

# Safety and Reliability Driven Task Allocation in Distributed Systems

Santhanam Srinivasan, *Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE*

**Abstract**—Distributed computer systems are increasingly being employed for critical applications, such as aircraft control, industrial process control, and banking systems. Maximizing performance has been the conventional objective in the allocation of tasks for such systems. Inherently, distributed systems are more complex than centralized systems. The added complexity could increase the potential for system failures. Some work has been done in the past in allocating tasks to distributed systems, considering reliability as the objective function to be maximized. Reliability is defined to be the probability that none of the system components fails while processing. This, however, does not give any guarantees as to the behavior of the system when a failure occurs. A failure, not detected immediately, could lead to a catastrophe. Such systems are *unsafe*. In this paper, we describe a method to determine an *allocation that introduces safety* into a heterogeneous distributed system and at the same time attempts to maximize its reliability. First, we devise a new heuristic, based on the concept of *clustering*, to allocate tasks for maximizing reliability. We show that for task graphs with precedence constraints, our heuristic performs better than previously proposed heuristics. Next, by applying the concept of *task-based fault tolerance*, which we have previously proposed, we add extra *assertion* tasks to the system to make it safe. We present a new heuristic that does this in such a way that the decrease in reliability for the added safety is minimized. For the purpose of allocating the extra tasks, this heuristic performs as well as previously known methods and runs an order of magnitude faster. We present a number of simulation results to prove the efficacy of our scheme.

**Index Terms**— Allocation, clustering, distributed systems, reliability, safety.

## 1 INTRODUCTION

DISTRIBUTED computer systems consist of loosely coupled processors which communicate with one another only by passing messages and which do not have common memory. Over the last several years, they have become popular as a very attractive option for fast computing and information processing. Increasingly, they are being employed for critical applications, such as aircraft control, industrial process control, and banking systems. The chief incentives for choosing distributed computing are higher throughput, improved availability, and better access to a widely distributed web of information. The increased commercialization of distributed systems means that ensuring system reliability is of critical importance. Inherently, distributed systems are more complex than centralized systems. The added complexity could increase the potential for system faults. Hardware redundancy is the traditional technique to introduce fault tolerance in these systems. However, this is an expensive approach. Moreover, many times, the hardware configuration is fixed and we do not have the freedom to introduce hardware redundancy. Hence, we have to turn to software techniques to achieve hardware fault tolerance. Our work is motivated by this need to improve the reliability of the distributed system with no extra hardware cost.

The software to be run on the distributed system is called a *job*, which is composed of intercommunicating *tasks*. These tasks are allocated to the different processors in the system. Allocation techniques can be either *static* [1], [2], [3], [4], [5] or *dynamic* [6], [7]. Static allocation schemes, unlike dynamic techniques, assume a priori knowledge of the characteristics of the workload with respect to the system including the component tasks, their execution times, etc. Such techniques can be applied at compile time itself and are very efficient. We shall be considering static allocation in this work. The allocation is done in order to optimize one or more of a number of objectives like minimization of interprocessor communication (IPC), load balancing on the processors, memory utilization, etc. Allocation schemes can be classified into two categories. First, there are the exact methods that try to find the optimal allocation for the given objective. There exist approaches based on graph-theoretic techniques [8] and integer programming [9], [10] to determine the optimal solution. However, as the problem is NP-complete, these methods are computationally very expensive. The other approach is to devise good heuristics [5], [11], [12], [13], [14] to solve the problem. These may give suboptimal results, but are much less expensive.

A number of heuristic methods like linear clustering [1], [15] and hierarchical clustering [2], [3] perform allocation and scheduling to minimize the schedule length. None of these methods give any consideration to improving the reliability of the system. Task redundancy has been used for fault detection and masking in [16] by running two or more copies of the task. Task allocation is done dynamically. However, if each copy of the task is allowed to complete, then the degradation in throughput can be substantial. If the copies are allowed to be preempted then fault detection

- S. Srinivasan is with the High Speed Networks Research Department, Lucent Bell Laboratories, 101 Crawfords Corner Road, Holmdel, NJ 07733.
- N.K. Jha is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544.  
E-mail: jha@ee.princeton.edu.

Manuscript received 21 Apr. 1995; revised 30 Sept. 1997.  
For information on obtaining reprints of this article, please send e-mail to: tpd@computer.org, and reference IEEECS Log Number 104520.

and/or masking may not be achievable for all tasks. In [14], a general scheme called *task-based fault tolerance* was developed to introduce fault security in the system. A system is said to be *fault secure* if in the presence of a fault, the system either detects the fault or always gives the correct outputs. Faults in the system are assumed to occur one at a time and before a second fault occurs, the checks are assumed to detect the presence of the first fault. This is a standard assumption made in the context of concurrent error detection schemes and self-checking circuits [17]. A system is defined to be *safe* if either the computed result is correct or if the system is able to detect the presence of faults/errors when they occur [18]. Fault security combined with the assumption about the occurrence of faults in the system, makes the system safe. The task-based fault tolerance scheme uses the concept of *assertions* to achieve fault security/safety. An *assertion task* checks some property of the output data from the original task and if the property is not satisfied, it declares the output data to be erroneous and hence indicates the presence of a fault in the system. Assertion tasks are added to the system for all tasks that are assertible and only for the remaining tasks are duplicates introduced. This takes advantage of the fact that a large number of real-life tasks are assertible. The overhead incurred by using assertion tasks is much less as compared to the case where all tasks are duplicated, since the assertions do not have to recompute the data but only check some property of the data.

In [14], however, no explicit reliability model is assumed, and minimization of schedule length is used as the objective. Reliability has been considered as the objective function for allocation in [19] and an explicit measure of the reliability of an allocation has been introduced. (An exact method for allocation to maximize reliability has been proposed in [20]; however, as remarked earlier, exact methods are computationally very expensive.) *Reliability* is defined to be the probability that the system will not fail during the time that it is processing the job [19]. This, however, does not give any guarantees about the behavior of the system when a failure does occur. Such failures, if not caught immediately, could be catastrophic in safety-critical applications.

Our main motivation in this paper is to introduce *safety* into a heterogeneous distributed system through software techniques. Hardware techniques for safety and the trade-off between reliability and safety have been addressed in the literature [21], [22]. Some techniques for reliability and safety analysis have been described in [23].

In introducing safety, we would like to encompass both transient and permanent faults. When a transient fault is detected one can retry the job or just the affected tasks till the fault goes away. If a permanent fault is detected, one can, after locating the fault, either reconfigure the system around the fault or replace the faulty component and then rerun the job. We need to note that a system can have very high safety but very low reliability. Clearly, this is not satisfactory. In this paper, we discuss techniques to introduce safety in the system such that its reliability is maximized.

The first phase of our two-phase scheme allocates the tasks of the original task graph with reliability as the objective. Our allocation procedure uses the concept of *clustering based on static levels*. In [19], the task graphs considered do not have any precedence constraints. We show that for task graphs with precedence constraints, our method performs better in terms of reliability and is comparable in terms of CPU time. In the second phase, the fault tolerance tasks are allocated such that the decrease in reliability is minimized. We introduce a new task-level allocation procedure for the extra tasks that produces results of the same quality as those in [19] but runs an order of magnitude faster.

The rest of the paper is organized as follows: In Section 2, we give the basic definitions and concepts used in the rest of the paper. Section 3 presents our allocation method for the original tasks. Section 4 discusses techniques for task-level allocation of the fault tolerance tasks. Section 5 briefly discusses the scheduling method employed. In Sections 6 and 7, we present our simulation results and conclusions.

## 2 DEFINITIONS

Our model of a distributed system is the same as that assumed in [14]. The input to the distributed system is a job composed of intercommunicating tasks represented as an acyclic directed graph called the *task graph*. We assume that the tasks, once defined, are not further divisible. The process of *task allocation* takes as input the task graph and the *target architecture* specified by the processor nodes, their interconnection, and a number of parameters defining the relationship between the tasks in the task graph and the processors. It then attempts to find a mapping of the tasks to the processors using some optimization criterion. In this work, we address the problem of task allocation in order to introduce safety in the system and at the same time maximize its reliability.

### 2.1 Task Graph

The task graph is a directed acyclic graph  $G = \{T, E\}$ , where  $T = \{t_1, t_2, \dots, t_n\}$  represents a set of  $n$  tasks and  $E$  represents the set of weighted and directed edges among the tasks. The communication from task  $t_i$  to task  $t_j$  is represented by a directed edge  $(t_i, t_j)$  between the two tasks. The weight on this edge,  $c_{ij}$ , represents the volume of data being transmitted from task  $t_i$  to task  $t_j$ . Each node in the task graph is labeled with a vector called *Exec\_cost*:

$$Exec\_cost(t_i) = [e_{i1}, \dots, e_{im}]$$

The  $j$ th element of this vector,  $e_{ij}$ , represents the execution time of  $t_i$  on processor  $p_j$ . If task  $t_i$  cannot be executed on processor  $p_j$ , the corresponding execution time  $e_{ij}$  is  $\infty$ . A *source task* in a task graph is defined as a task which has only primary inputs as fanins. A *sink task* in a task graph is defined as a task which does not have any task graph nodes as fanouts.

We use the concept of *task exclusion* [9] to force two tasks to be allocated to two different processors. Such a constraint may arise when both tasks use some resource very intensively and allocating both to the same processor might

overload it. For example, if two tasks which require large amounts of memory are allocated to the same processor, the processor will have to keep swapping data from the disk and, hence, will be slow. Such constraints can be specified by means of a matrix defined as the *exclusion matrix*  $Ex$  [9]. The exclusion matrix has tasks as rows and columns. Entry  $Ex(i, j)$  is "1" if task  $t_i$  cannot be on the same processor as task  $t_j$ . If there are no constraints between two tasks, the corresponding entry is "0." The matrix is symmetric by definition and the diagonal entries are all "0"s.

We use a variant of the concept of *task preference* used in [9] to handle multiple resource requirements of a task. In general, a task may not be able to execute on every processor as it may have some specialized requirements (like requiring a special floating point co-processor or I/O device) satisfiable only by certain processors. We represent these constraints by a *preference matrix*  $Pr$ . This is a binary matrix which has tasks as rows and processors as columns. Entry  $Pr(i, j)$  in the matrix is "0" if task  $t_i$  cannot execute on processor  $p_j$  and is "1" otherwise. Thus, if task  $t_i$  can be allocated to a number of processors, the row corresponding to  $t_i$  in the preference matrix has "1" in the columns corresponding to each such processor. A task with no constraints would then be represented by a row of all "1"s. Note that a row of all "0"s is not allowed as this precludes the task from being allocated to any of the processors. In contrast to the three-valued nature of the task preference matrix in [14], our binary matrix definition is computationally more attractive as we do not have to use algebraic operations (Boolean operations suffice).

A point to note is that apart from arising as natural system constraints, the concepts of task exclusion and preference also give a handle to the user to provide hints to the system in order to fine tune the performance with respect to second order effects that might not be captured by our model of the system.

Each node in the task graph is also assigned a label called *comp\_cost*. This is the average of the execution costs of the task on those processors on which it can be executed.

## 2.2 System Model

The distributed system is assumed to consist of a set  $P = \{p_1, p_2, \dots, p_m\}$  of heterogeneous processors connected by an arbitrary interconnection network. The processors only have local memory and do not share any global memory. The *processor graph* is a convenient abstraction of the processors together with the interconnection network. It has processors as nodes and there is a weighted edge between two nodes if the corresponding processors can communicate with each other. The weight  $w_{ij}$  on the edge between processors  $p_i$  and  $p_j$  represents the delay involved in sending or receiving a message of unit length from one processor to another. In order to have an approximate estimate of this delay, irrespective of the two processors, we use the average of the weights on all the edges in the processor graph. This is called the *average unit delay*.

Conventionally, IPC has been primarily viewed as a major factor inhibiting parallelism. Most schemes try to reduce IPC with the aim of decreasing schedule length. As will be evident from the discussion in Section 2.3, IPC is also

very detrimental to the reliability of the system due to the unreliable nature of the communication links.

Based on the parameters of the underlying architecture, our algorithm tries to minimize the IPC. We assume that once a task has completed, the processor computing the task stores the output data in its local memory. If the data is needed by a task being computed on the same processor, it reads it from the local memory. The overhead incurred by this is negligible, so for all practical purposes we will consider it to be zero. Using this fact, our algorithm tries to group together heavily communicating tasks and allocate them to the same processor. In case two communicating tasks are allocated to different processors, IPC overhead is incurred in communicating data between these two processors. In such cases, our algorithm tries to allocate the heaviest inter-task communication onto the most reliable links. We must note, however, that in order to improve the reliability the other constraint is that the tasks with higher computation times have to be allocated to more reliable processors. Concentrating on just one of the two factors may actually adversely affect reliability. Hence, our algorithm factors in both constraints in attempting to determine the most reliable allocation.

The *allocation matrix*  $X$  is an  $n \times m$  binary matrix representing the mapping of the  $n$  tasks to the  $m$  processors. Element  $x_{ij}$  is "1" if task  $t_i$  has been assigned to processor  $p_j$  and is "0," otherwise.

**Example 1.** Example task and processor graphs are shown in Fig. 1. The task graph has 15 tasks  $a, \dots, o$  and the processor graph has three processors  $p_1, p_2, p_3$ . The set of source and sink tasks are  $\{a, e, j, k\}$  and  $\{d, i, o\}$ , respectively. Tasks  $c$  and  $f$  are assumed to mutually exclude each other. Hence, the task exclusion matrix has a "1" in the position corresponding to these two tasks. Each node in the task graph is labeled with its *comp\_cost*. The matrix of execution costs is given below. The average unit delay is 1.0.

	$p_1$	$p_2$	$p_3$
a	1.8	2.2	2.0
b	3.0	2.9	3.1
c	8.1	8.1	7.8
d	6.1	6.2	5.7
e	3.2	2.9	2.9
f	4.1	4.1	3.8
g	2.0	2.1	1.9
h	4.1	3.9	4.0
i	2.0	2.0	2.0
j	2.0	1.9	2.1
k	2.9	3.1	3.0
l	4.0	4.1	3.9
m	1.1	1.2	0.7
n	2.0	1.9	2.1
o	2.9	3.0	3.1

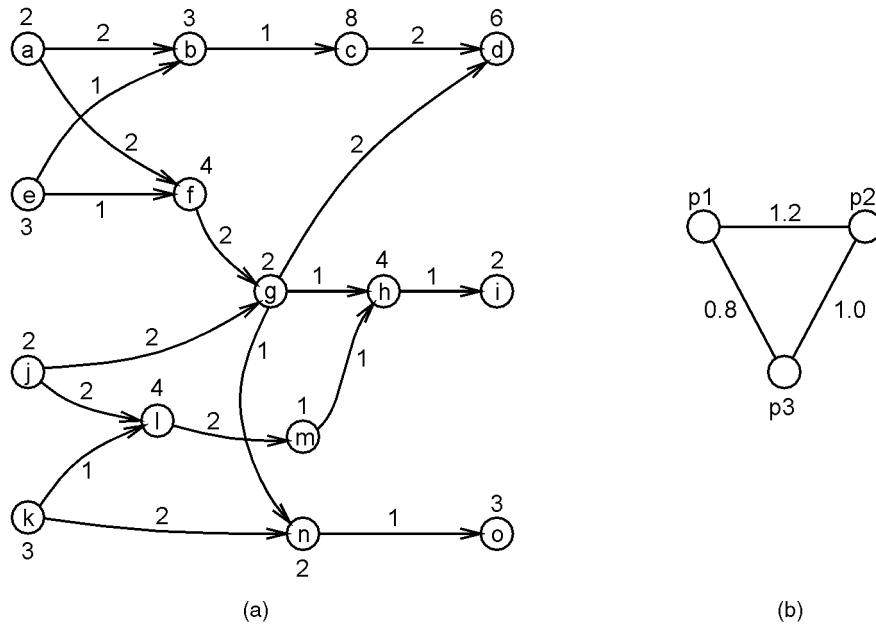


Fig. 1. Example task and processor graphs.

### 2.3 Reliability Model

We use the same model for reliability as in [19]. In this model, the processor and link failures are considered time-dependent and treated in a uniform fashion. An explicit cost function is derived for the reliability of an allocation. We will reproduce the important notations and definitions here for completeness.

Components are either *operational* or *failed*. Components failing during idle periods, when no jobs are running, are replaced and such failures are not considered critical. The future life of all components are assumed to follow a Poisson process with a constant failure rate. Failures of components are assumed to be statistically independent.

#### 2.3.1 Reliability Analysis

A closed form expression has been derived in [19] for the reliability of a given allocation. This expression can be used to drive our algorithms in searching for an allocation that maximizes the reliability. Consider an  $n$ -node task graph and  $m$ -node processor graph. We denote the failure rate of processor  $p_i$  by  $\lambda_i$  and the failure rate of the link between  $p_i$  and  $p_j$  by  $\mu_{ij}$ . Given the allocation matrix  $X$ , we can write the reliability cost,  $relcost(X)$ , as [19]:

$$relcost(X) = \sum_{j=1}^m \sum_{i=1}^n \lambda_j x_{ij} e_{ij} + \sum_{k=1}^{m-1} \sum_{b>k}^m \sum_{i=1}^n \sum_{j=1}^n 2\mu_{kb} x_{ik} x_{jb} c_{ij} w_{kb}, \quad (1)$$

where the reliability is given by  $e^{-relcost(X)}$ . The "2" in the above equation is due to the fact that each item of data being transmitted along a link incurs a *send* and a *receive* cost.

In order to increase the reliability, we have to reduce  $relcost$  as much as possible. We get the following insights from this equation. The first term on the right hand side is

the contribution due to the execution cost of the tasks on their corresponding processors. This tells us that allocating tasks with greater execution times to more reliable processors might be a good heuristic to increase the reliability. The second term on the right hand side is due to the communication of tasks across interprocessor links. In this context, we need to note two things. First, IPC needs to be reduced as much as possible. Of course, we cannot avoid IPC altogether. However, allocating larger volumes of data to more reliable links is a good heuristic to increase the overall reliability. We shall shortly quantify these ideas and show how to effectively handle both constraints simultaneously.

We shall be using expressions derived from (1) to drive our allocation algorithm for safety and reliability.

### 2.4 Assertions

We assume that a fault in a processor will result in an error in at least one task performed on it and that the failure of a link will corrupt at least one data item being sent along the link. In order to introduce safety in the system, each task in the task graph is either checked by an assertion task, whenever possible, or else is subject to duplication and comparison [14]. For assertion checking, the original task is modified by encoding its data elements using a system-level code. The encoded output data elements are given to the assertion task which checks their encoding. If the encoding is not satisfied, it indicates the existence of an error and correspondingly, a faulty processor or link. An assertion can be of any type, e.g., checksums in a matrix-matrix multiplication [24], sum-of-squares in FFT [25], sort-check for a sorted list [26], type and range check for certain data elements etc. In the absence of such a property, which can be checked, duplication of the task and comparison between the two copies is used as a means of checking. It must be noted that an assertion can be compromised if multiple errors collude in such a way as to compromise all

the checks being performed by the assertion. This phenomenon is called *aliasing* and has been addressed in [27] in which it is shown that the probability of random errors cancelling each other out is extremely small for any practical assertion. The model under which this has been proved encompasses errors that can occur due to both processor and link failures. The advantage of using assertions over full duplication is that, usually, assertions are much less expensive than duplications. Readers can refer to [14] for a more detailed discussion of the concept of assertions.

## 2.5 Exclusion and Preference Compatibility

A cluster is a collection of tasks typically grouped together because they communicate heavily with one another. The idea behind forming clusters is to reduce IPC by allocating all the tasks of a cluster to the same processor. The number of clusters can be an order of magnitude smaller than the number of tasks in the system which may drastically reduce the size of the search space to be explored to find a good allocation. The presence of task exclusion and preference places restrictions on the way tasks can be grouped together into a cluster, as will be explained shortly.

Each cluster in the system has a label which gives the *execution cost* of the cluster. This is the sum of the *comp\_costs* of all the tasks in the cluster. The clusters and the dependencies among them can be represented by a *cluster graph*. This graph is obtained from the task graph by creating weighted edges between communicating clusters. The weights on the edges indicate the total volume of data being transferred from tasks in one cluster to tasks in the other. Each cluster and task has an *exclusion vector* and a *preference vector* attached to it. Exclusion vector  $excl(t)$  of task  $t$  is the row corresponding to  $t$  in the exclusion matrix. This vector stores the information regarding which tasks cannot be allocated to the same processor as  $t$ . The exclusion vector,  $excl(c)$ , of a cluster  $c$  is the logical OR of the exclusion vectors of all the tasks in the cluster. The preference vector  $pref(t)$  of task  $t$  is the row corresponding to task  $t$  in the preference matrix. The preference vector  $pref(c)$  of cluster  $c$  is obtained by taking the logical AND of the preference vectors of the component tasks. The exclusion and preference vectors of a cluster represent the combined effect of the exclusion and preference vectors of the component tasks and are used to evaluate whether the cluster can be allocated to a particular processor. The binary nature of the exclusion and preference vectors and the use of logical operators for manipulating them enables a very efficient implementation (for example using a packed array of bits, otherwise known as sets, cf. Pascal). Note that a preference vector of all zeros is not allowed as this precludes the task or cluster from being allocated to any processor.

**Definition 1.** Task  $t_1$  is said to be exclusion compatible with cluster  $c$  (task  $t_2$ ) if the exclusion vector of  $c$  ( $t_2$ ) does not have a "1" entry in the column corresponding to  $t_1$ .

**Definition 2.** Task  $t_1$  is said to be preference compatible with cluster  $c$  (task  $t_2$ ) if, when  $t_1$  is added to  $c$  ( $t_2$ ), the resulting preference vector does not have all "0" entries.

**Definition 3.** Task  $t_1$  is said to be fully compatible with cluster  $c$  (task  $t_2$ ) if it is both exclusion compatible and preference compatible with  $c$  ( $t_2$ ).

Definitions 1, 2, and 3 can be applied to determine the compatibility of two clusters, too. If all the tasks in cluster  $c_1$  are fully compatible with cluster  $c_2$  or vice versa, then clusters  $c_1$  and  $c_2$  are said to be fully compatible.

## 2.6 Problem Definition

We are given a job in the form of a task graph, the target architecture in the form of a processor graph, the exclusion and preference matrices and the fault tolerance information. Our objective is to find a *safe mapping* of the tasks to the processors of the system such that the overall reliability of the allocation is maximized. Since the allocation problem is known to be NP-complete, our aim is to devise good heuristics for the problem. We will explore several heuristics to find the approach yielding the best results.

## 3 ALLOCATION FOR RELIABILITY

Now we show how allocation can be done to meet our desired objective.

### 3.1 Clustering

The first step of our allocation algorithm, called *clustering*, groups tasks of the task graph into *clusters* and forms the *cluster graph*. Our method uses techniques from linear clustering [1], [2]. For the clusters to be valid, the tasks in each cluster must be fully compatible with all other tasks in the cluster. The purpose of forming clusters is to reduce IPC as much as possible. However, the formation of clusters that are "too big" (in terms of their total execution cost) inhibits parallelism and has the effect of reducing the available degrees of freedom for the allocation stage of the algorithm, which affects the quality of the results. On the other hand, the clusters should not be too small as this results in an overly fragmented task graph which makes allocation expensive. Also, as will be evident from our simulation results, excessive fragmentation of the task graph also affects the quality of the results. Thus, it is very important to *size* the clusters appropriately.

Our clustering algorithm attempts to do three things simultaneously.

1. It tries to group together heavily communicating tasks in a single cluster since all the tasks in the cluster will end up being allocated to the same processor, thus reducing IPC which, as noted earlier in Section 2.3, increases the reliability.
2. In grouping tasks together, it makes sure that all the tasks in a cluster are fully compatible with each other.
3. It tries to size the clusters as they are being formed such that they are neither too large nor too small.

The first step of the clustering algorithm is to assign static levels to all the tasks. We define the static level of task  $t_i$ ,  $SL(t_i)$ , as [14]:

```

FORM_CLUSTERS() {
  ASSIGN_TASK_STATIC_LEVELS();
  sorted_list ← SORT_TASKS_BY DECREASING_STATIC_LEVELS();
  threshold ← (total average execution time of all tasks per processor);
  for each task  $t$  {  $t\_tag$  ← (NOT_CLUSTERED); }
  cluster_list ←  $\emptyset$ ;
  for each unclustered task  $t$  from sorted_list {
    allocate a new cluster  $c$ ;
    cluster_list ← cluster_list  $\cup$   $\{c\}$ ;
    CLUSTER_GROW( $c, t, threshold$ );
  }
}
CLUSTER_GROW(cluster  $c, task t, threshold$ ) {
  add  $t$  to  $c$ ;
   $t\_tag$  ← (CLUSTERED);
   $t_n$  ← CLUSTER_NEXT_FEASIBLE_TASK( $c, t, threshold$ );
  if ( $t_n \neq NIL$ ) CLUSTER_GROW( $c, t_n, threshold$ );
}
CLUSTER_NEXT_FEASIBLE_TASK(cluster  $c, task t, threshold$ ) {
  best_cost ← 0.0;
  best_task ← NIL;
  for each edge  $e$  corresponding to an unclustered fanout task  $t_f$  of  $t$  {
    if ( $t_f$  is fully compatible with  $c$ ) {
      if ( $t_f\_static\_level + 2 \times e\_comm > best\_cost$ ) &&
        ( $c\_cost + t_f\_comp\_cost \leq threshold$ ) {
        best_task ←  $t_f$ ;
        best_cost ← ( $t_f\_static\_level + 2 \times e\_comm$ );
      }
    }
  }
  return best_task;
}

```

Fig. 2. One-pass clustering algorithm.

$$SL(t_i) = comp\_cost(t_i) + \max_{t_j \in \{fanout\ of\ t_i\}} (SL(t_j) + 2 * comm(t_i, t_j)).$$

The static level of a sink task is just its comp\_cost. The static levels of all the tasks is calculated by assign\_task\_static\_levels() by first assigning static levels to all sink tasks and then applying this equation recursively for all the other tasks. The factor of "2" is included in the above equation because the total communication delay in sending data from  $t_i$  to  $t_j$  which are on different processors is the sum of the delay in sending the data from the processor computing task  $t_i$  and the delay in receiving it by the processor computing  $t_j$ .

Initially, all unclustered tasks are marked NOT\_CLUSTERED. The list of clusters is initialized to an empty set. Next, the tasks are sorted in the decreasing order of static levels and then considered for clustering in that order. Let  $t$  be the first task to be considered. A new cluster  $c$  is formed, added to the list of clusters and  $t$  is added to  $c$  and marked CLUSTERED. Next, the algorithm attempts to grow  $c$  along that fanout  $t_f$  of  $t$  that is fully compatible with all the tasks of  $c$  (just  $t$  at this point) and which has the highest static level among all the fanouts of  $t$ . Task  $t_f$  is then added to  $c$  and marked CLUSTERED and the algorithm next attempts to

grow the cluster along one of the fanouts of  $t_f$ . This process is continued till either the cluster can be grown no more or till the addition of any more tasks to the cluster makes cluster size (sum of the average execution costs of component tasks) exceed a threshold (so that the cluster is appropriately sized). We have chosen the total average computation of all tasks divided by the number of processors to be the threshold. This threshold ensures that no cluster has a size greater than the average load per processor in the system. There is room for tuning the value of the threshold depending on the characteristics of the task graph. At this point, a fresh cluster is created and the process repeated with the next unclustered task. At the end of the process, the whole task graph has been grouped into clusters. Then, the cluster graph is formed by creating edges between communicating clusters. The weight on the edge between clusters  $c_1$  and  $c_2$  is the total volume of communication from tasks in  $c_1$  to tasks in  $c_2$ .

Note that this clustering algorithm is very similar to the three-pass technique presented in [14] with the three passes consolidated into a single one. The detailed pseudocode of the clustering algorithm is given in Fig. 2.

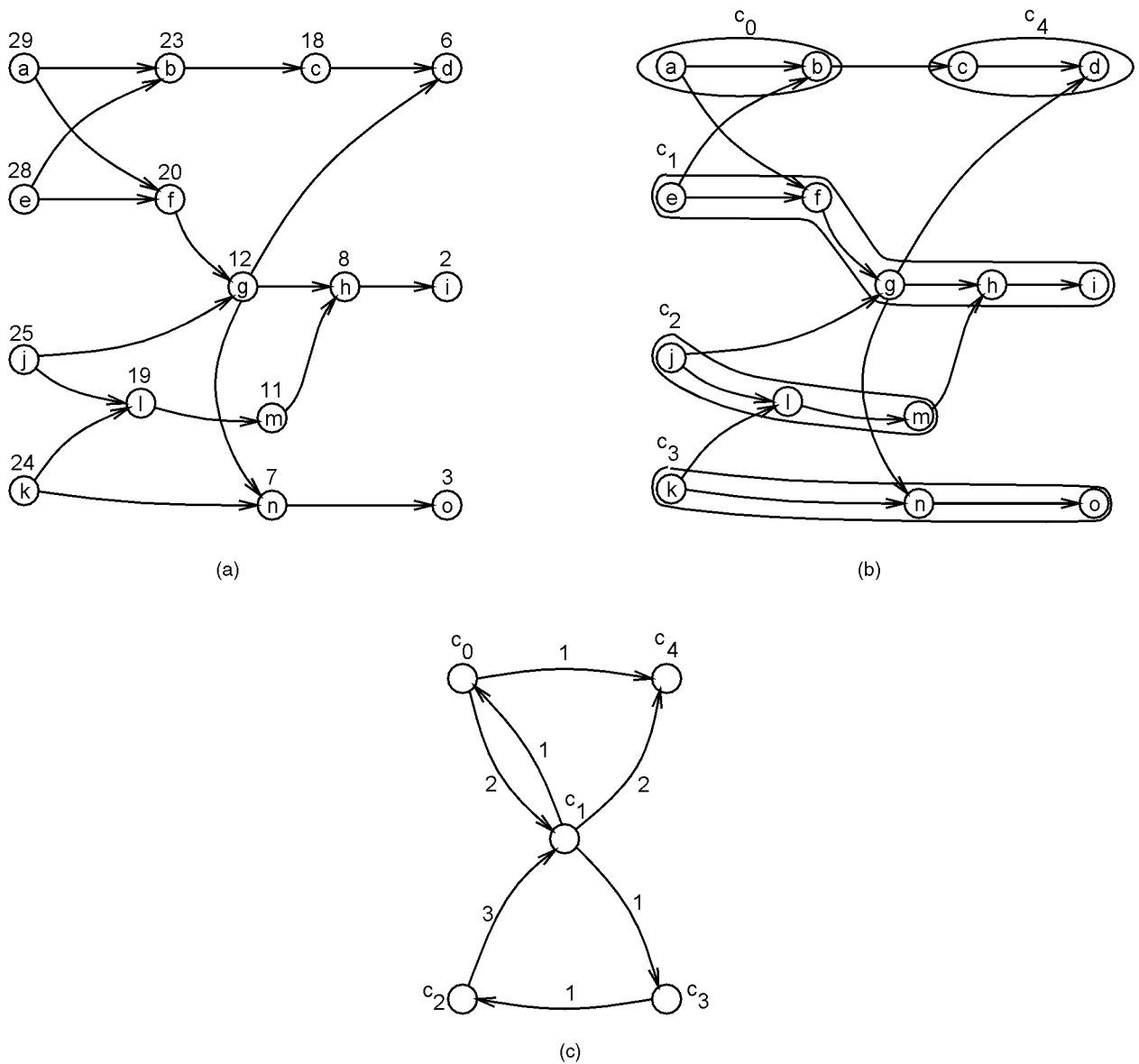


Fig. 3. (a) Static level assignment, (b) clustering, (c) resultant cluster graph.

Fig. 3 shows the result of assigning static levels to the example shown earlier in Fig. 1 performing clustering using our clustering algorithm and the resulting cluster graph. The sum of the average execution costs of all the tasks in this example is 49. Averaging this over three processors, we get a threshold of 16.33.

### 3.2 Allocation of Original Tasks

After the clustering stage, the task graph has been partitioned into clusters. The next step is to allocate the clusters to the processors in the system. Our objective for allocation is to minimize the *relcost*. In this section, we present a new cluster-based allocation strategy to maximize reliability. Task-level allocation strategies for maximizing reliability have been proposed in [19]. We first give our allocation method and then the best method from [19]. In Section 6, we compare the performance of these two methods.

#### 3.2.1 Cluster Allocation

In order to represent the combined effect of the exclusion vectors of all the tasks allocated to a processor, each processor in the system also has an exclusion vector associated with it, which is the logical OR of the exclusion vectors of all the tasks allocated to this processor. We initialize the exclusion vector of all processors to all "0"s. Before a fresh cluster is allocated to a processor, we have to verify whether the processor and cluster are exclusion and preference compatible, i.e., to verify that 1) no task in the cluster excludes any task already allocated to the processor, and 2) all tasks in the cluster are preference compatible with the processor. A processor is not exclusion compatible with a cluster if the exclusion vector of the processor has a "1" entry for some task which belongs to the cluster or vice versa. Similarly, a processor is not preference compatible with a cluster if the cluster's preference vector has a "0" in the column corresponding to that processor.

First, all the clusters containing tasks with specific preference are allocated. For this, the preference vectors of the clusters are checked. If any cluster  $c_i$  has a "1" in some single position  $j$ , then that cluster has to be allocated to processor  $p_j$ . It might, however, happen that although preference is forcing a cluster to go to some processor, exclusion is preventing it. This means that some task in the cluster is required to be mapped to a particular processor with which some other task is exclusion incompatible. If this happens, the cluster is broken up into smaller clusters such that the set of tasks contributing the "1" to the preference vector are all exclusion compatible with that processor. Assume that the tasks forming the cluster are  $t_1, \dots, t_k$ , where there is a fanin from task  $t_i$  to task  $t_{i+1}$ ,  $i \in \{1, \dots, k-1\}$ . A new cluster  $c_i'$  is initialized with task  $t_1$ . Then each task in  $\{t_2, \dots, t_k\}$  is examined in that order. If with the inclusion of the next task in  $c_i'$ , there exists a processor  $p$  with which  $c_i'$  is fully compatible, then the new task is deleted from cluster  $c_i$  and added to cluster  $c_i'$ . This process of adding tasks to  $c_i'$  is continued until the above condition continues to be met. Then, the part of  $c_i$  that remains is treated in the same fashion. Now, the clusters can be mapped to the appropriate processors. After any cluster is mapped to a processor, the exclusion vector of the processor is updated.

Next, the remaining clusters have to be allocated. The clusters are sorted in the decreasing order of their *total cost*, which is the sum of their execution and communication costs, and considered for allocation in that order.

The cluster allocation algorithm is a function of a user-specified parameter, which we shall denote by  $k_a$ . We refer to this as the *order* of the algorithm. The allocation algorithm keeps track of the *relcost* for the allocated clusters using (1). After any cluster is allocated, this is updated as follows. Suppose cluster  $c_u$  is allocated to processor  $p_v$  then the increase in *relcost*,  $\Delta relcost(c_u, p_v)$ , can be calculated in the following manner. The increase in *relcost*,  $\Delta relcost(t_i, p_v)$ , when a task  $t_i$  is allocated to  $p_v$  is

$$\Delta relcost(t_i, p_v) = e_{iv} \times \lambda_v + \sum_{t_j} 2 \times \mu_{vw} \times c_{ij} \times w_{vw}, \quad (2)$$

where  $t_j$  is a fanout of  $t_i$  such that  $t_j$  has been previously allocated to a processor  $p_w$  distinct from  $p_v$ . Now we can write

$$\Delta relcost(c_u, p_v) = \sum_{t_i \in c_u} \Delta relcost(t_i, p_v). \quad (3)$$

At any time, the algorithm considers a group of  $k_a$  clusters and tries to find the best allocation for them. This is done by considering all possibilities for allocating the  $k_a$  clusters and for each choice, marking the clusters as temporarily allocated and then computing the increase in *relcost*. Then, that allocation which results in the least increase in *relcost* (i.e., least decrease in reliability) is chosen. Thereafter, the algorithm moves on to the next set of  $k_a$  clusters. As soon as a cluster  $c$  is allocated to a processor  $p$ , all tasks in  $c$ , processor  $p$  and all the (relevant) clusters in the system are updated as follows. The execution cost of all the tasks in  $c$  is set to their execution cost on  $p$  and the total cost of  $c$  is updated. The weights on the edges of the cluster

graph are modified such that edges between clusters allocated to the same processor have a weight of 0 and the edge between clusters  $c_u$  and  $c_v$  allocated to distinct processors  $p_i$  and  $p_j$  has a weight equal to the product of  $w_{ij}$  and the volume of communication between  $c_u$  and  $c_v$ . The execution costs of all tasks in  $c$  are added to the *bin cost* of  $p$  (i.e., sum of the execution and communication costs of all the tasks allocated to  $p$ ). When  $k_a$  is set to the number of clusters, the algorithm explores all combinations of clusters and processors, and when  $k_a$  is set to "1," it is a greedy algorithm with a lookahead of just "1." Thus,  $k_a$  controls the fraction of the search space that the algorithm explores.

The pseudocode of the allocation algorithm is given in Fig. 4. The following result establishes the complexity of the allocation algorithm. It is provable through a simple counting argument.

**Lemma 1.** *If the number of clusters is  $n_c$ , the maximum number of tasks in a cluster is  $T_c$ , the maximum fanout of a cluster is  $F$ , and the number of processors is  $p$ , the complexity of cluster allocation is  $O\left((T_c \times F \times p)^{k_a} \times n_g\right)$ , where  $n_g = \left\lceil \frac{n_c}{k_a} \right\rceil$  denotes the number of cluster groups.*

### 3.2.2 Task Level Allocation Scheme

In [19], several task-level allocation schemes were proposed for maximizing reliability. We shall be using the method that performed the best (in our experiments), for comparison with our cluster-level allocation scheme.

We have to note that the notion of task exclusion and preference is not used in [19]. Also, the task graphs considered do not have any precedence constraints. For our experiments, we have adapted the spirit of this algorithm to accommodate these differences. The adapted allocation scheme works as follows. First, the tasks with specific preferences are allocated. The remaining tasks are sorted in the decreasing order of their *communication cost* (as compared to the total cluster costs used in our algorithm) and are considered for allocation in that order. For each possible processor that the first task can be allocated to, the remaining tasks are allocated in a single-step greedy process, that is, to that processor that leads to the minimum increase in *relcost*. Equation (2) can be used to compute the incremental increase in *relcost*. Of each of these possible allocations, the one that yields the lowest *relcost* (highest reliability) is chosen.

In order to be fair in the comparison of our  $k_a$ -step algorithm with this one, we next need to introduce the *order* concept in this method too. This can be done quite naturally by taking all combinations of processors for the first  $k_a$  tasks, allocating the rest in a single-step greedy process, and then picking the best solution.

## 4 SAFETY

After the cluster allocation stage, the reliability of the allocation has been maximized. However, no guarantees can be given about system behavior when a failure occurs. As remarked earlier, the allocation is unsafe. The next step is to add safety to the allocation. We apply the principle of task-based fault tolerance [14] to achieve this.

```

ALLOCATE_CLUSTERS( $k_a$ ) {
  for each cluster  $c$  with a single 1 in the preference vector {
     $p \leftarrow$  processor  $c$  has to be allocated to;
    if  $c$  is exclusion compatible with  $p$  then
      allocate  $c$  to  $p$  and update costs of  $p$ , tasks in  $c$  and relevant clusters;
    else break  $c$  and map the sub-clusters with specific preference;
  }
  while there are more unallocated clusters {
     $r \leftarrow$  number of unallocated clusters;
     $c\_list \leftarrow$  list of the next  $\min(r, k_a)$  unallocated clusters
      considered in the decreasing order of total cost;
    BEST_ALLOCATION( $c\_list, p\_list, best\_p\_list, \infty, 0, \min(r, k_a)$ );
    for each cluster  $c$  in  $c\_list$  {
      allocate  $c$  to corresponding processor in  $best\_p\_list$ ;
    }
    update costs of all relevant clusters and processors;
  }
}
BEST_ALLOCATION( $c\_list, p\_list, best\_p\_list, best\_cost, index, order$ ) {
  if ( $index < order$ ) {
     $c \leftarrow c\_list[index]$ ;
    for each possible processor  $p$  to which  $c$  can be allocated to {
       $p\_list[index] \leftarrow p$ ;
      mark  $c$  to be temporarily allocated to  $p$ ;
      BEST_ALLOCATION( $c\_list, p\_list, best\_p\_list, best\_cost, index+1, order$ );
      mark  $c$  to be unallocated;
    }
  }
  else {
     $cost \leftarrow$  increase in  $relcost$  due to allocating clusters
      in  $c\_list$  to processors in  $p\_list$ ;
    if ( $cost < best\_cost$ ) {
       $best\_cost \leftarrow cost$ ;
       $best\_p\_list \leftarrow p\_list$ ;
    }
  }
}

```

Fig. 4. Cluster allocation to maximize reliability.

Each task has five types of fault-tolerant information associated with it [14].

1. Is the task assertible, i.e., does an assertion particular to the task exist?
2. Exec\_cost vector of the assertion task, if the task is assertible.
3. Is the assertion *costly*, i.e., does it require as inputs all the inputs of the task it is checking?
4. Volume of data being communicated from the task to the assertion task.
5. Exec\_cost of the comparison task in the case when duplication is done.

Once the allocation of the original tasks is done, the task graph is taken through two fault-tolerant stages. The first stage is called *ft\_task creation*, in which the nodes required for fault tolerance are added to the task graph. The next

stage is *ft\_task allocation*, in which these new nodes are allocated to the processors under the fault tolerance constraints.

**Ft\_task Creation:** This step is described in detail in [14]. We shall briefly describe the concepts here for completeness. In this step, an assertion node is added to the task graph for each task that is assertible. The assertion node has to be allocated to a processor different from the original task. Hence, the assertion is made preference incompatible with the processor to which the original task has been allocated. If the task is costly, all the fanins of the original task are duplicated as fanins of the assertion. Next, corresponding to each nonassertible task, a duplicate and a comparison task are created. The duplicate is made exclusion incompatible with the original in order to force it to be allocated to a different processor from the original. The comparison node has as its fanins the outputs from the

```

ADD_SAFETY( $k_b$ ) {
  ft_task_list  $\leftarrow$  FT_TASK_CREATE();
  SORT_BY DECREASING TOTAL COST(ft_task_list);
  while there are unallocated tasks in ft_task_list {
     $r \leftarrow$  (number of unallocated tasks in ft_task_list);
    copy min( $r, k_b$ ) tasks from ft_task_list into t_list;
    mark all tasks in t_list to be unallocated;
    BEST_FT_ALLOCATION(t_list, p_list, best_p_list,  $\infty$ , 0, min( $r, k_b$ ));
    allocate each task of t_list to the corresponding processor from best_p_list;
  }
}
BEST_FT_ALLOCATION(t_list, p_list, best_p_list, best_cost, index, order) {
  if (index < order) {
     $t \leftarrow$  t_list[index];
    for each processor  $p$  in the system {
      if ( $t$  is fully compatible with  $p$ ) {
        p_list[index]  $\leftarrow$   $p$ ;
        mark  $t$  as being temporarily allocated to  $p$ ;
        BEST_FT_ALLOCATION(t_list, p_list, best_p_list, best_cost,
          index + 1, order);
        mark  $t$  as unallocated;
      }
    }
  }
  else {
    cost  $\leftarrow$  (increase in relcost for the current allocation of t_list);
    if cost < best_cost {
      best_p_list  $\leftarrow$  p_list;
      best_cost  $\leftarrow$  cost;
    }
  }
}

```

Fig. 5. Adding safety to the system.

original and the duplicate. The comparison task cannot be on the same processor as the original but may be mapped to the same processor as the duplicate (since the duplicate's output does not fan out to any other task). Therefore, the comparison task is made preference incompatible with the processor executing the original.

**Ft\_task Allocation for Safety to Maximize Reliability:** After all the extra tasks needed to introduce safety have been created, the next step is to allocate them so that the overall increase in *relcost* is minimized in order to maximize the final reliability. We shall present a task-level heuristic which, for this purpose, performs as well, in terms of the quality of the results, as the task-level heuristic from [19] described in Section 3.2.2, but runs an order of magnitude faster.

Our task-level heuristic is similar to the "order- $k_a$ " cluster allocation method and has an "order" parameter which we denote by  $k_b$ . The extra tasks added by the Ft\_task\_creation stage are sorted in the decreasing order of their total costs. The parameter  $k_b$  is a user-specified entity (the value of  $k_b$  used in this step may be distinct from the value of  $k_a$  used in the cluster allocation stage). The tasks are now considered in batches of at most  $k_b$  at a time. For

these  $k_b$  tasks, from all the possible choices of processors to allocate, the allocation that leads to the least increase in *relcost* is chosen. Then the next set of  $k_b$  tasks is considered, and so on.

The increase in *relcost*,  $\Delta relcost(t_i, p_v)$ , when task  $t_i$  is allocated to processor  $p_v$ , can be computed using (2).

The pseudocode for the algorithm is given in Fig. 5.

## 5 SCHEDULING

Once all the tasks have been allocated, the tasks and the inter-task communications are scheduled. Our scheduling algorithm is based on static levels. The tasks are sorted in the decreasing order of their static levels and considered for scheduling in that order. Each task is scheduled to the earliest available time slot on the processor to which it has been allocated. Next, the fanout edges of the task are sorted in the decreasing order of the static level of the fanout tasks and the *send* and *receive* corresponding to each fanout are then scheduled in the earliest available back-to-back time slots on the interprocessor link. We thus compute an "as-soon-as-possible" schedule. The details of this scheduling algorithm are described in [14].

## 6 EXPERIMENTAL RESULTS

In this section, we describe the performance of our algorithms on a large number of examples including a real-life digital signal processing (DSP) example.

### 6.1 Performance on Synthetic Workloads

Due to a lack of benchmarks for validating our algorithm, we have tested it on a variety of randomly generated task and processor graphs. Random graphs have been used by several researchers in the past (see, for example, [12], [19]). We have tried to choose random workloads with characteristics similar to available real-life examples.

#### 6.1.1 Generation of Random Examples

We chose several different types of task and processor graphs with structures which occur commonly in a variety of real-life algorithms. The chosen types of task graphs were as follows:

1. binary trees (BTREE)
2. lattices (LATTICE)
3. randomly interconnected chains (CHAIN)
4. server-client formations (SERVERCLIENT)

Fig. 6 shows task graphs of each type.

Each example had 100 tasks in the task graph and 10 processors in the processor graph. The processor graph connectivity (the probability of a given edge in the graph) was chosen to be 0.9. Fifty percent of the tasks were chosen to be assertible (at random) in each task graph with 10 percent of these assertions being costly. The execution cost of the tasks was chosen uniformly between [5, 195] units. The communication weights for the tasks were taken uniformly between [1, 10] units. The link failure rates were chosen uniformly from the range  $7.5 \times 10^{-6}$  to  $12.5 \times 10^{-6}$  per hour and the processor failure rates between  $0.95 \times 10^{-6}$  to  $1.05 \times 10^{-6}$  per hour. Each data point is an averaged value over 250 random graphs.

#### 6.1.2 Performance on Random Examples

As explained in previous sections, there are two stages in the allocation algorithm. The first stage allocates the original tasks with the objective of maximizing the reliability. Our algorithm is cluster-based and has an "order" parameter,  $k_a$ . We compare the performance of this algorithm with the best task-level allocation algorithm from [19], as presented in Section 3.2.2. In order to have a fair comparison, we have introduced the "order" parameter in this algorithm as well, as a means of forcing the algorithm to search through more or less of the search space. Our first set of experiments compare the performance of these two algorithms with respect to the reliability achieved and the amount of CPU time<sup>1</sup> consumed in searching for the solution. In the tables, for brevity, we shall refer to the task-level algorithm for allocating the original tasks as T0 and our cluster-based algorithm for this purpose as T1.

The second stage allocates the tasks needed to introduce safety. We compare the performance of the modified

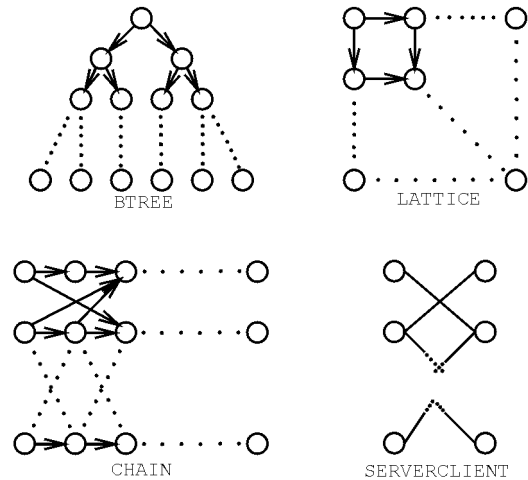


Fig. 6. Different types of task graphs.

version, with "order"  $k_b$ , of the task-level allocation algorithm from [19] (denoted by F0) with the performance of our "order"  $k_b$  task-level allocation algorithm (denoted by F1). Note that increasing  $k_b$  increases the complexity of both methods but in different ways.

In each case, we present the results for the four different types of workloads and for different values of  $k_a$  and  $k_b$ . Our objective is to determine that combination of T0, T1, F0, and F1 and the choice of values for  $k_a$  and  $k_b$  which yields the best results in terms of reliability and is most cost-effective in terms of CPU time.

**Comparison of cluster-based and task-based allocation schemes:** In Table 1, the results of the simulation runs comparing T0 and T1 are presented. Simulations were conducted for  $k_a$  set to 1 and 2 for both methods. In the table, T1 with  $k_a$  set to 2 is chosen as the base case. The "% extra" columns show the percent unreliability greater than the base case that is obtained by the particular method, for the particular type of workload. Also presented is the time taken in seconds by each method to complete the allocation of the original tasks.

As can be seen, T1 with  $k_a$  set to 1 performs consistently better than T0, for both  $k_a$  set to 1 and to 2. Also, T1 with  $k_a$  equal to 1 takes about the same amount of CPU time as T0. T0 shows a slight improvement with respect to itself on increasing  $k_a$  from 1 to 2. T1 shows hardly any improvement on increasing  $k_a$  and takes a lot more time to complete. (The results for both T0 and T1 saturate for values of  $k_a$  greater than 2, which is not shown in these tables.)

From this table, we can conclude that T1 with  $k_a$  equal to 1 is the most cost-effective in terms of minimizing the unreliability and consumes about the same amount of CPU time as T0,  $k_a = 1$ , in searching for a solution.

**Comparison of F0 and F1 for allocating extra tasks:** Tables 2 and 3 compare the performance of our algorithm for allocating the extra tasks and the algorithm presented in [19], adapted to fit our assumptions and scenario, in order to make a fair comparison.

Having decided that increasing  $k_a$  beyond 1 does not help much in decreasing the unreliability, we have set  $k_a$  to 1 for this step, for both T0 and T1. Choosing T1 with  $k_a$

1. All CPU times in this paper were measured on a SPARCstation-1 with a 16MB RAM.

TABLE 1  
Comparison Between T0 and T1

	$k_a = 1$				$k_a = 2$			
	T0		T1		T0		T1	
	% extra	CPU time (seconds)	% extra	CPU time (seconds)	% extra	CPU time (seconds)	% extra	CPU time (seconds)
LATTICE	22	6.1	1	6.8	20	10.9	0	16.2
BTREE	7	5.7	0	6.1	6	8.8	0	23.7
CHAIN	8	3.5	0	3.7	5	5.2	0	6.0
SERVER-CLIENT	4	4.9	0	5.1	4	6.7	0	11.3

TABLE 2  
Comparison of % Overhead in Unreliability After Addition of Safety

	$k_b = 1 (k_a = 1)$				$k_b = 2 (k_a = 1)$			
	T0 + F0	T0 + F1	T1 + F0	T1 + F1	T0 + F0	T0 + F1	T1 + F0	T1 + F1
LATTICE	82	89	60	61	75	81	56	56
BTREE	57	61	52	52	54	58	48	48
CHAIN	57	60	51	51	52	55	46	46
SERVER-CLIENT	53	58	49	49	51	56	44	44

equal to 1 as the base case, Table 2 shows the % increase in unreliability obtained by the different combinations of T0 and T1 for the allocation of the original tasks, and F0 and F1 for allocating the extra tasks. Our objective, of course, is to determine that method which leads to the least increase in unreliability.

As can be seen from the table, unlike  $k_a$ , increasing  $k_b$  from 1 to 2 does decrease the unreliability for both F0 and F1. (Increasing it further does not help; this is not shown in the table.) Also, although F0 does a little better than F1 for  $k_b$  equal to 1, both produce solutions of the same quality when  $k_b$  is increased to 2.

Table 3 shows the number of CPU seconds consumed by both the methods for both values of  $k_b$ . The CPU time taken by T0 and T1 is reproduced from Table 1 for comparison. As can be seen, when  $k_b$  is 1, F1 runs in less than one fifth the time taken by F0. It is an order of magnitude faster when  $k_b$  is increased to 2.

From these results, we can conclude that the combination of T1 with  $k_a$  equal to 1 for allocating the original tasks and F1 with  $k_b$  equal to 2 for introducing safety form the best combination, in achieving the most reliable allocation and in requiring the fewest number of CPU seconds in doing so. Also, using this combination of algorithms, we see that the average increase in unreliability due to the addition of safety is around 48.4 percent for the different types of workloads. The reliabilities of the system before and after the addition of safety were 0.99901 and 0.99846 for LATTICE, 0.99906 and 0.99863 for BTREE, 0.99905 and

0.99859 for CHAIN, and 0.99902 and 0.99859 for SERVER-CLIENT. Clearly, this small decrease in reliability is not too high a price to pay for the addition of safety. These experimental results validate the use of our technique to add safety.

## 6.2 Performance on a DSP Example

To show the performance of our scheme for reliable and safe allocation, we ran our algorithm on a sizeable DSP example from [5], which gives the details of the task graph for the example (there are 119 tasks in this task graph). About 33 percent of the tasks were assertible and none of the assertible tasks were costly.

We generated a number of heterogeneous processor graphs with seven fully connected processors, with processor failure rates chosen randomly between  $3.2 \times 10^{-3}$  and  $4.0 \times 10^{-3}$  per hour and link failure rates between  $1.8 \times 10^{-2}$  and  $5.4 \times 10^{-2}$  per hour. We performed our experiments with T1 ( $k_a = 1$ ) for allocating the original tasks and F1 ( $k_b = 2$ ) for allocating the tasks required to add safety. On an average, the reliability before safety was added was 0.999988 and the reliability after the addition of safety was 0.999969. This is a very marginal decrease in reliability which is not much of a price to pay for the addition of safety. The cluster allocation stage of the algorithm took 7.9 seconds of CPU time and the addition of safety took an extra 2.6 seconds.

TABLE 3  
CPU Seconds Needed to Add Safety

	Original Tasks		Addition of Safety			
	$k_a = 1$		$k_b = 1$		$k_b = 2$	
	T0	T1	F0	F1	F0	F1
LATTICE	6.1	6.8	5.7	1.0	46.2	4.0
BTREE	5.7	6.1	5.8	0.9	45.2	3.9
CHAIN	3.5	3.7	3.6	0.6	27.7	2.5
SERVER-CLIENT	4.9	5.1	5.1	1.0	49.3	4.1

## 7 CONCLUSIONS

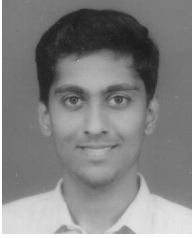
In the past, methods have been presented for task allocation to maximize reliability. However, such schemes do not give any guarantees about the system behavior when a failure occurs. This is a scenario for potentially undetected faults and catastrophe. In this work, we have addressed the problem of introducing safety into the system during task allocation. We have presented a new cluster-based allocation technique to maximize reliability and we have shown that it performs better than previously known heuristics for this purpose. Then, using the concept of task-based fault tolerance, we have devised a technique to introduce safety. We have presented a new task-level allocation scheme for allocating the extra tasks needed for this purpose. This method produces solutions of quality equal to the best known reliability driven task-level allocation heuristic and, for our purposes, runs an order of magnitude faster. Our experimental results show that the decrease in reliability due to the introduction of safety is very small, which proves the efficacy of our scheme.

## ACKNOWLEDGMENTS

This work was supported in part by the U.S. Office of Naval Research under Contract no. N00014-91-J-1199 and in part by the U.S. National Science Foundation under Grant no. MIP-9423574. This work was done when Santhanam Srinivasan was at Princeton University.

## REFERENCES

- [1] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing*, vol. 3, pp. 1-8, University Park, Penn., Aug. 1988.
- [2] G.C. Sih and E.A. Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 6, pp. 625-637, June 1993.
- [3] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175-186, Feb. 1993.
- [4] I. Page, T. Jacob, and E. Chern, "Fast Algorithms for Distributed Resource Allocation," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 188-197, Feb. 1993.
- [5] C.M. Woodside and G.G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 164-174, Feb. 1993.
- [6] Y.C. Chow and W.H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Trans. Computers*, vol. 28, no. 5, pp. 354-361, May 1979.
- [7] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel and Distributed Computing*, vol. 7, pp. 279-301, 1989.
- [8] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Eng.*, vol. 3, no. 1, pp. 85-93, Jan. 1977.
- [9] P.Y.R. Ma, E.Y.S. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. Computers*, vol. 31, no. 1, pp. 41-47, Jan. 1982.
- [10] W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, pp. 57-69, Nov. 1980.
- [11] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, pp. 50-56, June 1982.
- [12] V.M. Lo, "Heuristic Algorithms for Task Assignments in Distributed Systems," *IEEE Trans. Computers*, vol. 37, no. 11, pp. 1,384-1,397, Nov. 1988.
- [13] N.S. Bowen, C.N. Nikolaou, and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans. Computers*, vol. 41, no. 3, Mar. 1992.
- [14] S. Jaynik, S. Srinivasan, and N.K. Jha, "TBFT: A Task-Based Fault Tolerance Scheme for Distributed Systems," *Proc. Int'l Conf. Parallel and Distributed Computer Systems*, Las Vegas, Nev., Oct. 1994.
- [15] W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiple Processor Systems," *IEEE Trans. Computers*, vol. 24, no. 12, pp. 1,235-1,238, Dec. 1975.
- [16] S. Tridandapani and A.K. Somani, "Efficient Utilization of Spare Capacity for Fault Detection and Location in Multiprocessor Systems," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 440-447, Montreal, July 1992.
- [17] N.K. Jha and S. Kundu, *Testing and Reliable Design of CMOS Circuits*. Norwell, Mass.: Kluwer Academic, 1990.
- [18] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*. Reading, Mass.: Addison-Wesley, 1989.
- [19] S.M. Shatz, J.P. Wang, and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," *IEEE Trans. Computers*, vol. 41, no. 9, pp. 1,156-1,168, Sept. 1992.
- [20] C.J. Hou and K.G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.*, pp. 146-155, Dec. 1992.
- [21] N.H. Vaidya and D.K. Pradhan, "Fault-Tolerant Design Strategies for Reliability and Safety," *IEEE Trans. Computers*, vol. 42, no. 10, pp. 1,195-1,206, Oct. 1993.
- [22] M. Mullazani, "Reliability and Safety," *Proc. Fourth IFAC Workshop Safety Comput. Contr. Systems*, pp. 141-146, 1985.
- [23] B.W. Johnson and J.H. Aylor, "Reliability and Safety Analysis of a Fault-Tolerant Controller," *IEEE Trans. Reliability*, vol. 35, no. 10, pp. 355-362, Oct. 1986.
- [24] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers*, vol. 33, no. 6, pp. 518-528, June 1984.
- [25] A.L.N. Reddy and P. Banerjee, "Algorithm-Based Fault Detection for Signal Processing Applications," *IEEE Trans. Computers*, vol. 39, no. 10, pp. 1,304-1,308, Oct. 1990.
- [26] Y.H. Choi and M. Malek, "A Fault-Tolerant Systolic Sorter," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 621-624, May 1988.
- [27] S. Srinivasan and N.K. Jha, "Efficient Diagnosis in Algorithm-Based Fault Tolerant Multiprocessor Systems," *Proc. Int'l Conf. Computer Design*, pp. 592-595, Boston, Oct. 1993.



**Santhanam Srinivasan** (S'91-M'95) received the BTech degree in electronics and communications engineering from the Indian Institute of Technology, Madras, India, in 1991, and the MA and PhD degrees in computer engineering from Princeton University, Princeton, New Jersey, in 1993 and 1995, respectively. He joined Lucent Bell Labs, Holmdel, New Jersey, in 1995, where he is currently with the High Speed Networks Research Department. His research interests

include network management of QOS capable IP networks, ATM networks, and Policy-Based Networking. He is also a software architect in Lucent's PacketStar<sup>TM</sup> IP Switch group. Dr. Srinivasan is a program committee member of the International Symposium on Integrated Network Management, 1999.



**Niraj K. Jha** (S'85-M'85-SM'93-F'98) received his BTech degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India in 1981, the MS degree in electrical engineering from S.U.N.Y. at Stony Brook, New York in 1982, and a PhD degree in electrical engineering from the University of Illinois, Urbana, in 1985. He is a professor of electrical engineering at Princeton University. He served as an

associate editor of *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* and is currently serving as an associate editor of *IEEE Transactions on VLSI Systems* and of *Journal of Electronic Testing: Theory and Applications (JETTA)*. He served as the guest editor for the JETTA special issue on high-level test synthesis. He has also served as the program chairman of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems. He is the recipient of the AT&T Foundation Award and the NEC Preceptorship Award for research excellence. He co-authored two books titled *Testing and Reliable Design of CMOS Circuits* (Kluwer) and *High-Level Power Analysis and Optimization* (Kluwer). He authored or co-authored more than 150 technical papers. He co-authored three papers which have won the Best Paper Award at the IEEE International Conference on Computer Design (1993), the IEEE International Symposium on Fault-Tolerant Computing (1997), and the International Conference on VLSI Design (1998). He also received nominations for Best Paper Awards at three other conferences. His research interests include digital system testing, fault-tolerant computing, computer-aided design of integrated circuits, distributed computing, and real-time computing.