The Fourteenth International Conference on Fault-Tolerant Computing, June 1984, Kissimmee, Florida

Assigning Processes to Processors: a Fault-Tolerant Approach

Gautam Kar
Christos N. Nikolaou
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

John Reif
Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138

Abstract

Distributed systems become increasingly attractive as a means to achieve higher throughput for a given level of hardware computational power, and higher availability of the overall system. Applications running on distributed systems are usually organized as collections of communicating sequential processes. This paper examines the problem of allocating processes to processors under workload balancing and optimality constraints, and the problem of relocating groups of processes when one or more processors fail.

1. Introduction

Distributed systems become increasingly attractive as a means to achieve higher throughput for a given level of hardware computational power, and higher availability of the overall system. Applications running on distributed systems are usually organized as collections of communicating sequential processes. This paper examines the problem of allocating processes to processors under workload balancing and optimality constraints, and the problem of relocating groups of processes when one or more processors fail. Heuristic techniques are proposed that are suitable for distributed environments.

In section 2 we formulate the problem and define terms and symbols used in later sections. We also show how to use clustering algorithms from the literature (5) to organize processes and processors into hierarchical clusters, by trying to minimize the intercluster (process or processor) communication costs. In section 3 we present an allocation algorithm that maps a hierarchy of process clusters to a hierarchy of processor clusters. The problem of process relocation is addressed in section 4. The concept of a *detection unit* is introduced and it is shown how process relocation can be achieved within such a unit. Finally in section 5, we summarize the results and point out directions of further research.

2. Formulation of the problem

In the context of this paper, we conceive of a distributed system as composed of a set of *nodes* representing the active processing agents (e.g. host computer complexes in a computer network, individual processors in a multiprocessor system or a local area network, etc.) and an *interconnection* structure providing full connectivity between the nodes (e.g. communication lines in a long-haul computer network, shared memory modules or a bus in a multiprocessor system, a ring or a star topology in a local area network).

Given a set of processes $\mathscr{P}_p = \{p_1, p_2, \ldots, p_n\}$, a set of processors $\mathscr{P}_P = \{P_1, P_2, \ldots, P_N\}$ and their logical and physical interconnection structure respectively, we define the *process graph* $\mathscr{G}_p = (\mathscr{P}_p, \mathscr{L}_p)$, where $\mathscr{L}_p$ is a set of links defined as: $\mathscr{L}_p = \{l_{ij} / l_{ij}$ denotes logical communication between processes p_i and $p_j\}$. Similarly define the *processor graph* $\mathscr{G}_P = (\mathscr{P}_P, \mathscr{L}_P)$, where $\mathscr{L}_P = \{l_{ij} / l_{ij}$ denotes a physical communication link between processors P_i and $P_j\}$. Consider two processes p_i and p_j possibly residing on two different nodes. The costs incurred by their presence are the following:

- They contribute to the workload of the nodes (processors) that cannot exceed a maximum capacity M_i and that has to satisfy the *load balancing constraint* explained below. If the time complexity of the computations of the individual processes is roughly equal, it is realistic to assume that the workload of a processor is increased by one "workload unit", as a result of a process allocation. We denote by c_i the currently assigned workload of processor P_i. The load balancing constraint can now be expressed as follows:

$$\left| \frac{c_i}{M_i} - \lambda \right| \leq \epsilon$$

  for $i = 1, \ldots N$ and where λ represents the "ideal" ratio of the assigned workload over the capacity, and ϵ represents the tolerance from the ideal ratio. Assuming that in a typical system all M_i's are larger than a workload unit, that λ is greater than .5 and that ϵ is less than .05 it follows that the number of processes exceeds - usually by far - the number of processors.

- There is a communication cost (delay) associated with a message transmission from process p_i to p_j, that grows with:
  - the number of "connections" (i.e logical communication channels) established between processes p_i and p_j. Although this number may vary with time, we assume that it changes rather infrequently and that there are long periods of stability during which its value remains constant.
  - the average number of messages per unit of time sent from process p_i to p_j. We assume for simplicity that the same number is observed for messages flowing from p_j to p_i.
  - the average delay of routing a message through the processors' interconnection network.

We assume that the links of the process and processor graphs are labelled with weights. Link *(i, j)* in the process graph carries a weight that is proportional to the communication cost of processes p_i and p_j whenever these two processes are assigned to *different* processors and is zero otherwise. Similarly, link *(i, j)* of the processor graph carries a weight that is proportional to the delay of the communication link connecting processors P_i and P_j. The total communication cost to the distributed system with a given assignment of processes to processors is the sum of the communication costs of all pairs of processes (p_i, p_j), such that p_i does not reside on the same processor as p_j. Clearly, the total communication cost can be used as a yardstick to compare the desirability of two different assignments of processes to processors; minimization of this cost we call the *optimum allocation* problem. It has been shown that this problem can be formulated as a 0-1 integer linear programming problem, known to be NP-complete (4).

There is, therefore a need to develop heuristic algorithms for finding suboptimal feasible solutions for the allocation problem. We propose the following combination of heuristic techniques: making use of known clustering algorithms, organize the graphs $\mathcal{G}_p$ and $\mathcal{G}_P$ in hierarchies of clusters using the weights on their links as the clustering (similarity) measure. Call $\mathcal{T}_p$ and $\mathcal{T}_P$ the resulting process and processor cluster trees respectively. In the following section we show how to map the nodes of $\mathcal{T}_p$ to the nodes of $\mathcal{T}_P$, thereby effecting the assignment of processes to processors. Hagouel (5) has written a good survey of the hierarchical clustering techniques.

3. The allocation algorithm

We next present heuristics that achieve suboptimal allocation of processes to processors while maintaining the workload balancing constraint. Stone (7) and Rao et al. (8) have also dealt with the problem. Their model includes storage constraints and computing costs, that may vary from processor to processor. Our model, on the other hand, assumes the computing cost to be uniform and introduces the additional constraint of load balancing. The basic idea of our heuristic is to map the process cluster tree $\mathcal{T}_p$ to the processor cluster tree $\mathcal{T}_P$. The allocation algorithm assigns nodes of the process cluster tree at a given level to nodes of the processor cluster tree at the same level. Ideally, such a mapping would allocate clusters of heavily communicating processes to clusters of densely connected processors. The allocation, however, is complicated because neither of the cluster trees is necessarily balanced and because the number of nodes at some given level may in general be different in the two cluster trees. This difference will necessitate merging and splitting of nodes at the same level of the two cluster trees in order to achieve a one-to-one mapping and load balancing. The question of how suboptimal an allocation is, after readjustment of the two cluster trees, is an open problem to be addressed in the future.

We now introduce terms and symbols that are used in the presentation of the allocation algorithm. Let r be a non-leaf node of the process cluster tree assigned by the allocation algorithm to R, a non-leaf node of the processor cluster tree. The children of r represent clusters of processes; let w_i be a variable denoting the number of processes represented by child i of node r, and let $\mathcal{S}_r = \{w_1, \ldots, w_{n(r)}\}$, where $n(r)$ is the number of children of node r. Similarly, call $\mathcal{S}_R = \{M_1, \ldots, M_{n(R)}\}$ the set of variables denoting capacities of the children of node R; M_i represents the sum of the workload capacities of all processors belonging to the cluster represented by the i-th child of R.

For every pair of nodes r and R, where r has already been assigned to R, the allocation algorithm uses a heuristic method to allocate the children of r to the children of R, possibly by splitting, and/or merging the children of r. Notice that the workload balancing constraint is satisfied for the pair of nodes r and R; therefore, there should be at least one way of partitioning the clusters of processes represented by the children of r among the clusters of processors represented by the children of R without violating the workload constraints. This conclusion is additionally based on the unit workload assumption and the assumption that all weights are integer numbers.

Let $V_i = | \lambda - \frac{c_i}{M_i} | - \varepsilon$ be the *violation* of the i-th cluster of processors (child of R). The value of c_i is zero when the allocation algorithm starts, and grows to a value allowed by the workload constraints. The "violation" of cluster i gives the percentage of unfilled capacity of that cluster and therefore the amount of violation of the workload balancing constraint. The allocation algorithm uses the violation values of the children of R, to select a candidate cluster of processors to be assigned processes. Clusters with higher violations are chosen first. For convenient bookkeeping, call $\mathcal{S}_v = \{ V_i \,/\, V_i > 0 \}$ the set of non-zero violation values, and assume that it is ordered by decreasing violation values. Figure 1 shows how the various fractions of the maximum workload capacity of a cluster of processors are defined. Last we define the *auxiliary set* $\mathcal{S}_A$. This set consists of all variables V_i belonging to processor clusters with current assignment c_i, having values in the range:

$$\lceil (\lambda - \varepsilon) M_i \rceil \leq c_i \leq \lfloor (\lambda + \varepsilon) M_i \rfloor$$

$\mathcal{S}_A$ contains V_i's of non-violators which can still be assigned processes without overstepping the maximum workload bound. $\mathcal{S}_A$ is used by the allocation algorithm, to select candidate clusters of processors, whenever there are clusters of processes left unassigned but no clusters of processors violating the workload constraint.

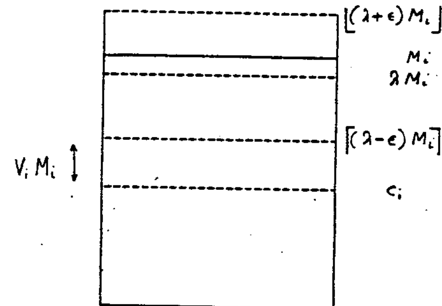

Figure 1: Defining tolerance range and violation

3.1 Description of the ALLOCATE algorithm

The allocation algorithm is presented here as a procedure applied on the process and processor cluster trees $\mathcal{T}_p$ and $\mathcal{T}_P$ (see figure 2). We assume that it is executed once at system generation time

at one of the processors of the distributed system. The algorithm takes two arguments, r and R, nodes of $\mathcal{T}_p$ and $\mathcal{T}_P$ respectively. We assume that at each level, the nodes of $\mathcal{T}_p$ and $\mathcal{T}_P$ are numbered from left to right. We now informally present the allocation algorithm; initially assume that

$$\mathcal{S}_V = \{ V_i / V_i = \lambda - \varepsilon, i = 1, \ldots, n(R) \},$$

reflecting the fact that initially all children of R are empty:

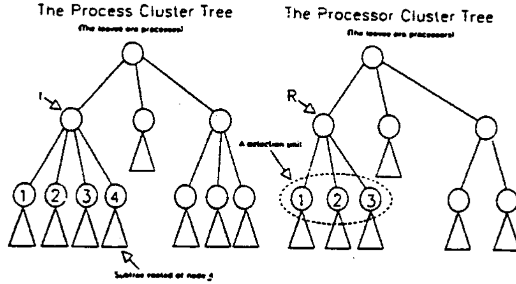

Figure 2: An example of two cluster trees

procedure ALLOCATE(r, R)

1. If r and R are roots of $\mathcal{T}_p$ and $\mathcal{T}_P$ respectively, then check the global workload constraint:

$$| \lambda - \frac{n}{\sum_{i=1}^{i=N} M_i} | \leq \varepsilon$$

where n is the total number of processes and each process represents a unit workload. If the above inequality is violated, there is no feasible solution and therefore terminate the algorithm; otherwise continue.

2. Initialize the sets RA_i, for $i = 1, 2, \ldots, n(R)$, to empty. Set RA_i will contain the indices of the process clusters (children of r) that have been assigned to the i-th child of R.

3. Repeat this step until either $\mathcal{S}_r = \phi$ or $\mathcal{S}_v = \phi$:

   a. Let i be the child of R with maximum value of V_i. If there are more than one, break ties by choosing the lowest indexed one. Call function SELECT($\mathcal{S}_r$) to choose a child j of r with weight w_j to assign to i. Function SELECT is described in section 3.2.

   b. Perform the following updates: $c_i = c_i + w_j$, $V_i = | \lambda - \frac{c_i}{M_i} | - \varepsilon$ If $V_i \leq 0$, remove V_i from $\mathcal{S}_V$. If $\lceil (\lambda - \varepsilon) M_i \rceil \leq c_i \leq \lfloor (\lambda + \varepsilon) M_i \rfloor$ then add V_i to $\mathcal{S}_A$ (intuitively, if V_i is added to $\mathcal{S}_A$ child i of R does not violate the workload constraint any more, but has available capacity to be used if there is a need for it). Insert j in set RA_i.

4. When this step is reached, either one of the following statements is true: a) both $\mathcal{S}_r$ and $\mathcal{S}_A$ are not empty, or b) $\mathcal{S}_r$ is empty. If the first statement is true, repeat the following steps until $\mathcal{S}_r = \phi$:

   a. Let i be the child of R, such that the associated value of $| V_i |$ in $\mathcal{S}_A$ is maximum. If there are more than one, break ties by choosing the lowest indexed one. Call function SELECT($\mathcal{S}_r$) to choose a child j of r with weight w_j to assign to i.

   b. Perform the following updates: $c_i = c_i + w_j$, $V_i = | \lambda - \frac{c_i}{M_i} | - \varepsilon$. If $c_i = \lceil (\lambda - \varepsilon) M_i \rceil$ or $c_i = \lfloor (\lambda + \varepsilon) M_i \rfloor$ then delete V_i from $\mathcal{S}_A$ (intuitively, V_i is removed from $\mathcal{S}_A$ if child i of R is filled to the maximum tolerated capacity). Insert j in set RA_i.

5. At this step $\mathcal{S}_r = \phi$. For each set RA_i do the following: Create a new node i in the process cluster tree $\mathcal{T}_p$. Make i a child of r and make all nodes contained in RA_i, children of i.

6. Recursively call ALLOCATE(i,i) for $i = 1, \ldots, n(R)$ (observe that now $n(r) = n(R)$.) End ALLOCATE.

3.2 The Function SELECT

We conclude the presentation of the allocation algorithm by describing function SELECT. In general terms, this function operates on the set $\mathcal{S}_r$ and on the child i of R, that upon invocation of SELECT has the maximum value of violation. According to some selection criterion, the function chooses a cluster of processes, that may be a child of r or a node inside the subtree of one of r children; this cluster is then assigned to i and the necessary updates are performed on $\mathcal{S}_r$. We list two possible selection criteria for the choice of a cluster of processes to be assigned to i:

- Select the maximum $w_j \in \mathcal{S}_r$ such that $| \lambda - \frac{c_i + w_j}{M_i} | \leq \varepsilon$.

  This criterion would tend to preserve the structure of the process cluster tree intact.

- Calculate the *affinity* of all clusters of processes in $\mathcal{S}_r$ and select the one with the maximum value that satisfies the workload constraint. The *affinity* of a cluster of processes is defined as follows: Let $\mathcal{S} = \{ p_1, \ldots, p_k \}$ be a set of processes, let $\mathcal{S}_l = \{ l_1, \ldots, l_m \}$ be the set of connecting links in the process graph and let $\mathcal{W}_l = \{ \beta_1, \ldots, \beta_m \}$ be the set of weights associated with the links. Then $affinity(\mathcal{S}) = \sum \beta_i$. This criterion would tend to choose densely connected clusters of processes as candidates for assignment.

The two criteria can, of course, be combined in more than one ways; for example, the second criterion may be used to break ties resulting from the first one, or the first may be used to break ties of the second. Observe however, that the selection function may still not be able to come up with a candidate for assignment, since they may all violate the workload constraint. In the following, we show how the selection function resolves this difficulty by splitting the process cluster with the maximum weight (a heuristic decision) and making its children, children of r:

Function SELECT ($\mathcal{S}_r$, i) where i is the child of R with maximum value of violation:

1. Using either one of the selection criteria - or any combination thereof - choose a child j of r; if successfull, return j.
2. Otherwise, let $w_j = \max (\mathcal{S}_r)$. Remove j from the process cluster tree and w_j from $\mathcal{S}_r$, make the children of j, children of r, and add them to $\mathcal{S}_r$.
3. Repeat the above two steps until a candidate process cluster for assignment can be successfully identified.

Termination occurs because the total number of processes exceeds the total number of processors and because each process represents a single unit of workload.

4. Process Relocation

In this section we present a distributed algorithm to cope with the

problem of processor failure by relocating the workload of a failed processor to a working one so that the distributed system as a whole can continue to function.

4.1 Isolation of Failure

The relocation algorithm presented in the next section relies on the availability of a method to detect processor failure in a distributed system. Considerable work has been reported in the literature on this topic. We refer the reader to some of them (1,2,6,9). In what follows it is assumed that we have at our disposal a failure detection algorithm that works on a network of processors and can be tailored to our specific environment. Prior to the discussion of the relocation algorithm some definitions are in order. A *Detection Unit* (see example in figure 2) is a cluster of processors that monitor each other's status and participate in executing the detection algorithm. When a processor fails, attempt will be made to distribute its workload within its containing detection unit. At any given time each detection unit has a unique leader. Let $D = \{P_1, \ldots ,P_m\}$ be a *Detection Unit*. We impose a total order on the set D by the indices of its elements, i.e., $P_i < P_j$ when $i < j$. This is referred to as the *Nomination Order*, and is used to nominate leaders. Each processor is aware of its position within the nomination order. At system generation time the processor which is first in the nomination order, assumes the role of leader L in the detection unit D. We assume that, when a leader fails, the detection algorithm is capable of nominating a new leader in accordance with the nomination order. When a non-leader processor fails, the leader of the detection unit confirms its failure and executes the relocation algorithm as described next section.

4.2 The Relocation Algorithm

The objective of this algorithm is to relocate the processes of the failed site so that:

- the departure from the optimality achieved by the allocation algorithm is as small as possible.
- the workload balancing constraint is observed.

The first condition is best satisfied by restricting all relocation activity to be performed within a detection unit. The second condition, however, may prevent this from being achieved. In such cases, relocation in "neighboring" detection units can be attempted through negotiation between the leaders and a designated central site. In this paper however, we restrict ourselves to performing relocation only within a detection unit. Designing a network wide relocation scheme is an activity for future research.

During initialization each processor within a detection unit constructs its copy of the following information:

- The set D.
- Let R_D be the root of the processor cluster subtree that corresponds to detection unit D, and let r_D be the root of the process cluster subtree that contains all the processes allocated to D. The subtrees rooted at R_D and r_D are kept at each processor of the detection unit.

The algorithm consists of the following steps.

1. The status of the failed processor is propagated to every site in the detection unit D. If P_j is the failed processor then SEND MSG (*leader* to P_x) : MARK_DOWN (P_j) for all $P_x \in D$
   Each processor on receiving the MARK_DOWN message updates its copy of the status for P_j. Note that the leader is also responsible for sending a MARK_UP message for a processor which recovers from a failed state.
2. The leader executes the allocation algorithm with arguments R_D and r_D. The set $\mathcal{P}_V$ is initialized to empty reflecting the fact that during the relocation phase no child of R_D is a violator. The capacities M_i of all nodes in the subtree rooted at R_D are updated to account for the failure of P_j. $\mathcal{P}_A$ is initialized using the updated values of M_i and the current values of c_i. The leader then calls ALLOCATE (r_D, R_D).

5. Conclusions

This paper has addressed the problem of process allocation in a distributed system and the subsequent dynamic relocation of workload under conditions of processor failure to achieve high availability. The approach presented in this paper is novel in that it presents a heuristic algorithm that seeks to partition the total workload evenly between a group of processors and, at the same time, minimize communication cost. The result is a suboptimal solution. Further work needs to be done in quantifying the degree of departure from optimality and generating a formal correctness proof for the algorithm.

The process relocation algorithm uses the notion of *Detection Units*, which allow a distributed system designer flexibility in defining subnetworks within a large network. This arrangement requires smaller network view tables to be maintained at each site and reduces the number of fault detection messages. Further work needs to be done on defining criteria in choosing detection units. The results of this work can then be applied to network design.

References

[1] Barigazzi G., Ciuffoletti A., Strigini L., Reconfiguration Procedure in a Distributed Multiprocessor System, Proc. of FTCS-12, 1982, pp. 73-80

[2] Corsini P., Simoncini L., Strigini L., MuTEAM: A Multimicroprocessor Architecture with Decentralized Fault Treatment, unpublished manuscript.

[3] Garcia-Molina H., Elections in a Distributed Computing System, IEEE Trans. on Computers, vol. c-31, no. 1, January 1982.

[4] Garey M. R., Johnson D. S., Computers and Intractability: A guide to the Theory of NP-completeness, W. H. Freeman & Co. 1979.

[5] Hagouel J., Issues in Routing for Large and Dynamic Networks, Ph.D. dissertation, Columbia Univ., 1983.

[6] Hammer M., Shipman D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, Vol. 5, No. 4, December 1980, pp. 431-466.

[7] Stone H. S., Multiprocessor Scheduling with the Aid of Network Flow Algorithms, IEEE Trans. on Software Eng., Vol. SE-3, No. 1, January 1977

[8] Rao G. S., Stone H. S., Hu T. C., Assignment of Tasks in a Distributed Processor System with Limited Memory, IEEE Trans. on Computers, Vol. C-28, No. 4, April 1979.

[9] Walter B., A Robust and Efficient Protocol for Checking the Availability of Remote Sites, Computer Networks 6(1982), pp. 173-188.