Experiences in building Cosy - an Operating System for Highly Parallel Computers

R. Butenuth[a], W. Burke[b], C. De Rose[b], S. Gilles[b] and R. Weber[b]

[a]Group for Operating Systems and Distributed Systems, Department of Mathematics and Computer Science, University of Paderborn, 33095 Paderborn, Germany,
butenuth@uni-paderborn.de

[b]Operating Systems Research Group, Computer Science Department, University of Karlsruhe, 76128 Karlsruhe, Germany,
{burke, derose, gilles, roweber}@informatik.uni-karlsruhe.de

COSY is a microkernel based operating system designed and optimized especially for parallel computers. The resident kernel on each node provides the basic functional abstractions of an operating system with low overhead, low latency and good scalability as demanded by the users of a parallel computer. Based on these abstractions all other functionality is provided by distributed services. As an example the management of parallel applications and a service for distributed dynamic partitioning of the parallel machine are described. All services provide good efficiency and scalability. MPI has also been implemented successfully on top of Cosy as well as a number of highly parallel applications. Looking at the results of the Cosy project we believe it is possible to provide the benefits of resident operating system services to a highly parallel machine with good efficiency as well as good scalability.

1. Introduction

Even though distributed memory machines are the prevailing class for all highly parallel machines built in the last years, there are still architectural differences in hardware and system software. Older systems are designed and used as a large coprocessor to a ‘front-end-computer’ (e.g. T3D, [11]), modern systems allow stand-alone operation (e.g. T3E, [12]). In contrast to the coprocessor solutions, stand-alone operation requires an operating system on the parallel computer.

Evolving a parallel operating system from a conventional microkernel based operating system (e.g. Mach, Windows-NT) seems to be a practible approach at first sight. But it is questionable if an operating system which was originally not designed with focus on parallel computing applications can provide adequate functional abstractions while still meeting the low latency, low overhead parallel application requirements. This causes many users to refuse operating system support and to rely on parallel runtime systems at the cost of e.g. lacking comfortable resource management facilities which can result in poor system and application performance.

With the beginning of the COSY (Concurrent Operating System) project in 1992 we ventured to build an operating system designed and optimized especially for parallel computers. In its intention to provide adequate functional abstractions even at the kernel level COSY has

similarities to other projects like e.g. the PEACE Operating System [13]. Apart from this, COSY has a strong focus on providing appropriate system services to ease the work of application programmers and users of parallel computers. It offers autonomous operation, dynamic multiprogramming (time- and space-sharing) and dedicated communication primitives. Additionally COSY provides support for automatic scheduling (partitioning and mapping) of parallel programs. All that is achieved with good scalability, i.e. even hundreds of processors can be supported efficiently. Currently, the kernel is implemented on two multicomputer platforms: the PowerPC-based Parsytec-PowerXplorer and the Transputer-based Parsytec MC, SC and GC family with up to 1024 computing nodes.

2. Kernel

The COSY kernel supports a small set of kernel objects that provide the abstractions needed by servers (which implement all other functionality of COSY) and application programs. It is the only part of the operating system that is guaranteed to be installed on every node of the parallel computer. Interaction across node boundaries is achieved by remote invocation of the kernel object methods, which makes all objects transparently accessible, local and remote.

The seven object types implemented by the COSY kernel (Figure 1) satisfy the basic requirements for system services and management issues. As an anchor to the objects on each node, there is the root object *kernel* with its predefined identification. The kernel can be used to look up node-specific attributes like the system time or the idle process id, but it is also a container object for memory *segments* and *address spaces*. A segment represents a chunk of physical memory which can be made accessible in an arbitrary number of address spaces by creating *mappings* into them.

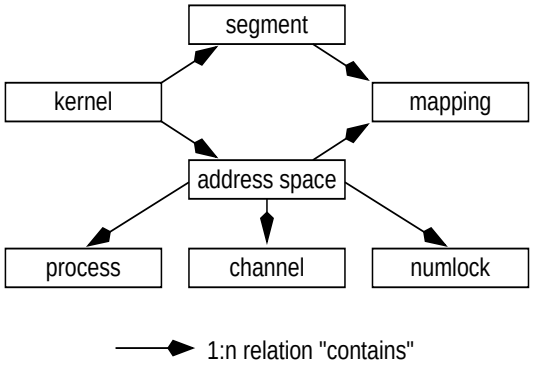

Figure 1. COSY kernel objects.

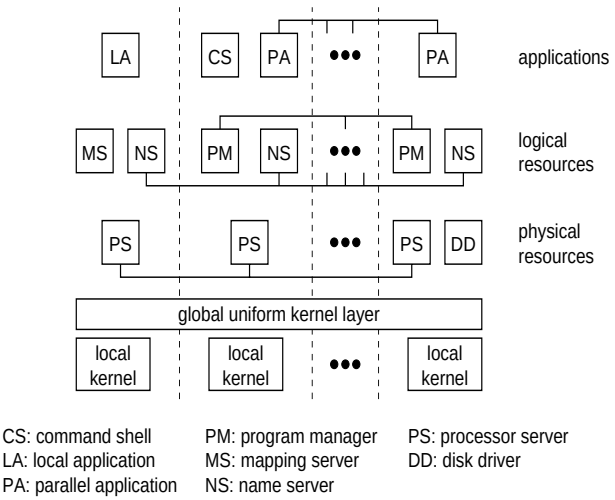

Figure 2. Architecture of COSY.

Address spaces serve two purposes. First of all, they represent memory as it is seen by *processes*. The separation of processes and address spaces facilitates a lightweight and efficient process implementation, including the possibility of building process groups. Multiple processes can be created in the same address space, sharing all their data. Consequently, address spaces are also container objects.

Apart from processes, an address space may contain *numlocks* and *channels*. Numlocks are generalized semaphores and can be used for process synchronization, e.g. to implement

atomic operations on shared data. Channels are the means of inter-process communication in COSY. Unlike in programming environments like PVM or MPI ([1], [9]), messages are not sent to processes, but to channels. This indirection allows to build dynamic client/server systems, where servers can be replicated or replaced unnoticed by the clients, since the request channel does not change. The fact that channels and numlocks are contained in an address space does not impose any restrictions on their use. They can be accessed across address space as well as node boundaries.

In COSY, channels provide *send* and *receive* operations in three kinds: *synchronous*, *asynchronous*, and *polling*. Another mode, *interrupting*, is currently under development. Senders and receivers select the kind of operation to be used by calling the appropriate communication method. In client/server systems, clients can send requests synchronously as well as asynchronously, while servers can receive them synchronously as well as polling or, in the future, by interrupt upon arrival. The implementation of channels in the kernel requires only a few internal methods to be implemented for any kind of send or receive operation. Adding new flavors of communication operations is therefore rather simple, and a new operation, e.g. an interrupting receive, can be used immediately with any complementary operation already implemented.

The design of the kernel turned out to be very flexible when the original implementation was ported from the Transputer-based Parsytec MC, SC, GC multicomputer family to the PowerPC-based Parsytec PowerXplorer architecture [15]. Without changes to the kernel interface, the MMU available on PowerPCs allows fine-grained control of the memory layout by setting the base addresses of mappings. This feature enables applications to map shared memory segments to the same base address in all address spaces. Memory protection is also supported on PowerPCs, where the read-only property can be set for segments as well as for mappings.

On a PowerPC 601 with 80 MHz, the latency for local communication is 19 µs, including two context switches. For communication to a neighbor node on a Parsytec PowerXplorer, the latency rises to 466 µs, due to the limited capabilities of this machine's communication hardware[1]. The corresponding values on a T805 Transputer-based System running with 33MHz, are a latency of 119 µs for local communication and 195 µs for next neighbour communication.

3. Servers

A resident kernel, as described in the last section, serves as the base for dynamic multiprogramming on every node of the parallel machine. Many different programs, separated from each other by protected environments, may be executed concurrently on the same node. This enables applications to share a processor in time, which has been shown to be advantageous for particular workloads [14]. Furthermore it permits the realization of some important classes of message passing applications like task farming and client-server systems, which was a main reason for incorporating dynamic process control in MPI-2 [10]. Besides these advantages, the resident kernel allows the implementation of a wide range of system services needed by many applications. This is accomplished by resident server processes existing independently from and concurrently with user applications on the same node. In this way the benefits of client-server computing can be used and the potential bottleneck of a single host computer, that pro-

1. The PowerXplorer uses a T805 Transputer (33 MHz) as communication Coprocessor.

vides the needed operating system functionality as a central management station to all the nodes of the parallel machine, can be avoided.

Because the services are used in a highly parallel environment they must be scalable, too. One way to support scalability is to provide a service by several server processes which are distributed over the parallel machine (Figure 2). The server processes of a particular service may be replicated on several nodes to increase performance. The level of replication is tunable according to the specific requirements of the given parallel system.

The services, which are offered to the user in form of simple interface functions, provide various kinds of operating system functionality. One service provides comfortable group communication that can be used to multicast data to several channels or to collect data which is combined to a single result (see [3]). Another service, implemented by a server process on every node, is responsible for monitoring the processor and link utilization. The monitoring information is used for a graphical monitoring tool as well as for the mapping server.

3.1 Program management

Creation and management of programs, i.e. processes and their environment (address space, code, data, stack and heap), is the task of another service, the program management. The management supports the joint creation of an arbitrary number of processes on any number of processors provided that the processes execute the same code. Distribution of the code, and of arguments that are identical for all programs, is performed by the multicast channels mentioned above. Using code sharing memory consumption and creation time is reduced for programs executing the same code on the same processor. A comparison with measurements performed in Parix [2] emphasizes the efficiency and the scalability of our solution (Figure 3)

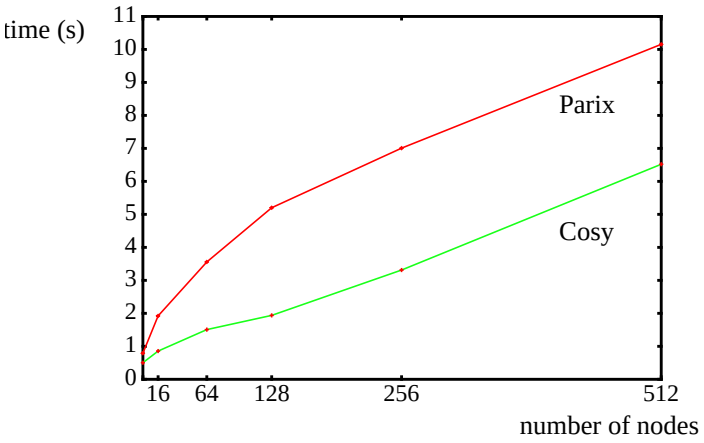

Figure 3. Time to load a program on a number of nodes in Cosy and Parix on Transputer.

The program management supports the dynamic creation of additional programs and of processes in an already existing environment at run time. After program termination, processes and their environment are removed automatically.

The parallelism in COSY applications has to be described explicitly. To make description and configuration of parallel applications, consisting of any number of processes and channels, easier for the user a configuration scheme for parallel programs was developed. The components needed for the execution of a parallel application are extracted from a configuration

description. Every component of the parallel program is installed on the parallel machine and removed after execution of the parallel application.

Mapping of parallel applications can be performed manually by the programmer or automatically by a mapping server. The interaction structure of a parallel application, extracted from the configuration description and described as weighted process interaction graph, is passed to this server. The server maintains all graphs in a queue and maps them as soon as enough resources are available. The mapping server may contain various mapping functions, which can be switched at run time.

3.2 Processor management

A processor server is responsible for automatically and dynamically shaping and allocating processor partitions of suitable size to the parallel applications, allowing a better utilization of the resources than a static partitioning of the machine. This processor management service [7] is implemented in a distributed way avoiding bottlenecks and resulting in better scalability than centralized management schemes (e.g. the machine-manager in [6]).

One of the implemented distributed algorithms is based on the principle of leaking water. From an origin point an amount of water leaks and flows in the directions where it does not encounter any resistance. An important factor is that the leaking water exhibits cohesion, which keeps the diameter of the resulting puddle as small as possible (Figure 4(a)). In the case of a distributed processor allocation, the number of processors to be allocated is represented by the amount of leaking water, the already allocated processors in the mesh by the resistance areas and the final area formed by the allocated processors corresponds to the resulting puddle.

Figure 4(b) exemplifies the execution of a 4-processor request. After an origin point is found (Figure 4(b1)) the possible flowing directions are determined and the remaining load is distributed (Figure 4(b2,3)). This procedure is repeated recursively until all load is allocated. Figure 4(b4) presents the processor mesh after the allocation operation. The essential feature of this algorithm is its free-form allocation strategy, i.e. partitions are no longer restricted to rectangles, but may have an arbitrary shape. This gives the managing component more flexibility to find a partition of suitable size and results in less fragmentation (internal and external). The size of the allocated partitions may also be changed during execution time, allowing a parallel application to dynamically adapt its partition to the current processor demand.

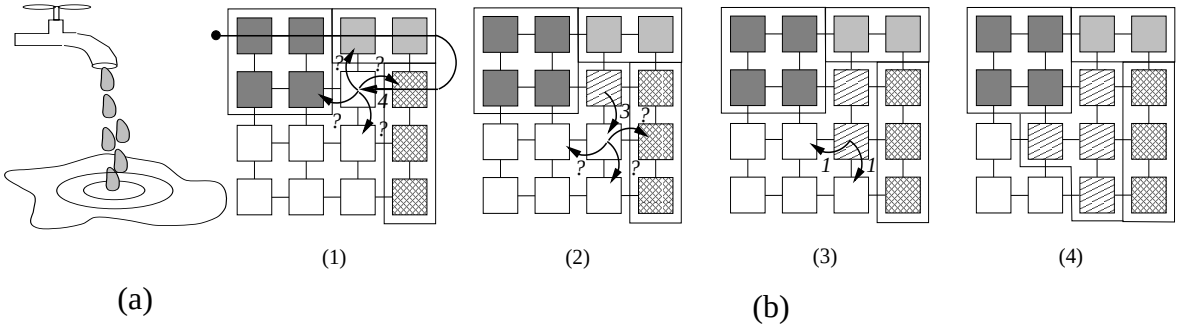

Figure 4. Leak algorithm.

Figure 5 presents the results obtained with the Leak partitioning algorithm in relation to the managed workload. The different workloads are obtained with a gradual increase in the requested mean partition size in a 8x8 mesh. It is possible to observe that the processor utilization remains high with a constant external fragmentation, in spite of a significant increase in the mean partition size. This shows the algorithm's capacity to manage free areas among allocated partitions (or to avoid the creation of such areas). This is achieved with the flexibility of the free-form partitions what on the other hand may result in partitions with a larger diameter than the regular ones (structure preserving [8]). The third curve (partition diameter) shows this effect. The 17% mean increase can be considered acceptable for the overall improvements.

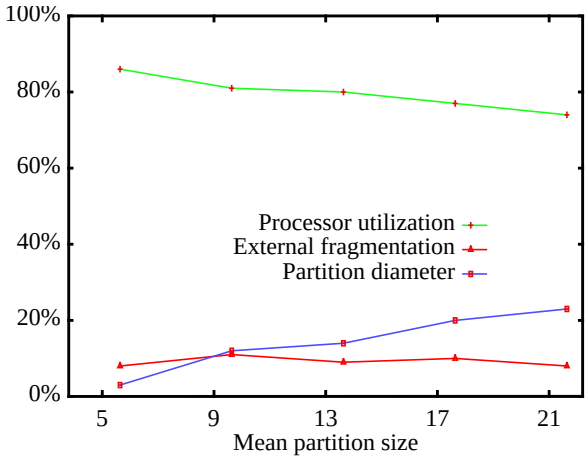

Figure 5. Behavior of a distributed management scheme with different workloads.

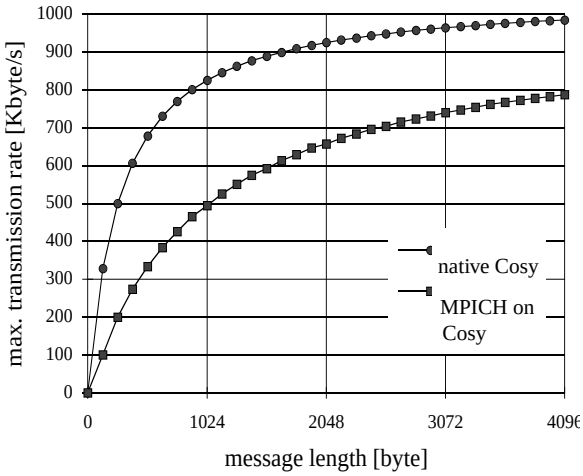

Figure 6. Transmission rates, native Cosy vs. MPICH on COSY on Transputer.

The obtained results with distributed management schemes showed that this approach permits a greater parallelization of the management operations, resulting in a better relation between the quality of the results and the time needed to solve the problem, allowing more complex policies to be used in a problem that must be solved at run-time.

4. MPI

Though it is possible to write application programs in standard-C extended by COSY system calls, a further step has been made by adapting MPICH [5] to COSY, a portable implementation of MPI. The COSY model of communication and the MPI counterpart are very different in the way they handle and deliver messages and the kind of protocols they use. Especially the nonblocking communication operations in MPI made it necessary to implement two special COSY processes per node: a send- and a receive-device process which handle queues with unexpected messages or pending communication requests. The MPI-application processes and the send-/receive-devices can access the queues which are located in a shared memory segment through a locking protocol.

The additional overhead of this straightforward implementation of MPICH on top of COSY compared to native COSY is a factor of 4 in message startup-time for nonlocal communication on a Transputer system. The resulting throughput graph is shown in Figure 6. A significant improvement is expected by using the interrupting receive operation in COSY (currently under development) which will make the send-/receive-device processes unnecessary. Then non-

blocking MPI-communication will be implemented by invoking an interrupt handling routine on the arrival of an unexpected message or a pending receive request. Furthermore, with respect to the single-sided communication primitives proposed by the MPI-2 standard draft, we believe the interrupting receive operation to be of key importance for an efficient implementation of MPI-2 on top of COSY. Additional features of MPI-2 such as dynamic process creation and deletion are already supported by the COSY program management service as already mentioned in section 3.1. To further improve overall communication performance certain low-level communication protocols (e.g. sliding window protocol) are currently under investigation.

5. Conclusions

From its start in 1992 COSY has been developed into a stable operating system. The portability has been shown by porting it to the PowerPC architecture without changing the programming interface. Various applications that have been implemented on top of COSY, e.g. a heat flow simulation and a molecular dynamics simulation [4] proved its efficiency and good scalability. Developers and applications have shown to benefit from the functional abstractions and services that COSY provides, e.g. multicast channels, trying and asynchronous communication primitives, dynamic process creation and payload monitoring. The COSY kernel and system services have been shown to scale up to at least 1024 processing nodes. All in all we believe that COSY has shown the benefits of an operating system specially designed for highly parallel computers.

References

1. O. A. McBryan: “An Overview of Message Passing Environments”, *Parallel Computing, vol. 20, pp. 417-444, 1994*.
2. A. Bachem, T. Meis, K. Nagel, M. Riemeyer, M. Wottawa: “Programming, Porting and Performance Tests on a 1024-processor Transputercluster”, in R. Grebe (ed.), *Transputer Applications and Systems ’93, pp. 1068–1075, Aachen, 1993*.
3. R. Butenuth, H.-U. Heiss: “Scalable Group Communication in Networks with Arbitrary Topology”, *2. GI/ITG-Workshop “Development and Management of Distributed Applications”, Dortmund, 9.-10. October 1995 (in German).*
4. R. Butenuth, H.-U. Heiss: “Highly Parallel Molecular Dynamics Simulation”, *will appear in High Performance Computing '97 (HPC), April 6-10, 1997, Atlanta, USA*
5. W. Cropp, E. Lusk: “MPICH ADI Implementation Reference Manual”, Argonne National Laboratory, *August 23, 1995.*
6. J. Gehring, F. Ramme: “Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations”, *Proceedings IEEE IPPS’96 Workshop on Job Scheduling Strategies for Parallel Processing, Hawaii, pp 41-54*.
7. H.-U Heiss: “Dynamic Partitioning of Large Scale Multicomputer Systems”, *Proceedings of the Conference on Massively Parallel Computing Systems (MPCS’94), Ischia, 2.-6. Mai, 1994*.
8. H.-U Heiss: “Processor Allocation in Parallel Machines” BI-Verlag Mannheim, Reihe Informatik Band 98, *1994 (in German).*

9. Message Passing Interface Forum: “MPI: A Message-Passing Interface Standard”, *The Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee, May 1994*.

10. Message Passing Interface Forum: “MPI-2: Extensions to the Message-Passing Interface”, *The Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee, May 1995.*

11. W. Oed: “The Cray Research Massively Parallel Processor System CRAY T3D”, *Technical Report, Cray Research, November 1993*.

12. S. L. Scott: “Synchronization and Communication in the T3E Multiprocessor”, *Operating Systems Review, vol. 30, no. 5, pp. 26-36, Cambridge, MA, December, 1996.*

13. W. Schröder-Preikschat: “The Logical Design of Parallel Operating Systems”, *Prentice Hall, Englewood Cliffs, NJ, 1994.*

14. S. Setia, M. S. Squillante, S. Tripathi: “Processor scheduling on multiprogrammed, distributed memory parallel systems”, *Proceedings of ACM SIGMETRICS Conference, pp. 158-170, 1993.*

15. R. Weber: “Porting COSY to PowerPC”, *diploma thesis, University of Karlsruhe, September 1996 (in German).*