Fault Tolerance for Cluster Computing Based on Functional Tasks*

Wolfgang Schreiner, Gabor Kusper, and Karoly Bosa

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
FirstName.LastName@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at

Abstract. We have extended the parallel computer algebra system Distributed Maple by fault tolerance mechanisms such that computations are not any more limited by the meantime between failures. This is complicated by the fact that task arguments and results may embed task handles and that the system's scheduling layer has only a little information about the computing layer. Nevertheless, the mostly functional parallel programming model makes it possible with relatively simple means.

1 Introduction

Distributed Maple is a portable system for implementing parallel computer algebra algorithms [5]. We have used it for parallelizing various applications in algebraic geometry [6] and executed them in numerous distributed environments. However, as we began to attack larger and larger problems, the meantime between failures became a limiting factor in the applicability of the system.

Most parallel programming environments pursue fault tolerance by *checkpointing* [1,2]. These approaches are complex because they deal with general parallel computations; for special problems much simpler solutions exist [3]. However, also parallel programming models that are more abstract than message passing allow to deal with fault tolerance in a simpler way. In particular, the *functional* programming model has this potential [4].

Distributed Maple runs programs in the imperative language of Maple, but its *parallel* programming model is essentially functional: it provides the ability to spawn function applications as concurrent tasks and to wait for their results (extended by a non-deterministic synchronization facility and single assignment shared data objects). Our primary goal is to extend the scheduling layer of Distributed Maple to provide fault-tolerance in a way that is *transparent* to the application. However, computation layer and scheduling layer are clearly separated by a communication protocol such that the scheduling layer is completely unaware of the actual nature of the computation. Our secondary goal is to preserve this distinction such that the use of the scheduling layer for different kinds of computing engines is not compromised. We thus have to deal with the *limited information* that the scheduling layer has about the computation.

* Supported by grant SFB F013/F1304 of the Austrian Science Foundation (FWF).

2 System and Execution Model

A session comprises a set of *nodes* each of which holds a pair of processes: a *kernel* which performs the actual computation, and a *scheduler* which coordinates the created tasks. The scheduler communicates with the kernel on the same node and with the schedulers on other nodes. The *root* is that node from which the session was established. Initially, a single task runs on the root kernel; this task may create new tasks which are distributed to other kernels and may in turn create new tasks. A kernel emits messages task:$\langle t, d\rangle$ where t identifies the task and d describes it. The task needs to be forwarded to some idle kernel which eventually returns a message result:$\langle t, r\rangle$ where r represents the computed result. When a kernel emits wait:$\langle t\rangle$, this task is blocked until the scheduler responds with the result of t. Thus, if this result not yet available, the scheduler may submit to the now idle kernel another task; when this task has returned its result, the kernel may receive the result waited for or yet another task.

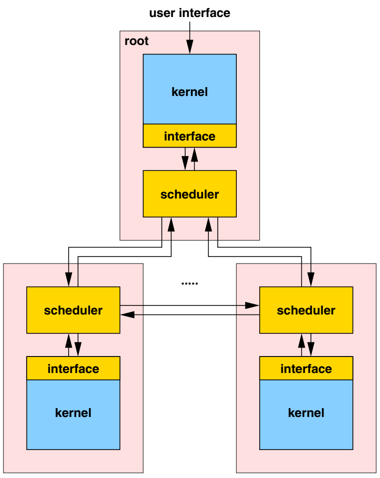

Fig. 1. System Model

A task identifier t encodes a pair $\langle n, i\rangle$ where n identifies the node on which the task was created and i is a running index. The node n thus serves as the rendezvous point between node n' computing the result of t and node n'' requesting this result. When the scheduler on n receives a task:$\langle t, d\rangle$ from its kernel, it thus allocates a result descriptor that will eventually hold the result r; the task itself is scheduled for execution on some node n'. When a kernel on some node n'' issues a wait:$\langle t\rangle$, the scheduler on n'' forwards a request:$\langle t, n''\rangle$ to n. If r is not yet available on n, this request is queued in the result descriptor. When the kernel on n' eventually returns the result:$\langle t, r\rangle$, the scheduler on n' forwards this message to n, which constructs a reply:$\langle t, r\rangle$ and sends it to n''.

3 Logging Results: First-Order Tasks

Our first step towards fault tolerance is to log task results on stable storage. If the session fails, we can thus re-run the session and read the logged results without re-executing the corresponding tasks. Our only assumption is that the root has access to a writable file system; this file system implements the stable storage and the root becomes in charge of maintaining the log. We now restrict our consideration to programs whose tasks are *first order*: no task description d and no task result r contains any task identifier t, i.e., task identifiers are not passed as parts of task arguments and or parts of task results. The logging mechanism utilizes the fact that the root is in charge of task distribution: every new task:$\langle t, d\rangle$ is forwarded to the root which eventually assigns it to some node for execution. The result logging and failure recovery operations are as follows:

Logging When the root receives a task:$\langle t, d\rangle$, it computes a hash code $h(d)$ and appends to file `taskid.`$h(d)$ the task identifier t. Then the root starts an asynchronous thread to write the task description d into a new file `descr.`t. When a node sends a result:$\langle t, r\rangle$ to some node n different from the root, it forwards a copy to the root. When the root receives this result, it creates an asynchronous thread to write the task result r into a new file `result.`t. All data are written in a format that enables a reader to recognize incomplete writes. At any time, `taskid.`$h(d)$ holds a sequence of task identifiers t for which there may exist description files `descr.`t and/or result files `result.`t. When the session terminates without failure, the files are discarded.

Recovery When a session is re-started after a failure, the root may receive from a node n a task:$\langle t, d\rangle$ such that a file `taskid.`$h(d)$ exists. If for some complete task identifier t' in this file there exist file `descr.`t' with a complete description identical to d and file `result.`t' with a complete result r, the root need not schedule the task but may immediately return result:$\langle t, r\rangle$ to n. The comparison of task descriptions is required because task identifiers are not valid across sessions; since the result r of a task only depends on its description d, the identity of descriptions ensures the identity of results.

4 Logging Results: Higher-Order Tasks

Assume that a task t creates another task and embeds its identifier t' in result r. If r is logged and the session fails, in the recovery session this result may be read from the log such that task identifier t' is re-created. A task may subsequently issue a wait:$\langle t'\rangle$ referring to a no more existing task or, even worse, to a task that computes a different result than in the failed session. Similar situations may occur if t' is passed as an argument to another task.

To support such *higher-order tasks*, we introduce a *session identifier* which distinguishes task identifiers from different sessions. In an original session, the session identifier is initialized to 0 and a file `session` is written with content 0. In a recovery session, the previous identifier s is read from `session`, the new identifier is taken as $s + 1$ and overwrites `session`. If the recovery session also fails, a new recovery session may be initiated. A *task identifier* now encodes a triple $\langle s, n, i\rangle$ that embeds the number s of the session in which the corresponding task was created. We generalize the mechanism of the previous section as follows:

Logging As long as the description of a task t created on node n is not completely logged, the root does not schedule the task for execution and does not start the logging of any other task created on n. As long as n has not received the confirmation that the description has been logged, it does not return the result of t to any other task. In this way, we make sure that the identifier of a task is not propagated across tasks before its description has been logged such that a later recovery session can restart the task.

Recovery The root is in charge of tasks whose identifiers refer to previous sessions. If a kernel on node n issues a wait:$\langle t\rangle$ where t refers to a previous session, the scheduler on n sends a request:$\langle t, n\rangle$ to the root. If the root receives this request, it looks up whether it holds a result descriptor for t; if yes, it responds with the result or, if this is not yet available, queues the request in the descriptor. If the root does not hold a result descriptor for t, it creates one and queues the request there. It then looks up file `result.`t for the result of t. If this file exists and holds a complete result r, the scheduler writes r into the descriptor and responds with reply:$\langle t, r\rangle$. Otherwise, the scheduler looks up `descr.`t for the description d of t. The scheduler creates a new task:$\langle t, d\rangle$ which is handled as usual. When a kernel issues a result:$\langle t, r\rangle$ for a task t of a previous session, the scheduler forwards it to the root.

5 Tolerating Node Failures

We have also introduced a mechanism that enables a session to cope with faults *without* aborting. We restrict our attention to the scenario where a non-root node becomes unreachable (stop failure) and the root continues operation with the remaining nodes (if the root fails, the session also fails). A necessary condition to detect this failure is that the root cannot contact a node for a certain period of time. We thus let the root periodically check whether a message has been recently received from every node and, if not, send a ping message that has to be acknowledged. If no acknowledgement arrives within a certain time bound, this node is considered as *dead*. However, we must assume that an allegedly dead node may still send messages to the root or to any other node. Thus, when the root designates a node as dead, it informs all other nodes correspondingly: every node closes the connection to the dead node and ignores any buffered messages from this node.

There are two main problem that the root now has to deal with:

1. the management of all result descriptors that have been stored on the dead node, and
2. the rescheduling of all tasks that were executing on the node at the time of its alleged death.

Since the root is in charge of task scheduling, the root sees every task created in the session. Furthermore, by the logging mechanism discussed in the previous sections, the root sees every result computed in the session. For every node n, the root can therefore maintain two sets T_n and S_n:

1. T_n denotes all tasks scheduled on n; for a subset T_n^{r} the results are available (in the logging files). All tasks in $T_n - T_n^{\mathrm{r}}$ have to be executed again; the root puts them back into the pool of tasks to be scheduled for execution.
2. S_n denotes all tasks whose descriptors are stored on n; for a subset S_n^{r} the results are available (in the logging files). The root becomes the owner of elements in S_n; it allocates the corresponding result descriptors and, for all elements of S_n^{r}, fills them with results.
   Subsequently, every node will send requests for a result in S_n to the root. However, there may be still outstanding requests sent to n but not yet answered at the time of its death. Every node n' therefore holds a table R_n of all request:$\langle t, n'\rangle$ messages sent to node n but not yet answered by a reply:$\langle t, r\rangle$. When n is marked dead, the node re-sends all messages in R_n to the root which will eventually answer them.

Thus all tasks scheduled on an eventually dead node n are executed (possibly on a different node n') and every descriptor originally housed by n finds a new home on the root to which all open and all future requests are redirected.

More details on the fault tolerance mechanisms can be found in the long version of this paper on `http://www.risc.uni-linz.ac.at/software/distmaple`.

References

1. A. Clematis and V. Gianuzzi. CPVM — Extending PVM for Consistent Checkpointing. In *4th Euromicro Workshop on Parallel and Distributed Processinge (PDP'96)*, pages 67–76, Braga, Portugal, January 24–26, 1996. IEEE CS Press. 712
2. G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353, Balatonfüred, Hungary, September 10–13, 2000. Springer. 712
3. A. Iamnitchi and I. Foster. A Problem Specific Fault Tolerance Mechanism for Asynchronous, Distributed Systems. In *29th International Conference on Parallel Processing (ICPP)*, Toronto, Canada, August 21–24, 2000. Ohio State University. 712
4. R. Jagannathan and E. A. Ashcroft. Fault Tolerance in Parallel Implementations of Functional Languages. In *21st International Symposium on Fault-Tolerant Computing*, pages 256–263, Montreal, Canada, June, June 1991. IEEE CS Press. 712
5. W. Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1998. http://www.risc.uni-linz.ac.at/software/distmaple. 712

6. W. Schreiner, C. Mittermaier, and F. Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. In *Euro-Par 2000, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin. 712