = NULL;
    par_cl->cluster_num = par_cl->cluster_num+
                cl->cluster_num;
  }


/*  This function copies the sub clusters of par1 into the   */
/*  sub clusters of par2 and returns par2                    */

struct cluster *copy_cl(par1,par2)
struct cluster *par1,*par2;
{
   struct cluster *cl,*cl1,*cl1_prev;

   /* For all the subclusters of par1                          */

   for (cl = par1->sub_cl;cl;cl = cl->cl_ptr)
   {

      /* Insert into par2                                     */

      if (par2->sub_cl == NULL)
      {
         par2->sub_cl = cl;
         par2->sub_cl->cl_ptr = NULL;
      }
      else
      {
         for (cl1_prev = NULL,cl1 = par2->sub_cl;
            cl1;cl1_prev=cl1,cl1=cl1->cl_ptr)
               ;
         cl1_prev->cl_ptr = cl;
         cl1_prev->cl_ptr->cl_ptr = NULL;
```

```
      }
      cl->par_ptr = par2;
      par2->cluster_num = par2->cluster_num + cl->cluster_num;
   }
   return(par2);
}

/*  This function splits the cluster spt_cluster into two   */
/*  sub clusters of size num and total size_num and returns */
/*  them to the calling functions                           */

struct cluster *split(spt_cluster,num)
int num;
struct cluster *spt_cluster;
{
   struct cluster *best_match();
   struct cluster *remove();
   int sum_rep,diff,req;
   struct cluster *cl_prev1,*save1,*rep_save,*cl_prev,*cl;
   struct cluster *sp_cl,*sp_cl_prev,*cl1_prev,*cl1;
   struct cluster *save,*best_found,*map_save,*extra_save;
   /* Creating and initialising the rep_save and map_save */
   rep_save = initialise();
   map_save = initialise();
   extra_save = initialise();

   /* Checking for the best match in sub clusters of the */
   /* spt_cluster for the size of num                    */

   if (best_found = best_match(spt_cluster,num))
   {
      /* Removing best size subcluster from spt_cluster */

      spt_cluster->sub_cl =
         remove(best_found->ser_num,spt_cluster->sub_cl);
      spt_cluster->cluster_num = spt_cluster->cluster_num-
                  best_found->cluster_num;
      if (spt_cluster->cluster_num == 1)
      {
         if (save1=cheksubcl(spt_cluster))
            spt_cluster = save1;
      }

      /* Inserting extra size of spt_cluster into extra_save*/
```

```
    extra_save->sub_cl  = spt_cluster;
    spt_cluster->par_ptr = extra_save;
    extra_save->cluster_num = extra_save->cluster_num +
              spt_cluster->cluster_num;

    /* Inserting best size subcluster into map_save     */

    map_save->sub_cl  = best_found;
    best_found->par_ptr = map_save;
    map_save->cluster_num = map_save->cluster_num +
              best_found->cluster_num;

    /* Linking mapping part and extra part together     */

    map_save->save_ptr = extra_save;
    return(map_save);
}
else
{

    /* If subcluster size of spt_cluster is greater   */
    /* than required size num                         */

    if (spt_cluster->sub_cl->cluster_num > num)
    {
       /* Put all the subclusters other than first */
       /* first subcluster in extra_save          */

       for (cl = spt_cluster->sub_cl->cl_ptr;cl;
             cl = cl->cl_ptr)
       {
         if (save1 = cheksubcl(cl))
            cl = save1;
         if (extra_save->sub_cl == NULL)
         {
            extra_save->sub_cl = cl;
            extra_save->sub_cl->cl_ptr = NULL;
         }
         else
         {
          for (cl1 = extra_save->sub_cl;cl1;
             cl1=cl1->cl_ptr)
                ;
```

```
        cl1->cl_ptr = cl;
        cl1->cl_ptr->cl_ptr = NULL;
       }
       extra_save->cluster_num =
        extra_save->cluster_num+cl->cluster_num;
       cl->par_ptr = extra_save;
        }

    /* Trying to split the first subcluster of   */
    /* the spt_cluster                           */

    if (rep_save = split(spt_cluster->sub_cl,num))
    {
       if (rep_save->cluster_num == num)
       {

          /* copying the extra_save part returned*/
          /* into extra_save                     */

          extra_save =
       copy_cl(rep_save->save_ptr,extra_save);

          /* Copying required size part into     */
          /*          map_save                   */

          map_save = copy_cl(rep_save,map_save);

          /* Linking extra and required together */

          map_save->save_ptr = extra_save;
          return(map_save);
       }
       else
       {
       printf("can't split acurately\n");
       exit(1);
       }
    }
}
else
{

/* If first subcluster size is less than the        */
/* required size                                    */
```

```
sum_rep = 0;

/* Create sum_rep cluster and initialise it  */

rep_save = initialise();

/* Accumulate subcluster until their total   */
/* size is greater than or equal to num      */
/* into rep_save cluster                     */

for (cl_prev = NULL,cl=spt_cluster->sub_cl;
   ((cl) && (sum_rep < num));
   cl_prev = cl,cl=cl->cl_ptr)
{
   sum_rep = sum_rep+cl->cluster_num;
   if (rep_save->sub_cl)
   {
     for (sp_cl = rep_save->sub_cl,
         sp_cl_prev = NULL;sp_cl;
      sp_cl_prev = sp_cl,
      sp_cl=sp_cl->save_ptr)
         ;
     sp_cl_prev->save_ptr = cl;
     sp_cl_prev->save_ptr->save_ptr = NULL;
   }
   else
   {
     rep_save->sub_cl = cl;
     rep_save->sub_cl->save_ptr = NULL;
   }
   cl->par_ptr = rep_save;
   spt_cluster->sub_cl =
   remove(cl->ser_num,spt_cluster->sub_cl);
}
for (cl=rep_save->sub_cl;cl;cl=cl->cl_ptr)
   cl->cl_ptr = cl->save_ptr;
rep_save->cluster_num = sum_rep;
if (sum_rep == num)

/* If the rep_save cluster is of required size*/

{
```

```
   /* Link extra and required parts and */
   /*   return                          */

   map_save = rep_save;
   extra_save = spt_cluster;
   map_save->save_ptr = extra_save;
   return(map_save);
}
else
{

/* If rep_save is not of exact size          */

  diff = sum_rep - num;
  if (diff <0)
   return;

  /* Find the last subcluster of rep_save    */

  for (cl_prev1=NULL,cl_prev=NULL,
   cl=rep_save->sub_cl;cl;cl_prev1 =
   cl_prev,cl_prev=cl,cl=cl->save_ptr)
      ;

   /* Delete the last cluster from rep_save  */

  if (cl_prev1)
   cl_prev1->save_ptr = NULL;
  else
      rep_save->sub_cl = NULL;

  /*save contains the last cluster of rep_save*/

  save = cl_prev;

  /* Calculate the required size for num     */

  req = save->cluster_num - diff;

  /* Updating the rep_save as to the removal */
  /* of save                                 */

  rep_save->cluster_num =
   rep_save->cluster_num-save->cluster_num;
```

```
            /* Call split with save for new required   */
            /*            size                         */

            map_save = split(save,req);

            /* Take the results of split and merge them*/

            extra_save = map_save->save_ptr;
            rep_save = copy_cl(map_save,rep_save);
             map_save = rep_save;
            map_save->save_ptr = extra_save;
            return(map_save);
         }
      }
      }
}

/*  This function works for cluster size of "size" in sub clusters */
/*  of start cluster and returns it if found                       */

struct cluster *best_match(start,size)
struct cluster *start;
int size;
{
   struct cluster *cl;

   /* Searching for the given size in subclusters of start      */

   for (cl = start->sub_cl;cl;cl = cl->cl_ptr)
   {
      if (cl->cluster_num == size)
         return(cl);
    }
   return(0);
}

/*  This function removes the cluster with serial number    */
/*  sernum from the cluster pointed by start                */

struct cluster *remove(sernum,start)
int sernum;
struct cluster *start;
{
```

```
   struct cluster *sp_cl,*sp_cl_prev;

   /* Searching for the cluster of sernum to be removed      */

   for (sp_cl=start,sp_cl_prev=NULL;sp_cl;
      sp_cl_prev = sp_cl, sp_cl=sp_cl->cl_ptr)
   if (sp_cl->ser_num==sernum)
   {

      /* Removing the cluster with given sernum from start */

      if (sp_cl_prev)
          sp_cl_prev->cl_ptr = sp_cl->cl_ptr;
           else
         start= sp_cl->cl_ptr;
      break;
   }
   return(start);
}

/*  This function maps the specification cluster pointed by */
/*  spec to the representation cluster pointed by rep       */

map(rep,spec)
struct cluster *rep;
struct cluster *spec;
{
   struct cluster *sp,*sp_prev,*cl,*cur_cl,*cl_prev,*cl_found;
   struct cluster *cl_prev1,*search();
   struct cluster *sp1,*sp_cl,*sp_cl_prev;
   struct cluster *save,*cl1,*cl1_prev,*sp1_prev,*rep_save;
   struct cluster *extra_save,*best_found;
   int inttemp,req,diff,flag,sum_rep;
   float fldiff,R,S,f,sum_req_rep,sum;

   /* Condition for breaking the Recursion                    */

   if ((spec== NULL) || (rep== NULL ))
   {
      start_rep = rep;
      start_spec = spec;
      return;
   }
```

```
/* Calculating the total Representation size                    */

R=0.0;
for (cl=rep;cl;cl=cl->cl_ptr)
   R = R + cl->cluster_num;

/* Calculating the total Specification size                     */

S=0.0;
for (cl=spec;cl;cl=cl->cl_ptr)
   S = S + cl->cluster_num;

/* Calculating f value                                          */

if ((S) && (R))
   f = R/S;
else
   {
      printf("Check R and S values \n");
      exit(1);
   }

/*If there are more Processors than the Specification requires*/

if (f >1 )
   f = 1.0;
ser_num_tag1 = ser_num_tag2;

/* Give serial number to all the clusters for identifying  */

giv_sernum(spec);

/* For all the clusters in the Specification                */

for (cur_cl = spec; cur_cl;cur_cl = cur_cl->cl_ptr)
{
     if (f==1.0)
   cur_cl->req_rep = cur_cl->cluster_num;
     else
     {
   sum = 0.0; sum_req_rep = 0.0;

   /* Finding the required size of Representation to be*/
   /* searched for the specification mapping */
```

```
   for (cl = spec;cl;cl=cl->cl_ptr)
   {
      if (cl->ser_num < cur_cl->ser_num)
         sum_req_rep = sum_req_rep + cl->req_rep;
      else {
      if (cl->ser_num > cur_cl->ser_num)
      {
         sum = sum + cl->cluster_num;
      }
      }
   }
   if (f)
      sum = sum * f;
   sum = sum + sum_req_rep;
   sum = R - sum;
   inttemp = sum;
   fldiff = sum - inttemp;
   if (fldiff >= 0.4)
      sum = inttemp+1;
   else
      sum = inttemp;
   if (cur_cl->req_rep == 0)
      cur_cl->req_rep =   sum;
}
}

/*  For all the specification clusters                          */

for (sp1 = spec;sp1; sp1 = sp1->cl_ptr)
{

   /* For all the specification clusters                 */

   for (sp = spec;sp; sp = sp->cl_ptr)
   {
   /* Searching for the cluster size equal to
   required size of specification                          */
   if (cl_found = search(rep,sp->req_rep))
   {
      /* Removing sp from spec                        */
      spec = remove(sp->ser_num,spec);
      sp1 = spec;
      flag = 1;
```

```
rep = remove(cl_found->ser_num,rep);
/* Invoking recursion if necessary        */
if (cl_found->sub_cl==NULL)
{
   if (sp->sub_cl==NULL)
   {

      /* if there are no subclusters*/
      /* to both sp and cl_found    */

                printf("%d mapped to %d \n",sp->pr_num,
                       cl_found->pr_num);

      if ((!rep)|| (!spec))
      {
         start_rep = rep;
         start_spec = spec;
         return;
      }
   }
   else
   {
       /* If Processor is reached and */
       /* Specification has subclusters*/

       printf("%d cluster is mapped to pr %d\n",
               sp->cluster_num,cl_found->pr_num);
       printf("The cluster matched is \n");
       Write_Graph(sp);
       if ((!rep)|| (!spec))
       {
      start_rep = rep;
      start_spec = spec;
      return;
       }
   }
}
else
{

         if (cl_prev)
    {
   if (sp->sub_cl !=NULL)
   {
```

```
        /* If both have subclusters      */

        if (sp->cl_ptr)
        {
        /* If sp1's cluster pointer has required */
           /* rep zero                                  */
          if (sp->cl_ptr->req_rep == 0)
       {
              for(cl_prev = NULL,cl = cl_found;cl;
                        cl_prev = cl,cl = cl->sub_cl)
                       ;
              printf("%d cluster is mapped to pr %d\n",
                      sp->cl_ptr->cluster_num,cl_prev->pr_num);
              printf("The cluster matched is \n");
              Write_Graph(sp->cl_ptr);
            spec = remove(sp->cl_ptr->ser_num,spec);
       }
        }

          map(cl_found->sub_cl,sp->sub_cl);
        if ((rep == NULL) || (spec == NULL))
        {
           start_rep = rep;
           start_spec = spec;
           return;
        }
   }
   else
   {
      /* sp has no subclusters but */
      /* cl_found has subclusters*/

      printf("%d mapped to %d\n",sp->pr_num,
              cl_found->sub_cl->pr_num);
          if ((!rep)|| (!spec))
          {
         start_rep = rep;
         start_spec = spec;
         return;
          }
   }
}
```

```
}
}
}
if ((rep==NULL) || (spec==NULL))
   return;

/* If first cluster in Representation has greater */
/* size than required size of present sp          */

if (rep->cluster_num > sp1->req_rep)
       {

    /* If Rep cannot be spilt                     */

    if (rep->sub_cl == NULL)
    {
    if (sp1->cl_ptr)
        {
           if (sp1->cl_ptr->req_rep == 0)
          {
                    for(cl_prev = NULL,cl = rep;cl;
                             cl_prev = cl,cl = cl->sub_cl)
                           ;
                    printf("%d cluster is mapped to pr %d\n",
                            sp1->cl_ptr->cluster_num,cl_prev->pr_num);
                    printf("The cluster matched is \n");
                    Write_Graph(sp1->cl_ptr);
                    spec = remove(sp1->cl_ptr->ser_num,spec);
          }
        }
   printf("%d cluster is mapped to pr %d\n",
           sp1->cluster_num,rep->pr_num);
   printf("The cluster matched is \n");
   Write_Graph(sp1);

   /* Removing sp1 from spec                  */

   spec = remove(sp1->ser_num,spec);

   /* Reducing the size of rep                */

   rep->cluster_num = rep->cluster_num -
            sp1->cluster_num;
   map_ptr = rep;
```

```
rep = rep->cl_ptr;

/*Insertng map_ptr into rep in right position*/

start_ins = rep;
start_ins = insert_clptr(map_ptr);
rep = start_ins;
 }
 else
 {

 /* First cluster in Rep has to be split          */

     map_ptr = rep;
 rep = rep->cl_ptr;
     for (cl = map_ptr->sub_cl;cl;cl=cl->cl_ptr)
   cl->save_ptr = cl->cl_ptr;
save=NULL;

/* Searching for the required size in the   */
/* subclusters of first cluster in Rep      */

for (cl1 = map_ptr->sub_cl;cl1;
   cl1=cl1->save_ptr)
     {
   if (cl1->cluster_num==sp1->req_rep)
   {
      if (save == NULL)
         save = cl1;
      else
      {
      start_ins = rep;
      start_ins = insert_clptr(cl1);
      rep = start_ins;
      }
   }
   else
   {
      cl1->cl_ptr = NULL;
      start_ins = rep;
      start_ins = insert_clptr(cl1);
      rep = start_ins;
   }
            }
```

```
  /* If required size is in the subclusters   */
       if (save!=NULL)
       {
      spec = remove(sp1->ser_num,spec);
      save->cl_ptr = NULL;
      if (sp1->cl_ptr)
      {
          /* If sp1's cluster pointer has required */
             /* rep zero                                  */
          if (sp1->cl_ptr->req_rep == 0)
         {
                 for(cl_prev = NULL,cl = save;cl;
                           cl_prev = cl,cl = cl->sub_cl)
                        ;
                 printf("%d cluster is mapped to pr %d\n",
                        sp1->cl_ptr->cluster_num,cl_prev->pr_num);
                 printf("The cluster matched is \n");
                 Write_Graph(sp1->cl_ptr);
                   spec = remove(sp1->cl_ptr->ser_num,spec);
         }
      }
        map(save,sp1);
      if ((rep == NULL) || (spec == NULL))
      {
         start_rep = rep;
         start_spec = spec;
         return;
      }
   }
   }
}
else
{

/* If first cluster in Rep is less than sp1         */

   sum_rep = 0;
   rep_save = initialise();

   /* Accumulate clusters in Representation    */
   /* until their size is greater than or equal*/
   /* to sp1 size                              */

   for (cl_prev = NULL,cl=rep;((cl) && (sum_rep <
```

```
 sp1->req_rep)); cl_prev = cl,cl=cl->cl_ptr)
{
   sum_rep = sum_rep+cl->cluster_num;
   if (rep_save->sub_cl)
   {
     for (sp_cl = rep_save->sub_cl,
         sp_cl_prev = NULL;sp_cl;
      sp_cl_prev = sp_cl,
      sp_cl=sp_cl->save_ptr)
         ;
     sp_cl_prev->save_ptr = cl;
     sp_cl_prev->save_ptr->save_ptr = NULL;
   }
   else
   {

   /* Accumulate into rep_save        */

     rep_save->sub_cl = cl;
     rep_save->sub_cl->save_ptr = NULL;
   }
   rep = remove(cl->ser_num,rep);
}
rep_save->cluster_num = sum_rep;

/* If exact match between rep_save and sp1  */

if (sum_rep == sp1->req_rep)
{
    for (cl=rep_save->sub_cl;cl;cl=cl->save_ptr)
    {
         cl->par_ptr = rep_save;
       cl->cl_ptr = cl->save_ptr;
    }
    spec = remove(sp1->ser_num,spec);
    rep_save->cl_ptr = NULL;
    if (sp1->cl_ptr)
    {
   /* If sp1's cluster pointer has required */
     /* rep zero                                  */
   if (sp1->cl_ptr->req_rep == 0)
   {
          for(cl_prev = NULL,cl = rep_save;cl;
                   cl_prev = cl,cl = cl->sub_cl)
```

```
                    ;
            printf("%d cluster is mapped to pr %d\n",
                   sp1->cl_ptr->cluster_num,cl_prev->pr_num);
            printf("The cluster matched is \n");
            Write_Graph(sp1->cl_ptr);
            spec = remove(sp1->cl_ptr->ser_num,spec);
   }
    }
      map(rep_save,sp1);
    if ((!rep) || (!spec))
    {
         start_rep = rep;
       start_spec = spec;
       return;
    }
}
else
{

/* If sum_rep is greater than sp1          */

  diff = sum_rep - sp1->req_rep;
  if (diff <0)
   return;

  /* Remove last cluster in rep_save       */

  for (cl_prev1=NULL,cl_prev=NULL,
   cl=rep_save->sub_cl;cl;cl_prev1 =
   cl_prev,cl_prev=cl,cl=cl->save_ptr)
      ;
  if (cl_prev1)
   cl_prev1->save_ptr = NULL;
  else
      rep_save->sub_cl = NULL;
  /* Last cluster is in save               */

  save = cl_prev;
  req = save->cluster_num - diff;
  rep_save->cluster_num =
   rep_save->cluster_num-save->cluster_num;
  for (cl=rep_save->sub_cl;cl;cl=cl->save_ptr)
   cl->cl_ptr = cl->save_ptr;
               /* If best match is found in sub clusters*/
```

```
                /* of save with size req                    */
if (best_found = best_match(save,req))
{
                        /* remove the best found from save */
   save->sub_cl =
    remove(best_found->ser_num,
           save->sub_cl);
   save->cluster_num = save->cluster_num
          -best_found->cluster_num;
   best_found->cl_ptr = NULL;
                        /* Inserting best found into rep_save */
   for (cl_prev = NULL, cl =
        rep_save->sub_cl;cl;cl_prev = cl,
         cl= cl->cl_ptr)
         ;
   if (cl_prev)
   {
      cl_prev->cl_ptr = best_found;
      cl_prev->cl_ptr->cl_ptr =NULL;
   }
   else
   {
         rep_save->sub_cl = best_found;
         rep_save->sub_cl->cl_ptr = NULL;
   }
   best_found->par_ptr = rep_save;
   rep_save->cluster_num =
    rep_save->cluster_num +
         best_found->cluster_num;
   rep_save->cl_ptr = NULL;
   save->cl_ptr = NULL;
   save->par_ptr = NULL;
   spec = remove(sp1->ser_num,spec);
                        /* Inserting the remaining part of */
                        /* save into rep                   */
   start_ins = rep;
   start_ins = insert_clptr(save);
   rep = start_ins ;
   rep_save->cl_ptr = NULL;
   if(sp1->cl_ptr)
   {
      /* If sp1's cluster pointer has required */
        /* rep zero                            */
    if (sp1->cl_ptr->req_rep == 0)
```

```
    {
            for(cl_prev = NULL,cl = rep_save;cl;
                    cl_prev = cl,cl = cl->sub_cl)
                ;
            printf("%d cluster is mapped to pr %d\n",
                  sp1->cl_ptr->cluster_num,cl_prev->pr_num);
            printf("The cluster matched is \n");
            Write_Graph(sp1->cl_ptr);
            spec = remove(sp1->cl_ptr->ser_num,spec);
    }
   }
   map(rep_save,spi);
   if ((rep== NULL) || (spec == NULL))
   {
      start_rep = rep;
    start_spec = spec;
    return;
   }
}
else
{
                        /* Spilting the last cluster of */
                        /* rep_save for size of rep     */
   if (map_ptr = split(save,req))
   {
                        /* Inserting extra part into rep */
    start_ins = rep;
    start_ins = insert_clptr(
       map_ptr->save_ptr);
    rep = start_ins;
    map_ptr->cl_ptr = NULL;
                         /* Inserting cluster of size req */
                         /* into rep_save                 */
    for (cl_prev = NULL,cl =
      rep_save->sub_cl;cl;
      cl_prev = cl, cl= cl->cl_ptr)
          ;
    if (cl_prev)
         cl_prev->cl_ptr = map_ptr;
    else
         rep_save->sub_cl = map_ptr;
    map_ptr->par_ptr = rep_save;
    rep_save->cluster_num =
       rep_save->cluster_num +
```

```
                    map_ptr->cluster_num;
                                    /* mapping sp1 and rep_save  */
                spec = remove(sp1->ser_num,spec);
                if(sp1->cl_ptr)
                {
              /* If sp1's cluster pointer has required */
                /* rep zero                                    */
             if (sp1->cl_ptr->req_rep == 0)
             {
                       for(cl_prev = NULL,cl = rep_save;cl;
                            cl_prev = cl,cl = cl->sub_cl)
                             ;
                    printf("%d cluster is mapped to pr %d\n",
                           sp1->cl_ptr->cluster_num,cl_prev->pr_num);
                    printf("The cluster matched is \n");
                    Write_Graph(sp1->cl_ptr);
                     spec = remove(sp1->cl_ptr->ser_num,spec);
             }
                }
                map(rep_save,sp1);
                     if ((!rep) || (!spec))
                     {
                          start_rep = rep;
                        start_spec = spec;
                        return;
                     }
          }
          else
          {
           printf("spliting not done properly\n");
          }
      }
      }
   }
}
if ((rep) && (spec))
{
   if (spec->cl_ptr)
   {
      /* If sp1's cluster pointer has required */
      /* rep zero                                    */
      if (spec->cl_ptr->req_rep == 0)
      {
           for(cl_prev = NULL,cl = rep;cl;
```

```
                    cl_prev = cl,cl = cl->sub_cl)
                     ;
                printf("%d cluster is mapped to pr %d\n",
                       spec->cl_ptr->cluster_num,cl_prev->pr_num);
                printf("The cluster matched is \n");
                Write_Graph(spec->cl_ptr);
                spec = remove(spec->cl_ptr->ser_num,spec);
            }
        }
        map(rep,spec);
    }
    if ((!rep)||(!spec))
    {
        start_rep = rep;
        start_spec = spec;
        return;
    }
    /* Assigning the values of start_rep and start_spec        */
}

/*  This function inserts the cluster cl_ins into the clusters     */
/*  pointed by start_ins in the ascending order using save_ptr     */

struct cluster *insert(cl_ins)
struct cluster *cl_ins;
{

    struct cluster *cl_prev,*cl;
    /* Searching for the appropriate position in start_ins     */
    for (cl_prev = NULL,cl=start_ins;((cl)
        && (cl->cluster_num>=cl_ins->cluster_num));
            cl_prev = cl,cl = cl->save_ptr)
            ;
    /* Starting cluster in the list start_ins                  */
    if (start_ins == (struct cluster *) NULL )
    {
        start_ins = cl_ins;
        start_ins->par_ptr = NULL;
        start_ins->save_ptr = NULL;
        return(start_ins);
    }
    else
    {
    /* To be inserted after cl_prev                            */
```

```
   if (cl_prev)
   {
      cl_prev->save_ptr = cl_ins;
      cl_ins->save_ptr = cl;
      cl_ins->par_ptr = NULL;
   }
   else
   {
      /* To be inserted after start_ins                    */
      if (start_ins->cluster_num > cl->cluster_num)
      {
         cl_ins->save_ptr= (struct cluster *) start_ins->save_ptr;
         start_ins->save_ptr = cl_ins;
         cl_ins->par_ptr = NULL;
      }
      /* To be inserted before start_ins                   */
      else
      {
         cl_ins->save_ptr = start_ins;
         start_ins = cl_ins;
         start_ins->par_ptr = NULL;
         cl_ins->par_ptr = NULL;
      }
   }
   }
   return(start_ins);
}


/*  This function inserts the cluster cl_ins into the clusters     */
/*  pointed by start_ins in the ascending order using cl_ptr       */

struct cluster *insert_clptr(cl_ins)
struct cluster *cl_ins;
{

   struct cluster *cl_prev,*cl;
   /* Searching for the position in the start_ins for cl_ins  */
   for (cl_prev = NULL,cl=start_ins;((cl)
      && (cl->cluster_num>=cl_ins->cluster_num));
         cl_prev = cl,cl = cl->cl_ptr)
         ;
   /* cl_ins is the first cluster in start_ins list           */
   if (start_ins == (struct cluster *) NULL )
```

```
   {
      start_ins = cl_ins;
      start_ins->par_ptr = NULL;
      start_ins->cl_ptr = NULL;
      return(start_ins);
   }
   else
   {
   /* To be inserted after cl_prev                                  */
   if (cl_prev)
   {
      cl_prev->cl_ptr = cl_ins;
      cl_ins->cl_ptr = cl;
      cl_ins->par_ptr = NULL;
   }
   else

   {
      /* To be inserted after start_ins                    */
      if (start_ins->cluster_num > cl->cluster_num)
      {
         cl_ins->cl_ptr= (struct cluster *) start_ins->cl_ptr;
         start_ins->cl_ptr = cl_ins;
         cl_ins->par_ptr = NULL;
      }
      else
      /* To be inserted before start_ins                 */
      {
         cl_ins->cl_ptr = start_ins;
         start_ins = cl_ins;
         start_ins->par_ptr = NULL;
         cl_ins->par_ptr = NULL;
      }
   }
   }
   return(start_ins);
}
/*  This function searches for the cluster of size "size" in the */
/*  clusters pointed by start and returns it                     */

struct cluster *search(start,size)
int size;
struct cluster *start;
{
```

```
  struct cluster *v;
   /* Searching the cluster with its size equal to 'size'     */
   for (v = start; ((v) && (v->cluster_num != size));
      v = v->cl_ptr)
         ;
   return(v);
}
/*  This function opens the input file, calls Input() to read */
/*  the input from the file and calls the map() function to   */
/*  map the clusters and then outputs the result by calling   */
/*  the Write_Graph() function                                */

main(argc,argv)
int argc;
char *argv[];

{
   struct cluster *v;
   int size;
   if (argc != 2)
   {
      printf("\n error : Input file expected.. \n");
      exit(1);
   }
   /*      Trying to open the Input File                  */

   if (( fp=fopen(*(argv+1),"r")) == NULL )
   {
      printf("\nCan't open Input File.. \n");
      exit(1);
   }
        /* Reading the input */
   Input();
        /* Mapping */
   map(start_rep,start_spec);
        /* Outputs the result of mapping */
   printf("After Mapping\n");
   printf("\nRepresentation Clusters are : \n\n");
   Write_Graph(start_rep);
   printf("\nSpecification Clusters are : \n\n");
   Write_Graph(start_spec);
}
```

REFERENCES

[1] G. Agha, *Actors: A Model of Concurrent Computations in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] F. Berman, and B. Stramm, "Prep-P: Evolution and Overview," Technical Report CS89-158, Department of Computer Science, University of California, San Diego, CA, 1987.

[3] S. H. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, March 1981.

[4] S. H. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publisher, 1990.

[5] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Transactions on Computer*, vol. 15, no. 6, pp. 50–56, 1982.

[6] H. El-Rewini, and T. G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 138–153, June 1990.

[7] M. Eshaghian, "Cluster-M Parallel Programming Model," In *Proceedings of 6th International Parallel Processing Symposium*, Beverly Hills, California, March 1992.

[8] M. Eshaghian, and M. Shaaban, "A Cluster-M Based Mapping Methodology," *Proceedings of 7th International Parallel Processing Symposium*, Newport Beach, California, April 1993.

[9] M. Eshaghian, and M. Shaaban, "Cluster-M Parallel Programming Paradigm," *International Journal of High Speed Computing*, Accepted for publication.

[10] I. Foster, and S. Tuecke, "Parallel Programming with PCN," Technical Report, Argonne National Laboratory, University of Chicago, January 1993.

[11] A. Gerasoulis, S. Venugopal, and T. Yang, "Clustering Task Graphs for Message Passing Architectures," In *Proceedings of 4th ACM International Conference on Supercomputing*, Amsterdam, Netherlands, vol. ICS 90, pp 447-456, June 1990.

[12] S. Lee, and J. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Transactions on Computers*, vol. C-36, pp. 433–442, April 1989.

[13] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, and J. A. Telle, "Oregami: Software Tools for Mapping Parallel Computations to Parallel Architectures," In *Proceedings of International Conference on Parallel Processing*, Pheasant Run Resort, Du Page County, St. Charles, Illinois, pp. 88-92, August 1990.

REFERENCES
(Continued)

[14] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox, "Mapping Realistic Data Sets on Parallel Computers," In *Proceedings of 7th International Parallel Processing Symposium*, Newport Beach, CA, pp. 123–128, April 1993.

[15] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 85–93, January 1977.

[16] J. Yang, L. Bic, and A. Nicolan, "A Mapping Strategy for MIMD Computers," In *Proceedings of International Conference on Parallel Processing*, Pheasant Run Resort, Du Page County, St. Charles, Illinois, August 1991.

[17] T. Yang, and A. Gerasoulis, "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors," In *Proceedings of IEEE Scalable High Performance Computing Conference*, April 1992.