COSY
An Operating System for Highly Parallel Computers

Roger Butenuth, University of Paderborn, butenuth@uni-paderborn.de
Wolfgang Burke, University of Karlsruhe, burke@ira.uka.de
Hans-Ulrich Heiß, University of Paderborn, heiss@uni-paderborn.de

This paper is dedicated to Prof. Horst Wettstein on the occasion of the 25th anniversary of his appointment.

1 Motivation

Even though distributed memory machines are the prevailing supercomputer architecture due to their superior scalability, programming for this class of parallel computers is still cumbersome and inefficient compared to sequential programming. Also resource utilization is often poor in comparison with sequential machines. This results from a lack of support by system software, i.e. compilers, operating systems, and programming environments. Especially operating systems are traditionally responsible to take some burden from the programmer by automatically managing resources and providing a logical, comfortable interface for resource usage.

To achieve efficient operation, we need an operating system that takes care of the peculiarities of multicomputer systems. Since the efficiency of interprocessor communication is a key factor for overall performance, the operating system should add as little overhead as possible to communication. To meet this goal, the common approach is to give up the concept of a resident operating system and to provide the necessary functionality by a run-time library linked to the parallel application program. This certainly has the desired effect but is accompanied by other disadvantages:

- Necessity of a front-end (host) computer

  The run time system basically consists of library functions for interprocess communication. The entire remainder of the operating system functionality is provided by a host computer for which the parallel machine can be considered as a complex coprocessor [5][11]. The main drawback of this approach is that the host becomes a bottleneck especially for extensive I/O operations.

- Single programming

  Usually, a parallel computer executes one parallel program after the other. The executing program has the entire machine at its disposal even if it cannot fully utilize it, due to a lack of parallelism or due to processes waiting for results of other processes [8]. Therefore, utilization in single programming mode is often low. If multiprogramming is available, then usually as static spatial partitioning, i.e. the machine is subdivided in several partitions each accommodating one parallel program. This approach reminds of the Sixties, when multiprogramming on uniprocessors was accomplished by providing fixed memory partitions. Meanwhile, dynamic and even virtual memory management is taken for granted. It should be clear that any static partitioning cannot serve well the varying and dynamic processor demands of the programs.

On the other hand, there is the approach to take microkernel-based distributed operating systems (e.g. OSF/1 [13], Chorus [14]). This is plausible since—in general—a multicomputer parallel machine is a special variant of a distributed system. It turned out, however, that those operating systems are often function-

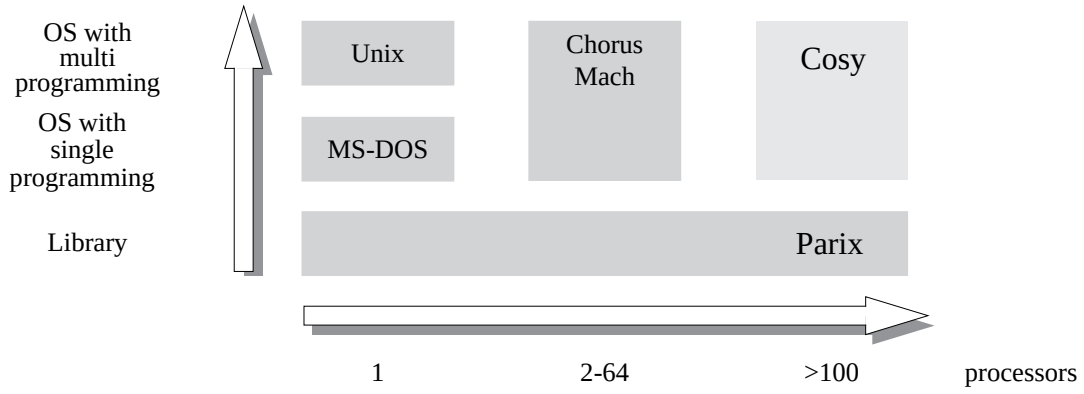

Fig. 1: Cosy in the spectrum of operating systems

ally overloaded with regard to their application area and thus too clumsy, time- and memory-consuming to support fine-granular applications appropriately [15].

With the Cosy project we ventured to build a real operating system that is up to the performance standards of run-time libraries but nevertheless provides the functionality of a classical resident operating system. It should allow autonomous operation, dynamic multiprogramming (time- and space-sharing) and dedicated communication primitives. To relieve the programmer, Cosy provides support for automatic scheduling (partitioning and mapping) of parallel programs. All that is being achieved with good scalability, i.e. even thousands of processors can be supported efficiently.

2 Architectural concepts of distributed/parallel operating systems

Operating systems—like other large program systems—are built in a modular way consisting of functionally decoupled components interacting with each other. This decomposition is usually done hierarchically resulting in a hierarchy of components becoming more and more elementary in a top-down point of view. As any other recursive process finally ends at the recursion base, there is a bottom layer in operating systems consisting of the elementary structures and operations. This layer is called the *kernel* and provides the programming infrastructure for the remainder of the operating system (higher layers) as well as for all other program systems. In practice, the kernel at least provides concepts for concurrent program execution and facilities for interaction, i.e. it provides processes (or tasks or threads) and communication and synchronization primitives. Using these elementary components, the rest of the operating system can be structured as concurrent processes providing services to others. So we basically have two parts of an operating system: The *kernel* and the *server area*.

A distributed system is characterized by program systems that spread across different computers. The operating system has to offer facilities for communication across processor boundaries. In order to achieve transparency, i.e. the property that the application program may not be aware of its distribution, the local interfaces have to be preserved.

Suppose two processes communicate via a local communication channel C and form a client-server relationship with A being the client and B being the server. If they are placed on different nodes, we need at both sides a substitute for the locally missing partner, i.e. a client stub A’ and a server stub B’ that communicate via the local ports to the respective partner and via a transport system (e.g. TCP/IP) with the buddy stub. The structure that is needed consisting of the two substitutes (stubs) and the transportation system in between can be called a *bridge*.

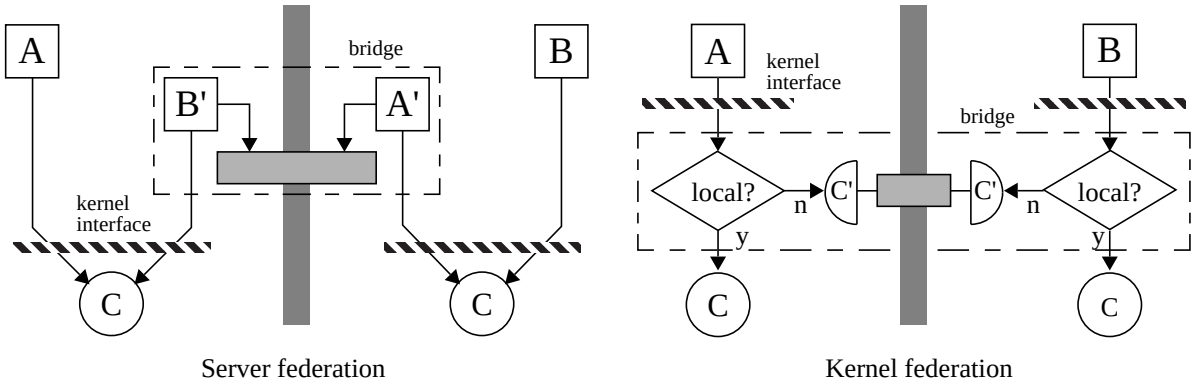

Fig. 2: Communication between processes, in the server federation (left) and in the kernel federation (right).

If the bridge is realized in the server area of the operating system we call it a *server federation*. In a server federation, the local kernels need not know that they are embedded in a distributed system. All boundary-crossing communication is handled by servers outside the kernel that can be reached via the local communication facilities. The operating systems Mach and Chorus are based on that paradigm.

The alternative is the *kernel federation* where the bridge is built at the kernel level. All local kernels are extended by an additional layer, the federation layer, where the substitutes are located. For transparent operation, each kernel call is funneled through a filter to check whether the kernel object to be accessed is local or remote. The federation of all local kernels along with the federation layer then forms a *global kernel*. This paradigm is realized in Amoeba [10].

The server federation can be easily realized, since it does not require modifications of a possibly already existing kernel. However, it implies a larger overhead at run-time because for each communication instance, several kernel calls are necessary. The kernel federation, on the other hand, is the more elegant solution since the distribution of the system is hidden below the kernel surface. It is the obvious solution if the functionality required for the data transport does not blow up the kernel too much. This is the case in parallel computers where the interconnection network can be considered reliable and functions dealing with communication errors can be omitted or at least be kept to a minimum. For these reasons the kernel federation was the architecture of choice for Cosy.

3 Kernel

3.1 Requirements

As mentioned above, the kernel should consist only of the elementary infrastructure, i.e. the necessary objects and functions required for realization of all operating system services. In addition, we have to consider the peculiarities of a parallel computer and possibly a large number of nodes.

Since only the kernel should be resident at the particular nodes it has to provide sufficient functionality to load and execute a program triggered from a remote node. This implies that kernel functions need to be called from remote. A kernel must be able to recognize calls for other kernels, to forward them and to deliver possible results.

Interaction between parallel processes across node boundaries has to be done via the communication primitives. The efficiency of these operations (latency, transmission rate) is a key issue for scalability. All algorithms and data structures employed have to be carefully designed. Internal operations of the operating system have to maximize inherent parallelism.

Another goal is a hardware-independent structure, since it should not only be suitable for currently available parallel computers, but also for future ones. Its importance is emphasized by the fact that hardware development proceeds at a faster pace than development of system software. In Cosy, portability is supported by a separation of the transport system which communicates with the kernel via a small and simple interface. Using a hardware with dedicated communication coprocessors would leave the kernel invariant. Also within the kernel, all processor-dependent parts are concentrated in a special component, the kernel entry layer [3].

Although the Cosy kernel is not strictly object oriented, many concepts of object orientation have been adopted. During the requirements analysis, a classification of responsibilities was performed resulting in a set of few orthogonal object classes (processes, address spaces, etc.) All objects have a systemwide unique identifier for addressing and localization. In the current version, the identifier is composed of the node number, a local index, and a random number which makes the identifier similar to a capability.

3.2 Kernel objects

For the design of the kernel and its objects, Bachmann-diagrams have been used. These are the usual tools for modeling data relations in network-databases. They are limited to 1:n-relations which, however, does not matter here, since m:n relations are anyway difficult to transform into effective data structures.

The entrance into the object network is the imaginary object *computer* that symbolizes the entire parallel machine but does not have a physical representation. From there we proceed to the nodes or kernels, respectively, since at each node there is exactly one kernel (Fig. 3). The kernel object exists as a real data structure which is generated at system initialization. It contains some attributes that can be read and (part of them) written. An example is the current clock value.

3.2.1 Processes

In many operating systems a distinction is made between *threads* and *processes*, or between feather-, light- or heavyweight processes or *tasks* which mainly comes from an insufficient separation of the concepts of *process* and *address space*. In Cosy, there is exactly one sort of a process: A Cosy-process is the representation of an activity, consisting of the current value of the instruction counter, the address of the stack and the contents of processor registers. Insofar, a Cosy process is extremely light. Since it can not work in a vacuum, it is assigned to an address space. The relation between process and address space is n:1 which means that several processes may share a common address space. (That does not exclude a situation of an address space without a process, i.e. a program residing in an address space without a process executing it.)

3.2.2 Address spaces

Address spaces comprise the entire space from the smallest to highest possible address. At creation time they are empty. Address ranges, on the other hand, represent pieces of physical memory, which in turn consist of a set of pages. At first glance a 1:n-relation between address space and address range seems to be sufficient, since the address space may contain ranges for code, data, and stack. However, shared ranges for shared code could not be realized. Therefore, we allow a m:n-relation, which requires an additional object connecting address spaces and address ranges. It contains the base address of the address range in the respective address space.

Since the memory management functions can be called across node boundaries, the memory needed for the execution of a program can be installed from a remote node. In addition, the initialization with program code and parameters has to be done. This requires an address space escape, which is supported by two

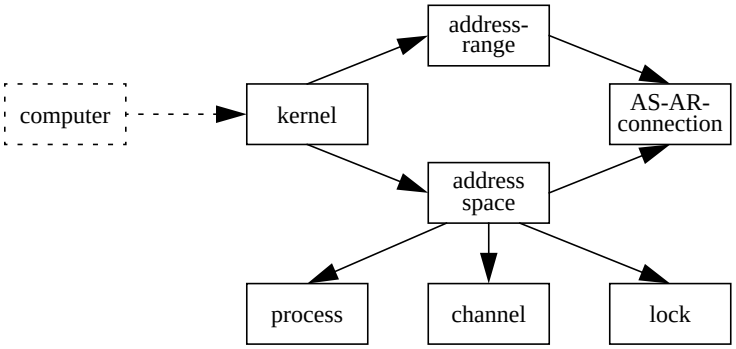

Fig. 3: Kernel objects and their relation

operations, *read* and *write*, applicable to any address space, local as well as remote. These functions are also used for the Cosy-debugger that runs in a different address space (possibly on a different node), but nevertheless reads and writes variables of the program under investigation.

Address spaces are usually connected to programs in a 1:1-relation. Therefore, there is no special object for a program in the kernel. Communication objects (channels) and locks are connected to address spaces for technical reasons. It does not mean that they can be used only from within this address space.

3.2.3 Communication objects

All communication in Cosy is based on so-called *channels*. A channel is a data object with operations *send* and *receive* that can exist independently from any process. A communication takes place when both sender and receiver have executed their respective operations. If the sender arrives first, it deposits the message itself (or the address of a message buffer) at the channel. If the receiver comes first, it deposits the address of the target buffer where the message should be copied to. At any time, the channel is in one of three states:

- empty
- one or more send requests (source buffers or messages) pending
- one or more receive requests (target buffers) pending

The communication partner that arrives later completes the communication by transferring the message into the target buffer. Regarding the coordination of sender and receiver, Cosy offers at both sides operations with different semantics that can be freely combined:

- **Synchronous:** The caller is blocked until the communication partner arrives at the channel. This is the usual semantic for receive operations, since the receiver often cannot proceed without the receipt of the message.
- **Asynchronous:** After deposition of the buffer address, the operation returns independent of the state of the communication partner. Asynchronous send is a means for latency hiding in parallel computation: As soon as intermediate results are available, they can be sent. If no receiver is waiting, the result is buffered in the channel and the computation can proceed without waiting.

- **Trying:** This is also a non-blocking operation, but communication is successful only if the communication partner has already called its respective operation. Otherwise, the operation has no effect. When used with receiving and placed in a loop, it can be regarded as “polling”
- **Interrupting:** The interrupting variant is also non-blocking, but has an instruction address as parameter. Upon the call of the partner’s operation, the program flow is interrupted and continues at the specified address, similar to a signal in Unix. This provides an elegant way to deal with rare or unexpected events. It has similarities with the concept of *active messages* [6][19].

The channels must not be regarded as 1:1-relations between exactly two processes. All processes holding the channel identifier can send to or receive from it (m:n-relation). Unlike ports in Mach—which by the way only provide m:1-relations—Cosy-channels can be used across node boundaries. This leads to flexible implementations of client-server relations: A server that gets its requests from a channel can be replicated without the clients being aware of it and without any additional intermediate server processes that distribute the requests to the replicated servers.

Under the given constraints—low latency, high throughput, many access variants—we need an implementation with reasonable overhead. The central idea of our realization is to separate the receive and the send part of the channel such that not any combination of variants must be considered at both sides. This reduces the overhead from quadratic (4*4) to linear (4+4) and, moreover, leads to an easily extendable structure, since adding another access method at one side does not require a change at the other side.

Send- and receive-operation—that are symmetrically realized except for the direction of the message flow—follow the same pattern: At the beginning, there is a switch whether or not the partner has already arrived at the channel. Depending on that, different branches for the different variants are taken. If the partner is not yet there, the caller deposits the respective information at the channel and returns immediately—for the trying-variant with a failure message. The information deposited at the channel tells the later arriving partner how to react. With blocking calls, for instance, the blocked partner needs to be deblocked. An elegant and straightforward way is to specify a method that the partner must execute upon call of its operation. This leads to a template that needs to be filled at three places with the respective code for the different variants:

1. A piece of code that is executed when the partner arrived first. Part of that code is the call of the method for collection or delivery as specified.
2. The complementary piece of code to be executed if the partner was not yet there. For the synchronous variant it includes the blocking of the caller, for the asynchronous one, the deposition of the message or a delivery address, respectively. For the trying-variant, this piece is void.
3. The delivery or collection method, which was specified in piece number 2 and must be executed by the partner from piece number 1. The trying-variant makes an exception: Since there is no information deposited, no method must be provided. This makes clear that the combination of trying-send and trying-receive will never lead to a communication.

To store the necessary information in the channel, we need memory space which can be obtained in different ways. The simplest idea is to use the memory management built in the kernel which is also responsible for kernel-internal memory requests. But due to its flexibility it is too time consuming to be called for each communication operation. Instead, upon creation, the channel is furnished with buffer space of variable size which can be efficiently managed as a cyclic buffer in a FIFO-fashion in constant time.

Unlike the usual partner communication where the message of exactly one sender is received by exactly one receiver, the *group communication* offers the possibility to send one message to a group of receivers in one operation (multicast), or, respectively, to receive a message originating from a group of

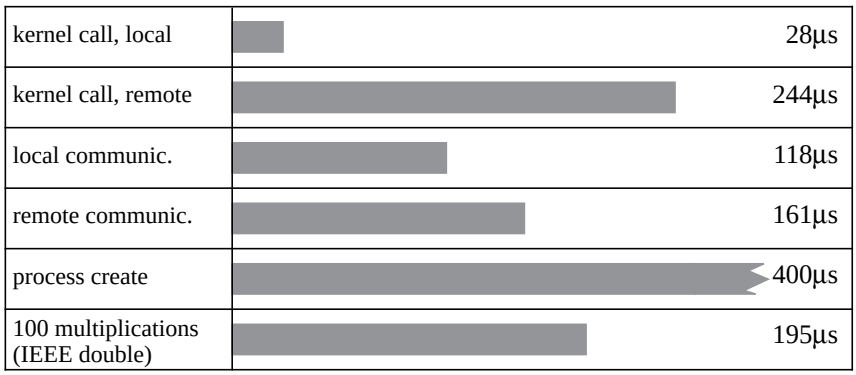

Fig. 4: Measured times for kernel operations

senders in one operation (combine). Especially for typical parallel programming paradigms like master-worker or SPMD, group communication has turned out to be effective and was included in the respective standards (MPI [9]). In a multicast operation, the message is delivered in identical copies to all receivers. In a combine operation, the messages of the senders are combined and form one single message that is delivered to the receiver. The way of combining can be chosen arbitrarily. For numerical messages, one may choose a sum-up or maximum operation. The implementation of the group communication is based on minimal trees in order to minimize network traffic [4].

3.3 Performance data

Many operating systems are characterized by the notion of a microkernel which turns out to be questionable since there is no agreement how to define “micro”. We prefer a functionality-based definition and require an operating system kernel to offer the minimal necessary infrastructure for a given use. The Cosy kernel satisfies that criterion. It contains some items that could have been placed outside the kernel but which for performance reasons in the given environment are better realized in the kernel. Despite these extensions, the kernel remained slim, consisting of 15500 lines of C-code[1], giving a total of 32 Kilobyte executable code, including library functions.

Since all data structures in the kernel are dynamic, the number of kernel objects is limited only by the available physical memory. This, however, makes it difficult to estimate the size of the data area. In a typical system configuration, the kernel data require 100 KB including buffer space for the channels and for the transport system. So the memory consumption of the kernel is below 150KB. Even with additional operating system components that usually reside at each node (loader, name server), the memory needed is below 300KB. This is pretty low compared to the 7-8 MB which are reported for the OSF/1-AD-kernel of the Intel Paragon [15].

More important for the performance of the system is the duration of the kernel calls. Fig. 4 shows the time consumption for some kernel calls. All measurement data were obtained from the current hardware platform of Cosy which is a transputer system with 30MHz-T805 processors. To assess the measured times with regard to the processor speed, the time for 100 floating point instructions is also given, that is in the same order of magnitude as the kernel calls. The first two measurements give the time that is needed for an empty kernel call, i.e. one that only enters the kernel specified as a parameter, but does not perform any

1. In addition, there are 250 lines assembler code for kernel entry and processor initialization.

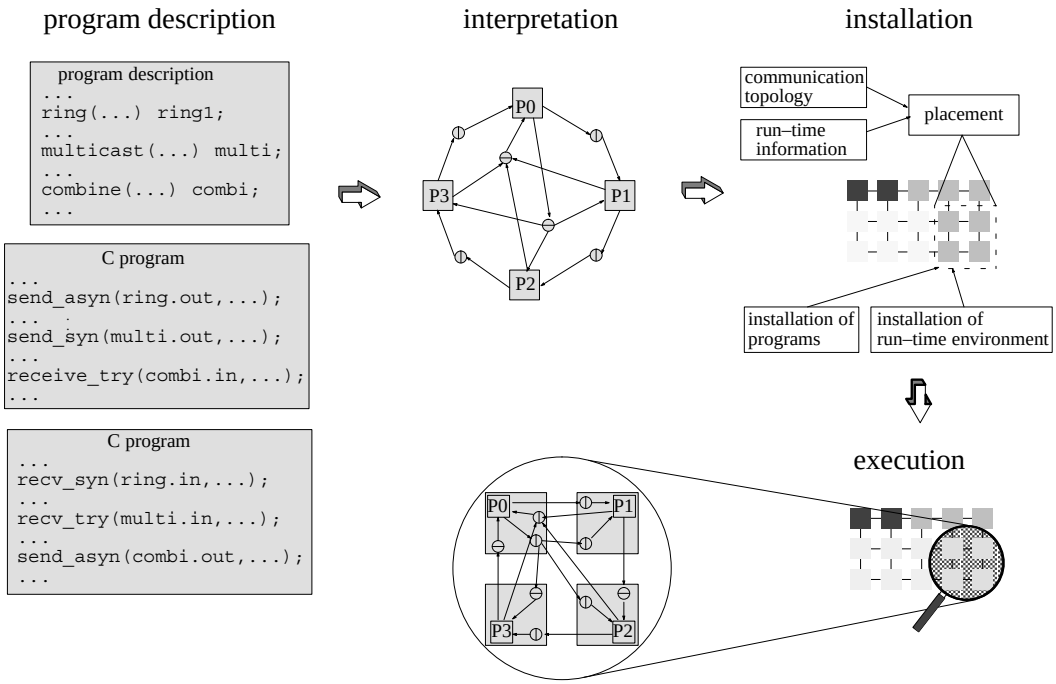

Fig. 5: Organization of program execution in Cosy

useful work. This value includes the overhead for the federation layer (global kernel) and for searching the object table.

The overhead for communication operations is difficult to characterize in one number, since there are so many variants with different complexity. For the measurement, we chose the standard combination of an asynchronous send and synchronous receive. Including the time for the necessary process switch, a send or receive costs 118 µs for a local call and 161 µs for a remote (neighbor node) call.

Fortunately, the most expensive kernel calls are required least frequently: the creation and deletion of objects which is due to the memory management involved.

4 Management of parallel programs

A parallel program in Cosy consists of processes and channels for interaction. The *program management* is the component in charge of the installation of these objects. The required information is provided by a program description which is built using a dedicated description language. For each process, its environment must be installed, which is an address space and some address ranges, the (sequential) program code, the data area and the stack. The particular processes may execute identical or different program codes (SPMD- or MPMD-model). Regular communication topologies (e.g. ring, mesh) are offered that can be parameterized. For irregular topologies, the graph must be specified explicitly. The names of the channels used in the description are symbolic and the same as in the program texts. For each installation request the following tasks have to be performed:

- Interpretation of the program description
- Placement of the objects of the parallel programs onto the physical resources of the machine
- Installation and configuration of the program

- Management of the parallel program.

Also during program execution, the programs can trigger the installation of new processes or channels via the program management component. The processes may be created in the same address space as the requesting process or in a different one.

With a spatial and temporal partitioning of the machine which is provided by Cosy, the placement depends on the current load situation. A decision has to be made, how many and which processors will be used for the program and where the particular processes are mapped to. Since the decision must be made at run-time, heuristics are used to solve the involved NP-hard problems (graph partitioning, load balancing, mapping) in acceptable time. The processes are hierarchically clustered to groups according to their communication intensities, their computational and memory demands. This cluster hierarchy is then used to distribute the processes across the system considering the current processor capacities with regard to free memory and utilization. Cosy does not provide dynamic process migration, since a questionable profit must be weighed against a high implementation effort and a high run-time overhead. Therefore, load distribution in Cosy is based on a reasonable initial placement of all objects at their creation time. Free capacities are utilized at the next request. How program execution is organized in Cosy is illustrated in Fig. 5.

If scalability is a design goal, then also installing and launching of parallel programs, which involves the creation of many objects, must not become a bottleneck. With regard to the creation of processes, especially the supply with the program text could cause a problem, since it requires time consuming I/O and communication operations. If the same program text is executed by a set of processes at different processors (SPMD-mode) the program text is delivered using the tree-based multicast operation of Cosy which makes the distribution very efficient. If more than one process shares a program code at the same node, Cosy provides code sharing, i.e. there is only one copy of the code in the memory. All in all, any independence of substages during program installation was exploited for parallelism. Therefore, the server processes of the program management service are replicated and internally organized as software-pipelines.

5 Links to the outside world

Parallel machines that use a run-time library only are tightly connected to the host computer and use its operating system for management and control. Since scalability was intended to be a key feature of Cosy, any potential bottleneck had to be avoided. A single host computer as a central management component is such a potential bottleneck. Cosy is therefore designed for autonomous operation.

Autonomy, however, does not mean autism, and links to the outside are required to make the parallel machine accessible from anywhere in the world. Therefore, a bridge is provided from the parallel system to a Unix workstation. The bridge has to cope with the physical and logical differences between communication inside and outside the parallel computer. Partly, the tasks are the same as for bridges between the parallel nodes. However, the semantic gap due to the heterogeneity of the parallel computer and LAN-connected Unix-workstations is too large to be bridged by a mechanism within the kernel. Therefore, the bridge is realized at the server level which neatly fits in with a well structured operating system where all the services are realized as processes. Instead of a server process that performs the service itself, there is a bridgehead that delegates the work to somewhere in the outside world. The necessary data transport to other computers can be done either via a network (e.g. Ethernet, FDDI) or via a direct link.

In the current version of Cosy, a workstation is *directly* connected. A heterogeneous bridge couples Cosy on the parallel computer with Unix on the workstation. At the node that is physically connected to the workstation, a client stub is running which communicates with the server stub at the workstation. Both processes are able to provide the bridging function simultaneously for several client-server-pairs (Fig. 6).

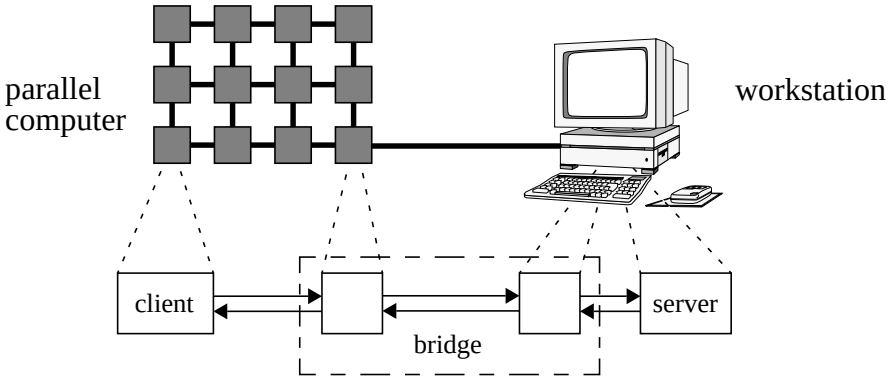

Fig. 6: Coupling parallel computer and workstation

A user who wants to work on the parallel computer calls the stub process at the Unix side which contacts its counterpart at the Cosy side that launches a command line interpreter. Input/output of the text-oriented interpreter is forwarded to a terminal emulator (xterm) at the user's site. For graphical I/O, the X-window system is used which also nicely fits in due to its spatial separation of program and window service. It works with any connection that provides a reliable bidirectional bytestream. Under Unix, usually internet protocols (TCP/IP) are used. With the integration of a service that realizes net access we not only have a X-connection but—as a side effect—access to the internet. Currently, the parallel machine uses the internet address of the workstation which is no longer required once the parallel machine gets its own network interface.

Another service provides file access with the same functions as Unix. When porting programs to Cosy there is no need to change the file operations. This interface will be the same for a file service currently under development that uses disks directly attached to some nodes of the parallel machine. The selection of a file service is done via a name service that gets a file name and returns a server address and a name local to that server. So it is easily possible to integrate other file services.

6 Conclusion

The development of Cosy required roughly 6man-years including some student theses. Although the product is not yet fully completed, we have already a bunch of applications running on Cosy, ranging from ray tracing over various types of equation solvers to tree search algorithms for optimization problems. For the performance of the applications, the communication primitives of Cosy turned out to be very useful and efficient (e.g. group communication, trying and interrupting receive). The operation of a 1024-node machine under Cosy did not show any performance problems and confirmed the good scalability. Current developments concentrate on the enhancement of the programming environment (debugging and monitoring) and the improvement of the autonomy (file system). A portation to a more modern hardware (PowerPC) and implementation of MPI will be completed soon.

All in all we believe that Cosy has shown the possibility to benefit from a resident, autonomous operating system for parallel computers without sacrificing speed or memory.

7 References

[1] V. Bala, et al.: “The IBM External User Interface For Scalable Parallel Systems”, *Parallel Computing, vol. 20, pp. 445-462, 1994*.

[2] O. A. McBryan: “An Overview of Message Passing Environments”, *Parallel Computing, vol. 20, pp. 417-444, 1994*.

[3] R. Butenuth: “The COSY-Kernel as an Example for Efficient Kernel Call Mechanisms on Transputers”, *Proceedings of the 6th Transputer/occam International Conference, Tokyo, June 1994*.

[4] 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)*.

[5] Cray: “CRAY T3D System Architecture Overview Manual”, *WWW, URL = http://www.cray.com/PUBLIC/product-info/mpp/T3D_Architecture_Over/T3D.overview.html*

[6] T. Eicken, D. Culler, S. Goldstein, K. Schauser: “Active messages: A mechanism for integrated communication and computation”, *Proc. Nineteenth Int. Symp. on Computer Architecture, ACM Press, New York, 1992*.

[7] H.-U. Heiss: “Processor Allocation in Parallel Computers”, *BI Wissenschaftsverlag, Mannheim, 1994 (in German)*.

[8] A. G. Hoekstra, P. M. A. Sloot, L. O. Hertzberger: “Comparing the Parix and PVM Parallel Programming Environments”, *ASCI 1995 Conference Proceedings*.

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

[10] S. J. Mullender et al.: “Amoeba, A Distributed Operating System for the 1990s”, *IEEE Computer, vol. 23, No. 5, May 1990.*

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

[12] P. Pierce: “The NX Message Passing Interface”, *Parallel Computing, vol. 20, pp. 463-480, 1994.*

[13] R. F. Rashid: “Threads of a New System”, *Unix Review, vol. 4, pp. 37-49, Aug. 1986.*

[14] M. Rozier et al.: “Overview of the CHORUS Distributed Operating Systems”, *Technical Report CS/TR-90-25.1, Chorus systèmes, France, 1991*.

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

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

[17] M. Snir, P. Hochschild, D. D. Frue, K. J. Gildea: “The Communication Software and Parallel Environment of the IBM SP2”, *IBM Systems Journal, vol. 34, no. 2, pp. 205-221, 1995.*

[18] C. B. Stunkel et al.: “The SP2 High Performance Switch”, *IBM Systems Journal, vol. 34, no. 2, pp. 185-204, 1995.*

[19] L. W. Tucker, Alan Mainwaring: “CMMD: Active Messages on the CM-5”, *Parallel Computing, vol. 20, pp. 481-496, 1994*.

[20] H. Wettstein: “System Architecture”, *Hanser, Munich, 1994 (in German).*