LETTER

Parallelizing SDP (Sum of Disjoint Products) Algorithms for Fast Reliability Analysis

Tatsuhiro TSUCHIYA[†], ***Member***, **Tomoya KAJIKAWA**[††], ***Nonmember***, ***and*** **Tohru KIKUNO**[†], ***Member***

SUMMARY The SDP (Sum of Disjoint Products) approach is a well-known technique for computing network reliability measures. So far several algorithms have been developed based on this approach. In this letter, we present a general framework for parallelization of these SDP algorithms. Based on the framework, we implemented a parallel version of an SDP algorithm called CAREL on a network of workstations. Experimental results show that it works fairly well with almost linear speedups.
key words: *network reliability, networks of workstations, dependability evaluation, parallel processing, coherent systems*

1. Introduction

The *SDP (Sum of Disjoint Products) approach* is a well-known and important technique for reliability analysis. It has been used for computing various measures related to network reliability, such as *terminal-pair reliability* [1], [5], [10], *distributed program reliability* [7], *distributed system reliability* [7], [8], and the *availability* of mutual exclusion mechanisms [12].

Given the reliabilities of components and minimal sets of components that can allow a system to function (which we call *min-paths*), an SDP algorithm computes the reliability of the system, i.e., the probability that all components in at least one of these sets are operational. Several SDP algorithms have been developed so far (e.g., [1], [5], [9], [10]). In general, when using the SDP approach, reliability evaluation is performed in two phases; 1) enumerate all min-paths, and 2) compute the reliability measure by using an SDP algorithm. Results reported in literature show that usually the time of the second phase is dominant in the total running time needed for evaluation. It is thus important to reduce the running time of the SDP algorithms.

In this letter, we present a framework for parallelization of the SDP algorithms, and report the results of implementing a parallel version of an SDP algorithm called CAREL, which is one of the most recent SDP algorithms [10].

Although parallelization is a common approach to running programs faster, there is few work in the field of reliability evaluation. (Exceptions include, for example, [3].) To the best of our knowledge, this letter is the first to deal with parallelization of the SDP algorithms.

In general, parallelization involves overcoming some difficulties, including extracting parallelism and managing communication overheads. In the remainder of the letter, we show that the SDP algorithms inherently contain parallelism, and the parallel version of CAREL works very well even on a network of workstations, where communication overheads are much larger compared to other kinds of parallel machines.

2. Preliminaries

2.1 Model

We consider a system L that consists of n components. Each component has two states: operational or failed. We denote the ith component by $l_i (1 \leq i \leq n)$, i.e., $L = \{l_1, l_2, \cdots, l_n\}$. We assume that for each component $l_i \in L$, the probability that it is operational, denoted by p_i, is given. Failures of the components are mutually independent. The current state of the system is defined as a set $S \subseteq L$ such that all components in S are operational and all components in $L \backslash S$ are failed.

The system is either operational or failed. We assume that the system is *coherent* [9], that is, if the system is operational in state S, then it is operational in any state S' such that $S \subseteq S'$. Then there is a set $\mathcal{S}$ of states such that the system is operational iff some state in $\mathcal{S}$ is a (proper or improper) subset of the current state. Each element in $\mathcal{S}$ is called a *min-path*. The event that the system is operational, denoted by U, can be then represented by

$$U = \bigvee_{MP_i \in \mathcal{S}} \bigwedge_{l_j \in MP_i} u_j$$

where u_j denotes the event that component l_j is operational.

As an example, consider the network shown in Fig. 1 and suppose $L = \{l_1, l_2, l_3, l_4, l_5\}$. (For the purpose of readability, we do not consider node failures in this example.) Now let us assume that the system is operational iff there is an operational path between s and t. Then, for example, $\{l_1, l_2\}$ is a min-path. In this case, there are a total of four min-paths and $\mathcal{S}$ is

Manuscript received November 30, 1999.

[†]The authors are with the Department of Informatics and Mathematical Science, Osaka University, Osaka-shi, 560–8531 Japan.

[††]The author is with CASIO Computer Co., Ltd., Hamura-shi, 205–8555 Japan.

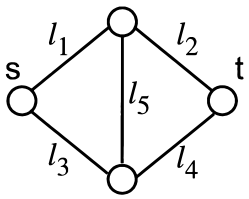

Fig. 1 A simple network.

$\{\{l_1, l_2\}, \{l_3, l_4\}, \{l_1, l_4, l_5\}, \{l_2, l_3, l_5\}\}$. Hence U is represented as $U = u_1u_2 \vee u_3u_4 \vee u_1u_4u_5 \vee u_2u_3u_5$.

Let E_i be the event that all components of $MP_i \in \mathcal{S}$ is operational, that is, $E_i = \bigvee_{u_j \in MP_i} u_j$. Then $U = \bigvee_{MP_i \in \mathcal{S}} E_i$, and the probability that the system is operation, denoted by R, can be written as

$$R = \Pr[U] = \Pr[E_1 \vee E_2 \vee \cdots \vee E_m] \tag{1}$$

where $|\mathcal{S}| = m$.

2.2 SDP Approach

The problem we consider is as follows:

Given $l_1, l_2, \cdots, l_n$
$p_1, p_2, \cdots, p_n$
$MP_1, MP_2, \cdots, MP_m$
Compute R.

(We assume that $MP_1, MP_2, \cdots, MP_m$ are sorted in an ascending order of their cardinality.)

In the SDP approach, R is computed based on the following equation.

$$\begin{aligned} &E_1 \vee E_2 \vee \cdots \vee E_m \\ &\quad = E_1 \vee \overline{E}_1E_2 \vee \cdots \vee \overline{E}_1\overline{E}_2 \cdots \overline{E}_{m-1}E_m. \end{aligned}$$

In the right hand side of the above equation, the product terms are disjoint and mutually exclusive with each other. Hence Eq. (1) can be transformed as

$$\begin{aligned} R = &\Pr[E_1] + \Pr[\overline{E}_1E_2] + \cdots \\ &+\Pr[\overline{E}_1\overline{E}_2 \cdots \overline{E}_{m-1}E_m]. \end{aligned}$$

The SDP algorithms obtain R by computing $\Pr[\sigma_i]$ for all $i(\leq m)$ where $\sigma_i = \overline{E}_1\overline{E}_2 \cdots \overline{E}_{i-1}E_i$.

If MP_j and $MP_{j'}$ contain no common elements for any $j, j'(j \neq j)$ such that $1 \leq j, j' \leq i$. then $\Pr[\sigma_i]$ can be easily obtained. In this case, E_j and $E_{j'}$ are independent, so $\Pr[\sigma_i]$ is given by

$$\begin{aligned} \Pr[\sigma_i] &= \Pr[\overline{E}_1]\Pr[\overline{E}_2] \cdots \Pr[\overline{E}_{i-1}]\Pr[E_i] \\ &= \prod_{1 \leq j \leq i-1} (1 - \prod_{l_k \in MP_j} p_k) \prod_{l_k \in MP_i} p_k. \end{aligned}$$

On the other hand, computing $\Pr[\sigma_i]$ is much complicated if MP_j and $MP_{j'}$ contain common elements. In such situations, $\Pr[\sigma_i]$ is given as

$$\Pr[\sigma_i] = \Pr[E_i]\Pr[\overline{E}_1\overline{E}_2 \cdots \overline{E}_{i-1}|E_i].$$

$\Pr[E_i]$ can be computed in a straightforward manner since $\Pr[E_i] = \Pr[\bigwedge_{l_k \in MP_i} u_k] = \prod_{l_k \in MP_i} p_k$. Evaluating $\Pr[\overline{E}_1\overline{E}_2 \cdots \overline{E}_{i-1}|E_i]$ is, however, a complex problem. In order to compute this probability, many techniques have been developed (see [9] for recent a survey). Different SDP algorithms use different techniques.

For example, given $\mathcal{S} = \{\{l_1, l_2\}, \{l_3, l_4\}, \{l_1, l_4, l_5\}, \{l_2, l_3, l_5\}\}$, an SDP algorithm CAREL [10] computes R by transforming Eq. (1) as follows:

$$\begin{aligned} R &= \Pr[u_1u_2 \vee u_3u_4 \vee u_1u_4u_5 \vee u_2u_3u_5] \\ &= \Pr[u_1u_2] + \Pr[u_3u_4]\Pr[\overline{u_1u_2}] \\ &\quad +\Pr[u_1u_4u_5]\Pr[\overline{u}_2\overline{u}_3] + \Pr[u_2u_3u_5]\Pr[\overline{u}_1\overline{u}_4] \\ &= p_1p_2 + p_3p_4(1 - p_1p_2) \\ &\quad +p_1p_4p_5(1 - p_2)(1 - p_3) \\ &\quad +p_2p_3p_5(1 - p_1)(1 - p_4) \end{aligned}$$

3. Parallelization and Load Balancing

Regardless of the difference in ways of computing $\Pr[\sigma_i]$, the task of computing $\Pr[\sigma_i]$ and that of computing $\Pr[\sigma_{i'}]$ for $i, i'(i \neq i')$ have no dependency in any SDP algorithm. This means that $\Pr[\sigma_i]$ and $\Pr[\sigma_{i'}]$ can be computed in parallel for any i, i'. Parallelism is thus intrinsic in the SDP algorithms.

The problem with computing R in parallel is then *load balancing*. It is known that in recent SDP algorithms the time for computing $\Pr[\sigma_i]$ grows linearly in i. Hence by statically allocating given min-paths to processors, it is possible to distribute the load equally to some extent, However, this may result in delay of execution because the execution time of $\Pr[\sigma_i]$ is not fully predictable. Hence we consider that dynamic distribution based on a master-slave model is preferable. Another important advantage of dynamic load distribution is that it is well suited for environments where machines run at different speeds.

As a distribution rule we adopt the *LPT (Longest Processing Time) scheduling rule* [6], which has been widely used for scheduling independent tasks onto multiprocessors (e.g.,[2]). By using this rule, we can balance loads among the processors, thus reducing the idle time spent by each processor after the last assigned task was finished. According to this rule, whenever a processor becomes free, the master process assigns to the processor a task whose execution time is the largest of the tasks not yet assigned. As stated above, although the time required for computing $\Pr[\sigma_i]$ is not fully predictable, it usually grows as the value of i increases. Thus we compute $\Pr[\sigma_i]$ in descending order of i.

The remaining problem is communication overheads. We use a *task clustering* technique to reduce the overheads. Task clustering means to lump several tasks together as a larger one. This technique thus reduces the number of message exchanges required for assigning tasks to the processors. Let P_i denote the task corresponding to the computation of $\Pr[\sigma_i]$ and let c be a

Table 1 Execution times (in seconds) of the parallel version of CAREL.

# of processors	1	2	3	4	5	6	7
Network 1	48.8	25.3	17.7	14.2	12.3	11.4	10.5
Network 2	130.6	66.3	45.1	34.8	28.8	25.0	22.2
Network 3	154.1	78.1	53.1	40.9	33.9	29.4	26.4
Network 4	337.6	170.4	115.3	88.0	72.1	62.0	54.8
Network 5	4918.3	2435.4	1625.8	1217.7	981.6	816.1	703.2

positive integer. We cluster $P_m, P_{m-1}, \cdots, P_{m-c+1}$ into task J_1 (i.e., $J_1 = \{P_m, P_{m-1}, \cdots, P_{m-c+1}\}$), P_{m-c}, $P_{m-c-1}, \cdots, P_{m-2c+1}$ into J_2, and so on, and then assign $J_1, J_2, J_3, \cdots$ in this order to the processors, according to the LPT scheduling rule. (It should be noted, however, that as the number of min-paths in a task, i.e., the value of c grows, the communication overheads incurred by distributing tasks decrease, but the idle time of each processor may increase due to increase of task granularity.)

The following describes the behavior of the master and slave processes. Here N denote the number of the slave processes. We assume that no two slave processes run on the same processor.

The master process
Step 1) Send the inputs (all the min-paths and the reliabilities of the components) to all the slave processes.
Step 2) Assign the first N tasks, i.e., $J_1, J_2, J_3, \cdots, J_N$, to the slave processes so that they will handle one task each.
Step 3) When receiving a result from a slave process, assign that process the next task. Repeat this step until no task remains.
Step 4) Wait for results from slave processes that have not completed their assigned tasks. When receiving results from all such processes, send termination messages to all the slave processes. Then stop.

The slave processes
Step 1) Receive the inputs from the master process.
Step 2) On receiving a task from the master process, execute the task. When finishing the task, send the result to the master.
Step 3) Wait for a message from the master. If the message is for assignment of another task, then repeat from Step 2. Otherwise, the message is a termination message. Then stop.

Note that a task can be sent simply by designating the indices of the first and the last min-paths to be processed. The result of task J_i is the value of $\sum_{P_j \in J_i} \Pr[\sigma_j]$. The reliability of the system, R, is obtained simply by summing the result values the slave processes returned to the master process.

4. Implementation and Experiment

Based on the proposed approach, we implemented a parallel version of CAREL [10] in the C language using the PVM message passing library [4]. In the program, a master process is first created, and then it spawns slave processes so that every workstation will run one slave process each. Thus the workstation running the master process also executes one slave process.

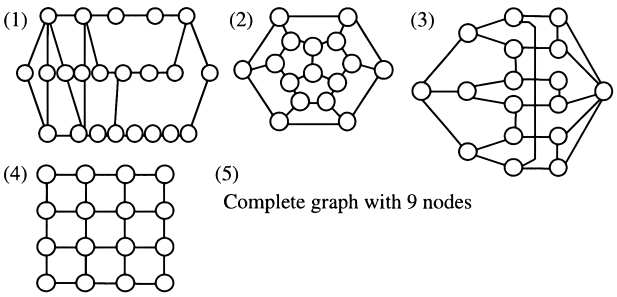

Fig. 2 Benchmark networks.

We conducted an experiment on a collection of workstations connected by a 100baseT Ethernet. Each workstation was running Free-BSD with a 400 MHz Pentium II processor and 128 Mbyte memory.

We set the value of c to $\lceil \frac{m}{10 \cdot (\#\text{ of workstations})} \rceil$ so that each workstation would run 10 tasks on average. (From preliminary experiments, we had found that this value is small enough to make the idle time of each workstation almost negligible.)

To assess the performance of the implemented program, we recorded the time spent in computing the availability of the *majority voting scheme* [11] for each network in Fig. 2. The availability of majority voting is defined as the probability that a majority of the nodes are connected by operational paths. Therefore $\mathcal{S}$ is the set of all minimal trees that span exactly $\lceil |V|/2 \rceil$ nodes where V is the set of nodes. The total number of the min-paths, $|\mathcal{S}|$, is 8738, 10797, 11777, 18616, and 15750 for Networks 1, 2, 3, 4, and 5, respectively. (The min-paths were obtained by using an algorithm proposed in [12].) Since we consider both node and link failures, $L = V \cup E$ holds where E is the set of edges.

Table 1 shows the running times needed for computing the availability from the set of min-paths. (Each value is the average of 10 runs.) Figure 3 shows the speedup ratios. The speedup ratio is defined as W/W_p where W is the execution time of CAREL on a single workstation and W_p is that of the parallel version. (Availabilities obtained are omitted due to the limit of space. Interesting readers are referred to [12].)

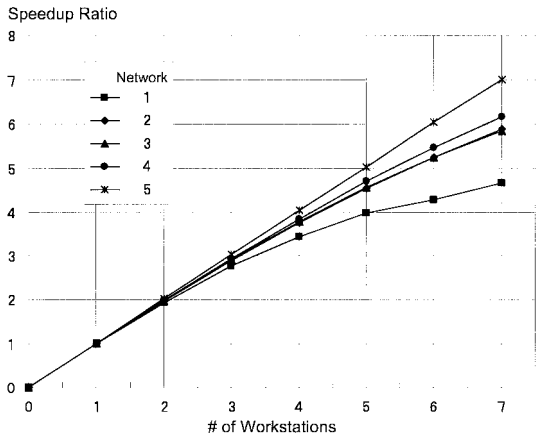

Fig. 3 Speedup ratios.

Table 2 Overheads (in seconds).

# of workstations	2	3	4	5	6	7
Network 1	1.5	1.9	2.2	2.5	2.9	3.5
Network 2	1.6	2.0	2.4	2.8	3.2	3.6
Network 3	1.8	2.3	2.8	3.3	3.6	4.2
Network 4	2.8	3.4	4.2	4.9	5.5	6.4
Network 5	2.7	3.3	4.0	4.5	5.1	6.1

From these results, it can be seen that except for Network 1, the speedup ratios are almost linear. For Network 1 the speedup ratio begins to saturate when the number of workstations exceeds 5. The reason is that the number of min-paths to be processed was relatively small for Network 1, so the communication overheads made tangible effects on the running time.

To investigate this reason further, we measured the overheads in the parallel program. This was done by measuring the finishing times of a program obtained by removing all statements contributing to reliability computation from the parallel version of CAREL. Table 2 shows the results. As expected, the overheads increase as the number of min-paths and that of workstations grow. In the case of Network 1, the ratio of the overhead to the total running time is much larger than the other cases. Thus it is found that the gain obtained by adding a workstation diminished more rapidly.

To illustrate this, let us consider the case where a workstation is added to six workstations. Since the running time for Network 1 is less than 49 seconds on a single workstation, the reduction in the running time achieved by this extra workstation is at most $49/6 - 49/7 \approx 1.17$ seconds. On the other hand, the overhead increases by $3.5 - 2.9 = 0.6$ second in this case, and thus lessens the speedup to a considerable extent.

5. Conclusions

In this letter, we discussed parallelization of SDP algorithms. We pointed out that the SDP algorithms inherently contain parallelism and present a general framework for parallelization. Based on the framework, we implemented a parallel version of CAREL, which is one of the most recent SDP algorithms, and conducted an experiment on a network of workstations. The experimental results show that when the workload is not small, the parallel version of CAREL achieves significant speedup, which is almost linear in the number of workstations. Future research includes developing an analytic model for predicting the performance.

References

[1] J.A. Abraham, "An improved algorithm for network reliability," IEEE Trans. Reliab., vol.28, pp.58–61, 1979.
[2] V. Cherkassky and C.-I.H. Chen, "Redundant task-allocation in multicomputer systems," IEEE Trans. Reliab., vol.41, no.3, pp.336–342, 1992.
[3] N. Deo and M. Medidi, "Parallel algorithms for terminal-pair reliability," IEEE Trans. Reliab., vol.41, no.2, pp.201–209, 1992.
[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing, The MIT Press, 1994.
[5] S. Hariri and C.S. Raghavendra, "SYREL: A symbolic reliability algorithm based on path and cutset methods," IEEE Trans. Comput., vol.36, no.10, pp.1224–1232, 1987.
[6] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham, "Worst-case performance bounds for simple one dimensional bin packing algorithm," SIAM J. Comput., vol.3, pp.299–325, 1974.
[7] V.K. Prasanna Kumar, S. Hariri, and C.S. Raghavendra, "Distributed program reliability analysis," IEEE Trans. Software Eng., vol.12, no.1, pp.42–50, 1986.
[8] C.S. Raghavendra, V.K. Prasanna Kumar, and S. Hariri, "Reliability analysis in distributed systems," IEEE Trans. Comput., vol.37, no.3, pp.352–358, March 1988.
[9] S. Rai, M. Veeraraghavan, and K.S. Trivedi, "A survey of efficient reliability computation using disjoint products approach," Networks, vol.25, pp.147–163, 1995.
[10] S. Soh and S. Rai, "CAREL: Computer aided reliability evaluator for distributed computing networks," IEEE Trans. Parallel and Distributed Systems, vol.2, no.2, pp.199–213, 1991.
[11] R.H. Thomas, "A majority consensus approach to concurrency control," ACM Trans. Database Systems, vol.4, no.2, pp.180–209, 1979.
[12] T. Tsuchiya and T. Kikuno, "Availability evaluation of quorum-based mutual exclusion schemes in general topology networks," Comput. J., vol.42, no.7, pp.613–622, 1999.