MIGRATION ALGORITHMS FOR AUTOMATED LOAD BALANCING

Niklas Widell
Department of Communication Systems
Lund University
Box 118
SE-221 00 Lund, Sweden
Phone: +46 46 2227195
niklasw@telecom.lth.se

Keywords: Load balancing, resource allocation, distributed computing

Abstract

As distributed systems continue to evolve, automatic resource management is becoming more and more important. The resource management system must be able to dynamically handle large heterogeneous systems in a way that gives good performance and resource utilization. In the performance context, allocating software modules to nodes in an efficient way is of high interest. This paper considers the problem of allocating software modules to processing nodes in an automatic dynamic manner using module migration algorithms. The module allocation problem is NP-complete and many heuristics have been proposed. However, in systems where the workload changes over time, it may be infeasible to update module allocation often enough to handle changes in workload. This paper presents the *Match-maker* algorithm that performs load balancing by pairing overloaded nodes with under-loaded ones, initiating module migration within the pair. The paper presents a load balancing optimization problem, and uses the benchmark problem to evaluate the algorithm. In addition, the Match-maker algorithm is compared with other previously described algorithms for module migration. The Match-maker algorithm is found to be fast and efficient in reducing load imbalance in distributed systems, especially for large systems.

1 Introduction

There is considerable interest in autonomic management systems for distributed systems. One important area for management is the efficient allocation of software to processing elements. Since the allocation will have impact on the performance of the system, it is of great interest to automate the allocation procedure, just as resources are automatically allocated ordinary computers today.

Several objectives can compete in performing the allocation procedure:

1. Avoid unnecessary communication: communication overhead is wasted resources.
2. Utilize available resources: try find allocations that share the load on the nodes.
3. Maintain security policies: certain parts of the software may only be executed at certain trusted nodes
4. Maintain reliability policies: parts, whether nodes, network elements or other, in a distributed system can and do fail, the resource management system must handle the failure in such a way as to reduce the impact on the users of the system
5. Economic concerns: minimize the economic cost of running the application for the owner of the application and maximize the revenue for the operator of the system.

We make a difference between static and dynamic allocation algorithms. In static allocation, which has been widely studied, a model of the system is used to formulate and solve an optimization problem, that gives an optimal allocation. In dynamic allocation, which has received less attention, no predefined allocation is used. Instead, the system migrates modules during run-time in response to changes in load and availability of resources.

The module allocation problem is really a family of problems with different formulations, where the common idea is to find an optimal allocation of modules according to a given obective function. An overview of results, including several formulations, is presented in a survey by Norman and Thanisch [1]. This reference also presents heuristics devised to solve module allocation. Of newer works not presented in [1], see for instance Woodside and Monforton [2], Stoyenko *et al* [3] or Braun *et al* [4].

Little has been published on module migration algorithms. Silaghi and Keleher [5] presented an algorithm for migrating objects in a Java system, where migration could happen at high rates. Melliar-Smith *et al* [6] introduced two types of simple migration algorithms for use in the real-time CORBA Realize environment: *Cooling* and *Hotspot*. The *Cooling* algorithm has also been implemented in CORBA by Schnekenburger and Rackl [7].

Module migration is not free from overhead costs. However, systems using static module allocations suffer from overhead. First, running the allocation optimization algorithm is, as mentioned before, very time consuming. Second, if load situation in the network changes, then modules must migrate just as they would in the migration algorithm case. Third, in the static case the migration of all modules is initiated at one time instead of being spread out over longer time.

Present research on module migration has not investigated any of the following: (1) how close to optimal the migration algorithm is, (2) the dynamic behavior of migration algorithms, (3) the scalability of the migration algorithm. This paper studies these questions. The paper does not cover implementation specifics. Implementation issues are discussed in Schnenkenburger [8] and Henning [9].

The main contribution of this paper is to show by example that a module migration algorithm with low overhead can be used to efficiently distribute load in a distributed system. The paper also discusses how migration algorithms may be analyzed by comparison with theoretically optimal solutions.

In section 2 we describe a model for a module based distributed system. In section 3 we describe a reference load balancing optimization problem. In section 4 we introduce a novel algorithm for module migration called the Match-maker algorithm. In section 5 we evaluate the Match-maker algorithm in comparison with allocations generated using the reference optimization problem as well as other migration algorithms. Finally in section 6 we present conclusions and general results.

2 Model

In this section we describe a model of a distributed module based system, to be used as reference model in the discussion of the migration algorithms. As a system's performance is the result of interaction between software and hardware, models for both hardware and software are presented.

We consider a distributed system where communicating software modules are to be allocated to physical processing *nodes* in a network. In figure 2, the rectangles are nodes, circles are modules, lines are communication requirements between modules and dashed lines indicate an allocation of modules to nodes.

The physical resources in the model is represented by a collection of N processing nodes. The nodes are heterogeneous, and a particular module incurs different costs when executed on different processors, due to varying computational facilities at each node. For the present study, assume that the nodes belong to a common domain with high network speed compared to node processing capacity, such as computers connected to a high speed LAN. Thus, network latency is negligible, and the nodes can be assumed to be fully connected.

In the model we use the term *load* as the workload generated by software modules running on physical processors. We assume that the load of a module running on

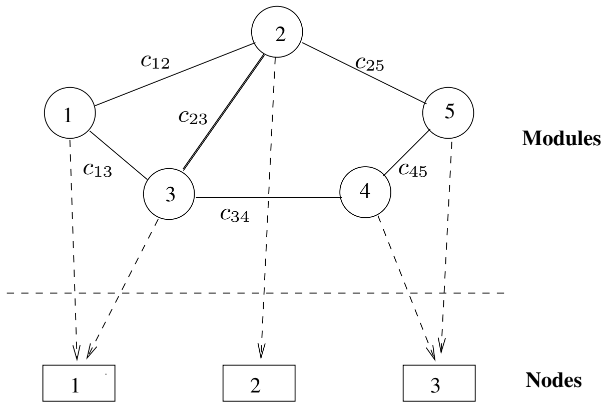

Figure 1. System model

a processor can be measured. Also, high load on a node means that delays become increasingly large.

The software in the system is represented by by M modules. A module is an atomic software entity, like an object or component, that can be migrated from node to node. The execution load for a module is given by an $N \times M$ matrix $\mathbf{E}$, where element e_{ij} is the execution load for module j allocated to node i. $e_{ij} = \infty$ if module j may not be allocated to node i, due to security or domain membership reasons.

Modules interact using an RPC-like protocol. The communication model is similar to Bokhari [10], where the actual communication cost between two modules is zero if they are co-allocated on the same node. Otherwise, the communication load is given by an $M \times M$ matrix $\mathbf{C}$, where element c_{jk} is the communication load if modules j and k are allocated to different nodes. Note that $c_{jj} = 0$ and we assume that $c_{jk} = c_{kj}$. The communication load models the protocol handling required to support communication between nodes, and is thus a cost for the node hosting the modules involved in communication. We do not consider the loading of the underlying network, since we have already assumed that it is of high enough capacity.

An activity is a sequence of module interactions, with given start and end times. It is the arrival of activities that make the modules work. The working modules generate load on the nodes they are allocated to.

The allocation of modules to nodes is given by an allocation matrix α with binary elements. Element $\alpha_{ij} = 1$ if module j is allocated to node i, $\alpha_{ij} = 0$ otherwise. When many allocations are considered at the same time, let $\alpha_{k,ij}$ be α_{ij} for allocation k.

3 Reference optimization problem

The module allocation problem is to allocate modules to physical nodes in a way that best satisfies a given set of objectives and constraints. Typically, several conflicting objectives exist and the exact formulation depend on the problem application.

The main conflict of objectives for performance is between clustering and distribution of modules. By clustering modules that communicate the communication cost can be reduced. However, heavy clustering results in imbalance allocations where some nodes have most modules and others have only few. Imbalanced allocations in turn typically results in long delays as some nodes become overloaded while others remain almost idle. By distributing modules higher parallelization is gained, at the cost of more resources wasted to communication processing. However, too much distribution results in allocations where a large part of available resources are used to support communication.

The objective may be formulated in many ways, depending on application and system parameters. Since the objective of this work is to study load balancing using module migration, we chose an objective that minimizes the load imbalance in the system.

First, let the the workload w_i for node i given allocation α be:

$$w_i(\alpha) = \sum_{j=1}(e_j\alpha_{ij} + \sum_{k=1}^{M} c_{jk}\alpha_{ij}(1-\alpha_{ik})) \qquad (1)$$

The objective of the reference problem is to allocate software modules to nodes so that system load imbalance is minimized. This means that the total load on node i should be as close as possible to the average load $w_{ref,i}$ for node i given allocation α. In our notation, the load imbalance L given allocation α is:

$$L = \sum_{i=1}^{N}\sum_{j=1}^{M} |w_i(\alpha) - w_{ref,i}(\alpha)| \qquad (2)$$

where $w_{ref,i}(\alpha)$ is the average load of the nodes weighted by node capacity.

Each node may not be allocated more than it's capacity:

$$0 \leq w_i < 1.0, \quad i = 1..N \qquad (3)$$

The optimization problem is to find the α that minimizes L subject to (3).

4 The Match-maker algorithm

In this section we describe a new algorithm for module migration, called the *Match-maker algorithm*. We also briefly describe other published migration algorithms and discuss important differences between them.

The objective of the Match-maker algorithm is to balance the load on nodes as efficiently as possible and thus minimize load imbalance. The balancing operation is performed by matching nodes with high loads with nodes with low load, and then letting migration take place within the pair. The pseudocode in figure 2 describes the algorithm.

```
node_list.sort()
for each n in node_list:
    n.module_list.sort()
$i = 1$
$k$=node_list.length() ($N$)
while $i < k$:
    ns=node_list(i)
    nd=node_list(k)
    if ns.load $>$ ns.$w_{ref}$ and nd.load $<$ nd.$w_{ref}$:
        $j = 1$
        while $j \leq$ns.module_list.length():
            m=ns.module_list($j$)
            if nd.can_accommodate(c) and
                        nd.load + m.load $<$ nd.$w_{ref}$:
                migrate(ns,nd,m)
            else:
                $j = j + 1$
    $i = i + 1$
    $k = k - 1$
```

Figure 2. The Match-maker algorithm

Each pair has one node with high load called the *source node*, n_s and one node with low load called *destination node*, n_d. In the pair (n_s, n_d), n_s will be the source from which a module may be migrated to destination n_d.

The algorithm works in intervals of length T. During the interval load in the system is measured. The matching operation is performed at a centrally located *coordinator* node collects load information and then redistributes it to other nodes. The coordinator is an ordinary node that acts as a hub for transferring load information in the network.

The matching procedure is made by load imbalance order: the first pair has the most overloaded node and the must under-loaded node, the second pair has the second most overloaded node and the second most under-loaded and so on.

For each (n_s, n_d) pair, the module to be migrated is selected in a greedy fashion: migrate the largest (in load) module that will fit on n_d, since this will simultaneously decrease the load imbalance on both nodes the most. The *can_accommodate(mod)* function handles non-performance related allocation policies, such as security or reliability.

4.1 Comments

Load metrics: We assume that the individual nodes can estimate the fraction of its total load that is due to a given module allocated to the node. The load metrics are recorded periodically and reflect the average load in a moving window of time. As the modules migrate to and from a node, the node's load is updated with an estimate of the migrating module's load.

Chosing a coordinator: Any node in the system can be used as coordinator. However, it is preferable to use a node with good connections to other nodes.
Scalability: The Match-maker algorithm is scalable in the sense that in large systems with many nodes the system can be partitioned into groups. Each group can have it's own coordinator and nodes are paired within the group, without necessarily impairing algorithm performance. If the node groups are used, nodes can rotate among the groups to improve the load balancing by allowing more choices.
Overhead: The overhead associated with the Match-maker algorithm includes the load information sent to the coordinator and the reply messages sent to each source node, a total of at most $\frac{3N}{2}$ messages. The decision process in the coordinator is simple. The module selection is also simple, possibly requiring information to be passed from destination nodes. The actual migration of a module is likely to be expensive and difficult. However, the migration workload is distributed as much as possible, as no node takes part in more than one migration during a single interval.

4.2 Comparison with other algorithms

Cooling [7] [6] The Cooling algorithm is similar to Match-maker on the surface, the difference lies in which order modules are chosen. The Match-maker algorithm first finds (n_s, n_d) pairs, and then attempts migration within each pair. In the Cooling algorithm, the most heavily loaded node is always the source node, and the module with highest load on that node is the primary candidate for migration. The destination node is the least loaded node that can accommodate the candidate module. The advantage of Match-maker is that several migrations can be made in parallel, this being more difficult in the case of the Cooling algorithm. The Cooling algorithm requires all lists of module loads to be submitted by nodes at end of every interval. This is not necessary for the Match-maker algorithm, since the source node knows what modules it has and is likely to be able to choose a good candidate for migration.
Hot spot [6] The hot spot algorithm is similar to Cooling, with the addition of looking at estimated queuing latency of an activity when deciding what to migrate. The module that gives the greatest latency to an activity is migrated to a node where it's added estimated latency is less than on the source node. As we do not presently consider task latency, we do not further discuss the Hot spot algorithm in this paper.

5 Evaluation

This section describes a series of experiments that we have run to illustrate the efficiency of the Match-maker algorithm and compare it with the Cooling algorithm and static allocation methods. No useful reference problems for allocation problems exist, therefore Parameters had to be randomly generated.

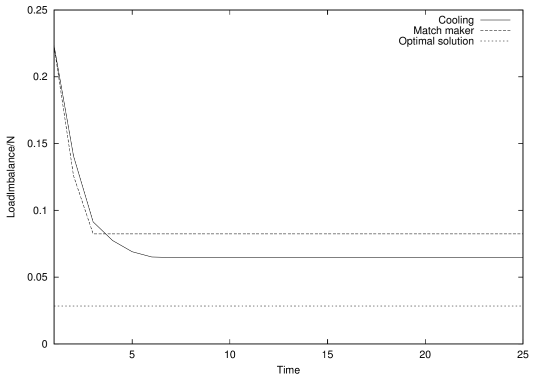

Figure 3. Experiment 1: Small network, M=14 and N=6. Optimal value for allocation is 0.03

Experiments were conducted using simulation. In the experiments, we were mostly interested in load balancing characteristics of the investigated algorithms. We therefore let communication cost be zero ($c_{ij} = 0$ for all i and j). Note that communication overhead is not taken into account by other work on migration algorithms ([5], [6] and [7]).

In all simulations, the total load on the system was $w_{total} = N/2$. All nodes are assumed to be equal in terms of processing powers, and all nodes could (potentially) accommodate all modules. Modules were randomly generated with e_{ij}:s normalized to bring system total load to w_{total}. In experiments 1, 2, 3 and 4 e_{ij}:s were constant. In experiment 5 e_{ij}:s were variable with mean e_{ij}.

The data in each figure represents averages taken over ten simulations in each case.

The measure used for evaluation is the measured load imbalance L divided by the number of nodes N.
Experiment 1: A small problem In this case $N = 6$ and $M = 14$. The optimal allocation was derived using AMPL and CPLEX. The solution time, using a SUN Ultra 60 Workstation, was 32 minutes to derive the optimal allocation. Figure 3 shows the results from the simulation. The dotted line at the bottom shows the result for optimal allocation. We see that for this small system neither migration algorithm perform very well, with Match-maker doing about 20% worse than Cooling. The reason for the poor performance is that with the small ratio of M to N, the migration algorithms are unable to move anything since overloaded nodes have no modules that the under-loaded nodes can receive.
Experiment 2: A mid-sized toy problem Since it is very time-consuming to solve the reference problem for larger systems, a problem with special parameters that had a known objective and known optimal allocation was developed. This special problem allows larger systems to be studied.

The idea was to create a system with only one possible optimal combination. Equivalent permutations of the

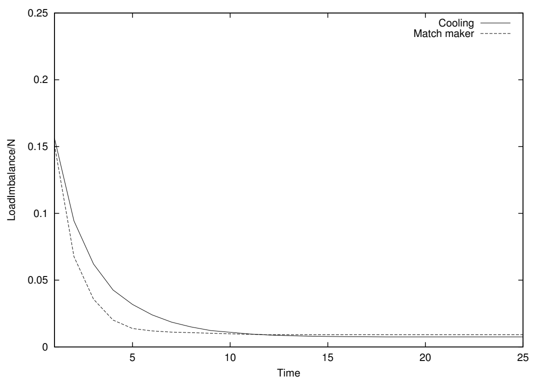

Figure 4. Experiment 2: Toy system, M=129 and N=10. Optimal value for allocation is 0.

optimal allocation was considered equal. The parameters for the system was generated in the following way: Let the number of nodes be N. Let p_l be the lth prime number, starting at $p_1 = 2$. Let $M = \sum_{l=1}^{N} p_l$. Now, let there be p_l modules, each with load $e_{ij} = \frac{1}{2p_l}$ for each l from 1 to N.

For example, for $N = 10$, there are two modules with load $\frac{1}{2 \cdot 2}$, three with $\frac{1}{2 \cdot 3}$, up to finally 29 modules each with load $\frac{1}{2 \cdot 29}$.

Given these modules, it was easy to see that the optimal value of the objective function is 0 and the optimal allocation placed the two modules with load $\frac{1}{4}$ on one node, and the three with load $\frac{1}{6}$ on another and so on.

Figure 4 presents the results given a toy problem system ($N = 10$ and $M = 129$). From the figure it can be seen that both algorithms find allocations that are close to optimal. Both algorithms solutions that are much more equal than in experiment 1.

Experiment 3: Transient loads In this experiment we let the workload at certain intervals. The change models a transient in module access pattern, for instance caused by a change in what activities are performed. The first transient removes 75 modules from the system. The load on the remaining modules is increased to keep total system load constant at $w_{total} = N/2$. The second transient returns the removed 75 modules and also returns remaining modules to their original loads. The results from the simulations are found in figure 5. In the figure we see that both algorithms adapt to the transients with the same behavior as in experiment 2, with impact on load balance scaled down to take into account that the system prior to the transient was load balanced already for a slightly different set of modules.

Experiment 4: Scalability In this experiment we investigated the influence of ratio between M and N. In the experiment, $N = 40$ was constant, while M was increased from 100 to 290 in steps of 10. Every point in the diagram represents the mean of ten samples. The samples were the best values attained by the algorithm during simulation. Figure 6 shows mean optimal values reached by algorithms as function of M. Both algorithms perform better when the ratio of M to N is large. Cooling outperforms Match-maker as is expected.

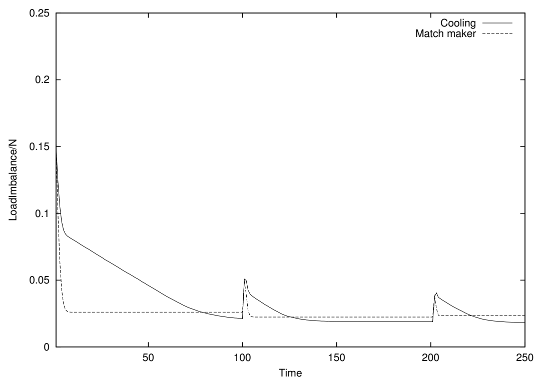

Figure 5. Experiment 3: A large system with transients at T=100 and T=200 (N=80, M=600)

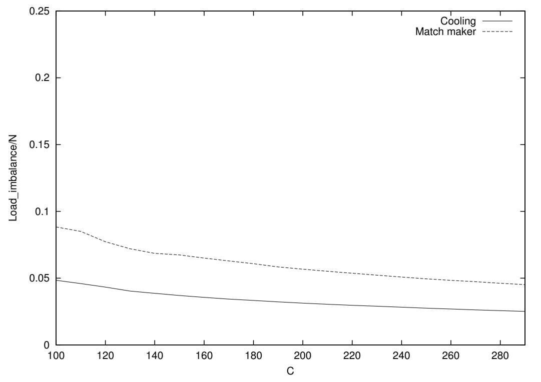

Figure 6. Experiment 4: Load imbalance as function of number of modules. (N=40)

Experiment 5: Stochastic input traffic In this experiment we used variable input traffic generated by a Poisson source. Activities arrived with $\lambda = 20s^{-1}$. Service times were deterministic and calculated to give mean system load $\overline{w}_{total} = N/2$. Activity structure was simple: each activity calls each module once. This simplification could be done as no internode communication overhead was modeled.

Figure 7 indicates that the averaged result over 10 simulations. The general difference in convergence speed between Match-maker and Cooling is apparent. The average load imbalance per node in the interval T=50 to T=150 shows only a 5% difference between Cooling and Match-maker.

5.1 Discussion of results

The simulations show that both Match-maker and Cooling algorithms quickly improve load balancing in system. Match-maker is faster in convergence, but Cooling finds a slightly better allocation. In small systems, Cooling is

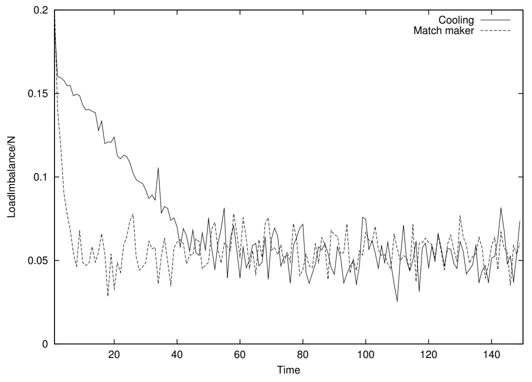

Figure 7. Experiment 5: Variable loads due to Poisson input traffic. N=40 and M=300.

more efficient. In large systems, Cooling takes a long time to converge, and the difference in load imbalance is not very large. In terms of overhead, Match-maker performs about 10% fewer migrations than does Cooling before convergence.

The choice of migration algorithm is between slower and more exact or faster and less exact for Cooling and Match-maker respectively. What should also taken into account is the need for centralized versus distributed decision making. Cooling depends on a centralized control to do all decision making as all system information must be collected at the central node. Match-maker depends on a central node only to do the match-making, the migration control is distributed to the nodes that may do it.

6 Conclusions

To shield the implementor of a distributed program from the difficult details of implementing efficient load balancing, transparent and automatic mechanisms are needed. The Match-maker and Cooling migration algorithms discussed in this paper are examples of such mechanisms. The algorithms are likely to provide sub-optimal solutions to the load balancing problem. However, the algorithms work with no a-priori knowledge of the system and are yet able to find good solutions even in large systems very quickly.

The Match-maker algorithm described in this paper migrates modules from overloaded nodes to nodes with spare capacity, continually improving the solution. The algorithm has low complexity and high adaptability, and as such is well suited for performing load balancing in dynamic systems where both traffic levels and activity mixes changes. The Match-maker algorithm is much faster than the previously published Cooling algorithm. The Cooling algorithm finds allocations that yield 5-10% lower load imbalance than the Match-maker for large systems. Both algorithms perform better when the ratio of modules to nodes is high. Both algorithms also efficiently handles changes in activity behavior, even though Cooling is slower to respond.

7 Acknowledgments

This work has been partially sponsored by the Swedish Research Council under contract no. 621-2001-3053.

References

[1] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Journals*, Vol 25, No 3, 1993.

[2] C. Murray Woodside and Gerald G. Monforton. Fast allocation of processes in distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):164–174, 1993.

[3] A. Stoyenko, J. Bosch, M. Aksit, and T. Marlowe. Load balanced mapping of distributed objects to minimize network communication. *Journal of Parallel and Distributed Computing*, Vol 34: 48-66, 1996.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bllni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, 1999.

[5] B. D. Silaghi and P. J. Keleher. Object distribution with local information. In *Proceedings of 21st IEEE International Conference on Distributed Computing Systems*, 2001.

[6] Moser Melliar-Smith, P. M., L. E., V. Kalogeraki, and P. Narasimhan. Realize: Resource management for soft real-time distributed systems. In *Proceedings of IEEE Information Survivabilty Conference, SC*, 2000.

[7] T. Schnekenburger and G. Rackl. Implementing dynamic load distribution strategies with Orbix. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997.

[8] T. Schnenkenburger. Load balancing in CORBA: A survey of concepts, patterns and techniques. *The Journal of Supercomputing*, Vol 15: 141-161, 2000.

[9] M. Henning. Binding, migration and scalability in CORBA. *Communications of the ACM*, Vol 41: No 10, 1998.

[10] S. H. Bokhari. Partitioning problems in parallel pipelined and distributed computing. *IEEE Transactions on Computers*, Vol 37, No 1, 1988.