submitted for publication

Parallel Programming in PANDA

H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, H. Peine, R. Schwarz

University of Kaiserslautern, Department of Computer Science
PO Box 3049, 67653 Kaiserslautern, Germany

Abstract

Distributed systems are an alternative to shared-memory multiprocessors for the execution of parallel applications. PANDA is a runtime system which provides architectural support for efficient parallel and distributed programming. PANDA supplies means for fast user-level threads, and for a transparent and coordinated sharing of objects across a homogeneous network. The paper motivates the major architectural choices that guided our design. The problem of sharing data in a distributed environment is discussed, and the performance of appropriate mechanisms provided by the PANDA prototype implementation is assessed.

1. Introduction

Distributed systems are getting more and more attractive as a platform for parallel applications. System architectures containing a large number of workstations connected by a high-speed network provide enough computing power to be used for a number of problem domains usually dedicated to supercomputers. However, offering mechanisms which allow good exploitation of the existing hardware potential is an open problem. Generally, the development of convenient models for parallel and distributed programming is still an important challenge. Several issues related to this research area are addressed by the PANDA project which aims at providing an environment for object-oriented programming of a homogeneous distributed system [Assenmacher et al. 93a]. PANDA is based on the C++ programming language [Stroustrup 86] and offers a class library containing all those mechanisms that are relevant to deal with the issues of parallelism and distribution.

An effective support of parallelism is central to a distributed programming environment. The definition of parallel activities, called threads, is a way to exploit the available processors, and also a structuring concept. In a distributed environment, the time one thread is waiting because of remote communication can be used by other threads running on the same processor. Moreover, parallel programming can lead to better application performance even on a single processor [Rao and Kumar 93]. The costs of parallelism have important implications for the programming style; the efficiency of thread handling determines the grain of parallelism and is thus reflected by the program structure. Only if thread management and procedure call induce comparable overhead, both threads and procedures may be employed interchangeably according to application requirements. User-level thread implementation is a key technique to achieve efficient management of parallelism. This approach has several consequences for the operating system architecture which are reflected by the design of the PANDA system [Assenmacher et al. 93b]. The main principle is to provide the system functionality not critical with respect to protection and monopolization considerations within a runtime package in unprivileged user mode. The runtime package uses the primitive mechanisms of the underlying operating system kernel to implement the programming interface. According to the object-oriented system organization, the user interface consists of a hierarchy of classes. These classes can be used directly for building an application, or, alternatively, as a basis for more specialized programming systems, since the user is free to define new abstractions by customizing the PANDA classes.

A distributed application consists of several threads which cooperate by accessing common objects. Threads and objects may be located on different network sites. Ideally, an environment for parallel programming should hide the node boundaries from the application programmer. Above all, this requires support for location independent interaction of program components. In order to achieve location transparency, PANDA supports the abstraction of a global virtual address space as well as object and thread mobility. A global address space spanning a distributed system ensures that all pointers of an application denote unique memory locations on some node participating in the computation. When a thread accesses a remote object either the thread or the object are subject to automatic migration; the strategy depends on the user defined policy. Object mobility in PANDA is realized by means of a distributed shared memory mechanism.

Although full location transparency as provided by distributed shared memory is a desirable system property, its implementation may cause substantial overhead for an application. Following the shared memory paradigm, the unit of sharing depends on the hardware and does not reflect the application specific granularity of data. Granting access at the level of application objects can often lead to a better performance. PANDA offers a set of appropriate mechanisms for object-level mobility to complement distributed shared memory.

This paper discusses the fundamental abstractions of the PANDA environment for parallel and distributed programming. Section 2 gives an overview over the system architecture. In Section 3, the concepts for an efficient realization of virtual distributed shared memory and their implementation in PANDA are presented. Mechanisms for global synchronization are described, and the general problem of sharing in a distributed system is discussed. Section 4 investigates an alternative way of object sharing. Experimental results with the PANDA prototype system are briefly presented in Section 5. Related work is surveilled in Section 6, followed by concluding remarks in Section 7.

2. The PANDA system architecture

PANDA has been designed for homogenous multiprocessor workstations connected by a local area network. Its architecture consists of an operating system kernel and a runtime package (RTP). Application software is based on the RTP interface (Figure 1).

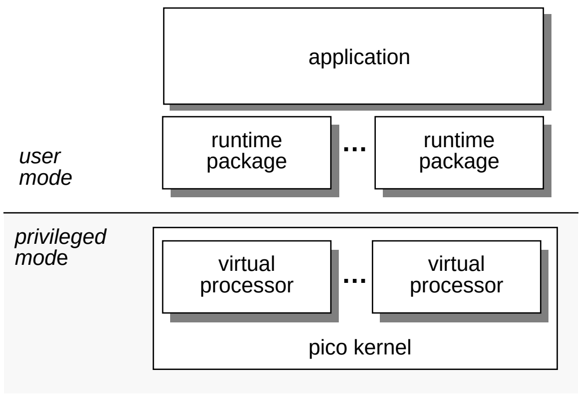

Fig. 1. The PANDA system architecture

Pico kernel

PANDA employs a small operating system kernel. One of the main design goals was to reduce the frequency of kernel calls. As a consequence, the kernel interface offers only those abstractions which are critical in respect to protection and monopolization considerations. All other functionality usually found in operating system kernels has been moved into user space. By minimizing the amount of kernel intervention, the overhead induced by the need to cross the trap boundary can be kept very low. The kernel interface is “skinny” and intended to be used only by system programmers. We chose the name *pico kernel* to stress the characteristics of our approach.

The hardware abstractions provided by the kernel are *protection domain* and *virtual processor*. A protection domain offers functionality to control access to address spaces on a per-page basis. A virtual processor is an abstraction of the processor and has the functionality to deal with interrupts, exceptions, and to communicate with other virtual processors on the same node. Every virtual processor is assigned a protection domain, but a single domain may be shared by several processors. The kernel applies preemptive scheduling to implement virtual processors; thus, different users and applications are protected against unfair processor monopolization. The time slices of virtual processors are rather coarse, fine-grained threads are realized in the runtime packages.

The kernel interacts with the runtime package in three different ways. First, the functions of the kernel interface are accessed via traps from user space into privileged mode. Second, the kernel informs the runtime package about exceptions and certain interrupts using an upcall mechanism (software interrupt). Third, read-only access from user space to dedicated kernel data allows efficient information flow without crossing the trap boundary. All kernel calls are non-blocking; threads will never block within the kernel, thus stopping the whole runtime package. Alternative approaches to prevent user-level threads from blocking the thread package may be found in [Anderson et al. 92, Draves et al. 91, Marsh et al. 91].

Only one virtual processor executes within each RTP. So there is no real parallelism within one runtime package even on a multiprocessor machine. This is an important design issue of our architecture. The runtime package offers very fast threads implying frequent scheduling activities. If several processors would simultaneously operate in a single runtime package, hot-spot system objects such as the scheduler or the memory administration would have to be protected against concurrent accesses. Besides the locking overhead, using common system objects on different processors may lead to frequent cache invalidation. Maintaining only a loose coupling between the processing units of a multiprocessor is a key to exploit its full computational power [Misra 91]. Our design implies that the potential of a multiprocessor architecture cannot be exploited by a single RTP. If an application wants to utilize multiple processors, it has to span several runtime packages. These RTPs may share their memory by operating within the same protection domain. In this way, application data is accessible by all RTPs while system objects are still separated. As a major drawback, load distribution has to be done at the application level. On the other hand, if an application spans runtime packages located on different network sides, explicit thread distribution is required anyway. To support the programmer, PANDA offers a number of simple schemes for load-balancing based on efficient thread migration mechanisms.

Runtime package

The RTP is realized as a hierarchy of C++ classes. They are divided into several class families reflecting the main entity types which constitute our distributed system model. As major abstractions, a family of thread classes, synchronization classes, and classes for distribution management are offered. The strength of PANDA's object-oriented approach is that a well-chosen collection of these classes is made available at the application level which may be tailored to the needs of the user by means of derivation.

In PANDA, a preemptive as well as a non-preemptive user-level scheduler may be supplied. The current runtime package schedules user threads in a non-preemptive way. We prefer the non-preemptive strategy because of its efficiency. By interrupting activities at well-defined points, only a small fraction of the thread context need to be saved. In comparison, preemptive scheduling implies storing the complete thread context and hence slows down context switches. An even more serious problem occurs if a thread holding a lock which is central to the computation is preempted. In this case, all succeeding threads are likely to be blocked immediately after their continuation — the computation is starved by the preempted lock owner. This is a common problem in applications consisting of closely cooperating threads. Furthermore, experience has shown that in most of our applications threads voluntarily release the processor due to blocking or termination before preemption would occur.

Fast thread management and context switches encourage programmers to structure their applications by using a high number of parallel activities. While the resulting performance overhead can be kept very low, memory consumption of threads becomes a serious problem. Above all, stack memory is a critical factor; stack requirements of threads must be calculated generously, especially if external library routines are called. In many cases, once activated a thread accomplishes its task and terminates without ever blocking. This observation is taken into account by the concept of *lazy thread creation*. In this approach, management data and stack of a thread are not allocated at its creation time, expecting that the control block and stack of the previously scheduled thread can be reused. Reusing is, of course, only possible when the predecessor thread terminates; if this thread blocks, a new control block and stack are allocated. By lazy thread creation better memory utilization can be achieved. Furthermore, this strategy improves the overall system performance, as shown in Table 1 (the measurements refer to a SunSPARC-10/Mod. 20).

As several threads may concurrently enter the same object, mechanisms for synchronization are required. It is the programmer's responsibility to ensure proper parallel object access. PANDA offers synchronization objects such as locks, semaphores, and signals. Being subject to its target language C++ and a conventional compiler, PANDA can hardly support more sophisticated synchronization paradigms that would require language extensions [Gehani and Roome 88, Nierstrasz and Papathomas 90, Saleh and Gautron 91]. Because of user level thread implementation, synchronization operations are extremely fast; their overhead is determined by the cost of procedure calls. By applying the inlining mechanism of C++ these costs can be further reduced. The efficiency of synchronization primitives is the decisive factor for the performance of parallel applications. Actually, cheap synchronization can be seen as the major reason for a user level thread implementation.

Thread creation, context switch, null execution, deletion • Reusing control block and stack • Allocating control block and stack	 13 µs 41 µs
Thread migration to a remote RTP	7 ms

Tab. 1. Performance of the PANDA prototype

System services

Most PANDA services are realized at user level. Services handling unique resources shared by several applications must not be included into any of the user RTPs. They have to operate in the context of a private runtime package. Consequently, calling such services requires interaction between local RTPs which entails kernel intervention. In particular, this applies to the remote communication handler responsible for network access. Therefore, sending or receiving messages requires one additional context switch at kernel level compared to a highly optimized communication service integrated into the kernel. The costs for kernel level communication, however, can be kept low as shown by the L3 system [Liedtke 93]. By separating services from the kernel, better modularity is achieved, and thus extending and maintaining the system is facilitated. Moreover, service implementation can profit from PANDA functionality.

3. Sharing data

The aim of PANDA is to support distributed applications written in C++. Such programs typically follow the spirit of C in that they rely heavily on pointer references. This yields a rather "impure" object model due to the lack of encapsulation, but restricting the use of pointers would be a major deviation from what is regarded as a C++ standard. Retaining the expressiveness and efficiency of pointers even in a distributed setting was a challenge for the design of our runtime environment.

Global virtual address space

The most natural way to cope with an unrestricted use of pointers is to provide the application with a global virtual address space, spanning all nodes of the distributed system. In a homogeneous environment, a global virtual address space guarantees that pointers may be passed freely across node boundaries, and have the same "meaning" everywhere in the system.

Ensuring the uniqueness of all pointers does, of course, not mean that arbitrary memory addresses can actually be accessed by a local thread of control. Instead, PANDA statically divides the global virtual address space into several partitions. Whether an object is currently accessible or not depends on the memory partition in which it resides. On the one hand, there are *local partitions* whose memory locations are protected so that they can only be addressed by local threads. On the other hand, PANDA offers *shared partitions* which denote identical address ranges mapped on all nodes. Whenever a shared partition is accessed, the runtime package ensures that an up-to-date memory image will be presented to the application. At the language level, there is no difference between an access to a local object, and an object which is remote but resides in a shared partition. Therefore, ordinary, sequential application code may be executed in a distributed environment essentially without any modification. This location transparency is highly desirable with respect to the portability of existing programs, and also for debugging a distributed application by running it on a single node.

Distributed virtual shared memory

In PANDA, protection and sharing of memory partitions is under hardware control by default. The memory management unit (MMU) raises a page fault interrupt whenever the application tries to access a virtual page which is not available at the local node. If an address is referenced which refers to a shared segment, the page fault is passed to a user-level page fault handler. This handler triggers a distributed shared memory (DSM) protocol which ensures that a valid copy of the corresponding memory page is provided, that appropriate access rights are granted to the faulted thread, and that the thread is reactivated from the asynchronous interruption as soon as the page has been made available.

Page-based shared memory employing hardware surveillance has several advantages over approaches which share data at an object level under software control (see, e.g., the *proxy object* approach [Shapiro 86]):

- If the data is present at the local node, any computational overhead is completely avoided. (In a well-balanced distributed program, this should be by far the most common case.)
- If the missing object is very large, it is fetched in a lazy fashion — page after page —, and only pages which are actually referenced are fetched at all.
- If a collection of logically related, small objects is clustered appropriately, then by fetching a whole page the system implicitly supports prefetching without user intervention.
- A general shared memory is "foolproof" in that it does not restrict object access to dedicated method invocations but permits arbitrary pointer dereferencing.

On the negative side, DSM at page granularity has also some potential drawbacks. The most disturbing pitfall is *false sharing*, i.e. a situation where two or more unrelated objects which happen to reside on the same memory page are requested concurrently by different nodes. In this case, the page may continuously be paged from one node to the other resulting in page thrashing. Another major concern is the *granularity* of data exchange. If an application deals mainly with small, isolated objects, then transferring the complete page just in order to obtain a few bytes is a waste of bandwidth.

To overcome these difficulties, a number of countermeasures are taken by PANDA's DSM protocol. First of all, an attempt is made to reduce the *frequency of page faults*, thus reducing the page traffic induced by it. The protocol follows an "exclusive-write-concurrent-read" policy by providing read copies for pages which are read but not written. Only if a "write access miss" is encountered, all read copies of the corresponding page are invalidated, and the faulting node is provided with an exclusive (writable) copy of the page. That is, following the classification in [Nitzberg and Lo 91], PANDA's protocol is of the "write-invalidate" type.

Providing concurrently accessible read copies may substantially reduce the page traffic in many applications, but fails for pages which change frequently. In such cases, however, it is commonly observed that only a small fraction of the page's contents are actually invalidated between successive page transfers. To avoid a transfer of redundant information, the DSM protocol tries to transmit *page differences* rather than full pages whenever possible. Only those parts of the page which have been altered are exchanged. To this end, each page copy is assigned a version number which is incremented whenever the page is transferred to a new destination in 'write' mode. If a node which has an invalid page copy of version n applies for the most recent version of the page, and the up-to-date version number of the page is n+1, then the current owner of the page just computes the differences between the two versions, and transmits only those bytes which have changed — the so-called page difference. We observed that the size of a page difference is often less than 5% of the size of a full page. Furthermore, computing a page difference is relatively cheap (on a SunSPARC-10/Mod. 20, it requires less than 0.28 ms in the worst case, and typically less than about 0.18 ms for a 4 KByte page), and differences may often be re-used several times. Unfortunately, differencing is only feasible if a shadow version is kept for every page which is writable. In the worst case, i.e. if *all* pages are *continuously* in 'write' mode, this would require an overhead of 100% in memory consumption; however, this is probably not the average case, and differencing could easily be restricted to the "hottest" pages under these circumstances.

In contrast to other DSM protocols which apply page differencing [Keleher et al. 93, Fleisch et al. 93], we currently make no attempt to compute differences between version gaps greater than 1. Provisional experiments seem to indicate that the simple approach suffices in most cases. If, for example, a page is mostly read and rarely written, then the probability for a version difference greater than 1 is small (most writers already have the current version in 'read' mode, and updating the read copies after a 'write' is accomplished before the next write occurs). Furthermore, if a page is shared between just two different nodes, their versions will never differ by more than 1. Thus, it is not clear whether the increased potential of generalized page differencing could ever outweigh the additional overhead in memory consumption and computational effort. However, further evidence is needed to decide the subject matter.

Page differencing does not prevent thrashing if false sharing occurs. To a certain degree the probability of false sharing is reduced by the provision of read copies, but as soon as at least one of the competing nodes requires write access, thrashing may occur. Therefore, the only way to avoid the danger of false sharing is to put unrelated data in separate memory pages. PANDA provides mechanisms which allow to declare an arbitrary number of logical *DSM clusters* (within the same shared partition), and these clusters can be mapped to disjoint sets of virtual memory pages. Thus, page faults related to objects which reside in different DSM clusters will never lead to conflicts. DSM clusters may also serve to support prefetching. Note that if an object is required at some node, then there is a high probability that this node will need to access the object's "relatives" in the near future, too. If related objects share the same cluster (and if the objects are small in size as it is typical for most C++ classes) then each page transfer fetches several objects at a time, and there are good chances that the related objects are already available when they are eventually referenced, thus avoiding a page miss. Prefetching strategies are particularly attractive for the emerging high-speed communication technology, where "wire time" is more and more dominated by communication latency.

By employing the above strategies — concurrent read access, page differencing, and clustering — our prototype implementation of the PANDA DSM achieves reasonable performance. Table 2 summarizes some provisional measurements that were carried out on three SunSPARC-10/ Mod. 20 workstations connected by a 10 MBit/s ETHERNET. The times were obtained by forcing a series of several hundred successive page misses, and by measuring the overall time between the first page miss and the last page access. The values labelled 'PAGE' denote the duration of a full page transfer (4 KByte), whereas the 'DIFF' values denote the corresponding time spent for the page update if only a small page difference (12 bytes) needs to be exchanged. Due to the rather coarse-grained timer that was used to obtain the measurements, the figures stated below are accurate within an interval of +/-0.27 ms. The results reflect end-to-end sustained rather than peak performance: Note that the timing includes all kernel traps, user-level context switching, communication and protocol overhead, page copying and differencing, as well as additional delays caused by message buffer allocation and deallocation, dynamic heap defragmentation, and also all strange effects which had their origin in the UNIX timesharing system on which our pico kernel emulation was executed. With these reservations in mind, it should be clear that the figures in Table 2 are conservative.

The huge difference between the times required to transfer full pages (PAGE) and page differences (DIFF) are a clear indication that a major performance bottleneck of our protocol is the limited bandwidth of the ETHERNET. With such a communication hardware, the ultimate lower bound for a 4 KByte page transfer is about 3.3 ms which is rather close to the estimated peak performance of a DIFF transfer.

Time to obtain valid page copy after page miss (ms)	page at home node	page not at home node
page miss at home node	—	PAGE 10.00 DIFF 5.87
page miss not at home node	PAGE 9.87 DIFF 5.67	PAGE 11.73 DIFF 6.93

Tab. 2. Performance of the PANDA DSM

Global synchronization

Supplying a local node with shared data is one problem — *coordinating* concurrent access is another. Note that shared objects can hardly ever be accessed without locking them first. Therefore, efficient means of global synchronization are as important as fast object transfer.

A straightforward approach to realize global locks would be to simply take ordinary local lock objects, and to put them into a shared memory partition. The DSM protocol would ensure that the lock is globally accessible. Although logically correct, such a solution yields unsatisfactory performance. Note that acquiring a lock is inevitably an update operation: Even readers have to apply at least *some* kind of change to the lock object in order to reflect their presence. Therefore, putting the lock into DSM memory would force a write miss even if only a read operation was intended by the lock owner. Consequently, locks have to be treated separately from DSM.

Another concern is that locks should offer *concurrent* access as long as only a 'read' lock is required. After all, it would be unreasonable to provide concurrent readers with multiple read copies of the shared data, but force them to compete for a unique lock instance in a mutually exclusive fashion. Thus, it is called for some kind of distributed read-write lock which supports concurrent 'read' locking, but exclusive 'write' locking operations. In PANDA, the class `RW_Lock` satisfies these requirements.

In essence, a `RW_Lock` object is a distributed data structure which is controlled by a dedicated RW_Lock protocol. Such a lock object supports the operations `readlock()`, `writelock()`, and `unlock()` at its interface, each with the usual semantics. The locking protocol has been designed in a such a way that a `readlock()` call typically entails no remote communication, but can be granted on a purely local basis by contacting the lock's local representative. Only if a `writelock()` operation is issued, then global coordination is (typically) unavoidable, and communication with other nodes is required. Consequently, the lock protocol ensures that the DSM protocol's effort to provide read copies is not spoiled by a bottleneck during lock acquisition. If objects are accessed without being changed, then after a short time all nodes have local read copies of the objects, and a local representative for all the corresponding locks — there is no further need for remote communication. DSM protocol and lock protocol complement one another in preserving the locality of computations.

Typically, the designer of a global synchronization mechanism is confronted with conflicting goals:

- On the one hand, one would like to prefetch the object in question on lock acquisition, so that accessing the object is possible as soon as the lock has been granted.
- On the other hand, a premature prefetch should be avoided, or else the current owner is deprived of its object while it is still locked (and is thus of no use to the impatient competitor).

Unfortunately, there is no clear escape from this dilemma. If it is assumed that critical code sections are typically short, then most lock acquisitions will immediately succeed. This, in turn, means that prefetching will not interfere with other lock clients (as they are currently not holding the lock) and should be the method of choice. If, on the other hand, most objects are found to be locked at access time, then prefetching should rather be avoided.

PANDA leaves the decision about a suitable prefetching policy to the application programmer. To this end, the class `RW_Lock` was constructed according to the so-called "envelope-letter" idiom [Coplien 92]. More specifically, an instance of this class is essentially a constant (i.e., an immutable "envelope") containing a reference to a mutable `Lock_Body` (the "letter") which keeps track of incoming lock and unlock requests. The `Lock_Body` is allocated in a local partition, regardless of where the enclosing `RW_Lock` object may reside. Therefore, `RW_Lock`s may be put into DSM segments and lock requests may be accepted without spoiling the DSM read copies. The application programmer can now decide where to put the lock:

- If the lock is *contained* in the shared object that it protects, then accessing the lock fetches the corresponding page which probably contains the whole object, and the object is locally available even before the lock has been granted.
- If the lock is kept *separate* from the object that it protects, then lock acquisition does not interfere with object acquisition, but the object is not prefetched, either.

Based on these two alternatives, an application is free to derive more advanced lock types from the fundamental class `RW_Lock`, and to find its own compromise between the two extremes. One could, for example, imagine a lock which knows the collection of objects that it is assumed to protect, and which prefetches a well-chosen subset whenever the lock is requested — simply by touching the corresponding object's pointers. PANDA does currently not offer such classes at its interface, but that kind of extensions should be relatively straightforward.

In PANDA, there is one remarkable exception from the rule that access to DSM objects typically requires synchronization. This is when a mobile thread puts its *private* objects into a DSM segment — not to share them with other threads, but just to make them available on all nodes. This strategy enables the system to migrate the thread in a "lazy", demand-driven fashion, by restricting the initial thread transfer to the essential data (scheduling information and stack). All other objects owned by the thread are left where they currently are. Should they actually be needed, the DSM protocol will fetch them; if not, superfluous transmission has been successfully avoided. When used as a device for demand-driven thread migration, the DSM subsystem is particularly efficient, because no local or even global synchronization is required: The data in question is known to be under exclusive control of the migratable thread.

In general, the need for global synchronization is *one* important potential problem for sharing models based on DSM. If we compare DSM with other paradigms of distributed cooperation such as, for example, a client-server architecture, it has to be admitted that such models have a big advantage in this respect. If the only way to gain access to a shared object is by calling the server which (exclusively) controls the object, then synchronization can always be restricted to the local node of the respective server, without any need for global activities. Nevertheless, we are convinced that the need for global synchronization in the DSM paradigm is outweighed by its transparency and flexibility. These features are particularly valuable for the design of system services such as, for example, thread migration or load balancing.

4. Bridging the object granularity gap

The advantage of a DSM system is the location transparency it provides to the application. It may, however, suffer from false sharing as described earlier. One way to circumvent this problem is to separate logically unrelated objects by placing them on different memory pages. Depending on the object size this may result in poor memory utilization.

As an alternative, one might consider to support sharing at the level of logical objects instead of pages. This avoids the granularity mismatch between application objects on the one hand, and the unit of sharing on the other. As the majority of current hardware architectures does not provide access control on a per-object level, such an approach requires a software-based solution. To this end, PANDA offers mechanisms to co-locate objects; sharing the same location avoids the crossing of node boundaries. More specifically, a programmer may explicitly "tie" and "untie" objects. Tied objects are kept co-located so that their further cooperation will never face an object miss. In case where an object relies on local hardware devices, it is also possible to tie the object to a specific node. This is called "fixing".

Tying may be seen as a way to *dynamically* specify clusters of related objects. This contrasts to DSM where only static clustering is feasible. An application may adapt to changing access patterns by dynamically specifying clusters which indicate to the system the set of objects that should be treated as a unit of sharing and mobility. Note that tying only specifies co-location, but does not prescribe the node where the cluster should be placed. The system is free to take an appropriate decision, based on runtime information.

The strength of DSM is that the monitoring of object misses is for free as long as no remote access occurs. In case of locality, redundant tying and untying calls cause a constant overhead (see Table 3). With an increasing rate of page faults, however, it is more attractive to rely on tying since explicit co-location calls avoid the additional overhead induced by interrupt handling.

The necessary calls for tying and untying objects may either be placed manually or by the PANDA preprocessor at the beginning and the end of each public object method. The former method is less prone to error but the latter may provide higher performance by rendering redundant tying operations superfluous. Provided that object access is solely granted via methods, PANDA offers a completely transparent mechanism for per-object level sharing. This, however, implies usage of the PANDA preprocessor. As an alternative, if an object's data is directly accessed or if preprocessing is to be omitted, full transparency is lost by explicitly maintaining the object relation which guarantees co-location and thus accessibility.

PANDA's tying mechanism and DSM complement each other. The latter may serve as a transparent default, while the former can be applied to improve application performance in cases where DSM is hampered by false sharing. Even a hybrid approach combining both strategies might be advantageous to bridge the gap between fine-grained and coarse-grained sharing. Such fine-tuning requires to statically partition the application objects in those subject to DSM and the others explicitly handled by tying calls.

Local tie + untie call	43 µs
Tie call + object transfer + untie call	2.5 ms
Local object fixing + unfixing	5.1 µs

Tab. 3. Performance of the co-location primitives

5. Experimental results

A number of parallel applications have been implemented in the PANDA environment. One of the most complex programs realized so far is a distributed traffic simulation, comprising 27.000 lines of code. The inherent parallelism of this problem domain made it an attractive choice. In this application, the behavior of individual vehicles is simulated and graphically animated in real time. Vehicles are the active entities in the simulation environment, each represented by an individual thread of control. Components of the street network are modelled by passive objects. This choice of design seems to be the most natural one.

A detailed simulation of large networks demands higher performance than one machine can deliver. The simulation task can be roughly partitioned by dividing the city street network into several "quarters"; this suggests a distribution on several machines, requiring flexible support from the runtime system. The quarters are, however, connected at the borders, making the task not as cleanly partitionable as, e.g. a matrix multiplication. To facilitate distribution, partitioning of the application into local components should not cause major changes to the program's structure. This goal is achieved by employing thread migration for vehicles changing quarters, DSM for remote data access, and global synchronization primitives for the coordination of parallelism.

Table 4 shows the DSM characteristics for a distributed execution of the simulation program. The figures emphasize the benefits of page differencing. The average difference size was only 1.4% of a full page, and 77% of all DSM page transfers were achieved by exchanging only a page difference. Furthermore, most page differences computed were eventually utilized for some page transfer. In fact, we observed the same general tendency in a number of other parallel applications.

Actually, DSM performance is not a dominating factor in the traffic simulation. The application's run time is mainly determined by remote synchronization costs. The simulation shares data at the level of small objects, each protected by a lock. The overhead for locking is moderate in the *local* case, since our synchronization primitives are realized in user space. In the *distributed* environment, retaining this structure and simply relying on location transparency leads, however, to severe performance penalties. The costs of global synchronization require locking at a coarse grain. Consequently, our application cannot fully exploit the computing power of the available nodes.

Since the application runs in real time, performance is not measured in terms of run time, but in terms of the tractable problem size. By moving from a centralized system to a configuration consisting of *two* nodes, we observed almost no performance gains in our experiments; the basic costs of global synchronization outweighed the additional computing capacity. However, further doubling the number of nodes increased performance by a factor of 1.7. Seemingly, global synchronization entails a significant basic amount of overhead that — in our case — does not proportionally grow with the number of nodes. In conclusion, our experimental results stress the relative importance of synchronization primitives compared to sharing mechanisms.

avg. size of page difference (% of a full page)	1.4%
avg. difference utilization (% of generated diffs)	95%
difference transfers (% of all DSM transfers)	77%

Tab. 4. DSM characteristics of the traffic simulation

6. Related work

Fast user-level thread implementations are offered by SCHEDULER ACTIVATIONS [Anderson et al. 92] and PSYCHE [Scott et al. 90]. Both systems support fine-grained parallelism on shared-memory multiprocessors, but do not address distribution. PSYCHE and SCHEDULER ACTIVATIONS supply efficient user-level threads by dedicated kernel architectures. In contrast to PANDA, their design allows several real processors to concurrently execute within the same runtime environment.

A number of DSM implementations have been described in the literature, see, for example, [Nitzberg and Lo 91] for an overview. Our protocol strongly resembles the approach taken in MIRAGE+ [Fleisch et al. 93]. The MIRAGE+ protocol guarantees sequential consistency, and in its most recent version page differencing is currently being integrated. To reduce the danger of page thrashing, DSM pages may be frozen on arrival for a certain Δ time window.

MUNIN [Carter et al. 91] was one of the first software DSM memory systems which used release consistency as a model of memory coherence. Release consistency enables the system to merge page updates in order to propagate them in a single message on the next release. MUNIN offers multiple consistency protocols. An appropriate strategy is chosen according to sharing annotations provided by the programmer. Possible annotations are, for example, 'read-only', 'write-shared', 'producer-consumer', 'migratory', or 'conventional'. Multiple concurrent writers may access the same page because it can safely be assumed that concurrent updates cause only false sharing without actual interference. To exploit such parallelism, however, most of the consistency aspects have to be controlled by software, and the advantage of cheap hardware surveillance by the MMU is partly lost. If the grainsize of objects tends to be small, then the page-based release consistency is probably less suitable than true sharing at the object-level.

TREADMARKS [Keleher et al. 93] extends MUNIN's eager release consistency to a lazy release consistency scheme. To reduce the communication bandwidth requirements, page differencing is rigorously employed. In contrast to PANDA, page versions differing by more than 1 can be handled. Vector timestamps are maintained to keep track of the causal order between the various differences.

MIDWAY [Bershad et al. 93] proposes so-called entry consistency as a model of page coherence. It tries to minimize communication costs by aggressively exploiting the relationship between shared objects and the synchronization objects which protect them. Entry consistency is somewhat weaker than release consistency because it is more dependent on the program's correct use of the available synchronization primitives offered by the system.

ORCA, a language for parallel programming in a distributed environment [Bal and Kaashoek 92], provides sharing at the level of shared objects. Due to its specialized programming model, ORCA avoids problems concerning the compliance with existing programming languages. A distributed prototype environment has been implemented based on object replication and reliable broadcast to achieve consistent object sharing. Consistency is maintained with a write-update protocol.

Generally, most systems supporting sharing on a per-object basis provide remote method invocation. Under these circumstances, no guarantees for co-location are required to grant access to shared objects. Nevertheless, tying objects together may prove effective in order to accelerate object interaction. Therefore, systems such as EMERALD [Jul et al. 88] and AMBER [Chase et al. 89] offer additional means to explicitly attach objects to each other. Such annotations only serve as hints but they are not guaranteed. Since PANDA enforces co-location, the requirement for remote object invocation does not occur inside a cluster of tied objects.

7. Summary

It has often been claimed that a cluster of workstations, connected by a high-speed local area network and equipped with suitable runtime support software, may serve as a "poor man's supercomputer". From the perspective of available hardware performance, such a point of view is justified. However, we still lack evidence that a distributed environment is a suitable substrate for the execution of classical parallel applications. To put it differently, the question is whether the distributed nature of the target environment could ever be hidden from applications which expect the monolithic appearance of a centralized platform. The aim of the PANDA project was to explore the potential of distributed computing on workstation clusters for general-purpose programs.

In principle, means for efficient parallel and distributed programming can be realized in the context of a sequential programming language. The object-oriented structure of C++ allowed an easy and natural integration of the required abstractions. Compared to an "in kernel" realization, placing appropriate mechanisms in user space gave much more freedom and flexibility to the system designers. Furthermore, such an approach was a necessary prerequisite for rapid prototyping and for customization to particular application needs. So far, supplying most functionality at the user level never led to undue performance penalties — on the contrary, it was the only possible way to achieve sufficiently fine-grained parallelism.

User-level threads are generally considered as highly efficient, encouraging the programmer to exploit "cheap" parallelism. In fact PANDA demonstrates that extremely fast user-level context switching is feasible. However, reducing the costs of parallelism to the performance of the scheduler is too narrow in scope, as it neglects the memory overhead of multithreading. Without concepts such as PANDA's lazy thread creation, a parallel application is always in danger to exhaust the memory resources. This may render parallelism too expensive to be utilized in practice — despite of efficient context switching.

Next to parallelism, the aspect of efficient data sharing is central to distributed computing. The notion of a global virtual address space is an effective way to support conventional programming languages. In particular, DSM — i.e., sharing part of the global virtual address space — naturally extends the local scope of the programming paradigm to the distributed environment. DSM is not only convenient, but can also be offered at reasonable costs to the application programmer in many cases. Potential problems may be caused mainly by false sharing and by the need for global synchronization.

The danger of false sharing may be alleviated by treating shared data at the granularity of application objects, but at the expense of reduced transparency. The flexibility of PANDA permits a co-existence of both DSM and application-level object sharing. We are currently about to explore whether these mechanisms may be utilized in order to not only *co-exist*, but also to *co-operate*.

While we were able to achieve high performance parallelism, our experiments confirmed our supposition that transparent, fine-grained distribution is beyond the limitations imposed, in particular, by the need for global synchronization. Consequently, only those application domains can be tackled in a distributed environment which share data at a relatively coarse grain. It would be naive to assume that a parallel program written for a monolithic architecture could be run on a system such as PANDA without change; to achieve reasonable performance, appropriately partitioning data into suitable local "chunks" is a minimum requirement. Should such a partitioning strategy turn out to be infeasible, a distributed environment is probably not an appropriate choice.

References

[Anderson et al. 92] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, Vol. 10, No. 1, Feb. 1992.

[Assenmacher et al. 93a] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. *PANDA — Supporting Distributed Programming in C++.* In: O. Nierstrasz (ed), Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP'93), LNCS 707, Springer 1993, pp. 361-383.

[Assenmacher et al. 93b] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. *The PANDA System Architecture - A Pico-Kernel Approach.* Proc. of the Fourth Workshop on Future Trends of Distributed Computing Systems (FTDCS'93), Lisbon, Portugal, Sept.1993. IEEE Computer Society Press, Los Alamitos, pp. 470-476.

[Bal and Kaashoek 92] H.E. Bal and F. Kaashoek. *Orca: A Language for Parallel Programming of Distributed Systems.* IEEE Transactions on Software Engineering, Vol. 18, No. 3, March 1992.

[Bershad et al. 93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. *The Midway Distributed Shared Memory System.* Proc. CompCon'93, pp. 528-537, Feb. 1993.

[Carter et al. 91] J.B. Carter, J.K. Bennet, and W. Zwaenepoel. *Implementation and Performance of Munin.* Proc. 13th ACM Symposium on Operating Systems Principles, Pacific Grove, CA, pp. 152-164, Oct. 1991.

[Chase et al. 89] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. *The Amber System: Parallel Programming on a Network of Multiprocessors.* Proc. of the 12th ACM Symposium on Operating Systems Principles, pp. 147-158, 1989.

[Coplien 92] J.O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley 1992, pp. 133-134.

[Draves et al. 91] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean. *Using Continuations to Implement Thread Management and Communication in Operating Systems.* Proc. of the 13th ACM Symposium on Operating Systems Principles, Oct. 1991

[Keleher et al. 93] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. *Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems.* Technical Report COMP TR93-214, Department of Computer Science, Rice University, Houston, Texas, Nov. 1993.

[Fleisch et al. 93] B.D. Fleisch, R.L. Hyde, and N.C. Juul. *Moving Distributed Shared Memory to the Personal Computer: The Mirage+ Eperience*. Technical Report UCR-CS-93-6, Department of Computer Science, University of Califorina at Riverside, CA, 1993.

[Gehani and Roome 88] N.H. Gehani and W.D. Roome. *Concurrent C++: Concurrent Programming with Class(es)*. Software — Practice and Experience 18(12), pp. 1157-1177, Dec. 1988.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. *Fine-grained Mobility in the Emerald System.* ACM Transactions on Computer Systems 6(1), pp. 109-133, Feb. 1988.

[Liedtke 93] J. Liedtke. *Improving IPC by Kernel Desig*n. Proc. 14th ACM Symposium on Operating System Principles, pp. 175-188, Dec. 1993

[Marsh et al. 91] B. D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P.Markatos. *First-Class User-Level Threads*. Proc. of the 13th Symp. on Operating Systems Principles, Pacific Grove (California), pp. 110-121, Oct. 1991.

[Misra 91] J. Misra. *Loosely-Coupled Processes*. Proc. Parallel Architectures and Languages Europe (PARLE'91), Springer-Verlag, LNCS 506, E.H.L. Aarts, J. van Leeuwen, and M. Rem (eds.), pp. 1-26, Eindhoven, The Netherlands, June 1991.

[Nierstrasz and Papathomas 90] O.M. Nierstrasz and M. Papathomas. *Viewing Objects as Patterns of Communicating Agents*. Proc. of European Conference on Object-Oriented Programming and ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (Ottawa, Canada, Oct. 21-25), ACM, New York, pp. 38-42, 1990.

[Nitzberg and Lo 91] Nill Nitzberg and Virginia Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms.* IEEE Computer, pp. 52-60, Aug. 1991.

[Rao and Kumar 93] V.N. Rao and V. Kumar. *On the Efficiency* of *Parallel Backtracking*. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 4, pp. 427-437, April 1993.

[Saleh and Gautron 91] H. Saleh and P. Gautron. *A Concurrency Control Mechanism for C++ Objects*. Proc. of the ECOOP'91 Workshop on Object-Based Concurrent Computing (Geneva, Switzerland, July 15-16), LNCS 612, Springer-Verlag, pp. 95-210, 1991.

[Scott et al. 90] M.H. Scott, T.J. LeBlanc, B.D. Marsh. *Multi-Model Parallel Programming in PSYCHE*. Proc. 2nd ACM Symposium on Principles and Practice of Parallel Programming, pp. 70-78, March 1990

[Shapiro 86] M. Shapiro. *Structure and Encapsulation in Distributed Systems: The Proxy Principle.* Proc. 6th Int. Conference on Distributed Computer Systems, pp. 198-204, Cambridge, MA, May 1986.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.