Published in: Proceedings of the ISUG 1993 Annual Users' Conference. St. Louis, Mo., October 1993.

The Pact Fault-Tolerant Parallel Programming Environment for the Paragon™

Joachim Maier
Institute of Parallel and Distributed High-Performance Systems (IPVR)
University of Stuttgart
joachim.maier@informatik.uni-stuttgart.de

Abstract

Parallel programming is hard enough, nevertheless parallel computers, which often are not far away from prototype status, have low reliability. This makes it difficult or sometimes impossible to successfully execute long-running applications, as they are typical for the 'Grand-Challenge' problems.

Pact is a parallel programming environment with user-transparent fault-tolerance based on atomic actions. Virtual shared memory with user-defined dataunits and an easy-to-use synchronization model make parallel programming easier without losing too much efficiency.

In this article the design for a massively parallel implementation on the intel® Paragon™ is presented which due to the availability of threads and a very efficient asynchronous message-passing by using message-passing processors promises even less runtime overhead than the current prototype which is running on a Tandem™ multiprocessor system.

1 Introduction

Distributed memory machines like the intel® Paragon™ usually support *message passing* as the basic paradigm for parallel programming. Message passing libraries abstract from the low-level networking details and give the programmer primitives to create processes on other processors and for inter-process communication.

Although the architecture of those machines is especially optimized for message passing, this is still a very expensive operation. This is the reason why the usual strategy to write efficient parallel programs is to minimize the number of messages. This and the large potential for data-parallelism in most algorithms are the reasons why parallel programs are often written following the SPMD (single program multiple data) method. Hereby domain decomposition methods are used to partition global data and distribute it statically to processes on different processors.

The problem with this approach is, that if the amount of work, which has to be done on the partitions of data, is not the same, some processes have to work longer than others and the whole program has to wait for the last process. Static load balancing by domain decomposition, which is hard-coded in the parallel program, tries to minimize these *scewing* effects. If the amount of work is not predictable, because it depends on the data, or the parallel program is running in a multi-tasking and multi-user environment, static load balancing cannot be very effective. A possible solution is to use task-parallelism instead, where a dynamic load-balancer can be used to schedule tasks to nodes with lower load. The problem here is that this implies the automatic migration of data to these nodes - but if the data is shipped dynamically, a transparent addressing scheme is needed as it is provided by *virtual shared memory* systems.

Although message passing is the obvious programming model for distributed memory computers message passing (remote procedure calls) is more adequate for client-server applications where they are used for function and data encapsulation and meet the requirements from software engineering. In this environment servers, like procedures in structured programming, have one defined entrance and exit point and defined fault semantics.

In parallel programming the situation usually is different. Messages are sent and received at different points in the program and usually all addressing has to be done explicitly by the programmer. Therefore, small mistakes by the programmer may cause a deadlock or completely wrong results. But the worst thing is that in most cases such mistakes are very hard to find [12], because if the program is running under the control of a symbolic debugger or writes a trace-file the order of the messages could be changed, because of timing effects. This is not only a problem with message passing programs, but also with shared-memory programs where mistakes in the synchro-

nization of shared data accesses may cause similar problems.

As if this would make parallel programming not hard enough, parallel computers which are often not far away from prototype status have low reliability. This makes it difficult or sometimes impossible to successfully execute long-running applications, as they are typical for the 'Grand Challenge' problems.

The usual way to solve this problem is to periodically checkpoint the application by writing intermediate results to persistent storage. This is not very efficient, because all of the application's processes have to be synchronized and all checkpoint information has to be written to the disk at the same time. But the most important problem is that the application program needs additional code for checkpointing and recovery which must be written by the application programmer. Since experience with commercial codes shows that even error handling can take up to one third of the code [29] it is clear that fault-tolerance should be user transparent.

The Pact[1] parallel programming environment solves these problems by providing an easy-to-use parallel execution and synchronization model based on task parallelism. The migration of global data which is necessary for enabling dynamic load balancing is provided by virtual shared memory. Fault tolerant execution in Pact is based on atomic actions and is guaranteed by the runtime system in a fully user transparent way.

The next two sections of this paper describe approaches to virtual shared memory and fault tolerance and their relation to the concepts in Pact. Section 4 gives a brief description of Pact's programming model. Pact's runtime system for the Paragon is described in section 5. The experience with the current prototype running on a Tandem system are described in section 6, and in section 7 some conclusions are given.

2 Virtual Shared Memory

Considering that many computational intensive algorithms use global data structures, the primary advantage of virtual shared memory over message passing is that of a simpler abstraction for the application programmer. It is not necessary to subdivide global data and distribute parts of it over the nodes, because this is done automatically and fully user transparent by the virtual shared memory system. Although the programming model of virtual shared memory looks similar to that of physical shared memory, it is not exactly the same. Remote references to virtual shared data have an administration overhead and a latency which is much higher than that of physical shared memory; therefore, only coarse grained algorithms have a good performance.

Although performance comparisons [28] show that a fine tuned message passing program mostly has better performance than a program using virtual shared memory for complex programs where this tuning is too difficult and too time-consuming, or where dynamic load-balancing is required, the performance of virtual shared memory programs could be equivalent or even better.

Since Li and Hudak first propagated *shared virtual memory* in [19] as an extension to the operating system's virtual memory management, a lot of research has been done to improve the performance of virtual shared memory systems[2]. The problems which have to be solved are the high *latency* for remote data accesses and *false sharing* [26]. In order to integrate virtual shared memory management in the operating system's virtual memory manager many systems use *virtual memory pages* as the unit of transportation and coherency. If the granularity of these units does not fit with the way the program accesses the shared data, the page can be "trashed" between some processors – when different processes concurrently update two different variables in the same page this page is passed continuously between these nodes.

Hardware based implementations like the Stanford Dash [18] or the KSR-1 [10] minimize latency by special hardware support. This leads to a more complex architecture and therefore higher costs. To reduce false sharing the KSR-1 as well as the DASH are using cache-lines rather than pages.

Since Cheriton's approach to *problem oriented shared memory* [6] many systems try to mask latencies and avoid false sharing by relaxed coherency protocols. In the software based virtual shared memory system Munin [2], as in the Dash system, *release consistency* [14] is used to do this. Munin additionally provides different types of shared variables with strict or weak coherency protocols. An overview of virtual shared memory algorithms and systems can be found in [26], [30], [32], and [33].

In Pact, virtual shared memory management is done with a *fixed distributed manager* algorithm with *write-invalidation* similar to Li's algorithm in [20]. This management is done in the runtime system instead of the operating system. This has the advantage that Pact can be implemented on top of any kind of operating system and therefore provides a basis for portable parallel programs. Besides this, it enables the system not to use fix-length pages, but gives the programmer the possibility to define

1. Pact is a synonym for parallel actions – the Pact programming model is based on atomic actions which are running in parallel.

2. Some systems are called *distributed shared memory* sytems to emphasize the physical distribution of the memory, especially when it is distributed over a workstation cluster.

the size of the *dataunits* himself, in a way which fits his program best. These user-defined dataunits avoid false sharing and make it unnecessary to use weak coherency protocols to receive good performance.

In contrast to most other systems Pact's goal is not to simulate physical shared memory by providing *strict consistency* [26], but to use even stronger consistency comparable to those of transactions. Pact follows the two laws of concurrency control [16]:

1. Concurrent execution should not cause application programs to malfunction.
2. Concurrent execution should not have lower throughput or much higher response times than serial execution.[1]

The first law is fulfilled by Pact's consistency model, which guarantees *serializability.* This means that shared memory is held coherent in a way of an arbitrary serial execution order of the concurrent tasks. Each node's updates are not seen by others before its successful end. Therefore, it is not necessary in Pact to use explicit synchronization primitives to guarantee the correctness of updates on shared data.

The second law is fulfilled because an analysis of typical algorithms shows that access conflicts on shared data usually are rare, if the granularity of the dataunits meets the requirements of the program. This consistency model improves the chance of a correct execution of parallel programs, because the programmer has not to worry about synchronization on shared data but only has to identify independent tasks for parallelization. If by mistake, conflicts on shared variables occur because of a wrong decision in the selection of the dataunits, the performance might be reduced, but the correctness of the parallel program is not affected.

To solve the latency problem in the Pact runtime-system, latency of remote data-references is masked by multiple threads, executing different tasks, in each node-process. Whenever a thread is waiting on remote data, another executable thread in the same process is executed. This is explained in section 5 where the runtime system is described in more detail.

3 Fault-Tolerance

Although parallelism can be used to build fault-tolerant computers (e.g. [1] and [7]) most parallel computers and their operating systems are not designed for this purpose. The design goal is high performance, but this often causes relatively low reliability [25]. Unfortunately, most parallel computers do not even provide mechanisms for fault-tolerant execution of application programs. So the programmer has to deal with faults himself.

The easiest way to do this is starting the program once again after a system failure occurs. This means that all work done before the crash is lost. If this is not acceptable, and for long-running applications it is definitely not, the program has to deal with fault tolerance, e.g by using *checkpoints*.

In the absence of a checkpointing library this can be done by synchronizing all processes at a consistent global state and write the results of the computations made up to this time in a checkpoint file. If the system fails after the checkpoint is completed, the program can be restarted by the programmer from the most recently saved state.

The problem here is that the programmer has to decide which data has to be written in the checkpoint file and how to recover from this data. This is an unsatisfying solution, because all work has to be done by the programmer which increases the complexity of the parallel program. To avoid these disadvantages a user-transparent checkpointing library should be used, which implements one of the numerous checkpointing protocols for parallel and distributed computer systems known in literature.

These checkpointing schemes can broadly be classified into *pessimistic* and *optimistic* schemes [21]. Pessimistic schemes as presented in [3] and [27] log each message synchronously as it is sent or received. This makes recovery fast, but the overhead of synchronously logging every message makes these schemes too expensive for applications on parallel computers where frequent message transfers occur. These schemes are named "pessimistic", because they assume high failure probability as it is typical for distributed computing and therefore optimize recovery on the cost of high overhead during normal processing.

Optimistic schemes, such as those presented in [17] and [31] are based on asynchronous logging of messages. This causes low overhead on normal processing and allows the processes to choose the time to checkpoint their internal state and their messages. This is achieved by a complex recovery algorithm. Recovery requires global rollback to a saved state and subsequent redo of logged messages to restore a consistent global state. Hence, recovery is much more expensive than in pessimistic schemes. The name "optimistic" comes from the optimistic assumption that failures are rare.

Recent approaches like the protocols proposed in [5], [8], and [21] optimize the drawbacks of checkpointing by periodically checkpointing consistent states of the program's processes instead of checkpointing all messages. If this is done with further optimizations like *copy-on-write* and *incremental checkpointing* [9] the runtime overhead is low.

1. This refers to the concurrent execution of transactions on a single processor.

For database systems fault tolerance is guaranteed in a fully transparent way by using *transactions* [15]. Transactions are atomic units of execution, which means that either the whole transaction runs successfully or, if a fault occurs during its execution, all changes on the shared database are undone. This leaves the database in a consistent state, even if a system fault occurs. Transaction processing systems have been the subject of studies over decades in various implementations for single- and multiprocessor-systems, and for distributed systems. Refer to [16] for a detailed description of the current state of transaction processing.

Transactions are not designed for parallel programming but for database applications, where a user's program should run in an *atomic* (enables fault tolerance due to clear fault semantics) and *isolated* (avoids multi-user anomalies) way on shared databases. All changes of the transaction have to be *durable* after the transaction's *commit* and the database should remain in a *consistent* state under all circumstances.

While atomicity and consistency are also desirable for parallel programming, isolation is not advantageous and durability is unnecessary. This is the reason why transactions are not useful for parallel programming, without any modifications. Some work in this direction is described in [22] and [34] where *atomic actions* are used for the implementation of object oriented distributed systems with durable objects.

Because transactions are independent units of execution, there is usually no need for a control structure between them. For parallel programming there is an essential need for a flexible, easy-to-use way to control the program's parallel execution.

In Pact, atomic actions are used to achieve fully user-transparent fault-tolerance with low runtime overhead. This is done with the following ideas in mind:

1. To use the long-standing experience in transaction processing systems for parallel programming to develop an efficient approach to fault tolerant parallel computing.
2. With atomic actions it is possible not only to build efficient restart recovery, but even to build systems with continuous execution by tolerating single-point failures.
3. One of the main advantages of atomic actions is not only the possible tolerance of system failures but also flexible reaction and recovery of faults inside the application program because of the simple fault semantics due to atomicity.
4. Using atomic actions avoids the *domino effect* (rollback of *all* processes to a globally consistent state) during recovery, because recovery of each action is independent of the execution of other actions.

4 Pact's Programming Model

Since the topic of this paper is the design of the runtime system for the Paragon, Pact's programming model here is only described as far as it is necessary to understand the basic ideas. The programming model and Pact's extensions to procedural programming languages are described in more detail in [23].

Because Pact is designed for MIMD [11] computers it is based on the task parallelization paradigm. Therefore, it is possible in Pact to use data-parallelism as well as function-parallelism [4].

Tasks in Pact are called *actions* to reflect their special semantics. The action's *atomicity* guarantees that if an action is executed sucessfully their results can be seen by other actions, or otherwise if an error happens in an action ("division by zero", "out of memory", etc.) the action returns without changing shared data. This makes it easy to react to errors. If, for example, the action gets an "out of memory" error, it is possible just to release some memory in an external error handling action and to restart the action once again. As reported in [29] using atomic actions increases the quality and reusability of (especially large) code, and reduces the complexity by a remarkable amount.

Each parallel action has access to shared data. Contrary to the "strict consistency" used on most shared-memory multi-processors, where each parallel process has uncontrolled access to the shared data, Pact's actions use an access-dependent two-phase locking scheme. This helps avoiding synchronization faults, which are one of the most frequent faults in parallel programs and usually hard to find.

The shared memory model in Pact is used to provide a high programming abstraction and to enable dynamic load-balancing. To choose the best access to the shared data the programmer should define *dataunits*, which are either simple shared variables, records, or parts of the large shared variables like vectors and matrices (e.g. rows of a matrix for a gaussian elimination algorithm, or submatrices for a block-tridiagonal solver). These dataunits are the unit of concurrency control and consistency, and therefore avoid lock conflicts.

The execution order of parallel actions is defined by using *events*. Each action is connected to three events: its *start-event*, its *termination-event* and its *error-event*. When the start-event occurs, the action is scheduled by the runtime system. If the action has successfully terminated its execution, the termination-event occurs which may start the execution of other actions (by triggering their start-events). Error-events give the programmer the possibility to react to transient software faults. A simple example for an execution plan showing only start- and termination-events in form of a dependency graph is illustrated in

figure 1. In this example, the actions g(i), g(i-1), and f(i) are executed in parallel after the termination of s() – the event ParBegin is triggered. The action e(y) is started after the event ParEnd is triggered. This is when the actions g(i), g(i-1), and f(y) have terminated.

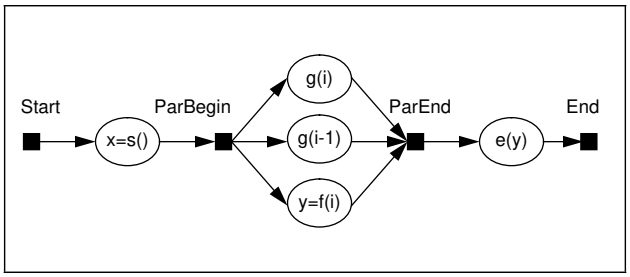

Figure 1. Pact execution plan.

The example in figure 2 shows error handling with error-events. If the action f(x, y) fails because of an internal error the error-event error is triggered instead of the termination-event term. After the error-handling in the action errhdlr() has been done, the action f(x,y) can be executed again. This is possible because in the error case all changes of action f(x,y) on shared data has been undone by the runtime system. If the execution of the action is successful now, the program can be continued.

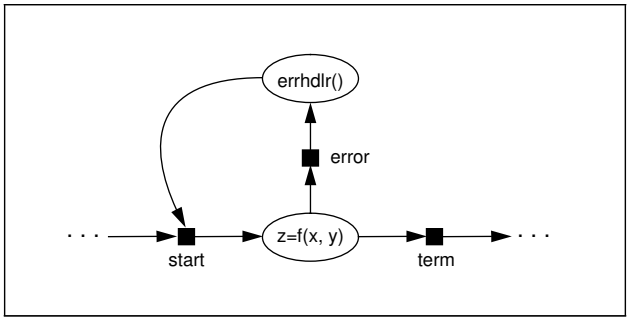

Figure 2. Error handling with error-events.

5 Pact's Runtime System for the Paragon

The runtime system for the Paragon is designed to provide the features of the programming model in the most efficient way. To do this, the following features of the Paragon's OSF/1 operating system and hardware are being used.

- Multi-threaded servers to mask the latency of remote references to shared dataunits and scheduling of actions.
- Almost all messages in the runtime system are sent asynchronously. While this is a general principle to obtain good performance, it is especially important on the Paragon, because asynchronous messages have very low latency due to the Paragon's message-passing processor.
- The hardware performance monitor is used to receive actual information of the processors' system load parameters (run-queue length, cpu-usage, page-faults, etc.) without any performance degradation. This is done to optimize the parallel execution of Pact programs by using dynamic load-balancing.
- Virtual shared memory is implemented using a fixed distributed manager algorithm similar to the algorithm in [20] adapted to Pact's consistency model and the architecture of the runtime system.
- Distributed Supervisors and Log-Managers provide scalability even for a high number of server processes (each server process should run on a separate compute node to avoid context switches).

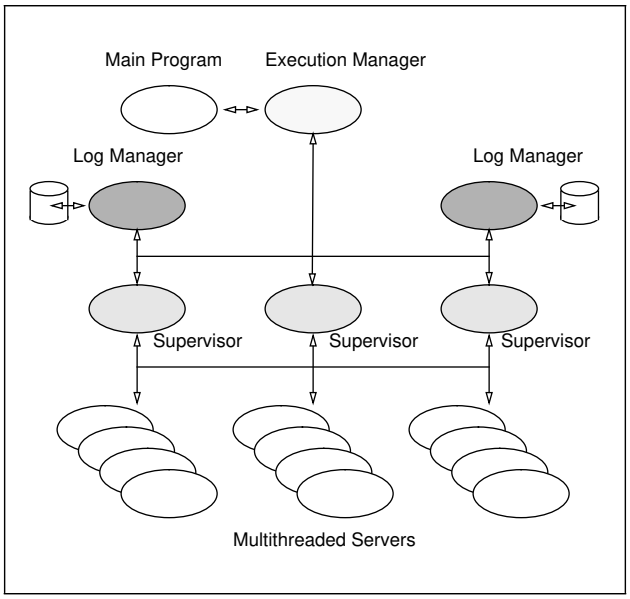

Figure 3. The Pact runtime system.

5.1 The Scheduling and Execution of Actions

As illustrated in figure 3, the runtime-system consists of a number of system processes (Execution Manager, Supervisors, and Log-Managers) and a number of processes which are generated by a precompiler from the Pact application program. The *Main Program* where the execution plan is defined and serial execution is done, sends its action calls to the *Execution Manager*. Each action-call includes an (optional) *data-description* which describes the shared dataunits, which the action will read and write. Based on this information the Execution Manager forwards the action to the *Supervisor* which manages most of the dataunits.[1]

This Supervisor schedules the action-call to "the best" *Server* process, where a new thread is created to execute the action. The Supervisor's scheduling decision is based on a load-balancing strategy which uses information about the server node's actual load factor (computed from run-queue length, cpu-usage, page-faults, etc.) and the ownership information of the virtual shared memory. To provide good performance with virtual shared memory it is essential to avoid data-migration as far as possible. Therefore, one of the most important load-balancing criteria is the current ownership and replication of the dataunits. The Supervisor compares the nodes' ownership information and the action's data-description and uses these in the load-balancing strategy to find "the best" server-process.

When the action during its execution needs access to a shared dataunit, it first looks in the directory of the server process. If its process is already the owner of the dataunit, it requests a lock for it. Otherwise it sends a request to the Supervisor which manages this dataunit (and the next executable thread is executed). If the dataunit is not currently locked by the owner node, the Supervisor forwards the request to the owner node and changes the ownership information in its directory. After receiving the request the owner node sends the dataunit to the requesting node, where the requesting thread is still waiting for it.

At the end of the action's execution, it uses a commit protocol as described in [24]. During the commit phase it releases all its locks on shared dataunits and sends all modified dataunits and its result parameters to the *Log-Manager* which is associated with this node. In this phase the action returns its result parameters to the calling Supervisor. The *termination-event* is triggered and the next action is executed and the results are returned to the Execution Manager and the Main Program.

To avoid that the Log-Manager has to force the log-data (to write it immediately on stable storage) in each action's commit, a concept similar to *group-commit* in [13] is used. The idea of group-commit is that log buffers are not written before they are full, or after a specific timeout. Unlike in transaction processing, it is not necessary in Pact to delay the commit of the action until the log is written, but it is possible to end the commit phase as soon as the log record has been received by the Log-Manager. This is possible because actions need not to be *durable*. If a log record is lost due to a system crash, a preceding state of the parallel execution can be recovered from the log-file.

When an error occurs inside of an action, due to the atomicity of actions, the action has to undo all its changes on shared dataunits. This is done by performing *abort* instead of commit. In the abort phase the action gets the previous (the last committed) state of the dataunits from Log-Managers, releases all locks and sends an error-message to the Supervisor. The error-event is triggered and depending on the execution plan, an error adjustment action is scheduled, the error message is sent to the Main Program, or the action is just scheduled again, assuming that it was a transient error which will not occur again.

5.2 Recovery and Further Optimizations

Fault-tolerance in Pact is based on the atomicity of actions. This property gives actions simple fault-semantics. Whenever a program error occurs or a node fails, it is easy to bring the shared data into a consistent state by getting the lost dataunits in the last committed state from the Log-Managers. Lost actions just have to be scheduled again.

Although Pact has the potential to guarantee the continuous execution of the parallel program by tolerating single node failures, this presumes that the underlying machine also tolerates single node failures. This presumption is correct for the Tandem machines where Pact's first prototype has been implemented, but it cannot be applied to the Paragon. The problem on the Paragon is not only that the OSF/1 operating system is not designed to tolerate partial failures, but that not even the hardware is designed to do this. When a Paragon node fails and a message is sent to this node the iMRC router chip does not delete this message, but the message blocks the connections from the sending node to the failed node. Because the routing is not able to find alternative paths, after some time more and more connections are blocked and the whole machine fails.

The Pact runtime-system for the Paragon deviates from the design in [24], where the goal is to provide continous execution. On the Paragon it is necessary to improve the performance of restart recovery. To do this, it is not only necessary to recover the last committed version of the dataunits very fast, but to recover the Main Program and the system processes (Execution Manager and Supervisors) as well. For optimizing this, these processes write checkpoints onto the log.

After these processes have been restarted and have read their last checkpoint from the log, they can reach the last consistent state by redo-recovery. Redo-recovery in this context means that the lost parallel execution after the checkpoint is simulated. This is done by sending action-calls to the Execution Manager, but the actions are not executed, instead the result parameters of preceding executions are fetched from the log.

Since the processes which need to checkpoint are only a few compared to the large number of server processes this checkpointing does not have the overhead of checkpointing the whole parallel application. But there is a second advantage of the checkpoints. By using checkpoints it is not necessary to keep the log-file of the whole execu-

1. The Supervisor S_x manages all dataunits i in the hash-class $x=h(i)$.

tion, instead older parts of the log-file which does not contain the latest checkpoints and only older versions of dataunits can be deleted. This is managed by using *low water marks* similar to the Log-Manager design in [16].

6 Experiences with the Current Prototype

The current prototype is running on a Tandem VLX *shared-nothing*[1] multiprocessor system [1], [7]. The Tandem hardware and the Guardian™ distributed operating system tolerate partial failures, which enables Pact to guarantee continuous operation [24]. Because the Tandem machine is designed for commercial (e.g. database) applications which need only integer and character processing and not for numerical computations it has no floating-point co-processors or vector units. Therefore, the absolute runtime of the measurements are not comparable to any supercomputer or high-performance workstation.

The measurements of the additional time which is needed for enabling fault-tolerance (commit protocols, logging, etc.) are shown in Table 1. The following applications are used: a gaussian elimination method to solve a dense 200×200 equation system, a multiplication of two 200×200 matrices, a finite element analysis of an 160 element object, where a conjugate gradient solver is used to solve the sparse equation system.

application	with logging	without logging	over-head
gauss (200×200)	09'27"	8'35"	10.10%
matmult (200×200)	32'40"	31'25"	3.98%
fem (160 elem.)	10'53"	10'33"	3.16%

Table 1. Log overhead in the Pact prototype.

Although these measurements still show low overhead, we believe that the implementation on the Paragon will decrease it once more. The reason for most of the overhead is not I/O, but the increased number of messages. Although messages are sent asynchronously on the Tandem implementation, as well, each message has to be processed completely on the same cpu where the application is running.

Due to the unavailability of threads in the Guardian operating system the masking of latencies is done by using multiple processes instead of multiple threads in one process. The necessary context switches cause overhead which limit the masking effect and reduce the efficiency of the virtual shared memory implementation.

1. Nothing except the interconnect is shared in this architecture, neither memory nor disks.

Besides this, additional measurements show that due to the higher management overhead for large files in the operating-system's file-system the logging overhead is increased for very large applications. Therefore writing checkpoints to reduce the size of the log-file is also necessary for the Tandem implementation.

Since the Paragon enables a much higher number of nodes than the Tandem VLX system it is necessary to distribute the Supervisor and the Log-Manager for the Paragon implementation. On our Tandem system with 8 processors the centralized Supervisor and Log-Manager has never been a bottleneck.

7 Conclusions

The Pact prototype on the Tandem system shows low overhead for fault-tolerant execution of parallel programs. This overhead will be reduced even more with the Paragon implementation due to lower message passing latency and higher bandwith with the availability of the message-passing processor on the Paragon.

Threads with minimal switching times will be used to mask the latencies in the Pact runtime system. User defined dataunits minimize false-sharing which makes weak consistency protocolls unnecessary. Distributed schedulers control the execution using system load parameters and information about the locality of shared data to balance the load and avoid data-transfers. Therefore, the parallel excution using virtual shared memory promises to be efficient.

But Pact does not only promise efficieny and low overhead, but also an easy-to-use programming model. With its consistency model, which is easy to understand and which makes synchronization simple, it reduces the mistakes when using shared data in parallel programs. The ease of error-handling with atomic actions and the encapsulation of shared data accesses in Pact promise good readability and reuseability of Pact programs which is extremely important for large (industrial) codes.

8 References

[1] Joel Bartlett. A NonStop Kernel. In *Proceedings of the 8th Symposium on Operating System Principles*. ACM SIGOPS, December 1981.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Using Multi-Protocol Release Consistency. In A. Kershmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, pages 56–60, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[3] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the Symposium on Operating System Principles*, pages 90–99, Atlanta, October 1983. ACM.

[4] N. Carriero and D. Gelernter. How to Write Parallel Programs - A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[5] John B. Carter, Alan L. Cox, Sandhya Dwarkas, Elmootazbellah N. Elnozahy, David B. Johnson, Pete Keleher, Steven Rodrigues, Weimin Yu, and Willy Zwaenepoel. Network Multicomputing Using Recoverable Distributed Shared Memory. In *Proceedings of the Compcon '93*, San Francisco, February 1993.

[6] D.R. Cheriton. Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *6th International Conference on Distributed Computing Systems*, pages 190–197, Los Angeles, CA, May 1986. IEEE Computer Society.

[7] Flaviu Cristian. Basic Concepts and Issues in Fault-Tolerant Distributed Systems. In A. Kershmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, pages 119–149, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[8] Flaviu Cristian and Farnam Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 12–20, Pisa, Italy, September 1991. IEEE Computer Society Press.

[9] Emootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenepol. The Performance of Consistent Checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[10] Steven Frank, Henry Burkhardt III, James Rothnie. The KSR1: Bridging the Gap Between Shared Memeory and MPPs. In *Proceedings of the Compcon '93*, pages 285-294, San Francisco, February 1993

[11] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, c-21(9):984–960, September 1972.

[12] H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a Distributed Computing System. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984.

[13] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS FastPath. *IEEE Database Engineering*, 8(2):3–10, 1985.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual international Symposium on Computer Architecture*, pages 15-26, Seattle, Washington, May 1990.

[15] Jim Gray. Notes on Database Operating Systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, pages 393–481. Springer Verlag, 1978.

[16] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[17] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of Seventh ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy. The Directory-based Cache Coherency Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, 1990.

[19] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, Calgary, Alberta, Canada, August 1986.

[20] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[21] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing Multicomputer Applications. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 2–11, Pisa, Italy, September 1991. IEEE Computer Society Press.

[22] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[23] Joachim Maier. Pact - a Fault Tolerant Parallel Programming Environment. In *Proceedings of the 1st International Workshop Software for Multiprocessors & Supercomputers: Theory, Practice, Experience*, St. Petersburg, February 1993.

[24] Joachim Maier. Fault Tolerance Lessons Applied to Parallel Computing. In *Proceedings of the Compcon '93*, pages 244–252, San Francisco, February 1993.

[25] Paul Messina. The Concurrent Supercomputing Consortium: Year 1. *IEEE Parallel & Distributed Technology*, 1(1):9–16, February 1993.

[26] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[27] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the Symposium on Operating System Principles*, pages 100–109. ACM SIGOPS, October 1983.

[28] Thierry Priol and Zakaria Lahjomri. Trade-offs Between Shared Virtual Memory and Message-passing on an iPSC/2 Hypercube. IRISA Technical Report 637, IRISA/INRIA, Rennes, France, February 1992.

[29] Bruce A. Rafnel. A Transaction Approach to Error Handling. *Hewlett-Packard Journal*, pages 71–77, June 1993.

[30] Sanjay Raina. Virtual Shared Memory: A Survey of Techniques and Systems. Technical Report CSTR-92036, University of Bristol, December 1992.

[31] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[32] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[33] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber. A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems. *Operating Systems Review*, 24(3):40–67, July 1990.

[34] C.T. Wilkes and R.J. LeBlanc. Rationale for the Design of Aeolus: A system programming language for an action/object system. In *International Conference on Computer Languages*, pages 107–122, New York, 1986. IEEE.