Published in: Proceedings of 1st International Workshop ‘Software for Multiprocessors and Supercomputers: Theory, Practice, Experience’ SMS TPE 93. St. Petersburg, Russia, February 1993.

Pact – A Fault Tolerant Parallel Programming Environment

Joachim Maier
IPVR
Stuttgart University
joachim.maier@informatik.uni-stuttgart.de

Pact is a parallel programming environment relieving the programmer from the burdens of parallel programming which are not really necessary to write efficient parallel programs. This is done by providing a simple synchronization model and virtual shared data with user-defined granularity and automatic consistency control. Pact guarantees user-transparent fault-tolerance with low overhead by using atomic actions as basic units of parallel execution. Additionally, the runtime system maps parallel actions to server processes using dynamic load-balancing. An included on-line visualization tool helps tuning and debugging parallel programs.

1 Introduction

Since operating systems for parallel computers have become commercially available (e.g. Tandem’s Guardian, Sequent’s Dynix, Intel’s and Kendall Square’s OSF/1), the main obstacle to broader acceptance is the absence of an easy-to-use parallel programming environment. There is an ongoing debate on what a programming language for parallel computers should look like without loosing too much efficiency. A lot of different languages have been proposed in the last years,[1] where most approaches directly reflect the hardware architecture. This allows efficient programming of the underlying hardware, but prevents from portability.

The *shared memory* programming model [Han75] is originally developed for single-processor computers where many processes share data-structures like in operating systems or database management systems. But this model can be used for shared memory multiprocessor systems in the same way [Ost86]. The fact that the shared memory model provides shared data-structures which can be concurrently used by all tasks of the parallel program is very important for parallel programming, because many parallel programs need common data. To guarantee consistency, access to shared data-structures has to be synchronized by explicitly setting semaphores which are used for other synchronization purposes, too.

The *message passing* model [Hoa78] is oriented towards *distributed memory* architectures. In this model synchronization has to be done by sending messages. Because there is no notion of common data, all data needed by the parts of the parallel program, running on different nodes, have to be distributed explicitly by sending messages.

Distributed memory seems more suitable for building massively parallel computers, because it provides better scalability, lower costs and less access conflicts than centralized shared memory. Because of its support of common data from the point of view of program-

[1] Refer to [BST89] for an overview on programming languages for distributed computing systems.

ming ease, shared memory seems to be the better choice. The approach of *virtual shared memory*[1] simulating logically shared memory on a distributed memory computer is obviously a solution of this conflict.

Li and Hudak called it *shared virtual memory* [LH86], [LH89] and showed the practicability and efficiency of their (distributed and centralized) algorithms on selected applications. Like in the well-known *virtual memory* algorithm references on non-local pages (page-faults) trigger the page's migration into local processor's memory. The algorithm is easy to implement in the operating system, but has the drawback that it cannot use application-specific information. Li and Hudak showed that the application-dependent granularity of the memory units (pages) has a significant influence on the efficiency of the implementation. Based on shared memory semantics it produces the same unpleasant network traffic like cache coherency algorithms for shared memory computers, but in a distributed memory system communication is much more expensive.

Some approaches to reduce message traffic use modified consistency semantics. Cheriton's *problem-oriented shared memory* [Che86] defines consistency depending on the requirements of the applications. They can eventually use an outdated version of the date and therefore do not need an actual copy, or are capable of computing it by themselves.

In *distributed database management systems* literature (e.g. [CP84] or [OV91]) topics like replication of data, different consistency models and object migration are discussed, too. But for database systems it is much easier to decide whether the operation has to be executed at the node where the object is located or to migrate the object to the node from where the operation is called, because the query-optimizer can decide on actual statistics about a few well-known (relational database) operations and data structures.

Since typical compute-intensive "grand-challenge" applications are running for a few days or hours, and today's massively parallel computers reliability is even lower than that of workstations, the programming environment should support *fault-tolerance*. Unfortunately most systems do not even have a checkpointing library. In this case the programmer has only two possibilities to deal with system-failures, either he restarts the program from the beginning and loses all preceding work, or he accepts the additional work to write checkpoints and recovery by himself. But even if there is such a library the complexity of parallel programming is increased. Therefore, fault-tolerance should be fully user-transparent. This can be done by *checkpointing of messages* like in [BBG83], [PP83], [JZ88], [SY85] or by using *transactions* [Gra78], [GR92].

In programming languages like Aeolus [WL86] and Argus [Lis88] fault tolerance is provided by using *atomic actions* and *reliable objects*. Atomic in this context means that actions have all-or-nothing semantics – either they run successfully and all updates on objects will become visible, or elsewhere all updates are rolled back to their previous state. Although in both languages actions can be distributed over different nodes, they were rather built for implementing system software for distributed systems than for parallel programming. Therefore, it is not necessary to migrate objects dynamically. This fits well if the objects are e.g. file-systems, but in parallel systems this makes dynamic load-balancing very difficult [Sch91].

Like in Aeolus and Argus the basic unit of execution in Pact[2] is the action. This helps providing user-transparent fault-tolerance. The actions' common data are organized in shared variables which are dynamically distributed by Pact's runtime system. This is done by using a virtual shared memory algorithm adapted from Li's and Hudak's distributed page ownership

---

[1] Some call it distributed shared memory.

[2] Pact is a synonym for <u>p</u>arallel <u>act</u>ions.

algorithm [LH89]. By embedding Pact in familiar procedural programming languages like ANSI-C, a parallel programming language is provided which is easy to learn.

In the next section the programming model is described in more detail. In section 3, Pact's extensions to procedural languages are presented. In section 4, there is a short presentation of the runtime system and the visualization tool *pactview*. Section 5 shows conclusions and further work in the Pact project.

2 Pact's Programming Model

The basic concept of Pact is to give programmers an environment which is almost as simple as that of sequential machines. This means that supplementary work for using parallelism, like synchronization of accesses to shared data, scheduling and load balancing is done by Pact's runtime system. Most people with practical experience in writing parallel programs will agree that the explicit synchronization of accesses to shared data and the coordination of parallel execution with barriers, semaphores, message passing, etc. are difficult. (As an example how difficult it can be, refer to the buffer manager code in [GR92]. In this code the access to a database buffer is synchronized with shared and exclusive semaphores.) Often complex parallel programs have completely unexpected effects or run into deadlocks. Synchronization failures are usually very hard to find, because they occur only in a special execution order. If the program is debugged using a source level debugger or print-statements the failure often cannot be found, because the execution order is modified.

Pact's approach is to avoid everything that is not really necessary to write efficient parallel programs and offer everything that helps doing this. This results in the following requirements to the parallel programming model:

- There is a need for a notion for *expressing parallelism* and to *coordinate* the execution of parallel actions.
- Actions[1] running in parallel should not deal with parallelism. They should only contain sequential code. This makes programming simpler and therefore helps avoiding programming faults.[2]
  Note that on this level it is not useful to think about which node should execute which part of the program. Therefore, typical constructions like `if (mynode() == k) do_something()` create programs which are difficult to understand and often cannot be executed on varying numbers of processors.
- Parallel actions need the possibility to *share data*. To comply with the above requirement this should be fully user-transparent.
- For providing this transparency it is necessary to guarantee *consistency* on shared data in the same transparent way by the runtime system. Therefore, there is no need for semaphores, locks, monitors, etc. in the programming language.

Similar to Aeolus and Argus, Pact's basic unit of execution is an *action*. Actions in Pact are written like usual procedures or functions. This means that they contain exclusively sequential code and that they are called using reference or value-parameters like in Pascal or ANSI-C.

[1] Some call it *tasks*, others *threads of control*.

[2] In [GR92] "KISS: keep it simple, stupid" is one of the general principles for building fault-tolerant software.

Similar to transactions, Pact's actions have the property of *atomicity.* They have all-or-nothing semantics – either an action runs up to its successful termination, and all its results and its changes to global variables get visible, or otherwise all its effects are rolled back to their previous state. If for example an action runs on an internal error (e.g. division by zero), this is reported to the main program. Now the main program may react on this error by executing the action once again with modified parameters. This is possible, because no effects of the faulty action are visible to other actions – the runtime system undoes them. If a processor fails, the runtime system is recovered and all actions which are lost – because of the processor failure – can simply be done again. This is the mechanism to provide user-transparent fault-tolerance. It is so easy here because of the atomicity of actions.

Actions can use *shared-variables* for data needed concurrently by some actions. Because consistency on shared data is provided by the runtime system, it is not necessary to think about concurrent accesses of other actions. This avoids situations like that in Figure 1. where two tasks concurrently updating shared variables compute two different results, depending on their execution order. In case 2, the update of task t_2 gets lost (this effect is called "lost updates"). In the absence of explicit synchronization on these variables the order of the updates is not in the user's hand but in that of the operating systems scheduler. This is definitively a synchronization failure in the programs, both should acquire locks on x, but the argumentation here is that these kinds of failures are frequent and therefore synchronization should better be automated. Note that in order of doing automatic concurrency control by the runtime system shared-variables do not have shared memory semantics. Therefore, it is not possible to update the same variable at the same time from two different actions. Shared variables are used exclusively to provide global data, communication and synchronization between actions are done by other mechanisms.

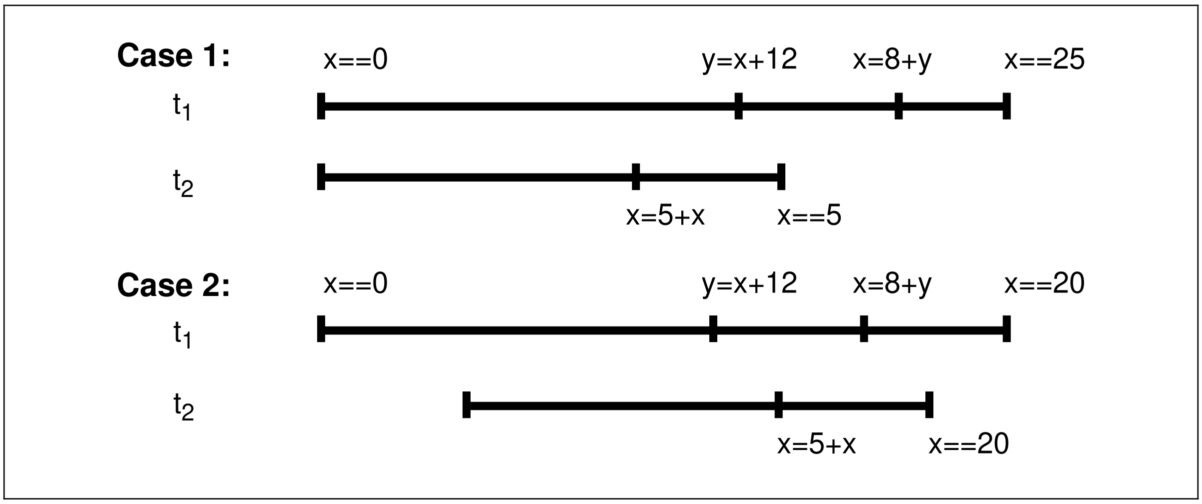

Figure 1. Lost updates caused by synchronization faults.

Following the requirements above, coordination of parallelism in Pact is completely done in the main program by defining the execution order. For simple parallel execution orders this can be done by using forall or pardo statements. In Figure 2., an execution graph for data-parallelism and function-parallelism is shown. On the left side the function f() is executed N times in parallel. Each function call is done with a different parameter. Because the same function works on different data, this is called *data-parallelism*. On the right side the functions f(), g(), and h() are executed in parallel. As in this case different functions are running in parallel, this is called *function-parallelism*. To define more complex execution orders like in Figure 4. a model similar to the precedence-graphs in [Sch91] is used.

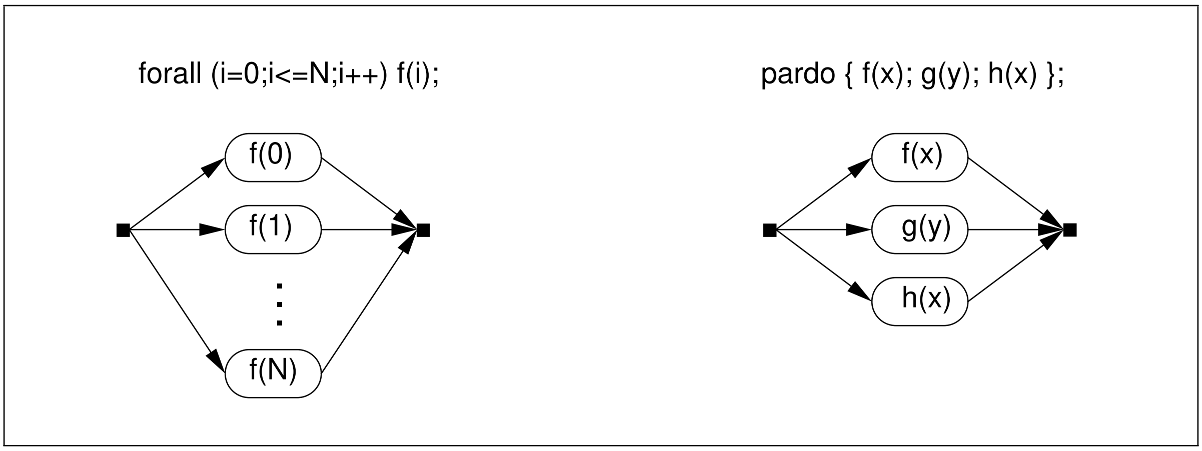

Figure 2. Data-parallelism vs. function-parallelism.

The ideas of these graphs are explained in the graph in Figure 3. The rounded rectangles symbolize the actions with the names in them corresponding to the actions' names. Rectangles which are drawn above each other represent actions which are running in parallel. Each rectangle has some arrows pointing into it and some arrows pointing out. These arrows reflect the control flow of the execution. The small squares with names besides them are called *events*. The idea is that if an event has been triggered by the termination of all preceding actions the following actions can be executed. In this example, after the action f(x) has terminated the event ParBegin is triggered. This starts the parallel execution of f(y), g(y) and h(y). After the termination of all of them the event ParEnd is triggered which causes the start of e(a,b,c).

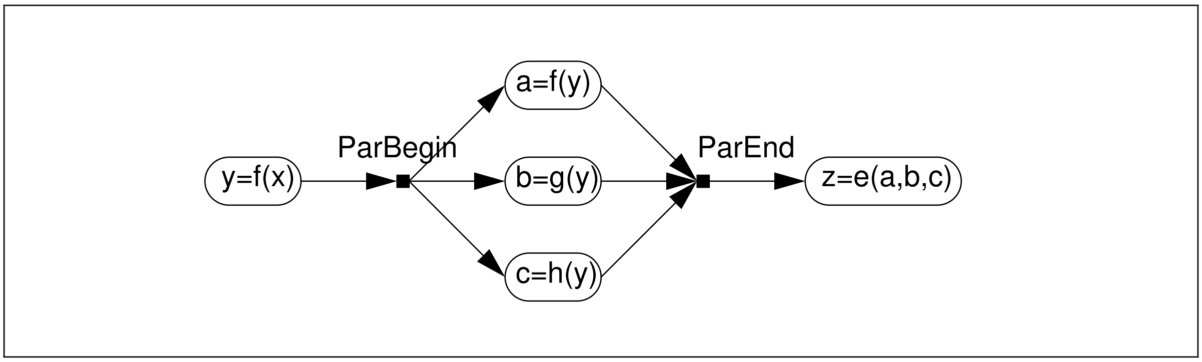

Figure 3. An example precedence-graph.

3 Pact's Extensions to Procedural Programming Languages

Pact is completely embedded in a procedural programming language. This makes it easier to learn parallel programming with Pact and simplifies its implementation, too. Hence, it is only necessary to write a precompiler which replaces Pact's language extensions by Pact's intermediate language library calls (ImPact). The extensions to C are described here in a BNF and by giving simple examples to help the reader's imagination. Pact's keywords are underlined to show the difference to C.

As described above the basic unit of execution in Pact is the action. Following the requirements of the programming model, action-definitions look like usual function-definitions containing only sequential code. The slight difference between usual C function-definitions and the definition of actions is the keyword action in front of the function's head.

```
<definition>            ::=   <action-definition> | <function-definition> | ...
<action-definition>     ::=   action <function-definition>
```

In the following example an action is defined which computes x^y.

```
action double mypower( double x, double y);
{ return( pow(x,y) ); }
```

As mentioned above, Pact has two different possibilities to define the parallel execution order of actions. The simpler way is to use well-known forall and pardo statements. The more complex one uses special actions-calls and events. Events are defined like usual variables with the predefined type event. This looks like a variable definition and it is possible to define vectors and arrays of events, too. In the LU-factorization on page 8 an array of events is used. Events are initialized with 1 by default and the event triggers subsequent execution if its value has been set to 0 by explicitly setting it with the Pact-function preset() or by the termination of a preceding action. To implement execution-plans as in Figure 3., where the execution of action e() has to be started not before all actions f(), g(), and h() have terminated, it is necessary to await all three actions' termination-events. This is done by initializing ParEnd with 3 (with preset() or like in the subsequent example in the event-definition). Each action's termination decrements the event value by one, until it equals 0. Then the event triggers the start of other actions.

Actions are called using an action-call which looks like a call to the "function" action. An action-call has four parameters: Three event-expressions and the action-expression. The first event-expression specifies the start-events (conjunction of events) which are necessary to start the execution of the action. To give more flexibility in defining execution-plans (e.g. non-deterministic execution) disjunctions of events are allowed for start-events, too. The second parameter gives the events which are triggered when the action has terminated successfully. The third event is triggered when the action runs on an internal error (e.g. division by zero or segmentation fault). All event-expressions can be omitted. If the start-event is omitted, the action runs immediately. If the termination or error-event is omitted, nothing special happens, but there is no connection possible to other actions.

```
<action-call>                     ::=   action( [<start-event-expression>],
                                                [<termination-event-expression>],
                                                [<error-event>],
                                                <action-expression> );
<start-event-expression>          ::=   <event-id> && <start-event-expression> |
                                        <event-id> || <start-event-expression>|
                                        <event-id>
<termination-event-expression     ::=   <event-id> && <termination-event-expression> |
                                        <event-id>
<error-event>                     ::=   <event-id>
<action-expression>               ::=   [<variable-identifier> =]
                                        <action-identifier>(<parameters>)
```

The following example defines the execution-order in the example in Figure 3. by using pardo on the left side and action-calls on the right side.

```
y = f(x);                          event ParBegin, ParEnd = 3;
pardo                              action( , ParBegin, , y = f(x));
    {
        a = f(y);                  action( ParBegin, ParEnd, , a = f(y));
        b = g(y);                  action( ParBegin, ParEnd, , b = g(y));
        c = h(y);                  action( ParBegin, ParEnd, , c = h(y));
    }
z = e(a,b,c);                      action( ParEnd, , , z = e(a,b,c));
```

To share data between some actions shared variables can be defined in the main program and in the actions using it. This is done by a special keyword shared before the variable definition, like in [Ost86]. For better efficiency it is possible to partition these shared variables into *dataunits*. With this mechanism the programmer has the ability to define the granularity optimal to the requirements of its program. Dataunits define the unit of concurrency control and transport. This means that locks are acquired for a complete dataunit. If the size of the dataunit fits the access, this reduces the number of lock conflicts to its minimum. Since the unit of transportation for the virtual shared memory algorithm it also a dataunit, problems with the granularity like reported in [LH89] are avoided. This improves the performance of virtual shared memory. The declaration of shared variables is embedded in variable declarations by defining a special storage-class:

```
<declaration>        ::=   ... | <class-or-type> <initialized-declarator-list> ;
<class-or-type>      ::=   [<storage-class>] <type-name>
<storage-class>      ::=   ... | shared
```

In the subsequent example a shared variable for a two-dimensional array is declared. Because the actions using this array are working on small sub-arrays of size 9x9 the size of the dataunit is 9x9, too.

```
shared double Array [N][N] unit [9][9];
```

An example of a parallel program with a corresponding precedence graph illustrating the parallel execution order of the actions is presented in Figure 4. The program implements a parallelized version of the Gaussian elimination algorithm.

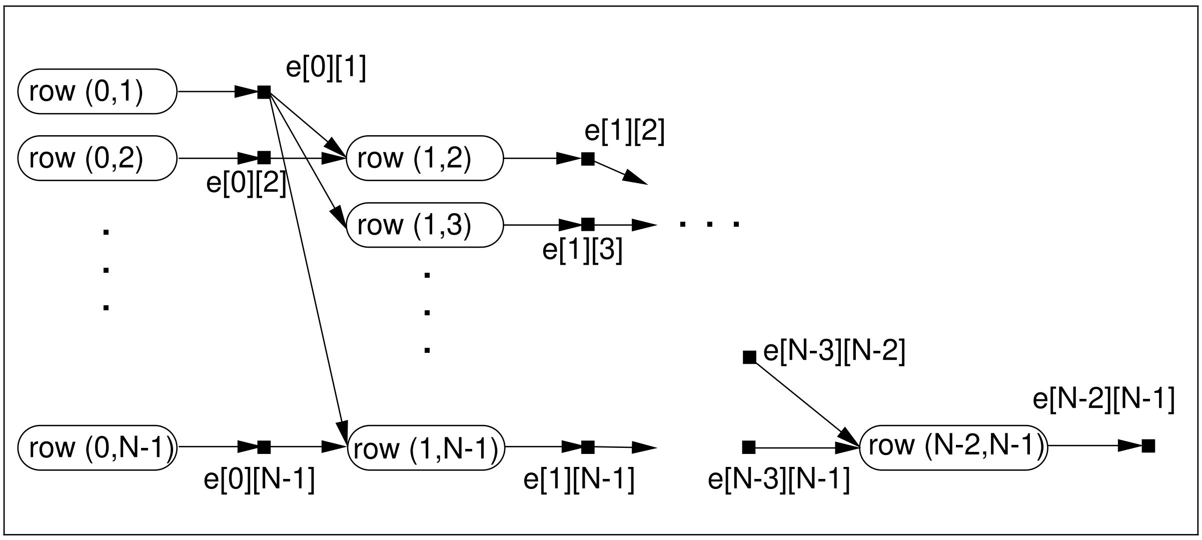

Figure 4. An precedence graph for the parallel LU-factorization using the Gaussian elimination algorithm.

Parallel LU-factorization in Pact:

```
shared double A[N][N+1] unit A[1];   /* dataunit is a row of the extended matrix A       */

action row( int i, int j)              /* computes a new row j from the old row j and row i */
{                                      /* read access to row i, write access to row j        */
    int k;
    A[j][i] = A[j][i] / A[i][i];
    for (k=j+1; k<=N; k++)
        A[j][k] = A[j][k] - A[j][i] * A[i][k];
    A[j][N] = A[j][N] - A[j][i] * A[i][N];      /* computes the vector b                    */
}

main()
{
    int i, j;
    event e[N-1]N];                    /* event matrix                                      */
                                       /* by default initialized for single events          */
    initialize_system(A);              /* equation system is loaded into A                  */
    for (j=1; j<=N-1; j++)
        action( , e[0][j], , row(0,j); /* action-call without a start-event starts immediately*/
    for (i=1; i<=N-2; i++){
        for (j=i+1; j<=N-1; j++)
            action( e[i-1][i] && e[i-1][j], e[i][j], , row(i,j));
    }
    await( e[N-2][N-1]);               /* wait on the termination of the last action        */
    store_results(A);                  /* store the resulting matrix somewhere              */
}
```

4 Pact's Runtime System

The prototype's runtime system which is currently implemented on a Tandem shared-nothing multiprocessor system is a process system as illustrated in Figure 5. The architecture is based on two major decisions:

- Shared data is emulated by a *virtual shared memory* algorithm.
- No dynamic process creation, but preallocated *server-processes* are used.

The decision to use server processes is made, because in absence of *threads* [ea86] process creation is too expensive for dynamic process creations. Instead of this, a preallocated number of server-processes for each type of action is started at the start-up of the runtime system. During execution of the parallel program new processes are started only as part of recovery. A virtual shared memory algorithm is necessary for an implementation of logical shared data on a distributed memory architecture. Note that more than one server process of each server type is running on one processor node. This is necessary because during the time a server process is waiting on a dataunit another server can execute on this node.

After the runtime system has been started the execution plan, which is defined in the Pact main program using actions and events, is passed to the Supervisor which controls the execution on all involved processor nodes and maps the actions to the server processes. The Node Manager controls the accesses to virtual shared data which is physically shared between the

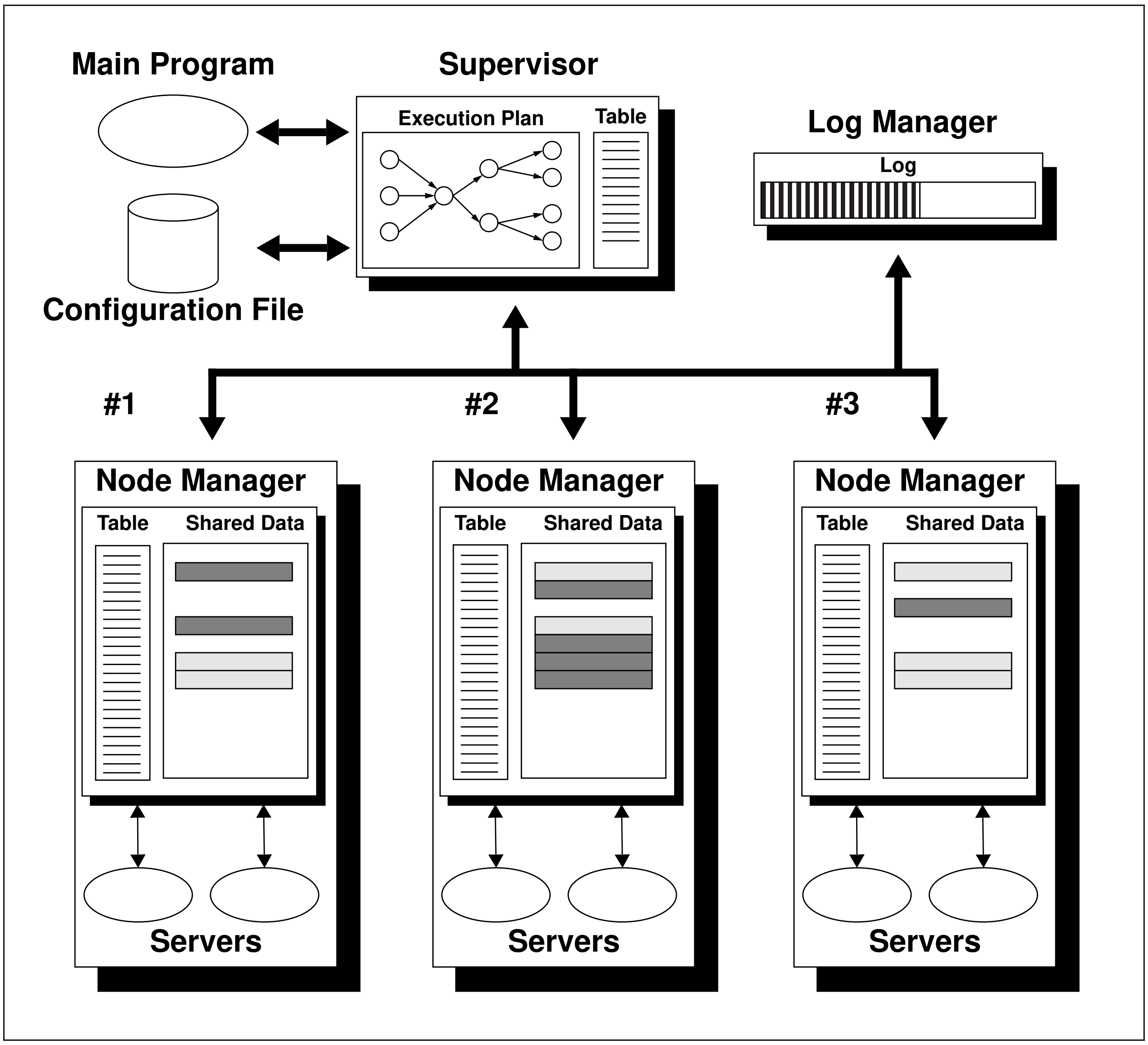

Figure 5. Architecture of the *Pact* runtime system

server processes on the processor. The Log Manager logs the actions in the runtime system in order to enable recovery.

The *Supervisor* distributes the actions to the corresponding servers on the processor nodes. This is done depending on the execution plan defined by the main program and the occurrence of start-events. For reasons of efficiency the Supervisor uses a data-dependent load balancing strategy. This means that it uses compiler generated or user given information about the dataunits an action wants to use, in order to decide if the action should run on the node which already has the necessary dataunits. A problem here is that depending on the virtual shared memory algorithm, it is impossible for the Supervisor to have actual information about the dataunits' locations. For reasons of efficiency the Supervisor cannot get involved in every data transfer between the nodes, but it gets actual information whenever an action terminates. At this time, the action server sends its results together with the identifiers of the dataunits, it has used, to the Supervisor in one message. Additional to this, the Supervisor's duties are those of a recovery manager. Whenever a component of the Pact runtime system fails, the Supervisor recovers all lost data and actions as described in more detail in [Mai93].

In order to recover system failures it is essential to hold an actual copy of all dataunits on the Log. The *Log Manager* coordinates this logging activity. Whenever an action gets ready during its *commit phase*, it has to send all changed dataunits and its results to the Log Manager. When this is done successfully and all locks are released, the action's results are sent to the Supervisor. If a node crashes after this commit, the Supervisor can recover the actual version of all lost data from the Log. Another situation in which the Log is needed is when actions get into a deadlock situation or when the action fails because of a software fault. If the system detects a deadlock, it *aborts* one of the participated actions and restores the unchanged data on the node. This is equal to a rollback of all of the action's updates. Although the Log Manager is centralized in this prototype, it is easy to distribute it like it is done in some database management systems.

The *Node Managers* administrate in a fully distributed way the system's virtual shared data. This is emulated by an algorithm adapted from Li's and Hudak's *dynamic distributed page ownership* algorithm [LH89]. For this purpose each Node Manager uses a table in which the actual *probable owner* of every dataunit is stored. In this table the locks are managed, too. Whereby each Node Manager manages exactly the locks of those dataunits which are owned by itself. If a server requests a dataunit, it sends a message to its Node Manager. The latter gives it a pointer to the dataunit in the nodes' local memory which is shared between the Node Manager and the servers on the node. If the Node Manager cannot satisfy the request from its local server, it asks the probable owner of the requested unit. If this is not possible because of lock conflicts, the request is queued. Tandem's communication protocol which notes the sender, if the recipient of a message does not exist, makes it possible for the Node Manager to detect system failures. Whenever it discovers a failure, it informs the Supervisor which coordinates all necessary recovery activities. Note that in opposite to Li's and Hudak's algorithm user-defined *dataunits* instead of virtual memory *pages* are passed through the system. This reduces the number of messages and the message length, because the granularity of the dataunits fits exactly with the granularity of access. If the program needs only a few bytes, only these few bytes are sent to (and locked for) the server. If the program needs thousands of bytes which need many pages, the dataunit is sent in only one message with only one message latency.

For a more detailed description of the runtime system and the virtual shared memory algorithm refer to [Mai92].

To help the programmer debugging and controlling the parallel execution of his Pact program a X-Window based visualization tool *Pactview* [Has92] is included in the programming environment. By turning the runtime system in "trace" mode a collector process is started on the parallel computer which collects control messages from the Supervisor and the Node Managers and sends them via TCP/IP to a workstation, where the visualization is done. Pactview supports two major modes: on-line and off-line. In on-line mode the ongoing execution is visualized. In this mode it is necessary to compute dynamically the layout of the execution graph which can be very compute intensive for large graphs. Therefore, it is possible to vary some layout parameters e.g. to switch off the centralization of the graph, as it is done in Figure 6. and Figure 7. In off-line mode it is possible to replay or to step forward and backward through an execution previously recorded.

Figure 6. shows the execution of a parallel LU-factorization. Similar to the graphs before rounded rectangles show actions with their parameters and squares and arcs show events connecting them. Actions can be in four different states. A solid black rectangle shows that the action has already terminated. White shows that the action has not executed yet. Striped actions are running and actions with plaids are waiting on dataunits. Since graphs can get very big it is necessary to zoom in and out in a very flexible way. Figure 7. shows the global view of an execution graph using the "world button".

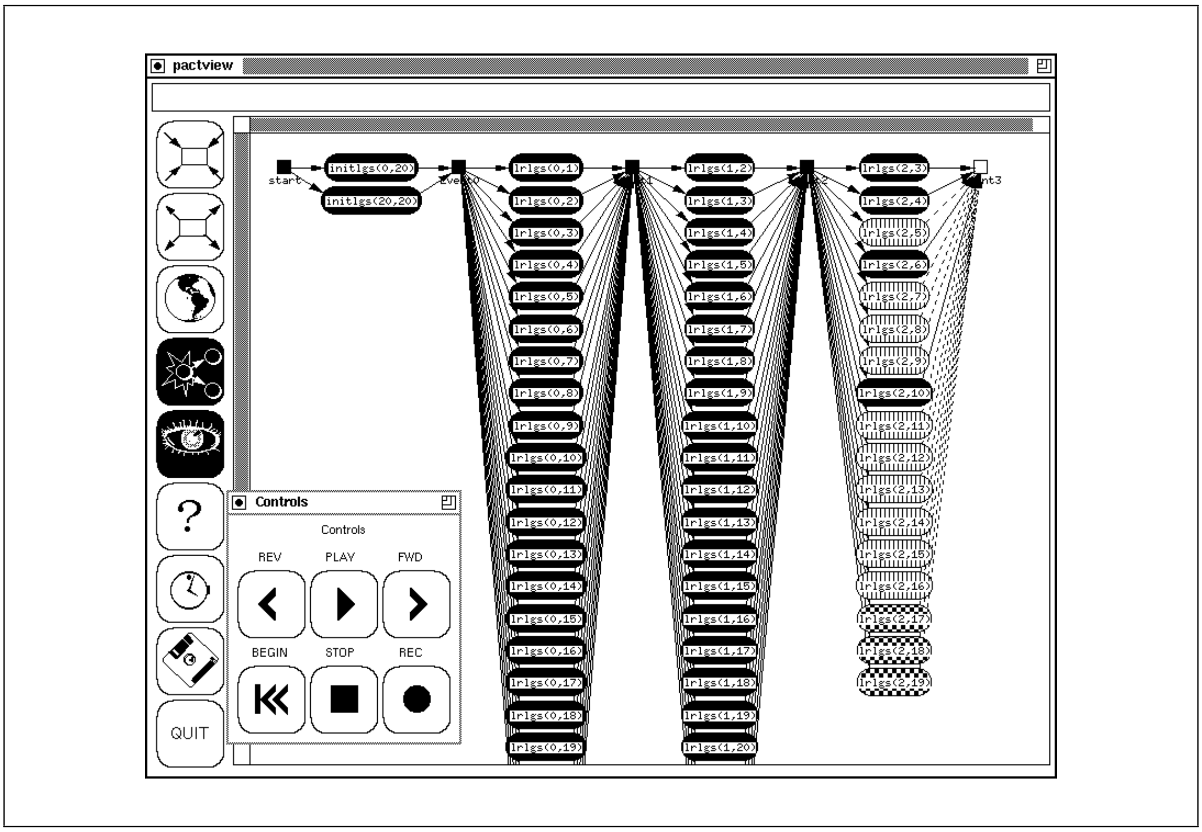

Figure 6. Screen-dump showing Pactview's visualization of a parallel LU-factorization.

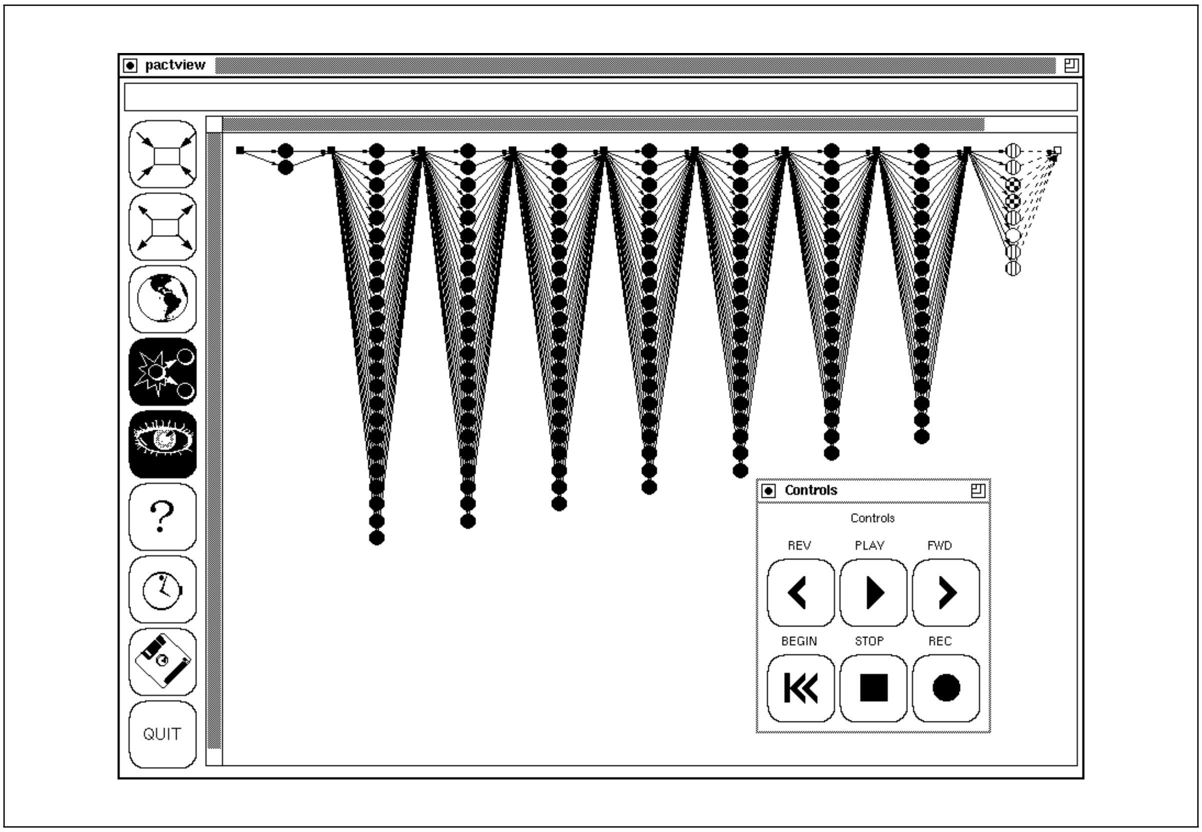

Figure 7. Global view of an execution graph for parallel LU-factorization.

5 Conclusions and Further Work

In this article, a new approach to parallel programming is presented which not only makes parallel programming very easy, but also provides user-transparent fault-tolerance. Programming ease in Pact is obtained by the programming model. To reduce the complexity of parallel programming the only place where it is necessary to deal with parallelism in Pact is the main program, where the parallel execution order is defined. This can be done because synchronization between parallel actions is either done by defining the execution order, or if it results from the coordination of concurrent accesses to shared data, it is completely automated.

User-transparent fault-tolerance in Pact is based on the atomicity of actions, which enables the repetition of failed actions. But atomicity helps not only against system failures but also gives Pact the possibility to handle programming faults in actions. This enables Pact to overcome transient software faults.

To provide shared data even on a parallel computer with distributed memory, a virtual shared memory implementation in the runtime system controls the distribution and replication of shared data which is partitioned into user-defined dataunits for better efficiency. To ensure highest performance the runtime system schedules the actions to server-processes using a dynamic data-dependant load-balancing strategy.

Talking about performance, it is necessary to note that last but not least the performance of the parallel program depends on the programmer. He has to choose the best parallel algorithm for his problem. Although the goal of Pact is to give the programmer a view to the parallel machine which abstract from most hardware details (distributed memory, number of processors, etc.), to get highest performance the programmer has to choose a granularity of parallelism which fits with the underlying hardware and he has to distinguish between real shared memory and virtual shared memory. The latter is necessary because the access to real shared memory is a magnitude less expensive than the access to virtual shared memory.

As mentioned above, the actual prototype of the Pact programming environment is implemented on a Tandem multi-processor system, but the architecture is designed to run on any other distributed memory computer as well. Only the recognition of failures, which is a precondition for implementing fault-tolerance, is much easier on a Tandem system than on most other parallel computers because it is provided by Tandem's Guardian operating system.

In the current prototype concurrency control is done by simple shared and exclusive locks. To keep the protocols simple these locks are held up to the actions' end (commit). This does not result in performance problems if the parallelism in the program is higher than the number of nodes and if lock conflicts are rare – in most applications these conditions are fulfilled – but if there is a "hot spot" dataunit there is a danger of lock contention which results in serialization. To reduce this danger, the implementation of additional concurrency control mechanisms (e.g. optimistic concurrency control [KR81], [RT90]) with weaker – user defined – consistency are planned.

The Pact prototype is currently implemented on a system with moderate parallelism. The next version will be implemented on a massively parallel Intel Paragon supercomputer under the OSF/1 operating system.

6 References

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 90–99, Atlanta, October 1983.

[BST89] Henri E. Bal, Jennifer E. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Che86] 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.

[CP84] Stefano Ceri and Guiseppe Pelagatti. *Distributed Databases*. McGraw-Hill, 1984.

[ea86] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *USENIX 1986 Technical Conference*, pages 93–112, June 1986.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[Gra78] 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.

[Han75] P.B. Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, pages 199–207, June 1975.

[Has92] Michael Hasan. Visualisierung von Pact Programmen unter X Windows. Master's thesis, IPVR, Stuttgart University, Stuttgart, September 1992.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, pages 666–677, August 1978.

[JZ88] 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*. ACM, August 1988.

[KR81] H. T. Kung and J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[LH86] 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.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lis88] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Mai92] Joachim Maier. Pact: A Fault Tolerant Parallel Programming System with Virtual Shared Data. Technical report, Fakultät Informatik, Universität Stuttgart, February 1992.

[Mai93] Joachim Maier. Fault Tolerance Lessons Applied to Parallel Computing. In *Proceedings of the Compcon '93*, San Franzisco, February 1993.

[Ost86] A. Osterhaug. Guide to Parallel Programming. Published by Sequent Computer Systems, Inc., Beaverton, Oregon, 1986.

[OV91] M. Tamer Oezsu and Patrick Valduriez. *Principles of distributed database systems*. Prentice-Hall, 1991.

[PP83] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 100–109. ACM, October 1983.

[RT90] Erhard Rahm and Alexander Thomasian. Distributed Optimistic Concurrency Control for High Performance Transaction Processing. In *Proceedings of Parbase*, pages 490–495. IEEE, 1990.

[Sch91] Gerhard Schiele. *Kontrollstrukturen zur Ablaufsteuerung massiv paralleler Datenbankanwendungen in Multiprozessorsystemen*. PhD thesis, IPVR, Universität Stuttgart, Germany, December 1991.

[SY85] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[WL86] 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.