Contents

Chapter 1

INTRODUCTION

Minimizing the execution time of a job through load balancing in a network of processors is a current research topic in parallel and distributed computing. In this project, load balancing schemes for a hierarchical interconnection network of hypercubes, called Extended Hypercube, have been studied. Modifications and improvements over some popular load balancing algorithms are also proposed in this project.

1.1 Introduction to Load Balancing

In a distributed multiprocessor system, load balancing refers to the *equitable distribution of work load among all the processors in the network*. *Work load* of a processor element (PE) at a particular instant is defined as the total time needed to complete execution of all the tasks waiting at that PE at that instant (including the task, if any, that is currently being executed on the PE). This time includes the total computation time of all these tasks and the various delays such as communication delay, synchronization delay and network queueing delay associated with the execution of the tasks. Often the interaction patterns of these tasks with tasks on other PEs are not known in advance. Again, we may not have apriori estimates of the computation times of these tasks. As a result, calculation of work load using the above definition is not possible. In such cases, we have to adopt heuristic definitions of work load. For

example, work load of a PE is often defined by the number of tasks waiting at that PE. In what follows, we shall use the terms task and job interchangably to denote a unit of work, for example, the execution of a program.

Load balancing schemes may be distinguished according to two criteria. The first criterion refers to the site(s) where load balancing decisions are taken. In a *centralized scheme*, the decisions are carried out in one (or more) central processor(s) or controller(s) in the network. The central processor(s) must have *global* information, i.e. information about work loads of all the PEs in the network. In a *distributed scheme*, each PE takes part in the load balancing activity. The information each PE uses for making a load balancing decision are *local* to the PE, for example, the loads of PEs adjacent to it in the network.

The second criterion is based on the knowledge of the apriori estimates of work distribution. In case we have this knowledge, we can use an off-line algorithm a priori to secure the goal of load balancing. This is called *static load balancing*. In most practical situations, however, we cannot make apriori estimates of load distribution. We must, therefore, have algorithms to work in a dynamic environment of continual and unpredictable arrival of new jobs to the PEs and departure of load through execution of jobs. Such algorithms are called *dynamic* or *adaptive load balancing* algorithms [11].

The static load balancing scheme consists of two parts. The first part is called *task partitioning* which involves breaking the task into subtasks or computation modules. Since the subtasks are dependent on one another, there must be interaction among them. This interaction pattern can be depicted by a directed acyclic graph, called *task graph* (TG), with the nodes representing the subtasks and the edges representing the precedence relations, i.e. communications, among them. If a subtask needs communication of some data from another subtask, we say that there exists a precedence relation between the two subtasks. In the second part of the static load balancing problem, we have to map the subtasks onto the different PEs in the network. This is called *static task mapping* and should satisfy the objective of minimizing the completion time, called the *makespan*, of the total task. The optimal

solution of the static task allocation problem depends on the task graph, the network interconnection pattern of the PEs, the speeds of the network and of the PEs and also on the order in which the subtasks mapped on each PE are executed (the determination of this order is called *static task scheduling*). An optimal solution to the static task mapping and scheduling problems has been proved to be NP-complete. As a result, heuristic algorithms are used to obtain suboptimal solutions. Sometimes it is possible to construct polynomial time optimal algorithms under several constraints on the general problem.

The working of a dynamic load balancing scheme is based on the idea of *migration* of excess load from heavily loaded PEs to lightly loaded ones. The first problem is to determine when to migrate a task. This decision is typically based on local load situation, for example, a comparison of a processor load with the load of the neighboring PEs. After this decision is complete, we have to determine the PE to which the task is to be migrated. As an example, the excess load may be migrated from the PE to the least loaded adjacent PE. Dynamic load balancing schemes may be further classified according to the initiator of the load balancing activities, viz. sender-initiated, receiver- initiated and combined.

1.2 Our Approach to Load Balancing

In what follows, we shall assume that tasks arrive randomly at the PEs in the network. *These tasks are independent of one another*, i.e. there is no communication among these tasks. Each task can, however, be split up into subtasks. We will not deal with the problem of task partitioning and shall assume that the task graph is provided to us. We, therefore, have a combination of the static and dynamic load balancing cases. Whenever a new task arrives, we have to use some static load balancing algorithm to map the subtasks of the corresponding task graph to the PEs of the network. Secondly, considering each task (or alternatively, each subtask) as a unit of load we have to achieve dynamic load balancing. Obviously the two problems are dependent on one another. However, for the sake of simplicity, we will consider the two problems

separately. The assumption of independence of the tasks of one another proves vital in this respect. In the dynamic case, it implies that the tasks can be executed in any order with each of these tasks mapped onto any PE of the network. Also each task has an independendent entity with its own encapsulated data. In order to migrate a task from one PE to another, it is only necessary communicate the data associated with the task. Similarly, for static allocation of the subtasks of a task, we can disregard the existence of other tasks in the PEs and can map the subtasks assuming that the PEs are initially empty (i.e. unloaded).

1.3 State-of-the-Art in Load Balancing

A variety of static and dynamic load balancing algorithms have been proposed in the literature. They widely vary in complexity, techniques adopted, system modelling, and so on. In this section we intend to survey some of the work reported in this area in the recent past.

One very common approach to static task scheduling is called *priority-list scheduling* [3-5]. Here all the schedulable nodes of the task graph are placed on a list which is sorted according to some priority function associated with each node. The node with the highest priority is scheduled on a PE that seems to be the most appropriate one for carrying out the task. Three different heuristic functions to determine the priority of a subtask are described in detail in [3,4]. A brief descriptions of these three heuristic functions is as follows. The *heaviest node first* heuristic assigns the subtasks of the task graph level by level and at each level it assigns the heaviest node (i.e. the node with the largest execution time) to the lightest PE (i.e. the PE for which the total time needed to finish all the subtasks currently assigned to it is minimum). The *critical path method* (CPM) works on the determination of the longest path (called the exit path) from each node to an exit point (i.e. a node of outdegree zero) of the task graph. The *weighted length algorithm* is a modification over the CPM algorithm, because the weighted length algorithm, unlike the CPM, takes into account the branching factor of each node (a term depending on the children of the node).

These three algorithms neglect the communication among the subtasks. [5] describes modified CPM algorithms which take into account the inter-module delays. Other modifications of the priority-list scheduling method include the LAST algorithm [6] and the PDTS algorithm [7].

[8] depicts a divide-and-conquer strategy to minimize the total interprocessor communication cost within specified tolerance limits of lack of balance of work load among the PEs. The authors also delineate *simulated annealing* as a tool for solving the problem of static task mapping. We will refer to this paper again in chapter 3.

In [9] a linear time algorithm, called the *join latest predecessor* algorithm, has been proposed to optimally solve the static task scheduling problem in linear time under several assumptions and constraints, like in-forest or out-forest precedence (the task graph has the form of a forest of in-trees or out-trees), sufficient processors (i.e. there are enough processors to run any available task at any moment), short communication delays, fully connected network topology etc. In [10] the static load balancing problem is formulated as a non-linear, non-convex, non-separable, minimax, resource allocation mathematical programming problem for which the authors have provided an analytical solution.

The literature for dynamic load balancing algorithms is equally rich and full of a number of ideas and techniques. The *diffusion scheme* [11] works on exchange of work load among each pair of adjacent PEs by an amount proportional to the work load difference in the PEs of the pair. A similar algorithm based on *edge coloring of graphs* has been proposed in [12]. The basic *gradient model* [13] migrates load units to direct neighbors in the direction of the nearest low loaded processor. The control information is gathered by the approximation of a gradient field. An extension of this basic gradient model has been proposed in [14], where analogous to the pressure surface, a suction surface is also defined. A unit capacity *network flow model* to solve the load balancing problem for a hypercube has been suggested in [15].

Bidd-average algorithm [16] works on the calculation of the average of the local load of a PE and the loads of all neighbors (adjacent PEs). If the local load is greater than the average, a task is sent to a randomly selected neighbor. In the *threshold*

algorithm [17], on the other hand, a PE probes some other PEs in the network (not necessarily neighbors) whenever certain threshold values are exceeded by the local load. If any probed PE is found suitable for a task transfer, a task is migrated. [18] proposes a variation of the threshold algorithm in which a PE always tends to transfer a newly arrived task irrespective of the local load. A special algorithm for a hypercube is described in [19] which achieves load sharing by balancing across pairs of PEs along each of the dimensions of the hypercube.

Another class of dynamic processor allocation algorithms, called *partitioned allocation algorithms*, has attracted much research attention. In a partitionable hypercube [21-24] or mesh connected [20] system, a subcube or a rectangular submesh is assigned to a newly arrived job. These algorithms are designed to achieve dynamic maintainence of the subcubes and the submeshes under continual arrival and departure of tasks.

1.4 Organization of the report

This project work deals with the study of static and dynamic load balancing algorithms for an Extended Hypercube. In the next chapter, we shall describe the Extended Hypercube (EH), a hierarchical interconnection network of hypercubes, and address the issues of the load balancing schemes to work on this network. The chapters following the next one describe algorithms for load balancing on this architecture. The algorithms which we use for the EH are modified and improved versions of some well-known algorithms. The modifications are intended to improve the performance of the algorithms in terms of speedup and to give the algorithms a hierarchical structure so as to exploit the multi-level connection among the processors of an EH. In chapter 3, we discuss three algorithms for static task allocation on an EH. Chapter 4 describes in detail the threshold algorithms for dynamic load balancing. Chapter 5 deals with a modification of the conventional threshold algorithm that is suitable for the EH. We conclude the report in Chapter 6.

Chapter 2

LOAD BALANCING ON EXTENDED HYPERCUBE : AN OVERVIEW

2.1 The Topology of EH

An extended hypercube EH(k,l) [1,2] with parameters k and l can be defined recursively in terms of 2^k EH(k,l–1)'s. The second parameter l is called the *level* of the EH(k,l). An EH(k,0), that is, an extended hypercube of level 0, comprises of a single *computation processor*. This processor is also called the *root* of the EH(k,0). The basic interconnection module among the roots of EH(k,l) and EH(k,l–1), for $l \geq 1$, is shown in Fig 2.1. The roots of 2^k EH(k,l–1)s are connected in the form of a k-dimensional binary hypercube (k=3 in the figure). The links forming the binary cube are called *hypercube links*. The EH(k,l) comprises of the 2^k EH(k,l–1)s, the binary cube formed by the roots of the EH(k,l–1)s, an additional processor which forms the root of the EH(k,l) a link from this processor to the root of each of the 2^k EH(k,l–1)s. The links connecting the root of the EH(k,l) to the roots of the 2^k EH(k,l–1)s are called *control links*. Fig 2.1 also shows the complete topology of an EH(3,1), since as mentioned earlier, an EH(3,0) comprises of a single processor forming the root of the EH(3,0). Fig 2.2 shows the interconnection network of an EH(3,2). As per definition,

Figure 2.1: The basic interconnection module of an EH

Figure 2.2: The interconnection network of EH(3,2)

EH(3,2) consists of $2^3 = 8$ EH(3,1)s with their roots connected as a 3-dimensional binary cube, a processor labelled 0 at the root of the EH(3,2) and 8 links from this root processor to the roots of the EH(3,1)s. All tasks that come to the network are serviced only by the computation processors at level 0. These processors are, therefore, called *servers*. All other processors, i.e. processors at levels 1 through l, are communication processors that take part in coordinating the activities of the servers and hence these are called *network controllers* (NCs).

As follows from the above description of EH, the conrol links of an EH(k,l) form a complete 2^k-ary tree of level l. Hypercube links connect sets of 2^k nodes such that all of the 2^k nodes in a set are at the same level of the tree and have the same parent. This is illustrated in Fig 2.3. A lot of important topological properties of EH(k,l) follow from this description.

Total number of interior nodes in the tree
= Total number of NCs in the EH(k,l)
$= 1 + 2^k + 2^{2k} + \cdots + 2^{(l-1)k}$
$= (2^{lk} - 1)/(2^k - 1)$

Total number of exterior nodes in the tree
= total number of servers in the EH(k,l)
$= 2^{lk}$

Hence,
Total number of nodes in the tree
= Total number of processors in the EH(k,l)
$= (2^{lk} - 1)/(2^k - 1) + 2^{lk}$
$= (2^{(l+1)k} - 1)/(2^k - 1)$

This implies that
Total number of arcs in the tree

Figure 2.3: The interconnection network of EH(3,3)

= Total number of control links in the EH(k,l)
= $(2^{(l+1)k} - 1)/(2^k - 1) - 1$

We also observe that
Total number of hypercubes in the EK(k,l)
= $1 + 2^k + 2^{2k} + \cdots + 2^{(l-1)k}$
= $(2^{lk} - 1)/(2^k - 1)$

Total number of links in each k-dimensional hypercube
= $k.2^k - 1$

Hence,
Total number of hypercube links in an EH(k,l)
= $[(2^{lk} - 1)/(2^k - 1)](k.2^k - 1)$

Finally, we notice that
Total number of links incident on each server
= k + 1,

Total number of links incident on each NC at level 1 through (l–1)
= $2^k + k + 1$,

and

Total number of links incident on the NC at the root
= 2^k

Before we describe addressing of nodes in an EH and routing among servers, let us introduce some terms that we will use in later chapters. A *subEH* with parameters k and i rooted at an ith level processor of the EH(k,l), for $0 \leq$ i $\leq$ l, is an EH(k,i) which is a subgraph of the EH(k,l) and which has the ith level NC at the root.

In analogy with the tree of control links, all processors (NCs as well as servers) at levels 0 through (i–1) in this subEH are said to be *beneath* the root of the subEH or the *descendents* of the same. Among those descendents, the processors at level (i–1) are said to be *directly beneath* of or the *children* of the root of the subEH. Conversely, the root of the EH(k,i) is called the *ascendent* or *ancestor* of all the nodes that are beneath the root. Similarly, the root of the EH(k,i) is called the *parent* of the nodes that are directly beneath the root. Two processors are said to have the *lowest level common ancestor* (LLCA) at level j, iff the smallest subEH (i.e. the subEH of minimum level) to which both of these processors belong, has a level of j. For example, two servers having the same parent (respectively, grandparent) are said to have the LLCA at level 1 (respectively, 2). For each node in an EH, there is a unique path from the node to the root such that each edge of the path connects a node with its parent. Therefore the LLCA of two nodes is a unique node in the EH. Any communication of data or message from a node to its ascendent (respectively, descendent) is called an *upward* (respectively, a *downward*) communication. On the other hand, communication between processors belonging to the same binary cube is called *hypercube communication*.

2.2 Addressing of Nodes in an EH

The root of an EH(k,l) is addressed as node 0. Any node of the EH other than the root is the member of a hypercube and has a unique parent. We know that the nodes in a k-dimensional binary hypercube can be numbered from 0 through (M–1) (where $M = 2^k$) such that each pair of two nodes adjacent along the ith dimension differ only in the ith bit of the binary representations of their addresses. Suppose a node (not at the root) of an EH(k,l) has a parent whose address is P. The node itself has an address of N in the binary cube to which it belongs, where $0 \leq N \leq (2^k - 1)$. Then the node is addressed by concatenating N at the end of P, i.e. by PN. See Figs 2.2 and 2.3 for the illustration of this addressing mechanism. For an EH(3,1), the 8 children of the root are addressed as 00, 01, ..., 07, for example. The eight children of the

node 01 are addressed as 010, 011, ..., 017, and so on. In general, a node at level i (0≤ i ≤l) is addressed by a string $D_l D_{l-1} \ldots D_i$, where
$D_l = 0$ and
$0 \leq D_j \leq 2^k - 1$, for all j, $i \leq j < l$.
In particular, a server at level 0 is addressed by $D_l D_{l-1} D_{l-2} \ldots D_0$.

We can treat an address $D_l D_{l-1} \ldots D_i$ as an (l–i+1)-digit number of base 2^k. In decimal representation this corresponds to

$$D_l D_{l-1} \ldots D_i \equiv 2^{(l-1)k} \cdot D_l + 2^{(l-i-1)k} \cdot D_{l-1} + \cdots + 2^k \cdot D_{i+1} + D_i \tag{2.1}$$

Obviously, the decimal representations of any two nodes at the same level i cannot be identical. But two nodes at different levels may have the same decimal representation. For example, nodes addressed 0, 00, 000, ..., all have the same decimal representation, viz 0. Therefore, the decimal representation together with the level number of a node, uniquely specifies the node. Fig 2.4 explains this addressing mechanism. In what follows, we shall follow this addressing scheme. But we must remember that the two addressing schemes are identical.

It follows that a node at level i with address N has 2^k children at level (i–1) whose addresses are
$2^k \cdot n + j, \ for \ 0 \leq j \leq 2^k - 1$ Conversely, a node at level i with address N has the parent at level (i+1) whose address is (N mod 2^k).

Therefore, two different servers S_1 and S_2 have the same parent if S_1 mod 2^k = S_2 mod 2^k. They have the same grandparent if S_1 mod 2^{2k} = S_2 mod 2^{2k}. In particular, S_1 and S_2 have the LLCA at level i, iff S_1 mod $2^{jk} \neq S_2$ mod 2^{jk}, for $j = 1, 2, \cdots, (i-1)$ and S_1 mod $2^{ik} = S_2$ mod 2^{ik}. The level of the LLCA of two servers with addresses S_1 and S_2 can, thus, be found using the following algorithm.

find_level_of_LLCA (S_1, S_2)
begin
    for i=1 to l do

Figure 2.4: Addrressing of nodes in an EH(3,2)

```
            if (S1 mod 2^ik = S2 mod 2^ik)
            then
                return(i);
        return(error);
    end;
```

This is an O(l) algorithm, since the "for" loop can be executed at most l times.

2.3 Message Routing between two Servers

Let us suppose that a server S_1 wants to communicate with another server S_2. If S_1 and S_2 are in the same k-cube (i.e. have the same parent), there are two ways by which the communication can occur. First, the message may be sent using a routing strategy for the k-cube. This mechanism does not involve the parent NC of S_1 and S_2. The second strategy sends the message from S_1 to its parent and then from the parent of S_1 (and also of S_2) to S_2. With the former strategy, the maximum number of hops is k, whereas with the latter strategy, the number of hops is always 2. But in the latter strategy, the parent has to take part in all communications within the k-cube directly beneath it. As a result, congestion may occur at the parent NC. To avoid this congestion, we will follow the former routing strategy, i.e. the hypercube routing strategy.

Secondly, suppose that S_1 and S_2 have different parents P_1 and P_2, but the same grandparent G. S_1 and S_2 are now in different k-cubes. If all edges incident on P_1 are removed from the EH, then the k-cube containing S_1 gets disconnected from the k-cube containing S_2. Hence all paths connecting S_2 to S_1 must pass through P_1 and similarly through P_2. Hence S_1 sends the message to P_1. P_1 and P_2 are in the same k-cube. Hence, as in the previous case, P_1 can send the message to P_2 either by using k-cube routing algorithm or along the links $P_1 - G$ and $G - P_2$. The latter strategy involves G and may cause congestion at G. We, therefore, follow the former routing strategy. After the message reaches P_2, the link connecting P_2 and S_2 is used

to complete the communication.

In general, suppopse that S_1 and S_2 have the LLCA at level i. Let this common ancestor of S_1 and S_2 be P_i. As stated in section 2.1, there are unique paths $P_iP_{i-1,1}P_{i-2,1}\ldots P_{1,1}S_1$ and $P_iP_{i-1,2}P_{i-2,2}\ldots P_{1,2}S_2$ from P_i to S_1 and S_2 such that each edge of the paths connects a node to its parent. All the nodes $P_{j,k}$, for $j = 1, 2, \ldots, (i-1)$ and $k = 1, 2$, must take part in the communication from S_1 to S_2. Therefore, (i–1) upward communication hops takes the message from S_1 to $P_{i-1,1}$. $P_{i-1,1}$ then communicates with $P_{i-1,2}$ using hypercube communication. Finally, (i–1) downward hops transmit the message from $P_{i-1,2}$ to S_2. The message routing procedure is as folows :

```
Message_Rout ( S1,S2,msg)
begin
    lev = find_level_of_LLCA( S1,S2);
    P1 = S1 mod 2^((lev-1)k);
    P2 = S2 mod 2^((lev-1)k);

    for i = 1 to (lev-1) do
        send msg from node ( S1 mod 2^((i-1)k)) at level (i–1) to
        node ( S1 mod 2^(ik)) at level i;

    hypercube communication of msg from P1 to P2;

    for i = (lev–1) downto 1 do
        send msg from node (S2 mod 2^(ik)) at level i to node
        (S2 mod 2^((i-1)k)) at level (i–1);
end;
```

With this routing procedure the distance between S_1 and S_2 in terms of number of hops is given by

$$distance(S_1, S_2) = 2(lev - 1) + hamming\_distance(P_1, P_2) \tag{2.2}$$

where hamming distance between P_1 and P_2 is the number of bits in which binary representations of P_1 and P_2 differ; this is the number of hops for the hypercube communication from P_1 to P_2.

Since the maximum values of lev and hamming_distance are l and k respectively, in an EH(k,l), we have from Eqn (2.2),

$$distance \leq 2(l - 1) + k \tag{2.3}$$

2.4 Load Balancing on an EH

We recapitulate that the servers (processors at level 0) carry out all the tasks that come to the network. It is the responsibility of the NCs at levels 1 through l to coordinate the activities of the servers. The tasks of the NCs are the following :

i) NCs are used as I/O processors. External jobs come to the NCs and not to the servers. The NCs distribute the tasks among the servers of the network.

ii) NCs take part in communication between two servers, as described in the previous section.

iii) NCs perform load balancing, i.e. they ensure uniform and equitable distribution of work load among all the servers of the network.

We mention that the load balancing activities are carried out in sites separate from the sites of execution of jobs. This can increase the speedup in an EH as compared to the nonhierarchical networks like meshes and binary cubes, since in a mesh or a binary cube, no additional control processors are present and hence the servers have to take load balancing decisions along with the execution of jobs.

Let us now address the issues of static and dynamic load balancing on an EH separately.

Figure 2.5: A task graph

2.4.1 Static load balancing

The problem of static load balancing on an EH can be stated as follows :

Given a task graph showing the computation modules of a task and the dependence relationship among them, it is desired to assign the subtasks to the servers of an EH(k,l) such that the total communication overhead associated with the execution of the task is minimized and such that congestion at higher level NCs can be avoided or reduced as much as possible, subject to the constraint that the total work load is uniformly distributed among the servers of the network.

If two subtasks need to comunicate, it is convenient to map the two subtasks onto the same server, since communication overhead will be zero in that case. But this does not take into account the load balancing constraint. As an example, consider the task graph of Fig 2.5. Let T_1 be mapped onto server 0. T_2, T_3 and T_4 need data from T_1 and hence all of those subtasks should be mapped onto server 0. Proceeding in this way, we can show that the communication overhead will be minimum, viz 0, if all

subtasks are mapped onto the same server. This is true for all task graphs. But this mapping seriously violates the load balancing constraint. Hence a weighted sum of the total communication cost and cost due to load imbalance should be minimized. While communications are permitted by mapping different subtasks on different processors, we must strive for reducing communication through higher levels of NCs, because an NC at level i has to handle communications from and to 2^{ik} servers — a number which grows exponentially with i. As a result, the problem of congestion becomes more serious with increase in the level of the NC. Communication costs must be adjusted to reflect effects due to the congestion problem.

Static load balancing algorithms are off-line algorithms and hence are generally centralized ones. These can be executed at the respective NCs at which the new tasks arrive. The algorithms can, however, be parallelized and distributed over different NCs. Tree structure algorithms (for example, divide-and-conquer algorithms) are particularly mentionable in this respect. Suppose a task arrives at the root of an EH. The root divides the task among its 2^k (or less) children. Each child, in its turn, divides the task among its children and so on. Algorithms that exploit this hierarchical property of the network are very suitable as static load balancing algorithms on an EH. We will describe these algorithms in detail in chapter 3.

2.4.2 Dynamic load balancing

While studying dynamic load balancing on an EH, we will, for the sake of simplicity, consider each task as an independent and indivisible entity with encapsulated data, that needs no interaction with other tasks existing in the system. With this approximate model, the dynamic load balancing problem on an EH can be stated as :

In a dynamic environment of continual arrival of jobs to the NCs and of departure of jobs through execution at the servers, the problem is to migrate tasks from heavily loaded servers and NCs[1] to lightly loaded servers and NCs so that at all times

[1]load of a server and an NC will be defined in Chapter 4.

the work load is uniformly distributed among all the servers in the network.

In case of dynamic load balancing, load situation changes with time at every node in the network. It is, therefore, inconvenient, if not impossible, for a single central processor to keep track of the varying load situations at all servers. Dynamic load balancing algorithms are thus in general distributed ones and are executed at every node. An EH on the other hand employs a combination of centralized and distributed schemes to secure the goal of load balancing. All the 2^k nodes in each k-cube of servers at level 0 have the same parent NC at level 1. Since a connection exists between an NC at level 1 and each server beneath it, it is possible for the NC to keep track of the load situations of all the child servers. Moreover, for an EH, management of these information is convenient, because new tasks arrive at the NCs and the NCs distribute the jobs among the servers. As a result, each NC at level 1 employs some centralized scheme for dynamic task allocation among its 2^k children. But the NC is not connected by direct links to servers that do not belong to the k-cube beneath it. As a result, it is not convenient for the NC to keep information about all servers of the EH(k,l). Therefore, a distributed load balancing scheme is required for the entire EH(k,l) as a whole.

2.5 Conclusions

An EH(k,l) is a recursively defined network consisting of computation processors called servers, IO processors called network controllers (NCs) and a hierarchical interconnection network among the processors. External tasks arrive at the NCs and are distributed by the NCs for execution among the servers of the network. The NCs in an EH take part in load balancing activities in parallel with the computation of servers. A static load balancing algorithm for an EH should be designed to minimize the sum of the communication overhead of the task and the cost due to load imbalance. At the same time the algorithm must attempt to minimize the possibility of congestion at higher levels. Static load balancing algorithms can be executed on a single NC or may be distributed over the NCs. The EH employs a combination of centralized and distributed schemes for dynamic load balancing.

Chapter 3

STATIC LOAD BALANCING ALGORITHMS ON AN EH

3.1 Introduction

In this chapter we describe three different static load balancing algorithms which have been implemented on an EH. Input to each algorithm is a task graph and a given EH architecture. The algorithm outputs a mapping function that maps the subtasks onto the servers of the given network. Associated with each mapping is a cost reflecting the combined contribution of communication overhead and load imbalance that correspond to the execution of the task in accordance with that mapping. Each algorithm starts with an initial mapping (for example, a random one) and performs changes on the mapping function so as to decrease the overall cost function. Before describing the algorithms in detail let us introduce some notations in the next section.

3.2 Some Notations

Let the given task graph be $\mathcal{G}(\mathcal{V}, \mathcal{E})$ where

$$| \mathcal{V} | \quad = \quad total\ number\ of\ subtasks\ in\ the\ task\ graph$$

$$= \quad N, \; say \tag{3.1}$$

We can address the subtasks by the integers 0 through $(N-1)$. The set of the subtasks is then $\mathcal{V} = \{0, 1, \ldots, \text{N-1}\}$. The subtask addressed by j is labelled by the computation overhead of that subtask and we will denote this as $comp_j$, for j = 0, 1, ..., N-1. Each edge, on the other hand, is labelled by the amount of communication between the subtasks connected by the edge. We shall denote the communication corresponding to an edge $(i, j)\epsilon\mathcal{E}$ by $comm_{ij}$.

The set of processors will be denoted by $\mathcal{K} = \{0, 1, \ldots, \text{P-1}\}$ whose cardinality is

$$| \mathcal{K} | = P = 2^{lk}, \tag{3.2}$$

for an EH(k,l).

The mapping function $map : \mathcal{V} \rightarrow \mathcal{K}$ maps each $i\epsilon\mathcal{V}$ to some $j\epsilon\mathcal{K}$. This is denoted as

$$map(i) = j \tag{3.3}$$

For each $i\epsilon\mathcal{V}$, $map(i)$ can take P different values, since the subtask i can be mapped onto any one of the P processors of the network. Hence the total number of mappings possible is P^N which is an exponential function of N. Even for a reasonably small task graph with 200 subtasks and EH(3,2) as the architecture, the total number of mappings is $64^{200} = 10^{361}$. It is impractical to search the entire domain of the mapping function. We should, therefore, seek for some heuristic algorithms that give only suboptimal solutions within reasonable amount of time.

3.3 Cost Functions

The total computational work load of a processor j, denoted by $load_j$, is the sum of computation overheads of all subtasks that map onto processor j. Thus $load_j$ can be

written as

$$load_j = \sum_{\substack{i \epsilon \mathcal{V} \\ and\ map(i)=j}} comp_i \tag{3.4}$$

for each $j \epsilon \mathcal{K}$.

The average load of a processor is given by

$$load^* = \frac{1}{P} \sum_{j \epsilon \mathcal{K}} load_j = \frac{1}{P} \sum_{i \epsilon \mathcal{V}} comp_i \tag{3.5}$$

In the ideal case

$$load_j = load^* \tag{3.6}$$

for all $j \epsilon \mathcal{K}$, that is, the total work load is perfectly balanced among all the servers. In practice, however, the processor loads are scattered about the mean value ($load^*$). The larger the deviations of the processor loads from $load^*$ are, the poorer is the balance of load among the servers. Therefore, any measure of dispersion of $load_j$ about the mean value $load^*$ can be used as a metric of the amount of load imbalance corresponding to the mapping function. One such measure is the variance of loadj, which is, given by

$$\frac{1}{P} \sum_{j \epsilon \mathcal{V}} (load_j - load^*)^2 = \frac{1}{P} \sum_{j \epsilon \mathcal{V}} load_j^2 \ - \ (load^*)^2$$

For a given task graph $load^*$ is constant (i.e. independent of the map) and hence

$$C_1 = \sqrt{\frac{1}{P} \sum_{j \epsilon \mathcal{V}} load_j^2} \tag{3.7}$$

can serve as a measure of cost due to load imbalance. The square root has been taken to make the value of C_1 of the same order of magnitude as $load^*$. This means that the best and worst values of C_1 are proportional to the value of $load^*$. In order to see how this happens let us first consider the ideal case of perfect load balancing (Eqn 3.6). In this case

$$C_1 = load^* \tag{3.8}$$

On the other hand, the worst value of C_1 occurs when all subtasks are mapped onto the same processor, say processor 0. That is, $load_0 = P \cdot load^*$ and $load_j = 0\ for\ 0 <$

$j < P$. Therefore,

$$C_1 = \sqrt{P} \cdot load^* \tag{3.9}$$

We also notice that the standard deviation of loadj is the square root of variance of $load_j$, and is given by,

$$\sigma = \sqrt{C_1^2 - (load^*)^2} \tag{3.10}$$

Another possible measure of dispersion of loadj is the arithmetic mean of the normalized absolute deviation of $load_j$ from $load^*$ given by,

$$\delta = \frac{1}{P} \sum_{j \epsilon \mathcal{V}} \left(\frac{\mid load_j - load^* \mid}{load^*} \right) \tag{3.11}$$

Let us now take the case of communication overhead associated with the mapping. Each edge $(i,j) \epsilon \mathcal{E}(\mathcal{G})$ corresponds to a communication of data or message of amount equal to $comm_{ij}$ from subtask i to subtask j. The contribution of this communication to the total communication cost is

$$pen_{map(i),map(j)} \cdot distance_{map(i),map(j)} \cdot comm_{ij}$$

where

$distance_{map(i),map(j)}$

= distance between the two servers on which the subtasks i and j are mapped according to the routing strategy of section 2.3 (See Eqn 2.2),

and

$pen_{map(i),map(j)}$

= a penalty factor included to reduce the possibility of congestion at higher levels of NCs.

Since subtasks i and j are mapped onto processors $map(i)$ and $map(j)$ respectively, any communication of message or data from subtask i to subtask j is a communication from the server $map(i)$ to the server $map(j)$. This communication follows the routing strategy described in section 2.3. The higher the maximum level reached by the communication is, the larger is the possibility of congestion incurred by the communication, since an NC at level i has to handle communication from and to 2^{ik}

servers beneath it (2^{ik} increases exponentially with i). We have seen in section 2.3 that the communication between two servers reaches a maximum level equal to the level of the LLCA of the servers minus 1. In other words, the level of the LLCA of two servers dictates how far a message has to move up in the hierarchy of NCs to effect any communication between these two servers. Since we want to minimize communication through higher levels in comparison with those at lower levels, we must assign larger costs to communications through higher levels in comparison to those through lower levels. The penalty factor is introduced to reflect the relative importance of the attempt to reduce communications though different levels. Obviously, the penalty factor should increase with the level of the LLCA of $map(i)$ and $map(j)$. We have taken the value of the level of the LLCA of $map(i)$ and $map(j)$ as the penalty factor. The total communication cost is, therefore, given by,

$$C_2 = \sum_{(i,j)\epsilon\mathcal{E}(\mathcal{G})} pen_{map(i),map(j)} \cdot distance_{map(i),map(j)} \cdot comm_{ij} \tag{3.12}$$

where,

$$pen_{r,s} = find\_level\_of\_LLCA(r,s) \tag{3.13}$$

as described in section 2.2.

A weighted sum of C_1 and C_2 is our total cost function to be minimized, that is,

$$cost \ = \ C_1 + \beta \ C_2 \tag{3.14}$$

where β is a crucial parameter indicating the relative importance of the two terms in the total cost function. If β is too large, the communication term will dominate and this will lead to poor load balancing. On the other hand, if β is very small, the communication term will be neglected in comparison with C_1. We have chosen β such that C_1 and βC_2 produce almost equal contributions to the total cost.

Now that we have defined our cost function, we can precisely state the optimization problem as

$$\min_{map} \ (C_1 + \beta \ C_2) \tag{3.15}$$

where C_1 and C_2 can be given as in Eqns (3.7), (3.12) and (3.13).

3.4 Two Approaches to Load Balancing

We mentioned in section 2.4.1 that the optimization problem of static load balancing can be solved in a single processor or may be distributed over different NCs in the network. A tree-like splitting of the problem can help in the second case. Let us now discuss how these two approaches differ.

3.4.1 Unified (or Centralized) Approach

In this approach, the mapping function maps subtasks to servers of the EH(k,l), as described in the previous section. Each iteration step of the algorithm tends to modify the mapping function so as to decrease the cost associated with the mapping function. When the algorithm terminates, the solution directly indicates the way the subtasks are to be assigned to the servers at level 0 of the EH.

3.4.2 Level-by-level Approach

In this approach, the algorithm starts execution at the root of EH(k,l). The mapping function now maps the subtasks onto the 2^k NCs at level (l–1) directly beneath the root at level l. This algorithm works as if it tends to optimize the cost function for an EH(k,1) instead of that for an EH(k,l). That is, the root NC considers the NCs at level (l–1) as servers to which to map the subtasks. As a result, this step of the algorithm minimizes the communcation through the hypercube at level (l–1) and ensures load balance among the 2^k subEHs directly beneath the root. When this step of the algorithm terminates, we are left with 2^k subgraphs of the original task graph mapped onto the 2^k NCs at level (l–1). In each subgraph, only those edges of the original task graph are retained which connect pairs of subtasks belonging to that subgraph only. The edges connecting subtasks in different subgraphs correspond to communications through the hypercube at level (l–1), and the total communica-

tion through this hypercube has already been minimized by the root NC during the execution of the first step of the algorithm.

Each NC at level (l–1) now gets an identical problem as the root has got at the start of the execution of the algorithm. Each NC then splits the subgraph assigned to it into 2^k subsubgraphs for the NCs at level (l–2) that are directly beneath it. In this step of the algorithm also, the optimization algorithm for an EH(k,1) is used. This step minimizes the communication through the hypercubes at level (l–2) and ensures load balancing among the subEHs of level (l–2). This process is repeated level by level, until the NCs at level 1 map the subtasks directly onto the servers, at which stage no further splitting of the subgraphs is necessary. The β value of Eqn (3.14) has to be adjusted at each level, since subgraphs at different levels are of different sizes.

One benefit of the level-by-level approach over the unified approach is that the task allocation algorithm is distributed over the entire network of NCs instead of being executed at a single NC. Secondly, each NC, except the one at level l, gets a subgraph smaller in size than the original graph. Moreover, the mapping function now maps subtasks onto 2^k processors unlike onto 2^{lk} processors as in the unified approach. In a later section, we will show that the overall execution time of the static load balancing algorithm, as well as the total amount of computation, can be reduced in the level-by- level approach in comparison with the unified approach.

3.5 Local Search Algorithm

The local search algorithm starts with an initial map. At each iteration step, the mapping function is perturbed. If the perturbation leads to a mapping with lower associated cost, the change is accepted, or else no change is done to the mapping function. Formally the algorithm can be outlined as follows.

Local_Search
begin

```
map := map_0;
cost := cost_0;

while ("equilibrium not reached") do
begin

    new_map := perturb(map);
    new_cost := cost associated with new_map;

    if (new_cost < cost) then
    begin
        map := new_map;
        cost := new_cost;
    end

end;

return(map);

end;
```

Let us now describe the details of the algorithm. The initial map (map_0) can be generated by randomly mapping the subtasks on the processors or by serially mapping task i on processor (i mod P). The initial cost associated with map_0 is $cost_0$. Within each iteration step of the "while" loop, the current map function is perturbed by randomly selecting a subtask and changing its map function value, that is, by moving the subtask to a separate processor. If this perturbation results in a decrease in the cost function, the map and cost function values are updated by the new values. If the perturbation produces a map with increased cost, the effects of the perturbation are discarded.

Now let us discuss how the equilibrium condition of the "while" loop can be checked. Since each of the N subtasks, mapped on a processor, can be moved to any one of the other $(P-1)$ processors, $N(P-1)$ iterations move each task to a particular processor one time on an average (since the probability that the particular task is selected is $1/N$ and the probability that the task is moved to a particular processor on which it is not currently mapped is $1/(P-1)$). So we can check the cost function after each set of $N(P-1)$ iterations. If the cost function does not change or changes by a very little value (that is, by a value less than a tolerance limit) after some set of $N(P-1)$ iterations, we assume that the equilibrium condition has been reached, that is, no further sizable decrease of the cost is likely to be produced by the continuation of the execution of the algorithm. The "while" loop is thus executed $MN(P-1)$ times for some integer M. Typically, equilibrium condition is reached for values of M between 6 and 10 for the unified approach and between 15 and 20 for the level by level approach.

We have experimented with four task graphs with 200, 400, 600 and 800 subtasks respectively. We have started with different initial maps. We have considered EH(3,2) and EH(2,3) as the extended hypercube architectures. Table 3.1 shows the cost functions associated with the initial mapping (averaged over different map_0) for the four tasks on the two EH architectures. These cost values correspond to the situation before the execution of any load balancing algorithm. COM_j refers to the total communication (without the penalty factor) that is routed through a jth level node at the highest level. Thus, for an EH(2,3)
$C_2 = COM_0 + 2\ COM_1 + 3\ COM_2$.
Tables 3.2 and 3.3 show the cost function values after the local search algorithm is run on the task graphs using the unified approach ($M = 10$) and the level-by-level approach ($M = 20$) respectively. For these values of M we have reached equilibrium for each of the four task graphs. We observe that both C_1 and $\beta\ C_2$ and hence the total cost function have been reduced by the execution of the local search algorithm. For an EH(3,2), COM_0 has increased, whereas COM_1 has decreased. For an EH(2,3), both COM_0 and COM_1 have increased and COM_2 has decreased. Thus the algo-

rithm has reduced communication through higher levels at the expense of increased amount of communication through lower levels. The variations of the cost functions with different values of M for the task graph with 800 subtasks are shown in Tables 3.4 and 3.5 for the unified and level-by-level approaches.

Table 3.1 Four task graphs and their initial cost function values

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	189	189	352	352	514	514	664	664
βC_2	184	185	380	379	593	588	787	785
$cost$	373	374	732	731	1107	1102	1451	1449
$load^*$	168	168	330	330	491	491	640	640
σ	84	84	119	119	146	146	171	171
$\delta(\%)$	41	41	28	28	24	24	22	22
COM_0	905	285	2096	682	3132	1135	4298	1334
COM_1	17989	3178	36931	7127	57765	11155	76501	14766
COM_2	_	22463	_	45494	_	70605	_	94341

Table 3.2 Cost function values after the execution of the local search algorithm (Unified approach with M = 10)

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	172	171	334	333	497	497	649	653
βC_2	115	116	233	237	362	368	481	472
cost	289	287	567	570	859	865	1130	1125
load*	168	168	330	330	491	491	640	640
σ	37	33	55	44	71	74	101	124
$\delta(\%)$	16	16	13	11	11	12	12	16
COM_0	2587	1341	4321	2502	6315	3672	7573	4650
COM_1	10234	4873	21148	8075	33023	12306	44277	16642
COM_2	_	11779	_	25317	_	39620	_	50310

Table 3.3 Cost function values after the execution of the local search algorithm (Level-by-level approach with M = 20)

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	174	170	336	332	505	495	651	644
βC_2	107	114	230	231	343	358	479	489
cost	281	284	566	563	848	853	1130	1133
load*	168	168	330	330	491	491	640	640
σ	45	27	65	39	117	59	116	63
$\delta(\%)$	20	13	16	10	18	10	15	8
COM_0	2249	1211	3485	2125	5667	2911	6187	3185
COM_1	9582	4769	21306	8558	31507	13064	44787	16808
COM_2	_	11579	_	24462	_	38031	_	52972

Table 3.4 Variation of the cost functions with M for
the 800 node task graph ($load^* = 640$)
(Local Search Algorithm : Unified Approach)

	EH(3,2)					EH(2,3)				
	M=2	M=4	M=6	M=8	M=10	M=2	M=4	M=6	M=8	M=10
C_1	648	649	649	649	649	650	649	651	652	653
βC_2	494	485	482	481	481	496	487	480	474	472
$cost$	1142	1134	1131	1130	1130	1146	1136	1131	1126	1125
σ	97	102	105	103	101	109	108	115	123	124
$\delta(\%)$	12	12	13	12	12	14	14	15	16	16
COM_0	8754	8000	7728	7636	7573	5349	4954	4750	4726	4650
COM_1	44994	44471	44310	44285	44277	16652	16577	16514	16683	16642
COM_2	_	_	_	_	_	53208	52220	51419	50533	50310

Table 3.5 Variation of the cost functions with M for
the 800 node task graph ($load^* = 640$)
(Local Search Algorithm : Level-by-Level Approach)

	EH(3,2)					EH(2,3)				
	M=1	M=5	M=10	M=15	M=20	M=1	M=5	M=10	M=15	M=20
C_1	643	648	651	651	651	641	643	644	644	644
βC_2	589	496	485	480	479	613	517	495	490	489
$cost$	1232	1144	1136	1131	1130	1254	1160	1139	1134	1133
σ	53	99	114	115	116	32	52	60	63	63
$\delta(\%)$	7	13	14	15	15	4	7	8	8	8
COM_0	5068	5798	6217	6236	6187	2830	3101	3427	3425	3185
COM_1	56347	47637	45371	44892	44787	16141	15951	16399	16331	16808
COM_2	_	_	_	_	_	69990	57248	53928	53239	52972

Figure 3.1: A one-variable function demonstrating the problem of local minima

3.6 Simulated Annealing Algorithm

The local search algorithm described in the previous section accepts moves only in the direction that reduces the cost. A move increasing the cost is never accepted. As a result, the solution may get stuck in a local minima. To explain this phenomenon let us consider a function f of a single variable x to be minimized. The $f(x)$ versus x relationship is as shown in Fig 3.1. Suppose that we start at an initial value of $x = x_0$, i.e. at a point S. The domain of x for the search is $x_{min} \leq x \leq x_{max}$. We perturb the situation by changing x by a small value from the current value. Since we are initially at x_0, any move decreasing x value cannot be accepted by the local search algorithm, since such a move increases $f(x)$. So our movement is restricted towards right only. Suppose, after some number of iterations, we reach the point A. We note that A is a point of local minima of the function $f(x)$, since $f(x)$ is an increasing function on both sides of A. As a result, no small change of x at this point is accepted by the local search algorithm. That is, the solution gets stuck at this point. The

global minima in the domain $x_{min} \leq x \leq x_{max}$ is, however, at B. The solution can never reach this global minima. This problem can be avoided to a great extent using a simulated annealing algorithm [8].

The solution of Fig 3.1 can get out of the local minima at A, if moves increasing $f(x)$ are accepted so as to enable $f(x)$ to climb the hill at C. This process is analogous to the process of annealing in metallurgy in which a metal is melted and cooled down very slowly. At high temperatures, thermal perturbations may lead the material from lower to higher energy states. As temperature is reduced, hill climbing moves become less and less likely. At very low temperatures, the metal accepts only those moves that decrease its energy and the energy settles down to a minimum value.

Analogously, our optimization problem of load balancing should initially accept hill-climbing moves (i.e. moves increasing the cost function). As the process continues, these moves are made less and less likely. The probability of acceptance of hill climbing moves can be controlled by a parameter T analogous to the temperature of the material in the annealing process. If a move increases the cost function by a value $\Delta c \geq 0$, then the move is accepted with a probability of $\exp(-\Delta c/T)$. We see that the probability decreases as we decrease T keeping Δc constant. Hence we start with a high value of T and decrease it gradually till the probability of hill climbing moves becomes very low. The simulated annealing algorithm [8] can now be outlined as follows :

Simulated_Annealing
begin

$T := T_0$;
$map := map_0$;
$cost := cost_0$;

while ($T > T_{freeze}$) do
begin

```
        while ("equilibrium is not
        reached at current temperature") do
        begin

                new_map := perturb(map);
                new_cost := cost associated with new_map;
                Δc := new_cost − cost;

                if (Δc < 0) then
                begin
                        map := new_map;
                        cost := new_cost
                end

                else
                begin
                        r := random number between 0 and 1;
                        if (r < exp(−Δc/T)) then
                        begin
                                map := new_map;
                                cost := new_cost
                        end
                end

        end;

        T := update(T)

end;
```

return(map)

end;

As in the local search algorithm, we start with an initial map (map_0) and with the corresponding associated *cost* ($cost_0$). The initial temperature T_0 is chosen such that hill climbing moves have high probability of acceptance, say 0.9. Specifically, we identify all the moves from the initial configuration (map_0) that increase the cost, calculate the average increase in cost (Δc_{av}) for these moves, and use the equation $0.9 = \exp(-\Delta c_{av}/T_0)$ to calculate T_0.

The operation of the code segment within the inner "while" loop is the same as that of the local search algorithm except only that in the present case a perturbation increasing the cost function is accepted with a probability of $\exp(-\Delta c/T)$. As in the local search algorithm, the "while" loop is executed $MN(P-1)$ times for some integer M for each execution of the outer loop.

The temperature updation at the last line of the outer "while" loop of the simulated annealing algorithm presented earlier is called the *cooling schedule*. We use the relation $T := 0.9\,T$ for the updation of T. The freezing point is reached when hill climbing moves are very unlikely. Specifically, the freezing point temperature T_{freeze} is one at which a move increasing the cost by unit value has very low probability of acceptance, say 2^{-32}. After the freezing point is reached, the map is output as the solution to the optimization problem.

As in the local search algorithm, we experimented with the four task graphs of Table 3.1. The solutions obtained for these graphs using the unified and level-by-level approaches are shown in Tables 3.6 and 3.7. The variations of the cost functions with M for the 800 node task graph are shown in Tables 3.8 and 3.9.

Table 3.6 Cost function values after the execution of the Simulated Annealing algorithm (Unified approach with M = 3)

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	174	176	340	341	504	512	651	656
βC_2	109	104	219	213	339	321	451	433
cost	283	280	559	554	843	833	1102	1089
load*	168	168	330	330	491	491	640	640
σ	47	52	83	88	109	142	119	145
$\delta(\%)$	20	24	20	21	18	25	19	23
COM_0	3252	1743	6437	3725	9573	5685	12815	7861
COM_1	9222	5604	18707	10114	29129	16408	38728	20978
COM_2	-	9630	-	20446	-	29990	-	41192

Table 3.7 Cost function values after the execution of the Simulated Annealing algorithm (Level-by-Level approach with M = 3)

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	175	171	340	333	511	497	667	648
βC_2	102	105	213	215	320	332	428	441
cost	277	276	553	548	831	829	1095	1089
load*	168	168	330	330	491	491	640	640
σ	49	32	85	50	139	72	185	99
$\delta(\%)$	21	14	20	12	22	12	23	12
COM_0	2163	1237	3508	2039	5521	2701	7066	3536
COM_1	9139	4703	19465	8443	29272	13216	39315	16724
COM_2	-	10509	-	22403	-	34611	-	46477

Table 3.8 Variation of cost functions with M for the 800 node task graph ($load^* = 640$) (Simulated Annealing Algorithm : Unified Approach)

	EH(3,2)			EH(2,3)		
	M=1	M=2	M=3	M=1	M=2	M=3
C_1	650	650	651	656	656	656
βC_2	454	449	451	435	433	433
$cost$	1104	1099	1102	1091	1089	1089
σ	111	113	119	144	145	145
$\delta(\%)$	18	18	19	23	23	23
COM_0	12655	12467	12815	7624	7503	7861
COM_1	39104	38699	38728	21216	20695	20978
COM_2	_	_	_	41268	41447	41192

Table 3.9 Variation of cost functions with M for the 800 node task graph ($load^* = 640$) (Simulated Annealing Algorithm : Level-by-level approach)

	EH(3,2)			EH(2,3)		
	M=1	M=2	M=3	M=1	M=2	M=3
C_1	665	666	667	647	647	648
βC_2	433	427	428	448	443	441
$cost$	1098	1093	1095	1095	1090	1089
σ	179	182	185	94	95	99
$\delta(\%)$	22	22	23	11	12	12
COM_0	7109	6945	7066	3541	3445	3536
COM_1	39757	39268	39315	17323	17009	16724
COM_2	_	_	_	47012	46575	46477

3.7 A Greedy Algorithm

A greedy algorithm accepts moves that are locally optimal. Whenever a subtask is moved, the algorithm selects the best processor to carry out the subtask and assigns that subtask to that processor. In order to find out the best processor for the subtask, we use the cost functions discussed in section 3.3. Suppose, a subtask $i \epsilon \mathcal{V}$ has been selected for a possible reassignment of a processor. The current map provides a mapping for each subtask of the task graph. For all subtasks other than i, the mapping function value is kept unchanged, whereas $map(i)$ is varied. The cost function is calculated for every possible value of $map(i)$. The minimum value of the cost function corresponds to the best processor for serving the subtask i. If the best processor is different from the processor on which the subtask i is currently mapped, the subtask is moved to the best processor. The algorithm can be formally stated as follows.

Greedy
begin

$map := map_0$;
$cost := cost_0$;

while ("equilibrium is not reached") do
begin

for each subtask $i \epsilon \mathcal{V}$ in some order do
begin

$best\_cost := cost$;
$best\_proc := map(i)$;

for $proc = 0$ to $(P - 1)$ do
begin

```
                    if (proc ≠ map(i)) then
                    begin

                        new_map := map;
                        new_map(i) := proc;
                        cost := cost associated with new_map;

                        if (new_cost < best_cost) then
                        begin
                            best_cost := new_cost;
                            best_proc := proc;
                        end;
                    end
                end;

                if (best_proc ≠ map(i)) then
                begin
                    cost := best_cost;
                    map(i) := best_proc
                end
            end
        end;

        output(map)

    end;
```

As in the two algorithms described before, we start with initial map and cost functions. Each execution of the "while" loop tends to move every subtask of the task graph. The order in which the subtasks are considered for possible movement is a predetermined one. For each subtask, the best processor is found out. The

map function value for that subtask is correspondingly changed. The "while" loop may be executed several times, because each execution begins with a different map function and the best processor for the same subtask may be different for different maps. Equilibrium condition is reached when all subtasks are currently assigned to the respective best processors under the current map and hence no subtask can further be moved. The algorithm perturbs the map $(P-1)$ times for each subtask during each execution of the "while" loop. Hence the "if" clause of the innermost "for" loop is executed $MN(P-1)$ times where M is an integer denoting the number of times the "while" loop is executed before the attainment of equilibrium. Typically M varies from 8 to 15.

We experimented with the task graphs of Table 3.1. The results are listed in tables 3.10 through 3.13. We have executed the greedy algorithm with several initial maps and with different orders of selecting the subtasks for movement (in the outer "for" loop of the code). It has been observed that the quality of the solution depends considerably on the initial map and the order of selecting the subtasks for movemetnt. The best and worst solutions obtained for the 800 node task graph are shown in tables 3.12 and 3.13.

Table 3.10 Cost function values after the execution of the greedy algorithm (Unified approach)

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	170	172	332	332	498	500	645	646
βC_2	123	115	245	243	363	361	498	493
cost	293	287	577	575	861	861	1143	1139
load*	168	168	330	330	491	491	640	640
σ	26	37	39	39	80	86	72	76
$\delta(\%)$	12	17	9	9	13	14	9	10
COM_0	2205	1354	3764	2394	5719	3445	6608	4081
COM_1	11231	4600	22653	7744	33476	12134	46476	14936
COM_2	_	11805	_	26403	_	38860	_	54387

Table 3.11 Cost function values after the execution of the greedy algorithm (Level-by-Level approach)

	200 tasks		400 tasks		600 tasks		800 tasks	
	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)	EH(3,2)	EH(2,3)
C_1	173	170	342	331	513	494	673	644
βC_2	109	115	217	239	330	369	438	488
cost	282	285	559	570	843	863	1111	1132
load*	168	168	330	330	491	491	640	640
σ	43	25	90	33	147	46	208	67
$\delta(\%)$	19	12	21	8	23	8	25	9
COM_0	2189	1175	3967	2052	6095	2840	7688	3497
COM_1	9838	4810	19752	8162	29934	12650	39915	16718
COM_2	_	11687	_	25769	_	39889	_	52714

Table 3.12 The best, worst and average solutions for the 800 node task graph ($load^* = 640$) (Greedy Algorithm : Unified Approach)

	EH(3,2)			EH(2,3)		
	average	best	Worst	average	best	worst
C_1	645	648	642	646	654	641
βC_2	498	474	522	493	455	525
cost	1143	1122	1164	1139	1109	1166
σ	72	96	39	76	133	28
$\delta(\%)$	9	11	5	10	17	4
COM_0	6608	7253	6111	4081	4481	3957
COM_1	46476	43812	49165	14936	16600	13206
COM_2	_	_	_	54387	48135	59864

Table 3.13 The best, worst and average solutions for the 800 node task graph ($load^* = 640$) (Greedy Algorithm : Level-by-Level Approach)

	EH(3,2)			EH(2,3)		
	average	best	Worst	average	best	worst
C_1	673	673	667	644	647	641
βC_2	438	433	450	488	470	520
cost	1111	1106	1117	1132	1117	1161
σ	208	206	187	67	88	30
$\delta(\%)$	25	25	24	9	11	4
COM_0	7688	7533	7297	3497	3911	3261
COM_1	39915	39537	41343	16718	18235	14261
COM_2	_	_	_	52714	49252	58766

3.8 Comparison among the Algorithms

Now that we have described three algorithms for static load balancing on an EH, let us discuss the relative performances of the algorithms in terms of solution quality and run time. For the comparison, we will refer to the data obtained using the unified approach. We will conclude this section by a comparison between the unified and the level-by-level approaches.

3.8.1 Solution Quality

All of the three algorithms which we have pesented in the last three sections are suboptimal ones. As mentioned before, it is impractical to solve the static load balancing problem optimally because of the unreasonably long run time of the algorithm that gives the optimal solution after inspecting every possible map. The optimal value of the cost function is therefore unknown to us and we cannot compare the performances of the suboptimal algorithms with that of the optimal algorithm in terms of the value of the cost function. However, we can compare the performances of the three suboptimal algorithms among one another. The basis of the comparison is that the smaller the value of the cost function output by an algorithm is, the closer the cost is to the optimal value. As a result, if the cost output by an algorithm A is smaller than that by algorithm B, we will say that algorithm A gives better solution than algorithm B.

In terms of the total cost function, the simulated annealing algorithm gives best and the greedy algorithm gives worst solutions. The variation is within 1% of the suboptimal cost obtained. It has been observed that better solutions are generally obtained by reductions in the communication cost at the expense of increased load imbalance. The greedy algorithm provides the best load sharing but the highest communication overhead. The simulated annealing algorithm, on the other hand, provides the worst load balance but the least communication overhead. The performance of the local search algorithm is in between the performances of the simulated annealing and greedy algorithms in terms of both load balancing and communication overhead. Figs 3.2 through 3.6 show pictorial comparison of the parameter values

obtained using the three algorithms. At the same time, the values are compared with the initial values before the execution of any load balancing algorithm. Figs 3.2 and 3.3 show the values of C_1, βC_2 and the total cost function obtained by the execution of the different algorithms. Fig 3.4 shows the values of δ (defined in Eqn 3.11) that correspond to the final map output by the three algorithms. Figs 3.5 and 3.6 compare the relative communication volumes through different levels for the three algorithms. The relative communication volume COM'_j is defined to be the value of COM_j output by the algorithm divided by the initial value of COM_j (i.e. the value before the algorithm is executed). In the figures, the abbreviation NL corresponds to the initial cost function values (before load balancing is done), whereas LS, SA and GR refer to the cost function values obtained by the execution of the local search, simulated annealing and greedy algorithms respectively.

3.8.2 Run Time of the algorithms

In section 3.5 we have seen that the map is perturbed for $MN(P-1)$ times for the local search algorithm where M is between 6 and 10. The simulated annealing algorithm perturbs the map for $MN(P-1)$ times at each temperature for M between 1 and 3. Typically, the algorithm is run for 100 different values of the temperature. Hence the entire execution of the simulated annealing algorithm perturbs the map $MN(P-1)$ times for M between 100 and 300. Finally, the greedy algorithm causes $MN(P-1)$ perturbations for M between 8 and 15. For each of the three algorithms, each perturbation involves changing the map and calculating the new cost for this changed map. The acceptance criterion for the perturbations varies from one algorithm to another. We can, however, consider the run time proportional to the number of perturbations done during the execution. In terms of run time, therefore, the local search algorithm is the fastest. The greedy algorithm is almost as fast as the local search algorithm. But the simulated annealing algorithm is 10 to 30 times slower than the other two algorithms.

3.8.3 Consistency of the Results

The solutions obtained with different initial maps in both the local search algorithm and the simulated annealing algorithm do not vary greatly from one another. For the greedy algorithm, however, sizable variations have been observed for different initial maps and orders of selecting the subtasks for reassignment of processors. Typically, we have found as much as 5% variations in the total cost function using the greedy algorithm.

3.8.4 Susceptibility of the solution to the local optima problem

Both the local search algorithm and the greedy algorithm are more susceptible to the local optima problem described in section 3.5 than the simulated annealing algorithm.

In fact, the simulated annealing algorithm has been devised to reduce the effect of this problem. The cost paid for achieving this is a many times slower execution of the simulated annealing algorithm in comparison with the other two algorithms.

3.8.5 Comparison between the Unified and Level-by-Level Approaches

We have not observed any significant difference in the solution qualities obtained using the two different approaches in any of the three algorithms. In some cases, the level-by-level approach gives a slightly better solution than the other approach. This may be because the unified approach uses an overall cost function reflecting combinations of several factors, like load imbalance at the servers' level and communication overheads through all levels. Each step of the level-by-level approach, on the other hand, takes care of some specialized factors, like load imbalance and communication overhead in a particular level. This may be a possible cause of better regulation of individual parameters resulting in better overall solutions in case of the level-by-level approach.

In terms of run time, the level-by-level approach gives a faster algorithm than the unified approach. To see how this happens, we recapitulate that the run time of each algorithm is proportional to the number of perturbations performed on the mapping function. Though the time taken to calculate the cost associated with the perturbed map depends on the size of the task graph, we shall assume in this section that each perturbation requires some constant time on an average irrespective of the size of the task graph. (If we remove this assumption, then the estimated execution time for the level-by-level approach will be smaller than that which we get with the assumption. This implies that the lower bound which we will provide for the speedup of the level-by-level approach over the unified approach is less than the lower bound which can be obtained without the assumption.) We have seen that an execution of each algorithm takes time proportional to $MN(P-1)$, where N is the number of subtasks, P is the number of processors onto which the algorithm maps the subtasks and M is an integer that depends on the algorithm and also on the approach. For the

unified approach, P is 2^{lk} in an EH(k,l). Thus the run time of the unified approach is given by,

$$T_{uni} = M_{uni} \cdot N \cdot (2^{lk} - 1) \tag{3.16}$$

where M_{uni} is the value of M for the unified approach. The level-by-level algorithm runs in several phases. The first phase is executed at the root. The second phase runs in parallel in each (l–1)st level NC. In general, the ith phase is executed in parallel in the (l–i+1)st level NCs. We see that there are l phases for the level-by-level approach, where each phase corresponds to the execution of the algorithm at one level of NCs. Since different NCs at the same level run in parallel in each phase, the total run time of the level-by-level approach is equal to the sum of the run times of l executions of the algorithm with the ith execution occurring in an (l–i+1)st level NC. The root takes time $M_{lev} \cdot N \cdot (2^k - 1)$, where M_{lev} is the value of M for the level-by-level approach. Each NC at level (l–1) gets a subgraph with $N/2^k$ subtasks on an average. So the second phase takes time $M_{lev} \cdot N/2^k \cdot (2^k - 1)$. Similarly, each NC at level (l–2) gets a subgraph of $N/2^{2k}$ subtasks on an average. Therefore, the second phase takes time $M_{lev} \cdot N/2^{2k} \cdot (2^k - 1)$. In this way we can calculate the run time for other phases of the algorithm. Therefore, the level-by-level approach takes a run time given by,

$$\begin{aligned} T_{lev} &= M_{lev} \cdot N \cdot (2^k - 1) + M_{lev} \cdot N/2^k \cdot (2^k - 1) + \\ &\qquad \cdots + M_{lev} \cdot N/2^{(l-1)k} \cdot (2^k - 1) \\ &= M_{lev} \cdot N \cdot (2^k - 1)(1 + 1/2^k + 1/2^{2k} + \cdots + 1/2^{(l-1)k}) \\ &< M_{lev} \cdot N \cdot (2^k - 1)(1 + 1/2^k + 1/2^{2k} + \cdots to\ infinity) \\ &= M_{lev} \cdot N \cdot 2^k \end{aligned} \tag{3.17}$$

The speedup of the level-by-level approach over the unified approach is, therefore,

$$\begin{aligned} speedup &= T_{uni}/T_{lev} \\ &> M_{uni} \cdot N \cdot (2^{lk} - 1)/(M_{lev} \cdot N \cdot 2^k) \\ &\approx \left(\frac{M_{uni}}{M_{lev}}\right) 2^{(l-1)k} \end{aligned} \tag{3.18}$$

In our experiments, it has been observed that

$$M_{lev} \leq 2\ M_{uni} \tag{3.19}$$

Therefore,

$$speedup > 2^{(l-1)k-1} \tag{3.20}$$

Thus for an EH(3,2), $speedup > 4$ and for an EH(2,3) $speedup > 8$.

Finally, suppose that the level-by-level algorithm is executed on a single processor instead of on different NCs. This means that the same processor executes different phases of the algorithm and for each phase the processor serially performs the executions that are performed in parallel on different NCs in the actual execution of the algorithm. In other words, the single processor serially carries out the executions done on all the NCs. In that case the run time T'_{lev} will be proportional to the total number of perturbations produced in the entire execution of the algorithm. In fact, this is a measure of the total computation overhead associated with the execution of the level-by-level algorithm.

The single processor first splits the initial task graph into 2^k subgraphs with $N/2^k$ subtasks in each subgraph on an average. This first phase takes time $M_{lev} \cdot N \cdot (2^k - 1)$. In the second phase, the processor serially divides 2^k subgraphs into a total of 2^{2k} subsubgraphs, and this takes a total time given by $2^k \cdot M_{lev} \cdot (N/2^k) \cdot (2^k - 1) = M_{lev} \cdot N \cdot (2^k - 1)$. Similarly, the splitting of the 2^{2k} subsubgraphs takes time equal to $2^{2k} \cdot M_{lev} \cdot (N/2^{2k}) \cdot (2^k - 1) = M_{lev} \cdot N \cdot (2^k - 1)$, and so on. Thus the entire execution takes time

$$T'_{lev} = M_{lev} \cdot N \cdot (2^k - 1) \cdot l \tag{3.21}$$

The speedup of this execution over that of the unified algorithm is given by,

$$\begin{aligned} speedup' &= T_{uni}/T'_{lev} \\ &= \frac{1}{l}\left(\frac{M_{uni}}{M_{lev}}\right)\left(\frac{2^{lk}-1}{2^k-1}\right) \end{aligned} \tag{3.22}$$

Using the inequality 3.19 gives

$$speedup' \geq \frac{1}{2l}\left(\frac{2^{lk}-1}{2^k-1}\right) \tag{3.23}$$

In particular, for the EH(3,2),
$speedup' \geq 9/4 \ = 2.25$
and for the EH(2,3)
$speedup' \geq 7/2 \ = 3.5$

3.9 Conclusions

This chapter starts by defining the cost function to be minimized by the static load balancing algorithm. Lack of load balance among the processors is given equal importance as the communication overhead in determining the overall cost. The unified approach maps subtasks directly onto servers. The level-by-level approach assigns the subtasks in different phases down the hierarchy of NCs starting from the root. Three static load balancing algorithms for EH have been described. Each algorithm starts with an initial map. The local search algorithm produces random changes on the map and accepts only those changes that reduce the cost function. The solution may, however, get stuck in a local minima. To avoid the problem, the simulated annealing algorithm provides controlled acceptance of the changes on the map that increase the cost function. Finally, each step of the greedy algorithm modifies a given map function value such that the cost function is minimized by that change. In terms of the cost function, the simulated annealing algorithm performs the best followed by the local search and the greedy algorithms in succession. At the same time, the simulated annealing algorithm is about 20 times slower than the other two algorithms. The level-by-level approach performs slightly better than the unified approach in terms of solution quality. However, the run time of the level-by-level approach is smaller than that of the unified approach.

Chapter 4

THRESHOLD ALGORITHMS FOR DYNAMIC LOAD BALANCING

4.1 Introduction

A dynamic load balancing algorithm tends to achieve uniform load distribution among the PEs by migration of excess load from heavily loaded PEs to lightly loaded ones. The different dynamic load balancing algorithms differ from one another in one or more of the following respects : i) The load information each PE keeps, for example, load value of that PE and load values of adjacent PEs. ii) The information which each PE uses for making a task migration decision, for example, comparison of the load of that PE with the loads of the adjacent PEs. iii) The particular time instants when the algorithms are executed, for example, when a new job comes to a PE, or when a job finishes execution at a PE, or at regular intervals of time. iv) The strategy adopted to decide when migration decision is to be carried out, which task(s) is(are) to be migrated and where the excess load is to be migrated. In the following section, we shall describe a well-known algorithm called the *Threshold Algorithm* (TA).

4.2 Threshold Algorithm

In a threshold algorithm [17,18]

i) Each PE keeps only local load information, i.e. the load value of that PE only. As argued in section 1.1, load of a PE is defined heuristically by the number of tasks that are awaiting execution on that PE (including the task, if any, that is currently being executed).

ii) For making load balancing decisions, a PE uses load information of a subset of the set of all PEs in the network. The size of this subset of PEs is limited by a maximum value. Since remote load information (i.e. load values at other PEs) is not maintained by the PE, it has to *probe* the remote processors across the network to get the remote load values. In order to probe a remote processor, a processor sends a request to the remote processor. The remote processor, upon reception of the request, replies by sending its current load value.

iii) The algorithm is executed during arrival and/or departure of a job.

iv) Load balancing decision is based on the comparison of local load with a certain predefined constant value, called *threshold*. If the local load is more than or equal to the threshold value, the PE is assumed to be *overloaded*. On the other hand, if the local load is less than the threshold, the PE will be called *underloaded*. There are two different types of TAs, namely *sender-initiated* (SI) and *receiver-initiated* (RI) TAs. This classification is based on the *initiator of the load balancing activity*. Let us now describe these two types of algorithms in details.

4.2.1 Sender-Initiated Threshold Algorithm

Whenever an external task arrives at a PE P, the local load value increases by unity. The PE compares this increased load value with the local threshold value T_p. If the load value is more than T_p, the PE is overloaded and it tries to migrate the new task to some remote node. In an attempt to decide the destination node for the task transfer, the PE P probes a maximum of L_p remote nodes chosen randomly in the network. L_p is called the *probe limit of the sender-initiated algorithm*. The probed

nodes reply by sending their respective current load values. If each of these load values is larger than or equal to the threshold value of the corresponding probed PE, it is not possible to transfer the new task to an underloaded processor, and hence the new task is executed locally (i.e. not migrated). On the other hand, if some probed node Q has local load less than T_q, the new task is migrated from PE P to PE Q.

The probes which P carries out can be made serially or parallelly. In *serial probing*, the node P probes the remote processors one by one until a probed node is found whose load is less than the corresponding threshold or until L_p probes have been carried out without finding an underloaded remote PE for the task transfer. In *parallel probing*, on the other hand, the processor P sends out L_p probes in parallel and waits for the replies. If all probed PEs are overloaded, no migration occurs. If one or more probed PEs are found to be underloaded, the new task is sent to any one of the underloaded probed PEs. In our implementation we shall assume serial probing only. The limit L_p on the number of probes is intended to limit the overhead associated with each migration decision not to exceed an acceptable value. We note that the overhead is more in case of serial probing than in case of parallel probing. L_p is chosen to meet the overhead constraint. For example, suppose, each probe, on an average, takes 1% of the average time for the execution of a job. If we want the migration decision overhead not to exceed 10% of the average execution time of a job, we must choose $L_p \leq 10$. On the other hand, choice of T_p can be based on some previous experience with the network or on the use of a suitable heuristic strategy. For example, the average processor load can be calculated by observing the processor over a period of time and the threshold is set to the average load value. The significance of this choice of T_p is that if the PE P gets loaded by a value more than the average, the processor will be called overloaded and will attempt to migrate a new task that arrives to the PE.

4.2.2 Receiver-Initiated Threshold Algorithm

In this case an underloaded PE attempts to receive a task from an overloaded PE in the network. Whenever a task completes execution at a node P, the load value

of P decreases by unity. If it happens that the load value is less than the local threshold T_p, the node P probes a maximum of L_p remote nodes chosen randomly in the network. L_p is called the *probe limit of the receiver initiated algorithm.* The probed nodes reply by sending their respective current load values. If all these probed nodes are underloaded (that is, have loads less than the respective thresholds), then no task migration takes place. On the other hand, if an overloaded PE Q is found by P, a task is migrated from Q to P. The PE Q can choose any task waiting at Q for the transfer, but it cannot choose the task currently serverd by Q for the migration, since in our algorithm a task assigned to a server is never preempted and is allowed to continue till the completion of its execution.

As in the sender initiated algorithm, we shall consider only serial probing. The choices of T_p and L_p are also similar in the two algorithms.

Before we conclude this general discussion about TAs, we mention one point. We have assumed that each PE has its own threshold and probe limit. In what follows, we shall assume that our system of PEs is *homogeneous*, that is, all PEs are identical and so are the interconnection links. Moreover, the interconnection network is assumed to be *symmetric* with respect to each PE. For a binary cube, a mesh with wrap-around connections and for an EH, the network enjoys this symmetry property. For a mesh without wrap-around connections, the assumption of network symmetry is a reasonable one. For such a system, we are justified in taking

$T_p = T$ and $L_p = L$

for each PE P of the network. With this assumption, a probing node can compare remote load values with the local (and system-wide unique) threshold value.

4.3 Dynamic Load Balancing on EH

We recapitulate that external tasks arrive at the NCs and are distributed by the network of NCs among the servers (PEs at level 0) of the network for execution. Each server executes the jobs assigned to it in a FIFO discipline and never takes part in any load balancing decision. Only the NCs execute a distributed algorithm to

ensure the uniformity of load distribution among the servers. Thus, we have separated the job servicing and load balancing activities to work on different sites of the EH network.

4.3.1 Information maintained by each NC

Work load at each server of an EH(k,l) refers to the number of tasks currently present at that node. Work load of an NC at level 1 is the sum of work loads of the 2^k servers directly beneath it. In general, work load of an ith level NC is the sum of the work loads of 2^k NCs/servers at level (i–1) which are directly beneath the ith level NC. In other words, the work load of an ith level NC is the sum of work loads of all 2^{ik} servers in the subEH of level i rooted at the ith level NC. For example, work load of the root NC is the total work load of all servers in the EH(k,l).

Since each NC at level i is connected by control links to 2^k NCs/servers at level (i–1), it is convenient for each NC at level i to keep track of the information regarding the work loads of its 2^k children. In addition, each ith level NC maintains its own work load value which is the sum of the load values of its 2^k children. We notice that only the NCs at level 1 keep track of individual server load values.

4.3.2 Thresholds at NCs

Let t be the threshold value of the work load of a server. By the assumption of homogeneous system with symmetric interconnection network, this value is the same for all the servers of the EH. However, this value does not correctly characterize the threshold of an NC. This is because the work load of an NC has been defined to be the sum of loads of all the servers beneath the NC. Since there are 2^{ik} servers beneath an ith level NC, the equivalent threshold value for the NC is given by

$$T^{(i)} = 2^{ik} \cdot t \tag{4.1}$$

4.3.3 Relation between arrival and deperture rates in equilibrium

In an *equilibrium condition*, the rate of arrival of tasks to each server equals the rate of service of tasks in that server. We shall assume that both the arrival and service processes are *Poisson processes*. Let

λ = rate of arrival of tasks to each NC, and

μ = rate of service of tasks in each server.

We have assumed that λ is same for all NCs irrespective of the level.

To see how external tasks are distributed by the NCs, let us consider Fig 4.1, where only the control links of an EH(k,l) are shown. At this point we assume that no load balancing is performed so that tasks move only down the hierarchy of NCs from higher to lower levels. The root receives external tasks at a rate of λ. A task can be passed to each NC at level (l–1) with equal probability $(1/2^k)$. Thus the input Poisson stream at rate λ is split into 2^k substreams each of rate $\lambda/2^k$ along the 2^k control links from the root NC. Each NC at level (l–1) receives two independent Poisson streams, one of rate $\lambda/2^k$ passed down from its parent and the other is an external arrival of rate λ. Hence the total arrival at the (l–1)st level NC is a Poisson one with rate $\lambda + \lambda/2^k$. This is split by the NC into 2^k Poisson streams each of rate $(\lambda + \lambda/2^k)/2^k = \lambda/2^k + \lambda/2^{2k}$ and these 2^k streams are passed to the 2^k NCs at level (l–2) directly beneath the NC at level (l–1).

This process is repeated down the hierarchy of NCs. Each NC at level 1 receives a Poisson stream of rate $\lambda/2^k + \lambda/2^{2k} + \cdots + \lambda/2^{(l-1)k}$ from its parent and an external arrival of rate λ. The sum is a Poisson arrival of rate $\lambda + \lambda/2^k + \lambda/2^{2k} + \cdots + \lambda/2^{(l-1)k}$ which is split equally among the 2^k servers beneath the NC. Thus at equilibrium

$$\begin{aligned}
\mu &= (\lambda + \lambda/2^k + \lambda/2^{2k} + \cdots + \lambda/2^{(l-1)k})/2^k \\
&= \frac{\lambda}{2^k}(1 + 1/2^k + (1/2^k)^2 + \cdots + (1/2^k)^{l-1}) \\
&= \frac{\lambda}{2^k}\left(\frac{1 - (1/2^k)^l}{1 - 1/2^k}\right)
\end{aligned}$$

Figure 4.1: Distribution of tasks by the NCs among the servers

$$= \lambda\left(\frac{1-2^{-lk}}{2^k-1}\right)$$

Rearranging gives

$$\begin{aligned}\lambda &= \mu\left(\frac{2^k-1}{1-2^{-lk}}\right) \\ &\approx \mu(2^k-1) \end{aligned} \tag{4.2}$$

For an EH(3,2), k=3 and l=2. So we have,

$\lambda = \mu(2^3-1)/(1-2^{-6}) = 7.11\mu$.

For an EH(2,3), k=2 and l=3. Therefore,

$\lambda = \mu(2^2-1)/(1-2^{-6}) = 3.05\mu$.

4.3.4 Load balancing in an EH(k,1)

EH(k,1) consists of an NC with 2^k servers directly beneath the NC. We have mentioned that the NC maintains the current load value of each server beneath it. Whenever a new task comes to the NC, it finds out the server which is the least loaded among all the servers. The new task is then allocated to this least loaded server. As a result, the k-cube of servers is perfectly load balanced and there is no need of task migration among the servers. This is the way an EH(k,1) performs load balancing. Obviously, this is a centralized algorithm. In any EH(k,l) also, whenever an NC at level 1 wants to assign a task to the k-cube of servers beneath it, it maps the task onto the least loaded server in the k-cube using the centralized algorithm of an EH(k,1).

4.3.5 Load balancing in an EH(k,l)

Now that we have a centralized algorithm for load balancing in each k-cube of servers, there is a need of load balancing among different k-cubes of servers in the network. This can be achieved by using a distributed algorithm working on the hierarchy of NCs at levels 1 through l. We have chosen the threshold algorithm to solve this purpose. The TA, in its original form, can be applied on the network in which case

each NC at level 1, upon arrival and/or departure of a job, compares its total load with the threshold value for a first level NC (i.e. with $T^{(1)}$) and probes other NCs at level 1, if the threshold value is exceeded by the total load of the NC. The probes, probe replies and the tasks to be migrated are routed through the network of NCs.

This simple TA, however, does not have a multilevel structure to exploit the hierarchical connection among the NCs. In the next chapter, we describe a modification of the algorithm, called the *multi-level threshold algorithm*, that is very suitable for an EH.

4.4 Conclusions

In this chapter we describe a popular algorithm for dynamic load balancing on a distributed multiprocessor system. The algorithm is called the Threshold Algorithm (TA). The algorithm is executed at each PE when a new task arrives at the PE or when a job finishes execution at the PE. If the load of the PE exceeds certain threshold value, the PE probes some other PEs in the network in an attempt to migrating a task from an overloaded to an underloaded PE. This conventional TA does not have a hierarchical structure to exploit the multi-level connection of NCs in an EH. A modification of this conventional algorithm suitable for the EH is discussed in the next chapter.

Chapter 5

MULTI-LEVEL THRESHOLD ALGORITHM

5.1 Introduction

In this chapter we introduce a modification of the simple threshold algorithm (TA) described in the last chapter. We call this modified algorithm the *multi-level threshold algorithm* (MLTA). We will mainly deal with the sender-initiated algorithm. The working of the receiver-initiated algorithm is very similar to the working of the former algorithm except for certain modifications or differences that we will describe at appropriate places.

5.2 Motivation Behind MLTA

The threshold T and the probe limit L of the conventional sender-initiated TA described in section 4.2.1 assume fixed predetermined values. We shall shortly see that we can expect better load sharing among the processors if we let the values of T and L adapt to the dynamic load situation.

Let us first consider the effect of changing probe limit with load. From a local point of view, task transfer from a PE is more urgent at larger processor loads. This means that the more a PE gets loaded, the more it is necessary for the PE to get rid

of some excess load. As a result, the PE should perform larger number of probes at higher loads. Thus L should increase with the load. From a global point of view, on the other hand, if the overall system load is high, a large fraction of PEs is overloaded, i.e. has load larger than T. Hence the probability of a successful probe, that is, the probability of probing an underloaded PE, becomes low. Hence, the chance of probing an underloaded PE increases if we increase the number of probes, i.e. the probe limit L.

Similarly, the threshold value T should be adjusted in accordance with the load situation. A threshold sets up a level of comparison so that two PEs, one with load above the level and the other with load below the level, may exchange loads between each other so as to make their final load values closer to each other as well as closer to the threshold. However, if both the PEs have loads either greater than T or less than T, the conventional TA cannot provide a means to reduce the load imbalance within the pair, however large the load difference may be. For example, consider a system of 5 PEs, P_1 through P_5. At some instant each of the PEs P_1 through P_4 has load $T + 5$ and P_5 has load $T + 6$. Suppose 4 tasks arrive successively at P_5 before any task completes execution in any one of the PEs. Since all the PEs have loads larger than T, no task migration occurs and the load of P_5 increases to $T + 10$. On the other hand, if we adjust the threshold value to $T_{adj} = T + 6$, P_5 can migrate the 4 new tasks to the other PEs resulting in a situation when each PE has a load of $T + 6$. Therefore, we expect better load sharing, if we increase T with increase in load value.

Following this idea helps us to explore a threshold algorithm with c thresholds $T_1, T_2, \ldots, T_c$ and corresponding probe limits $L_1, L_2, \ldots, L_c$, respectively, with

$$\begin{aligned} T_1 < T_2 < \cdots < T_c \\ L_1 \leq L_2 \leq \cdots \leq L_c \end{aligned} \tag{5.1}$$

This implies that for larger processor loads, larger thresholds are exceeded and correspondingly, larger number of probes are carried out. Specifically, if the work load W_p of a processor P, upon arrival of an external task, satisfies the inequality

$$T_1 < T_2 < \cdots < T_j < W_p \leq T_{j+1} < \cdots < T_c \tag{5.2}$$

that is, if T_j is the largest threshold exceeded by W_p, then a maximum of L_j probes are carried out by the node P and the comparison level is set at T_j. Thus, if any probed node with load less than T_j is found, the task migration takes place.

For a receiver-initiated algorithm, we similarly maintain c thresholds $T_1, T_2, \ldots, T_c$ and the corresponding probe limits $L_1, L_2, \ldots, L_c$, respectively, such that

$$\begin{aligned} T_1 > T_2 > \cdots > T_c \\ L_1 \leq L_2 \leq \cdots \leq L_c \end{aligned} \tag{5.3}$$

That is, the smaller the processor load, the larger is the number of probes carried out by the processor so that the possibility of finding an overloaded PE from which to take a task is higher.

Ideally, we should maintain a threshold corresponding to each positive integer value of the work load. In that case, the sender-initiated MLTA will be executed upon the arrival of each external task to a non-empty PE (i.e. a PE with load > 0). This is because such an arrival will cause at least one threshold to be exceeded by the processor load, since the increased processor load will be larger than 1, the smallest threshold. Therefore the PE will attempt to transfer the new task to a PE whose load is less than the load of the former PE. However, this may lead to the maintenance of a large number of values, viz L_i for each integer i. In practice, therefore, the number of thresholds and probe limits (i.e. c) and their values may be chosen heuristically. For an EH(k,l), as an example, we take $c = l - 1$ using a heuristic that exploits the hierarchical structure of the network (see section 5.4).

5.3 Applying MLTA on Mesh and Binary Hypercube

As described in section 4.2.1, we cannot increase the probe limit indefinitely with increased system load for our modified algorithm. This is because larger number of probes incurs larger time for taking the migration decision of each new task. This increase in overhead associated with probing is more serious for serial probing. Also,

for the sender-initiated case, higher overhead occurs at higher loads which is undesirable. Thus, increasing L at higher loads may lead to better load sharing among the PEs but need not necessarily guarantee better speedup which is our ultimate goal. As a result, it remains uncertain if applying MLTA onto a mesh or a binary hypercube network can really improve system performance in terms of speedup. We shall shortly see that an EH with the NCs available for carrying out the load balancing activities is very suitable for implementing the MLTA.

5.4 Applying MLTA to Extended Hypercube

Before describing how the MLTA works on an EH, we mention that each NC at level 1 of any EH(k,l) uses the centralized algorithm described in 4.3.4 in order to assign a task to the k-cube of servers directly beneath the NC. In a similar manner, whenever a new task arrives at the root of an EH(k,l) for $l > 1$, the root allocates the task to the least loaded NC at level (l–1). It remains to describe how a task is migrated from one NC where the task comes, to a different NC in order to perform load balancing in accordance with the multi-level threshold algorithm.

5.4.1 Thresholds maintained at different processors

In terms of load per server in an EH(k,l) , we maintain (l–1) thresholds $t_1, t_2, \ldots, t_{l-1}$ with

$$t_1 < t_2 < \cdots < t_{l-1} \tag{5.4}$$

for the sender-initiated algorithm. As suggested by Eqn 4.1, we can equivalently think of (l–1) thresholds $T_1^{(i)}, T_2^{(i)}, \ldots, T_{l-1}^{(i)}$ in terms of load per processor (NC or server) at level i, where

$$T_j^{(i)} = 2^{ik} \cdot t_j \tag{5.5}$$

for i = 0, 1, 2, ..., l, and
j = 1, 2, ..., (l-1).

It follows that for a server (i=0)

$$T_j^{(0)} = t_j \tag{5.6}$$

With these definitions the following theorems follow trivially.

Theorem 5.1 : If the load of an NC at level i in an EH(k,l) exceeds the threshold $T_j^{(i)}$, there must be at least one child processor (NC or server) at level (i–1), whose load exceeds $T_j^{(i-1)}$.

Proof : If it were otherwise, the total load of the NC at level i would be less than $2^k \cdot T_j^{(i-1)} = 2^k \cdot 2^{(i-1)k} \cdot t_j = 2^{ik} \cdot t_j = T_j^{(i)}$, since there are 2^k children of the ith level NC.

Theorem 5.2 : If the load of an NC at level i in an EH(k,l) exceeds the threshold $T_j^{(i)}$, there must be at least one server beneath the NC, whose load exceeds t_j.

Proof : This follows from the repeated application of the above theorem. That is, one child of the NC at level i must be a processor at level (i–1) whose load exceeds $T_j^{(i-1)}$. If (i=1), then this latter processor is a server whose load exceeds t_j. If (i>1), then this latter processor has a child processor at level (i–2) whose load exceeds $T_j^{(i-2)}$ and so on.

In a similar manner we can prove the following two theorems.

Theorem 5.3 : If the load of an NC at level i in an EH(k,l) is less than the threshold $T_j^{(i)}$, there must be at least one child processor (NC or server) at level (i–1), whose load is less than $T_j^{(i-1)}$.

Theorem 5.4 : If the load of an NC at level i in an EH(k,l) is less than the threshold $T_j^{(i)}$, there must be at least one server beneath the NC, whose load is less than t_j.

5.4.2 The least-loaded-child path

Suppose that we want to allocate a task to some server of a subEH of level i. Initially the task is at the root of the subEH. Let us also assume that the load of the root of the subEH is less than $T_j^{(i)}$. Now theorem 5.4 suggests that there is at least one server in the subEH whose load is less than t_j. The problem is how to find out such a server to which to allocate the task. Theorem 5.3 suggests that the least loaded child of the root of the subEH must be a processor at level (i–1) whose load is less than $T_j^{(i-1)}$. So the root of the subEH sends the task to its least loaded child at level (i–1). The task is now at the root of a subEH of level (i–1) whose load is less than $T_j^{(i-1)}$. The root of this latter subEH now sends the task to its least loaded child at level (i–2) and the task finds itself at the root of a subEH of level (i–2) whose load is less than $T_j^{(i-2)}$. This process is repeated till the task reaches a server whose load is less than $T_j^{(0)} = t_j$. Henceforth, we will call the path followed by the task the *least-loaded-child path* or LLC path, since at each step the task is assigned to the least loaded child of an NC.

5.4.3 The probing heuristics

Suppose a new task arrives at an NC called NC_1. The allocation of the task takes place in two phases. First, a migration decision is carried out if at least the smallest threshold at NC_1 is exceeded by the increased load of the NC. Let us assume that the task is migrated to an NC called NC_2 as a result of this migration decision. (If, however, no migration occurs, we have $NC_1 = NC_2$). After the task reaches NC_2, it is assigned to a server following the LLC path starting from NC_2.

As described in section 5.2, we are motivated by the idea of increasing the number of probes at higher loads. Thus we will tend to probe larger subEHs (i.e. subEHs of larger levels) in case larger thresholds are exceeded. To be more specific, let us formulate that when the largest threshold exceeded by the work load of an NC upon the arrival of a new job is the jth one, we will not make an attempt to probe a subEH whose level is greater than (j+1) before allocating the task to a server along

the LLC path. In terms of the notations introduced in the previous paragraph this implies that whenever NC_2 is different from NC_1, NC_2 cannot be at a level larger than (j+1). The implication of this heuristic will be clear in the next section. At this point, we mention that since the largest possible subEH of an EH(k,l) is the entire EH itself, the largest subEH probed has a level equal to l. Thus the maximum possible value of (j+1) is l and that of j is, therefore, (l–1). This explains the existence of (l–1) thresholds for the algorithm.

5.4.4 The migration decision

Let us suppose that a new task arrives at an NC called NC_1 at level i. As a result, the load of the NC increases by unity. If the increased load at NC_1 does not exceed any threshold, that is, the load is less than $T_1^{(i)}$, then no migrtion decision is carried out and the task is assigned to a server along the LLC path starting from NC_1. So let us assume that some thresholds are exceeded. Suppose that the largest threshold exceeded is the jth one, that is, $T_j^{(i)} < \ load\ of\ NC_1\ \leq T_{j+1}^{(i)}$. If i > j, we are already at the root of a subEH whose level $\geq$ (j+1). So we make no attempt to migrate tha task, since that violates our objective of not probing a subEH of level greater than (j+1). The new task is assigned to a server following the LLC path from NC_1.

Finally, suppose $i \leq j$. In this case the task allocation proceeds in two phases. The first phase, i.e. the probe phase, is an upward phase. NC_1 sends a request to its parent NC at level (i+1). If this parent NC has a load less than $T_j^{(i+1)}$, then following the LLC path from this NC will guarantee that the new task can be assigned to a server whose load is less than t_j. On the other hand, if the jth threshold is exceeded by the load of the (i+1)st level NC, the request is passed to the next higher level, that is, to the grandparent of NC_1. This NC at level (i+2) takes identical decisions by comparison of the load at level (i+2) with the threshold $T_j^{(i+2)}$. However, as per our formulation, the upward passage of the request should not proceed beyond level (j+1). If the request reaches (j+1) without finding an NC for the task transfer, failure of the probe is reported to the initiator of the probe, that is, to NC_1, and the task is allocated to a server following the LLC path from NC_1. On the other hand, if an

NC, say NC_2, is found during the probe, such that the load of NC_2 is less than the jth threshold at NC_2, then the new task is migrated from NC_1 to NC_2 and from NC_2 to a server along the LLC path. In either case of success or failure of the probe, the task allocation along the LLC path constitutes the second and downward phase of the task-transfer algorithm.

5.4.5 Complexity of MLTA

The downward phase for both $i \leq j$ and $i > j$ has to find out at most l least loaded child elements one at each level of the EH(k,l). If a linear search is performed to find out the least loaded among the 2^k children, each search makes O(2^k) comparisons. So a total of O(l·2^k) comparisons are carried out during the downward phase. On the other hand, if binary search is performed to find the least-loaded element at each level, a total of O(l·k) comparisons are necessary, which is O(lg N), where

N = the number of servers in the EH(k,l) = 2^{lk}.

In case of binary search, however, each NC must maintain a sorted list of the loads of its children. Whenever the load of a child changes (during assignment of a job to the child or when a server beneath the child completes execution of a task), the list of loads of the children must be updated and adjusted to keep the list sorted after the updation. This can be done in O(k) time, since the list is of length 2^k.

The upward phase, on the other hand, passes the probe at most j levels up the hierarchy of NCs. Each NC in the path makes a single comparison of its current load value with its jth threshold. The entire upward phase, thus, makes O(j) number of comparisons which is O(l).

So each migration decision in the MLTA makes O(l·2^k) comparisons in case of linear search and O(l·k) comparisons in case of binary search.

We also note that the upward phase consists of at most 2j communications between NCs to pass the probe upwards and to get the reply of the probe downwards. The downward phase needs at most l communications to allocate the task. So the MLTA requires O(l) communications on the whole.

5.4.6 Modifications for the receiver-initiated MLTA

So far we have described the working of the sender-initiated MLTA. The working of the receiver-initiated MLTA is similar except for three aspects. First, the initiator of each migration request is some NC at level 1. No NC at level greater than 1 need initiate a task reception request. This follows from theorem 5.3. Suppose, the completion of a job at a server makes the load of an NC at level i>1 to drop below a threshold $T_j^{(i)}$. By theorem 5.3, we argue that there is an NC at level 1 beneath the ith level NC in concern, whose load falls below $T_j^{(1)}$. As a result, the NC at level 1 issues a task reception request. Any further request on the part of the ith level NC will merely be a duplication of effort.

Secondly, whenever a new task comes to an NC, no migration decision is carried out. The new task is sent to a server following the LLC path starting from the NC. Task migration decisions are carried out only when a job departs from the network and as a result the decreased load falls below a threshold. The upward passage of the probe and the downward passage of the reply are exactly similar to those corresponding to the sender-initiated case.

Thirdly, a task assigned to a server may be migrated from the server. As preemption of a task is not allowed, only those tasks waiting in a server can be migrated, whereas that being serviced by the server cannot be migrated. Suppose an underloaded NC at level 1 called NC_1 requests a task from another first level NC, say NC_2. If a new job is present at NC_2 and the job is yet to be allocated by the NC, then that task is sent to NC_1. Otherwise, NC_2 requests its heaviest loaded child server to send back a task for transfer to NC_1.

5.5 Simulation Results

The MLTA algorithm for EH(2,3) and EH(3,2) has been implemented on a transputer-based system. The performance of the MLTA algorithm on EH has been compared with that on an 8*8 mesh and on a 6-dimensional binary cube. We note that the

number of servers in each of the above three networks is the same, viz. 64. As argued in section 5.3, we cannot increase the probe limit for the mesh and the binary hypercube networks as much as we want, since that increases the overhead of each migration decision. On the other hand, in an EH(k,l) the probe requires a maximum of only j comparisons before the allocation phase, when the jth threshold is the largest threshold exceeded.

Figs 5.1 and 5.2 show the total execution time (service time + communication time + waiting time) (TET) of a set of thousand tasks for the different network topologies plotted as a function of R/C, where R and C are the average service time and average time of communication respectively of a task. R/C is varied by keeping R constant and by allowing C to vary. The interarrival and service times of the jobs are generated randomly following exponential distribution. The EH has been observed to perform better than the other two topologies. For high values of R/C (R/C = 50), EH shows similar performance as mesh and hypercube. For low values of R/C (R/C $<$ 30), the superiority of EH over the other two architectures is pronounced.

Fig 5.3 shows the variation of TET of the same set of tasks with varying threshold values for the SI MLTA algorithm on the two EH networks.

5.6 Conclusions

The conventional TA assumes fixed predetermined values of the threshold T and the probe limit L. We can ensure better load sharing if we allow both L and T to increase with the load value. This leads to an algorithm that maintains more than one thresholds and corresponding probe limits so that for larger load values larger thresholds are exceeded and as a result, larger number of probes are carried out. But in a mesh or a binary cube, we cannot increase the number of probes indefinitely, because that increases the overload for each migration decision. For an EH, however, the algorithm can be nicely applied in a manner that exploits the hierarchical structure of the network.

Chapter 6

CONCLUSIONS

In this project, load balancing algorithms for extended hypercube (EH) networks have been studied. Static load balancing on an EH has been formulated as an optimization problem – a problem of allocating the computation modules or the subtasks of a task to the servers of the network in a way so as to reduce the imbalance among the loads of the servers as well as to keep the communication overhead associated with the execution of the task as small as possible. Three algorithms for solving the optimization problem have been implemented and compared among each other in terms of the solution quality obtained by the algorithms.

Threshold algorithms are very popular algorithms for dynamic load balancing in a distributed multiprocessor system. An extension of the conventional threshold algorithm has been proposed and implemented. The performances of the modified algorithm for mesh, binary cube and EH networks have been compared among each other. EH has been observed to give better speedup than mesh and binary cube topologies.

Most of the load balancing algorithms in vogue do not take into account the interconnection pattern of the processors in the network. Although current literature depicts some special algorithms for load balancing on mesh and binary cube architectures, such architecture–specific algorithms are very few in number. In this project, we have stressed on the multilevel structure of the interconnection network

in an EH. In an attempt to exploit the hierarchical connection of processors in an EH, we have extended and modified some popular algorithms. In case of static load balancing, these modifications include the design of a tree-structured level-by-level distribution of the task allocation problem among the controllers of an EH. Similarly, the modified threshold algorithm for dynamic load balancing on an EH is designed to suit the toplogy of an EH.

In this project, we have also been able to combine the centralized and distributed approaches for load balancing. The set of all computation processors (called servers) has been partitioned into subsets such that all servers in a subset are centrally controlled by one controller. The different controllers, however, work in a distributed fashion to secure load balancing over the entire network.

Possible extensions of the work : In the project, we have made certain simplifying assumtions, for example, the independence of the tasks of one another. A more general approach is not to ignore the communication among the tasks. Thus while migrating a task from one processor to another, one should consider the effect of the migration on the communication overhead of the task with other tasks present in the network.

Secondly, we have studied the static and dynamic load balancing algorithms independently. In practice, however, these two problems may not be separated from one another. Designing an algorithm that combines both static and dynamic load balancing strategies is obviously a challenging extension of the project work.