On Integrating Error Detection into a Fault Diagnosis Algorithm for Massively Parallel Computers

Jörn Altmann‡, Tamás Bartha†, András Pataricza†

† Department of Measurement and Instrumentation Engineering
Technical University of Budapest
Müegyetem rkp. 9, H-1521 Budapest, Hungary
email: pataric@mmt.bme.hu

‡ Department of Computer Science (IMMD) III
University of Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
email: jnaltman@informatik.uni-erlangen.de

Abstract

Scalable fault diagnosis is necessary for constructing fault tolerance mechanisms in large massively parallel multiprocessor systems. The diagnosis algorithm must operate efficiently even if the system consists of several thousand processors. In this paper we introduce an event-driven, distributed system-level diagnosis algorithm. It uses a small number of messages and is based on a general diagnosis model without the limitation of the number of simultaneously existing faults (an important requirement for massively parallel computers). The algorithm integrates both error detection techniques like <I'm alive> messages, and built in hardware mechanisms. The structure of the implemented algorithm is presented, and the essential program modules are described. The paper also discusses the use of test results generated by error detection mechanisms for fault localization. Measurement results illustrate the effect of the diagnosis algorithm, in particular the error detection mechanism by <I'm alive> messages, on the application performance.

Keywords: Error detection, distributed diagnosis, syndrome decoding, massively parallel systems

1 Introduction

The production cost of complex, highly integrated electronic components is decreasing due to the development of manufacturing technology. As a result, massively *parallel multicomputers*, capable of operating simultaneously several thousand processing elements (*PEs*), are gaining importance in computation-intensive scientific and technical applications. Beside the huge processing capacity achieved by utilizing massively parallel architectures, reliable operation over a long time period is also a crucial requirement. The large number of processors of such systems increases the probability of faults. Thus, the aim of fault tolerance is to ensure the specified operation in spite of faults by preventing detected errors from becoming failures [11].

In design and application of massively parallel computers *scalability* is a significant requirement. A multiprocessor system is called scalable, when extending it with new resources performance increases proportionally. Due to this requirement, centralized devices would limit the number of PEs. Thus, like other functions of the system, diagnosis must be *distributed* as well by using the PEs themselves for determining the system fault status: this approach is known as *distributed fault tolerance* [13][14][16]. In recent years several improvements for distributed diagnosis algorithms were published [3][7][10].

The paper presents a distributed diagnosis algorithm which integrates *error detection* mechanisms and minimizes the number of diagnostic messages. The aim of the algorithm is to generate a correct diagnostic image in every fault-free processor. If the diagnosis is correct, the fault-free processors can logically disconnect the faulty units from the system by stopping the communication with them. Employing this method, the number of tolerable faults depends only on the properties of the system interconnection topology.

The algorithm was developed for the Parsytec GCel[1]. This computer incorporates all features of a massively parallel multiprocessor, like scalability, regular distributed system structure, and a large number of PEs. Scalability is achieved by extending the hardware in units of 16 processors (called clusters) up to 16'384 processors in its full configuration [17]. The PEs (INMOS T805 transputers) are interconnected by a two-dimensional grid (see *Fig. 1*).

1. Supported by the EU (European Unit) as part of the Esprit Project 6731, Fault Tolerance for Massively Parallel Systems, and the Hungarian-German Joint Scientific Research Project #70 with additional support from OTKA-F007414.

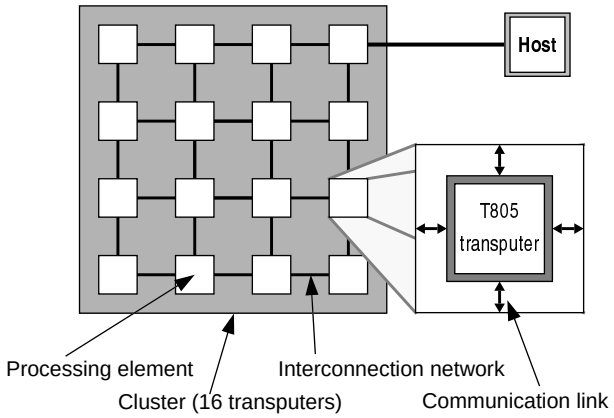

Fig. 1. Structure of the Parsytec GCel

The following sections explain how the diagnosis algorithm fulfils the requirements from a practical view. In *Section 2*, we describe the diagnosis model of the algorithm. The structure of the diagnosis algorithm and the proof of correctness are presented in *Section 3*. The implementation aspects of the algorithm are introduced in *Section 4*, where the essential program modules are explained in detail. *Section 5.* deals with measurements results, which illustrate the impact of these fault tolerance techniques on application performance.

2 Diagnosis model

The application of the developed distributed system-level diagnosis algorithm (described in *Section 3*) requires the following conditions to be fulfilled:

- **Individual and incomplete tests**. The algorithm treats the processing elements as "intelligent units" performing tests (of less than 100% error coverage) on units directly accessible via a communication link. Since tests covering every possible errors in such complex components as the modern processors are practically impossible to implement, the proposed testing mechanism only detects:
  - cpu errors by self-test,
  - crashs of processors and links,
  - errors (e.g., data or control structure errors) on application level detected by application-dependant test.

  Tests are independently performed on each processor. There is no explicit request message from a tester processor to a neighboring processor for performing a test. Results obtained by self-tests are sent by fault-free nodes within a predefined time-out limit to all neighboring processors (*<I'm alive>* messages). On receiving a message, each neighbor compares the received self-test result with the (saved or processed) local reference values. Therefore, the algorithm is prevented from a deadlock due to lost messages and test requests [8], and reduces the number of required tests [19].

  Only the normal interconnection facilities may be used for testing purposes. All messages are assumed to be protected by error-correcting encodes, which serves as an additional test for both the neighboring processor and the communication link connecting the PE with its neighbor. Consequently, an error in the communication will also result in a bad test outcome, which supports the diagnosis of the interconnection network as well.
- **Symmetric test invalidation**. The algorithm uses the symmetric test invalidation model (*PMC*) introduced by Preparata et al. [18]. In this model, a fault-free tester always determines the condition of the device under test (*DUT*) correctly: the result is 0 if the test passes, 1 if the test fails. A test performed by a faulty tester may result in an arbitrary outcome. Since such test results may not correspond to the actual fault state of the DUT, they must be left out of consideration. Note, that the PMC model is the most pessimistic test invalidation model, applying the highest degree of diagnostic uncertainty in the faulty case. At the same token it is the most general one. Thus, incorporating the PMC model the algorithm is applicable in systems of other fault models as well.
- **Diagnosability**. The majority of the diagnosis algorithms introduces an upper limit on the number of simultaneously existing faults in order to simplify the handling of uncertainty originating from the pessimistic test invalidation model. The underlying assumption of this *t-limit* is that a small number of faults are more likely to occur in a properly designed multiprocessor system if the individual faults are independent, thus, uncorrelated. The t-limit is the largest number of arbitrary located faults for which a proper diagnosis is always assured (e.g., for the two-dimensional grid the t-limit is as low as 2). Note, that the t-limit is a *worst-case* diagnostic measure, in most situations it provides too pessimistic estimation [11][15]. For this reason, we introduce a model which supports the diagnosis of an arbitrary number of faulty processors.

  Let us consider now the *interconnection graph* $G = (N, A)$ of the system. Its nodes $u_i \in N$ ($i = 1 \ldots n$, where n denotes the total number of processing elements in the system) correspond to the processors. A directed arc $(u_i, u_j) \in A$ exists between two nodes if direct communication is possible between the corresponding processors in the given direction. Diagnosis of the whole system is possible until G remains *strongly connected*. However, faulty PEs or links can cut the in-

terconnection graph into isolated subgraphs. The case of a system with three isolated connected subgraphs is presented in *Fig. 2*. If this happens, diagnosis is restricted to the group of fault-free processors located in the same connected subgraph. Since subgraphs are isolated, the host computer can access only the PEs located in the connected subgraph containing the host link (see subgraph 1 in Fig. 2).

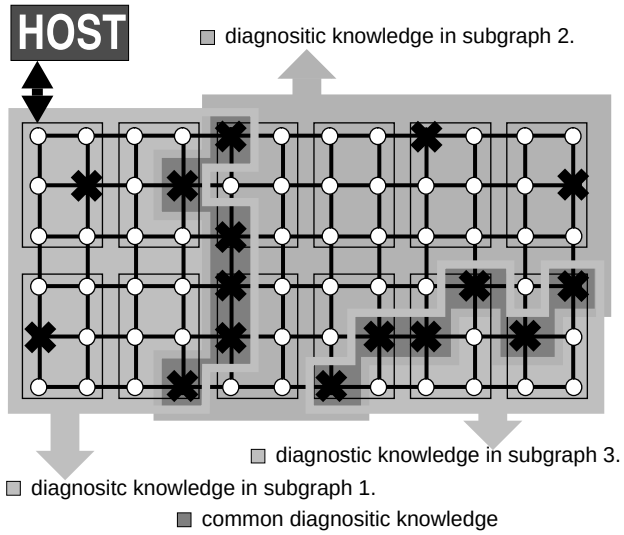

Fig. 2. Diagnostic knowledge in different subgraphs

Messages exchanged between non-neighboring nodes are transferred via paths of processors and links. Faulty processors or communication links cannot be included in this path, because they would block the correct information flow or make it unreliable. Therefore, a set of faulty processors and links may *isolate* a group of fault-free processors. In such cases the diagnostic image of the whole system is incomplete, but a complete diagnostic image within connected subgraphs can be generated.

The algorithm does not require the limitation of the number of faults, rather it includes only the nodes in the same connected subgraph in diagnosis, classifying the state of other processors as unknown. Each fault-free PE diagnoses its own subgraph, as indicated in *Fig. 2*. Unknown processors are identified by detecting the isolating *barriers* made of faulty processors [4].

Although the host computer can access only the processors in its own subgraph, the diagnosis procedure running in other regions is also important. Due to the distributed nature of the algorithm, every processor maintains a consistent local diagnostic image. When a faulty node within a barrier is repaired or replaced, two isolated regions will be joined together. In such cases the two different local diagnostic images can be combined in order to obtain a consistent diagnostic image of the joined subgraphs.

- **Determining the real message order**. The arrival of messages at a processor will not always correspond to the order of their creation, due to communication delays. Such a situation can occur if a PE becomes faulty during the testing process. Then, messages received in a wrong order will cause the algorithm to generate an incorrect diagnosis. To avoid this, logical time-stamps related to test execution must be attached to the diagnostic messages, and the real order of messages must be determined using a distributed event-ordering procedure [14].

3 Structure of the diagnosis algorithm

The diagnostic process described below is almost identical for the different processors (only the inhomogeneity at the grid borders has to be taken into account), so each processor can use the same diagnosis algorithm. The algorithm consists of two phases: an *initial* and a *working* phase. Two observations motivated the splitting of the algorithm:

- Current peaks during power on/off may damage the electronic components of the system. Hence, the majority of faults occurs (or already exists) in the initial phase. Moreover, typically the power-on tests serve as major means for the detection of permanent faults. The failure rate is expected to be lower during further operation.
- Processors do not have any information on the fault state of other components in the initial phase of the diagnosis algorithm (i.e., communication links and other PEs). All processors have to be tested once to generate the initial diagnosis image. Later the system fault state does not change significantly compared to the first diagnostic image due to low fault rate. For this reason, a considerable overhead in communication and administration can be saved by calculating and distributing only the *differences* between the current (diagnosed) fault state and the stored diagnostic image.

3.1 Initial phase

Inter-processor communication starts after the local diagnostic images has been generated by testing the *neighbouring* processors, and it continues until each fault-free processor has received diagnostic information from all the others in the same connected subgraph. Every PE sends the local test results to its neighbors, and further on it receives and forwards the messages sent by other units. PEs maintain a list of the processors from which they have not received a diagnostic image yet, in order to evaluate the termination

criterion of communication. For this purpose they must also discover which nodes are accessible via a path of fault-free processors and links.

There are processors that meet the termination condition before others, due to the inherent inhomogeneity of the two-dimensional grid topology (i.e., processors located on grid edges do not have certain neighbors) and obstacles in communication formed by faulty components [5]. These processors must inform their neighbors before termination, otherwise the neighbors would possibly try to communicate with the already terminated processor. To avoid this *deadlock* situation, the algorithm has a *termination* period. Ready-to-terminate PEs send special messages to their neighbors during this period, so the still active nodes will not communicate with these PEs further on [4]. After the information is sent to each neighbor, processors decode the received syndromes using the algorithm described in *Section 4.4*, thus completing the initial phase.

For the transputer system, the initial phase of the diagnosis algorithm is integrated into the boot and loading process.

3.2 Working phase

The algorithm continues with the working phase after finishing the initial phase. All processors have an initial, *system-level diagnostic image* at this point. Let u_i be an arbitrary processor in the multiprocessor system which periodically tests its neighboring processors u_j, u_k, u_l, u_m. The test compares the values (results of self-test programs) received in <I'm alive> messages from the neighbors with stored or processed local reference values. Hence, an error is detected in four different ways:

- a <I'm alive> message does not arrive within the predefined time-out interval,
- a <I'm alive> message contains incorrect error correcting code,
- the value of the <I'm alive> message does not match the local reference value.

Assume, that the current result of comparison shows that u_k, u_l, u_m are fault-free and u_j is faulty. Then the local test result is $a_{i,k} = a_{i,l} = a_{i,m} = 0$, but $a_{i,j} = 1$. The processor u_i compares the obtained local test results to the stored local diagnostic image during further operation. If it finds a difference indicating a new fault occurrence, it invokes exception handling. Assume, that the local test result $a_{i,j} = 1$ indicates a new fault. In this case, processor u_i starts to broadcast messages containing the local test result $a_{i,j} = 1$. Now, two different cases can be distinguished regarding the state of the tester:

i) processor u_i is fault-free or

ii) processor u_i is faulty.

In the first case processor u_i broadcasts a correct local test result to each neighbors indicated as fault-free in the current local diagnostic image. If there is no other new faulty processor except u_j, these processors are really fault-free. Because every fault-free processor diagnose their neighbors correctly each faulty processor can be isolated; broadcast messages are sent only between fault-free neighbors. Hence, processor u_i sends its local test results only to processors u_k, u_l, u_m. The same procedure continues until all fault-free processors within a connected subgraph receive the local test result of processor u_i.

If there are multiple new faulty processors in the system, it may happen that the local test result becomes corrupted. If this remains undetected by the coding of messages an incorrect local test result is broadcasted, falsely indicating the occurrence of a single processor fault in the system. Although some of the fault-free processors now receive an incorrect local test result, it does not result in a wrong fault localization as it will be shown in *Section 4.4*. The reason is that only changes within the system are reported by the local test results, not the fault state of a processor.

In the second case processor u_i sends either a correct or a wrong local test result to some of its neighbors. On the one hand, if the neighbor is faulty, the communication between these two faulty processors has no impact on the correctness of the diagnosis. On the other hand, if the neighbor is fault-free and the message format from processor u_i is correct, a wrong local test result will be broadcasted. The same situation as described above in the previous paragraph.

After all, every fault-free processors within a connected subgraph have received the message regarding the local test result. After receiving the first message about the local test result, the processor stops the application as soon as possible and waits for further local test results from other neighbors of the faulty processor for a fixed *time interval (time-out termination criterion)* [9]. Furthermore, all fault-free neighbors will initiate further, more exact tests on the probably faulty processor and on the processor initiating the exception handling.

All processors can determine the fault state of the whole system by processing the incoming local test results (*syndrome decoding* process), unless some faulty processors cut the system into different isolated subgraphs (*Section 4.4*).

The working phase of the diagnosis algorithm is *event-driven*, as both the local test result distribution and the syndrome decoding process are activated by the changes in the local diagnostic image [6].

4 The implemented diagnosis algorithm

The implementation details of the initial phase of the algorithm are not discussed in the paper, as the boot process

(and thus the initial phase) has no impact on the application performance. During this phase no application is running, so efficiency requirements do not have high priority.

The realization of the working phase can be done in different ways, depending on the splitting of the processing power between the diagnostic process and the running application. In the following the implementation of the algorithm will be described. Alternative approaches to the testing mechanism of neighboring processors, termination rules, distribution of local test results, and processing of diagnostic information are presented to show that the algorithm can be adapted to several systems and requirements.

The main structure of the implementation for the working phase of the algorithm is shown in *Fig. 3*. If no fault event is detected, the algorithm periodically tests the neighboring processors. Testing is accomplished by assigning independent modules to each tested unit [2].

If the tests detect an error in one of the neighboring processors, exception handling is invoked by issuing an error indication from the corresponding *testing* module to the *local diagnosis* module. The local diagnosis module gives control to the *supervisor* module, which handles the exceptions caused by the detected error. The supervisor module activates the modules responsible for *terminating the current application*, for *distribution of the local test results*, and for *processing of the diagnostic information* (as described in *Section 4.4*) [1].

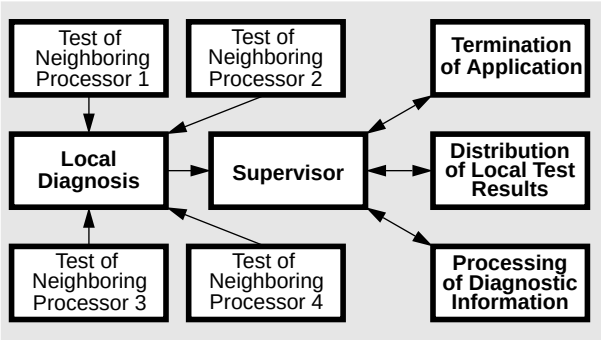

Fig. 3. Main modules of the implemented algorithm

4.1 Test of neighboring processors

Each testing module is comprised of three threads: one for receiving local test results from the neighboring processors, one for sending such messages (<I'm alive> message), and one for evaluating the result (i.e., verifying whether the responses are delayed or incorrect), respectively. If the evaluation indicates a faulty behavior of the neighboring processor, the thread sends an error message to the local diagnosis module.

This <I'm alive> testing mechanism offers a possibility to control the testing related run-time overhead. On the one hand very precise and thoroughgoing self-tests can be used resulting in a decreased application performance due to the more intensive diagnosis process. Such self-tests can take two forms: either realized in software or in hardware.

On the other hand, the use of <I'm alive> messages indicating the alive or dead state of a processor reduces the requirements of processing capacity to a fraction, thus yielding more computational power to the application. Although this kind of tests is easy to process, it can only detect processor and link crashs, and more post-processing is required later.

However, since the application is quickly stopped after error detection, there is a sufficient time remaining for more finely granulated tests and post-processing in a subsequent separate testing phase.

4.2 Terminating the application

The function of this module is to interrupt the execution of the application on all the PEs as soon as possible. This is necessary for the prevention of error dissemination and to decrease the fault latency in the multiprocessor. If the application is quickly suspended, the probability of error dissemination is reduced, because no further communication - with the exception of diagnostic information transfer - will take place.

For quick termination of the application, the module initiates a fast broadcast. The broadcast messages are received at every node by the local diagnosis module, which initiates immediately exception handling. No specific routing mechanism is required for the implementation of the fast broadcast, as the existing routing mechanism of the Parsytec GCel system extended with a high-level, fault-tolerant communication protocol is fully sufficient. The broadcast is based on flooding the multiprocessor with a so-called *stop message* indicating the occurrence of an error. Each processor sends this message to all of its fault-free neighbors. The neighbors forward the message to their neighbors (excluding the sender), continuing this process until the message arrives at every accessible fault-free node. The advantage of using flooding is its easy algorithm and its inherent fault-tolerant behavior; all fault-free processors within a connected subgraph are reached.

4.3 Distribution of local test results

The module for distribution of the local test results is activated after terminating the application. At first, only the neighboring processors of the faulty processor start a separate testing phase, executing fault localization tests. The assumed causes are faulty links and faulty processor components. These additional tests even assure the classification of faults as *temporary* or *permanent*. The outcome of these tests as well as the tests results obtained by the error

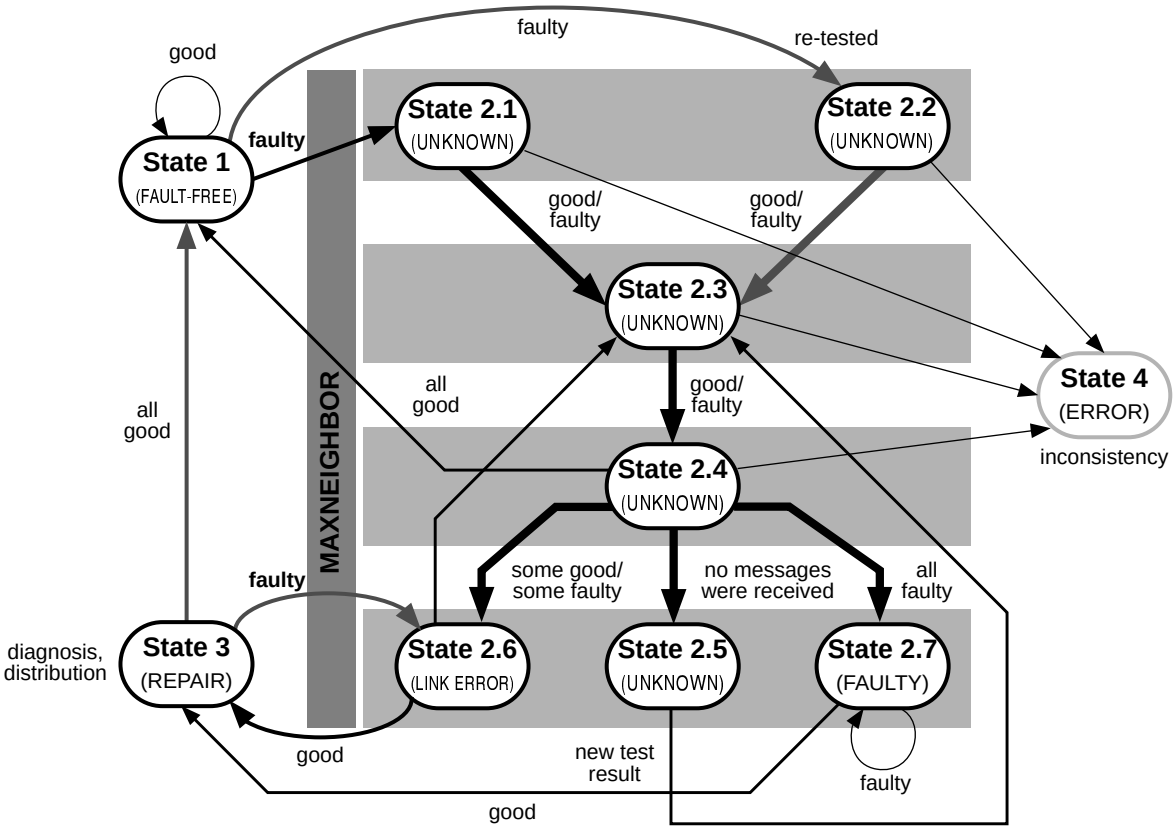

Fig. 4. State diagram of the syndrome decoding algorithm

detection mechanism constitute the local test results.

The distribution module transfers the local test results to the supervisor module of each processor using the fault-tolerant flooding broadcast.

Different criteria can be used for terminating the distribution phase. Our first implemented criterion was to wait until all the local test results from each tester of the faulty node have been received, but we found that this method is not robust against errors during the diagnosis (e.g., lost messages due to node failure). A time-out criterion is used for elimination of this lack of robustness [8]. The distribution process waits for a certain time interval, in which all of the local test results must be received. The main advantages of this method are its safety and simplicity, but the optimal time-out limit must be estimated in the design phase.

4.4 Fault localization

It is necessary to analyze the obtained local test results in order to determine the fault state of each accessible unit of the system. This task is accomplished by a syndrome decoding algorithm (defined by the state diagram in *Fig. 4*). Syndrome decoding is invoked by the supervisor module on receiving a new local test result. This way the local test result distribution and the fault localization modules are executed alternatively. Therefore, even if the time-out limit used in the distribution process is not optimal, the processor will not be idle.

Fig. 4 describes the different states of the classification process of a unit under diagnosis (*UUD*). Three possible fault classifications are taken into consideration: the UUD can be *faulty*, *fault-free*, or a *link fault* in the communication link between the testing processor and the UUD may occur. (Note, that actually there is a fourth classification for the inaccessible PEs: *unknown*.) In each state (represented by nodes in the graph) the actual classification is shown in parentheses. Transitions between states are indicated by directed arcs in the graph. A transition is enabled by receiving a local test result.

There are four possible starting states in the diagnosis of a UUD: State 1, State 2.5, State 2.6, and State 2.7; depending on the classification created in the initial phase of the diagnosis algorithm. If the UUD was found to be *fault-free* in the initial phase (i.e., State 1 is the starting state), this classification remains unaltered, until a message indicating a

fault in the UUD has been received from one of its neighboring processors. Two different activities must be performed after receiving a fault indication, depending on the topological relationship between the diagnosing PE and the UUD:

- if the PE is one of the neighbors of the UUD (i.e., it is assigned to test it), the PE executes a new test, then it broadcasts the local test result in the system, as well as the received test result. This procedure assures that each processor will receive an up-to-date test result from all the testers of the UUD. These activities are performed in State 2.2.
- if the PE is not one of the neighbors (State 2.1), it only forwards the received local test result to its neighbors and enters the next state.

Test results from the testers of the UUD are obtained and then analyzed during the subsequent four state classes (from State 2.1 to State 2.7, marked by gray background). The number of these state classes (indicated as MAXNEIGHBOR in *Fig. 4*) equals to the number of neighboring (testing) processors. Transitions are independent of the received test results, as the purpose of these states is to obtain all information from the tester processors. A time-out mechanism is used in receiving the local test result messages. During the obtaining of local test results, the classification of the UUD is set to *unknown*. Decision is made when all local test results are received, or the time-out period used in the distribution phase expires. Local test results not received within the time-out period due to an extremely large communication delay are assumed to be missing.

Missing messages which generate diagnostic inconsistency are taken into consideration as indications of a new error occurrence during diagnosis (State 4). In this case the diagnosing processor broadcasts a fault indication in the system. All processors receiving the message will begin to test their neighbors.

If no messages are received within the time-out limit, the UUD is inaccessible, so its classification remains unknown (State 2.5). This classification is valid, till a new local test result related to the UUD is received from one of its testers later on, then the algorithm continues the diagnosis of the unit in State 2.3.

If every tester found the UUD to be faulty, then the unit itself is considered to be *faulty* (in State 2.7). A processor crash, and the simultaneous fault in all of its communication links are equivalent as they produce identical syndromes. Such syndromes are represented as processor faults.

If a tester processors produces a bad test outcome for its neighbor and vice versa, then the corresponding communication links between these testers are assumed to be faulty (classification *link fault* in State 2.6).

State 3 provides the possibility of taking on-line repairs into consideration. Here an extra diagnostic process is required to assure consistency between the diagnostic images stored in processors belonging to different connected subgraphs. If faulty links or nodes - previously isolating two or more connected subgraphs - have repaired, the subgraphs are joined, but the diagnostic knowledge (described in *Section 2*) of the processors remains potentially different. For this reason a special broadcast procedure is performed between the nodes of previously isolated subgraphs to merge the various diagnostic knowledge into one.

5. Measurements

As stated in *Section 3*, faults seldom occur during the operation of the system. In a fault-free system the event-driven diagnosis algorithm performs only error detection. Thus, the <I'm alive> testing mechanism has the largest impact on the application performance.We have examined the run-time overhead related to testing. The minimal run-time overhead can be achieved using the <I'm alive> message. Therefore, this testing mechanism was measured.

5.1 Application run-time overhead

As the application run-time overhead is an important criterion for the evaluation of a fault tolerance mechanism, we measured the application run-time overhead by running a benchmark-like practical application (*Ising*) and the diagnosis algorithm concurrently on each processor. The Ising application calculates the spin of electrons in a gas at various temperatures. In *Fig. 5* the run-time overhead is displayed as a function of the time between two consecutive <I'm alive> messages.

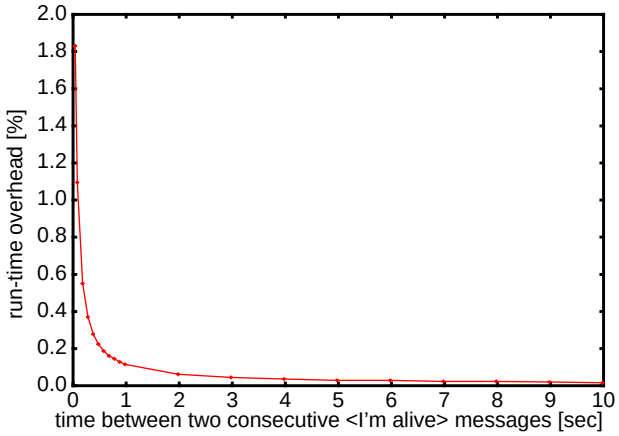

Fig. 5. Run-time overhead caused by the <I'm alive> testing mechanism (Ising application)

The overhead is approximately inversely proportional to the time between two consecutive <I'm alive> messages.

The sending of <I'm alive> messages has a very little impact on the application run-time, if the interval between two messages is longer than one second. If the <I'm alive> messages are sent in every 500 milliseconds, the overhead is larger, but does not exceed 0,2 percent.

The reason for the coarse of the curve is the increasing usage of computational power for receiving, sending, and evaluating <I'm alive> messages by the <I'm alive> mechanism, if the time interval between the two consecutive <I'm alive> messages decreases.

Furthermore, performance measurements were made with various other benchmark-like applications, differing in the intensity of communication. The shape of curves describing the overhead corresponding to the different applications are similar to the curve in *Fig. 5*. The collection of these curves can be represented by the grey marked region in *Fig. 6*, bounded by two curves of the application overhead. The curve at the lower border of the marked region represents the run-time overhead of the *whetstone* benchmark program, which does not communicate. The curve at the upper border describes the run-time overhead of a dummy program which performs only communication.

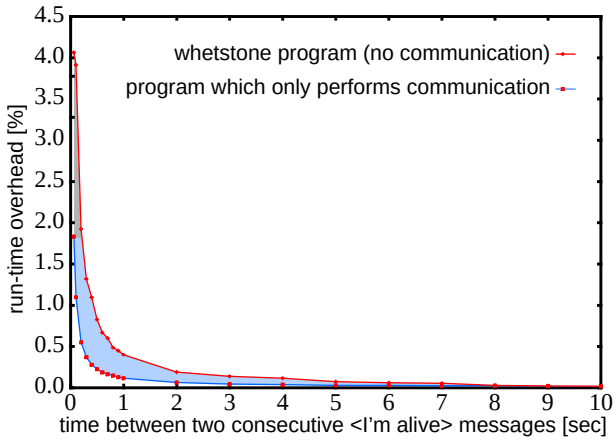

Fig. 6. Run-time overhead caused by the <I'm alive> mechanism for various applications

The reason for the difference in the run-time overhead between the whetstone and the dummy application is the load of the communication network.

5.2 Impact of the application on the <I'm alive> message testing mechanism

Additionally to the influence of the <I'm alive> mechanism on the application performance, the inverse effect is another important criterion for the assessment of diagnosis software. Since the same communication network is used for sending the <I'm alive> messages as well as the application messages, and since both kind of messages are sent with the same priority on the multiprocessor Parsytec GCel, the impact of the application messages on the <I'm alive> message testing mechanism has to be examined.

The time between two consecutive <I'm alive> messages has been measured. This time is composed of the *<I'm alive> time interval* and the communication time needed for sending this <I'm alive> message to the neighboring processor. The <I'm alive> time interval was chosen to be one second to have a small run-time overhead as indicated in *Fig. 6*.

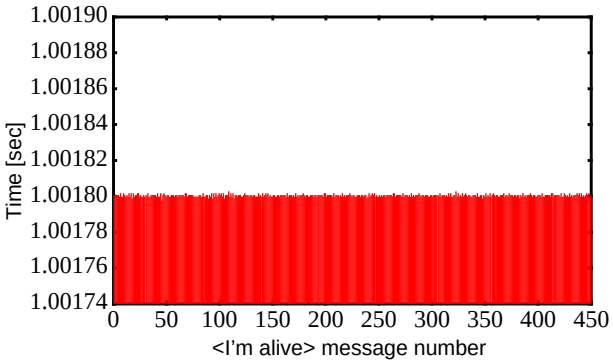

Fig. 7. Time between two consecutive <I'm alive> messages running the whetstone application

For the whetstone application (*Fig. 7)*, the time between two consecutive <I'm alive> messages is nearly constant, with a variance only in the microsecond range. The average time is 1.00180 seconds. That could be expected because the whetstone application does not communicate. Therefore, the average time will be used as the base for the comparison with the following measurements.

In a dummy application exclusively performing communication, the average time is 1.00187 seconds (*Fig. 8*). This shows that the time between two consecutive <I'm alive> messages is only a little bit larger running the dummy application than for the whetstone benchmark program.

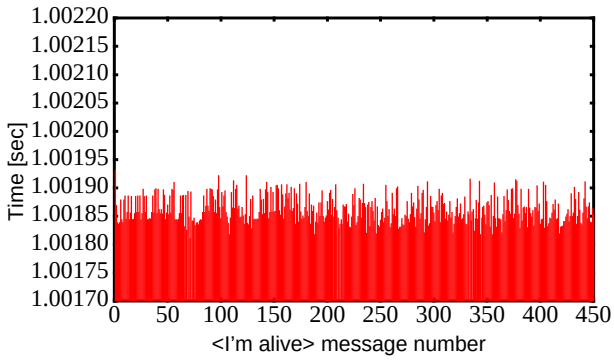

Fig. 8. Time between two consecutive <I'm alive> messages running an application which only communicates

Furthermore, even the measured time values vary not much more than the values displayed in *Fig. 7*. They are in the range between 1.00182 and 1.00193 seconds (see *Fig. 8*). Considering these results, it can be stated that the messages sent by the applications have a moderate impact on the <I'm alive> message testing mechanism. An upper limit for the delay of the messages can easily be found. For the example given, the maximal delay is 0.00193 seconds which can be recognized in *Fig. 8*.

5.3 Fault latency

The *fault latency* (T_l) in a processor is defined as the time interval between the fault occurrence and the error detection by a neighboring processor (tester). The latency depends on the time (T_c) between two consecutive checks of incoming messages (*check time interval*) and the <I'm alive> time interval (T_{ia}). $T_{ia} < T_c$ The model of fault latency is shown in Fig. 9.

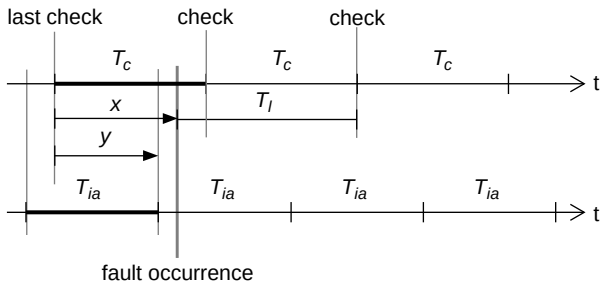

Fig. 9 Model of fault latency

As the checking process in the tester processor runs asynchronously to the <I'm alive> message generating process in the processor under test, we assume that the remaining time y from the last check to the next <I'm alive> message is an equiprobable distributed random variable.

Additionally, as faults are assumed to be uncorrelated with both the message sending and the checking process, the random variable x denoting the time between the last check and the occurrence of an error is assumed to be equal distributed, and the distributions of x and y are independent.

Then, two cases concerning the latency T_l have to be distinguished:

i) the latency T_l is $T_l = 2T_c - x$ and $0 \le x \le T_c$ or

ii) the latency T_l is $T_l = T_c - x$ and $0 \le x \le T_{ia}$.

The density functions for the two cases are:

case i)
$$f_i(x) = \begin{cases} x > T_{ia} & f_{i1}(x) = 1 \\ x \le T_{ia} & f_{i2}(x) = \dfrac{x}{T_{ia}} \end{cases}$$

case ii)
$$f_{ii}(x) = \begin{cases} x > T_{ia} & f_{ii1}(x) = 0 \\ x \le T_{ia} & f_{ii2}(x) = 1 - \dfrac{x}{T_{ia}} \end{cases}$$

The *theoretical mean fault latency* T_{ml} is calculated by $T_{ml} = T_{ml1} + T_{ml2}$, where T_{ml1} and T_{ml2} are:

$$T_{ml1} = \frac{1}{T_c} \int_0^{T_{ia}} \left((2T_c - x) \frac{x}{T_{ia}} + (T_c - x) \left(1 - \frac{x}{T_{ia}} \right) \right) dx$$

$$T_{ml2} = \frac{1}{T_c} \int_{T_{ia}}^{T_c} (2T_c - x)\, dx$$

After completing the above integration, the following formula results for the theoretical mean latency T_{ml}:

$$T_{ml} = \frac{3}{2} T_c - \frac{1}{2} T_{ia}$$

Assume an <I'm alive> message period time of 1.0 second and a check time interval of 1.1 second. Then, a mean latency T_{ml} of 1.15 seconds would result.

The model suggests, that the mean fault latency can be reduced in two ways. Firstly, the <I'm alive> message interval T_{ia} must be closely equal to the check time interval T_c. But here, the variance caused by the communication (ΔT_{max}) must be taken into account:

$$T_c \ge T_{ia} + \Delta T_{max}$$

Secondly, the check time interval (and so the <I'm alive> message interval) can be decreased. However, reducing T_c and T_{ia} the application run-time overhead will increase. Therefore, the trade-off between the reduction of the check time interval and the application run-time overhead has to be taken into consideration determining the *optimal mean latency.*

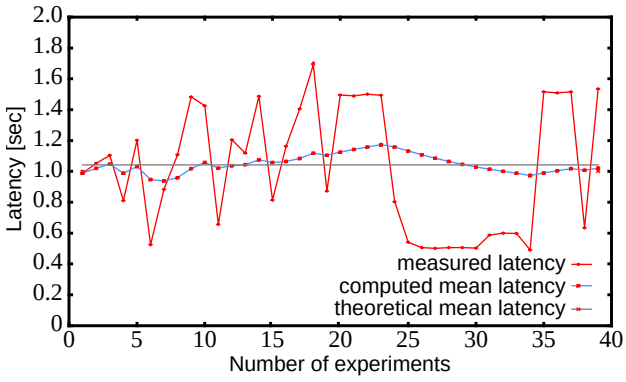

Fig. 10. Fault latency

The theoretical mean latency of the <I'm alive> message testing mechanism was computed using our model. Fault injection experiments were performed to validate the result. Permanent faults, always resulting in crash failure of a processor, were injected. The measured latency and its mean value (*computed mean latency*) for the above example are shown in *Fig. 10*.

The measured latency varies between 0.5 and 1.7 seconds. These values are in the theoretical latency range between 0.1 and 2.2 seconds. After 39 measurements the computed mean latency is close to its theoretical value.

5.4 Run-time of the broadcast needed for stopping the application

A important aspect of the diagnosis, as already mentioned in *Section 4*, is the time needed for stopping the application after the occurrence of an error. To reduce the latency in a multiprocessor system and the probability of error propagation, the mechanism for stopping the application has to work fast. That is achieved by a fast broadcast.

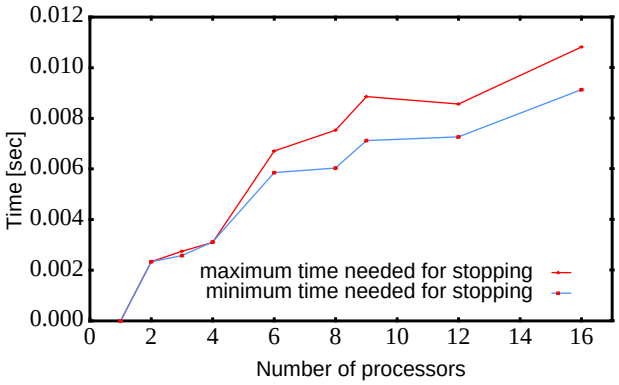

Fig. 11. Time needed for stopping the application

In *Fig. 11* run-time measurements of this broadcast are shown. The run-time depends on the grid structure and on the location in which the broadcast is initiated after the occurrence of an error. Therefore, in *Fig. 11* two curves of the run-time are given, the upper one describing the maximum run-time, the lower one describing the minimum run-time. Considering the dependencies of the run-time, it is obvious that the curves are nonlinear; the number of hops performed by the broadcast does not increase linearly on increasing the number of processors.

6 Conclusions

In this paper we introduced a new system-level diagnosis algorithm. The algorithm is distributed, which makes it applicable in scalable systems; and event-driven, thus it processes diagnostic information fast and efficiently, requiring small amount of communication and computation. Additionally, we concentrated on the relation between the tests for error detection and the tests for error localization.

The general structure of the algorithm consisting of two separate phases has been described. A new syndrome decoding method, which produces the diagnosis gradually was given. Furthermore, we presented an extended diagnosis model, which makes possible to obtain all accessible diagnostic information without limiting the number of tolerated faults within the system.

Additionally, we presented an implementation based on the algorithm which uses different tests for error detection and localization, using a separate testing phase after quick termination of the running application. It executes the local test result distribution and the syndrome decoding procedures alternatively, thus creating diagnostic images gradually, taking every test outcome into consideration during diagnosis. The implementation was examined, highlighting the advantages and disadvantages. Furthermore, we have proven the efficiency of our algorithm by some measurement results. The measurement results show that the testing causes only a small overhead.

References

[1] Altmann, J., "Diagnoseprotokolle in Multiprozessorsystemen," Diploma Work, University of Erlangen-Nürnberg, February 1993.

[2] Altmann, J., F. Balbach, and A. Hein, "An Approach for Hierarchical System Level Diagnosis of Massively Parallel Computers Combined With a Simulation-Based Method for Dependability Analysis," *IEEE 1st European Dependable Computing Conference*, Berlin, October 1994.

[3] Bagchi, A. and S. L. Hakimi, "An Optimal Algorithm for Distributed System-Level Diagnosis," *IEEE Proc. 21st Int. Symposium on Fault-Tolerant Computing*, pp. 214-221, Montreal, June 1991.

[4] Bartha, T., "Diagnostic Algorithms of Multiprocessor Systems," Diploma Work, Tech. University of Budapest, 1993.

[5] Behr., P. M., W. K. Giloi, and W. Schröder, "Synchronous versus Asynchronous Communication in High Performance Multicomputer Systems," *Aspects of Computation on Asynchronous Parallel Processors*, pp. 239-247, North-Holland, 1989.

[6] Bianchini, R. and R. Buskens, "An Adaptive Distributed System-Level Diagnosis Algorithm and its Implementation," *IEEE Proc. 21th Int. Symposium on Fault-Tolerant Computing*, pp. 222-229, Montreal, June 1991.

[7] Bianchini, R., R. Buskens, and M. Stahl, "On-Line Diagnosis in General Topology Networks," *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 114-121, Amherst, July 1992.

[8] Ciompi, P., F. Grandoni, and L. Simoncini, "Distributed Diagnosis in Multiprocessor System: The MuTeam approach," *IEEE Proc. 11th Int. Symposium on Fault-Tolerant Computing*, pp. 25-29, Portland, June 1981.

[9] Cristian, F., H. Aghili, and R. Strong, “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement,” *IEEE Proc. 15th Int. Symposium on Fault-Tolerant Computing*, pp. 200-206, Ann Harbor, June 1985.

[10] Dal Cin, M. and F. Florian, “Analysis of a Fault-Tolerant Distributed Diagnosis Algorithm,” *IEEE Proc. 15th Int. Symposium on Fault-Tolerant Computing*, pp. 159-164, Ann Harbor, June 1985.

[11] Dal Cin, M. and A. Pataricza, “Increasing Dependability in Multiprocessors,” *Proc. of the 8th Symp. on Microcomputer and Microprocessor App.* µ*P'94*, pp. 55-64, Budapest, October 1994.

[12] Kime, C. R., “System Diagnosis,” in *Fault-Tolerant Computing: Theory and Techniques*, Prentice-Hall, New York, pp. 577-623, 1985.

[13] Kuhl, J. G., and S. M. Reddy, “Distributed Fault-Tolerance for Large Multiprocessor Systems,” *ACM-Sigarch Newletter 8*, no. 3, pp. 23-30, 1980.

[14] Kuhl, J. G., S. M. Reddy, and S. H. Hosseini, “On Self Fault-Diagnosis of the Distributed Systems,” *IEEE Proc. 15th Int. Symposium on Fault-Tolerant Computing*, pp. 30-35, Ann Harbor, June 1985.

[15] Malek, M. and Y. Maeng, “Partitioning of Large Multicomputer Systems for Efficient Fault Diagnosis,” *IEEE Proc. 12th Intl. Symposium on Fault-Tolerant Computing*, pp. 341-348, Santa Monica, June 1982.

[16] Meyer, F.J. and G. Masson, “An Efficient Fault Diagnosis Algorithm for Symmetric Multiprocessor Architecture,” *IEEE Transaction on Computer*, vol. EC-27, pp. 1059-1063, November 1978.

[17] Parsytec Computer GmbH., “The Parsytec GCel Technical Summary,” Version 1.0, Aachen, 1991.

[18] Preparata, F., G. Metze, and R. Chien, “On the Connection Assignment Problem of Diagnosable Systems,” *IEEE Transaction on Computer*, vol. EC-16, no. 6, pp. 848-854, December 1967.

[19] Rangarajan, S. and D. Fussell, “A Probabilistic Method for Fault Diagnosis of Multiprocessor Systems,” *IEEE Proc. 18th Int. Symposium on Fault-Tolerant Computing*, pp. 278-283, 1988.