Algorithm–based Fault–tolerant Programming in Scientific Computation on Multiprocessors

J. Altmann, A. Böhm

University of Erlangen–Nürnberg
IMMD III, Martensstr.3, 91058 Erlangen, Germany
jnaltman@informatik.uni-erlangen.de

Abstract

Efficient parallel algorithms proposed to solve many fundamental problems in scientific computation are sensitive to processor failures. Because of its low costs, algorithm-based fault tolerance is an interesting concept for introducing fault tolerance into existing multiprocessors. To facilitate fault–tolerant programming in scientific computation, we have modified and developed further an existing parallel run–time environment. In this paper the aspect of tuning known error processing techniques to the algorithm–based approach is primarily examined. Design issues for implementation and execution time overhead of a fault–tolerant application in our run–time environment are studied. In contrast to many other environments for parallel fault–tolerant programming, which use the master/slave programming model, our environment enables one to add fault tolerance to existing parallel applications in scientific computation.

1 Introduction

Over the last several years, a lot of methods for algorithm–based error detection have been proposed. Simple and effective mechanisms for error detection are available for many applications. These methods allow the detection of hardware faults at the application software level with a run–time overhead of about 10 %. Because of the low costs, algorithm–based error detection is interesting for introducing fault tolerance to existing multiprocessors.

After closely studying algorithm–based fault tolerance methods on multiprocessors, we noticed that it is easy to implement error detection but difficult to do error processing. Algorithm–based error detection assumes the existence of standard techniques for error processing. These could be part of the system software but are not yet provided on existing multiprocessors. Therefore, it was necessary to choose, implement, and tune known error processing techniques to the algorithm–based approach.

To get a simple but efficient system which can be used on several multiprocessors, we choose an explicit fault–tolerant programming scheme instead of a transparent scheme. In order to re–use implemented fault tolerance techniques and to facilitate the programming effort, these techniques were integrated into an existing environment for parallel programming. Our environment for fault–tolerant parallel programming was installed on two multiprocessors, a Meiko T800 Transputer system and a German vector–processor Suprenum. Then the provided services were refined by implementing and testing several algorithms with algorithm–based error detection.

In this paper it is shown how fault tolerance based on algorithm–based error detection is integrated into parallel applications making use of our run–time environment. Our environment enables one to add fault tolerance to existing parallel applications in scientific computation. For a general algorithm–based approach, services for fault–tolerant programming, such as checkpointing, recovery, diagnosis, and reconfiguration, are provided in a fault tolerance layer between the application and the operating system.

This paper is organized as follows. In Section 2 the background of fault–tolerant computation on existing multiprocessors is described. The software implemented fault tolerance techniques of our run–time environment are pointed out in Section 3. Error detection, fault diagnosis, and recovery mechanisms are discussed in particular. Based on the results of Sections 2 and 3, an example of a fault–tolerant application is given in Section 4. Run–time overhead of the fault–tolerant application and loads in the event of an error are shown.

2 Fault–tolerant computation

Multiprocessors offer a cost effective approach to super-computing by connecting together a large number of general purpose and cheap processors. An important issue for the utilization of multiprocessors is reliability for long running computations. The number of failures caused by hardware faults increases for these applications. In response to this problem, fault tolerance is introduced in multiprocessor systems through hardware and software redundancy.

In this paper we study software techniques to work around hardware faults. Software techniques could be integrated into existing multiprocessors, instead of hardware techniques that are only suitable during new system design. Most of the proposed software techniques in parallel architectures focus on mainly one aspect of fault tolerance. For example, rollback recovery techniques only assume that an error detection facility and off–line diagnosis exist. The assumptions of error detection and error localization are not treated adequately by the rollback recovery software. From the other side, system level diagnosis does not take into account requirements of rollback recovery. Little interaction occurs between error detection, error localization, and the rest of error processing.

Therefore, we propose an integrated fault tolerance scheme based upon algorithm–based error detection integrated into parallel applications. The algorithm–based error detection has been studied for many applications, such as matrix operations [7], solving of partial differential equations [9], Fast Fourier Transforms [8], and solving of systems of linear equations [1]. All further services for fault–tolerant programming are tuned to the requirements of a general algorithm–based approach. This is in contrast to existing fault tolerance schemes that are based on the system's point of view. Fault tolerance at the system level is dedicated to one machine, whereas algorithm–based fault tolerance is portable. Nevertheless some assumptions about the hardware (machine model) and application software (programming model) have to be made.

2.1 Machine Model

We implemented our applications on MIMD computers (machine model) with distributed memory. There are no shared variables for communication between concurrent processes. These machines have one desirable characteristic: a faulty processor can only corrupt data in other processors via messages. This means that fast error detection and processing prevent error propagation.

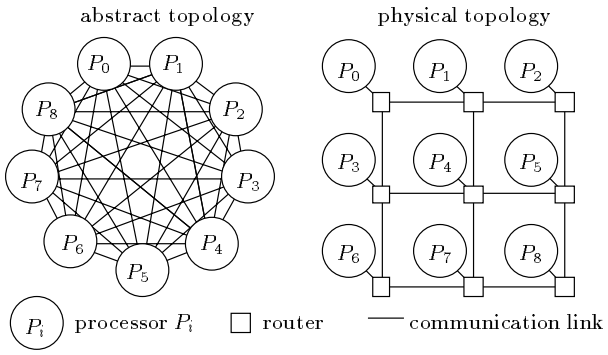


Figure 1: Abstract and physical machine

We chose a message passing machine with full connection as our topology for abstracting as far as possible from the target machine (s. Fig. 1). This abstraction is justified because new multiprocessors have hardware routers to speed up message passing. The physical topology of the multiprocessor is transparent to the user. Furthermore, the user can realize a fully connected topology using our run–time environment. The topology of an abstract message passing machine is useful for our run–time environment. We define an abstract/physical machine relation which can be changed by reconfiguration.

2.2 Programming Model

Processes on MIMD computers with distributed memory normally communicate by message passing. In order to meet the requirements of long running applications in the area of scientific computation, we chose the programming model of communicating sequential processes (CSP) [6]. This is in contrast to fault–tolerant applications based often on the master/slave programming model. Many other environments for parallel fault–tolerant programming use the master/slave programming model to introduce fault tolerance by replication and voting. Our environment enables one to add fault tolerance to existing parallel applications in scientific computation. Because of it high costs, replication and voting is not suitable for scientific computations.

Scientific programs for solving of engineering or physics problems usually contain the following three phase:

Initialization phase: This phase starts the application and the fault tolerance services. It is not necessary to protect this phase against errors, because the initialization is short.

Working phase: Solving a specific problem, this is the phase where most of the time is spent. The working phase is subdivided into processing and communication phases. Within our programming model, the node processes may communicate with each other during parallel computation. The whole working phase should be fault–tolerant, which means that checkpoints have to be written at regular intervals.

Output phase: The output of a calculation is performed in most cases by writing to a file. Usually, this phase begins near the end of a program, but some programs frequently switch between working phases and output phases.

In our programming model, program loading, distribution of data to node processors, and collection of the results are performed by a host program. The host program is not involved in the parallel computing. There is no redundancy in the host program. To introduce fault tolerance into the host program, conventional techniques, e.g. replication and voting, could be used. This aspect is not addressed in this paper.

2.3 Fault Model

The fault model, which describes the considered system errors, comes from structural and functional analysis of the system. Due to the algorithm–based approach, the fault model is determined by the machine model and the programming model. As mentioned above, we use the machine model of an abstract message passing machine and the programming model of CSP. Considering possible errors at this high level, it was noted that a node process must handle the following situations:

1. Timeout during communication
2. Exception occurrence
3. Errors detected by algorithm–based checks

Fault–tolerant behavior is achieved by detecting these errors and working around them with support of the run–time environment. In order to support long running applications, it is not our task to find the fault which causes the error. For example, a timeout can happen during communication if one of the processes has control flow errors or if faults in the communication network occur. What is important is that the communication error is detected; determining the fault is beyond the scope of our fault tolerance scheme.

Multiple faults can also result in an exception. An exception is the reaction of the operating system to an error; the operating system attempts to avoid the system failure.

Error detection by computational checks is characteristic for the algorithm–based approach. The programmer has to design and implement these checks during implementation. Errors detected by these checks can be caused by temporary or permanent hardware faults. A run–time environment has to support this.

3 Environment for Fault–tolerant Programming

Many software implemented techniques are known for achieving fault tolerance. We have chosen and modified several mechanisms to meet the requirements of the algorithm–based approach. These services (s. Fig. 2) are provided in a run–time environment, called *Portable Instrumented Communication Library for Fault Tolerant Programming* (PICLFT). It is a modified and further developed version of PICL [5]. PICLFT is comprised of modules such as *checkpointing*, *error detection*, *diagnosis*, *recovery*, *reconfiguration* and *rollback*.

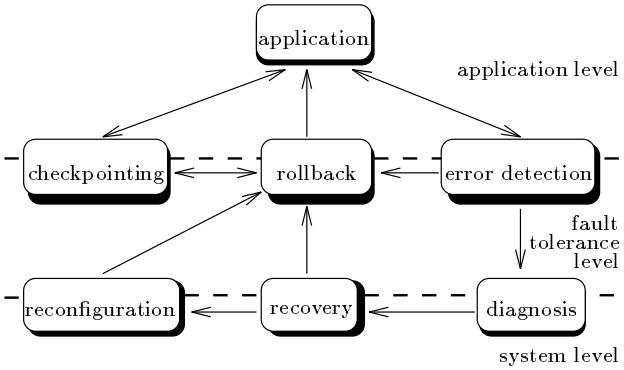


Figure 2: Mechanisms of fault tolerance in PICLFT

An example is given, to present the services of our run–time environment. Interdependencies of the different modules and the interaction of the user are indicated. Algorithm 1 shows the typical structure of a fault–tolerant application in PICLFT. At first, the application and the run–time environment are initialized. This means that an error handler is installed (`ft_error_handler0()`), the data for the first checkpoint is specified (`ft_add_memblock0()`)), and a checkpoint is written (`ft_take_cp0()`). As mentioned above, the computation is divided up into processing phases and communication phases, which are enclosed

by a loop. Within the communication phase, all processors check whether or not an error message can be received. This is done in the statement for receiving messages (`ft_recv0()`). If no error occurs, the next checkpoint is written. Please note that the user have not to be concerned with error processing. This is done by the run–time environment. Only initialization of the run–time environment and algorithm–based checks for error detection have to be installed by the user.

```
void main()
{ /* initialization phase */
  ft_setarc0();
  ft_error_handler0();
  switch (setjmp()) {
    case 1: /* rollback recovery */
      ...
    default: /* set up checkpointing */
      ft_add_memblock0();
      ft_take_cp0();
  }
  for () {
    /* processing phase */
    /* communication phase */
    ft_send0(); ft_recv0();
    /* checks */
    if (error) ft_error0();
    ft_take_cp0();
  }
}
```

Algorithm 1: A fault–tolerant application in PICLFT

3.1 Error Detection

One issue that is often neglected in the design of parallel architectures is the mechanism for detecting faulty processors. Algorithm–based error detection is a good choice, because it works concurrently with the application. It detects errors which are caused by permanent and temporary faults. Error detection by off–line testing can only detect errors caused by permanent faults.

Algorithm–based error detection, which has been studied for a lot of applications, is very much akin to error detection in the recovery block scheme [12]. However, the aim is completely different: algorithm–based error detection has to detect hardware faults, whereas checks in the recovery block scheme has to recognize software faults. Nevertheless, algorithm–based techniques and recovery block scheme use similar techniques for error detection. These techniques for error detection can be adopted to each algorithm, but it is not based on a general test method. Instead, there are a number of distinct checks: reversal checks, coding checks, reasonable checks, and structural checks. In the case of errors, detected by algorithm–based checks, error processing is initiated by a call to `ft_error0()` (s. Alg. 1).

Furthermore, error detection is done on–line by watchdog timers for communication and operating system (system exceptions). In the case of error occurrence, detected by watchdog timers and system exceptions, the error handler is automatically started. First, the detecting processor broadcasts an error message. On receiving an error message, the application is stopped, and the error processing (diagnosis, reconfiguration, rollback) is started. In order to avoid error propagation, the error processing has to start fast. Therefore, the error messages are immediately distributed by a fault–tolerant broadcast [4].

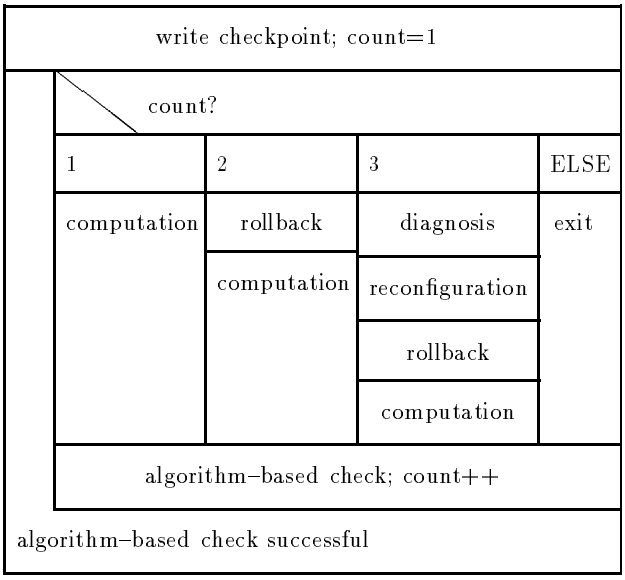
3.2 Recovery

In contrast to error detection, which has to be programmed by the user, our run–time environment PICLFT provides a simple and effective strategy for error processing. It is the task of error recovery to choose a convenient technique. The strategy of the recovery in PICLFT is shown in Algorithm 2.

In order to recover from faults, it is necessary to write checkpoints. If an algorithm–based check fails once, we assume a transient error. A rollback of the application to the last checkpoint is carried out which is sufficient to recover from transient faults. Should rollback or the algorithm–based check fail after another computation, diagnosis is started. If the diagnosis detects permanent faulty units, a reconfiguration is performed. Finally, if the system crashes more than twice for the same reason, the application is terminated.

3.3 Checkpointing

Checkpoints are written to stable memory to be available for a restart after the system breaks down. This means that the nodes write checkpoints to the file system of nodes in the multiprocessor or to the file system of the host. The checkpointing is organized by the user and supported by our environment. With respect to the structure of applications in scientific computation, the user creates checkpoints after a communica-



Algorithm 2: Strategy of recovery in PICLFT

tion phase and by a call to `ft_take_cp0()` (s. Alg. 1). It is the task of the user to specify the time of the checkpoint and data (`ft_add_memblock0()`) which is stored. On calling `ft_take_cp0()`, each node writes its local checkpoint. Using the two phase commit protocol [2], global checkpoints are achieved.

3.4 Rollback

A rollback is initiated (s. Alg. 2), if an error has occurred. On rollback, the values of the variables which were stored at the last checkpoint are loaded first. Then, the application is started execution from a predefined statement in the program (s. Alg. 1, `setjmp()`). Rollback requires no interaction with the user. If the rollback fails, the diagnosis will be started without repeating the computation.

3.5 Diagnosis

If the error occurred more than once, diagnosis will be started to locate the error by additional off–line tests. Furthermore, the diagnosis has to classify the error as permanent or intermittent. Because diagnosis works distributedly, a loosely synchronized application is necessary. The structure of scientific applications meets this requirement. All processors are loosely synchronized during the communication phase. Therefore, $<$*I'm alive*$>$ messages are not needed to achieve loose synchronization. This approach shows that we use the structure of the application for an efficient diagnosis.

Important details for the diagnosis are obtained by considering an error from the application point of view. The user agrees to rollback of an application after reconfiguration, if and only if the parallel program can process efficiently on a degraded system. Besides this, the number of processors for efficient processing is fixed for many scientific applications. Therefore, the user is usually not interested in how many processors are really faulty. Only the number of connected fault–free processors is important. The diagnosis has to process the fault–free processors remaining in operation and not the faulty processors.

Additionally, there is no reason for localizing the physical sources of such errors. In a self–diagnosing system, two processors should determine whether or not they can communicate. But processors are not capable of repairing the faulty components. For these reasons, the diagnosis algorithm has to diagnose only fault–free processors and not the faulty components. Faulty communication links and router hardware are detected only by disconnecting processors.

Taking this into consideration, we can permit all kinds of faulty components and any number of faulty components in our diagnosis model. Our diagnosis algorithm does not assume a t–diagnosibility of the multiprocessor structure. It diagnoses the connected fault–free processors. This diagnosis goal does not cause any problems, if we consider again the diagnosis from the application point of view. For example, even if the multiprocessor is divided into two sets of connected processors, the recovery decides whether a reconfiguration of the application on the connected fault–free processors can be carried out. The criteria for this decision are the number of fault–free processors, the number of spare processors, and whether there is a connection between the host and the sets of connected fault–free processors.

3.6 Reconfiguration

Permanent processor faults necessitate reconfiguring the processors. Because graceful degradation is not possible for all applications and dynamic allocation of processors is not yet supported by existing multiprocessors, we chose a more general approach. At the start of the application, we reserve some processors as spares. This means that an application which runs on p processors is loaded on $p+s$ processor for a fault–tolerant computation. After loading $p+s$ processes, only p processors are working while s pro-

cesses are blocked in the initialization phase (s. Alg. 1, `ft_setarc0()`). When required, up to s spare processes are unblocked for reconfiguration.

Such reconfiguration can be easily carried out on an abstract message passing machine. At first, the faulty processors are isolated by stopping communication with these processors. Then spare processors are activated and integrated into the application. Since we hide the physical machine from the user, we can easily change the relation between the abstract and the physical machine. The computational tasks of the faulty processors are moved to the spare processors. Therefore, reconfiguration is transparent to the user.

4 Experimental Results

To demonstrate the use of our run–time environment, an example is given. In this section we show how the *Conjugate Gradient (CG)* algorithm can be implemented in a fault–tolerant way using our run–time environment for fault–tolerant applications (PICLFT). A *concurrent error detecting CG algorithm* (CEDCG) was proposed in [1], but no experimental results in terms of speed–up and error coverage are given. As in all known algorithm–based fault tolerance schemes, the main focus is on error detection; error processing is neglected. Supported by PICLFT, error detection is achieved by checking algorithm specific properties at the end of each iteration. For error processing, services from our run–time environment are available. This means that we can offer services for fault tolerant programming in a more general way without concentrating on a single application.

In the area of scientific computation, the user is interested in availability and performance of a multiprocessor. Therefore, the quality of a fault–tolerant application is measured by the error coverage and the run–time. Within our example, we examine the run–time overhead, the error coverage, and the additional load in an error event.

4.1 CG Algorithm

The *Conjugate Gradient (CG)* method is popular for solving large, sparse, linear systems of equations of the form $Ax = b$ with $x, b \in \mathcal{R}^n$ and $A \in \mathcal{R}^{n \times n}$ on parallel architectures. The solution of the linear system of equations is encountered in many scientific problems. Especially in *Finite Element Applications* (FEM), such a matrix equation is repeatedly formed and solved. The sequential CG algorithm is presented in Algorithm 3.

$x_0 = 0$; $\beta_0 = 0$; $p_0 = r_0 = b - Ax_0$
for $k = 0, 1, 2, \ldots$
  $q_k = Ap_k$
  $\alpha_k = \frac{<r_k, r_k>}{<p_k, q_k>}$
  $r_{k+1} = r_k - \alpha_k q_k$
  $x_{k+1} = x_k + \alpha_k p_k$
  $\beta_k = \frac{<r_{k+1}, r_{k+1}>}{<r_k, r_k>}$
  $p_{k+1} = r_k + \beta_k p_k$

Algorithm 3: Sequential CG algorithm

All basic operations of the CG algorithm, such as matrix vector multiplication, dot–product $(<,>)$, and vector addition, can be performed concurrently by distributing the rows of A and the corresponding elements of the vectors b, x_k, r_k, q_k and p_k among the processors. This means that each processor has to compute n/p elements of a vector, if n is the number of unknowns and p is the number of processors.

In our example we use the reversal checks, which are proposed in [1]. It is shown that it is necessary to compute four extra dot–products ($< p_i, Ap_{i-1} >$, $< r_i, r_{i-1} >$, $< b, p_{i-1} >$, $< x_i, Ap_{i-1} >$) per CG iteration for performing the checks. These checks can detect the presence of an error.

4.2 Run–Time Overhead

Let k be the number of non–zeroes per row in matrix A, n the number of unknowns, and p the number of processors. Furthermore, let t_{calc} be the time of one floating point operation, t_{su} the start up time of communication, and t_{trans} the transmission time of a double precision floating point number (64 bit). The time complexity of the CG algorithm can then be modeled by:

$$t_{cg} = 2(k+5)\frac{n}{p}t_{calc} + 4ld(p)(t_{su} + t_{trans}) + L_c$$

The first term describes the computation complexity. It is made up of two dot–products $\mathcal{O}(4n/p)$, three vector additions with scalar multiplication $\mathcal{O}(6n/p)$, and one matrix vector multiplication $\mathcal{O}(2kn/p)$. Communication complexity is modeled by the remaining terms. The two dot–products are considered separately in the second term. The factor $2ld(p)$ consists of the number of communications for collecting and distributing provisional results through a hypercube topology. The rest of the communication complexity is hidden in the term L_c.

Processors	Meiko	Suprenum
1	11.7 %	11.6 %
2	11.7 %	11.5 %
4	11.6 %	11.4 %
8	11.4 %	11.0 %
16	10.9 %	10.3 %
32	9.6 %	9.0 %
64	7.9 %	7.0 %

Table 1: Overhead of CEDCG algorithm on FEM problem with 12.000 unknowns

As mentioned above, the CEDCG algorithm needs four additional dot–products. So the time redundancy of the CEDCG algorithm can be modeled by:

$$t_{red} = 8\frac{n}{p}t_{calc} + 8ld(p)t_{trans}$$

Note, that no additional communication is necessary by including the additional data into the existing messages. Hence, communication redundancy can be neglected. The overhead t_{over} is only determined by the computational redundancy which is processed by:

$$t_{ov} = \frac{t_{red}}{t_{cg} + t_{red}} = \frac{4}{k+5}$$

For a system of linear equations arising from a two–dimensional FEM problem with triangle as basic elements, the number of non–zeroes per matrix row is $k = 19$. This means that the overhead will be estimated 16.6 %. Experimentally, we measured an overhead of only 11.7 % for a sequential program (s. Tab. 1).

Due to the high cost of index computation in our sparse matrix representation, the real overhead is much lower than estimated. We observe that the overhead will be reduced, if a fixed problem is solved on an increasing number of processors. In this case, the additional computation can be carried out during idle times (s. Tab. 1). The overhead is measured on a Meiko Transputersystem (T800) and the German vector–processor Suprenum. The scaled speed–up is nearly linear, which is not presented in the graph.

The additional operations of checking increase the work load on each processor, while the communication load remains constant. Therefore, the scaled and the normal speed–up (s. Fig. 3) are comparable or even better than those for the non-error detecting CG algorithm. The error detecting CG algorithm can be used with high efficiency on Suprenum and Meiko.

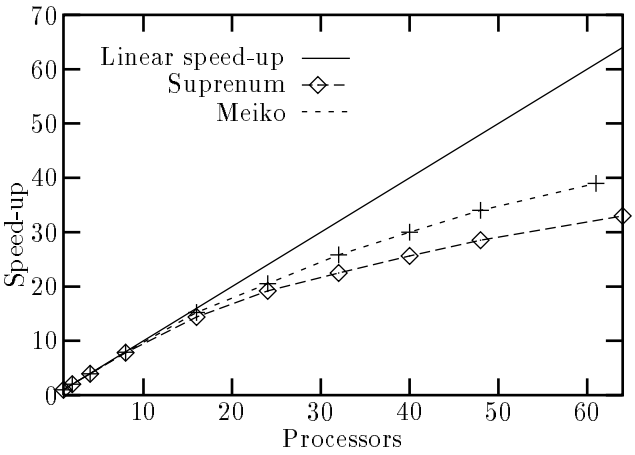


Figure 3: Speed–Up for CEDCG algorithm on FEM problem with 12.000 unknowns

4.3 Error Coverage

The error coverage of the CEDCG algorithm is studied by error injection. Errors are injected by randomly flipping bits of the vectors p_k, q_k, r_k and x_k which are computed during each iteration. For each injection, we select one floating point value out of one of the vectors and flip one bit. This is not a realistic fault model, but it is the smallest possible impact of a fault. It is shown that this slight modification can be readily detected. Results from the error injection are independent of the multiprocessor. Both machines use the same representation of floating point values (IEEE Standard 754-1985).

In spite of the high level approach, we have to take the floating point representation into account. Thus, we can not check for equality. We have to choose a tolerance. The tolerance depends on the problem and causes false alarms if it is not suitably chosen. Because it is not possible to give a rate for the error coverage in general, we chose a test problem from a FEM package, where we normalized the solution vector x in such a way, that the elements of x are in the interval [0.0,1.0). Results from 10,000 error injections are analyzed in the graph presented.

Figure 4 shows the error coverage by injecting bit errors into the solution vector x. The coverage is studied by varying the problem size and the iteration number where the error is injected. We recognized two effects. Firstly, if we increase the number of unknowns, we have to choose a larger tolerance to avoid false alarms caused by higher roundoff errors during checks. Secondly, if the error is not injected during the first

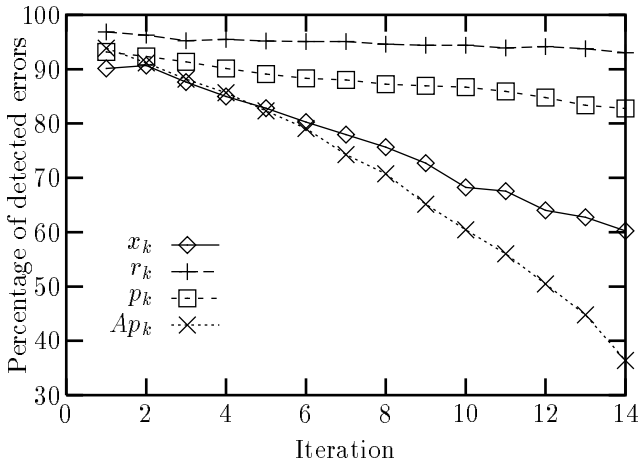


Figure 4: Error coverage of bit errors in all data structures for the CEDCG algorithm

iteration, the coverage rate decreases. This effect is caused by the dot–products, we use for computational checks. During the iteration, the solution vector becomes more and more exact. Only if a modification causes a huge change, is the error detected.

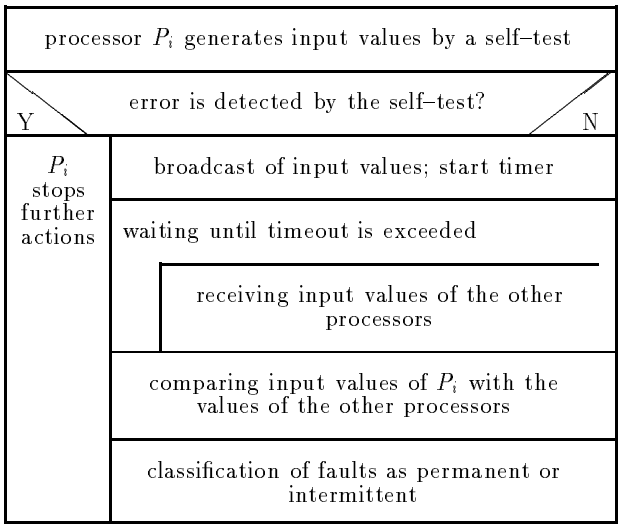
In addition to this, an error could happen during the computation of one of the other vectors. The effect of injecting errors into the other data structures, i.e. p_k, q_k, and r_k (k is the number of the iteration) is described in Figure 4. Injecting errors into vector q_k shows the same tendency as in x_k. In contrast to this, the error coverage is higher for errors injected into r_k and p_k.

In the presence of finite precision arithmetic, any high level encoding defined on the data will have incomplete coverage. However, this technique detects all errors which could cause a failure. This is an easy way to achieve error detection in an existing multiprocessor without having any hardware support.

4.4 Computational load in an error event

In this subsection we give the computational load of the fault tolerance services in an error event. We investigate in the load of diagnosis and neglect rollback recovery load, because rollback recovery load depends on the checkpointing interval and amount of data to be stored.

To illustrate the load of the diagnosis in details, the structure chart of the diagnosis algorithm is shown in Algorithm 4. Within it, three parts can be distinguished: the generation of input values for the comparison, the distribution of the input values, and the analyzes of the comparison results.



Algorithm 4: Diagnosis algorithm

In the first part of the algorithm, the processors generate input values for the comparison. The input values can be processed by different methods, such as by a hardware self–test program or by reading state information of the processor. In our algorithm, a hardware self–test program is used, whereas input values of a faulty processor are arbitrary. The input values are generated by self–test programs by each processor only once [11, 3].

In the second part of the algorithm, the generated input values are distributed to all other processors. For distribution, a fault–tolerant broadcast is used which fulfills the following assumptions: no messages may be lost or falsified; if there is a communication path between fault–free processors, it will be found. Furthermore, the time for the broadcast is limited by a timeout which is equal to the duration of the processing phase in the scientific application [4]. At the end of the second part of the algorithm, all fault–free processors have received the input values from all connected fault–free processors and some messages from the faulty processors.

The last part of the algorithm analyzes the input values. Firstly, the received input values are compared with the input values of processor P_i. Depending on the comparison, the fault vector is determined.

Considering our algorithm, it is obvious that the load of the diagnosis is mainly determined by the number of messages. Measurements of the load on the multiprocessors Suprenum and Meiko are shown in Figure 5. Self–test and analysis of the input values do not

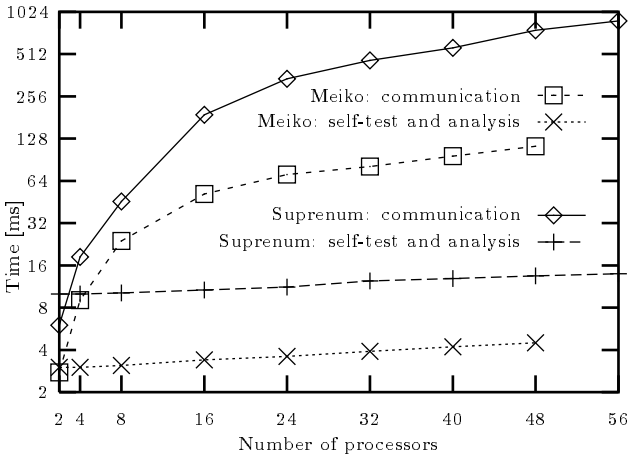


Figure 5: Load of the diagnosis algorithm

cause large computational load. The load of the self–test is fixed, whereas the load for the analysis grows with the number of processors. Further, the load of self-tests and analysis is considerably less than of communication, since number of messages increases by a power of two. Other off–line diagnosis algorithms generate the same number of messages. These algorithms cause much more overhead by tests and diagnosis protocols for distributing the diagnosis information. We see in Figure 5 that communication on the Suprenum has more load than on the Meiko.

Finally it has to be noted that a great number of messages has to be sent in large multiprocessors. But the number can be reduced by partitioning the multiprocessor [10]. Due to the strategy of the presented diagnosis algorithm, an efficient scheme working on partitions was developed.

5 Conclusion

We presented a scheme for developing fault–tolerant applications on arbitrary multiprocessors. Our run–time environment for fault–tolerant programming (PICLFT) was studied on two multiprocessors. All fault tolerance services of PICLFT meet the requirements of the algorithm–based approach. Furthermore, interactions between algorithm–based error detection, fault diagnosis, and the rest of the error processing were considered in detail from the application point of view.

As a result, we obtained an efficient fault diagnosis, which determine all connected fault-free processors. Efficiency of the diagnosis was shown by measurements. Furthermore, effectiveness of error detection was validated by measurements of run–time overhead and error coverage on the example of a concurrent error detecting CG algorithm. The presented scheme is an efficient way to program a fault–tolerant application on a multiprocessor without any fault tolerance mechanisms in hardware.

References

[1] C. Aykanat and F. Ozguner. A Concurrent Error Detecting Conjugate Gradient Algorithm on a Hypercube Multiprocessor. *IEEE Transactions on Computers*, pages 204–209, 1987.

[2] S. Ceri and G. Pelagatti. *Distributed Databases – Principles and Systems*. McGraw–Hill, 1984.

[3] P. Ciompi, F. Grandoni, and L. Simoncini. Distributed Diagnosis in Multiprocessor System: The MuTeam Approach. In *FTCS 11*, pages 25–29. IEEE Computer society press, 1981.

[4] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *FTCS 15*, pages 200–206, 1985.

[5] G. Geist, M. Heath, B. Peyton, and P. Worley. A Portable Instrumented Communication Library, C Reference Manual. Technical Report ORNL/TM–11133, Oak Ridge National Laboratory, 1990.

[6] C. Hoare. Communicating Sequential Processes. *Communications of ACM*, pages 666–667, 1978.

[7] K. Huang and J. Abraham. Algorithm–Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, pages 518–528, 1984.

[8] J. Jou and J. Abraham. Fault-Tolerant FFT Networks. *IEEE Transactions on Computers*, pages 548–561, 1984.

[9] L. Laranjeira, M. Malek, and R. Jenevein. On Tolerating Faults in Naturally Redundant Algorithms. In *Proc. Tenth Symposium on Reliable Distributed Systems*, pages 122–129, 1991.

[10] J. Maeng and M. Malek. Partitioning of Large Multicomputer Systems for Efficient Fault-Diagnosis. In *FTCS 12*, pages 1059–1063. IEEE Computer Society Press, 1982.

[11] S. Rangarajan and D. Fussell. A Probabilistic Method for Fault Diagnosis of Multiprocessor Systems. In *FTCS 18*, pages 278–283. IEEE Computer Society Press, 1988.

[12] S. Shrivastava. *Reliable Computer Systems*. Springer, 1985.