

Task Redistribution in Faulty Networks using Evolutionary Strategies

Garrison Greenwood
Department of Electrical Engineering
Western Michigan University
Kalamazoo, Michigan 49008, USA
garry.greenwood@wmich.edu

Ajay Gupta* and Mark Terwilliger
Department of Computer Science
Western Michigan University
Kalamazoo, Michigan 49008, USA
{ajay.gupta,mark.terwilliger}@wmich.edu

Abstract

Fault tolerant distributed systems must be able to continue operation in the presence of hardware faults (albeit with degraded performance). If one or more processors fail, the tasks assigned to those processors must be reassigned to other operational processors. The recovery mechanism thus reduces to a task allocation problem in a faulty network. This problem is known to be NP-complete.

In this paper we redistribute tasks in faulty networks using evolutionary strategies. We show that this technique is able to quickly determine a good reallocation of tasks with little degradation of performance.

1 Introduction

Hardware faults may reduce the availability or usability of a distributed computing system. Systems which continue to operate (albeit, with a degraded performance) in the presence of hardware failures are referred to as *fault tolerant systems*. The goal of any fault tolerant system is to recover from the fault as quickly as possible with a minimal performance degradation.

Duplicating hardware resources is a well-established means of minimizing this performance degradation. When faults occur, these redundant resources are brought on-line and the system operates as before. Unfortunately, one does not always have the luxury of having these additional hardware resources. For example, in most satellites space is at a

*Research supported in part by a Fellowship from the Faculty Research and Creative Activities Support Fund, WMU-FRACASF 94-040 and by the National Science Foundation under grant CCR-9405377.

premium; there is often no room to provide redundant hardware resources. If a processor fails in this satellite system, the remaining processors must continue operation with the tasks from the failed processor reallocated among the remaining processors. But, what is a good reallocation of these tasks in the faulty system and can this reallocation be quickly determined?

This paper presents a task reallocation technique for the class of fault tolerant distributed systems that do not have additional resources which can be brought on-line. This technique uses *evolutionary strategies* (ES) to determine the task reallocation. Our results indicate that good reallocations can be found within minutes when the ES is executed on a single processor and within seconds when executed on the PVM distributed computing system [6].

The paper is organized as follows. Section 2 discusses the fault tolerant systems of interest in our research. Section 3 gives an overview of the ES and how it is adapted to task reallocation. The main results of our research are presented in section 4. Conclusions and directions of future research are given in section 5.

2 Fault Tolerant Distributed Systems

A distributed memory multicomputer system consists of multiple computers interconnected by a message passing network. Each computer consists of a processor, local memory, and possibly some I/O peripherals. Parallel processing systems execute algorithms which require close cooperation between tasks thus requiring a message passing scheme which has minimal latency. Distributed systems have a series of autonomous processors and typically have a communication latency which does not permit close cooper-

ation between tasks. The granularity of an algorithm often dictates whether a parallel processing or a distributed system architecture is appropriate. The distributed systems of interest in our research consists of P nodes with a dual-ring ICN. The nodes are homogeneous so that any task can be assigned to any node. The dual-ring ICN allows messages to be passed in either direction.

Multicomputer scheduling involves the assignment of a set of computational tasks onto processors such that the overall schedule length is minimized. The precedence relationships between tasks are often depicted as a directed, acyclic graph (DAG). The graph nodes represent tasks and the arcs incident to nodes represent precedence relationships. Similarly, multicomputer systems can be modeled as an undirected processor graph where the vertices represent processors and the edges incident to the vertices represent links between processors. The scheduling problem is therefore a form of graph embedding problem; the task graph is to be embedded into the processor graph.

An optimal assignment of these tasks to nodes in the distributed system yields the minimal schedule length (i.e. the minimal time required to execute all of the tasks). The minimal schedule length must take into consideration the computational load at each node and the amount of communications overhead. Even in fully operational distributed systems, finding optimal assignments of tasks with precedence constraints is a known NP -complete problem. The problem is exacerbated when the network becomes faulty.

Fault tolerant systems must have the capability of detecting hardware failures and continue to operate albeit in a somewhat degraded manner. These hardware conditions may be 1) one or more processor failures, 2) one or more routing circuitry failures, or 3) one or more failures of links connecting processors. It is interesting to note the characteristics of the faulty network in each of these error conditions. For error condition 1, the ring topology is still maintained but the tasks from the faulty processor(s) must be redistributed. Since the routing circuitry is still intact, communications between some processors will exhibit greater propagation delay. Error condition 2 isolates the associated processor which also requires a task redistribution. Additionally, the ring now con-

sists of one or more linear arrays. Error condition 3 also converts the ring topology into one or more linear arrays. However, in the case of a single link failure, no task redistribution is necessary as the P -node ring becomes a P -node linear array. When the ring array is converted into a linear array by the fault condition, we always redistribute the tasks into the linear array with the largest number of processors.

The difficulty in assigning tasks in distributed systems has led researchers to adopt heuristic approaches. Recently there has been a great deal of interest (and success) in using ES to quickly find reasonable task assignments [2, 3]. This naturally raises a question. If ES can find good task assignments in fully operational systems, why can't they find reasonable assignments in faulty systems? In section 4 we address this specific question.

3 Evolutionary Strategies

ES are based upon the principles of adaptive selection found in the natural world. Each generation (iteration of the ES algorithm) takes a population of individuals (potential solutions) and modifies the genetic material (problem parameters) to produce new offspring. Both the parents and the offspring are evaluated but only the highest fit individuals (better solutions) survive over multiple generations. ES have been successfully used to solve various types of optimization problems. The reader is referred to Michalewicz for an excellent discussion of the ES technique and its applications [4].

The particular genetic encoding for an individual is referred to as the *genotype*. New genotypes are created by special operations (e.g., mutation) which modifies the genetic material. Decoding this genetic material gives the observed characteristics of the individual which is referred to as the *phenotype*. In our case, the genotype consists of P integer lists which reflects the tasks allocated to the P processors in the distributed system. The left-to-right order of tasks in a list indicates the order of execution. The phenotype is the resulting schedule length based upon this task allocation and execution ordering.

It is important to note that the ES investigates the search space of all possible task allocations to find those that produce minimal schedule lengths. Those regions of the search space which represent low sched-

ule lengths are selected for further investigation. Every point in the search space is a genotype. Search operations are conducted at the genotype level while selection (i.e., survival) is done at the phenotype level.

The initial population of individuals is randomly generated but, ideally, should be uniformly distributed throughout the search space so that all regions may be explored. Each individual in each generation is evaluated to determine its fitness. Individuals with high fitness represent task assignments producing low schedule lengths. The ES terminates after an acceptable solution has been found or a fixed number of generations (Γ) have been produced and evaluated. The ES is implemented as follows:

1. Conduct a breadth-first search of the task graph numbering the vertices in the order visited.
2. Create an initial population of μ individuals by selecting vertices from the task graph in numerical order and randomly assigning them to processors in the distributed system. Task numbers are appended onto the array corresponding to the selected processor.
3. For each individual, generate one offspring by applying the mutation operator (described below). This creates a population with a total of 2μ individuals.
4. Evaluate all individuals to determine their fitness. This is done by computing the schedule length based upon the indicated task assignments and their order of execution.
5. Select the μ most fit individuals for survival. Discard the other individuals.
6. Proceed to step 3 unless an acceptable solution has been found or Γ generations have been evaluated.

In distributed systems, balancing the computational load is an important factor in achieving reasonable speedups. Effective load balancing is achieved by a mutation operator which removes a small thread of execution from one processor and transfers it to another processor. This decreases the computational load on one processor and increases it on the other processor. Consider the example below which schedules tasks on a two processor system. (These concepts

are easily extended to the case where there are $P > 2$ processors.) The “♠” indicates the point of mutation.

Initial Assignment

Processor 1: 1 5 9 4 2 8 6 11 14 ♠ 15 13

Processor 2: 3 7 ♠ 12 10

The two tasks after the mutation point on processor 1 are transferred to processor 2. These tasks are inserted after the mutation point in the execution thread of processor 2. This results in the mutated assignment shown below. Notice how the mutation operator effectively balances the computational load.

Mutated Assignment

Processor 1: 1 5 9 4 2 8 6 11 14

Processor 2: 3 7 15 13 12 10

In practice a processor is randomly chosen and σ tasks after the mutation point from that processor are inserted after the mutation point in another processor’s task array. This action moves threads of execution between processors which helps to balance the computational load. σ is also randomly chosen but its range of values is dependent on the fitness of the individual.

If the individual has maximal fitness in the current generation, we choose $1 \leq \sigma \leq 3$. Otherwise, $2 \leq \sigma \leq 5$. When the communication to computation ratio is very low, maximally fit individuals have a reasonable load balance. By selecting only a small number of tasks to transfer, we minimize the chances of upsetting this balance. A larger number of tasks can be selected for individuals with lower fitness. This self-adaptation of the number of tasks transferred between processors helps to fine tune the final solution.

Task allocations are considered invalid if a schedule for the tasks cannot be obtained. Recall that the execution order of tasks assigned to a processor is reflected in the left to right order of tasks in the processor list. Certain ordering of tasks violate the precedence constraints indicated by the task graph. Consider the initial assignment in the example shown above. Task 5 assigned to processor 1 is scheduled to execute prior to task 9. If task 5 requires a result from task 9 prior to execution, a deadlock situation exists and no schedule can be obtained. Similar situations exist if task 5 requires results from tasks assigned to processor 2. (For example, task 5 needs a result from task 7, but task 3 needs a result from task 5.) It is entirely possible that the mutation operator will produce these invalid task allocations. This is not a prob-

lem as the ES is only concerned with the phenotype level; invalid allocations are assigned a low fitness and do not survive.

4 Results

Our research investigated task redistribution in distributed systems configured in a ring topology. All processors were homogeneous so tasks could be assigned to any processor. We chose distributed systems with 8, 16, 32, and 64 processors (PEs). Two types of task graphs were chosen for testing: complete binary trees and random graphs with selectable densities. Initially the tasks were allocated to processors using the ES. A fault was then chosen at random and all the tasks were then reallocated in the faulty network. Task execution times were randomly chosen to be between 1.0 and 20.0 seconds.

The schedule lengths were determined as follows. At each instant in time a processor is either actively executing a task or it is idle. Once a task had executed, a message containing the result was sent to any processor that has a task waiting for the result. Included in the schedule length was the communications overhead. This overhead consists of the propagation time associated with transmission over a link (τ) plus any queueing delay. The total amount of communications overhead (t_d) is given by

$$t_d = (1 + U + \lfloor \lambda t \rfloor) k \tau$$

or,

$$t_d = k \tau + k \tau U + \lfloor \lambda t \rfloor k \tau$$

where t represents time, $\tau = 2.3$ ms is the propagation delay per link, U represents a uniformly distributed number between 0 and 1, and k is the number of “hops” the message must traverse.

A task at processor i may have to send a message to processor j which requires transmission through k links. If there were no other messages in transit in the network, this would take $k \tau$ time. However, when the message arrives at each intermediate processor, there is the probability that another message is in the process of being transmitted. This causes a queueing delay which is modeled by the $k \tau U$ term.

A message may experience additional queueing delay because other previously sent messages are still

# of Faults	Binary Tree		Random Graph	
	8 PE	16 PE	8 PE	16 PE
None	6.9	11.5	4.8	7.1
1 PE	6.2	11.1	4.5	6.8
2 PE	5.5	10.5	4.1	6.7
3 PE	4.6	10.0	3.5	6.4

Table 1: Speedup for a 511-node task graph when PEs fail.

waiting for a link to become free. This situation exists even in virtual I/O systems as there are only a finite number of buffers available for temporary message storage. The Poisson process with parameter λ has been used for a number of years to model the arrival times of tasks [5]. The expected value of a Poisson process is λ and, therefore, the term $\lfloor \lambda t \rfloor$ represents the expected number of tasks waiting at a processor to be transmitted. We use this approach to model queueing delays in the distributed system. For our work we set $\lambda = 0.2$.

For test cases we chose both large binary trees and randomly generated task graphs that are distributed among processors interconnected in a ring topology. Faults were chosen randomly and the ES was run to find a reasonable task redistribution. In all cases a population size of $\mu=10$ was evaluated for 15 generations unless otherwise stated.

The tables 1 and 2 indicate the speedups that can be obtained with various PE or link faults. (Speedup in this instance refers to the ratio of the task graph execution time on a single processor to the execution time on P processors.) Table 1 indicates the speedup for both complete binary tree task graphs and random task graphs with 511 tasks where random PE failures occurred but all links remained intact. These task graphs were scheduled on a P -PE dual-ring ICN where $P = 8$ or $P = 16$. For the random graphs we investigated various edge densities though only the results from scheduling N node task random graphs with $4N$ edges are given. Comparable performance with other edge densities was achieved. Table 2 gives the case where random links have failed but all PEs remained operational. In both tables the no fault condition speedup is given as a reference so that the degraded performance can be assessed.

Comparing the two tables, the random graphs

# of Faults	Binary Tree		Random Graph	
	8 PE	16 PE	8 PE	16 PE
None	6.9	11.5	4.8	7.1
1 Link	6.2	11.0	4.5	6.7
2 Link	5.5	10.4	4.1	6.6
3 Link	4.6	9.9	3.5	6.3

Table 2: Speedup for a 511-node task graph when links fail.

have lower speedups overall because the graph densities are higher than that of the binary trees. These higher densities introduce more message traffic in the network which leads to increased schedule lengths. It appears that PE faults and link faults are equivalent as the resultant speedups are almost identical. However, there is a slightly greater speedup degradation with link failures. This result is expected as link failures not only isolate PEs (which no longer are involved in computations), but they also create greater edge congestion in the remaining links.

Once a fault condition has been detected, two methods can be used to determine the task redistribution. The first method lets one of the processors in the faulty network 1) run the ES, and 2) direct the task redistribution. The second method is offline; another processor or multiprocessor (possibly at a remote location) runs the ES and then informs the distributed system how to redistribute the tasks. Presently we will show that the second method is capable of finding better task redistributions but at a higher computational cost.

The first method incorporates a “self recovery” mechanism that provides a distinct advantage for distributed systems that are only periodically monitored by the user. For example, suppose a small distributed system is used to position a satellite in orbit. Clearly the performance of such a system is not easily monitored, and, depending on the fault, may not even be possible. With the first method the distributed system could detect the fault, run the ES algorithm, and redistribute the tasks accordingly without intervention from an earth bound station. This would at least keep the system operational though at a degraded level. The second method could then be run at a later time on the earth bound station to improve the level of operation.

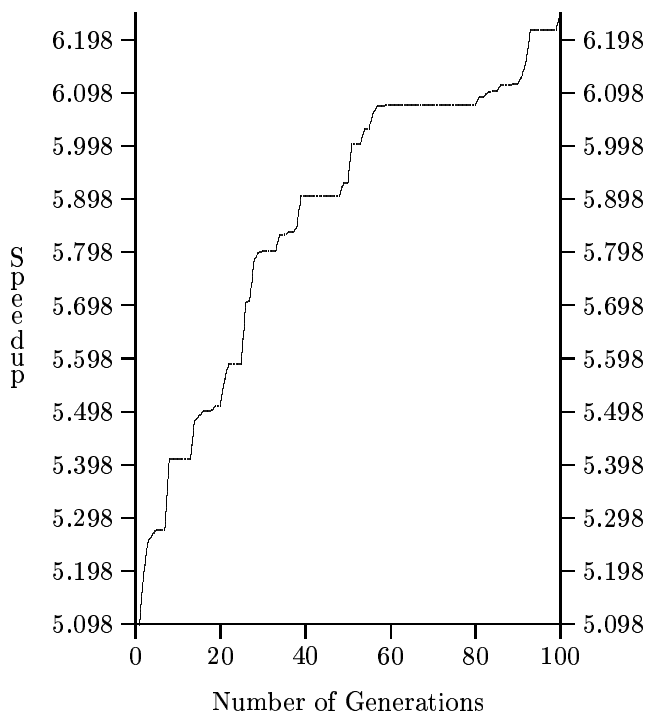


Figure 1: Speedup versus Number of Generations.

The quality of the solution provided by the ES improves with increasing population size and number of generations evaluated. Figure 1 shows how the solution quality improves as the number of generations increases. However, this larger number of generations also greatly increases the execution time of the ES algorithm. For example, with a population size of 200 evaluated for 1000 generations, the single processor execution time was almost 5 hours when run on a SPARC 1+ workstation. Since the objective is to find a good task redistribution quickly, the first method above can only use a small population size and a small number of generations. (We used a population size of 10 individuals evaluated for 15 generations.) The second method is free of such restrictions and is thus able to produce task redistributions with higher speedups. To minimize the execution time, we used the following parallel ES algorithm described below. The algorithm was implemented on a PVM system with \mathcal{P} processors.

1. Create an initial population of μ individuals by selecting vertices from the task graph in numer-

# of Faults	Seq. ES	Par. ES	Increase
None	5.1	6.5	27%
1 PE	4.7	6.0	28%
2 PE	4.3	5.0	16%
3 PE	3.8	4.4	16%

Table 3: Comparison between small population size & small number of generations and the larger ones.

ical order and randomly assigning them to processors in the faulty network.

- Partition the population into \mathcal{P} sets of M chromosomes each, where $M = \lceil \frac{\mu}{\mathcal{P}} \rceil$. Send one set to each processor.
- Each $p \in \mathcal{P}$ executes the ES for 10 generations. Then sends the best chromosome to the master processor.
- Master processor sends global best chromosome to all \mathcal{P} processors. Each $p \in \mathcal{P}$ replaces worst fit chromosome with global best chromosome received from master processor.
- Proceed to step 3 unless an acceptable solution has been found or Γ generations have been evaluated.

Table 3 compares the quality of the solution for both the redistribution methods described above. A random task graph with 255 nodes was scheduled on a 8 PE ring. The sequential ES used a population size of 20 and was evaluated for 15 generations. The average run time was 65 seconds. The parallel ES used $\mathcal{P}=20$, $M=5$, $\Gamma=1000$, and had an average run time of 1.25 hours. Notice that the larger population size is able to find task redistributions with a significantly greater speedup than that of the smaller size, sequential version. The long computation time of the sequential version makes this infeasible for running on a single processor of the faulty distributed system. However, the computation time for the PVM version is quite reasonable.

5 Conclusions and Future Research

This paper has introduced the use of ES to determine task redistributions in faulty networks with no

spare resources. Both a sequential and a parallel implementation of the ES were presented. Our results indicate that task redistributions with low degradation can be quickly found using this technique.

Future research in this area should concentrate on finding task redistributions in real-time, faulty networks. This problem is considerably harder in distributed systems that do not have spare resources. In this case, the objective is not to find the minimal schedule, but rather to find a feasible schedule. Such schedules may not always exist. However, we believe the speed in which the ES can redistribute tasks in faulty networks is a distinct advantage as the existence of a feasible schedule can be quickly ascertained.

References

- [1] S. Dutt and J. P. Hayes. Designing Fault-Tolerant Systems Using Automorphisms. *Journal of Parallel and Distributed Computing*, no. 12, pp. 249-268, 1991.
- [2] G. Greenwood, A. Gupta and K. McSweeney. Scheduling Tasks in Multiprocessors using Evolutionary Strategies. *1st IEEE Conference on Evolutionary Computation*, pp. 345-349, June 1994.
- [3] E. Hou, *et al.* A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113-120, Feb. 1994.
- [4] Z. Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs. *Springer-Verlag*, 1992.
- [5] J. Martin. Systems Analysis for Data Transmission. *Englewood Cliffs, NJ: Prentice-Hall*, 1970.
- [6] V. S. Sunderam, G. A. Geist, J. Dongarra and R. Manchek. The PVM concurrent computing system: Evolution, experience, and trends. *Parallel Computing*, vol. 20, no. 4, pp. 531-546, March 1994.
- [7] P. Yang and C. S. Raghavendra. Embedding and Reconfiguration of Binary Trees in Faulty Hypercubes. In the *Proceedings of the 6th International Parallel Processing Symposium*, pp. 2-9, March 1992.