TRANSPUTER COMMUNICATIONS, VOL. 2(4), 241–257 (DECEMBER 1994)

Fault Tolerance in Massively Parallel Systems

G. DECONINCK, J. VOUNCKX, R. CUYVERS AND R. LAUWEREINS
Katholieke Universiteit Leuven, ESAT-ACCA Laboratory
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium
(email: Geert.Deconinck@esat.kuleuven.ac.be)
B. BIEKER, H. WILLEKE AND E. MAEHLE
Universität-GH Paderborn / Med. Universität zu Lübeck, Germany
A. HEIN, F. BALBACH, J. ALTMANN AND M. DAL CIN
F.A. Universität Erlangen-Nürnberg, Germany
H. MADEIRA AND J.G. SILVA
Universidade de Coimbra, Portugal
R. WAGNER AND G. VIEHÖVER
Parsytec Aachen, Germany

SUMMARY
In massively parallel systems (MPS), fault tolerance is indispensable to obtain proper completion of long-running computation-intensive applications. To achieve this at reasonable low cost, we present a global approach. A flexible and powerful backbone is provided through the combination of hardware and software error detection techniques, fault diagnosis and operator-site software together with reconfiguration of the system. Application recovery is based on checkpointing and rollback. The red line (i.e. applicability for a massively parallel system) comprises scalability as well as simplicity. A unifying system model is introduced that allows the mapping of a global concept for fault tolerance to a wide variety of MPS. The framework for implementation in an existing MPS is discussed.[1]

KEY WORDS fault tolerance; massively parallel system; error detection; fault diagnosis; backward error recovery; reconfiguration

1 THE NEED FOR FAULT TOLERANCE IN AN MPS

Although existing multiprocessors are growing into *massively* parallel systems, the demand for even more computation performance is still there, and many programs require days, weeks or months to execute. Though the probability of the failure of a single chip decreases (owing to better technologies), the statistical chance that one element in these large systems breaks down is not at all negligible (see Section 3). Nevertheless, a single defeat should not result in the failure of the complete system. Hence, *fault tolerance* is indispensable: knowing that faults will occur in an MPS, how can we tolerate them, without influencing the results or seriously degrading performance?

[1] This work is partially supported by the EU as ESPRIT project 6731 (FTMPS)

CCC 1070–454X/94/040241–17
©1994 by John Wiley & Sons, Ltd.
Received 15 November 1993
Revised 5 July 1995

- The control-net contains control processors that are shared among multiple applications, and others that are used for a single application. The former define the *global control-net*; the latter the *local control-net.*

Note that these data-net and control-net processors are logically different; on some architectures however, they map to the same physical processors.

A control processor on the local control-net defines a *local control entity* among the data-net processors. This control entity contains the data processors that are considered as being failed, when the involved control processor is faulty. Analogously, a global control processor defines a *global control entity.*

In this project, implemented on machines that conform to the USM, we further assume that the MPS is a 2- or 3-dimensional mesh, which is divided into partitions. Every partition is devoted to a single application or user (hence space-sharing, no time-sharing). The MPS is connected via host links to (possibly several) host machines.

Examples of machines that fit in this USM approach are Parsytec's GC machine (which would have been built with T9000 transputers and C104 routeing switches), the Parsytec GC_{el} and the Parsytec (Power)Xplorer series. The Connection Machine CM-5 also provides an explicit control-net. Also the Intel Paragon XP/S fits into this USM approach.

3 MOTIVATION AND THE OVERALL SOFTWARE STRUCTURE

In making a system reliable, a first approach is to prevent faults from occurring. This means that all aspects of the system must be designed, engineered and manufactured with reliability in mind. A design that provides excellent engineering solutions for cooling, packaging and system-wide interconnections is mandatory for MPS and improves their reliability at a relatively low cost.

Nevertheless faults may occur quite often in a *massively* parallel system. An idea about the order of magnitude of the *MTTF* (mean time to failure) of a 1024-node multiprocessor without any fault tolerance features is given in the following calculation. Suppose that the reliability function for the components (i.e. the probability that the component works correctly at time t is $R(t) = e^{-\lambda t}$, and the MTTF for a single node is 10 years. If an error in a single node causes the whole system to go down, then the MPS has an MTTF of only 85.5 hours. Empirical measurements show the same tendencies[3]. This emphasizes the *need for fault tolerance* in (massively) parallel systems.

An MPS, however, should have some built-in hardware fault tolerance features that can alleviate this task:

- Most processors have integrated support for error detection. This may cause a trap to be taken or flags to be set within the status register. Possibly, errors can be detected on a per *process* basis (as, for example, within a T9000 transputer).
- Special components may be added. For example, a system ASIC may monitor several system functions and contain a Hamming code-based EDC unit (error detection and correction), which *corrects* single bit errors and *detects* double bit errors in the external memory data words. To avoid an accumulation of bit errors in seldomly used words, a memory *scrubbing* mechanism may be implemented.
- A redundant node per reconfiguration entity can be provided.

Although permanent hardware faults can be detected by periodic or on-demand self-test mechanisms, the errors caused by transient faults can only be detected by *continuous* and *concurrent* error detection techniques. Furthermore, they should provide a high fault coverage, but cause neither any significant performance degradation nor any significant price increase.

We consider the system to be composed of three different building blocks: the processors, their associated memory and the communication network:

- In the *communication network*, several error detection methods can be directly supported by the hardware: such as a parity bit in each byte, detection of disconnected links and several verifications of integrity of the packet protocol. A high level end-to-end protocol may also be implemented.
- For the *memory*, an EDC code (an error detection and correction code—for example, based on a Hamming code which requires 8 redundant bits for each 64 bit word) is able to correct all single bit errors, detect all two bit errors and detect the majority of all other multiple bit errors.
- For the *processor*, the sole possibility is the use of *behaviour based* concurrent error detection. In this approach, information describing a particular aspect of the system behaviour (e.g. the program control flow) is previously collected (normally, this is accomplished at compile/assembly time). At run-time this information is compared with the actual behaviour information gathered from the object system to detect deviations from its correct behaviour. An excellent survey of this family of error detection techniques can be found in [8]. However, many of them cannot be used, because the off-the-shelf components used in the project do not have the necessary hardware support.

Beside the error detection techniques for which the system provides *hardware support*, *software-only* techniques may be used. These include software control flow monitoring, watchdog timer processes (also called *<I'm alive>* messages), periodic diagnostic routines and error capturing instructions (ECIs). This latter technique consists of inserting special trap instructions in unused memory locations, so that the execution of an ECI indicates that a control flow error has occurred. For software control flow monitoring, the program is divided into blocks with only one entry point and only one exit point. The block execution always starts at the *entry* point and always ends at the *exit* point. The identifier of the last traversed entry point is stored; at each exit point it is verified that the corresponding entry point was used.

Although able to detect a significant percentage of errors, software-only techniques are insufficient if many subtle transient errors will occur. The use of techniques supported by hardware can significantly increase the error detection coverage—these include misaligned memory accesses, arithmetic exceptions, illegal instruction detection, privilege breaks and accesses to non-existent memory. This requires flexible memory protection in the processor or in extra components.

An important question, remaining after the previous discussion, is how much coverage may be expected from the described error detection methods. Clearly it will not be 100%; this can only be attained by duplication, voting and an extensive use of self-checking logic. From experiments done with other processors[9,10,11] some assumptions can be made.

messages periodically to their neighbouring control processors (Figure 1). This is executed concurrently with the application programs.

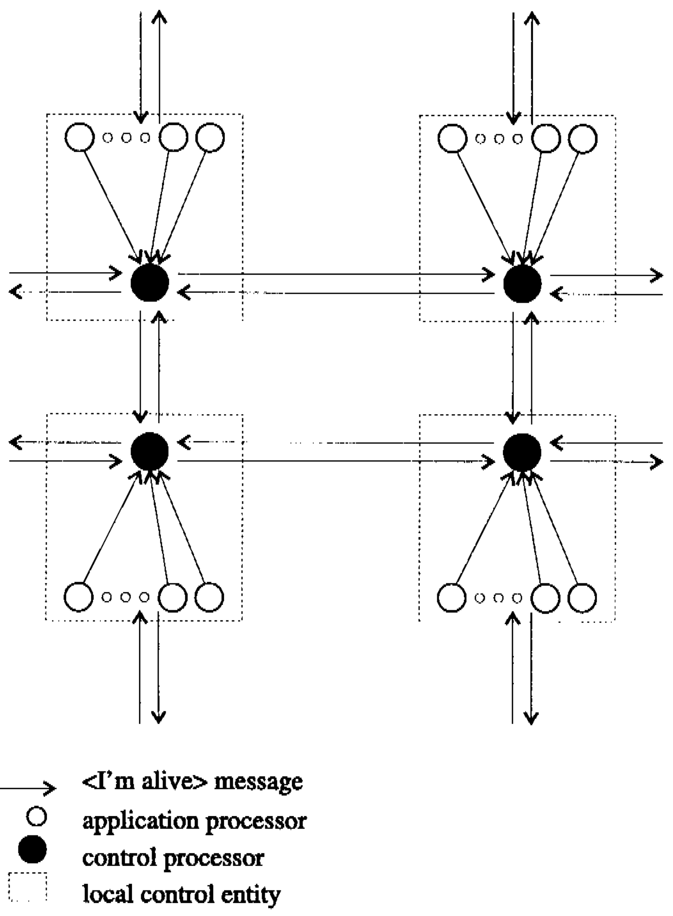

Figure 1. Fault diagnosis in the fault-free case

4.2.2 Faulty application processor

As soon as a fault is detected in an application processor, because an error is caught or the processor stops sending *<I'm alive>* messages, the application is stopped on all application processors of this partition (Figure 2, step 1). Then the status of the application processor is diagnosed by the control processor (Figure 2, step 2). After that, the control processor, which detected the error, informs the host and the other control processors related to the stopped application (Figure 2, step 3). The diagnosis software on the host updates the database of the OSS.

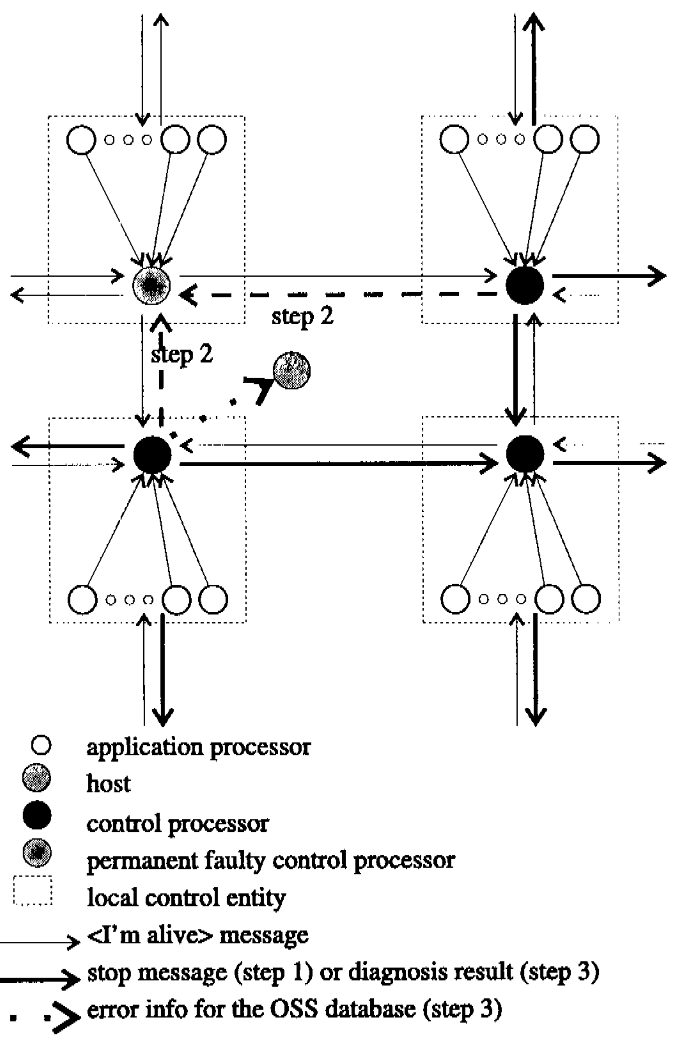
step 2
step 2
○ application processor
host
● control processor
permanent faulty control processor
local control entity
<I'm alive> message
stop message (step 1) or diagnosis result (step 3)
error info for the OSS database (step 3)

Figure 3. Fault diagnosis in the case of the failure of a control processor

4.3 Recovery

4.3.1 Backward error recovery

Backward error recovery, based on checkpointing and rollback, relies on saving the system status during the execution of an application program to stable storage. This set of information about the system status is called a *checkpoint*. In case of a failure the application can be *rolled back* to the most recent checkpoint and restart from there. If the last saved checkpoint was set at a point in time when the system was fault-free, a restart of the application from the state saved in the checkpoint will then lead to a fault-free execution of the application.

Checkpoints can be set at fixed time intervals (*time-driven*) or at distinct points in the

In the next two subsections, two approaches for gaining consistent sets of checkpoints are described. They both have the advantage that overhead is produced only while saving a checkpoint. This contrasts with the message logging protocol[14] or checkpoint propagation[15], where overhead is produced for every communication. In Section 4.3.2 a user-driven (semi-automatic) scheme is introduced, where the recovery line is indicated by the programmer and the checkpointing interval is data-dependent. In Section 4.3.3 a user-transparent protocol is presented, where the checkpointing is time-driven—i.e. triggered by the fault tolerance software at periodic intervals. They both co-operate with a scalable disk controller that is responsible for the management of the checkpoint data[16].

4.3.2 User-driven checkpointing

In this approach, the programmer calls the checkpointing routine at the places in the program code that exhibit a *consistent* recovery line. In parallel applications where processes show substantial geometrical symmetry (as are many target applications of this kind of MPS), it is rather easy to find these recovery lines, (e.g. at the end of a loop in iterative algorithms). When the processes of an application are more asymmetric, finding a recovery line may be more difficult. The programmer also indicates which data items (i.e. variables, structures, pointers, files, etc.) belong to the contents of the checkpoint.

The invocations of these routines are the only places where the user is involved. The rest (e.g. management of the storage of the checkpoint, determining what recovery lines are completely saved and the necessary bookkeeping) is handled *automatically* by the recovery software. The rollback is also handled user-transparently: first a consistent recovery line is determined and then the relevant checkpoints are restored to the application processes.

The advantages of semi-automatic checkpointing are:

- *Minimal and user-adaptable overhead*: only when the checkpoint is taken (on request of the programmer) does the application process suspend its operation (until the checkpoint is transferred to the disk controller). During normal computation, there is no need for logging events or messages—hence minimal *time* overhead. As the programmer specifies which data items are included in the checkpoint, the checkpoint size (and, hence, the *storage* overhead) is minimized.

- *Non-blocking*: the different application processes do not all have to reach the recovery line before their status is saved. Indeed, after the local checkpoint data has been gathered, the process can continue. This non-blocking approach alleviates the problem that the transfer of the checkpoint to secondary storage causes a communication bottleneck, because the checkpoints do not all have to occur at the same time. Whenever necessary (e.g. to include the contents of a shared writable file in the checkpoint), a synchronization barrier is imposed automatically by the fault tolerance software.

- *Hardware independent approach*: as the programmer indicates which data items contribute to the consistent status at the recovery line, this approach can be easily ported to other hardware platforms: no hardware or system software specific data (heap, stack, process queues, etc.) has to be included.

The disadvantage is the required awareness of the programmer to incorporate the fault tolerance in the application—i.e. the task of specifying and indicating the recovery line.

- When a *control* processor fails, the routeing must be adapted to isolate the local control entity associated with that failed processor.
- In the case of a *routeing switch* failure, the routeing has to be adapted to isolate the failed switch and allow a valid routeing of messages.
- In the case of a *link* failure, messages have to be routed around this link. A *local* approach is chosen for reasons of scalability.
- In the case of a *reconfiguration entity* failure, the failed reconfiguration entity needs to be isolated to allow a valid routeing of the messages, and a spare reconfiguration entity has to be found.

Hence, the two main tasks of the reconfiguration are *rerouteing* and *remapping*:

- *Rerouteing*: when an entity (a link, a processor, a reconfiguration entity, ...) has failed, the routeing must be adapted so that those failed entities are *detoured*. Possibly the routeing towards the spare processors must be included. Note that the routeing of an MPS can be based on *interval routeing*[18,19]; this mechanism is very well suited as, for a regular structure, *compact* routeing tables can be used. As faults destroy this regular structure, these routeing tables must be recalculated and adapted. The number of available intervals can be a limiting factor in adapting an injured system to a newly occurred fault[20].
- *Remapping*: when, due to the fault, the current partition is not useful any more, another partition where the application can be run should be found. Different problems need to be solved: the search for spares in the same partition, the search for a new partition without affecting the other partitions and a global re-partitioning of the system (as a tool for the system operator).

If the reconfiguration would involve the disturbance of other users, the system operator will be informed instead and should take the ultimate decision, possibly assisted by the above-mentioned remapping tool.

4.4 Operator-site software

The fault tolerance techniques described in the preceding sections will reduce the effect of hardware faults on users' programs significantly. Nevertheless, the maintenance staff, the producer of the machine, and the producer of the chips are still interested in information about the 'whens?', 'wheres?', and 'whys?' of faults. Therefore the operator-site software (OSS) and its database are set up.

For the maintenance staff, it is important to know which boards are damaged by permanent faults and have to be replaced. The producers of the machine and the producers of the chips try to localize weak points in their hardware in order to avoid them in future designs. For these reasons an automatic *error log* is integrated. Additionally, a software *monitor*[21,22,23] provides data about the system's workload in order to reveal failure/load relationships. Because of the complexity and the size of the machine, there are new problems with respect to the amount of data which has to be routed to and evaluated on the control host.

software. Communication with the other software modules and the transformation of all incoming and outgoing data is performed by the OSS-control unit.

Evaluation and visualization of the recorded data are done by a set of output modules. The statistics cover basic dependability characteristics (error/failure distributions, hazard rates, etc.) as well as workload/failure relationships (e.g. load-dependent hazard rates). Estimation methods (e.g. maximum likelihood estimation[24]) will be implemented which determine the parameters of detected faults with a particular distribution (e.g. Weibull distribution for transient, intermittent, and permanent faults[25]). Such calculations may allow error prediction which may reduce system downtime because *preventive* maintenance can take place.

In order to make complex statistical data comprehensible, a graphical representation is necessary. Therefore, the statistics module is integrated into an interactive user interface which is the platform for the operator to manage the OSS. All actions including the handling of exceptions (e.g. system crashes, shutdown of parts of the system in case of repairs or maintenance), the visualization of the current system's state (defect components, partitioning) and statistics and the starting of test programs are initiated from here. Additionally, the operator has the possibility to generate database entries manually for faults not covered by the fault tolerance software (e.g. shutdown for upgrading system hardware or software).

In consideration of the various tasks it has to carry out the OSS can be thought of as the *front-end* of the fault tolerance software. The built-in visualization features simplify the handling of massively parallel systems for the operator.

5 CURRENT STATUS

Owing to the unavailability (in 1993) of the Parsytec GC machines (based on the T9000 transputer and C104 routeing switches), the first prototype of the software was delivered at the beginning of 1994 on the Parsytec GC_{el} and the Xplorer, both based on the T805 transputer and under the Parix operating system. For the final implementation (summer 1995), the unifying system model allowed us to port our software successfully to the Parsytec PowerXplorer series, where every node consists of a PowerPC601 (for computation) and a T805 (for the communication). The operating system is PowerParix 1.3.

In the above-mentioned prototype, the basic functionality was integrated and the global concepts were proven. It contained error detection and fault diagnosis based on the sending of *<I'm alive>* messages. The reconfiguration of the system comprised rerouteing and remapping. Application recovery based on user-driven checkpointing and rollback was implemented, as well as a visual operator interface and error-log tool. Furthermore, a software fault injector tool was used for validation. Beside the saving of the checkpoint data to secondary storage, this prototype had a run-time overhead due to the sending of the *<I'm alive>* messages (once every second) between neighbouring processors, and due to the rerouteing and remapping after a failure. Depending on the application, this is less than a few per cent run-time overhead. The application recovery influences the overhead only during the saving of the checkpoint data; this overhead is mainly determined by the available I/O bandwidth.

The final implementation integrates the complete fault tolerance approach (as described in Section 4) into dedicated system software. A validation of modules of this fault tolerance software by internal and external beta-users is currently taking place. Some modules will be commercialized in future MPS by Parsytec.

[12] E. Gelenbe, 'On the optimum checkpointing interval', *ACM Journal*, **26**(2), 259–270, (April 1979).
[13] B. Randell, 'System structure for software fault tolerance', *IEEE Trans. on Software Engineering*, **1**(2), 220–232, (June 1975).
[14] D.B. Johnson and W. Zwaenepoel, 'Sender-based message logging', in *IEEE Proceedings of FTCS-14*, pp. 14–19, (June 1987).
[15] R. Koo and S. Toueg, 'Checkpointing and rollback-recovery for distributed systems', *IEEE Trans. on Software Engineering*, **13**(1), 23–31, (January 1987).
[16] B. Bieker, G. Deconinck, E. Maehle, and J. Vounckx, 'Reconfiguration and checkpointing in massively parallel systems', in *Proceedings of EDCC-1*, volume 852 of *Lecture Notes in Computer Science*, pp. 353–370, Berlin, Germany, (October 1994). Springer-Verlag.
[17] A. Bauch, B. Bieker, and E. Maehle, 'Backward error recovery in the dynamical reconfigurable multiprocessor system damp', in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 36–43, Amherst, MA, (July 1992). IEEE.
[18] N. Santoro and R. Khat, 'Labelling and implicit routing in networks', *The Computer Journal*, **28**(1), 5–8, (1985).
[19] J. van Leeuwen and R. B. Tan, 'Routing with compact routing tables', Technical Report RUU-CS-83-16, Rijksuniversiteit Utrecht, (November 1983).
[20] J. Vounckx, G. Deconinck, R. Cuyvers, R. Lauwereins, and J.A. Peperstraete, 'Network fault tolerance with interval routing devices', in *Proceedings of the 11th IASTED International Symposium Applied Informatics, Annecy, France*, pp. 293–296, (May 1993).
[21] F. Castillo and D.P. Siewiorek, 'Workload, performance, and reliability of digital computer systems', in *IEEE Proceedings of FTCS-11*, pp. 84–89, (June 1981).
[22] R.K. Iyer and D.J. Rossetti, 'A measurement-based model for workload dependence of cpu errors', *IEEE Trans. on Computers*, **C35**(6), 511–519, (June 1986).
[23] E. Maehle and W. Obelöer, 'Delta-t: a user-transparent software-monitoring tool for multi-transputer systems', in *Proceedings EUROMICRO 92, Microprocessing and Microprogramming*, volume 32, pp. 245–252, (September 1992).
[24] J.L. Melsa and D.L. Cohen, *Decision and Estimation Theory*, McGraw-Hill, New York, 1978.
[25] T.T.Y. Lin and D.P. Siewiorek, 'Error log analysis: statistical modelling and heuristic trend analysis', *IEEE Trans. on Reliability*, **39**(4), (February 1988).