

Dependable High Performance Computing on a Parallel Sysplex Cluster

Wolfgang Blochinger
Symbolic Computation Group*
University of Tübingen
D-72076 Tübingen, Germany

Reinhard Bündgen
IBM Deutschland Entwicklung GmbH
D-71003 Böblingen, Germany

Andreas Heinemann†
Symbolic Computation Group
University of Tübingen
D-72076 Tübingen, Germany

Abstract *In this paper we address the issue of dependable distributed high performance computing in the field of Symbolic Computation. We describe the extension of a middleware infrastructure designed for high performance computing with efficient checkpointing mechanisms. As target platform an IBM Parallel Sysplex Cluster is used. We consider the satisfiability checking problem for boolean formulae as an example application from the realm of Symbolic Computation. Time measurements for an implementation of this application on top of the described system environment are given.*

Keywords: High Performance Computing, Efficient Checkpointing, IBM Parallel Sysplex Cluster, Symbolic Computation, Parallel Satisfiability Checking

1 Introduction

The need for high performance computing is no longer restricted to problems from engineering applications that typically result in computations on huge matrices filled with floating point numbers. Also many problems from

the realm of Symbolic Computation can only be tackled with the power provided by parallel computers. In contrast to many of the number crunching problems mentioned above problems from Symbolic Computation cannot easily be partitioned into many subproblems of similar type and complexity. Problems from Symbolic Computation are typically parallelized using a (recursive) divide and conquer style that partitions a task into different subtasks of different complexity. In some cases even aspects of search parallelism come into play. I.e., different processes search in parallel through an (unbalanced) search tree. Clearly the parallelization of applications from the field of Symbolic Computation requires an adequate hardware and software environment. In addition to the described difficulties, we have to face extreme long running times. In this context dependability gets most important and will be the focus of this paper.

The dependability problem can be attacked by two different approaches: *fault prevention* and *fault tolerance* [1]. In the presented work, the issue of fault prevention is addressed by providing a simple and high level programming paradigm by a special middleware component. This frees the programmer from dealing with low level system details leading to compact and clearly structured programs and thus reduc-

*<http://www-sr.informatik.uni-tuebingen.de>

†The contributions of this author are part of his Diploma thesis sponsored by the IBM Deutschland Entwicklung GmbH.

ing the likelihood of software faults. Additionally, we use hardware that exhibits outstanding availability and reliability features reducing the risk of hardware faults.

Fault tolerance is commonly defined as preventing errors from becoming failures. In the context of high performance computing the definition of the term failure clearly must include the issue of gaining poor or no speedups in the presence of errors. Therefore the fault tolerance mechanisms presented in this paper are primarily targeted towards this point.

2 Hardware and Network Architecture

The IBM Parallel Sysplex architecture [2] combines up to 32 instances of the OS/390 operating system to a single cluster. It provides for fast communication paths among the system, access to memory (caches, queues, locks) that is shared by all systems (coupling facility), a common timer and access to shared disks. Each of the OS/390 systems as well as the coupling facilities can either run on a dedicated S/390 system or on a logical partition within an S/390 system. The S/390 architecture is an SMP architecture with up to 12 processors and excellent I/O capabilities. Both the S/390 hardware, the OS/390 operating system and the Parallel Sysplex cluster architecture are primarily designed for high reliability, high availability and good performance on mixed workloads. In addition it provides a perfect architecture for mid-size parallel programming exploiting both shared memory parallelism and distributed computation.

OS/390 is the successor of MVS. It includes support for Parallel Sysplex exploitation like the Work Load Manger (WLM) and the UNIX System Services subsystem, that provides for a UNIX API and a UNIX user interface. To support the development of parallel applications OS/390 UNIX System Services already include the Parallel Environment, a port of the MPI implementation known from the SP2 parallel computers.

In addition to a proper Parallel Sysplex environment we also ran experiments in an heterogeneous network consisting of both a Parallel Sysplex and workstations running under Windows NT. Combining OS/390 and Windows NT is particularly challenging because both systems use different data formats for standard data types (e.g., EBCDIC vs. ASCII and Hex-Float vs. IEEE floating point).

3 Middleware for High Performance Computing on Parallel Sysplex Clusters

As underlying middleware for building parallel applications on a Parallel Sysplex cluster the Distributed Object-Oriented Threads System (DOTS) [3] is employed. DOTS was originally intended for the parallelization of applications belonging to the field of Symbolic Computation (we give an example in Section 5), but it has also been successfully used in other application domains such as computer graphics [4] or computational number theory [5].

Since DOTS is designed to run on a cluster consisting of shared memory multiprocessor systems it matches ideally the hardware setup of a Parallel Sysplex.

The main idea of DOTS is to provide a uniform and high-level programming paradigm on such kinds of hierarchical multiprocessor systems. It releases the programmer from the burden of dealing with conceptually different paradigms (i.e. message passing and shared memory threads) within a single parallel application and from getting in touch with low-level (often platform dependent) system details. This approach leads to compact and clearly structured programs and therefore helps to improve program correctness.

Basically DOTS provides object-oriented asynchronous remote procedure call services accessible from C++ through an easy-to-use *fork/join* threads-style programming API. It supplies primitives for (remote) thread creation (`dots_fork`), thread joining (`dots_join`) with join-any semantics when applied to

a thread group, and thread cancellation (`dots_cancel`). Additionally it supports object-serialization for parameter passing.

Each DOTS application consists of a set of processes running an instance of the parallel program and a Commander process. The role of the Commander process is to coordinate the communication among the parallel processes. It keeps track of the workload of each process and schedules each `dots_fork`-request accordingly. In addition the Commander provides a controlling interface for the user of a DOTS application.

Figure 1 shows a pseudo-code fragment and the resulting distributed threads of control mapped to three nodes. The execution starts at node B. The entry point for the application is the `main` routine. The initial process is called the *root process*. This is the only process running completely through `main`. All processes become compute server processes for later `dots_fork` calls. Non-root processes are allowed to fork new threads as well, taking the client role (for clarity, this is not shown in Figure 1). It is up to the programmer whether the root process acts as a master and dispatches work to other hosts, or whether it takes no special role, e.g. in a distributed divide-and-conquer scheme.

The implementation of DOTS is based on the Adaptive Communication Environment ACE [6], which is available for many platforms. So far DOTS has been used on Windows NT, Solaris, IRIX, Linux and OS/390 UNIX System Services. It provides a homogeneous programming paradigm on heterogeneous clusters based on nodes running these systems. A complete description of DOTS is given in [3].

4 Dependable Parallel Computing with DOTS

In this section we address the issue of extending DOTS with a checkpointing mechanism suitable for high performance computing.

In many approaches to checkpointing the whole system state is saved in a checkpoint

at predetermined times. This strategy is adequate for many application domains and from a programmer's point of view it is often considered to be very desirable, since checkpointing and restart can be carried out completely transparent. However in the field of high performance computing this solution has some drawbacks. Periodically saving the whole, usually very extensive system state imposes an unacceptably high overhead to all program runs, including failure free runs which are the normal case. Moreover, in the case of a failure a restart is also quite expensive, since reconstituting a consistent system state out of the checkpointed raw system states is not a trivial task [7].

We describe a different approach to checkpointing that is more suitable for a high performance computing environment like DOTS. Its design goal is to minimize the overhead imposed by the checkpointing mechanism, both in terms of system resources and human resources, i.e. programming overhead. The main idea is to take into account the specific execution pattern of DOTS and additional knowledge about the actual application when deciding about the content of a checkpoint and the points in time when checkpointing is to be carried out. This is realized by two orthogonal approaches for adding checkpointing to DOTS. The first one, referred to as *implicit checkpointing* (see Section 4.1), is completely transparent from a programmer's point of view, no application code modifications are necessary. The second one, named *explicit checkpointing* (see Section 4.3) adds three new API calls, allowing the programmer to explicitly control the granularity of the checkpoints and consequently to control the imposed overhead. According to the particular needs of the application, the programmer can use none, one or both of these checkpointing modes.

Another design criterion that has to be considered is where the checkpoint data associated with a DOTS thread should be stored on stable storage. Basically checkpoints can be stored locally on the node that is executing the thread (provided it has dedicated local stable storage),

```

main() {
    dots_fork( func1, arg1 );
    dots_fork( func2, arg2 );
    ...
    result1 = dots_join(func1);
    result2 = dots_join(func2);
}

```

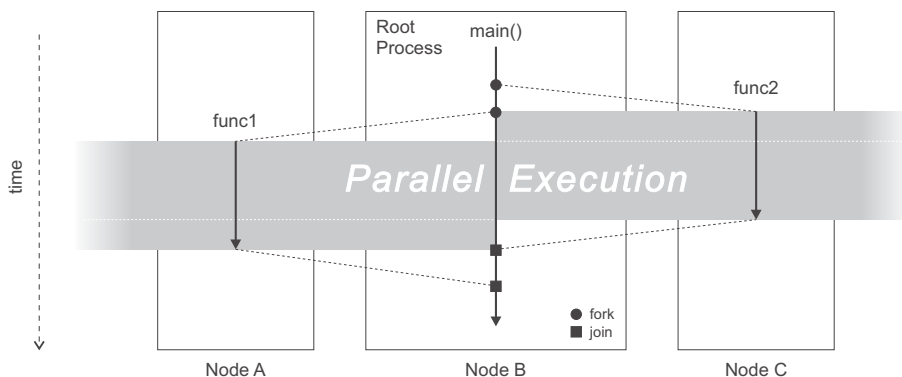


Figure 1: Distributed Threads of Control in DOTS

or all checkpoint data can be collected at a central storage place. Despite the increased communication overhead, in the context of DOTS, a central checkpoint mechanism is more appropriate for the following reasons. First, the Commander is the natural central collector for checkpoint data. Second, in the case of a non-transient fault of a node a DOTS thread can easily be remapped to a different node as before, starting with its last checkpoint available from the central checkpoint store. Third, we can run the Commander on an OS/390 system that is renowned for its high availability. In particular storing the checkpoints on a shared disk adds to the robustness of the system because it allows to restart a parallel application from an alternative system.

4.1 Implicit Checkpointing

For realizing implicit checkpointing we have adopted a technique from Symbolic Computation called *function memoization*. A checkpoint consists of a pair containing the argument and the computed result of a DOTS thread. Whenever a thread has successfully

finished its execution a checkpoint is made. The checkpointed data is stored in a hash-table called *fork/join history table*. In the case of a restart of a DOTS thread, actually a kind of reply of communication and computation takes place: If the argument of the thread is present in the fork/join history table the result is taken from the table and immediately send back to the caller without carrying out any further computation or communication. This strategy is feasible whenever the function executed by a `dots_fork()` is stateless. Actually only those threads that had not finished their execution before the failure occurred must be restarted. In this case explicit checkpointing (see Section 4.3) can be used to further reduce the overhead of restarting. Implicit checkpointing is best suited for DOTS threads of a fine granularity.

As a side effect the implicit checkpointing mode can even be useful in failure free runs. Speedups can be obtained in certain applications (e.g. applications with a recursive structure where intermediate results can be reused) by always making lookups in the fork/join his-

tory table when `dots_fork` occurs before actually starting the execution of the DOTS thread.

4.2 Restrictions to Implicit Checkpointing

Implicitly checkpointing a function makes only sense if that function returns results that are valid regardless of the context and time of its execution. We call such a function *always-valid*. Certainly all deterministic functions belong to the class of always-valid functions. But also functions joined with join-any semantics may belong to that class if their results are always qualitatively equivalent.

If the return value of a function depends on the time or context where it is called the function is called *per-context-valid*. Typical contexts a (formally stateless) function may depend on are the system time, the processor it runs on or the contents of a file. Earlier results of per-context-valid functions may be invalid if the function is later called again (with the same parameters); be it a second call in a program or a call in a recovery run.

Our checkpointing extension to DOTS allows to prevent a function from being implicitly checkpointed. This is done by providing an additional optional parameter to `dots_fork`. This parameter may be used to switch implicit checkpointing (and thus result retrieval from a checkpoint) on or off. The default of this parameter enables implicit checkpointing. Therefore implicit checkpointing of a function can be switched of on a call by call basis.

4.3 Explicit Checkpointing

To use explicit checkpointing, the application programmer has three new API primitives at hand. They can be used to customize the granularity of the application’s checkpointing behaviour both in terms of space and time: The entities that are chosen to be checkpointed are objects and the checkpointing process can be triggered by an explicit API call.

The new API calls available to the programmer for controlling explicit checkpointing are:

- `dots_reg_cpobj`
- `dots_init_cpobj`
- `dots_do_cpobj`

The purpose of `dots_reg_cpobj` is to register an object to become a member of the set of objects to be checkpointed. As stated above, this is a very lean strategy. Only objects worth to be checkpointed are considered.

A special role in the checkpointing API plays the call `dots_init_cpobj`. Every previously registered object must be initialized through this call. A default object has to be passed as an argument to `dots_init_cpobj`. This default object is used as “checkpoint value” when no checkpoint is available for this object, i. e., in the non-recovery case. In the recovery case `dots_init_cpobj` initializes an object to its previously checkpointed value.

The function `dots_do_cp` actually carries out a checkpoint by saving all registered objects to stable storage.

The explicit checkpointing API imposes the DOTS application programmer to outline the algorithms and problems in a certain manner. In practice, the API has revealed a good adaptation to algorithms and problems.

5 Application

We consider as an application from the realm of Symbolic Computation a parallel satisfiability checker for boolean formulae (SAT). Important applications of the SAT checking problem include planning and scheduling [8], model checking for hardware verification [9], and checking formal assembly conditions for motor-cars [10]. The computational complexity of these problems can vary considerably, where the hardest practical problems often require thousands of hours of running time. Clearly, in this context dependable parallelization is highly desirable.

Actually a SAT checking algorithm performs an optimized search in an unbalanced binary search tree. The current state of the search

can be completely described as a so called *guiding path* [11], which basically is a path from the root of the search tree to the current node. Therefore in our implementation objects describing guiding paths are checkpointed. Guiding paths are also used to split the search space into disjoint regions which are assigned to different processors. Splitting cannot be done statically since the generated subproblems have considerably different and completely unpredictable run-times. Hence every time a processor has completed the search in its assigned region the search space of another processor must be split to generate new work. This dynamic approach leads to a highly irregular splitting behaviour. A major challenge of parallel SAT checking is to deal with this dynamic evolution of irregular parallelism.

For the realization of time measurements we took as example input a satisfiable boolean formula with 680 clauses containing 200 variables. All parallel program runs were carried out with both implicit and explicit checkpointing. For failure free experiments no significant overhead for implicit and explicit checkpointing could be observed.

Table 1 compares the sequential run-time with the parallel run-times on 2 nodes with 6 processors within the Parallel Sysplex environment described in section 2. It also shows the gained speedups when performing a complete restart at an early and a late stage of the computation. (In these cases the given total run-times include the time for restarting.)

6 Outlook

Our future research will include two topics. Based on the presented implementation of checkpointing we will add thread migration with preemptive resume scheduling to DOTS. Along with this point it would be interesting to closer integrate the system environment of the IBM Parallel Sysplex architecture into DOTS, e.g. using the Work Load Manager (WLM) for thread scheduling. Another topic of our future research is the integration of several Parallel

Sysplex clusters with a Java based infrastructure to form a meta-computing environment. This could also include porting DOTS to the recently released Linux for S/390 operating system.

7 Related Work

The Egida project [12] also addresses the issue of low-overhead fault-tolerant computing. It provides an object-oriented toolkit enabling transparent rollback-recovery for an MPI implementation. In contrast our approach takes into account the particular needs of high performance symbolic computing and is therefore tightly integrated into the distributed threads system DOTS.

References

- [1] Joao Carreira and Joao Silva. Dependable clustered computing. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1. Prentice Hall, 1999. 1
- [2] J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen. S/390 cluster technology: Parallel sysplex. *IBM Systems Journal*, 36(2), 1997. 2
- [3] Wolfgang Blochinger, Wolfgang Küchlin, and Andreas Weber. The distributed object-oriented threads system DOTS. In A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, editors, *Solving Irregularly Structured Problems in Parallel*, number 1457 in Lecture Notes in Computer Science, pages 206–217. Springer-Verlag, 1998. 3
- [4] Michael Meißner, Tobias Hüttner, Wolfgang Blochinger, and Andreas Weber. Parallel direct volume rendering on PC networks. In *The proceedings of the International Conference on Parallel and Distributed Processing Techniques and Ap-*

Table 1: Sequential vs. Parallel Run-Times with and without Restart

# nodes	# DOTS Threads	total run-time	restart at	speedup
1	-	482 s	-	1.0
2	6	112 s	-	4.3
2	6	134 s	36 s	3.6
2	6	114 s	109 s	4.2

- plications (PDPTA '98)*, Las Vegas, NV, U.S.A., July 1998. 3
- [5] Wolfgang Blochinger, Wolfgang Kuchlin, Christoph Ludwig, and Andreas Weber. An object-oriented platform for distributed high-performance symbolic computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999. 3
- [6] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An object oriented network programming toolkit for developing communication software., 1993. <http://www.cs.wustl.edu/~schmidt/ACE-papers.html>. 3
- [7] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990. 4
- [8] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1996. 5
- [9] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, 1999. 5
- [10] W. Kuchlin and C. Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1/2):145–163, February 2000. 5
- [11] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996. 5
- [12] S. Rao, L. Alvisi, and H.M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Proceedings of IEEE International Conference on Fault-Tolerant Computing (FTCS)*, pages 48–55, June 1999. 7