Report on the PIONIER project

“Programming Parallel and Distributed Computer Systems”

(January 1993 - June 1998)

Henri E. Bal

Department of Mathematics and Computer Science
Vrije Universiteit, Amsterdam

1. Introduction

This report summarizes the research results of the project “Programming Parallel and Distributed Computer Systems,” which took place at the department of Mathematics and Computer Science of the Vrije Universiteit during 1 January 1993 to 30 June 1998. This project was funded by the Netherlands Organization for Scientific Research (NWO) through a PIONIER grant (PGS 62-382) awarded to Henri Bal.

Parallel (or high-performance) computing is being used more and more often for solving computationally intensive problems. Unfortunately, developing correct, portable, and efficient parallel software is a difficult task, which limits further acceptance of parallel computing. The goals of the PIONIER project are to ease the task of parallel programming as much as possible, while also achieving good performance and high portability of parallel programs. These three goals (ease of use, performance, and portability) are often conflicting, but all are crucial to the success of parallel programming.

Our research followed three directions: to find the right level of abstraction (or model) for a parallel language, to study efficient and portable implementation techniques for parallel languages, and to evaluate all our ideas (about models and their implementation) using realistic applications. The research therefore has aspects of:

- Programming language design and programming models.
- Systems software (compilers, runtime systems, communication software).
- Parallel applications.

A distinguishing feature of our research program is that we pay considerable attention to each of these areas. Many other parallel languages have been designed that were implemented only in a prototype way (or not at all) and that have been used only for trivial applications.

As a result, our work has obtained a high visibility, resulting in many international contacts (see Section 10). The PIONIER project has resulted in several publications in top journals with a high impact, including *ACM Transactions on Computer Systems*, *ACM Transactions on Programming Languages and Systems*, *IEEE Computer*, and three papers in *IEEE Concurrency*. The success of the project also resulted in two major research grants from the Vrije Universiteit (VU), one funding a group of substantial size for the next four years and one funding a large-scale parallel system. The VSNU research evaluation gave the highest possible ranking for the research of the Computer Systems group, in which the PIONIER group participates.

The report is structured as follows. Section 2 gives some background information, including our earlier work on parallel languages and the hardware infrastructure used during the project. The rest of the report consists of three parts, discussing systems software (Sections 3-5), applications (Section 6), and programming language design issues (Sections 7-9). In Section 3 we discuss a new Orca system, which is a cornerstone of our research. Section 4 examines parallel programming on high-speed networks. Section 5 discusses wide-area parallel programming. Experience with parallel

Orca applications is described in Section 6. This work resulted in many new insights, which stimulated further research on programming models, as described in Sections 7 and 8. Section 9 outlines our work on very high-level languages. In Section 10, we look at our cooperation with other researchers. Finally, in Section 11 we analyze the outcome of the project. For each project, the primary researchers working on the project are given. Prof. Bal supervises all projects. A list of all the members of the PIONIER group is given in Appendix A.

2. Background

Below we describe our earlier work on the Orca language and we look at the hardware infrastructure used in the project.

The Orca language

Before the PIONIER project started, we already had some experience in designing and implementing parallel languages. In particular, we developed the Orca language, which is a procedural, object-based language for parallel programming on distributed systems. The design and prototype implementation were done mainly by Henri Bal and Frans Kaashoek, as part of their Ph.D. research (during 1987-1992). Since Orca plays an important role in the PIONIER project, we describe it briefly here.

Most parallel languages and libraries use either message passing or shared variables for expressing communication. Unfortunately, message passing is difficult to program, while shared variables require shared memory for an efficient implementation, which is hard to implement on a large-scale system. The idea behind Orca is to provide a programming model similar to shared variables, but designed in such a way that it can be implemented efficiently on (scalable) distributed-memory systems.

The basic model of Orca is that of a collection of processes communicating through *shared objects*. Shared objects are variables that can be accessed by multiple processes. Unlike normal shared variables, shared objects are manipulated exclusively by user-defined, high-level operations, which are expressed using abstract data types (see Figure 1). All operations on an object are executed indivisibly (atomically), and each operation is applied to a single object. The main advantage of the Orca model is that it hides the distributed nature of the system from the user. The shared object model is close to shared variables, so for a programmer it looks as if all processors are connected by a shared memory, which is much easier to program than distributed-memory machines. The shared object model, however, can be implemented without using physical shared memory. Orca can therefore be described as an *object-based distributed shared memory (DSM) system* [4].

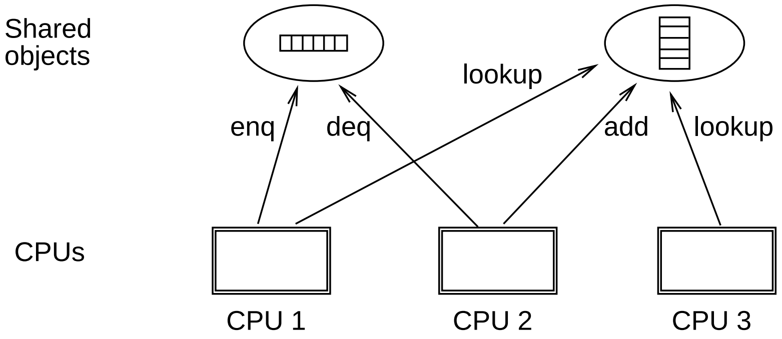

Figure 1: Two shared objects with user-defined operations.

A prototype implementation of Orca was built on top of the Amoeba distributed operating system, which was running on a parallel machine at the VU [11, 13, 39, 40]. An important weakness in this prototype implementation was its lack of portability to other systems. The runtime system, for example, depended on the multithreading and communication primitives provided by Amoeba. In addition, the prototype compiler was rather inefficient.

During the PIONIER project, we benefited in several ways from this earlier work. We studied the Orca programming model in depth and used it to develop more general object-based models. Also, we used Orca as a research vehicle for studying advanced implementation techniques for parallel languages on modern parallel machines. These techniques include runtime systems, compilers, and network communication software. In addition, we also used the Orca system as a tool for implementing several parallel applications. We have developed numerous parallel applications in Orca, often together with people from other research areas (e.g., Physics [37] and AI [3]). Finally, we have applied ideas from the Orca system to other programming languages, including Java, Linda, and SR.

Hardware infrastructure

A significant part of our research is experimental and requires access to a parallel computer whose systems software we are able to change. The machines we used can be classified as so-called *cluster computers* (or *Networks of Workstations*). Unlike supercomputers, clusters are built entirely from off-the-shelf components. Using standard components instead of specially-designed processors and interconnects results in a dramatic reduction in cost. Moreover, the advances in microprocessor technology (e.g., the Pentium Pro, PowerPC, and DEC Alpha) allow clusters to obtain high processing speeds.

We have used two cluster computers (owned by our department) during the course of the project. The first system, called the Zoo*, is a collection of 80 single-board computers (see Figure 2) each consisting of a 50 Mhz MicroSparc with 32 Mbyte local memory. All machines are connected by a 10 Mbit/sec Ethernet. This system runs the Amoeba distributed operating system that was developed by prof. Tanenbaum's group.

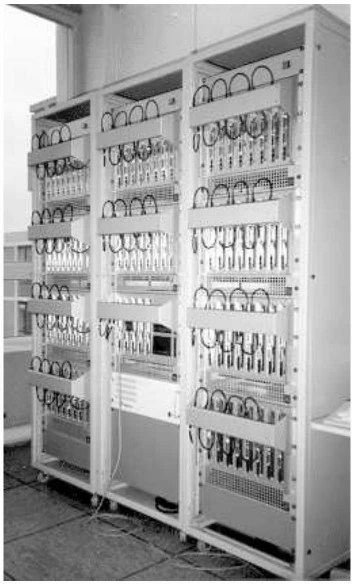

Figure 2: The Zoo: 80 single-board computers (MicroSparcs) connected by Ethernet.

The main problem with this type of system is the high communication overhead of the Ethernet network. For the successor of the system, we therefore decided to use a modern, high-speed network. To determine which network technology is most suitable for parallel computing, the PIONIER group built an experimental testbed consisting of three 8-node clusters that are identical except for the interconnection network; the three clusters use Fast Ethernet, Myrinet, and ATM, respectively. We did

* Because it runs Amoeba, Orca, Panda, Hawk, and other creatures.

performance measurements on communication benchmarks and applications, showing that Myrinet obtains the highest performance of these three networks [9, 26]. We therefore used Myrinet as the high-speed network for the successor of the Zoo.

The second system we used is a cluster computer consisting of 128 PCs. Unlike with the Zoo, the nodes are complete PCs, including a motherboard, hard disk, and PCI cards. Each node contains a 200 MHz Pentium Pro, 128 MByte of memory, and a 2.5 Gbyte local disk. All boards are connected by two different networks: Myrinet (a 1.28 Gbit/sec network) and Fast Ethernet (100 Mbit/sec Ethernet). Myrinet is used as fast user-level interconnect, while Fast Ethernet is used by the operating system. Myrinet uses System Area Network technology, consisting of LANai-4.1 interfaces connected by 8-port crossbar switches. The switches are connected using a 2-dimensional torus topology. The entire system is packaged in a single cabinet and was built by Parsytec (Germany). The system runs the BSD/OS operating system from BSDI.

This cluster computer is part of a wide-area parallel system, called the Distributed ASCI Supercomputer (see Section 5). An initial 64-node cluster was financed partly by an equipment fund from NWO (awarded to the ASCI research school), partly by the VU, and partly by the PIONIER grant. This machine was installed in May 1997. In May 1998, the cluster was upgraded to 128 nodes, using a research grant from the VU given to the departments of Mathematics & Computer Science, Physics & Astronomy, and Chemistry. These departments will do joint research on cluster computing using this 128-node machine. Figure 3 shows a picture of the 128-node cluster, which is called the *Betacluster*.

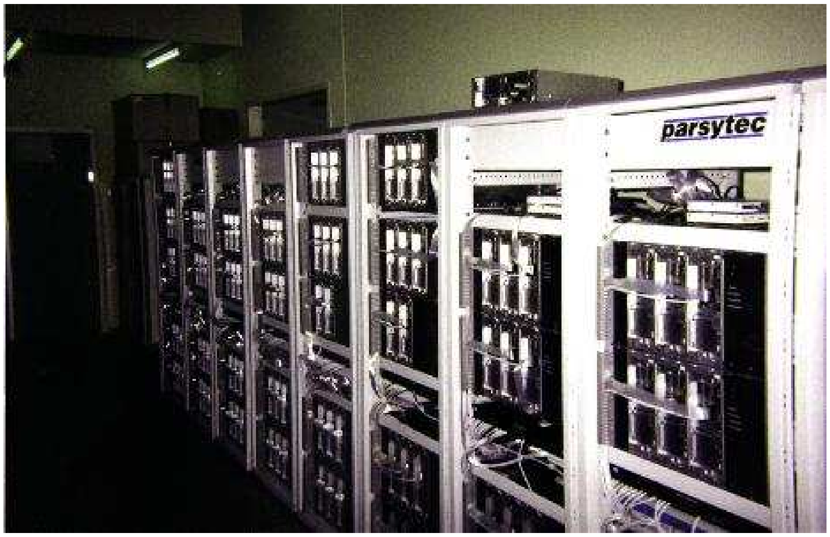

Figure 3: The Betacluster: 128 Pentium Pro PCs connected by Myrinet.

3. The portable Orca system

Project members: Bhoedjang, Langendoen, Rühl, Hofman, Jacobs, Verstoep.

An important result of the PIONIER project is a new, high-performance Orca system that is highly portable and modular. Unlike the original prototype system mentioned above, the current Orca system runs on a wide variety of machines and has been used for many applications. In addition, the software system is a cornerstone of our research on language implementation techniques, communication software, and applications. Below, we describe the design of the system and the most important lessons that were learned from building it. Also, we discuss other languages that we have implemented using certain modules from the Orca system, and we look at a performance visualization tool that is part of the system.

Design

One of the key ideas in the new Orca system is to hide all aspects of the underlying operating system and hardware in a virtual machine, which is called *Panda* [4, 17]. The structure of the system is shown in Figure 4.

Orca compiler
Orca runtime system
Panda
Operating system and hardware

Figure 4: Structure of the Orca/Panda system.

The Orca system consists of three layers, which are implemented on top of the operating system (OS) and/or the hardware. The lowest layer in our system is the Panda virtual machine, which provides a certain functionality to the upper layers, using a well-defined interface. The primitives supported by Panda include lightweight threads, point-to-point message passing, remote procedure call (RPC), and totally ordered group communication (broadcast). We have developed a simple and flexible user-level threads package, called OpenThreads [19], together with Prof. Matthew Haines (University of Wyoming). In addition, we have done extensive research on high-performance communication protocols, in particular for totally ordered group communication and RPCs.

Orca's shared objects are implemented by the second layer, which is the runtime system (RTS). The Orca RTS is implemented on top of Panda. Unlike the original Amoeba RTS, the Panda-based RTS does not depend in any way on the OS; instead, it uses only the primitives provided by Panda. The RTS layer therefore only is concerned with managing objects, and not with doing communication. The most important optimization in the RTS is to *replicate* objects that are read very often. The advantage of replication is that read-only operations can be performed on the local copy, without doing any communication. The RTS uses Panda message passing to implement remote invocations on non-replicated objects and broadcasting to update all copies of a replicated object when the object is changed. Consistency of the replicas is obtained through the total-ordering semantics of Panda's broadcast primitive.

The highest layer in the system is the Orca compiler. During the PIONIER project, we have redesigned the original prototype Orca compiler. The new compiler is much more portable (it generates ANSI C code as output). Also, the code produced by the compiler is of high quality, so Orca programs obtain a performance close to that of C. The compiler performs several optimizations, including common subexpression elimination, code motion, loop-unrolling, and live-variable analysis. The compiler translates operations on objects by generating calls to the RTS. In addition, it generates information (e.g., about object usage) that the runtime system uses to determine which objects should be replicated [10].

A major advantage of the system is its flexibility and modularity. The underlying platforms differ significantly in the functionality they provide. Our software is structured in such a way that it can exploit the functionality provided by the underlying platform without giving up portability. For example, if the message passing primitive provided by the underlying OS or communication hardware is reliable, Panda will make use of that. If it is unreliable, Panda will run its own protocol to make message passing reliable.

In addition, virtually all our software runs in user space (outside the OS). The compiler and RTS always run in user space. Panda can be configured to use the multithreading and communication primitives of the OS, or to run partly or entirely in user space. On modern high-speed networks such as Myrinet, for example, Panda runs entirely in user space and directly accesses the network device, which greatly reduces the overhead of communication calls. Moreover, user space protocols can be changed much more easily than OS protocols.

We have verified the suitability of using a layered approach for obtaining portability by implementing the Orca system on a variety of platforms. We have ported the system to several operating systems (including Solaris, BSD/OS, Linux, Amoeba, and Parix), parallel machines (the CM-5, SP-2, Parsytec GCel, Parsytec PowerXplorer, Meiko CS-2), and networks (Ethernet, Fast Ethernet, Myrinet, ATM). Our experiences in porting the Orca system indicate that our approach to portability indeed is successful. Typically, only a small part of the Panda layer has to be adapted to a new environment. The compiler, RTS, and Orca application programs remain unchanged.

Evaluation of the Orca system

We have made a thorough evaluation of the Orca system and of our design choices. The results of this study were published in a paper in *ACM Transactions on Computer Systems* [4]. Below, we summarize this work.

Most Distributed Shared Memory (DSM) systems replicate (or cache) shared data. Orca differs from most other DSMs, however, in the way replicas are kept consistent. If a write operation (i.e., an operation that changes the shared data) is applied to a replicated object, the Orca system must make sure that the replicas remain coherent. Orca uses a write-update protocol with function shipping: write operations on shared objects are broadcast to all processors and are applied to all copies of the object, thus updating the replicas. Virtually all other DSMs use an invalidation approach (i.e., they delete the replicas after a write). Our performance analysis has shown that write-updating is a good approach to implement an object-based DSM, especially if it is used in combination with other techniques that avoid replicating objects with a low read/write ratio.

Another interesting aspect is the way the Orca system determines which objects to replicate and where to store non-replicated objects. The Orca runtime system uses information about object usage provided by the compiler and also maintains dynamic statistics. By combining this information, the runtime system makes its decisions about object placement. An analysis of ten applications shows that the system is able to make near-optimal decisions in most cases. Most programs achieve a speedup within 1% of that obtained by a version in which the programmer makes all decisions.

An important insight from our work is that two decisions have had a profound impact on the design and performance of the Orca system. The first decision was to access shared data only through abstract data type operations. Although this property requires work from the programmer, it is the key to a high-performance implementation. It often reduces the communication overhead, because an operation always results in only one communication event, even if it accesses large chunks of data. A second important decision was to let the Orca system replicate only those objects that have a high read/write ratio. Since replicated data are written relatively infrequently, it becomes feasible to use a write-update protocol for replicated objects.

As part of the performance evaluation of Orca, we have also done a quantitative comparison between Orca and two other distributed shared memory systems, Treadmarks and CRL. For this experiment, we ported these two systems to the Betacluster. The comparison shows that the Orca programs generally have a lower communication overhead and better speedup.

Other parallel programming systems

Although the Panda portability layer was originally designed for Orca, its modular structure allows it to be used for implementing other languages as well. The advantage of using Panda as an intermediate layer is that it results in portable, efficient, and modular systems. We have also implemented several other parallel programming languages and libraries on top of the Panda interface, some in cooperation with other researchers:

- The SR (Synchronizing Resources) language developed at the University of Arizona and the

University of California at Davis was implemented on our Panda system by Greg Benson from UC Davis.

- A Linda system (MPI-Linda) designed by Joao Carreira (University of Coimbra in Portugal) was ported to Panda [18].
- We ported MPI and PVM (popular message-passing libraries) to Panda.
- Three M.Sc. students (Ronald Veldema, Rob van Nieuwpoort, and Jason Maassen) implemented Java on top of Panda, including a native Java compiler and a fast Remote Method Invocation scheme.

Part of this work is described in a conference paper [36]. The conclusion is that our Panda-based approach indeed is suitable for implementing a variety of systems. An additional advantage of this work is that we get many applications written in other languages that we also use for our performance study of high-speed networks.

Performance visualization

High-level languages like Orca ease writing of parallel programs by increasing the distance between the programming model and the hardware. Unfortunately, this also makes it more difficult to understand the performance of parallel programs. To ease performance debugging, it is essential to provide tools that present the user with performance data at the language level. We have designed a trace package and a viewing tool, *Orcshot*, that address this issue for Orca. Orcshot is based on the Argonne tool *upshot*; we have adapted this tool to Orca and we have made various extensions to it [25].

The Orcshot tool takes as input a *trace* file that is generated by running an Orca application with a special trace package. The trace file contains complete and detailed information on the state transitions of each process and on all system-level communication events and all Orca object accesses. On the systems where Orca runs, logging of an event is accomplished in a few microseconds, so tracing is relatively non-intrusive (except for writing the events to disks, which occurs occasionally and is made visible to the user).

The trace files are visualized with Orcshot (see Figure 5). Orcshot's main display is a Gantt chart, with time along the horizontal axis and threads (processes) along the vertical axis. The Gantt chart can be scrolled in the time domain and arbitrarily zoomed. Events are displayed as small boxes; clicking on an event box opens up a larger box, containing the complete data of the event. Orcshot also allows the user to display only certain types of events. For instance, all events related to one specific object can be selected so the user can focus on this object. Additionally, Orcshot supports optional visualization of lower-level (communication) events. This level is used mainly by the developers of the Orca system. It can also be used by application programmers, but then knowledge of the language implementation is required. Finally, the tool supports user-defined events that signal progress or signify relevant states of the program.

4. Parallel computing on high-speed networks

Project members: Bhoedjang, Langendoen, Rühl, Hofman, Jacobs, Verstoep.

The most important difference between a supercomputer and a cluster computer is the communication speed. Supercomputers like the T3E and SP-2 use specially designed, high-speed interconnects, whereas most cluster computers use off-the-shelf LANs (e.g., Ethernet). During the past few years, however, several network technologies have been developed that obtain performance close to that of supercomputer networks. Examples of such networks are ATM, Myrinet, SCI, and ServerNet. Unlike the proprietary networks used in supercomputers, these new networks are generally available. They are particularly interesting technologies, because they can bridge the performance gap between a supercomputer and a cluster. This becomes clear by comparing the two systems used for our

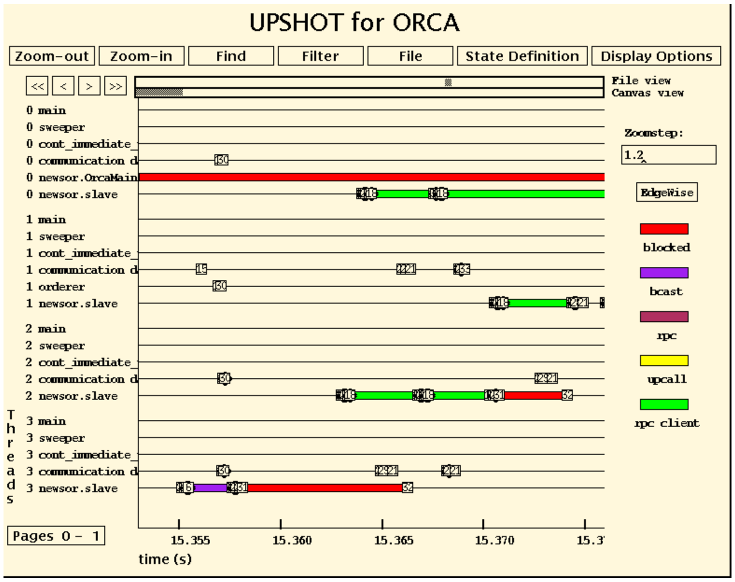

Figure 5: Example output from Orcshot.

research (see Section 2). The Zoo is based on traditional network technology (Ethernet), for which our Panda communication software achieves roundtrip latencies over a millisecond. The Betacluster uses the much faster Myrinet, for which we achieve a minimum roundtrip latency of 20 microseconds. The latter system achieves a communication performance that is close to that of supercomputers like the SP-2, but at a fraction of the cost.

The main problem with high-speed networks, however, is the *software* needed to exploit their potential power. With low-latency networks like Myrinet, the software will become the communication bottleneck. Traditional software designed for Ethernet will have a relatively high overhead on a high-speed network. Typical communication protocols or remote object invocation mechanisms have an overhead of hundreds of microseconds, whereas modern networks support (one-way) latencies on the order of 1-10 microseconds. Therefore, novel communication software is needed for fast networks.

To overcome the problem of high software overhead on high-speed networks, several research groups have resorted to low-level communication models that are easy to implement but hard to program. An important part of our research is to investigate if a high-level programming language like Orca can be implemented efficiently on a high-speed network. We have developed various optimizations for the Orca system. Our basic conclusion from this work is that a high-performance implementation is possible, but that software optimizations at all levels of the system are required to obtain a high performance. We have developed optimization techniques for all layers of our system, including the compiler, runtime system, communication protocols and even the software of the network interface processor [5, 15, 27, 28, 41]. Below, we describe some of this work.

An example of a software-bottleneck is the operating system. With traditional communication protocols, the network device is managed by the operating system kernel and is accessed from user programs through system calls. On a high-speed network, however, the time for a system call may already exceed the message latency. Therefore, much research is being done on user-level communication protocols. We have implemented Panda's threads and protocol stack in user space [19, 29]. On high-speed networks, the network interface also is mapped into user space, thus avoiding all operating system overhead.

An important problem with user-level protocols is how to retrieve incoming messages from the network and handle them. The traditional solution is to let the network device generate an interrupt, but the costs of delivering interrupts to a user process (e.g., through Unix signals) often are very high (exceeding the message latency). An alternative mechanism is to use polling, in which case the application periodically checks if the network has a message available. Polling is not without its problems either, however: it adds a burden on the programmer and it is difficult to get the polling frequency right. We have done extensive research on this issue, and we have designed new solutions that use a combination of polling and interrupts, without any involvement from the programmer [16, 28].

Our research on network interface software focused on what kind of abstraction the low-level communication software should provide to allow higher-level systems (e.g., languages) to be implemented efficiently. We have identified six issues that determine the performance and semantics of a communication system: data transfer, address translation, protection, control transfer, reliability, and multicast [16]. We have done research in several of these issues, resulting in a new network interface protocol for Myrinet, called LFC [15]. One issue we have looked at in detail is broadcast communication. We have devised several novel strategies for implementing broadcast communication on Myrinet by exploiting the programmability of the network interface processor [15, 41].

As a result of this research on compilers, runtime systems, and communication software, we have shown that it is possible to build a programming system for high-speed networks that meets the (often conflicting) goals of ease of use and efficiency. Orca provides a high level of abstraction to the programmer, but still obtains a high performance. On the Betacluster, for example, the latency of a remote object invocation in Orca is 39 microseconds. In addition, we have applied some of our implementation techniques to other programming languages and libraries. In this way, we have obtained high-performance implementations of MPI, PVM, and Java. For example, our implementation of Java Remote Method Invocation (RMI) achieves a best latency (for methods without parameters) of 35 microseconds over Myrinet, which is at least an order of magnitude faster than other implementations of RMI.

5. Wide-area parallel computing

Project members: Plaat, Kielmann, Hofman.

An important emerging trend in parallel computing is to combine computational resources at different locations into integrated, large-scale parallel systems. Such wide-area parallel systems are usually referred to as metacomputers or computational grids. Several large projects are studying software infrastructures for metacomputers, such as the Legion and Globus projects in the US.

In 1997, the Dutch research school ASCI (Advanced School for Computing and Imaging) started a new national project in this research area. The goal of this project is to build a wide-area distributed supercomputer from off-the-shelf components, and to use this system for joint research on parallel and distributed computing. The wide-area system is called DAS (Distributed ASCI Supercomputer) and was funded in part through an equipment grant of NWO/SION.

The DAS system consists of four clusters, located at four ASCI universities: the Vrije Universiteit, the University of Amsterdam, the University of Leiden, and Delft University of Technology (see Figure 6). The four clusters are connected by a wide-area ATM network (Surfnet-4). Each local cluster consists of a number of Pentium Pro processors connected by Myrinet. Three of the clusters have 24 processors each; the cluster at the VU (the Betacluster) has 128 processors, as described in Section 2.

The PIONIER group played an important role in the design of the DAS system and also developed much of the systems software for the DAS clusters. Several other researchers in ASCI have parallel programs based on PVM or MPI, and use our implementations of these libraries (and our Panda communication software).

In 1997, our research group started a new project on wide-area parallel computing (funded partially through a SION grant), using the wide-area DAS system. The goal of this project, called

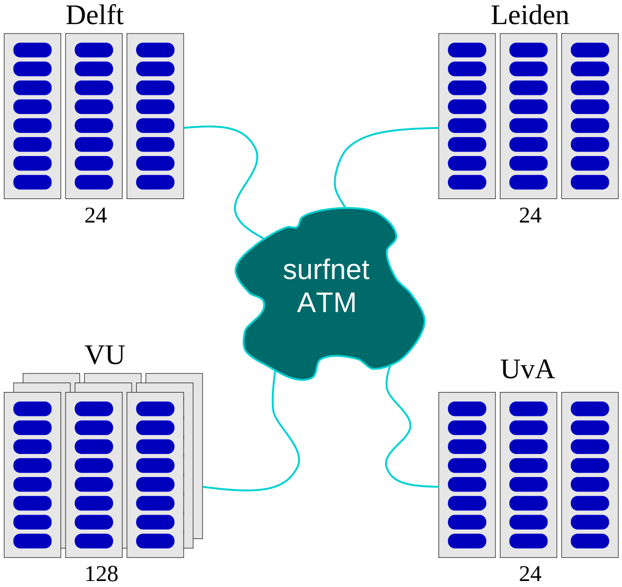

Figure 6: the Distributed ASCI Supercomputer (DAS).

Albatross, is to study languages and applications (and their performance) for wide-area parallel systems. In contrast, most other metacomputing projects focus on issues like fault-tolerance, I/O, resource management, and heterogeneity.

In the first phase of the Albatross project, we have studied the performance of parallel applications on wide-area systems such as DAS. A distinguishing feature of wide-area systems is that the latency and bandwidth of the wide-area network (WAN) are orders of magnitude worse than those of local networks. On the DAS system, for example, the roundtrip latency of the local area network (Myrinet) is about 20 microseconds, while the wide-area network has a latency of several milliseconds. Likewise, the measured throughput is about 50 Mbyte/sec for Myrinet and 0.75 Mbyte/sec for the WAN. So, there is a performance gap on the DAS system of about two orders of magnitude between the LAN and the WAN. Most applications used for metacomputing therefore are embarrassingly parallel (i.e., they barely communicate at all). Unfortunately, this severely restricts the type of application that can be used. An interesting issue therefore is to study the impact of this ‘‘gap’ on the performance of more challenging parallel applications (that do communicate).

We have implemented several nontrivial Orca and MPI applications on the wide-area DAS system and we have analyzed their performance [12, 30]. The results show that many applications experience a dramatic performance degradation when run on a wide-area system, due to the slowness of the WAN links. We have subsequently studied how to optimize these programs, by taking the hierarchical structure of the system into account. The optimizations we used reduce communication traffic between clusters (i.e., over the WAN) or hide intercluster latency. The optimizations substantially improve performance of most applications. As a result, most of the applications run faster on multiple DAS clusters than on a single cluster, showing that the range of applications suited for a meta computer may be much larger than previously assumed. In the next phase of the Albatross project, we will study programming systems that ease the implementation of wide-area parallel applications. For example, we are currently designing a library of collective communication primitives for hierarchical wide-area systems.

6. Parallel applications

Project members: Wilson, Romein.

An important goal of our work is to have many people use the Orca system for real applications. We believe that actual user experience is important to assess the strengths and weaknesses of our approach. Below, we summarize the experiences obtained so far.

Dr. Greg Wilson did an extensive study on the usability of Orca while he was a member of our group. He used a suite of applications (that he had developed earlier) for assessing the usability of parallel programming systems for writing parallel programs. In contrast, benchmark suites such as SPLASH aim to assess only the performance of programming systems and not their ease of use. Wilson's suite, called the *Cowichan problems*, has been selected carefully to cover a wide spectrum of application domains and parallel programming idioms. The suite includes numerical as well as symbolic applications. The applications are: the Turing ring, polygon overlay, image skeletonization, skyline matrix multiplication, game-tree search, and active chart parsing.

During the experiment, six different students each implemented one of the Cowichan problems in Orca, as a project for their M.Sc. thesis, supervised by Wilson. Wilson used this work to do a thorough evaluation of the usability of the Orca system. The lessons learned from this research are described in a joint research paper published in *IEEE Parallel & Distributed Technology* [42].

We are collaborating with dr. Spoelder of the department of Physics and Astronomy of the VU on parallel applications. One application we implemented in Orca is a Monte Carlo simulation of high energy particles in a nonpolar liquid. The results of this work are described in a journal paper [37]. Another application is spline-based modeling of the surface of the human cornea. This technique is used in a system (designed by the department of Physics and Astronomy and the department of Medicine of the VU) for measuring the shape of an eye. A problem with this application is that the spline computations take about an hour on a single workstation, whereas a doctor using the equipment would like to see the result almost immediately. We have reduced the computation time by developing a parallel Orca program for solving nonlinear estimation problems.

Several other M.Sc. students and visiting researchers have worked on various Orca applications, including neural networks [38], retrograde analysis [3, 6], N-body simulation [31], the arc consistency problem [1], DNA sequence comparison, ray tracing, automatic test-pattern generation for combinatorial circuits, and CYK-parsing using dynamic programming. Finally, we have implemented most of the SPLASH applications in Orca.

We have learned several important lessons from all this work. The basic conclusion is that Orca achieves its primary goal of being easy to learn and use [42]. Orca's shared object model significantly simplifies parallel programming, by hiding the underlying communication hardware from the programmer. Also, this model is easy to learn, since it is based on abstract data types, with which most programmers are familiar. In addition, most Orca applications obtain reasonable or good performance (speedups).

However, Orca's model and implementation also have some shortcomings. The main problems with its programming model were:

- It is difficult to write data-parallel applications in Orca, because its shared data structures (objects) cannot be partitioned among multiple machines.
- The model allows operations to be applied to only a single object, which is a severe restriction for several applications.

These observations stimulated further research on the Orca model, as described in the next two sections.

7. Data-parallel Orca

Project members: Ben Hassen, Jacobs, Rühl.

Like many other parallel languages, Orca is based on *task parallelism*. With this model, the programmer can create any number of parallel tasks (processes) that can interact in arbitrary ways. Other parallel languages (e.g., High-Performance Fortran, pC++) use *data parallelism*. Data parallelism is based on applying the same operation in parallel on different elements of a data set. Unlike with task parallelism, all processors conceptually execute the same program, on different data elements. The advantage of data parallelism is that it uses a simpler model. The programmer mainly is responsible for specifying the distribution of data structures and the compiler takes care of generating the necessary code for communication and synchronization. Unfortunately, the simple model of data parallelism also makes it suitable only for a restricted class of applications. In particular, applications that use irregular data structures often do not match the model and impose difficult problems on both the language designer and compiler writer. Such applications are often easier to write in a task parallel language.

Since both task and data parallelism thus have their strengths and weaknesses, it is attractive to integrate both forms in one model. Several research projects are working on such an integration [8]. We have done research on adding data parallelism to Orca. The key idea is to allow array-based objects to be partitioned and distributed among multiple machines. An operation on such a partitioned object is executed in a data-parallel way, on the different elements of an array. The new object model is much more general than the original model, and supports single-copy, replicated, and partitioned objects.

The result of this work is a language that supports task and data parallelism in a clean way, using a simple model. We have extended the Orca compiler with these new language constructs. We have developed a runtime system (called Hawk) for data parallel objects and we have integrated it in the original Orca RTS. Also, we have implemented several applications in the extended language. Data-parallel applications indeed are much easier to write in the new language. In addition, several applications exist that can exploit task and data parallelism in a single program. Such applications are hard to express in a language that only supports one form, but they turned out to be easy to write in the new language. This work is described in several papers [20, 21, 22, 23, 24].

8. Advanced object-based programming models

Project member: Rühl.

An important restriction in the original Orca model is that it allows atomic operations to be applied to only a single object. This restriction was imposed because it allows an efficient implementation on a distributed system. For several applications, however, this restriction complicates programming. We have designed two extensions to the basic Orca model that address this problem. One extension, called *atomic functions*, allows the programmer to define functions that are applied indivisibly to a collection of objects [33]. Another extension, *nested objects*, allows objects to be structured using multiple sub-objects, which may be located on different machines. Operations on the entire (structured) object will execute indivisibly, even if they access multiple sub-objects on different machines. To a certain extent, the data-parallel extensions described in Section 7 also have this property, but (as in most data-parallel languages) they restrict objects to contain only array-based data structures. With nested objects, arbitrary data structures can be defined.

The new models greatly enhance the expressiveness of the original Orca model, but also make an efficient implementation much more challenging. We have designed a flexible framework for implementing these (and possibly other) extensions. The new system also uses the Panda virtual machine as its lowest layer, to provide basic communication and threads primitives. On top of Panda, the new system defines the following layers:

- A layer with high-level communication primitives (e.g., collective communication).
- A *generic* runtime system, which provides primitives (e.g., data dependency resolution, continuations) that are useful for implementing several different programming models.
- A number of *model-specific* runtime systems, each of which implements an object model (e.g., single-copy objects, replicated objects, nested objects, atomic functions).

The implementation model underlying the system is called *collective computation*. The idea of this model is to have all processors participate in operations (e.g., atomic functions), so they can together decide on the best execution of the operation and minimize the communication overhead.

We have also developed a new mechanism, called *communication schedules* [34], which can be used to implement collective communication efficiently. With communication schedules, the communication pattern of a collective operation is expressed (by the programmer or a compiler) before the operation is executed, using a set of simple functions. The runtime system uses this communication pattern to implement the collective operation as efficiently as possible. It can, for example, decrease the number of acknowledgement messages needed or eliminate the overhead of context switching during data forwarding [34]. We have used communication schedules for implementing several collective operations (similar to those used in the MPI standard).

Another novel idea in the implementation concerns the synchronization of operations on multiple objects. To address this problem, we have developed an abstraction called *weaver* [35], which is a function that is executed by a collection of processors. We have applied this idea in a runtime support system for atomic functions and nested objects.

9. Very high level languages

Project members: Romein, Plaat.

Another research direction we are taking is that of very high level, application-oriented, languages with implicit parallelism. The idea is to define application-specific languages that are very easy to use for one application domain. The language's compiler uses knowledge about the application domain to automatically generate a parallel program from the source code. The advantages of this approach are ease of use and automatic parallelization. The disadvantage is reduced flexibility, since each language is suitable for only one application domain.

As part of this project, we have developed a very high-level language for game playing, called *Multigame* [32], in which board games like chess, checkers, and othello can be expressed. A Multigame program essentially is a formal description of the rules of a game. From this description, the Multigame compiler automatically generates a parallel program that will play this game (see Figure 7). The Multigame system uses four different parallel search strategies, based on the alpha-beta search, MTD(f), negascout, and IDA* algorithms. In addition, the system provides a number of well-known heuristics, such as transposition tables, the history heuristic, and iterative deepening. An important issue is how to reduce the communication and search overhead of a parallel game tree search program.

As an example, consider transposition tables, which store values of positions that have already been analyzed. If each processor keeps a private transposition table, there will be no communication overhead. However, processors will then not have any knowledge about which positions have been analyzed by other processors, which leads to a large search overhead. If, on the other hand, all transposition table information is communicated among all processors, this search overhead will be avoided but the communication overhead will be large. We have studied this problem in detail, and investigated various alternatives (including replicated and partitioned transposition tables).

A related idea that we investigated is to exploit programmable network interfaces to speed up transposition table operations [14]. Several modern network interfaces (e.g., Myrinet) contain a programmable processor. Many research groups have optimized message passing libraries by

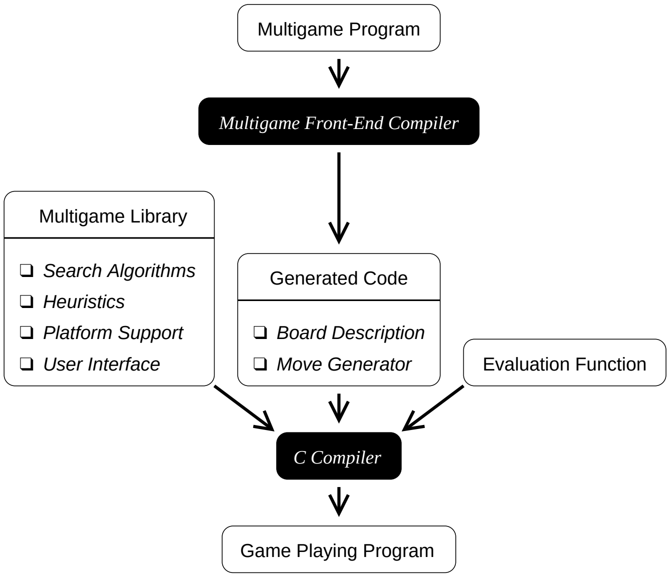

Figure 7: structure of the Multigame system.

redesigning the software (firmware) of this processor. We have investigated firmware support for application-specific shared data structures, in particular transposition tables. We have developed customized software that runs partly on the network processor and partly on the host. The customized software greatly reduces the overhead of interactions between the network interface and the host. Also, the software exploits application semantics to obtain a simple and fast communication protocol. Our results show that for partitioned tables the maximum number of data structure operations per second per processor is increased from 21,000 to 72,000. In addition, a performance improvement up to a factor 2.5 is achieved at the application level.

10. Contacts with other researchers

During the project, we have cooperated with numerous researchers in the Netherlands and elsewhere. Prof. Frans Kaashoek (MIT Laboratory for Computer Science) has worked with us on the Orca and Panda systems for many years and has contributed many valuable ideas. We cooperated with the group of Prof. Tanenbaum of our department on communication and operating system support for parallel languages; also they developed the Amoeba distributed operating system, on which we did most of our experimental work during the first four years of the project. Dr. Dick Grune of our department cooperates with us on the Multigame project. Dr. Victor Allis worked with us on parallel retrograde analysis while he was in the AI group of the VU. In 1996, we started an extensive collaboration with the Physics Applied Computer Science group of the Department of Physics and Astronomy of the Vrije Universiteit, to work on parallel applications in physics, as described in Section 6.

Several scientists from the US and elsewhere stayed for a few months in our research group. Prof. Matthew Haines (University of Wyoming) spent one month (March 1996) in our group to initiate joint research on thread packages and on languages with integrated task and data parallelism. This work so far resulted in a joint publication in an IEEE journal [8] and a conference publication [19]. Greg Benson (UC Davis) spent four months (March through June 1996) in our group. He used our Panda system to design a portable implementation of the SR language [36]. Joao Carreira (University of Coimbra) visited our group for one month (November 1996) to implement a Linda system on top of Panda [18]. Matt Welsh (currently at UC Berkeley) ported the U/Net system to Myrinet while working in our group (in May and June 1997). Dr. Mario Lauria (currently at UC San Diego) stayed in our group during August 1997 to work on collective communication using our DAS system.

We also participated in the TEMPUS project ’DISCO’ ("Development in Romania of short-term higher education in computing, centered on distributed processing and its applications"). Several staff members and students from Romanian universities spent a few months in our group as part of this project, to do research on parallel programming. Prof. Irina Athanasiu (Polytechnical University of Bucharest) cooperated with us on data-parallel Orca [21] and parallel applications [1].

Members of our group spent some time at MIT, the University of Edinburgh, and Cornell University and attended numerous international conferences and workshops during the course of the project. Bal was program chair of the IEEE Computer Society 1994 International Conference on Computer Languages (Toulouse, 1994), co-chair of the Internet Programming Languages workshop (Chicago, 1998), and program committee member of about fifteen conferences and workshops. Dr. Langendoen was program chair of the first and second Workshop on Runtime Systems for Parallel Programming (Geneva, April 1997 and Orlando, March 1998).

We have also had several contacts with industry, mainly through M.Sc. students. Various students supervised by Prof. Bal did their final project at an industrial or governmental research institute, including NLR, GAK, RIVM, CBS, and Tijl Data. Other students did projects at foreign universities, including the University of Bologna, the University of Lancaster, and the University of Lingkoping. Several foreign students (from Technische Universität Graz and Polytechnical University of Bucharest) did a M.Sc. project in our group.

All researchers of our group participate in the ASCI (Advanced School for Computing and Imaging) research school. Bal is a member of the board of ASCI and was chairman of the ASCI’97 conference and of the DAS workshop (March 1998). Our group also plays an important role in the DAS project.

11. Discussion

In the project, we did research on programming support for parallel and distributed computer systems. The PIONIER grant allowed us to do this research in much greater depth than the majority of other (international) projects. We were able to study (1) programming models and languages that make parallel programming on distributed systems as easy as possible; (2) advanced techniques to implement these models efficiently and in a portable way; and (3) to evaluate the models and implementation techniques using realistic applications. In comparison, many other projects investigate only one of these issues in isolation. Since the combination of ease-of-use, efficiency, and portability is a challenge, studying all these issues in depth is of great value.

Our research project has obtained a high visibility. We have many international contacts and members from our group are often invited for giving talks or joining program committees (e.g., Bal was selected for the IEEE CS European Distinguished Visitors Program). The project resulted in several publications in top journals. The most important publications of the PIONIER project are our papers in *ACM Transactions on Computer Systems* [4] and *ACM Transactions on Programming Languages and Systems* [23]. TOCS is the most prestigious journal in the area of computer systems and publishes about 16 papers a year. Also, we have a paper in *IEEE Computer*, and three papers in *IEEE Concurrency* (formerly called *IEEE Parallel and Distributed Technology*). A paper about our LFC protocol [15] received the best-paper-award (out of 214 submissions) at the 1998 International Conference on Parallel Processing (Minneapolis). The project also produced useful software, which further increased our visibility. The sources of the Orca system are available through the World Wide Web and have been downloaded by more than 200 people.

The PIONIER project also had a large impact within the university. The VSNU research evaluation gave the highest possible ranking (i.e., ‘‘excellent’’ for all criteria) for the research of the Computer Systems group (which consists of the PIONIER group and prof. Tanenbaum’s distributed systems group). On 1 April 1998, Dr. Bal was appointed part-time (0.4) professor in the Department of Mathematics and Computer Science and the Department of Physics and Astronomy, to work on physics-applied computer science (in particular parallel and interactive applications). As a result, the collaboration with the Physics department will be intensified significantly in the near future. Also, the departments of Mathematics & Computer Science, Physics & Astronomy, and Chemistry received a substantial research grant from the VU to cooperate in a new cluster computing project that started recently. This grant allowed us to extend our cluster to 128 nodes. Finally, Bal received a research grant from the VU (‘‘Universitair Stimulerings Fonds’’), funding a group of substantial size during the next four years. This group will do research in several areas, including parallel object-based languages, wide-area parallel programming, distributed shared memory, and interactive applications.

Appendix A: Research group

The table below gives an overview of the research group. Dr. Plaat was funded through the PIONIER grant during May to October 1997 and subsequently by a SION grant. Dr. Hofman was funded through the PIONIER grant until 1 June 1998, and currently is employed by the VU.

Name	Position	From	To	Funding
Prof. dr. ir. H.E. Bal	Professor			VU
Dr. K.G. Langendoen	Associate Researcher	1/1/93	31/1/98	NWO
Drs. T. Rühl	Ph.D. student/researcher	1/7/93	31/5/98	NWO
Drs. R.A.F. Bhoedjang	Ph.D. student/researcher	1/7/94		VU
Drs. J.W. Romein	Ph.D. student	1/9/94		VU
Drs. J.C.H. Jacobs	Programmer			VU
Dr. R.F.H. Hofman	Programmer			NWO/VU
Drs. C. Verstoep	Programmer			VU
Dr. S. Ben Hassen	Postdoc	1/9/94	31/12/96	NWO
Dr. G.V. Wilson	Postdoc	1/1/94	31/9/94	EC
Dr. A. Plaat	Postdoc	1/5/97		NWO/SION
Dr. T. Kielmann	Postdoc	1/3/98		VU

Acknowledgements

I am very grateful to the Netherlands Organization for Scientific Research (NWO) and the Department of Mathematics and Computer Science for supporting this research. I also thank all project members for all their work and enthusiasm, and for making this project a success. Many other people also made important contributions to the project in various ways, including Andy Tanenbaum, Frans Kaashoek, Dick Grune, Reind van der Riet, all the visitors mentioned in Section 10, and the more than thirty students who did their M.Sc. work as part of the project. Finally, I thank Rob Laan and Ingrid Pijpers for doing the financial administration of the project.

Publications

Below is a list of our publications during the PIONIER project. The list includes a book [7], several chapters of books (based on papers that were published earlier) [11, 13, 40], a contribution to a special issue of a journal [2], and many refereed papers. Many papers can be obtained through the World Wide Web, see http://www.cs.vu.nl/orca/ and http://www.cs.vu.nl/albatross/.

1. I. Athanasiu and H.E. Bal, ‘‘The Arc Consistency Problem: a Case Study in Parallel Programming with Shared Objects,’’ *7th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, pp. 816-821 (Oct. 1994).
2. H.E. Bal, ‘‘Evaluation of KL1 and the Inference Machine,’’ *Future Generations Computer*

Systems **9**, pp. 119-125 (1993).

3. H.E. Bal and L.V. Allis, ‘‘Parallel Retrograde Analysis on a Distributed System,’’ *Supercomputing ’95*, San Diego, CA (Dec. 1995).

4. H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M.F. Kaashoek, ‘‘Performance Evaluation of the Orca Shared Object System,’’ *ACM Transactions on Computer Systems* **16**(1), pp. 1-40 (Febr. 1998).

5. H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and K. Verstoep, ‘‘Performance of a High-Level Parallel Language on a High-Speed Network,’’ *Journal of Parallel and Distributed Computing (Special issue on Workstation Clusters and Network-based Computing)* **40**(1), pp. 49-64 (Jan. 1997).

6. H.E. Bal, R. Bhoedjang, K. Langendoen, and F. Breg, ‘‘Experience with Parallel Symbolic Applications in Orca,’’ *Journal of Programming Languages* (1998).

7. H.E. Bal and D. Grune, *Programming Language Essentials*, Addison-Wesley, Wokingham, England (1994).

8. H.E. Bal and M. Haines, ‘‘Approaches for Integrating Task and Data Parallelism,’’ *IEEE Concurrency* **6**(3), pp. 74-84 (July-August 1998).

9. H.E. Bal, R. Hofman, and K. Verstoep, ‘‘A Comparison of Three High Speed Networks for Parallel Cluster Computing,’’ *Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC’97), Lecture Notes in Computer Science, Vol. 1199, D.K. Panda and C.B. Stunkel (Eds.),* San Antonio,Texas, pp. 184-197, Springer-Verlag (February 1997).

10. H.E. Bal and M.F. Kaashoek, ‘‘Object Distribution in Orca using Compile-Time and Run-Time Techniques,’’ *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications 1993 (OOPSLA)*, Washington, D.C., pp. 162-177 (26 Sept. 1993 - 1 Oct. 1993).

11. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ‘‘Orca: A Language for Parallel Programming of Distributed Systems,’’ pp. 36-51 in *Programming Languages for Parallel Processing*, ed. D.B. Skillicorn and D. Talia, IEEE Computer Society Press (1994).

12. H.E. Bal, A. Plaat, M.G. Bakker, P. Dozy, and R.F.H. Hofman, ‘‘Optimizing Parallel Applications for Wide-Area Clusters,’’ *International Parallel Processing Symposium*, Orlando, FL, pp. 784-790 (April 1998).

13. H.E. Bal and A.S. Tanenbaum, ‘‘Distributed Programming with Shared Data,’’ in *Distributed Shared Memory: Concepts and Systems*, ed. J. Protic, M. Tomasevic, and V. Milutinovic, IEEE Press (1997).

14. R. Bhoedjang, J. Romein, and H.E. Bal, ‘‘Optimizing Distributed Data Structures Using Application-Specific Network Interface Software,’’ *International Conference on Parallel Processing*, Minneapolis, MN, pp. 485-492 (August 1998).

15. R. Bhoedjang, T. Rühl, and H.E. Bal, ‘‘Efficient Multicast On Myrinet Using Link-Level Flow Control,’’ *International Conference on Parallel Processing (best paper award)*, Minneapolis, MN, pp. 381-390 (August 1998).

16. R. Bhoedjang, T. Rühl, and H.E. Bal, ‘‘User-Level Network Interface Protocols,’’ *IEEE Computer* (Nov. 1998, accepted for publication).

17. R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek, ‘‘Panda: A Portable Platform to Support Parallel Programming Languages,’’ *Symposium on Experiences with Distributed and Multiprocessor Systems*, San Diego, pp. 213-226 (22-23 September 1993).

18. J. Carreira, J. Gabriel Silva, K. Langendoen, and H.E. Bal, ‘‘Implementing Tuple Space with Threads,’’ *Euro-PDS’97*, Barcelona, Spain (June 1997).

19. M. Haines and K. Langendoen, ‘‘Platform-Independent Runtime Optimizations Using OpenThreads,’’ *11th International Parallel Processing Symposium*, Geneva, Switzerland (April 1997).

20. S. Ben Hassen, ‘‘Prefetching Strategies for Partitioned Shared Objects,’’ *Proceedings of the 29th Hawaii International Conference of System Sciences*, Hawaii, pp. 261-271 (Jan. 1996).

21. S. Ben Hassen, I. Athanasiu, and H.E. Bal, ‘‘A Flexible Operation Execution Model for Shared Distributed Objects,’’ *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’96)*, San Jose, CA, pp. 30-50 (Oct. 1996).

22. S. Ben Hassen and H.E. Bal, ‘‘Integrating Task and Data Parallelism Using Shared Objects,’’ *10th ACM International Conference on Supercomputing*, Philadelphia, PA, pp. 317-324 (May 1996).

23. S. Ben Hassen, H.E. Bal, and C. Jacobs, ‘‘A Task and Data Parallel Programming Language based on Shared Objects,’’ *ACM. Trans. on Programming Languages and Systems* (1998, accepted for publication).

24. S. Ben Hassen, H.E. Bal, and A.S. Tanenbaum, ‘‘Hawk: a Runtime System for Partitioned Objects,’’ *Journal of Parallel Algorithms and Applications* **12**, pp. 205-230 (Aug. 1997).

25. R. Hofman, K. Langendoen, and H.E. Bal, ‘‘Visualizing High-Level Communication and Synchronization,’’ *IEEE Int. Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, Singapore, pp. 37-43 (June 1996).

26. K. Langendoen, R. Hofman, and H.E. Bal, ‘‘Challenging Applications on Fast Networks,’’ *Fourth International Symposium on High-Performance Computer Architecture (HPCA-4), IEEE CS*, Las Vegas, Nevada, pp. 68-79 (Feb. 1998).

27. K. Langendoen, R. Bhoedjang , and H.E. Bal, ‘‘Models for Asynchronous Message Handling,’’ *IEEE Concurrency*, pp. 28-38 (April-June 1997).

28. K. Langendoen, J. Romein, R. Bhoedjang, and H.E. Bal, ‘‘Integrating Polling, Interrupts, and Thread Management,’’ *Proceedings of Frontiers’96*, Annapolis, MD, pp. 13-22 (Oct. 1996).

29. M. Oey, K. Langendoen, and H.E. Bal, ‘‘Comparing Kernel-space and User-space Communication Protocols on Amoeba,’’ *15th International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, pp. 238-245 (May 30-June 2, 1995).

30. A. Plaat, H.E. Bal, and R.F.H. Hofman:, ‘‘Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects,’’ *Fifth International Symposium On High Performance Computer Architecture (HPCA-5)*, Orlando, FL (Jan. 1999, accepted for publication).

31. J. Romein and H.E. Bal, ‘‘Parallel N-Body Simulation on a Large-Scale Homogeneous Distributed System,’’ *Euro-Par’95 (Lecture Notes in Computer Science 966)*, Stockholm, Sweden, pp. 473-484 (Aug. 1995).

32. J. Romein, H.E. Bal, and D. Grune, ‘‘An Application Domain Specific Language for Describing Board Games,’’ *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’97)*, Las Vegas, NV, pp. 305-314 (July 1997).

33. T. Rühl and H. E. Bal, ‘‘Optimizing Atomic Functions using Compile-Time Information,’’ *Working conference on Massively Parallel Programming Models (MPPM-95)*, Berlin, pp. 68-75 (Oct. 1995).

34. T. Rühl and H.E. Bal, ‘‘A Portable Collective Communication Library using Communication Schedules,’’ *5th EUROMICRO Workshop on Parallel and Distributed Processing*, London, pp. 297-304 (Jan. 1997).

35. T. Rühl and H.E. Bal, ‘‘Synchronizing Operations on Multiple Objects,’’ *2nd Workshop on*

Runtime Systems for Parallel Programming, Orlando, FL (March 1998).

36. T. Rühl, H.E. Bal, G. Benson, R. Bhoedjang, and K. Langendoen, ‘‘Experience with a Portability Layer for Implementing Parallel Programming Systems,’’ *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’96)*, Sunnyvale, CA, pp. 1477-1488 (August 1996).

37. F.J. Seinstra, H.E. Bal, and H.J.W. Spoelder, ‘‘Parallel Simulation of Ion Recombination in Nonpolar Liquids,’’ *Journal of Future Generation Computer Systems, (Special issue on the High-Performance Computing and Networking Conference ’97)* **13**(4-5), pp. 261-268 (March 1998).

38. A.S. Tanenbaum, H.E. Bal, S. Ben Hassen, and M.F. Kaashoek, ‘‘An Object-Based Approach to Programming Distributed Systems,’’ *Concurrency Practice & Experience* **6**(4), pp. 235-249 (June 1994).

39. A.S. Tanenbaum, H.E. Bal, and M.F. Kaashoek, ‘‘Programming a Distributed System Using Shared Objects,’’ *Proc. 2nd Int’l Symposium on High-Performance Distributed Computing*, Spokane, WA, pp. 5-12 (1993).

40. A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, ‘‘Using Broadcasting to Implement Distributed Shared Memory Efficiently,’’ pp. 387-408 in *Readings in Distributed Computing Systems*, ed. T.L. Casavant and M. Singhal , IEEE Computer Society Press (1993).

41. K. Verstoep, K. Langendoen, and H.E. Bal, ‘‘Efficient Reliable Multicast on Myrinet,’’ *1996 Int. Conference on Parallel Processing (Vol. III)*, Bloomingdale, IL , pp. 156-165 (August 1996).

42. G.V. Wilson and H.E. Bal, ‘‘Using the Cowichan Problems to Assess the Usability of Orca,’’ *IEEE Parallel and Distributed Technology* **4**(3), pp. 36-44 (Fall 1996).