Collective Computation in Object-based Parallel Programming Languages

VRIJE UNIVERSITEIT

Collective Computation in Object-based Parallel Programming Languages

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der exacte wetenschappen \ wiskunde en informatica
op donderdag 16 november 2000 om 15.45 uur
in het hoofdgebouw van de universiteit,
De Boelelaan 1105

door

Tim Rühl

geboren te Amsterdam

Promotor: prof. dr. ir. H. E. Bal

Contents

List of Tables

List of Figures

Acknowledgments

The research presented in this thesis would have been impossible without the help of many people, whom I want to recognize here.

First, I would like to thank my advisor, Henri Bal, for all the things he thought me.

Samenvatting

In dit proefschrift onderzoeken we het gebruik van *collective computation* (collectief uitvoeren) in de context van object-gebaseerde parallelle programmeertalen. Bij parallel programmeren gaat het erom computers samen te laten werken aan het oplossen van één probleem. Met collective computation bedoelen we het synchroniseren van een verzameling van computers om gezamenlijk een deel van de operaties uit te voeren die door een parallelle applicatie moeten worden gedaan. Het voordeel van deze samenwerking is synergie: omdat de computers weten dat ze samenwerken, kunnen we bepaalde operaties, zoals synchronisatie en communicatie, goedkoper uitvoeren. Hierdoor wordt de performance van de applicatie beter dan wanneer alle computers ongecoördineerd werken en iedere computer onafhankelijk bepaalt wanneer hij wil synchroniseren of communiceren.

Collective computation integreert een aantal ideeën van andere technieken die gebruikt zijn om parallelle programmeeromgevingen te implementeren. Deze ideeën zijn:

- Sta de programmeur toe om te kiezen of de data naar de computer die de operatie aanroept gaat of de operatie naar de computer die de data heeft.
- Sta toe dat data gerepliceerd wordt op meerdere computers, zodat lees acties efficiënt op deze computers uitgevoerd kunnen worden. In combinatie met het verplaatsen van de operatie naar de computer die de data heeft is het vaak efficiënter om replicatie te combineren met het uitvoeren van de operatie op alle computers, zodat de data correct blijft.
- Laat computers die gesynchroniseerd zijn door middel van collective computation efficiënter communiceren doordat alle computers weten welke data de andere computers nodig hebben. Een computer hoeft niet om de data te vragen, want één van de computers die de data al heeft kan zelf besluiten om de data op te sturen.
- Verspreid data die logisch gezien bij elkaar hoort over meerdere computers, zodat iedere computer zijn deel van de operatie lokaal kan uitvoeren.

Om het gebruik van collective computation voor object-gebaseerde parallelle programmeertalen te illustreren, hebben we twee uitbreidingen toegevoegd aan een bestaande programmeertaal: Orca. Orca is een parallelle programmeertaal

waarin processen met elkaar kunnen communiceren door het uitvoeren van operaties op objecten. Ieder object bevat data; de operaties zorgen ervoor dat de data van het object toegankelijk wordt voor een proces.

We maken onderscheid tussen operaties die alleen de data van het object lezen en operaties die de data van het object ook veranderen. Dit onderscheid maakt het mogelijk om het programmeermodel van Orca ook efficiënt te implementeren op parallelle systemen waarbij het geheugen verdeeld is over de verschillende computers. Het systeem maakt hierbij gebruik van het repliceren van objecten en het consistent houden van de data door operaties op alle replicas van het object uit te voeren. Voor objecten die vaker geschreven dan gelezen worden heeft slechts één computer het object, zodat de kosten van het consistent houden minimaal zijn.

Iedere operatie op een object wordt altijd in zijn geheel uitgevoerd voordat de volgende operatie toegelaten wordt (atomaire synchronisatie). Hierdoor kunnen operaties complexe veranderingen uitvoeren op alle data die binnen een object zit, zonder dat een andere operatie de (inconsistente) tussenresultaten kan zien. Verder kunnen operaties blokkeren totdat de data binnen een object aan een bepaalde voorwaarde voldoet (conditionele synchronisatie).

Orca's programmeermodel heeft twee belangrijke beperkingen. Ten eerste laat het niet toe dat de data van een object verspreid wordt over de computers. Iedere computer heeft ofwel alle data van het object ofwel geen data. Een andere beperking is dat we alleen synchronisatie hebben per operatie; het is niet mogelijk om een reeks van operaties, mogelijkerwijs op verschillende objecten, uit te voeren zonder er zeker van te zijn dat de data van deze objecten in de tussentijd niet verandert.

De uitbreidingen die we in dit proefschrift voorstellen leveren een oplossing voor deze beperkingen. De eerste uitbreiding is het *geneste object*. Een genest object lijkt voor de gebruiker gelijk aan een gewoon object. De data van een genest object bestaat echter niet alleen uit normale data, maar kan ook andere objecten bevatten. Ieder deelobject kan onafhankelijk van de andere deelobjecten op een eigen verzameling van computers geplaatst worden, zodat het mogelijk wordt de data van het stam object te verspreiden. Operaties op het stam object zijn atomair. Verder is het mogelijk om synchronisatie condities uit te drukken gebaseerd op de data van het stam object en/of zijn deelobjecten.

De tweede uitbreiding, de *atomaire functie*, staat het toe om een serie van operaties op een dynamische selectie van objecten atomair uit te voeren. Ook binnen een atomaire functie is het mogelijk om conditionele synchronisatie te gebruiken. Dit maakt een atomaire functie uitermate geschikt voor hoog-niveau synchronisatie, waarbij objecten die tijdens de normale executie van de applicatie ongerelateerd zijn toch in een atomaire step beschouwd en veranderd kunnen worden.

In dit proefschrift bespreken we verder de architectuur van een systeem dat gebruik maakt van collective computation om het originele Orca model en de extensies die we hebben voorgesteld te implementeren. Binnen deze architectuur onderscheiden we de *wever* abstractie, die een implementatie levert van het collective computation model. De naam wever is geassocieerd aan de Engelse naam voor een lichtgewicht proces, de *thread* (draad). De wever zorgt ervoor dat de threads samenwerken.

Een wever is een aanroep van een operatie op een verzameling van computers. Als verschillende wevers tegelijkertijd actief zijn, worden ze op alle computers in dezelfde volgorde uitgevoerd. Dit zorgt ervoor dat de veranderingen die een wever uitvoert op een locale versie van de data van een object op alle computers hetzelfde is. Hierdoor kan een gerepliceerd object consistent gehouden worden.

Een wever kan zichzelf blokkeren en kan weer wakker gemaakt worden door een andere wever. Wanneer een aantal wevers wakker gemaakt worden, zorgt het systeem ervoor dat ze weer uitgevoerd worden in de oorspronkelijke volgorde. Dit vergemakkelijkt het implementeren van conditionele synchronisatie.

Iedere computer waarop een wever wordt uitgevoerd weet dat op de andere computer de wever ook wordt uitgevoerd. Hierdoor is het mogelijk dat iedere computer bepaalt welke gegevens de andere computers nodig hebben zonder dat de andere computers hierom hoeven te vragen. Om dit communicatiegedrag optimaal te laten verlopen hebben we een aparte communicatielaag geschreven die is toegesneden op zulke communicatiepatronen. Door gebruik te maken van karakteristieken van de onderliggende parallelle configuratie (de processoren, het netwerk en het besturingssysteem) kunnen deze communicatiepatronen nog verder geoptimaliseerd worden.

Aangezien ons systeem geschikt is voor verschillende parallelle configuraties (portable), gebruiken we de LogGP methodiek om deze karakteristieken te bepalen. Belangrijke kenmerken zijn: hoe lang duurt het voordat een bericht van een computer een andere computer bereikt; hoeveel berichten kunnen er per seconde verstuurd worden van een computer naar een andere computer; en hoeveel data kan er per seconde verstuurd worden. Vooral voor kleine berichten, waar we in een object-gebaseerde parallelle programmeeromgeving veel mee te maken hebben, levert het gebruik van deze karakteristieken om de communicatie te versnellen winst op.

De architectuur scheidt de modules die de afhandeling van Orca objecten, geneste objecten en atomaire functies implementeren. Om deze scheiding mogelijk te maken, is er een generieke module die voor de interactie tussen deze modules en voor de integratie met de wever abstractie zorgt. Verder zorgt deze generieke module voor het aanmaken van objecten en processen en voor het aanroepen van een operatie op een lokale versie van een object. De combinatie van wevers en de mogelijkheid om lokaal een operatie uit te voeren is voldoende om gerepliceerde objecten te implementeren. Een schrijf operatie creëert een wever op alle computers die een replica van het object hebben. Deze wever voert vervolgens op al deze computers de operatie op het lokale object uit. De generieke module zorgt ervoor dat de operatie wordt uitgevoerd door de module die bij het object hoort.

Door een scheiding aan te brengen tussen de uitvoering (i.e., de module) en de uitvoerder (i.e., de wever) is het mogelijk om een wever die gemaakt is door de ene module een operatie te laten uitvoeren op een object dat beheerd wordt door een andere module. De wever voor een operatie op een genest object, die gemaakt is door de module voor geneste objecten, kan daardoor operaties op de deelobjecten uitvoeren, zelfs als die door de Orca module beheerd worden.

Het schrijven van complexe applicaties is versimpeld door de extensies die we hebben aangebracht aan het originele Orca model. Vooral het programmeren van conditionele synchronisatie over meerdere objecten is versimpeld, doordat de programmeur zich nu alleen hoeft te concentreren op de conditie binnen de atomaire functie en niet op alle mogelijke veranderingen die deze objecten in de tussentijd kunnen ondergaan. Met behulp van geneste objecten kunnen objecten geïmplementeerd worden die specifieke prestatiekarakteristieken bezitten die niet met Orca bereikt kunnen worden, zoals bijvoorbeeld objecten die goed om kunnen gaan met relatief veel schrijf operaties. Door zulke objecten op te nemen in een verzameling van standaard implementaties (een bibliotheek) zijn ze makkelijk herbruikbaar.

De metingen die we met het systeem hebben uitgevoerd geven aan dat de prestaties goed zijn. Als een applicatie gebruik kan maken van de specifieke voordelen van collective computation (dat een set van operaties parallel uitgevoerd kan worden) dan is de applicatie sneller dan de Orca applicatie. Verder laten geneste objecten de mogelijkheid aan de programmeur om optimalisaties aan te brengen binnen het object. Het voor collective computation slechtste scenario, waarin veel synchronisatie plaatsvindt gedurende de operatie, kan toch nog redelijk efficiënt afgehandeld worden door gebruik te maken van collectieve communicatie. Hierdoor wijkt de performance niet veel af van de performance van Orca, en wordt bovendien de schaalbaarheid van de applicatie verbeterd.

Chapter 1

Introduction

Parallel computer systems are designed to let multiple processors work together to solve complicated problems. Typically, such problems require a large amount of computation that has to be performed within a limited amount of time. An example is weather forecasting. Generally, weather forecasting consists of simulating air flow based on measurements such as atmospheric pressure and temperature. The more detailed the simulation, the better the predictions it will make, at the cost of more compute time. Most people consider a good weather prediction useful only if the computation is finished in advance of the day being predicted.

Recent examples of the (possible) use of parallel computer systems are:

- In 1997, the World Chess Champion, Gary Kasparov, lost a six-game match against the computer chess program Deep Blue. Deep Blue ran on a 32-node IBM RS/6000 SP2 computer system. Each node of the SP2 contained eight dedicated VLSI chess processors, for a total of 256 processors [63, 115].

- The Grand Challenge Project involves a large number of research institutes in the United States, which try to solve fundamental problems in science and engineering with broad economic and scientific impact, and whose solutions can be advanced by applying HPCC (High Performance Computing and Communications) technologies. A common feature of many of these Grand Challenge applications is that they involve simulation. Due to limitations of speed and memory in computing systems available at the beginning of the HPCC Program, many simulations could not be completed with sufficient accuracy and timeliness [96]. Examples of Grand Challenge application areas are: the environment (e.g., climate modeling, energy management), manufacturing, biomedicine, national security and national defense, and basic research (e.g., physics simulations, information systems).

- In 1995, France performed nuclear tests in the South Pacific. French officials said the tests were needed to develop simulation technology on computers to make future tests unnecessary. Such simulations require a large amount of computation, which is feasible only on parallel hardware.

The processors of a parallel computer system cooperate to solve such complicated problems. Since processors can only cooperate if there is some form of interaction, the most important part of a parallel computer is the communication infrastructure. If we look at the communication infrastructure, we can distinguish two layers [51]. The first layer, *communication hardware*, describes the communication interface that is provided in hardware. The second layer, the *communication abstraction*, implements the programming model that is presented to the application writer.

A number of programming models exist, such as shared memory, message passing, data parallel, dataflow, and systolic approaches. The first three of these programming models are of interest for this thesis. In a shared memory system, all processors can access (part of) the same memory. Communication between processors takes place by reading and writing certain memory locations. In a message passing system all processors only have their own local memory, which is inaccessible by the other processors. Processors can only communicate with each other using explicit send and receive operations. Finally, in the data parallel programming model, all processors own a part of a large, regular data structure. Each processor can perform operations on all elements of its partition in parallel. The system manages consistency, so that reading an element from a remote partition returns the correct value.

For all programming models mentioned above parallel computer systems exist that are mostly implemented in hardware: hardly any support is needed from the communication abstraction in such systems. Other systems, however, use the communication abstraction layer to implement a programming model on hardware that does not directly support this model. For example, a shared memory programming model can be implemented on hardware that only supports message passing and vice versa.

In this thesis, we are interested in systems that implement a shared memory abstraction on top of message passing hardware. Such systems, called *software distributed shared memory systems*, present a good tradeoff between implementation complexity and performance. In particular, we will look at the expressiveness of software distributed shared memory systems that are based on *shared objects*. We will describe the problems that we encountered in Orca, a parallel programming language based on shared objects. Efficient solutions for these problems will be presented that increase the expressiveness of Orca.

Orca has been used for a large number of parallel applications. Orca is easy to use for small applications, because the shared object model is easy to understand and use. In addition, the language reduces the chance of making programming errors by providing strong typing and advanced data structures (graphs). Larger applications, however, are hindered by the fact that Orca only provides atomic operations on *single* objects, and that Orca does not allow the state of an object to be partitioned over different processors. In this thesis, we will describe two extensions to the shared object model, *atomic functions* and *nested objects*, that increase the expressiveness of distributed shared memory systems based on shared objects.

Section 1.1 describes the generic parallel architecture that we will assume in the remainder of this thesis. Section 1.2 gives a short evaluation of the message passing and shared memory programming models. In Section 1.3, we discuss the limitations of Orca and describe the atomic functions and nested objects extensions. Section 1.4 presents a generic execution model, called collective computation, that can be used to implement these extensions efficiently. Section 1.5 presents an overview of the runtime support system that integrates these extensions. Finally, Section 1.6 presents the contributions made in this dissertation and Section 1.7 gives an outline for the remainder of the thesis.

1.1 Generic parallel architecture

During the early days of parallel computers, most of the research was aimed at supporting a programming model in hardware. A typical example is the data parallel programming model. The first systems that supported data parallel applications actually consisted of arrays of processors [18]. A separate control processor would broadcast the instructions, which were executed in parallel on the array processors. Later generations of data parallel computers had less stringent demands on the hardware. Nowadays, most data parallel systems consist of a generic shared memory or message passing architecture and a compiler that translates the application to a single-program multiple-data (SPMD) executable. This executable program runs on all processors, and each processor updates only its local part of the data and synchronizes only after each phase of the computation, instead of after every instruction. An example of this data parallel hardware trend is the development of the parallel computer systems designed by Thinking Machines: the CM-2 is a SIMD machine, while its successor, the CM-5, is a SPMD machine.

To facilitate reasoning about parallel programming models, we will present the generic parallel architecture (see Figure 1.1) that is presented in [51]. Most research on parallel architectures is converging to this generic architecture (e.g., NOW [7] and Beowulf [104]). The architecture consists of a high-performance, scalable interconnect that connects essentially complete computers. Each computer contains one or more processors, memory, and a communication interface.

The communication interface determines how communication is presented to the user. For message passing systems, the communication interface is connected via the I/O bus (although in some systems, it may be connected via the memory bus, because this bus has a higher clock frequency and bandwidth [95]). Send and receive are implemented as explicit I/O operations. The communication interface

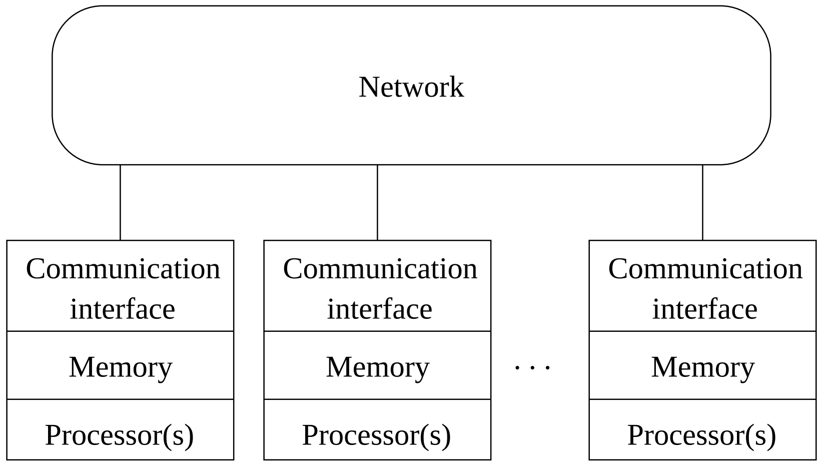


Figure 1.1: Generic parallel architecture (taken from [51]).

and the processor cooperate to determine the source and destination memory locations of each message that is sent or received. Since most I/O buses are standardized, it is relatively cheap to build communication interfaces for message passing systems.

For shared memory, the communication interface is integrated in the memory system, because it has to access memory without any interaction with the host processor(s). For the generic parallel architecture, this results in the nonuniform memory access (NUMA) approach. Access to local memory does not require communication, while access to remote memory does require communication. Caches only hold local data, and do not cache memory from remote nodes.

An extension to the NUMA approach is to allow caches to also hold values from remote nodes. This approach, called cache-coherent nonuniform memory access (CC-NUMA), effectively replicates the data. The cached values, however, have to remain consistent with the corresponding memory location. The *cache coherency protocol* propagates updates to a memory location by either *updating* or *invalidating* each copy [1].

The *memory consistency model* is a description of the behavior of the memory system as seen by the user. Traditionally, the goal was to present a model as close as possible to the model presented by single-processor systems. This model, called *sequential consistency*, guarantees that

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program [82].

The strict requirements imposed by sequential consistency, however, limit the performance of parallel computer systems. Therefore, other consistency models have been introduced that allow more optimizations. These consistency models can be divided in two major categories: models that impose less strict requirements than sequential consistency (e.g., causal memory and processor consistency) and models that provide the illusion of sequential consistency, but only at synchronization points (e.g., weak ordering and release consistency) [94]. In both categories, however, the user has to be aware that the semantics are different from sequential consistency. This makes programming such systems more difficult.

Integrating the communication interface with the memory bus is more difficult and more expensive than integration with the I/O bus, due to the higher performance of the memory bus and because the memory bus is less standardized. Since the granularity of data access is in the order of words (e.g., a cache line), the amount of startup overhead that can be tolerated for each message has to be small. In addition, implementing the cache coherency protocols requires additional hardware, especially if optimizations are required to achieve good performance. For message passing systems, the overhead of message startup can be alleviated by sending fewer and larger messages.

In conclusion, shared memory systems are more complex (and therefore more expensive) to build than message passing systems. The programming model provided by shared memory systems, however, relieves the programmer from working out the exact data communication between processors, thereby making shared memory easier to use for applications that do not map directly onto the message passing model. We will discuss this issue in the next section.

1.2 Parallel programming models

Programming parallel applications is more difficult than sequential programming, due to the communication and synchronization among processors. The advantage of the shared memory model is that it is easier to program than message passing. *Naming* in shared memory systems only involves identifying the right memory location, whereas in message passing the destination processor has to be specified and a message tag that allows the destination processor to handle the message correctly. *Operations* in shared memory are normal read and write operations, while message passing requires explicit send and receive operations.

One of the major research areas in parallel programming is to combine the best of both worlds; i.e., to implement a shared memory model (which is easy to program) on a message passing system (which is easy and cheap to build). In this approach, called *distributed shared memory* or *shared virtual memory*, the communication abstraction layer (sometimes aided by hardware to detect memory accesses) provides the illusion to the programmer of a shared memory system [88].

Implementing shared memory on top of a message passing system imposes the same problems as implementing shared memory in hardware, but now the cost of communication is much larger. For example, accessing a remote memory location requires that the invoking processor sends a message to the owner processor. This owner processor has to receive the message, handle the operation (e.g., read the requested memory location), and send a result message back to the invoking processor. A hardware implementation would involve the communication interface of the destination node, but not its processor.

To avoid excessive communication overhead, these systems try to reduce the number of messages and the amount of data to be sent. One approach to reduce communication is to represent shared data in objects. All accesses to the data in a shared object have to go through the runtime system (i.e., the communication abstraction). This allows the runtime system to maintain consistency per operation, instead of per memory access. The runtime support system regards each object as a unit of coherence.

1.3 Shared data objects

Orca is a distributed shared memory system based on shared data objects. A shared data object is a programmer-defined data structure with a fixed set of operations that can be applied to this object. A shared object can be accessed by multiple processes, possibly running on different processors. Apart from the state of shared objects, no other state is shared between processors or processes. Communication between processors takes place by performing operations on shared data objects. Each operation is atomic, which means that each operation is always completed before another operation can access the object. Each shared data object looks like an area of memory shared by all processors that have a reference to this object. On a distributed system, the runtime support system takes care of all communication and consistency issues if more than one processor accesses an object. This way, an Orca application is the same for a shared memory and for a distributed memory system, only the runtime support systems differ.

During the last ten years, a large number of applications have been written in Orca. For most applications, the shared data object model is easy to use. Furthermore, the model can be implemented efficiently on various parallel systems [11, 12]. Larger applications, however, show some limitations of the shared data object model [123].

One of the most important limitations of the shared data object model is an object cannot be partitioned over the processors. In current implementations, the state of a single object is always stored together, either on a single processor, or replicated on all processors that have a reference to this object. For many applications, however, it would be natural to express a single object in which the state is partitioned (distributed) over the processors.

Take, for example, a job queue object, which can be used to distribute jobs among processes. An implementation of a job queue in a single Orca object would cause a large amount of communication, since the object will either be replicated or stored on a single processor. An application programmer who uses this job queue object (which may be available in a library of standard objects) has to be well aware of the performance implications. One solution is to use one job queue object per processor. The application programmer, however, has to modify the application to handle multiple job queue objects. What the programmer really prefers is a single object that contains a job queue per processor, so that each processor can efficiently access its local job queue. Then, the application programmer only has to replace the implementation of the original job queue object with the new distributed job queue, and does not have to change the code of the application.

Another important limitation of the Orca shared object model is that processors can perform atomic operations on a *single* object only. Atomicity makes it easy to reason about parallel applications, because it guarantees that other processors cannot see intermediate states of the data. Take, for example, the job queue object. The dequeue operation should check whether there is an entry at the head of the queue, and if so, remove this entry from the queue. Operation atomicity guarantees that no other dequeue operation can steal the head entry from the queue after the first operation checked the head of the queue but before removing it. Thus, the Orca system automatically performs mutual exclusion synchronization, which facilitates parallel programming. For operations on multiple objects, however, the shared data object model does not provide such atomicity. Reasoning about the correctness of applications that use multiple objects without shared atomic operations can be complex.

These two limitations (no partitioning of the object's state and atomic operations on only a single object) of the shared data object model severely limit its expressiveness. The restrictions were applied because the designers of Orca felt they were necessary to allow a simple and efficient implementation [16]. In this dissertation, we describe two extensions to the shared data object model that provide efficient solutions for these two limitations. The problem of data partitioning is handled by *nested objects*, an extension in which the programmer can define *subobjects* within an object that are treated differently from normal data inside an object. The runtime system can place these subobjects on different processors, while still providing atomicity for all operations on the entire object. Another Orca extension, partitioned objects, only allows the partioning of regular data structures within a single object, whereas the nested object model provides the programmer with the means to determine the partitioning using subobjects [23].

The second extension, *atomic functions*, can be used to perform atomic operations on multiple objects. No other operations on these objects can intervene with the operations in the atomic functions. Therefore, the programmer can safely assume that the state of an object after the first operation is the same state that will be seen by a second operation on the same object, if these two operations are inside an atomic function.

Adding these extensions to Orca, however, should not have a negative impact on the performance of normal shared data objects. We have designed an execution model, called *collective computation*, that allows these extensions to be integrated efficiently with normal shared data objects. To keep the design of the system simple, a layered approach is used, which will be presented in the next section.

1.4 Collective computation

In this section, we present the collective computation execution model, which is used to implement the extended shared data object model.

A collective computation is the execution of the same function on an arbitrary *set* of processors, where each processor performs the same statements, with the restriction that operations on objects are only performed if the object is locally available (like the single-program, multiple-data paradigm). A simple example of collective computation is an update on a replicated object. All processors that have a copy of the object receive a message that contains the input arguments and perform the update operation. Since all processors perform the same operation on the same state of the object, all replicas will have the same value after the operation. The current Orca implementations use this method to update replicated objects.

The main benefit of collective computation is that executing the same function on a set of processors allows us to make distributed decisions about the optimal communication pattern. Since all processors perform the same statements, each processor can determine which data other processors need in order to continue their execution of the collective computation function. This implies that a processor never has to ask (i.e., send a data request message) for a certain result, but instead all processors that have the result will decide, in a distributed fashion, which processor is going to send the result. Since all processors have complete knowledge of data dependencies, this distributed decision does not require communication. Furthermore, data can be collected in a single message, which saves the overhead of sending multiple messages.

A drawback of collective computation is that the workload is increased on the processors that participate in the collective computation, since they all execute the same function. This restricts the amount of computation that can be performed within a collective computation.

Performing the same function on a set of processors and exploiting this to optimize communication and perform operations in parallel is not a new idea, since this is the most common approach for data-parallel applications. New in our approach, however, is that we apply collective computation in the context of object-based parallel programming languages, rather than the implementation of data-parallel languages.

To perform the communication between processors that participate in a collective computation, a specialized communication library was designed. This communication library exploits the property of collective computation that all processors perform the same function to perform low-level optimizations on the communication pattern. These optimizations include: reducing the number of acknowledgment messages on unreliable networks, efficient handling of incoming messages, and reducing the number of messages by applying message aggregation.

1.5 Runtime support system

To implement the extended shared data object model presented in this thesis in a modular and efficient manner, a layered approach has been taken. First, the requirements shared by runtime support for normal Orca objects and for the two extensions were determined. These shared requirements are handled in a *generic* runtime system. All specific implementation details (to implement shared objects, nested objects, or atomic functions) are handled in specific runtime support systems, which are implemented on top of this generic runtime system.

This approach has several benefits. First, it avoids extensive code duplication, since most generic functionality is only present in the generic runtime system instead of each of the specific runtime systems. Second, the generic runtime system is designed such that multiple specific runtime systems can be used for a single application. This allows shared data objects, nested objects, and atomic functions to be used in a single application. The generic runtime system interface is designed such that each specific runtime system is unaware of which other specific runtime systems are also used in the application. This makes it possible to extend the system with other models.

A drawback of a layered approach is that the overhead introduced by each layer may harm the performance of the system. Most of the overhead caused by layering, however, is due to interface mismatches between the layers. To reduce software complexity, lower layers hide information from higher layers, but if a layer hides too much, additional overhead can be introduced. An example is an interface for a message passing layer that requires the higher layer to pass it a pointer to the message data. If the message passing layer has to add a header to the message data, it may need to copy the message data into a new message buffer that contains space for the header and the data. Better solutions would involve some interaction between the two layers, so that the message data can be copied immediately to the right place. During the design of our layered runtime support system, we tried to avoid such mismatches.

The main abstraction provided by the generic runtime system is the *weaver*. Weavers are an implementation of the collective computation execution model. In addition, weavers provide support to synchronize operations on multiple objects. Weavers are used within the generic layer as well by the model specific runtime systems to implement runtime support for processes and objects.

The other abstractions provided by the generic runtime system are data types, marshaling, processes, and objects. To handle the specific implementation requirements for each model extension, the generic runtime system associates a specific runtime system with each type of object. This associated runtime system controls all operations on an object. Each runtime system can invoke operations on objects (even objects that are not associated with this runtime system) by calling the operation invocation functions in the generic runtime system. This way, the generic runtime system provides the glue between the specific runtime systems, while the implementation of each specific runtime system is unaware of the other specific runtime systems.

1.6 Contributions

This dissertation makes the following contributions:

- It extends the shared data object model so that the programmer can partition the state of a single object. This extension is called *nested objects* (Chapter 2).

- It extends atomic operations so that a single function can perform multiple operations on a set of objects atomically. This extension is called *atomic functions* (Chapter 2).

- It extends the *collective computation* model to the object-based programming environment. This model exploits the global knowledge that results from executing the same function on a group of processors to determine computation distribution and to optimize communication patterns (Chapter 3).

- It presents the design and implementation of an efficient communication library that exploits the collective computation model to optimize the communication pattern between processors involved (Chapter 5).
- It describes an extendible runtime support system for object-based parallel programming languages. This runtime support system is structured in two layers: a generic layer that captures most of the common functionality and a model-specific layer in which each model has its own specific runtime support. The generic runtime system integrates these model specific runtime systems (Chapter 6).
- It gives an implementation overview of the extended shared data object model and a performance evaluation of the system (Chapters 7-9).

The problem of synchronizing and executing operations on multiple replicated and partitioned objects is not specific for the extensions we propose to Orca, but applies to all parallel programming languages based on shared objects. The collective computation model can be used to implement efficient runtime support for such extensions. The thesis of this dissertation therefore is:

> Distributed shared memory systems lack high-level support for atomic operations on multiple objects and for partitioned objects. It is possible to implement these extensions efficiently by using collective computation. The collective computation model allows us to integrate these extensions with normal Orca objects without degrading the performance of applications that use normal Orca objects. Applications are easier to program using the extensions and achieve good performance.

1.7 Outline of the thesis

Chapter 2 gives a detailed description of the shared data object model and the two extensions described in this thesis, nested objects and atomic functions. In Chapter 3, the collective computation execution model is presented, and an overview is given of the system implementation. From then on, a bottom-up description of the complete system is given. First, in Chapter 4, Panda is described, which is a virtual machine that is used to implement multiple portable runtime systems for parallel programming. Then, Chapter 5 describes a new communication library that optimizes communication patterns within collective computations. Chapter 6 presents the generic runtime system. In Chapters 7-9, the specific runtime systems for shared data objects, atomic functions, and nested objects are presented, respectively. In Chapter 10 we draw the conclusions from this dissertation.

Chapter 2

Extensions to the Shared Data Object Model

Orca is an object-based distributed shared memory system that is designed for writing portable and efficient parallel programs [8, 16]. Orca hides the communication substrate from the programmer by providing an abstract communication model based on *shared data objects*. Processors logically share objects, even when there is no physical shared memory present in the system.

The main idea behind shared objects is to encapsulate shared data. The data can only be accessed and manipulated through *operations* on an abstract data type. An object may contain any number of internal variables of arbitrarily complex data types. *Processes* communicate with each other by performing operations on shared objects. The code for an operation may be arbitrarily complex. The model, however, defines that each operation on an object is *atomic*. It is not possible for one operation to find an object in an intermediate state of another operation. Conceptually, this means that each object behaves like a monitor with respect to its internal state.

The original shared data object model only supports atomic operations on a single object. This design decision was made to be able to build an efficient implementation. Some applications, however, need higher-level constructs than those provided by the shared data object model. Often an Orca application programmer needs to be able to specify atomic operations on *multiple* objects. In this chapter, two extensions are described that provide atomic operations on multiple objects.

The first extension, *nested objects*, allows the programmer to combine multiple objects into one composite object. Operations on this composite object are atomic, so conceptually all objects in the composite object can only be accessed by one process at a time. In contrast to the shared data object model, all subobjects are treated separately by the runtime system. Therefore, it is possible to place subobjects on different processors, while still preserving atomicity.

The second extension, *atomic functions*, allows the programmer to specify a sequence of statements that is executed atomically on a set of objects. The sequence of statements is described in an atomic function, and all objects passed as *shared* (i.e., call-by-reference) parameter can be accessed within this function. Atomic functions are more flexible than nested objects with respect to the set of objects they can operate on, because these objects are passed as parameters and do not have to be composed into one object. Nested objects, however, do not require a different interface (i.e., the atomic function) and therefore can replace any normal shared data object without modifying the application.

The rest of this chapter describes these extensions in more detail. Section 2.1 describes the Orca shared data object model. Section 2.2 gives an evaluation of this model. Section 2.3 and Section 2.4 describe the two extensions to the shared data object model, nested objects and atomic functions, respectively. Section 2.5 focuses on implementation aspects of these extensions.

2.1 Shared data objects

Orca is a high-level programming language based on parallel processes that communicate through shared data objects. The programmer describes shared objects through abstract data types: the internal representation is invisible to the user of an object. Orca supports most of the sequential constructs of Modula-2 [125] (statements and expressions). Its data structuring facilities differ significantly, though. In addition to the above features Orca supports modules and generic program units. Finally, Orca provides a secure type system.

An Orca application writer's first decision is how to divide the work over multiple processes. Parallelism in Orca is based on tasks; the programmer explicitly creates a process on a specified processor by using a `fork` statement. When an Orca program starts execution, a process is automatically created on one of the processors (processor zero) which executes the code in *OrcaMain*. From here on, the programmer can create processes on other processors.

Next, the Orca application writer has to define some objects that can be used for the interaction between processes. Shared objects can be used as a form of shared memory or as a high-level communication medium between processes. Once an abstract data type has been defined, objects can be created by declaring variables (*instances*) of this type. A process declaring an object can share the object with its children (and possibly further descendants) by passing it as a shared parameter when forking the child processes.

Mutual exclusion is provided in Orca by the atomicity of the operations on an object. In addition to mutual exclusion the Orca model also provides condition synchronization through *guard expressions*, which are based on Dijkstra's guarded commands [52]. A guard consists of a boolean expression that must be satisfied before an operation can begin, and a sequence of statements that will be executed when the guard condition is true. Each operation may contain multiple guards. An operation blocks as long as all guards are false. A blocking operation suspends the process that invoked the operation, but allows other processes to continue invoking operations on the object. Since operations may change the state of an object, guards can become true at some point. If one or more guards are true, one true guard is selected nondeterministically and its sequence of statements is executed.

2.1.1 The traveling salesman problem

To illustrate programming in Orca, a simplified version of the traveling salesman problem (TSP) program is used as an example. This application has been described in earlier publications [16], but is also described here to explain how Orca is used and to illustrate some of the problems with the Orca model.

The TSP application finds the shortest route for a salesman who has to visit each of the cities in his territory exactly once. An algorithm that gives an optimal solution is the branch-and-bound method. Each path is considered by traversing the tree of possible solutions, and the path with the shortest length is remembered. This shortest length is used as a bound; whenever a (partial) path is considered with a length that is larger than or equal to the current best length, it cannot become a better solution, so the path does not have to be considered anymore.

First, the programmer decides that this application can be parallelized by traversing the solution tree in parallel. This parallel tree search can be expressed in Orca by using replicated-workers parallelism [4]. The main process creates a worker process on each processor. After that, the main process generates jobs by doing the first steps of a path, and then passes this partial path to an arbitrary worker process, which will search the remainder of the solution space (see Figure 2.1). The search algorithm uses an ordering heuristic (nearest city first) that causes the optimal path to be found early, thereby generating a tight bound.

Second, the programmer has to specify the objects that are used to communicate between processes. This application uses three shared objects:

- An object for the bound (the length of the shortest path found so far).
- An object used by the main process to pass jobs to the worker processes.
- An object for termination detection.

These three objects are created (declared) in the main process and passed as shared parameters to the worker processes. Each worker also requires a table with distances; since this table is never changed, it is passed by value to the workers.

Figure 2.2 gives the code for the branch-and-bound function. Starting with the partial path that the main process generated, a subtree is searched to find the best solution. The `bound` object is used to eliminate parts of the tree that cannot become the optimal solution. If a better solution is found, the worker process performs an update operation on the `bound` object. However, another process

```
process worker(bound: shared Bound; q: shared JobQueue;
        WorkersActive: shared IntObject; distance: DistTab);
   job: JobType;
begin
   while q$GetJob(job) do   # Get the next job
      # Invoke a sequential TSP routine
      tsp(MaxHops, job.len, job.path, bound, distance);
   od;
   WorkersActive$dec();  # Used for termination detection
end;

process OrcaMain();
   bound: Bound;        # Declare the three shared objects
   q: JobQueue;
   WorkersActive: IntObject;
   distance: DistTab;   # Distance table
begin
   InitDistance(distance);
   WorkersActive$assign(NCPUS());

   # Fork worker processes on all other processors
   for i in 1.. NCPUS() - 1  do
      fork worker(bound, q, WorkersActive, distance) on i;
   od;

   # Generate all jobs
   GenerateJobs(q, distance);
   q$NoMoreJobs();

   # After all jobs have been  generated, start a worker process
   # on the master
   fork worker(bound, q, WorkersActive, distance) on 0;

   # Wait until all worker processes have finished
   WorkersActive$AwaitValue(0);

   WriteLine("Shortest route: ", bound$value());
end;
```

Figure 2.1: Main and worker process for the TSP application.

```
function tsp(hops, len: integer; path: shared PathType;
            bound: shared Bound; distance: DistTab);
    city, dist, me: integer;
begin
    # Search a TSP subtree that starts with initial route "path"
    # If partial route is longer than current best full route
    # then forget about this partial route:
    if len >= bound$value() then return; fi;

    if hops = NrTowns then
        # Found a full route better than current best route.
        # Update bound, using indivisible "min" operation
        bound$min(len);
    else
        me := path[hops];

        # Try all cities that are not on the initial path,
        # in "nearest-city-first" order.
        for i in 1.. NrTowns do
            city := distance[me][i].ToCity;
            if not present(city, hops, path) then
                path[hops + 1] := city;    # Append new city to end of path
                dist := distance[me][i].dist;
                tsp(hops + 1, len + dist, path, bound, distance);
            fi;
        od;
    fi;
end;
```

Figure 2.2: Branch-and-bound function that extends a partial path.

might have found a better solution in the meantime. The *min* operation on the `bound` object guarantees that only the lowest value will be stored.

```
object implementation Bound;
   from Types import Infinity;

   x: integer;     # Local state of the object

   operation value(): integer;
   begin
      return x;
   end;

   operation min(v: integer);
   begin
      if v < x then x := v; fi;
   end;

begin
   x := Infinity;
end;
```

Figure 2.3: Bound object for the TSP application.

Figure 2.3 gives the code for the `bound` object. The object state consists of a single integer, `x`. Two operations are defined, one to update the state (*min*) and another to get the current state (*value*). This object illustrates several properties of the shared data object model. The implementation of the operations does not contain any code to synchronize access to the object's local data, since the model guarantees that all operations are executed atomically. This ensures that the *min* operation always writes the lowest value in the object state, since no other update can intervene between the check in the if statement and the assignment to `x`.

Another important aspect is that it is feasible for the compiler to determine whether an operation will only do reads or can also update the object. As we will see later, the runtime system can use this information to determine how to perform operations on an object and how to distribute objects.

The code for the job queue object is given in Figure 2.4. The object state contains a queue data structure, `Q`, in which each element is a partial path that needs to be searched. The queue data structure is an instantiation of the generic built-in graph type. A graph type is defined by specifying which fields the nodes should have (`next` and `data`), and fields that are global to the graph, including pointers to certain graph nodes (`first` and `last`). Pointers and nodes are only

```
object implementation JobQueue;
   from Types import JobType;

   type ItemName = nodename of queue;
   type queue = graph   # Queue is a singly-linked list
                  first, last: ItemName;
            nodes
                  next: ItemName;
                  data: JobType;
            end;

   done: boolean;
   Q: queue;

   operation AddJob(job: JobType);
      p: ItemName;
   begin
      p := addnode(Q);    # Allocate a new node
      Q[p].data := job; Q[p].next := NIL;
      if Q.first = NIL then Q.first := p;  # Add it to the end of the list
      else Q[Q.last].next := p; fi;
      Q.last := p;
   end;

   operation NoMoreJobs();
   begin done := true; end;

   operation GetJob(job: out JobType): boolean;
      p: ItemName;
   begin
      guard Q.first /= NIL do
         p := Q.first;   # Get first element and delete it from the list
         Q.first := Q[p].next;
         job := Q[p].data;
         deletenode(Q, p);
         return true;
      od;
      guard (Q.first = NIL) and done do return false; od;
   end;

begin
      Q.first := NIL; done := false;
end;
```

Figure 2.4: Job queue object for the TSP program.

valid within an instance of a graph type. To access a node of the graph the pointer has to be used as an index for the graph variable (e.g., `Q[p]`). Operation *AddJob* is used by the main process to add jobs to this job queue `Q`. When all jobs have been generated, the main process invokes the operation *NoMoreJobs*. This operation sets the boolean flag `done` in the object state. This boolean flag is used to avoid worker processes to terminate before all work has been generated.

Worker processes acquire jobs from the job queue with *GetJob*. This operation shows the use of guarded operations. If a job is available (`Q.first /= NIL`) it is removed from the queue and returned to the worker process. If the job queue is empty and all jobs have been generated (`done` is true) *GetJob* returns false to let the worker process know that the application is about to finish. Finally, if both conditions fail, the operation blocks. In this case, the job queue does not yet contain any elements, but new elements may still be added.

This object also shows some aspects of Orca. First, the state of the object may be arbitrarily complex: *AddJob* and *GetJob* each perform some graph manipulations atomically. Second, condition synchronization has a high level of abstraction. It allows complex conditions to be expressed easily. The implementation takes care of checking the conditions, in the worst case reevaluating the boolean expressions each time the object is modified. Also, synchronization is hidden inside the object's operation. Users of the object do not include any code for synchronization.

2.1.2 Implementation of Orca

To implement programs efficiently, the Orca runtime system takes into account how processes access objects. In our TSP example, the bound object is written only a few times (when a better path is found) and read millions of times (whenever a bound check is made with respect to the current path; see Figure 2.2). The job queue object, on the other hand, will only be written, since both *AddJob* and *GetJob* change the queue data structure[1]. Objects with a high read/write ratio, such as the bound object, can best be implemented by replicating the data. Objects with a low read/write ratio are best implemented by keeping them on one processor. In the TSP example, the job queue object will only be stored on the processor that generates the jobs.

The Orca language specifies that operations on objects are atomic. These semantics are implemented in the runtime system by using the *sequential consistency* model. For nonreplicated objects, sequential consistency can easily be achieved since all operations are serialized on the single copy. If a processor

[1]To be more precise, only the first guarded operation of *GetJob* changes the state of the object. The second guarded operation, however, will only be called a few times.

performs an operation on an object that is local, it locks the object and calls the operation. A processor that accesses a remote object has to marshal the operation and arguments and send it to the processor that has the object. This remote processor unmarshals the operation, locks the object, performs the operation, and sends the return values in a reply message to the invoking processor. When this reply arrives, the return values are unmarshaled and the process can continue.

For replicated objects, however, sequential consistency is more complex to achieve. All processors must see the write operations from each other processor in the same order, even if they involve different objects. This is necessary to provide sequential consistency. In this chapter we will present two protocols for replicated objects: write invalidate and write update.

With *write-invalidate replication* a write operation invalidates all current copies of the data. One processor is designated as the owner of an object, and this processor always has the most up-to-date value. The owner of an object may change during the execution of the program. Other processors get a copy of the data when they initiate a read operation. As long as this copy remains valid, all read operations can access this copy. Whenever a write operation has to be executed, all current copies are invalidated, so that processors will fetch the new value on the next operation. The runtime system has to guarantee that the processor that executes the write operation becomes the owner of the object so that no two write operations can execute concurrently. CRL, a library that allows the programmer to manipulate regions of shared memory, uses write-invalidate replication to keep these regions consistent [72, 73]. More advanced implementations allow multiple writers and order the updates when a processor fetches a new copy [6, 43].

With *write-update replication*, write operations are multicast to all processors that have a replica, and all replicas are updated in place by executing the operation locally. It has been proven that using a total ordering on the multicast messages guarantees sequential consistency [48]. Furthermore, it is possible to integrate this update mechanism with the access mechanism for nonreplicated objects [15, 55]. Since the function is shipped to all processors that have a replica of the object, the runtime system must have access to the operation code and the arguments. One of the main benefits of accessing shared data through high-level operations is that this information is available.

The Orca runtime system uses a write-update replication protocol and updates the copies of an object using function-shipping. This design choice was made, because for the shared object programming model the disadvantages of write-update are more than compensated by its advantages [12]. Multiple consecutive write operations by the same processor (without intervening accesses from other processors) do not occur often in Orca programs. In many cases, programmers combine such write operations in a single operation. In addition, the Orca system only replicates objects that have a high read/write ratio.

The decision to replicate an object depends on the access patterns for that object. At compile time it is in general not known which and how many processes will be forked during the execution. Since processes determine the access patterns to objects, the decision to replicate an object or not has to be taken at run time. The current Orca system uses an integrated approach. The compiler determines the access patterns of each type of process on its shared object parameters, and passes this information to the runtime system. Based on the type and number of processes forked, the runtime system computes an estimate of the access pattern of each object, and takes the appropriate decision [15]. This technique is extended with runtime statistics to alleviate wrong compile-time estimations [85].

Since the runtime system is responsible for object placement, the programmer does not have to worry about this. The performance of the application, however, strongly depends on this object placement. The original Orca system provided the following performance semantics in the reference manual [9]:

> The most important issue is how to implement objects that are shared among multiple processes on different machines. The key idea in the implementation is to replicate such shared objects in the local memories of these machines. As a result, operations that only *read* the object will usually be executed without doing physical interprocessor communication. Operations that *write* (i.e., change) a shared object usually will involve IPC. This is not to say an implementation of Orca guarantees this property for all objects, but it is the general model programmers should keep in mind.

As can be seen, the reference manual does not give any guarantees which objects will be replicated or not. The only thing that the performance semantics imply is that reads are never more expensive than writes.

2.2 Evaluation of the shared data object model

Orca is easy to use for small programs, such as the TSP example. A large number of such small programs have been written, such as all-pairs shortest path, alpha-beta search, and successive over-relaxation. Larger applications, however, show some limitations of the model. Therefore, a thorough evaluation of the Orca parallel programming system was initiated by Greg Wilson in 1994, based on a test suite called the *Cowichan problem set* [122, 123]. In this section, we will summarize the results of this evaluation, and use these results to motivate the extensions that we have proposed (i.e., atomic functions and nested objects).

The Cowichan problems is a suite with medium-size, realistic applications that can be used to evaluate parallel programming systems. The focus of the Cowichan problems is not to evaluate the performance of the system, but to determine the *usability*. Since the programs have a modest size (at most a few thousand lines of code), it is feasible to write them entirely in a new language.

The Cowichan problem set covers a wide spectrum of application domains and parallel programming idioms. The suite contains *numerical* applications, which typically manipulate large distributed arrays and are often easy to express in data parallel languages such as High-Performance Fortran [81, 89]. In addition, the suite contains *symbolic* applications, which deal with dynamic data structures and often exploit task parallelism.

The original paper [122] describes seven applications, namely dynamic programming (optimal matrix multiplication ordering), the Turing ring, skyline matrix solver, image thinning, polygon overlay, crossword construction, and active chart parsing (see Table 2.1). Implementations of these applications existed or were written on a variety of architectures, including shared memory, message passing, dataflow, and data-parallel systems. The paper gives a detailed description of the algorithms, and some preliminary observations are made with respect to parallelization. As can be seen in Table 2.1, the applications cover a large number of the problems that occur in writing parallel programs.

Application	**Parallelization problems**
Dynamic programming	- complex data movement
The Turing ring	- spatial decomposition - dynamic load balancing
Skyline matrix solver	- data replication - irregular data representation - communication intensive
Image thinning	- preserve sequential data access ordering - communication intensive
Polygon overlay	- I/O management - load balancing
Crossword construction	- dynamically allocate work - dynamically interrupt work - search overhead
Active chart parser	- fine-grained - termination detection

Table 2.1: The Cowichan problem set.

Starting in 1994, a group of students implemented this problem set in Orca [35, 36, 38, 83, 97, 117, 123]. The students had experience with systems programming (in C), operating systems, and networking, but had little or no experience with parallel programming. Each student was assigned one of the applications. The students first wrote a sequential ANSI C program for the application. After that, sequential and parallel versions were written in Orca. The C version was used to gain experience with the application and served as a baseline for the performance evaluation of the Orca versions. Each student spent three to seven months on this project. During this period, they were supervised by Greg Wilson.

The aim of the project was to assess the usability of Orca rather than its performance. However, performance is in general the most important reason to parallelize a given application. Therefore, the students had to make some effort to tune the performance of their parallel program. They had to obtain reasonable speedups or explain why these were absent.

This experiment showed that Orca had achieved its main goal: a parallel programming language that is easy to use. The concept of a shared data object turned out to be easy to learn and use. Creating processes and using advanced data structures (e.g., graph types) was straightforward. A secure type system proved to be a useful property to implement parallel programs correctly. All students managed to write a working parallel version.

Apart from some parallel efficiency problems, several problems related to the Orca programming model were identified. The main disadvantage is that the model lacks a way of partitioning objects. Numerical applications (e.g., image thinning), for example, typically use large multidimensional arrays. These arrays are then partitioned among the processors, and each processor updates the part of the grid that is assigned to it and only reads data stored on other processors (*owner-computes rule*). This model is used in many data parallel programming languages, such as High Performance Fortran.

Originally, Orca did not support objects with a partitioned state. Either the whole object state is on one processor, or it is replicated on all processors that can access the object. Programmers had to partition the state manually and store each partition on a different processor as different objects. Objects then were only used to communicate updates between processors. A new extension to the shared data object model, called *partitioned shared objects* [21–23], was designed to overcome these problems.

With partitioned shared objects, the implementor of an object describes a multidimensional array data structure and the operations that can be applied to the object. The programmer describes which variables each element of the array contains. Two types of operations are provided, sequential and partitioned. *Sequential* operations are similar to normal Orca operations. They operate on the full object, and all elements are accessed in the order specified by the programmer.

Parallel operations, however, logically operate in parallel on all elements. Each parallel operation updates a single element; all other elements may only be read. The runtime system resolves the data dependencies when elements must be read to compute the value of the element to be updated. The runtime system is guided by the partitioning and distribution of the object that is specified by the user.

Partitioned shared objects are a suitable solution for numerical applications that use regular grid data structures. Other applications, however, require one to partition the state of irregular data structures in one object (e.g., active chart parsing and the Turing ring application). In the TSP application described in Section 2.1.1, for example, the job queue object contains one queue. Since all operations change the object (both *AddJob* and *GetJob* change the object state), it will not be replicated, but be placed on the processor on which the main process runs. If a large number of processors is used and the amount of work per job is small, this processor can become a bottleneck. It cannot generate jobs fast enough, because it is too occupied with handling job requests. What the programmer would like to express is a semi-queue (i.e., a queue that does not strictly provide FIFO ordering) that is distributed over multiple processors. The semi-queue allows the master process to push jobs to sub-queues on different processors, thereby relieving the master processor from performing the dequeue operations as well.

One solution would be to use multiple job queues and multiple job generator processes. This requires rewriting a large part of the application, since it is not possible to put multiple queues in one job queue object such that these queues are on different processors. Instead, the programmer is forced to use multiple job queue objects. Worker processes can be matched with a job queue object when they are forked, but this match cannot be changed at run time, so it may lead to load imbalance.

The solution presented in this thesis is based on *nested objects*. With nested objects, an object may contain other objects (called *subobjects*) that are treated separately by the runtime system. Therefore, the state of each subobject may be placed on a different processor. This implies that the state of the *root object* becomes partitioned. Nested objects are presented in more detail in Section 2.3.

Another problem with the Orca programming model is that operations are always applied to a single object. In many situations it is not necessary to do atomic operations on multiple objects, but occasionally such functionality is useful (e.g., the termination detection code in the Turing ring application). In the TSP application, for example, the job queue object now contains a boolean flag that triggers a termination condition if the queue is empty. For larger applications, however, it may be inconvenient or not even possible to combine this kind of state information in one object. If the termination condition consists of checking whether an active workers object has value zero *and* the job queue is empty, the condition has to be checked atomically, otherwise incorrect behavior can occur.

A solution for this problem presented in this thesis is called *atomic functions*.

An atomic function is a function that can operate atomically on multiple shared objects that are passed as parameters. During the execution of an atomic function, no other operations can logically be applied to the shared object parameters of that function. Atomic functions are presented in more detail in Section 2.4.

The conclusion of this evaluation is that Orca achieves its primary goals of being easy to learn and use, but that the Orca model also has some important shortcomings. An important aspect of this thesis is to address these problems and enhance the expressiveness of the shared data object model.

2.3 Nested objects

The nested object model is based on the idea that the implementor of an object can distribute the state of this object among multiple subobjects. These subobjects can be placed on different processors, may be replicated, or may even be nested objects themselves. The key idea is that the implementation should be able to allow operations that only read replicated objects and read or update *local* subobjects to be executed without communicating with other processors.

```
object implementation counter;
    import IntObject;
    c: array[integer 1..NCPUS()] of IntObject;

    operation inc();
    begin
        c[MYCPU()]$inc();  # increment local counter
    end;

    operation value(): integer;
        sum: integer;
    begin
        sum := 0;
        # Add all counters in one atomic operation
        for i in 1..NCPUS() do
            sum +:= c[i]$value();
        od;
        return sum;
    end;
end;
```

Figure 2.5: A nested object using multiple local counters.

An example of a nested object is given in Figure 2.5, where all processors share a counter that is incremented frequently, while the current total value is needed only occasionally[2]. In the normal shared data object model, it is not possible to have the write (increment) operation to be cheaper than the read (see Section 2.1.2). In the nested object model, however, we can use a local counter object for each processor (i.e., the `IntObject` elements of array `C`), so that the increments can be executed locally. Reading the current counter value is an atomic operation that reads all counters and adds them. Since operations of a nested object are executed atomically, all subobjects can be accessed without explicit synchronization. The fact that the counter object is implemented as a nested object is hidden from the user of the object. The user only sees a shared counter that is cheap to write and expensive to read.

Since the user of a nested object does not have to know that the object is a nested object (for example, because it is in a library and the source code is not available), operations on nested objects should have the same semantics as normal objects. In particular, they should provide sequential consistency. For example, if an operation accesses multiple subobjects distributed over different processors, no other operations may interfere. The runtime system has to achieve this by synchronizing accesses to subobjects, even for operations that do not require communication.

To enable an efficient implementation of this synchronization, the nested object model imposes a restriction on the relationship between the replication of a *parent* object and the replication of its subobjects. The rule is that the parent object should be at least replicated on all processors that contain one or more subobjects. In other words, the degree of replication of an object must be less than or equal to the degree of replication of its parent object. This guarantees that if an operation accesses multiple subobjects, it is sufficient to deny access to the parent object for all other operations. The implementation of synchronization is explained in more detail in Chapter 9.

Figure 2.6 shows the implementation of the partitioned job queue that we needed for the traveling salesman problem. Figure 2.7 shows a possible configuration of this object with three subobjects on three processors. Since each processor contains a queue, the root object has to be replicated on all processors. Each processor has its local copy of the root object, containing array `Q` with references to the subobjects. The *AddJob* and *GetJob* operations only update the local subobject; no changes are made to array `Q`. Because the root object is not changed, and only one subobject is accessed, no global synchronization is required.

[2] In this example, NCPUS( ) is a function that returns the number of processors that are assigned to this Orca program execution. MYCPU( ) returns the processor identifier of the invoking processor. Within an object, this is defined as the processor that issued the operation. Object placement is specified explicitly when the object is created.

```
object implementation MultQueue;
        from Types import JobType, NR_QUEUES;
        import JobQueue;

        Q: array [integer 1 .. NR_QUEUES] of JobQueue;

        operation AddJob(job: JobType; queue: integer);
        begin
                Q[queue]$AddJob(job);
        end;

        operation GetJob(job: out JobType; queue: integer): boolean;
        begin
                return Q[queue]$GetJob(job);
        end;

begin end;
```

Figure 2.6: Capturing multiple job queues in a single nested object.

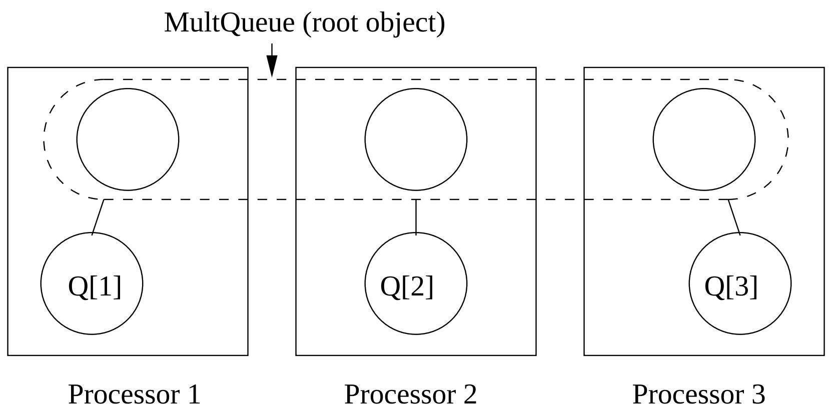


Figure 2.7: A `MultQueue` object with three subobjects.

The nested object model provides the following performance characteristics:

- Operations that only read or update local (sub)objects are cheap, since they do not require communication.
- Operations that only read or update objects stored on a remote processor are relatively cheap, since they only require point-to-point communication to that specific processor.
- Operations that update replicated objects or access multiple objects stored on different processors involve synchronization, and are therefore more expensive.

These new performance characteristics are closely related to the actual (sub)object placement and replication. Since the focus of this thesis is to show how to implement this model efficiently, the programmer of an object is given explicit control over object placement (see Chapter 9). We do not show these placement control statements in our sample code.

2.4 Atomic functions

A second extension to the Orca model is the atomic function. An atomic function is a sequence of statements that is executed atomically on all objects that are passed as shared (call-by-reference) parameters. An atomic function is declared by adding the keyword `atomic` to the function declaration. Conceptually, no other operations on these objects can be performed while the atomic function is executed. The major difference between a nested object (which also provides atomicity of its operations on all its subobjects) and an atomic function is that the atomic function is more dynamic: only the (run-time selected) objects that are passed as parameter to the atomic function are synchronized; other objects and other operations operate as in normal Orca.

```
atomic function copy(x, y: shared IntObject);
   r: integer;
begin
   r := x$Value();
   y$Assign(r);
end;
```

Figure 2.8: Atomic function that copies the integer value of object `x` to object `y`.

An example atomic function is given in Figure 2.8. The value of integer object `x` is read into variable `r`, and subsequently written in object `y`. During the execution of the atomic function, object `x` is guaranteed not to change, so after the assignment on object `y`, both integer objects contain the same value. Note that there is a data dependency between the read operation on object `x` and the write operation on object `y`, since they both use variable `r`.

Figure 2.9 gives an implementation of an atomic function that determines the termination condition using a job queue and an active workers counter, as described in Section 2.2. The atomic function *Terminate* checks whether the termination condition is fulfilled, i.e., whether the job queue is empty and the active workers counter is zero. Other processes cannot access these objects while the atomic function is executing. To express synchronization, guarded statements can be used inside an atomic function.

First, a check is made whether the queue contains work. This can occur if another processor adds work to the queue after the worker process tried to get a job from the empty queue. If the queue contains work, the atomic function returns `false` and the worker will subsequently dequeue the job. If the job queue is empty and no other processors are active, the process may terminate, so `true` is returned. If both guard conditions are false, the invoking process is blocked until either of the two objects (the job queue or the active workers counter) is changed. This way, no polling is required on the two objects.

Atomic functions are somewhat similar to atomic transactions [56]. Atomic transactions, however, allow atomic actions on arbitrary sets of data. Deadlock avoidance, atomic commit, concurrency control, and fault tolerance are complicated and expensive to implement if arbitrary data can be accessed in a transaction. Moreover, for parallel programming, such a general mechanism is hardly ever required. Atomic functions differ from atomic transactions in that the objects to be accessed must be known when the function is started. This makes synchronization (which involves concurrency control and deadlock avoidance) much easier. Also, atomic functions do not deal with fault tolerance. The implementation details for synchronization are described in more detail in Chapter 8.

Algorithms that can conveniently be written with atomic functions include global termination detection, load balancing algorithms, weak consistency, and global state snapshots. The atomic function *Terminate* on the job queue and active workers counter is an example of global termination detection.

Load balancing strategies benefit from atomic functions, because they allow the programmer to poll a number of objects, and based on these results get work from the best candidate. Since the operations are performed atomically, it is guaranteed that the object still contains the work it returned at the poll. This relieves the programmer from taking into account concurrent accesses to these objects. Furthermore, the objects on which the atomic function operates can be determined

```
atomic function Terminate(q: shared JobQueue;
                    workers: shared IntObject): boolean;
begin
    guard not q$Empty() do return false; od;
    guard q$Empty() and workers$value() = 0 do return true; od;
end;

process Worker(q: shared JobQueue; workers:  shared IntObject);
    job: JobType;
begin
    do
        workers$inc();
        while q$GetJob(job) do
            # handle job
        od;
        workers$dec();
    while not Terminate(q, workers);
end:
```

Figure 2.9: Global termination detection using an atomic function.

dynamically.

Weak consistency can be implemented by replacing a single object with multiple objects, one for each process. A watchdog process then performs an atomic function to gather all information, perform a global reduction, and assign the new value to all objects. The time between these atomic function invocations determines how often communication will take place, instead of at every update. For example, the computation of a minimum value in a branch and bound algorithm can be implemented this way.

Finally, atomic functions allow a process to gather the state of a set of objects at specific points in time, without disturbing the normal execution of the program. This can be useful for debugging and for implementing fault tolerance at the Orca level.

2.5 Implementation issues

Both nested objects and atomic functions have to deal with operations on multiple objects while still preserving atomicity. Therefore, implementations of compiler and runtime support for these two extensions can share algorithms to deal with these operations.

In these algorithms, a number of implementation issues have to be addressed. First, the runtime system has to implement an efficient way to achieve the synchronization required to provide atomic operations. Second, the system has to deal with arbitrary sets of objects that are arbitrarily replicated over the processors. (The nested object model does impose some restrictions on the replication of subobjects, but this is used only for synchronization.) Finally, the runtime system has to resolve data dependencies between operations.

Nested objects and atomic functions provide different models for accessing multiple objects. With atomic functions, the set of objects that the function will be applied to can be directly determined when the function is invoked, since these objects are passed as shared arguments. With nested objects, however, the set of objects is determined by the operation, since most operations will only access a subset of the objects in the hierarchical tree. The nested object model, however, restricts the set of objects that require synchronization to be subobjects of one shared ancestor object. This gives the implementation the opportunity to synchronize a set of objects by locking this common ancestor (see Chapter 9).

Objects can be arbitrarily distributed over the processors. Some objects may be single-copy objects, so only one processor has its state. Others may be partially replicated over a subset of the processors, or fully replicated. Since the replication of objects may not be known at compile-time, the runtime system has to deal with this.

After synchronization has taken place, some function will be executed on the synchronized objects. This function can do arbitrary operations on these objects, and can use results from operations to compute the arguments of other operations. For example, the *copy* atomic function given in Figure 2.8 on page 29 uses the result of the value operation on object `x` as the parameter to the assignment on object `y`. This implies that if object `y` has an instance on a processor that does not have a copy of object `x`, the result of the operation (i.e., local variable `r`) has to be forwarded. The runtime system has to make sure that these data dependencies are resolved.

One of the goals of the runtime system proposed in this thesis is to integrate the original shared data object model with nested objects and atomic functions. To facilitate this integration, the runtime system will be structured as an extensible kernel, on top of which specific runtime modules can be added. The runtime system kernel has to deal with demultiplexing invocations and messages between these specific runtime modules. The structure of the runtime system will be described in Chapter 6.

The issue of synchronization for the specific runtime system implementations will be discussed in Chapter 8 and Chapter 9. The next chapter describes a generic execution model that can be used after all objects are synchronized.

Summary

In this chapter, we described the shared object model and its implemented in the Orca parallel programming language. Orca has been used for a wide range of applications. For small applications, the language is easy to learn and use. Larger applications, however, are harder to implement due to some restrictions in the shared object model. The most important restrictions are that only operations on single objects are atomic and that the state of an object cannot be distributed over different processors.

We have presented two extensions that solve some of these restrictions. The first extension, nested objects, allows the programmer to partition the state of a single object by specifying subobjects. These subobjects can be distributed over the processors. The second extension, atomic functions, provides a interface to perform operations on multiple objects atomically.

These extensions share a number of implementation properties with each other and with the implementation of normal shared objects. First, the implementation has to be able to deal with replicated objects, since object replication is necessary to achieve good performance for some applications. Second, mutual exclusion and condition synchronization have to be integrated. Finally, data dependencies between operations need to be resolved efficiently.

Chapter 3

Collective Computation

The previous chapter described two new extensions to the shared data object model: atomic functions and nested objects. This chapter will describe an efficient execution model, *collective computation*, that allows implementation of the extended shared data object model (i.e., the normal shared data object model combined with atomic functions and nested objects). The runtime support needed for the extensions share several requirements. First, the system has to handle two types of synchronization: mutual exclusion and condition synchronization. Second, the runtime system has to deal with objects that are replicated on some (but not necessarily all) processors, since replicated objects are important to achieve good application performance.

A simple execution model to perform an atomic function is to let the invoking processor perform all the operations. This requires a protocol to lock all objects in advance (e.g., a synchronization broadcast message). The processor then invokes operations as in the Orca runtime system. Whenever the atomic function blocks or finishes, the locks have to be released (another synchronization message).

This simple execution model has several problems. First, all operations on objects are executed in a sequential order, even if they are on different processors and could run in parallel. Second, the processor that invokes the operations always has to send all parameters to perform an operation, even if some of those parameters were already computed on the processor that has to perform the operation, because they were the result of an earlier operation on an object. Finally, it is not clear how this execution model can be applied to nested objects, since operations can not only access multiple subobjects, but can also change the state of a nested object, which may be replicated.

In this chapter, we will present an alternative execution model, called collective computation. First, we will first discuss some of the techniques used for implementing parallel programming languages that influenced the development of collective computation (see Figure 3.1). These techniques cover different aspects of parallel programming and focus on different problems to solve. For those techniques, we will look into the following issues:

- how the technique handles the trade-off between data locality versus load balancing;
- whether the technique can handle data replication;

- whether knowledge of the communication pattern can be employed to improve the performance; and
- whether the technique is flexible enough to be used in an object-based parallel programming system.

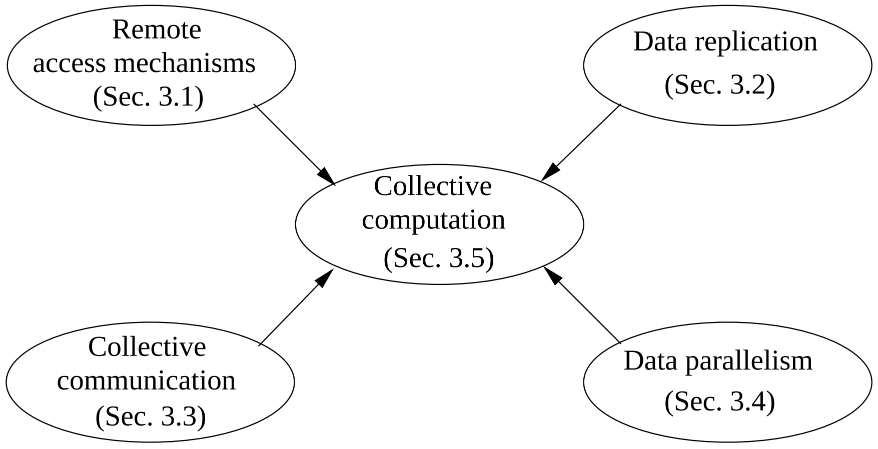

Remote access mechanisms (Sec. 3.1)
Data replication (Sec. 3.2)
Collective computation (Sec. 3.5)
Collective communication (Sec. 3.3)
Data parallelism (Sec. 3.4)

Figure 3.1: Techniques that influenced collective computation.

In Section 3.1, remote access mechanisms are discussed in systems where only a single copy of each object is available. Section 3.2 describes techniques that allow multiple copies of each object to be available on different processors at the same time. Section 3.3 describes collective communication, in which all participating processors execute the same communication operation. Finally, in Section 3.4, we discuss data parallelism, in which all processors perform the same computations, but on different parts of the data.

After presenting the different techniques that influenced collective computation, Section 3.5 will describe collective computation and will compare collective computation to the other techniques. Finally, Section 3.6 will present an overview of a runtime support system implementation for the extended shared data object model that is based on collective computation.

3.1 Remote access mechanisms

If a process needs to execute an operation on an object[1] that is stored on a remote processor, communication is required to get both the operation arguments and the

[1]We will use the terms objects and invocations on objects, even though most techniques have originally been applied in systems that are not object based.

object data on the same processor. Since there are only two parties (the invoking processor and the processor that contains the object), two mechanisms are available: move the operation to the object or move the object to the operation. In this section, these two mechanisms will be described and their benefits and drawbacks will be discussed.

Implementations that move the operation to the object must decide how much of the execution is moved to the other processor. Obvious choices are to move only the operation (*remote procedure call*), or to move the complete thread of control to the other processor (*thread migration*).

Remote procedure call [33] is an extension of the procedure call mechanism that provides for transfer of control and data across the network. When a remote procedure is invoked, the calling process is suspended and the parameters are passed to the other processor. The remote processor receives this invocation and executes it locally. Results are then passed back to the invoking processor and the suspended process is resumed. In an object-based system, procedures correspond to operations on objects.

Thread migration [45, 49] is the other extreme. Whereas remote procedure calls transfer the minimum amount of state to perform the operation, thread migration moves the complete execution state of the invoking thread. After a thread is migrated to the remote processor, execution continues there, even after the operation that caused the migration has completed. The original processor only executes this thread again if this thread happens to migrate back to the original processor.

In other implementations, thread (or process) migration is not used as a mechanism to access remote data, but as a mechanism to perform dynamic load balancing [53]. Threads are moved from heavily occupied processors to less occupied processors, so that the total computation can be executed faster. The main problem with thread migration, however, is that a thread is expensive to move. First, threads contain a large amount of state, either on the stack or on the heap (typical stack sizes of threads range from below 1 KB up to more than 100 KB [58]). Second, all pointers used by this thread have to be translated to reflect the new memory layout on the destination processor.

Thread migration clearly shows the tradeoff between *load balancing* and *data locality*. If thread migration is used as a remote access mechanism, all further accesses to data on the same processor are handled locally. However, this migration can put too much load on the remote processor, thereby causing load imbalance. On the other hand, if thread migration is used to do load balancing, all accesses to remote data must be handled with another remote access mechanism. Since in most cases remote accesses take less time than the whole computation, the load is more evenly distributed over the processors. This requires communication to access the remote data, even if the same remote processor was accessed before.

With remote procedure calls, the computation always stays at the original processor, except during the operation. All remote accesses to data are treated separately, and no data locality is exploited. One advantage of remote procedure calls, however, is that the extra load on the remote processor is small. An invocation is executed on the remote processor, and the results are sent back to the invoker. No state remains at the remote processor apart from the updated object. Therefore, it is easier to do static load balancing, thereby reducing the need for dynamic load balancing in a large number of applications (e.g., producer-consumer algorithms).

Hsieh [65, 66] proposes a mechanism that gives the benefits of both remote procedure calls and thread migration. With *computation migration,* only part of the thread's stack is moved to the remote processor. The programmer specifies the part of the stack that has to be migrated by using annotations. Computation migration allows both the benefits of load balancing and data locality. If load balancing is a major concern, the programmer specifies that only the activation record that performs the remote invocation is migrated, which is equivalent to a remote procedure call. To exploit data locality, the stack is split at the activation record of a function higher up in the call chain. The amount of thread state and computation that is moved to the remote processor is still small, and all accesses to objects on the same processor can be performed locally. After the function where the stack was split finishes its computation (including all operations on objects), the thread continues on the original processor. An extended version of CRL, called MCRL, uses computation migration.

Olden [105] provides a similar mechanism, called *thread splitting*, in which only the current activation record is sent that is necessary for the current procedure to complete execution[2]. To facilitate the implementation, Olden disallows the use of global variables and also does not allow a thread to take the address of stack-allocated objects.

Figure 3.2 depicts communication pattern when processor P_0 makes n consecutive accesses to each of m data items on processors P_1 through P_m, respectively. In the case of remote procedure calls, each access must be handled by a separate invocation. Therefore, $2nm$ messages are needed. When using computation migration, however, the first access of remote data will move part of the invoking thread to that processor. Therefore, only a total of m messages is needed, although the message size may be larger than for the remote procedure calls.

Instead of moving the computation to the data, it is also possible to move the data to the computation. Emerald [74] is an example that provides both remote procedure calls and data migration. Emerald is an object-based language and system designed for the construction of distributed programs. It provides a

[2]The implementation of computation migration described in Hsieh's dissertation also provides only migration of single activation records.

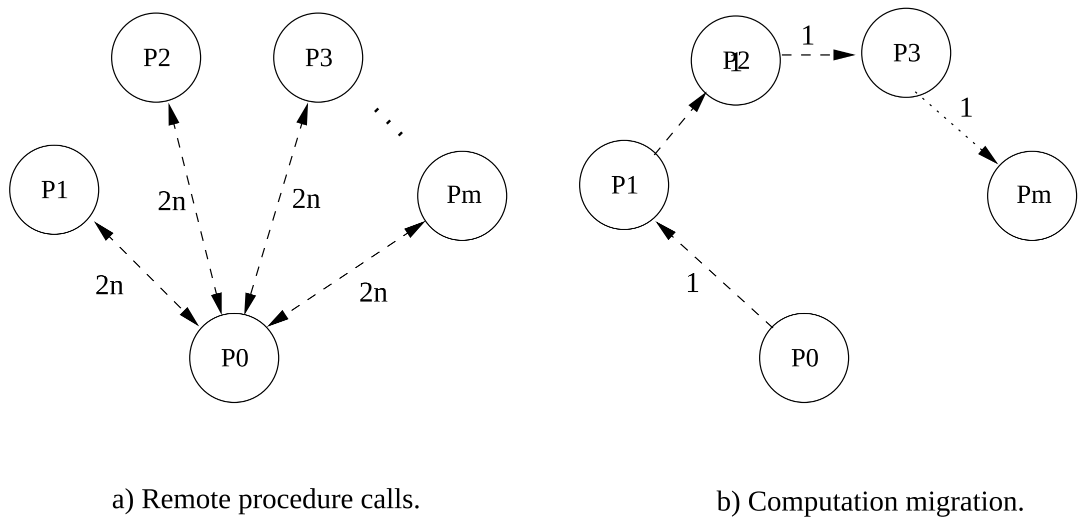


Figure 3.2: Message patterns for a) remote procedure calls and b) computation migration (taken from [65]).

single object model: both small, local data objects and large, active objects are defined by the same object definition mechanism. Each object contains data (including references to other objects), a set of operations, and an optional process (to describe active objects). Invocations on objects involve moving the invocation frame to the remote processor. The programmer can specify whether objects passed as parameter to an operation are moved to the destination processor or not. Furthermore, the programmer has several primitives to move objects to increase data locality.

In conclusion, both computation migration and data migration try to provide the programmer with the means to trade data locality against load balancing. In computation migration, the programmer specifies which part of the computation should be executed on a remote processor. With data migration, the programmer specifies which remote data should be fetched to do the computation locally. Both methods, however, do not deal with data *replication*. In the next section, we will describe how data migration can be extended to allow replication.

3.2 Data replication

With data migration, the remote object is moved to the processor that issued the invocation, after which the invocation can be executed locally. Instead of moving the object, it is also possible to *copy* the object. After the first invocation on an object, this copy of the object is available locally, so further read operations do not need to communicate. The goal of data replication is to exploit this type of temporal locality. When the object is updated, however, all remote copies need to be invalidated or updated to guarantee coherence (for example sequential consistency).

Managing the coherence of the replicated data can be performed in hardware or in software. Shared-memory multiprocessors such as the DASH [87] and the Alewife [2, 46] use data replication. In these systems, the hardware detects accesses to a memory location that is not in local cache memory. The hardware then fetches the data from the remote memory that does have a consistent copy, and puts it in the local cache. Whenever a processor updates a memory location, all other copies in the other caches are invalidated. By using a directory data structure (possibly extended in software to handle overflow) to identify all processors with a copy of the data, the amount of communication to invalidate the caches is limited.

Data replication is also applied in software distributed shared memory systems, such as Munin [43] and TreadMarks [6,78]. These systems provide a global shared memory address space with page-sized granularity. The hardware memory management unit detects accesses to invalid data, after which a software control system fetches the correct data. Using the hardware to detect accesses to shared memory, however, has two problems. First, writes to an invalid page have a high overhead, since they are detected with a page fault. Second, the granularity of a virtual memory page is too coarse to handle shared data efficiently [126]. Optimizations exist that allow multiple processors to write to the same page and allow the system to send only that part of the page that is actually changed instead of the whole page.

Other distributed shared memory systems, such as Midway [25], CRL [72,73], and Shasta [113] use software techniques to detect accesses to shared data. In Midway, the compiler inserts inline code to timestamp updates on writes to shared data. In CRL, the programmer has to declare regions of shared memory, and use special library calls to obtain valid pointers to the region's data and to encapsulate sections of code that read or write a region. Finally, in Shasta the generated binary is patched such that all potential accesses to shared memory are forwarded to the runtime support system.

All systems try to avoid having to communicate for every write to a memory location. With these systems, the programmer has to use explicit synchronization primitives, such as locks and barriers. On acquiring (e.g., Midway) or releasing (e.g., TreadMarks) a synchronization variable, communication takes place between processors. These messages are also used to invalidate data copies and to update directory entries. Doing communication only at synchronization points reduces the number of messages that are needed to invalidate the remote copies. In object-based systems, such as Orca, an operation can be seen as a sequence of read and write operations on the object state, protected by a lock, so similar techniques can be used.

When data are invalidated, a new consistent copy has to be fetched when the data are accessed again. For some applications, it is clear in advance that this copy is going to be needed. Instead of waiting for the processor to access the stale data before fetching a copy, it is better to update the replicated data. DiSOM [44] and Orca apply this mechanism. In DiSOM, an object is updated when the lock that is associated with it is released. The new state of the object is sent to all processors that have a copy of the state (data shipping). With Orca, on the other hand, the update operation with its arguments is sent to all processors that have a copy of the object (function shipping). Since all processors perform the same update in the same order, data is kept consistent (see Section 2.1.2). A drawback of update protocols is that they are more expensive than invalidate protocols if one processor performs multiple consecutive writes, because all updates will cause communication to all replicas.

Different applications can have different access patterns to their shared data structures, and even within a single application different data structures may require different consistency protocols to perform optimally. Therefore, some systems provide the programmer with the possibility to select the consistency protocol for each data structure. Munin [43] and SAM [114] specify a number of predefined protocols that the programmer can select. Other systems, such as Tempest [103] and Ace [102] allow the creation of completely new, application-specific, consistency protocols.

In conclusion, data replication can increase the performance of parallel applications by exploiting temporal locality. By using a write-update protocol instead of a write-invalidate protocol, synchronization messages can also be used to transmit the data. Write-update protocols, however, suffer from overhead for some access patterns (multiple consecutive writes). A problem for all systems that access remote data or that replicate data is the overhead in detecting accesses to shared data. Furthermore, most software-based systems manage consistency by exploiting the synchronization operations. This implies that it is hard to combine messages for multiple data regions or objects. Finally, no single protocol is best for all situations.

3.3 Collective communication

The previous two sections described remote data access mechanisms that are based on independent processes. Whenever a process needs to access remote data, it has to send a message to this processor, either to transfer execution state or to request the remote data. Only when this request message is received will this remote processor participate in the communication that is necessary to access the data.

In some parallel applications, however, it is known in advance which communication has to take place and when this communication will happen. So, if a process needs some data from a remote processor, both sides will know this in advance. Therefore, the request message to the remote processor is not necessary. Instead, the remote processor can send the requested data immediately and save one message, even before the receiver tries to access it.

Collective communication is communication that involves a group of processes (groups can contain more than two processors). A collective communication operation is executed by having all processes in the group call the communication routine with matching arguments. The communication library takes care of synchronizing all processors in the group and moving all data to their destinations. One of the main benefits of using collective communication operations is that the implementation can exploit the knowledge about the communication behavior of each operation to optimize the communication pattern.

Operation	**Before**	**After**
broadcast	x at P_k, k given	x at all processors
gather	x_j at P_j	x at P_k, k given
allgather	x_j at P_j	x at all processors
scatter	x at P_k, k given	x_j at P_j
reduce	x_j at P_j	$\oplus_{j=0}^{p-1} x_j$ at P_k, k given
allreduce	x_j at P_j	$\oplus_{j=0}^{p-1} x_j$ at all processors

Table 3.1: Summary of some collective communication operations.

MPI [92] is a message-passing standard that defines a set of core library routines that are efficiently implementable on a wide range of computers. Table 3.1 gives a summary of some of the collective communication operations defined by MPI (see also [19]). For each operation, a description is given of the data that each processor has before and after the operation. There are p processors, labeled P_0 ... P_{p-1}. Before an operation starts, a processor P_j can have all (x) or part (x_j) of the data. For example, in the broadcast operation one processor (P_k) has data x when the operation starts. All processors invoke the broadcast operation with arguments that specify which processor is going to broadcast and which data buffer should be used to send or receive the data. After a processor returns from the operation, its local buffer contains data x.

Instead of only moving data, some operations can also apply a computation to the data. The operation $\oplus$ in the table represents an associative and commutative combine operation. Because the operation is associative and commutative, the operands do not have to be applied in a specific order. Therefore, the communication library can apply this operation at different processors on partial data, and forward the intermediate results to other processors.

For example, a typical reduction operation is to get the minimum value of a large, distributed array of integers. First, all processors compute the local minimum. After that, all processors start the execution of the reduction operation, specifying the local minimum, the operation to apply to each element (i.e., a function that computes the minimum of two integers), and the processor that will receive the result. After the destination processor returns from the operation, the minimum of all values is known. The behavior for allreduce is the same, except that the minimum value will be known on all processors after the operation.

MPI also defines a *barrier* operation to do process synchronization. A processor that calls a barrier operation is blocked until all processors have called it.

An application that uses MPI can specify which processors participate in each collective communication operation by using a *communicator*, which defines the group of participating processors. This provides more flexibility in the way that applications can apply collective communication primitives, because part of the processors can be scheduled to perform the computation that needs collective communication, while other processors can do other work. Creating a communicator, however, is a collective operation and may require interprocess communication. For example, the MPICH [57] implementation uses an allreduce operation to find a context identifier that is not in use. Thus, it may be too expensive to build many different communicators on-the-fly. This limits the use of processor groups in collective communication operations.

A large number of implementations of optimized collective communication libraries exist for different architectures [17,19,20,41,67,91,93]. All these implementations try to exploit the characteristics of the underlying architecture to make the operations as efficient as possible. Typical characteristics that influence these libraries are bandwidth, latency, and contention. *Bandwidth* limits the amount of data that can be sent across a network link. *Latency* determines how long it takes for an empty message to arrive at a remote processor. Finally, *contention* occurs when a processor or network link is overloaded [68].

The main benefit of collective communication is that all participating processors know what must be done. This knowledge allows them to use a communication pattern that is optimized with respect to the data size, bandwidth, latency, and contention. Another benefit is that by defining a single interface and implementing that interface on different architectures, applications are more portable than when they are written for a specific network.

Drawbacks of most collective communication libraries are that they specify a fixed set of communication primitives with limited data manipulation capabilities. The MPI standard provides sixteen collective communication primitives and twelve predefined reduction operations. The programmer can also define reduction operations, provided that the functions are associative (commutativity can be exploited, but is not a requirement). In all these collective communication primitives, all processors that participate behave the same. For example, in a reduction all processors send one buffer; it is not possible for a processor to send more than one buffer. Also, a processor that does not have any data to send and therefore does not want to participate in the communication pattern must be excluded from the processor group, which may be expensive.

One of the reasons that processors might have more than one data buffer is because of multithreading. To execute data parallel codes independent of other threads, collective operations and relative indexing among threads is required. Ropes [62] is a mechanism that provides this functionality. Ropes is part of Chant [60], a thread package that provides communication operations between threads instead of processors. Placing multiple threads that participate in a data parallel operation on one processor results in a processor that has multiple data buffers. The Chant system adds functionality on top of existing communication libraries (such as MPI) to deal with this.

Another drawback of collective communication is that only global reduction operations can be defined by the programmer. In some applications, the communication pattern is known in advance, but does not resemble any of the predefined collective communication operations. It is not possible to exploit this knowledge then.

A final drawback of collective communication is that all operations are considered separately. There is no compiler that can apply communication optimizations covering a group of collective communication primitives. [71] describes a system that aggregates acknowledgment messages by analyzing communication patterns. The patterns are constructed as iterations consisting of sequences of basic primitives (send-receive, exchange, request-reply, reduction, multicast, and reduction-multicast). By using compile-time information and global knowledge about the behavior of the program, a more optimized communication pattern will be generated. This solution, however, requires a complete description of the communication pattern of the application at compile time.

3.4 Data parallelism

The fourth technique that has influenced our work on collective computation is data parallelism (see Figure 3.1), which we will discuss here.

One of the main uses of collective communication is to implement data-parallel programming systems. A data-parallel program is a sequential program, in which the programmer specifies how the data (usually arrays) are partitioned over the processors. In the data-parallel programming model, the sequential part is executed by all processors, while the parallel loops (which are either detected by the compiler or are identified by the programmer with special programming constructions such as *forall*) that access the partitioned data are distributed across processors (data parallelism). The compiler rewrites this sequential program such that each update will take place at the processor that owns the data. High Performance Fortran [81,89] is a typical example of a parallel language that fits into this programming model.

Compilers for data-parallel languages are hard to write, because they must determine the complete communication pattern. This implies that a full data dependency analysis must be applied to the application program. Whenever it is not possible to do this analysis, the compiler has to revert to run-time algorithms to resolve these data dependencies. A typical Fortran example for which the compiler cannot accurately determine the data dependencies at compile-time is the irregular loop nest. In an irregular loop nest, an indirection array is used to access elements from another array. Since the values of the elements of the indirection array are determined at run time, the compiler cannot determine all data dependencies.

```
do n = 1, n_step
  do i = 1, sizeof_indirection_arrays
    x(ia(i)) = x (ia(i)) + y(ib(i))
  end do
end do
```

Figure 3.3: An irregular loop nest in Fortran.

In Figure 3.3 the data arrays `x` and `y` are partitioned and distributed over the processors by the programmer. The indirection arrays `ia` and `ib` are filled in at run time and determine the data dependencies between the elements of array `x` and the elements of array `y`.

Whenever a compiler has to deal with an irregular loop nest, it has to use some run-time algorithm to resolve the data dependencies. If the runtime system would resolve dependencies on demand, a large amount of communication would be needed. Based on the actual distribution of `x` and `y`, however, a run-time preprocessing step (the *inspector*) can determine which global elements stored on a remote processor are accessed by each processor and thereby determine the communication pattern [116]. Two main communication optimizations are to request a remote element only once, even if it is used multiple times (software caching), and to combine prefetches of remote data into a single message (communication vectorization). In applications where the indirection arrays do not change (static irregular problems), this preprocessor step has to be applied only once. If the application does change the indirection array (adaptive irregular problems), the preprocessing stage must be repeated in order to determine the new communication pattern.

When the inspector is executed, it generates a *communication schedule*. A communication schedule for processor p contains all the information that this processor needs to send and receive all necessary data. The *gather schedule* can be used for fetching remote elements, while the *scatter schedule* can be used to send updated remote elements [101]. The loop iterations are partitioned using the *almost-owner-computes* rule, which assigns an iteration to the processor that owns the majority of data array elements accessed in that iteration.

Finally, the *executor* uses the data partitioning information and the communication schedule generated by the inspector to execute the irregular loop nest. The executor uses the gather schedule to prefetch remote data, executes the loop nest, and moves the elements back to their home locations using the scatter schedule.

The main benefit of data-parallel programming systems is that they allow a large part of the communication pattern to be precomputed in advance, since the data distribution and the number of processors are known at compile time. If the communication pattern is determined by run-time properties, communication schedules can be used to describe these communication patterns. Since the communication pattern is described in terms of data dependencies, the runtime system can determine the appropriate communication schedule, for example by selecting collective communication operations to do the actual communication. Drawbacks of current systems based on communication schedules is that they are focused on (irregular) data parallelism and have not been applied to task parallelism. Also, these systems designate a single owner for every data element, so they do not deal with update replication. (Software caching can be seen as invalidate replication, in which the data is invalidated implicitly after the loop nest.)

3.5 Collective computation

The previous four sections described techniques for implementing parallel programming languages. Each of these techniques has its own application area, benefits, and drawbacks. In this section, we will use the concepts of these techniques for the design of a new execution model: collective computation. Collective computation will be used to implement the extended shared data object model.

The programming model that the system should support is an object-based parallel programming language for task-parallel programming. One implication of this environment is that the placement of objects is determined at run time, so the communication pattern of the application is not known at compile time. We will assume, however, that objects do not often migrate or create new replicas [12]. This allows all processors to maintain the set of current replicas of each object. Processes are also created at run time; it is unknown at compile time how many forks of each process type will occur at run time, and on which processors these processes will run.

The main benefit of computation migration, as discussed in Section 3.1, is that it allows a tradeoff between data locality and load balancing. By shipping part of the state of the invoking thread, multiple consecutive accesses to remote objects can be handled in one message.

Compared to remote access mechanisms, data replication (Section 3.2) can provide even more data locality by moving a copy of the object state to the invoking processor. Since more processors have local access to a copy, more operations can be performed without communication. On the other hand, all remote copies have to be either invalidated or updated when a processor writes an object. By using an update-replication protocol, consistency management and data movement messages can be integrated, thereby causing fewer coherency messages than by using an invalidate-replication protocol. The application behavior, however, determines the best replication and consistency strategy. For some applications, different objects can be handled by a selection of algorithms implemented in the runtime system. The Orca runtime system, for example, provides a single-copy and an update replication algorithm. Other applications, however, show large benefits if the programmer can provide an application-specific protocol for some of its objects, which implements a less stringent consistency protocol. Therefore, it should be possible to add new protocols on a per-object basis such that they can cooperate in one application.

Collective communication (Section 3.3) shows that a runtime system can apply communication optimizations if it knows the exact communication pattern. However, most collective communication libraries present a nonextensible set of primitives. This makes their usage in object-based parallel languages restricted. Furthermore, communication optimizations are limited to one operation at a time. Ideally, we would like to be able to describe arbitrary communication patterns with arbitrary intermediate computations, and let the system handle the architecture-specific optimization of this pattern.

Finally, communication schedules (Section 3.4) allow the description of arbitrary communication patterns. An inspector code analyses the application code (i.e., the irregular loop nest) and generates an appropriate communication schedule. Every time the application code is executed, an executor uses the communication schedule to make the local data consistent. However, this technique has only been applied to handle (irregular) data-parallel applications.

This thesis presents a new technique, called *collective computation*. With collective computation, a process can start the same user-defined function on a set of processors. Each processor executes its *instance* of the collective computation in a total order with respect to the other collective computations (i.e., for each set of processors that execute the same collective computation, they all execute the collective computation in the same order; the moment in time that the execution starts is not synchronized, though). Each collective computation function is written in a single-program, multiple-data style, which implies that all processors that run this function know exactly what the other processors are doing. An operation on an object, however, is only executed if a local copy of the object is present. Since every processor knows which processors have local copies of each object, it can be determined at run time which processors can execute an operation.

Collective computation functions can execute arbitrary code. Since each function is executed in a total order, it is not difficult to implement sequential consistency using collective computation. The collective computation function can contain not only operations on objects, but also arbitrary expressions and control flow statements that depend on values returned from earlier operations. The total ordering only restricts the *order* in which collective computation functions are executed; it does not enforce extra synchronization to execute a collective computation function on a set of processors at the same *time*.

A simple example of a collective computation is a write operation on a single Orca object with update replication (see Section 2.1.2). The invoking processor starts a collective computation on all processors that have a copy of the object, passing as argument the parameters to the write operation. The collective computation executes this operation on all processors that have a local copy of the object. Since collective computations are executed in total order with respect to each other, sequential consistency is preserved (see Section 2.1.2). After the write operation is finished, one of the processors is assigned the task to return any results to the invoking process (preferably the instance on the invoking processor). When the collective computation is finished, the invoking process can continue.

The main benefit of collective computation is that executing the same function on a set of processors allows us to make distributed decisions about the optimal communication pattern. Since all processors perform the same statements, all processors know which processors have a result and which processors need this result. This implies that a processor never has to ask (i.e., send a data request message) for a certain result, but instead all processors that have the result will make a distributed agreement decision which processor is going to send the result. Since all processors have complete knowledge of the data dependencies, this distributed decision does not require communication. In addition, multiple results can be collected in a single message, which saves the overhead of sending multiple messages.

Collective computation also eliminates the necessity for separate lock messages. Locks can be acquired when the collective computation function is started, because all collective computations are executed in a total order. It is also not necessary to send a lock release message at the end of the collective computation, since each processor knows that the other processors will not need data from it when it has finished its execution. Therefore, a collective computation function can release the locks on the local processor when it terminates.

Another benefit of collective computation is that operations on objects can occur in parallel. A processor only performs operations on a local copy and skips operations on the other objects. Furthermore, a processor only has to wait if it depends on the result of an earlier operation. Operations on objects on different processors that do not depend on each other are therefore executed in parallel.

A drawback of collective computation is that the workload is increased on the processors that participate in the collective computation, since they all execute the same function. This restricts the amount of computation that can be performed within a collective computation.

Collective computation uses a total ordering to preserve sequential consistency. To obtain a total ordering introduces extra overhead (e.g., fetching the sequence number from a centralized sequencer). An earlier study using Orca, where total ordering is also used to obtain sequential consistency, shows that this overhead is marginal. On a set of ten applications, nine show an overhead of less than 1 percent; the last application shows an overhead of less than 5 percent.

Collective computation on multiple objects

To illustrate collective computation on multiple objects, we will discuss how the atomic function given in Figure 3.4 would be executed using remote procedure calls, computation migration, and collective computation. For all three cases, the synchronization required for the atomic function is implemented using a totally-ordered multicast message. In the following examples, processor 2 invokes the atomic function while object `x` has a single copy at processor 3 and object `y` has a single copy at processor 1.

Figure 3.5 shows the communication pattern that would occur if atomic functions were implemented using remote procedure calls. First, all processors involved in this atomic function have to be synchronized. This can be implemented efficiently by using a totally-ordered group message that is sent by processor 2 (the invoking processor). Since this message is received on all processors in the same order with respect to other group messages, it determines the sequence in which this atomic function is executed with respect to other atomic functions and normal Orca operations. Each processor locally acquires locks on all objects involved in this atomic function. Other processors that are not involved in this atomic func-

```
atomic function copy(x, y: shared IntObject);
    r: integer;
begin
    r := x$Value();
    y$Assign(r);
end;
```

Figure 3.4: Atomic function that copies the integer value of object x to object y.

tion (because they do not have a copy of one of the objects) can ignore this group message. In the example, processor 1 locally locks object y and processor 3 locally locks object x. Processor 2 does not have to acquire a lock, because it does not have a local copy of any of the two objects.

When the synchronization message has been handled and all involved objects have been locked, processor 2 starts the execution of the atomic function. The first statement involves a read operation on remote object x. Therefore, processor 2 issues a remote procedure call to processor 3, specifying the operation and its arguments[3]. When the result of this operation is returned, processor 2 invokes a second remote procedure call to perform the write operation on object y.

After all statements of the atomic function have been executed, all locks have to be released to allow other operations to continue. Since processor 2 is the only processor that knows when the atomic function is finished, it has to send a message to all other processors telling them to release the locks on the objects. From then on, other operations and atomic functions that use any of these objects can continue.

Note that this scheme is easily extendible to replicated objects. Again the synchronization message will guarantee that all processors that have a local copy of any of the objects involved will acquire locks on these objects. After synchronization, the invoking processor again starts the execution of the statements of the atomic function. Nonreplicated objects are handled in the same way as described before. For replicated objects, however, we distinguish two cases. If the operation is a read operation, only a single copy of the object needs to be accessed. For example, if object x is replicated on all three processors, processor 2 would perform the read operation locally, so without doing any communication. For write operations, a group message is sent and all processors locally update their copy of the object. Finally, after all statements of the atomic function have been executed, again a group message has to be sent to release the locks.

In Figure 3.6, the communication pattern is presented that occurs if the imple-

[3]The runtime system has to guarantee that handling the request message at processor 3 occurs after all locks have been granted.

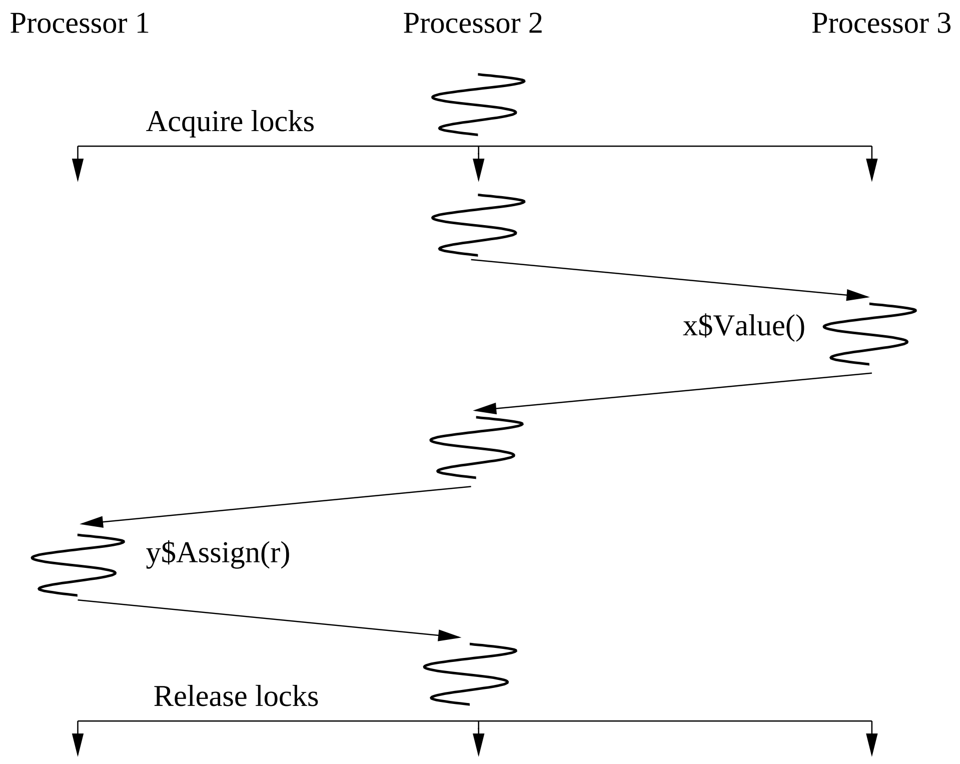


Figure 3.5: Communication pattern for the atomic function `copy` using remote procedure calls.

mentation would use computation migration. Again processor 2 sends a synchronization message to acquire all locks and starts the execution of statements of the atomic function. Instead of doing a remote procedure call to perform the operation on object x, however, the computation is migrated to processor 3. When the computation continues on processor 3, it invokes the operation on object x locally. To perform the write operation on object y, the computation migrates to processor 1. After the write operation is finished, processor 1 sends a group message to release all locks. When processor 2 receives this message, it also knows that the atomic function is finished, and that the invoking process may continue.

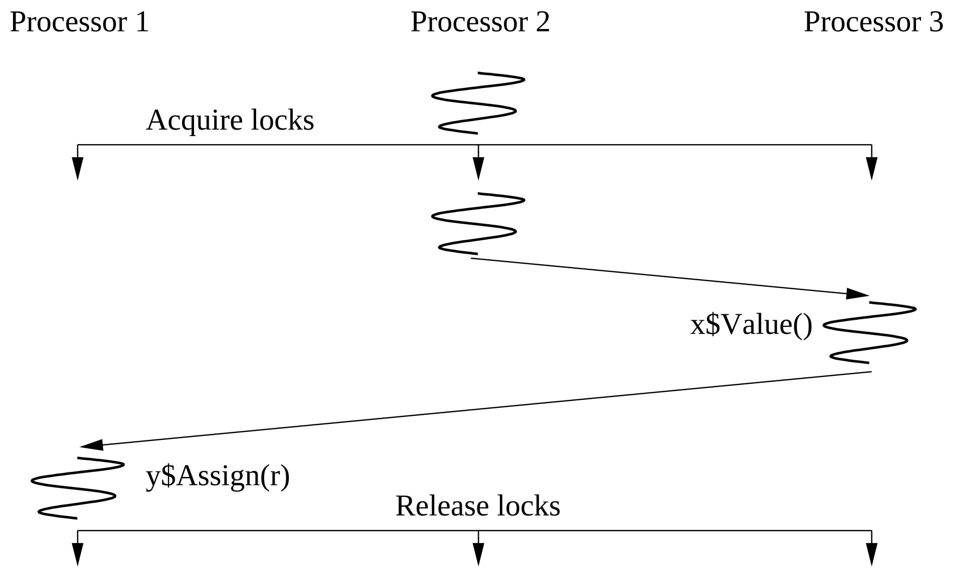


Figure 3.6: Communication pattern for the atomic function copy using computation migration.

As already mentioned in Section 3.1, computation migration is not able to deal with replicated data. A solution would be to use computation migration for all operations on nonreplicated objects and for read operations on replicated objects, since only one copy needs to be accessed then. For write operations on replicated objects, it is still necessary that the currently executing processor invokes a group message.

Figure 3.7 shows the communication pattern if collective computation is used for the atomic function given in Figure 3.4. Again, a synchronization group message is sent to acquire the locks on all involved objects. After acquiring the locks, however, all involved processors start the execution of the atomic function (i.e., the collective computation function), and those processors that have a local copy of object x perform the read operation. In our example, only processor 3 executes the read operation, while the other processors skip the operation. All processors then reach the write operation on object `y`. At that moment, all processors can determine which processors have the result value `r` (i.e., processor 3) and which processors need this result value (i.e., processor 1). Based on this information, all instances collectively decide that processor 3 will send the value of variable `r` to processor 1. Processor 1 is blocked until it has received this value, after which it can perform the write operation locally. The other processors skip the write operation, since they do not have a local copy of object `y`.

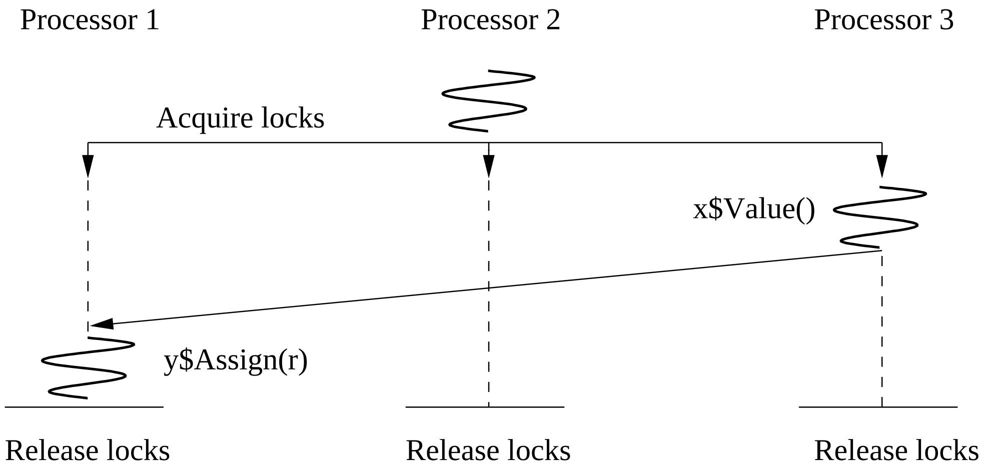


Figure 3.7: Communication pattern for the atomic function `copy` using collective computation.

When each processor reaches the end of the atomic function, it knows that it does not need to receive a message from another processor, because in that case it would have been blocked. Furthermore, it has sent all data that other processors need to continue their execution of the atomic function. Therefore, it can immediately release all the locks on the objects, and no group message is needed. When processor 2 finishes the execution of the atomic function (i.e., after skipping both operations), the invoking process can continue immediately. Since the atomic function does not return a result, and processor 2 does not have any of the two objects, it is even possible that the invoking process continues immediately after starting the atomic function. The sequential ordering of the synchronization group messages assures that sequential consistency is preserved.

Replicated data can easily be handled by collective computation, since all processors perform all operations on their local copies of each object. For write operations, this implies that all updates on objects are performed in total order, thereby preserving sequential consistency. For read operations, this implies that the results of the operation are available for all further operations that will be performed within the atomic function. For example, if object `x` is replicated on all processors, no communication occurs to resolve the data dependency, because all processors know that processor 1 can locally compute the consistent value of variable `r`.

To summarize, the remote procedure call version uses two group messages and four unicast messages for the example situation. Using computation migration, the unnecessary stage of receiving a reply and forwarding it directly to another processor can be avoided. This reduces the number of unicast messages to three. Collective computation avoids the initial migration message to the processor that has the first object that will be accessed. In addition, no release group message is required, because all processors can release the locks locally. Therefore, collective computation only needs one group message and one unicast message for the example situation.

A more difficult aspect of collective computation is how to deal with control flow. Control flow determines the statements that will be executed inside the collective computation function, in particular which operations will be executed and in which order. Since this information is crucial for the optimization of the communication pattern, each control flow condition expression has to be evaluated on all processors. The control flow condition expression, however, can depend on the result of an earlier operation. Therefore, it is necessary to propagate the result of this operation to all participating processors, so that they all can evaluate the condition and perform the same statements.

Conclusions

The collective computation model combines the benefits of all four techniques that are presented in this chapter. Changing the number of operations inside a collective computation operation allows the programmer to trade off data locality and load balancing. Data replication is handled by executing each operation on all processors that have a local copy of the object. Since all processors execute the same code and all processors know about the actual replication of objects, it is possible to determine an optimized communication pattern before the communication takes place. Finally, the collective computation function can be regarded as a combined inspector and executor. The collective computation function determines data dependencies on-the-fly, while the runtime system resolves them.

To summarize, the benefits of collective computation are:

- It provides a user-defined trade-off between data locality and load balancing: within a collective computation all accesses to objects are local, and communication takes place only to resolve data dependencies. The granularity of the collective computation determines the load on the other processors.

- It supports replicated objects in a simple way.
- One collective computation can synchronize multiple objects.
- It is possible to generate optimized communication patterns.

Collective computation also has its problems, however. One problem is the computational overhead caused by duplicating the computation on all involved processors. In the previous example where object x is replicated, processor 2 and processor 3 perform the local read operation without using the result. Furthermore, all processors perform all statements in the atomic function that do not perform operations on objects. Our assumption is that this overhead in computation is compensated for by the reduction in communication. We believe that this assumption is valid for current and future architectures. For example, consider a cluster of workstations with Pentium Pros running at 200 MHz, using Myrinet as its network architecture. The minimum message latency will be around 10 μs. Within this 10 μs, 2000 clock ticks occur, that could have been used to compute a result locally instead of receiving it from the network.

Another problem with applying collective computation in an object-based parallel language such as Orca is that the replication and distribution of objects is not known at compile time. Therefore, the communication pattern, which depends on the object placement, has to be computed at run time. We claim that on average the overhead of computing the communication pattern is outweighed by the gain of reducing communication. Furthermore, the overhead of sometimes performing redundant computations is negligible compared to the performance gain caused by reducing the amount of communication. We will support these claims with the performance numbers that will be presented in Chapters 7- 9.

The third problem with collective computation is the way in which control flow has to be handled. Since all involved processors have to perform the same statements and operations, they have to make the same flow control decisions. This requires that the variables on which the flow control condition depends have to be consistent on all processors. If such a variable is the result of an operation on an object, the result has to be sent to all processors that do not have a copy of this object. A typical case is a nested job queue object, which contains subobjects that each contain a job queue. A dequeue operation first tries its own queue. If this queue is empty, however, we would like the dequeue operation to try other queues until a job is found. Since the loop that accesses the subobjects end when a queue is not empty, all processors must know whether a queue is empty or not. In Section 6.4, we will present the functionality to resolve data dependencies that arise during the execution of collective computations.

3.6 Architecture overview

The thesis of this dissertation, which is given in Section 1.6, states that collective computation allows the efficient implementation of high-level support for atomic operations on multiple objects and for partitioned objects. To support this claim, we have implemented a programming system that is designed around collective computation. The goals of this prototype implementation are:

- Provide a flexible runtime support system for implementing and using extensions to the shared data object model.
- Show that the extended shared data object model can be implemented efficiently using collective computation, and assess the performance of using collective computation with respect to other implementation methods.
- Illustrate that our extensions (i.e., atomic fucntions and nested objects) facilitate writing efficient parallel programs.

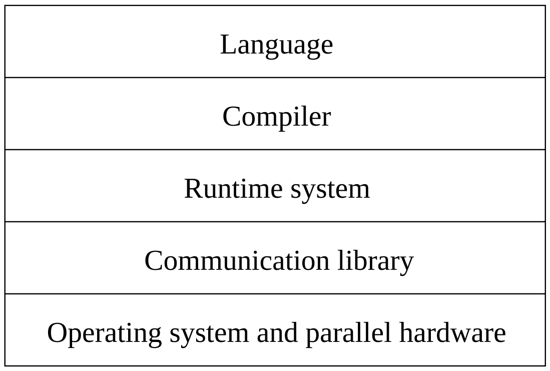


Figure 3.8: System layers for a parallel programming system on a generic parallel architecture.

Figure 3.8 presents the layers of a parallel programming system. An important design issue is the definition of the interfaces that are used to access the functionality provided by a layer. On the one hand, this interface must allow the higher layer to can use its functionality without knowledge of the underlying implementation. On the other hand, the interface should not decrease the overall performance of the system. Therefore, to evaluate the collective computation model, we need to evaluate it in the context of a complete parallel programming system.

First, we will describe each layer in more detail and explain the requirements that each layer has to fulfill. The *language* describes the way the application programmer communicates with the programming system. Chapter 2 gives an overview of our language: the Orca programming language and the two extensions that we have proposed: atomic functions and nested objects. We will provide suppot for our claim that these extensions are useful for writing parallel programs.

The *compiler* translates the application to a binary format that can be executed on a specific parallel architecture. A large fraction of the code that is required to run the application can be shared by all applications. Therefore, this part is encapsulated in the runtime system (discussed below). The Orca compiler performs the following tasks:

- It translates the program code to object files that can be linked to the runtime system to create a binary that can be executed on a specific architecture.
- It generates code to initialize the runtime system and to register object and data types, object operation implementations, marshal and unmarshal routines, and process code fragments at the runtime system.
- It generates code that translates the language primitives to the corresponding runtime system interface calls (e.g., to create a process on another processor or to invoke an operation on an object.) In addition, the compiler passes annotations (i.e., hints specified by the programmer to the runtime system to improve the performance of the application, such as object placement strategies) to the corresponding runtime system interface calls.

Implementing the extended Orca system using collective computation adds three extra tasks to the compiler (see also Section 2.5):

- The compiler has to pass data dependency information to the runtime system. In the original Orca system, only one processor needs the result of the invocation of an operation on an object. With collective computation, however, multiple processors will need these results, which may require communication to get the results to those processors that do not have a copy of the object. Since all communication is handled by the runtime system, the compiler needs to pass this dependency information to the runtime system.
- The compiler has to inform the runtime system when data dependencies need to be resolved (the synchronization point). Preferably, the compiler has to generate as few synchronization points as possible, to allow the runtime system to optimize communication patterns (the more data dependencies are resolved within one synchronization point, the better).
- In order to fully benefit from the collective computation model, the compiler should recognize specific object operation patterns within the code and pass

this information to the runtime system, to allow optimized communication patterns to be used.

Although the use of collective computation puts extra requirements on the compiler, we do not present a compiler implementation in this thesis. The extra requirements imposed on the compiler are similar to those in other research areas, such as compiler optimizations (the generation of data dependency information and the optimization of synchronization points) and compilers for data parallel programming languages (recognizing regular patterns). Therefore, we focus on the runtime aspects, and provide an interface that allows a simple compiler strategy to work correctly and that also allows hand-written translations of an application to achieve good performance.

The *runtime system* manages the state of an application at run time. In addition, the runtime system hides all communication. Therefore, the compiler only has to generate code that deals with objects and processes, abstractions that are already present in the language. For the standard Orca system, the tasks of the runtime system include:

- Managing processes running on different processors.
- Managing object placement and replication.
- Performing object operation invocations.
- Handling mutual exclusion synchronization and condition synchronization on a single object.

In the extended Orca system, the runtime system is the layer that provides collective computation. Collective computation is implemented by an abstraction called *weavers*. A weaver is an invocation of a single function on a set of processors in a total order. Within a weaver, the participating processors can communicate with each other. We will show how this weaver concept is used to implement the following additional requirements that the extended runtime system must provide:

- Handle mutual exclusion synchronization and condition synchronization on *multiple* objects.
- Generate optimized communication patterns for the communication between the processors that participate in a weaver.
- Manage and resolve data dependencies within the execution of a single weaver.

The extended system will contain two classes of objects, namely normal shared objects (called Orca objects) and nested objects. In addition, within a nested object operation and an atomic function, both kinds of objects can be invoked. To handle the complexity that such interactions impose, the runtime system is split in two parts. The upper part of the runtime system consists of a set of model-specific runtime systems (also called *modules*), which implement an object class or a high-level invocation mechanism. In this thesis, we will present the Orca module, the nested object module, and the atomic function module.

The lower part of the runtime system, the *generic runtime system*, provides the common parts that are shared by the runtime modules, such as the weaver abstraction. The generic runtime system also provides the glue between these runtime modules so that they can be used together in one application. A runtime module only needs to be aware of the generic runtime system; all interaction between modules is handled by the generic runtime system. By splitting the runtime system in a generic part and model-specific runtime modules, it is easy to extend the system with other modules.

The *communication library* provides a portable communication interface that hides the differences in parallel architectures. Most often, communication is strongly related to issues like processor allocation, application startup and termination, and thread management. Therefore, this functionality is also provided by the communication library.

One of the benefits of collective computation is that it allows the communication between the participating processors to be optimized, because all involved processors are aware of the data dependencies that need to be resolved within the weaver. To facilitate the translation of run time data dependencies to communication, we designed an abstraction, called a *communication schedule*, which captures a complete communication pattern. When the generic runtime system needs to resolve data dependencies between the processors of a weaver, it first generates the communication schedule. After the communication schedule is built completely, it is executed by the communication library.

The reason to split the generation of the communication schedule from the execution is that it allows the communication library to exploit the whole communication pattern for performing optimizations. In this thesis, we will present optimizations to reduce the number of acknowledgment messages (for unreliable communication systems) and to eliminate context switching overhead on message arrival. An additional reason is that it allows the library to define template communication patterns (such as reduction and all-to-all) that can be instantiated within a communication schedule and combined with other communication patterns. Since the template communication patterns are provided by the communication library, they can be tuned for a specific parallel architecture.

Finally, the *operating system and parallel hardware* provide the minimal support that is required to run parallel applications. We assume in this thesis the generic parallel architecture that is presented in Section 1.1.

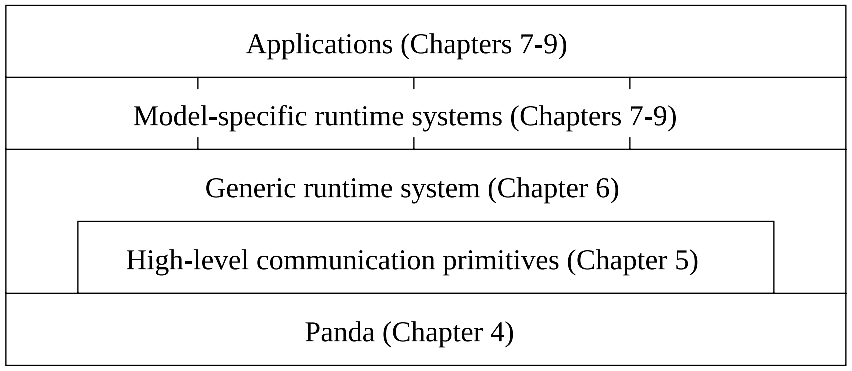

Applications (Chapters 7-9)
Model-specific runtime systems (Chapters 7-9)
Generic runtime system (Chapter 6)
High-level communication primitives (Chapter 5)
Panda (Chapter 4)

Figure 3.9: Overview of the prototype implementation.

Figure 3.9 presents an overview of the prototype implementation. At the bottom layer we use Panda, which was originally designed to implement a portable version of the Orca system [10,31,108]. Panda provides threads, which are used to implement multiple processes and collective computation instances per processor. Panda also provides reliable message passing and reliable totally-ordered group communication. Note that all higher-level layers are built on top of Panda; no operating system functions are used directly, except those provided by the ANSI-C language [80], in which all layers are implemented. Therefore, to port the system to another architecture it suffices to port Panda. Panda is described in Chapter 4.

A high-level communication layer is built on top of the communication primitives directly provided by Panda (message passing and group communication). It provides the *communication schedule* abstraction, that can be used to describe arbitrary communication patterns. The most important property of this layer is that it is designed to execute these arbitrary communication patterns as efficiently as possible. This communication layer is described in Chapter 5.

The generic runtime system provides the common abstractions of the runtime system, including the collective computation implementation based on weavers. During the execution of collective computation operations, certain communication patterns will occur. The generic runtime system tries to gather as much information as possible about these communication patterns, and will use this information to generate an optimized communication schedule. The generic runtime system is described in Chapter 6.

On top of the generic runtime system, model-specific runtime systems are implemented for the shared data object model, the nested object model, and for atomic functions. Since all these model-specific runtime systems use the generic runtime system, they are able to interact without being aware of the implementation of each other. For example, the runtime system that provides atomic functions can apply an atomic function on a nested object and a shared data object, without being aware of the implementation details. The runtime system is only aware of the object interface that is provided by the generic runtime system. The model-specific runtime systems are described in Chapters 7-9.

Finally, applications are built on top of these runtime systems. These applications have to be aware of the specifics provided by the model-specific runtime systems, because applications depend on their behavior. Each application therefore selects only those runtime systems it needs. Each chapter that describes a model-specific runtime system also discusses some applications. Since no compiler support is available, applications and benchmarks are hand-written in ANSI-C [80].

Summary

This chapter described the collective computation execution model. Collective computation is the execution of the same function on a set of processors in a total order. Collective computation is designed to implement efficient runtime support object-based parallel programming languages, since it is well suited for performing operations on replicated and single-copy objects.

To evaluate the collective computation model, we looked at other techniques that have been used to implement runtime support for parallel programming. These techniques, remote access mechanisms, data replication, collective communication, and data parallelism, each have their benefits and drawbacks when they are applied to implement parallel object-based languages. Collective computation combines the benefits of these four techniques.

Chapter 4

Panda

Panda is a portable virtual machine designed to support implementations of parallel programming systems. Originally, Panda was used to implement a portable version of the Orca programming system [31]. By using a virtual machine, all machine-specific details are hidden from the Orca runtime system. Therefore, to port the Orca runtime system to a new architecture it suffices to port Panda. Later, Panda has also been used to implement other parallel programming systems, such as PVM, SR, and Linda [42, 108].

Two basic abstractions are provided by Panda: threads and communication. Table 4.1 gives an overview of the most important Panda primitives. The left column gives the abstractions provided by Panda, and the right column gives the functions associated with these abstractions. These functions will be explained in this chapter.

Threads & synchronization	
threads	create, exit, join, yield, self, set priority
mutex	lock, unlock
condition variable	wait, timed wait, signal, broadcast
Communication	
Message Passing	register, send, receive, poll
Group Communication	register, send

Table 4.1: Overview of the Panda interface.

Portability of the Panda system is achieved by defining two layers: a *system layer* that is architecture dependent; and an *interface layer* that provides the primitives that higher software layers can use. To obtain high efficiency, Panda is designed as a flexible system, in which the communication modules can be adapted statically (i.e., when the system is compiled) so that they benefit from the properties provided by the underlying operating system and hardware.

Not only the Panda library itself, but also the applications that run on top of Panda must achieve high performance. Therefore, the Panda interface went through a number of revisions to remove all bottlenecks that the earlier versions introduced, while still providing a high-level interface that hides the implementation. The version of Panda used in this thesis is Panda 3.0.

Panda is used by the high-level communication library and by the generic runtime system (see Figure 4.1). The generic runtime system uses message passing and group communication to create weavers. When a weaver is executed, its participating processors can communicate using the communication schedules provides by the high-level communication library. This communication library also uses the Panda communication primitives.

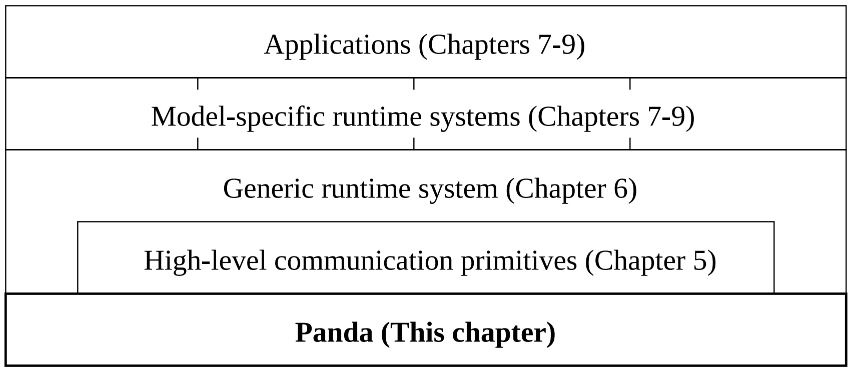

Applications (Chapters 7-9)
Model-specific runtime systems (Chapters 7-9)
Generic runtime system (Chapter 6)
High-level communication primitives (Chapter 5)
Panda (This chapter)

Figure 4.1: Panda in the architecture overview.

This chapter gives a detailed description of Panda. Section 4.1 describes how Panda is structured and how this structure allows high efficiency. In Section 4.2, the threads interface is described. Section 4.3 and Section 4.4 describe the message passing module and the group communication module, respectively. Section 4.5 gives an overview of the systems that Panda has been ported to. Finally, Section 4.6 presents performance numbers for the Panda implementations that will be used in this thesis.

4.1 Structure of the Panda portability layer

Figure 4.2 gives an overview of the Panda system. Panda consists of an internal system layer that provides threads and two low-level communication primitives: unicast and multicast. On top of that the interface layer defines multiple communication modules. Most operating system and machine specifics are hidden in the system layer. Three properties, however, cannot be hidden from the communication modules without causing too much overhead:

- Are the low-level communication primitives reliable or unreliable?
- Does the system layer provide a totally-ordered multicast primitive?

- Does the system layer impose a fixed upper bound on the packet size, and if so, how large is the maximum packet size?

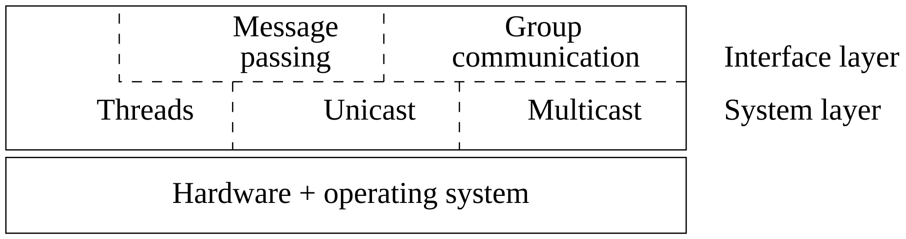


Figure 4.2: Structure of the Panda system.

The interface of the system layer primitives is fixed, so the signatures (number and types of parameters) of all procedures are identical on all architectures. The *semantics* of the primitives, however, change from architecture to architecture. In general, the system layer provides the semantics that fits best with the underlying hardware and operating system. For example, the unicast primitive is reliable only if the underlying machine provides reliable communication. For each architecture, the system layer primitives thus may or may not be reliable or totally-ordered, and may or may not accept messages of arbitrary size. These properties of a system layer for a specific architecture are expressed in a set of *system configuration parameters*.

A naive way to implement the communication modules would be to always assume the worst case (unreliable communication, no totally-ordered multicast, and messages with a limited size), but this would be inefficient on many systems. Therefore, each Panda communication module has a well-defined interface, but it can have multiple implementations, depending on the semantics of the system layer primitives. For example, if the *unicast* primitive of the system layer is unreliable, the message passing module will be implemented using a time-out and retransmission protocol; if unicast is reliable, the message passing module will be straightforward and therefore will not have this overhead. Likewise, the message passing layer uses a fragmentation protocol only if the unicast primitive imposes an upper bound on the message size. The group communication module also has multiple implementations, depending on the need for retransmissions, ordering, and fragmentation. In total, the Panda library provides user primitives with fixed semantics, that exploit the properties of the underlying system (reflected in the system layer) where possible.

The threads module is placed in the system layer, so it can directly access the threads primitives provided by the operating system, if available.

The configuration of the Panda system is done at compile time, using the system configuration parameters described above. The structure of the system is flexible and extendible. For example, users can add new modules to the interface layer. An interface layer module can also be built on top of other modules in this layer. For example, the Orca runtime system uses a remote procedure call module, which is built on top of the message passing module. Finally, and perhaps most importantly, the Panda system is highly portable. All architecture-dependent parts are isolated in the system layer. Implementing Panda on a new architecture thus involves: (1) implementing (or porting) the system layer and (2) configuring the interface layer. Panda has been implemented on several operating systems (Amoeba, SunOS, Solaris, Parix, AIX, and BSD/OS) and machines (the Thinking Machine CM-5, Parsytec GCel, Parsytec PowerXplorer, Meiko CS-2, IBM SP2, and on clusters of Pentium Pro workstations).

The system layer uses an *upcall* mechanism [47] to deliver packets to a communication module. Each communication module registers at startup a *handler function* that has to be called whenever a packet for this module arrives. When a packet arrives at a processor, the system layer determines which communication module must handle this packet and calls this handler function. An application that uses a communication module also registers a handler function to receive the complete message. After all packets of a message have arrived, the communication module also makes an upcall to deliver the complete message to the application. The implementation of the system layer may use a single thread to receive messages from the network and to make the upcall to the application. A consequence of this design choice is that the application program may not block in the upcall function waiting for another message [26,84]. This contract between Panda and the application makes it possible to implement efficient communication on a large number of architectures.

4.2 Threads

The Panda threads interface resembles the Pthreads [70] interface without the real-time features. The *pan_thread_create* function creates a new thread on the local processor and returns a pointer to a thread handle. When the thread is created, it starts to run a function that is passed as argument to the *pan_thread_create* function. Threads can remove themselves by calling *pan_thread_exit*. Other threads can wait for a thread to exit by calling *pan_thread_join* with as argument the thread handle of the exiting thread. Finally, threads can retrieve their own thread handle by calling *pan_thread_self*.

On most systems, Panda supports preemptively scheduled, prioritized threads. If preemption or priorities cannot be implemented on the underlying operating system, the runtime system or compiler can use *pan_thread_yield* to cause thread switching. As an example, the Orca system should always service incoming requests for operations as soon as possible. Therefore, it is important to be able to preempt (possibly long-running) computations. On systems that do not support preemptive scheduling, the Orca compiler therefore uses calls to *pan_thread_yield* to force preemption at certain points (e.g., loop headers and function entry points) in the program [40].

Two kinds of primitives are supported for synchronization between threads: mutexes and condition variables. Mutexes are used to provide mutual exclusion between threads. A thread can acquire a lock by calling *pan_mutex_lock*. Other threads that invoke *pan_mutex_lock* are blocked until the thread that owns (i.e., has acquired) the lock releases it by calling *pan_mutex_unlock*. Mutexes are typically used to protect access to some shared data; as long as a thread owns the lock associated with the data, no other thread can access this data, provided that the other threads try to acquire the lock first. Mutexes are used for short-term synchronization and are always used as a lock/unlock pair.

Condition variables are used for long-term synchronization. For example, a thread that waits for a certain message to arrive would block on a condition variable. When a condition variable is created, it is associated with a mutex. A thread must acquire this mutex before it can do an operation on the condition variable. Figure 4.3 gives an example. The code on the left-hand side first locks the mutex, and then checks the condition by evaluating a certain expression (for example whether a message has arrived). If the condition is not true, the function *pan_cond_wait* is called, which atomically unlocks the associated mutex and blocks the thread.

```
pan_mutex_lock(mutex);
while(<condition not true>) {
        pan_cond_wait(cond);
}
pan_mutex_unlock(mutex);
```

```
pan_mutex_lock(mutex);
<make condition true>
pan_cond_signal(cond);
pan_mutex_unlock(mutex);
```

Figure 4.3: Condition synchronization using condition variables in Panda.

The code on the right-hand side updates the condition. First, it acquires the lock associated with the condition variable to ascertain that no other thread will evaluate the condition expression while the condition is changed. Then it performs the update and calls *pan_cond_signal*, which will move *one* of the threads that is currently blocked on the condition variable to the queue of threads blocked on the associated mutex. If no thread is currently blocked on the condition variable, the signal is discarded. When the mutex is unlocked, the thread that was blocked on the condition variable can continue. It rechecks the condition, and if it is now true, it unlocks the mutex and continues. Instead of using *pan_cond_signal*, it is also possible to use *pan_cond_broadcast*. This function will move *all* threads currently blocked on the condition variable to the queue for the associated mutex. Later on, each of these threads will in turn acquire the lock.

Pan_cond_timedwait can be used to wait for a condition to change within a certain time. An absolute time is passed as parameter to *pan_cond_timedwait*. If the condition variable is signaled, it behaves like a normal *pan_cond_wait*. If the condition variable is not signaled before the current time reaches the time passed as parameter, *pan_cond_timedwait* returns with a return value that specifies that a timeout has occured. *Pan_cond_timedwait* is used, for example, to implement reliable communication protocols on unreliable networks.

4.2.1 Implementation

Many systems already have some threads support. Therefore, the Panda system was designed such that the whole threads module is in the system layer. Another design possibility would have been to define a system layer interface that provides functions to create a thread stack and to do context switching, like in QuickThreads [79]. An interface layer module would then use this interface to build a portable threads package.

We decided not to use such a system structure for the following reasons. First, it would not allow us to directly use the threads system provided by the vendor. Such a threads package could be more efficient than a threads system that is split in two layers. Second, it is important to integrate communication and threads efficiently [26]. This integration would not be possible if the low-level communication primitives are in the system layer while the thread scheduler is implemented on top of the system layer.

Since our threads interface resembles the Pthreads interface, it is often trivial to port the Panda threads interface on top of an existing threads interface. On Solaris, for example, the threads module merely wraps up the primitives of the Solaris threads library to make them look like Panda threads. On systems without (efficient) threads support, a user-level threads package is used [61], based on QuickThreads.

4.3 Message passing

The message passing module provides reliable point-to-point communication between processors. Two properties of the system layer determine which implementation can be used: whether the system layer supports arbitrary large messages and whether or not the system layer provides reliable communication. The combination of unreliable communication with arbitrary large messages is rare, because this would result in severe flow-control problems.

The message passing layer provides *ports* to be able to distinguish messages for different higher-level modules. Each module registers a port with the message passing layer by calling *pan_mp_register_port*. All ports have to be registered in the same order on all processors, since the sequence in which they are registered determines the key on which messages are demultiplexed.

The message passing module provides two ways to receive a message. A server can register an asynchronous handler function that will be called when a message arrives at the specified port. This handler function may be called directly from the upcall from the system layer, and therefore may not block (see Section 4.1). Another option is to issue an explicit receive call at a port. In this case, the thread that invokes the receive is blocked until the message arrives. Threads are also allowed to poll a port to see if a message has already arrived.

To send data, the sending processor constructs a message and calls *pan_mp_send* with the destination processor identifier and the port number as parameters. The sending processor can either wait until it is certain that the message will be delivered at the destination processor (synchronous send) or it can continue immediately (asynchronous send). With asynchronous send, an upcall is made at the sending processor when the data buffer that contained the message can be reused. Only asynchronous send can be used in an upcall, since upcalls are not allowed to block. The sending processor can also poll to check whether the asynchronous send has already been finished.

4.3.1 Implementation

If the system layer does not provide support for arbitrarily large messages, the message passing layer splits the message into fragments. Each fragment is sent separately to the destination processor, where the fragments are reassembled. When the final fragment has arrived, the complete message is delivered to the application with an upcall to a registered handler function.

The implementation of reliable message passing on unreliable communication is based on a stop-and-wait protocol. Each fragment is sent as a system layer packet[1] and is confirmed by an explicit acknowledgment message from the receiver to the sender. If no acknowledgment arrives, the packet is retransmitted. For efficiency, however, the message passing module provides three modes for

[1]A system layer packet may be larger than the packet size provided by the underlying hardware. On Solaris, for example, a 8 KByte UDP packet is sent over an Ethernet, which support packets containing a maximum of 1500 data bytes. Solaris performs the fragmentation and reassembly for the UDP packets.

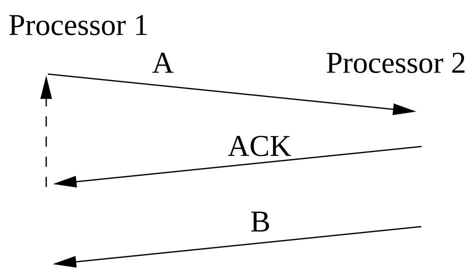
the last acknowledgment. (Optimizing the last acknowledgment is important, because most messages are small and consist of only one fragment.) In Figure 4.4, these three modes are presented where processor 1 sends a request message A to processor 2, which responds with reply message B. The three modes are:

a) explicit acknowledgment. An explicit acknowledgment message (ACK) is sent immediately when the last fragment of message A arrives. When the acknowledgment message is received, message A is known to be delivered (dashed arrow in Figure 4.4a). This is the default mode. No use is made of the fact that processor 2 will send message B in reply to message A. The sender has to set a timer to retransmit the fragment in case the acknowledgment message is lost.

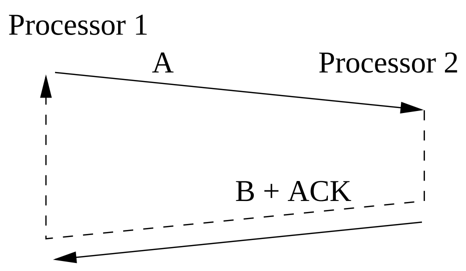
b) piggybacked acknowledgment. An acknowledgment is piggybacked on the first fragment of message B (B + ACK). When this first fragment of message B arrives, a field in the header acknowledges message A (dashed arrow). If message B is not sent soon enough, an explicit acknowledgment is sent (i.e., like in case a). Both the sender and receiver have to set a timer in this mode: the sender to retransmit the packet and the receiver to send an explicit acknowledgment.

c) implicit acknowledgment. The sender knows that it will receive a message that is causally related to message A, and assumes that it will not take too long before this reply is sent. When this reply message arrives, the application has to make a call to the message passing module to confirm that the message is delivered at the destination (dashed arrow). Only the sender has to set a timer to retransmit a packet.

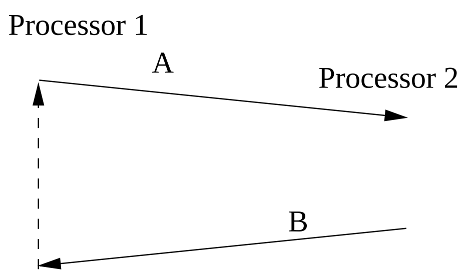
The piggybacked acknowledgment scheme is used in the remote procedure call module that is implemented on top of the message passing module. When the client sends the request message, it uses the piggybacked acknowledgment mode. The client knows that the sender will respond with a reply message. It is not known, however, whether this reply will be ready immediately when the request is handled. For example, remote object invocations are implemented using the remote procedure call module. If the operation blocks on a guard, it may take an arbitrary long time before the operation can execute the guard statements and generate the reply. Therefore, the remote procedure call module does not use the implicit acknowledgment mode, because this would cause the whole request message to be retransmitted if a timeout occurs. With the piggybacked acknowledgment mode, only a small explicit acknowledgment message is sent. The piggybacked acknowledgment mode is also used for the reply message, assuming



a) Explicit acknowledgment



b) Piggybacked acknowledgment



c) Implicit acknowledgment

Figure 4.4: Different acknowledgment schemes for message passing.

that the client will invoke another operation on the server soon. If no new request is sent, only a small overhead occurs to send the explicit acknowledgment.

The most important difference between the piggybacked and the implicit acknowledgments is how the acknowledgment can be received. With the piggybacked acknowledgment mode, the acknowledgment is always added to a message from the original receiver to the original sender using the message passing module. With the implicit acknowledgment mode, however, the acknowledgment can also be determined from other events. For example, a group message sent by the original receiver could function as an implicit acknowledgment. In Chapter 5, this acknowledgment mode property of the message passing layer will be exploited to reduce the number of acknowledgment messages that is necessary for a given communication pattern on an unreliable network.

If the message passing module is implemented for a system layer that provides reliable communication, the acknowledgment mode parameter is ignored. Therefore, the same interface is provided to the application, and only the implementations that can exploit this mode parameter use it.

4.4 Group communication

The group communication module provides reliable, totally-ordered group communication with one static group that contains all processors that run the current Panda program. Only one static group is provided in this implementation, because it reduces the overhead of group management and demultiplexing. Most applications, including the Orca runtime system and the runtime system presented in this thesis, use only a single group.

Three properties of the system layer determine which implementation can be used: whether the system layer supports arbitrary large messages; whether the system layer already provides a total ordering on the multicast messages; and whether the system layer provides reliability. Again, some combinations are unlikely (e.g., total ordering without reliability).

As in the message passing module, the group communication module uses ports to distinguish messages for different higher-level modules. The ports have to be registered in the same order on all processors by calling *pan_group_register_port*. When a port is registered, a function is passed that will be called with the received message as argument.

All group messages are received in a total order. A total ordering implies that all processors receive all messages in the same order. If processor A and processor B try to send a group message at about the same time, all processors in the group will receive either the message from processor A first, or all processors will receive the message from processor B first. This total ordering property of the group communication module is used in the Orca runtime system to guarantee sequential consistency (see Section 2.1.2).

To send data, the sending processor constructs a message and calls *pan_group_send* with the port number as parameter. This send primitive is asynchronous, so the sending thread immediately continues execution. When the message is ordered with respect to the other messages (the total ordering property), a local upcall is made to deliver this message buffer at the sending processor. At the other processors, the system layer builds a message buffer that contains the message data and delivers this to the handler function.

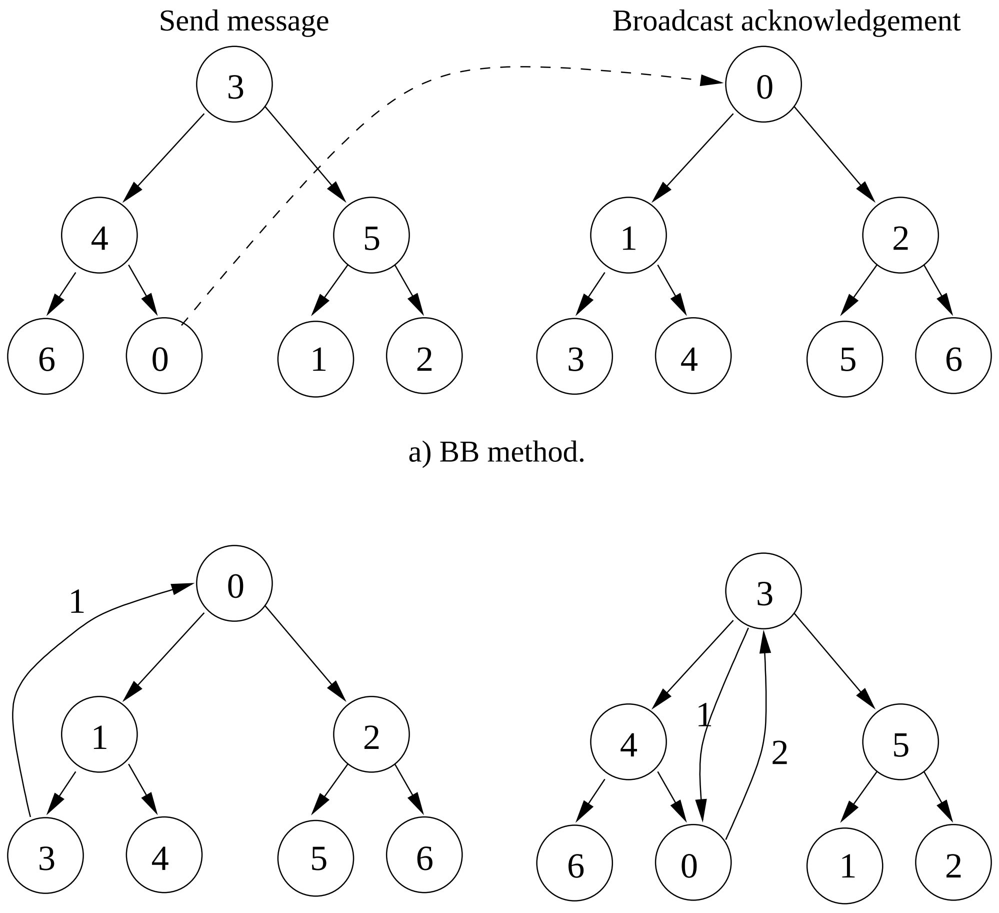
4.4.1 Implementation

Although many machines and operating systems do not directly provide reliable, totally-ordered multicast, this functionality often can be implemented very efficiently directly on top of the operating system. For example, the CM-5 port of Panda uses active messages [121] to implement an efficient spanning tree broadcast protocol. If this implementation would use the generic Panda group communication module, performance would be much lower. Therefore, the CM-5 system layer provides reliable, totally-ordered multicast and uses a very lightweight Panda group communication module that only provides the right interface. Being able to tradeoff between an easy port (using the existing group communication module that provides total ordering) and more performance (implementing the total ordering in the system layer) is an example of the flexibility provided by Panda.

If the system layer does not supports arbitrary large messages, the group communication module splits the message in fragments. Since the fragments are sent in a total order, the order of the last fragment determines the order of the whole message. For example, consider the case where processor 1 starts to send a large group message, and processor 2 later sends a small group message consisting of a single fragment. The first fragment from processor 1 will be received first, but the single fragment from processor 2 will be received before the final fragment from processor 1. Since the order of the final fragments determines the order of the messages, the message from processor 2 will be delivered before the message from processor 1 on all processors (including processor 1).

Three schemes have been devised to provide total ordering efficiently on systems that do not support it directly. All schemes use a designated processor (called the *sequencer*) that maintains a *sequence number*. This sequence number is an integer that is incremented by one for each fragment of the group message. Two of these schemes, PB (Point-to-point followed by a Broadcast) and BB (Broadcast followed by a Broadcast of the sequence number), originated from the Amoeba group communication implementation [75]. The third scheme, GSB (Get Sequence number and Broadcast), was originally designed for the Panda port on the Parsytec GCel [64].

Figure 4.5 presents these three methods. In all three pictures, processor 3 sends a broadcast message to all members of the group. The broadcast is presented by using a binary spanning tree, in which each processor forwards to two other processors (except for the leave nodes).



a) BB method.

b) PB method

c) GSB method.

Figure 4.5: The BB, PB, and GSB methods for implementing totally ordered group communication. Processor 0 contains the sequencer.

With the BB method (Figure 4.5a), the sending processor broadcasts the message itself. Before sending the message, it adds an unique key to the message, such as the tuple <processor id, local counter>. When the sequencer receives this message, it broadcasts a small message that contains the key of the original message and the next sequence number. When this sequence message arrives, all processors match the original message with the sequence message, and order the message.

In the PB method (Figure 4.5b), the sending processor sends the message with a point-to-point message to the sequencer (message 1). On receiving this message, the sequencer adds the next sequence number to it, and broadcasts the message to all processors. When a processor receives this broadcast message, it checks whether the message is received in order by looking at the sequence number. If the message arrived in order, it is delivered to the upcall handler function. On the processor that sent the original point-to-point message to the sequencer, the original message is delivered, instead of the copy received from the network.

In the GSB method (Figure 4.5c), the sending processor first sends a small sequence number request message to the sequencer (message 1). The sequencer replies with a small sequence number reply message (message 2). When the sending processor receives the sequence number, it adds it to the original message, and broadcasts the message.

The three ordering methods have different communication characteristics. The PB method is very efficient for small messages, but for large messages the data have to be sent over the network to the sequencer. On network architectures that support broadcasting, such as Ethernet, this implies that the data have to be sent twice over the network, first from the sending processor to the sequencer (point-to-point) and then from the sequencer to all other processors (broadcast). The BB method only broadcasts the data message, but has the problem that all processors receive two messages (the data message and the sequence number). Therefore, the BB method is more appropriate for larger messages, since the overhead to handle the sequence number broadcast message is constant. Finally, the GSB method is more appropriate for architectures that have a low latency to retrieve the sequence number, since it costs one extra message to get the sequence number (the reply message), but it does not have the overhead of sending the data message once too often and it does not cause two receives on all processors.

Implementing total ordering by a designated sequencer aids in implementing reliable communication. When the sequencer hands out a sequence number, it knows the processor that sends the message. If a processor receives a group message with a sequence number that is higher than the sequence number it expected, it can assume that it has missed the previous group message. In that case, it can ask the sequencer about the missing message. In case of the BB and PB methods, the sequencer can keep a copy of the data. This copy can be sent with a point-to-point message to the processor that missed it. In case of the GSB method, the sequencer can forward the retransmission request to the processor that sent the message. This processor can then retransmit the message to the processor that missed it.

4.5 Implementations of Panda

Panda has been implemented on a large number of systems. The first implementation of Panda, Panda 1.0, was built for a network of SPARC workstations running SunOS [31, 107] and connected by Ethernet. The original Panda system interface for messages was based on an abstract data type that allowed the user to push data on the message and to pop data from this message on the receiving processor. This first version of Panda did not support message fragmentation. Furthermore, no message passing module was provided, but only a remote procedure call module.

After the initial implementation on SunOS, a new implementation was written for Solaris. Furthermore, two ports were made for the Amoeba operating system: one that was just a thin layer on top of the Amoeba group communication and remote procedure call primitives, and another port that used a lower-level communication protocol, called FLIP [76]. The performance of these two ports for Amoeba were discussed in [99]. This implementation was based on Panda 2.0, which added support for message fragmentation.

The first port of Panda to a supercomputer was done for the Thinking Machines CM-5 at MIT. This port was built on top of the active messages interface [121]. Therefore, this version was the first port in which the system layer provided reliable communication. Since the CM-5 hardware did not have a broadcast primitive, broadcasting was implemented in the system layer using a spanning tree to forward messages. To improve the efficiency, the total ordering was also provided at the system layer. The communication modules in the interface layer were adapted or new implementations were written to benefit from the increased functionality in the system layer [10].

Around 1996, several fast networks started to become available for workstations. Three of these new technologies were evaluated on the Amoeba cluster: 100 Mbit/s FastEthernet, ATM, and Myrinet. Fast Ethernet is the successor of 10 Mbit/s Ethernet, and is based on the same CSMA/CD protocol (Carrier Sense Multiple Access with Collision Detection [119]). ATM (Asynchronous Transfer Mode) is originally designed for telecommunication, but is also used as LAN for clusters of workstations. Finally, Myrinet is a specially designed network for parallel computing on clusters of workstations [34]. Of these three, Myrinet is the only network that provides reliable communication in hardware.

To do the performance evaluation, Panda ports were written for each of these modern network architectures [14]. During this development, it turned out that the message interface (push and pop) caused too much overhead for these networks. The latest Panda version, Panda 3.0, therefore discarded the message abstraction at the system interface, and only allowed a single buffer to be sent. The only requirement imposed on this buffer is that some space after the user data must be available for the Panda implementation to add a trailer. This space is used by the communication modules to handle messages.

In 1997, four Dutch universities started with the construction of a distributed computer system based on Intel Pentium Pros running BSD/OS from BSDI. This system, called the DAS, consists of a cluster of processors per university, and all processors within a cluster are connected by FastEthernet and Myrinet. The clusters are connected by ATM. Ports of Panda 3.0 have been implemented for all these configurations.

Finally, ports of Panda exist for the Meiko CS-2, the Parsytec GCEL [64], the Parsytec PowerXplorer, the IBM SP2 [39], and Linux [106].

Operating System	**CPU**	**Network**	**Comm. layer**
SunOS	SPARC	Ethernet	UDP
Solaris	SPARC	Ethernet	UDP
Amoeba	SPARC	Ethernet	FLIP
CMOST (CM-5)	SPARC	fat tree	active messages
Meiko CS-2	SPARC	Elan (Fat tree)	UDP sockets
Amoeba	SPARC	FastEthernet	FM
Amoeba	SPARC	ATM	FM
Amoeba	SPARC	Myrinet	FM
Parsytec	T800	Transputer links	Parix
Parsytec	PowerPC	Transputer links	Parix
AIX (SP2)	RS6000	High Performance Switch	MPI
Linux	Intel 80486	Ethernet	UDP
BSD/OS	Pentium Pro	FastEthernet	UDP
BSD/OS	Pentium Pro	Myrinet	FM

Table 4.2: Overview of the systems that Panda has been ported to.

4.6 Performance

All measurements presented in this thesis are performed on the DAS workstation cluster available at the Vrije Universiteit Amsterdam. This system consists of 64 Intel Pentium Pros running at 200 MHz. Each processor contains 64 MB local memory and the processors are connected by Myrinet and FastEthernet. We will discuss the Panda ports to Myrinet and FastEthernet in turn.

The implementation of Panda using the Myrinet communication system is based on the FM communication layer [100]. The Myrinet network interface is mapped into the address space of the process that uses Myrinet. This allows that process to access the device very efficiently, but it does not allow multiple processes on the same machine to share the Myrinet network interface.

The Myrinet network interfaces are connected through high-speed links (the theoretical bandwidth of a link is 153 MB/s) and crossbar switches. The system is configured as a two-dimensional torus.

The Myrinet network provides high reliability at the hardware level [27]. This property has been exploited by FM, which assumes that all packets that a network interface sends will be received by the destination network interface. Reliability at the hardware level, however, needs flow control support to guarantee that receive buffers are available in order to provide reliability at a higher level. Therefore, FM uses a sliding window protocol to guarantee buffer space at the receiving host processor. Finally, the host processor assembles all packets into a message, and delivers this message to the application (in our case, the Panda system layer).

To provide efficient multicast communication, the FM implementation has been extended with multicast support [120]. In this implementation, network interfaces not only send and receive packets, but also forward packets along a binary spanning tree. This improves the performance of multicast communication considerably. Flow control is implemented using a global credit manager, which recollects packet buffers using a rotating token. Another mechanism to provide flow control for multicast communication is described in [28, 29].

In contrast to the Myrinet communication system, FastEthernet has to be shared among all users of a machine. In particular, FastEthernet is used to provide system services, such as remote login and network file system support. Therefore, access to the Fast Ethernet network interface is protected by the kernel; all sends and receives require a context switch. Together with the much lower maximum bandwidth (12.5 MB/s), this makes FastEthernet a less suitable communication system for parallel programming. A major benefit of FastEthernet, however, is its price and its availability. Most standard workstation configurations will be equiped with FastEthernet.

Panda uses the UDP protocol to access the FastEthernet network. Although transmission errors are unlikely under normal load, this protocol does not guarantee reliable communication. Especially under heavy load messages will get lost. Therefore, the Panda protocols implemented on top of Fast Ethernet implement their own reliability scheme. Saving on the number of acknowledgment messages that need to be sent to provide reliability is important to achieve good performance. Performance measurements are presented for configurations up to 12 processors; when more processors are used, performance becomes too unpredictable due to message losses.

In order to relate the performance measurements that will be presented in the remainder of this thesis, we present a number of low-level communication bench-

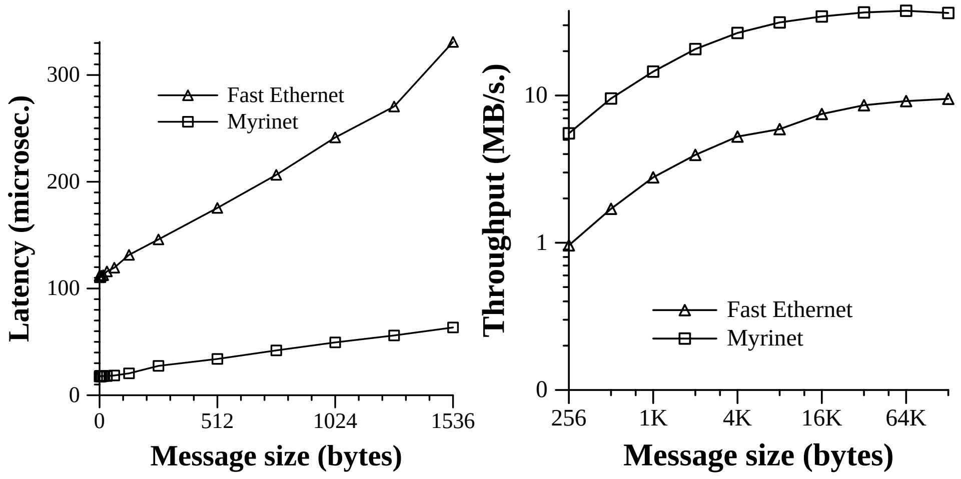


Figure 4.6: Unicast latency and throughput.

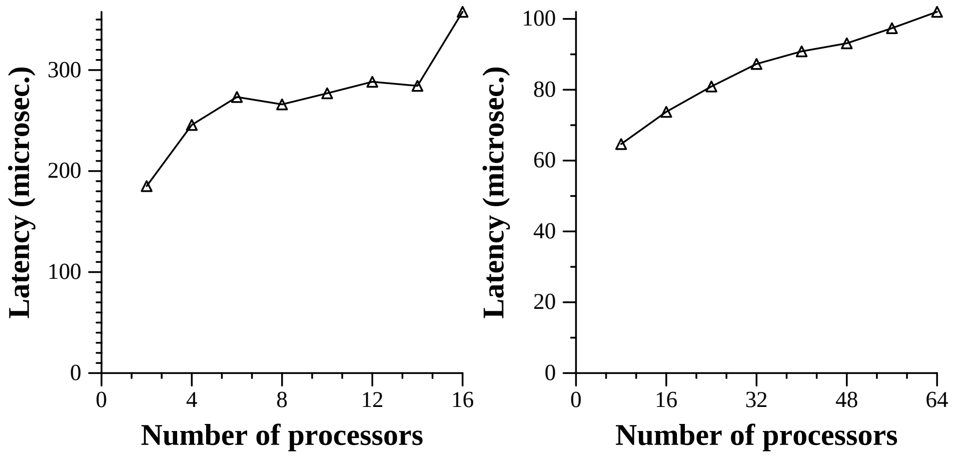


Figure 4.7: Group communication latency on FastEthernet and Myrinet.

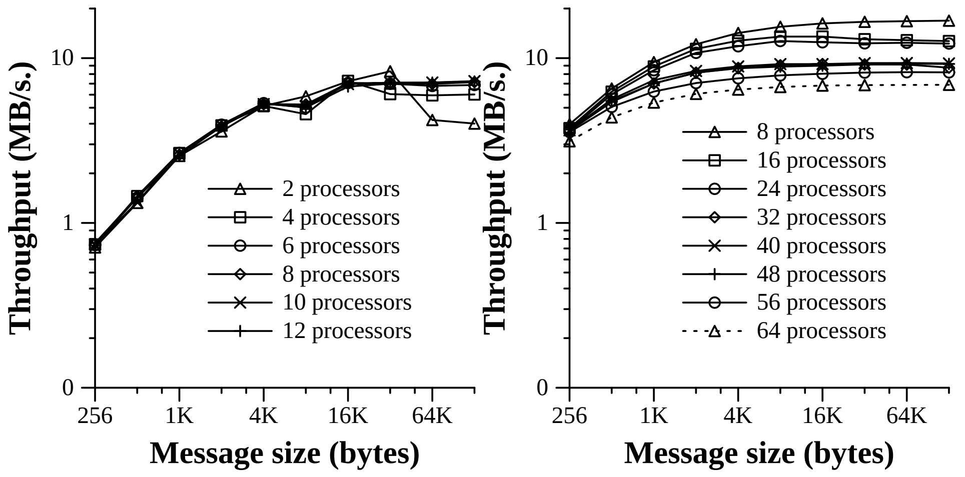


Figure 4.8: Group communication throughput on FastEthernet and Myrinet.

marks results. In particular, we look at the latency and throughput of the message passing and group communication modules. These modules are used extensively in the implementation of collective computation.

Figures 4.6–4.8 present the performance results for the Panda communication primitives. They present latency and throughput numbers for both message passing and group communication. The message passing latency test is performed by sending a ping-pong message between two processors. Based on the total time, the one-way latency is computed. Both throughput measurements are performed by having a single sender blast a number of large messages. After the last message is sent, the sender waits until it has received an acknowledgment from the destination or all destinations. For the group communication throughput, the average throughput is computed with each processor as the source.

Measuring group communication latency, on the other hand, is less trivial. A simple ping-pong test would cause interference between the messages that go to the other destinations. We used the method described in [98] to measure latency. In this method, a single sender sends group messages, and a single destination sends a unicast acknowledgment. This acknowledgment, however, is not sent directly when the group message arrives, but only after a certain delta time has passed. This delta guarantees that the acknowledgment does not interfere with the remaining communication of the group message. The group latency is computed by subtracting the delta and the one-way latency of the unicast message from the measured latency at the sending processor. The figure presents the latency averaged over all combinations of source and destination processors.

Based on these performance results, we can conclude that the Myrinet port of

Panda is better suited to parallel programming than the Fast Ethernet port. For message passing, the Myrinet port has a much lower latency than the FastEthernet port (17.5 versus 112 μs for an empty message) and a much higher bandwidth (37.6 versus 9.5 MB/s. peak throughput). Remarkable, however, is that the throughput of the FastEthernet port is much closer to the theoretical bandwidth (12.5 MB/s); the Myrinet port shows more profoundly the software overhead that is introduced by Panda.

For group communication, there is again a large difference in latency (65 versus 185 μs for a small message). The throughput performance differences are less severe, however. For eight processors, the FastEthernet port reaches a peak throughput of 7.2 MB/s, whereas the Myrinet port reaches 16.9 MB/s. A possible explanation for this more severe degradation in performance of the Myrinet group communication with respect to the message passing throughput is presented in [86]. The Myrinet hardware does not provide support for multicasting; therefore, multicast packets are forwarded along a spanning tree.

Summary

In this section, we have described Panda, a portable layer that is designed for the implementation of runtime support systems for parallel programming. Panda is an integrated system that provides threads, message passing, and group communication. By splitting Panda into two layers, a system layer and an interface layer, the amount of code that needs to be adapted to port Panda to a new platform is small. Only the system layer needs to be adapted, while the interface layer can be reused on different platforms.

To achieve good performance, it is very important to have a tight match between the support provided by the underlying operating system and the implementation of the communication primitives. Therefore, the Panda system layer provides a fixed interface, but the semantics of the operations of the system layer may differ in a restricted way from platform to platform, depending on the support of the operating system. For example, if the operating system provides a reliable communication primitive, the system layer also provides reliable communication; otherwise, the system layer provides unreliable communication.

The interface layer contains different implementations of a communication module, and each implementation is targeted to the semantics of the system layer. For example, if the system layer provides reliable communication, the message passing layer does not implement its own retransmission protocol. The match between the different implementations in the interface layer and the system layer is made during the configuration of Panda. This way, we guarantee that the communication primitives provided by Panda are efficient.

Finally, the interface of Panda evolved during the years to allow the implementation of efficient applications. In particular, the message interface has changed drastically, from a high-level abstract data type that allowed push and pop operation to a low-level single buffer. By specifying the contract that the user is responsible to reserve additional space for trailer information, good performance can be achieved. Another example is the acknowledgment optimization, which allows Panda to exploit application-level knowledge of the communication pattern in which a message occurs to reduce acknowledgment communication (e.g., as in the remote procedure call module).

Chapter 5

High-level Communication Primitives

In this chapter, we will present a high-level communication module that is based on collective computation. The main idea of this communication library is to allow the runtime system to exploit the global knowledge that we have available in a collective computation. All processors know what the other processors are doing, and all processors also know what data each processor has to receive to be able to continue its execution of the collective computation function. In addition, all processors involved in the collective computation make the same calls with the same arguments to the communication library. We will show that we can exploit this knowledge to optimize the communication pattern.

Important for the performance of parallel programming systems is the number of messages that have to be sent. Knowledge of the communication pattern can be used to improve the performance by sending fewer acknowledgment messages [71]. A simple example is a remote procedure call implementation, in which it is possible to exploit the fact that the communication is a request-reply pair. Instead of explicitly acknowledging each message, it uses piggybacked acknowledgments (see Section 4.3). In most applications (where multiple request-reply pairs are issued in a short time), this reduces the communication to two messages instead of the four messages that would be needed if every message was treated separately (i.e., a request and a reply message and two acknowledgments).

Remote procedure call is an example in which the communication pattern is known and where this pattern can be used to optimize the number of messages that have to be sent. In this chapter, a mechanism will be presented to describe and optimize *arbitrary* communication patterns, which contain both unicast and multicast messages. This mechanism is based on an abstraction, called *communication schedules*, that is used to describe the communication pattern. A library implementation of this abstraction is built on top of Panda (see Figure 5.1).

We distinguish two phases of a communication pattern: describing the pattern and performing the actual communication of a pattern. This distinction allows us to reuse communication patterns during the execution of an application. For example, to assess the usability and performance of this approach, a set of MPI collective communication operations has been implemented using communication schedules. The communication schedules that this collective communication

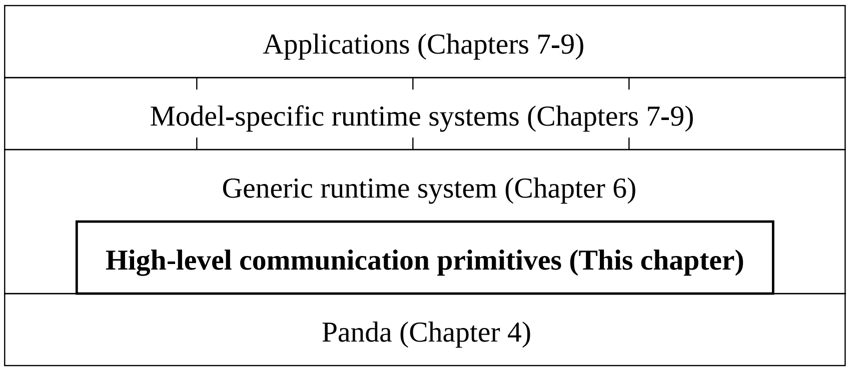

Applications (Chapters 7-9)
Model-specific runtime systems (Chapters 7-9)
Generic runtime system (Chapter 6)
High-level communication primitives (This chapter)
Panda (Chapter 4)

Figure 5.1: The high-level communication library in the architecture overview.

implementation uses are generated at initialization time, and are used for each execution of a collective operation. A performance evaluation of this MPI implementation is presented in this chapter.

For collective communication operations, the communication pattern and the number of processors are known when an application is started. Therefore, we can generate the communication schedules for the collective communication operations during initialization and use them during the remainder of the execution of the application.

After the communication pattern is described in the communication schedule, optimizations can be applied to it. We have designed and implemented two such optimizations. First, by analyzing the communication schedule, the number of acknowledgment messages can be reduced. Second, the library implementation uses the information in the communication schedule to handle incoming messages without incurring context-switching overhead.

A communication schedule that is defined and optimized can be *executed*. Executing a communication schedule involves passing the data and the schedule to the communication runtime system. This communication system will perform the send and receive operations as specified in the schedule, using the data passed as argument to fill the messages.

Generating the optimal communication pattern for a specific architecture depends on the characterists of the underlying communication system. In our case, the communication library is implemented on top of Panda. A set of microbenchmarks is used to obtain the LogGP performance measures of the specific architecture. The LogGP results are used to generate optimized communication patterns.

With nested objects and atomic functions, data dependencies are only known at invocation time. Therefore, the generic runtime system will generate and exe-

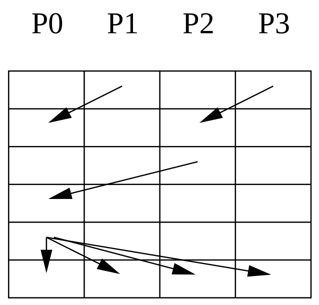


Figure 5.2: Communication pattern for a barrier on 4 processors.

cute a communication schedule when the processors that participate in a weaver have to communicate. The same optimizations are applied on this communication schedule as on the communication schedules for the MPI primitives. We present the conversion from operation invocations and data dependencies to communication schedules in Chapter 6.

In Section 5.1, communication schedules are described. Section 5.2 describes the execution of communication schedules on top of Panda and Section 5.3 describes how the two optimizations are implemented. Section 5.4 describes how to implement optimized collective communication operations on top of the communication schedule library using the LogGP performance numbers. Finally, Section 5.5 gives a performance evaluation of this implementation of collective communication and our optimizations.

5.1 Communication schedules

A communication schedule is a complete description of a communication pattern. Figure 5.2 shows an example of a communication pattern. This communication pattern describes a barrier operation (see Section 3.3) on four processors (a column per processor). The barrier synchronizes all processors using a tree with the root at processor 0. When a processor arrives at the barrier, it starts to execute its part of the communication schedule (i.e., all actions in the column that belongs to this processor). Processor 1 and processor 3 first send a message to processor 0 and processor 2, respectively. After processor 2 has received the message from processor 3, it sends a message to processor 0. When both messages have arrived at processor 0, it sends a broadcast message to all processors. After a processor has received this broadcast message the communication schedule is finished, and therefore the barrier operation is also finished on this processor.

A communication schedule consists of actions that describe the complete communication pattern. Each action consists of the type of action (send or receive), the processor that performs the action, and a description of the data buffer it should use. In the barrier example, each unicast message (e.g., the message from processor 1 to processor 0) consists of a send action (outgoing arrow) and a matching receive action (incoming arrow). The broadcast message (from processor 0) consists of a single send action and four matching receive actions.

A communication schedule is represented as an abstract data type. To create a communication schedule, the function *pan_cc_sched_create* is provided, which returns an empty schedule for a given set of processors. This set of processors indicates the processors that have to execute the communication schedule at run time. Three functions are provided to fill in a communication schedule. Each of these functions inserts an action for a specific processor in the communication schedule:

int pan_cc_sched_send(sched, pid, data) Send the data specified by the given data descriptor. This function returns a send handle, which is used to register corresponding receive actions.

void pan_cc_sched_receive(sched, pid, send handle) Receive the data that corresponds to the send action that returned the send handle. The data is received using the same data descriptor as the send action.

void pan_cc_sched_compute(sched, pid, function) Execute `function` after performing all preceding actions on this processor.

A communication schedule can be used multiple times in an application. To allow a communication schedule to operate on arbitrary data buffers, an indirection is used to access the data. When an action is inserted in a communication schedule, it does not store a pointer to the data, but only an index into a data description table. When the communication schedule is executed, it gets as argument a *data descriptor table*, which contains pointers to the actual data and functions to prepare a message buffer and a receive function to handle the message. If no prepare and receive functions are registered, a memory copy is used.

Pan_cc_sched_send returns a send handle, which is used to register *pan_cc_sched_receive* operations. This way, no cyclic dependencies can be introduced. Figure 5.3 illustrates the creation of the communication schedule for the barrier example in Figure 5.2

5.2 Communication schedule execution

The communication schedule library is built on top of the Panda communication modules presented in Chapter 4. Each unicast send action uses the message pass-

```
pan_cc_sched_p build(void)
{
  pan_cc_sched_p sched = pan_cc_sched_create(
);
  int sid;

  sid = pan_cc_sched_send(sched, 1, 0);
  pan_cc_sched_receive(sched, 0, sid);
  sid = pan_cc_sched_send(sched, 3, 0);
  pan_cc_sched_receive(sched, 2, sid);
  sid = pan_cc_sched_send(sched, 2, 0);
  pan_cc_sched_receive(sched, 0, sid);

  sid = pan_cc_sched_send(sched, 0, 0);
  pan_cc_sched_receive(sched, 0, sid);
  pan_cc_sched_receive(sched, 1, sid);
  pan_cc_sched_receive(sched, 2, sid);
  pan_cc_sched_receive(sched, 3, sid);

  return sched;
}
```

Figure 5.3: Creation of the communication schedule for the barrier example in Figure 5.2.

ing module, while the multicast messages use the group communication module. The communication schedule library therefore registers a port with each of these modules. Using this port mechanism, it is possible to combine the communication schedule library with other communication libraries without them being aware of each other.

The execution of a communication schedule consists of performing all actions of a processor in the order they are specified in the schedule. For a send action, the prepare function is retrieved from the data descriptor and this function is then used to generate the message. If only a single receive action was registered for the send action, the message passing module is used. If multiple receive actions were registered (on different processors), the group module is used.

A receive action blocks until the message arrives. When the message arrives, the message passing or the group communication module makes an upcall to the communication schedule library, with the message buffer as argument. To determine how this message must be handled, the communication schedule uses the data descriptor of the corresponding send action to find the receive function. Since this receive function can perform arbitrary code (e.g., a reduction operation), messages have to be delivered in order, even if they arrive out of order. Therefore, messages are buffered until the receive action is scheduled.

To handle multicast messages, a processor has to determine whether it has a matching receive action. Therefore, each processor stores in its local version of the communication schedule whether the corresponding send action is matched by a local receive action or not.

All processors that want to execute a communication schedule build a data description table that contains an entry for each data descriptor that is used in the schedule. Each data descriptor consists of a pointer to the actual data and a set of pointers to a function to prepare the data for sending and for receiving the data. This data description table is passed as argument to *pan_cc_sched_exec*, which executes the given schedule.

To illustrate the use of data descriptors, we give a simple communication schedule for a remote handler invocation from processor 0 to processor 1 (see Figure 5.4). The request (initiated by processor 0) uses data descriptor 0, while the reply uses data descriptor 1 (the descriptor index is the third argument of the function *pan_cc_sched_send*). Before executing the schedule, both processors initialize a data description table. Data descriptor 0 contains functions to marshal and unmarshal the request, while data descriptor 2 contains these functions for the reply.

Before the communication schedule is executed, both processors build an argument vector, which contains the arguments for the handler function. A pointer to this argument vector is stored in the two data descriptors on both processors. On processor 0, the input arguments are also filled in. When the communication

```
sched = pan_cc_sched_create( );
req_id = pan_cc_sched_send(sched, 0, 0);
pan_cc_sched_receive(sched, 1, req_id);
pan_cc_sched_compute(sched, 1, handler);
rep_id = pan_cc_sched_send(sched, 1, 1);
pan_cc_sched_receive(sched, 0, rep_id);
```

Figure 5.4: Communication schedule for a remote procedure call.

schedule is executed, the request send action marshals all input arguments and sends the resulting message to processor 1. On processor 1, the receive action unmarshals the input arguments into the argument vector. The compute action invokes the handler function on processor 1, which performs the required computations. This handler function gets the data descriptor table as argument, so it can access the argument vector. The reply send and receive actions, then, send the output arguments back to processor 0.

An important property of communication schedules is that it must be possible to select a collection of processors that execute the communication schedule (all processors for which actions are registered must participate). In addition, multiple communication schedules can be invoked concurrently by different collections of processors. As we will present in Chapter 6, communication schedules can be built and executed during the execution of an atomic function. Since multiple atomic functions can be invoked in parallel, the communication schedule library must be able to deal with this.

When multiple communication schedules are executed in parallel, messages from each execution may arrive at a processor. Therefore, when a message arrives, we need to determine to which communication schedule execution it belongs. To distinguish the schedule executions, a unique tag is generated for each execution. When a communication schedule is executed, each participating processor fetches a sequence number for the collection of processors that execute this schedule and attaches this sequence number to each message it sends in the schedule. Since we do not want all processors to send a sequence number request before every communication schedule execution, the management of these sequence numbers is replicated on all processors.

Whenever a collection of processors executes a communication schedule, the lowest numbered processor sends a sequence number request to a central manager (e.g., processor 0). This manager broadcasts a range of sequence numbers (e.g., 1000 sequence numbers) for that specific collection of processors. All other processors of the collection block until this broadcast arrives. After the first operation, a collection of processors can fetch the next sequence number without any communication, until the range of sequence numbers is exhausted. When this occurs, the same process is repeated.

When another collection of processors starts the execution of a communication schedule, the central manager will hand out the next range of sequence numbers. Since all processors keep track of the distribution of sequence numbers, a sequence number can also serve as an indication whether a broadcast message is destined for a local schedule or not. If the sequence number indicates a collection of processors that does not contain the local processor identifier, it can be discarded. When the message is not discarded, the communication schedule is queried to determine whether a receive action is posted for this message or not.

An important aspect of using communication schedules is that the complete communication pattern is known on all processors before the communication occurs. This allows the implementation to perform a number of optimizations, two of which are described in the next section.

5.3 Optimizations

After the communication schedule has been generated, we can use the information in it to optimize its execution. Since a communication schedule is generated before it is executed, these optimizations also take place before the execution. In this section we will discuss two optimizations that we have developed that make use of this information: implicit acknowledgments and continuations. The first optimization reduces the number of acknowledgment messages; the second eliminates context switching during data forwarding. The performance improvements obtained from these optimizations will be presented in Section 5.5.

5.3.1 Implicit acknowledgments

The Panda message passing module provides three acknowledgment modes to reduce acknowledgment traffic (see Section 4.3). We want to exploit this property to reduce the number of acknowledgment messages in collective communication operations. Figure 5.2 shows, however, that the use of the piggybacked acknowledgment mode does not gain anything in this case. The group communication module and the message passing module are independent. Since the group communication module sends the final broadcast message, the message passing module will not be aware of this message. Therefore, an explicit acknowledgment will be sent.

We would like to exploit the fact that the final broadcast message can function as an implicit acknowledgment for all messages in the spanning tree. To determine which unicast messages are implicitly acknowledged, the communication pattern of the schedule is analyzed after it has been created. For each receive action, all *causally preceding* sends originating from the receiving processor are implicitly acknowledged by this message. This information is stored in the communication schedule, so that the best mode in which a unicast message can be sent is known during the execution of the schedule. In the example in Figure 5.2, all unicast messages are sent in implicit acknowledgment mode and all are acknowledged when the broadcast arrives.

The precedence relation is defined as follows. First, each receive action is preceded by the corresponding send action. Second, a send action is preceded by all receive actions that occur earlier on the same processor. Finally, the transitive closure of these two definitions is taken.

When a communication schedule is executed by a processor, all actions in the communication schedule for that processor are executed in order. The information stored in the communication schedule determines the acknowledgment mode in which unicast messages are sent. Therefore, the number of messages is minimal (no spurious acknowledgments) if no messages are lost. When a message is lost, however, no implicit acknowledgments will arrive, and more than one message may be retransmitted by the message passing module. In the barrier example, if any of the messages within the spanning tree would be lost, the final group message will not be sent. Therefore, all messages in the spanning tree that have already been sent would be retransmitted.

To solve this spurious retransmission problem, an extra function has been implemented in the message passing module that allows the receiver to specify that it expects a message. If the message is not received within a short amount of time, a request for retransmission is sent. In the communication schedule library, this function is invoked when a receive action is scheduled and the message has not arrived yet. If a message gets lost, a request for retransmission will be sent before the send timeout occurs. If this retransmission succeeds, the communication schedule can continue without retransmission of any other messages [20].

Optimizing implicit acknowledgments can improve the performance of a communication schedule when the schedule already contains a communication pattern that allows this optimization. It is also possible, however, that extra empty messages are added to a schedule to enforce this optimization. For example, the communication schedule of Figure 5.2 can also be used for a reduction to processor 0. In this case, the final broadcast message would not be required, but by leaving it in the communication schedule, the implicit acknowledgment optimization can be exploited. Another benefit of the final broadcast message is that it enforces flow control. A sequence of reduction operation executions will not overflow one of the receivers, because every reduction will be terminated before the next reduction can start.

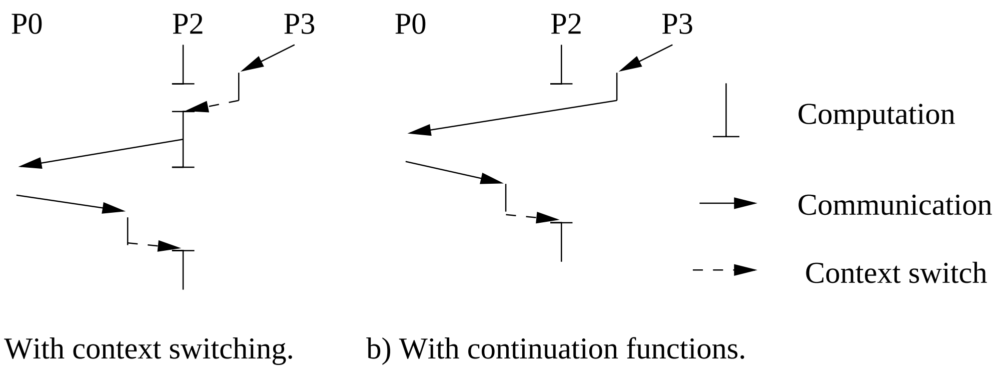


Figure 5.5: Execution on processor 2 of the barrier implementation.

5.3.2 Continuations

The second optimization reduces the number of context switches that occur during the execution of a communication schedule. One way to implement this execution is to let the user thread that calls the operation perform all its actions. To receive a message the user thread blocks on a condition variable until the message arrives. When a message arrives, the upcall handler has to wake up the user thread, which results in a context switch for every message. In Figure 5.5a we show this execution behavior for the barrier implementation on processor 2. When the message from processor 3 arrives, the handler function wakes up the user thread. The user thread then sends a message to processor 0 and blocks until it receives the broadcast message.

A more efficient way is to let the upcall handler perform all the actions that are caused by the receipt of a message. The user thread starts to execute its actions in the communication schedule up to the first receive action. Then, the user thread leaves some state information, a *continuation*, that represents the remainder of the schedule to be executed [54]. Note that it is not possible to block the upcall thread, since this would block the reception of all other messages, including the message that the upcall thread is waiting for (see Section 4.1).

When the message that corresponds to the receive action arrives, the upcall thread uses this state to perform (part of) the work that would normally be performed by the user thread. If the upcall thread reached a receive action for which the corresponding message has not arrived yet, a (new) continuation is created, and the upcall terminates. When the communication schedule is finished the user thread is signaled.

Figure 5.5b again shows the behavior for the barrier implementation on processor 2. In this case, however, the handler function that handles the message from processor 3 calls a continuation function that performs all actions up to and including the send to processor 0.

Implementing continuations by hand is hard and error prone. Most systems that provide continuations (such as functional languages) depend on compiler support. An exception is the work presented in [26], where hand-coded continuations are added to the Orca runtime systems to execute operations on shared objects from within the upcall handler. In our system, the communication schedule contains most of the state that has to be preserved, such as what to do with the message data and what to do after a message is received. The only functions that need to be written as continuation function are inside the core library for the execution of communication schedules. This implies that all operations implemented on top of communication schedules, including our collective communication library, benefit from continuation functions without any additional coding.

5.4 Collective communication

To evaluate the usability and performance of communication schedules, we implemented a set of collective communication operations on top of the communication schedule library [111]. We use this collective communication library to perform a number of performance evaluation experiments.

An important aspect of writing a collective communication library for a portable system like Panda is that the library should be easily adaptable to the different machine architecture characteristics. A model that captures such characteristics is the *LogP* model [50]. This model consists of four parameters:

Latency (L)	An upper bound on the delay associated with delivering a message from its source to its destination.
overhead (o)	The time period during which the processor is engaged in sending or receiving a message. Often, the overhead parameter is split into send overhead (o_s) and receive overhead (o_r).
gap (g)	The minimum time interval between consecutive message transmissions or consecutive message receptions at a processor.
Processors (P)	The number of processors.

An extension of this model, called *LogGP*, also incorporates the effective bandwidth for bulk transfer, where G is defined as the reciprocal of the bandwidth [5].

Using the five parameters of the LogGP model, optimal communication patterns can be computed for reduction and gather trees [5, 24, 77]. For our collective communication library, we implemented generator routines that compute such trees for arbitrary processor sets and message sizes. Based on such trees, the

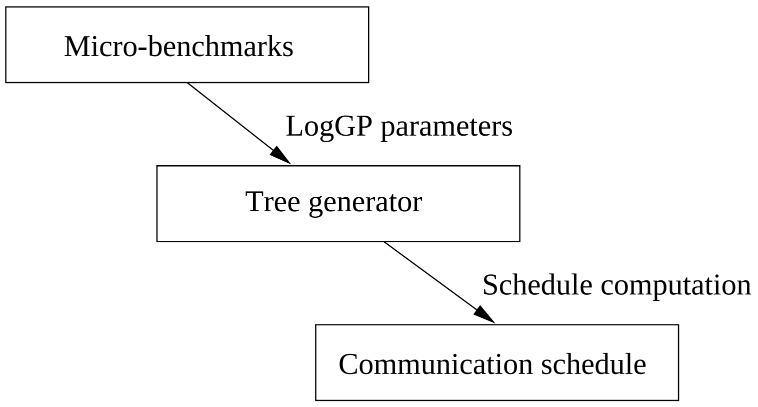


Figure 5.6: Using the LogGP parameters for the generation of communication schedules.

corresponding communication schedules can be computed (see Figure 5.6). The benefits of this approach are twofold. First, the LogGP parameters are the only architecture-dependent information that is required. Second, it allows us to generate an appropriate communication schedule on demand, which reduces overall memory consumption by the communication schedules.

To compute the LogGP parameters for a specific Panda platform, two micro-benchmarks are used. Together with the results of a round-trip latency and a throughput benchmark, these micro-benchmarks determine the LogGP parameters [69].

The round-trip time measured with the latency test gives the quantity $2(o_s + L + o_r)$. The first micro-benchmark measures the time needed to send N messages of size M. For small values of N, the completion time of this benchmark divided by N gives the send overhead (o_s). As N increases, the communication pipeline approaches its steady state, and the completion time of the benchmark divided by N corresponds to the gap.

The second micro-benchmark measures the round-trip time when a delay of Δ μs is inserted between the request and the receipt of corresponding reply (see Figure 5.7). When Δ is smaller than $L + o_r + o_s + L$ (i.e., the amount of time in which the request reaches the other processor and the reply gets back), the reply message arrives when the delay has passed. Therefore, the completion time of the micro-benchmark corresponds to the round-trip time (i.e., $2(o_s + L + o_r)$). For larger values of Δ, however, the completion time becomes $o_s + \Delta + o_r$, because the reply arrives while the sender is waiting for the delay to pass. Based on the (known) value of Δ, the quantity $o_s + o_r$ can be computed. Together with the results from the first micro-benchmark the receive overhead can be derived.

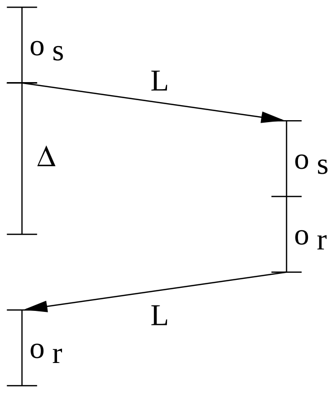


Figure 5.7: Obtaining the send and receive overhead using a delay Δ at the sending side.

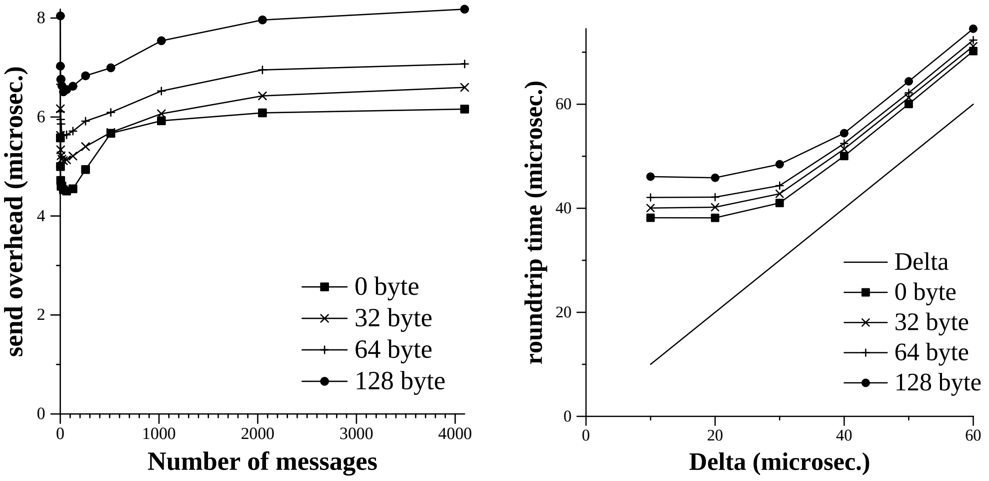


Figure 5.8: LogGP micro-benchmark results for the Myrinet/FM Panda port.

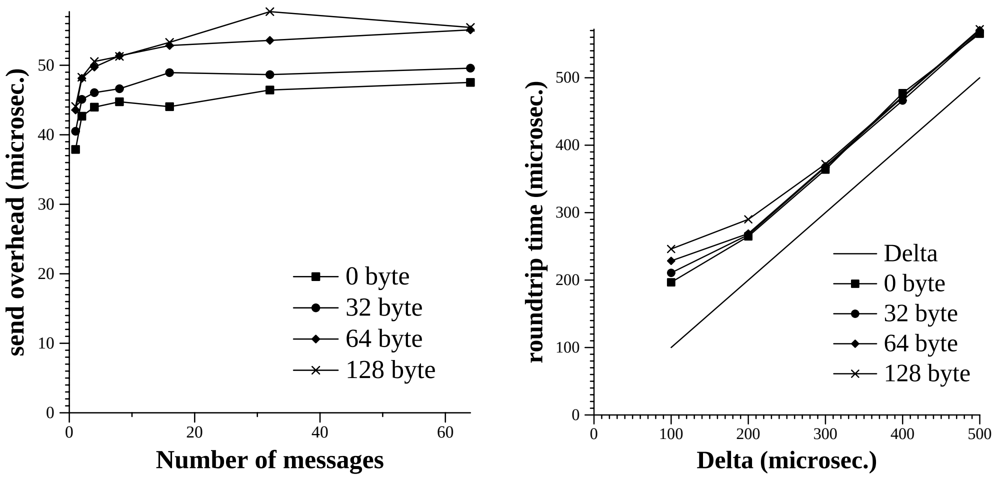


Figure 5.9: LogGP micro-benchmark results for the FastEthernet Panda port.

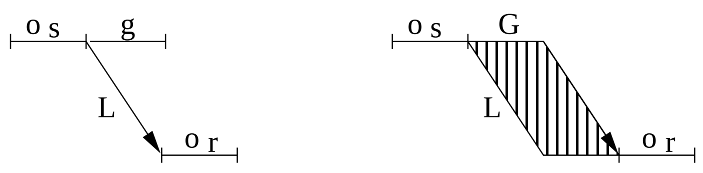
Figure 5.8 and 5.9 present the benchmark results for the Myrinet and the FastEthernet port, respectively. Based on these numbers, and on the results of the latency and throughput benchmarks, we can derive the results presented in Table 5.1.

Parameter	**Myrinet port**	**FastEthernet port**
Latency (L)	8.4 μs	32 μs
Send overhead (o_s)	4.5 μs	39 μs
Receive overhead (o_r)	5.7 μs	37 μs
Gap (g)	6.1 μs	47 μs
Bandwidth (G)	27 MB/s	9 MB/s

Table 5.1: Results of the LogGP micro-benchmarks.

Three issues are important to consider. First, the gap defines the maximum rate of the communication pipeline; therefore, it must always be larger than (or be as large as) the send overhead as well as the receive overhead.

Second, the bandwidth of the Myrinet port is lower than the maximum bandwidth that is achieved using Panda's message passing module (see Section 4.6 on page 77). This difference is caused by the copy at the receiver. In the unicast test for the message passing module the data is not copied out of the message. The message can be preserved after the upcall terminates, so most applications do not need a copy. For collective communication, on the other hand, we need at least one copy to get the data to the user buffer. Therefore, our benchmark also makes this copy.



a) Small message. b) Large message.

Figure 5.10: Using the LogGP model to predict the communication time for small and large messages.

Finally, the LogGP model assumes a fully-connected network, whereas the FastEthernet is a bus-based system[1]. Therefore, the bandwidth that is achieved gives an upper limit on the bandwidth that is available during more complex communication patterns. We do not take this into account for the communication schedules that we generate.

To illustrate the use of the LogGP parameters in generating communication schedules for collective communication primitives, we first describe how to build an optimal broadcast tree using only unicast operations. In a broadcast tree, one node (the root node) has some data that must be sent to all other nodes. The idea is that a node that receives the data must forward it as soon as possible. The root starts the broadcast at time 0, and after o_s time units the data enters the network. After L more time units, the data starts to arrive at the receiving node, which will complete the receive after o_r time units have passed (see Figure 5.10a). From that moment, the receiving node can forward the data to its children. The original root node can send the next message after g time units.

If the data is large, the bandwidth (G) of the network starts to influence the time when the data fully arrives at the receiving node. If the amount of time that a message is in transit is larger than the gap, we use this number instead of the gap (see Figure 5.10b). This way, the rate with which the sender is scheduled to send large messages is controlled, and the time at which the full message is received is delayed.

In the communication schedules that we create, we use the Panda group communication module for broadcasting instead of a broadcast tree. Based on the principles of broadcast tree generation, however, we can also generate a reduction tree. During the generation of such a reduction tree, the roles of the send and receive overhead are reversed. The resulting tree gives an optimal communication pattern for performing the reduction operation. The same principles apply to the

[1]The switches can be regarded as a fully-connected network, but they cover only part of the total network.

generation of scatter and gather trees [5].

Although the generators can be used to generate a specialized communication pattern on demand, the generation of this tree and the corresponding communication schedule can take more time than the performance gains. Therefore, we cache a communication schedule whenever it has been generated for the first time. The optimal communication pattern, however, can differ for different message sizes. To increase the chance that a reasonably similar communication schedule will be reused, our collective communication library generates for an operation one communication schedule for a range of sizes.

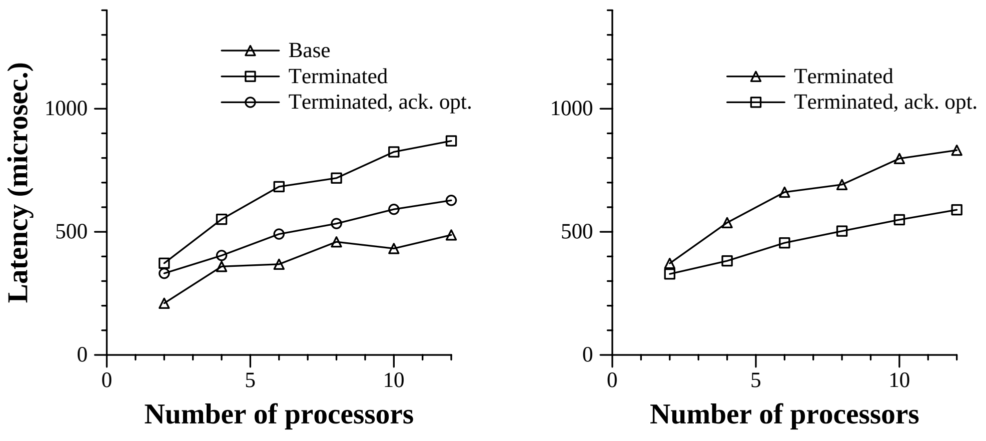
5.5 Performance assessment

In this section, we will look at the benefits of using communication schedules to implement collective communication operations. First, we will look at the benefits of the optimizations discussed in Section 5.3. Second, we will illustrate the benefits of using the LogGP model for building a portable collective communication library.

To determine the performance improvement caused by our optimizations, we used three versions of the communication schedule execution code. The first implementation performs none of our optimizations, the second uses only the implicit acknowledgment, and the third applies both implicit acknowledgments and continuation functions. Since the acknowledgment optimization is only beneficial for Panda ports that are based on unreliable communication, we performed some of the measurements on the FastEthernet port. For this experiment we used the reduction operation on small data items (i.e., one integer).

As described in Section 5.3.1, the communication schedule for a reduction operation can apply a termination broadcast for two purposes: to enable the implicit acknowledgment optimization and to provide flow control. Our first experiment performs a sequence of reduction operations in which the destination processor of the reduction cycles through all processors (i.e., the i^{th} iteration has processor $i \bmod P$ as its destination, where P is the number of processors). Figure 5.11a presents the results of this experiment. The base curve presents the average reduce operation latency without a termination message. The other two curves use an additional termination message, one with and one without the acknowledgment optimization. As the results illustrate, the termination broadcast message takes a significant amount of time, but when it has to be used, the acknowledgment optimization saves considerably (up to 30 percent).

In general it is not possible to do without the flow control that the termination message provides. Our second experiment also performs a sequence of reduction operations, but now the destination processor alternates between processor 0 and



a) Cyclic root. b) Alternating root.

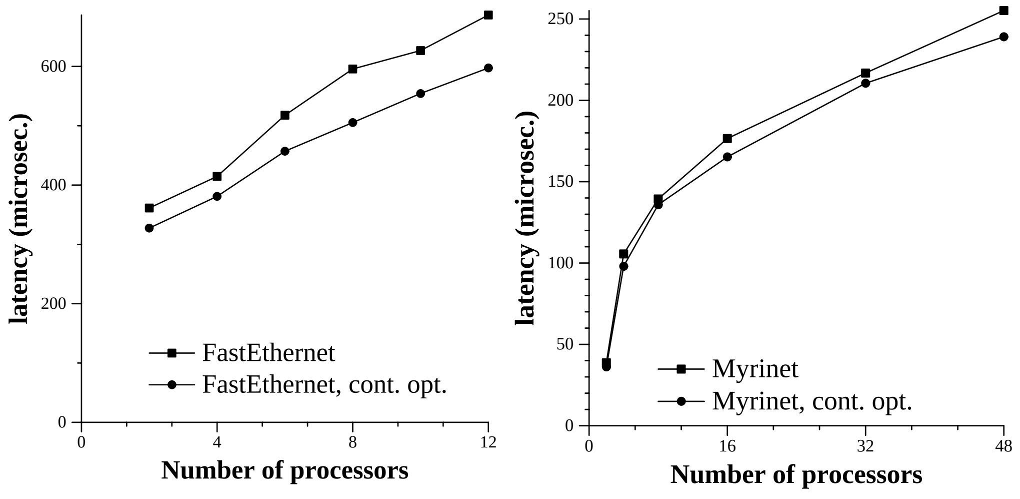
Figure 5.11: Results for the acknowledgment optimization experiments on FastEthernet.

processor $P-1$. Since all others processors do not receive any messages if no termination message is used, there is no high-level flow control. This causes some of the processors to receive messages faster than they can handle, which finally leads to memory exhaustion. Figure 5.11b presents the results of this benchmark. Based on these results, we decided to make the termination broadcast the default behavior for reduction operations.

Figure 5.12 shows the performance results of the default reduction operation (i.e., with the termination broadcast message and the acknowledgment optimization) with and without the continuation optimization. For the FastEthernet port, the continuation optimization results in up to 13 percent performance improvement. For the Myrinet port, in which the communication is better integrated with the thread package, the continuation optimization gives only up to 7 percent performance gain.

In the last experiment, we evaluate the use of the LogGP parameters for generating the communication schedules. In this experiment, we perform a reduction on a range of data sizes. Figure 5.13 shows the results for the optimal tree generated using the LogGP parameters, as well as the results for the binomial tree, the binary tree (i.e., all nonleaf nodes receive two messages), and the flat tree (i.e., the root node receives a message from all other processors).

For small data sizes, the communication pattern generated using the LogGP parameters outperforms the binomial tree by up to 10 percent (see Figure 5.13a). All other trees perform consistently worse than both the binomial and the LogGP tree. For larger data sizes the LogGP tree and the binomial tree give the same per-



a) FastEthernet port. a) Myrinet port.

Figure 5.12: Results for the continuation optimization experiment.

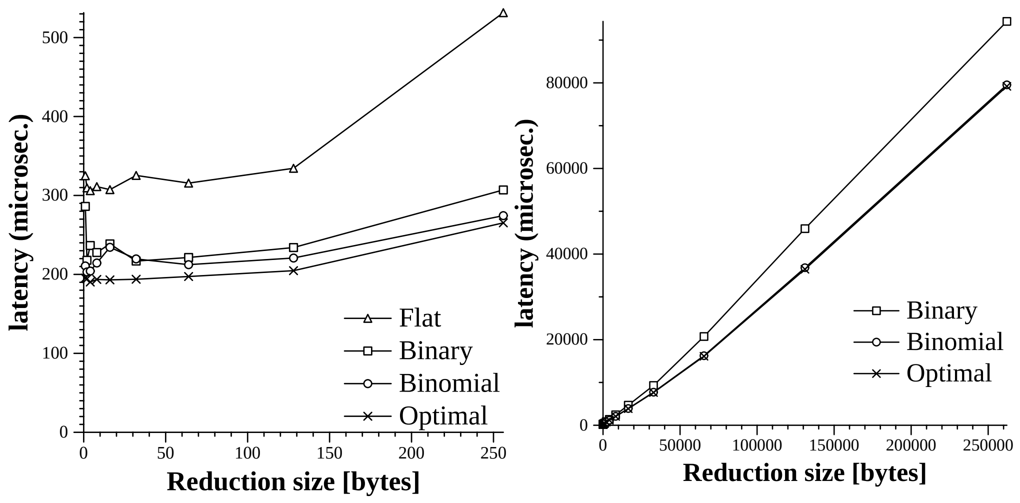


Figure 5.13: Results for the LogGP evaluation experiment on the Myrinet/FM Panda port using 32 processors.

formance. This behavior can be explained using Figure 5.10b; for large messages, the G parameter starts to dominate all other parameters, since the G parameter is the only one that takes the message size into account. When the message is sent, the receive is also almost finished (with respect to the total communication time). Therefore, the LogGP tree has the same shape as the binomial tree.

In conclusion, the optimizations that we have implemented have a positive influence on the performance of the communication schedules. Since the optimizations are hidden in the communication library, they apply automatically to all communication schedules. Using the LogGP model we can generate efficient communication schedules for collective communication operations in an architecture-independent manner.

Summary

In this chapter, we have described a flexible mechanism for describing communication patterns. By describing the communication pattern before it actually takes place, we can perform a number of optimizations. We have described two such optimizations, implicit acknowledgments and continuations, and investigated their effects on the performance.

For the performance evaluation, we have implemented a set of MPI primitives on top of the communication schedules. To keep this collective communication library as architecture independent as possible, we use the LogGP model to characterize the underlying Panda implementation. These architecture parameters can easily be obtained by running a set of micro-benchmarks. Based on these LogGP parameters, we can generate suitable communication trees. The performance improvement that is obtained by using the LogGP parameters instead of a generic reduction tree (e.g., the binomial spanning tree), are also presented.

Chapter 6

Generic Run Time Support

In this chapter, we present the generic runtime system, which provides the basic primitives to build the model-specific runtime systems for shared data objects, nested objects, and atomic functions (see Figure 6.1).

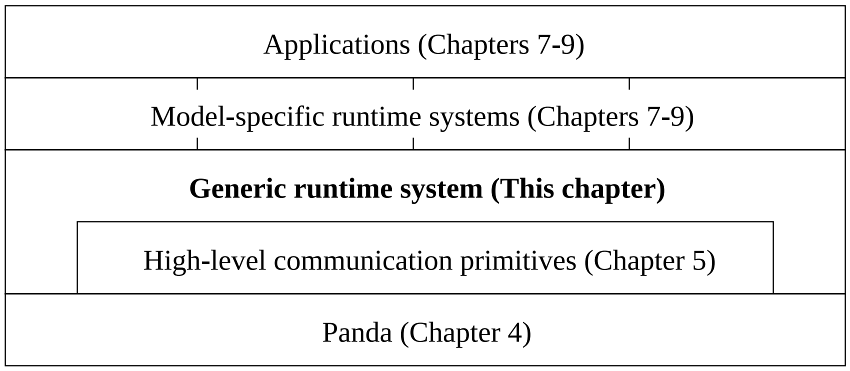

Applications (Chapters 7-9)
Model-specific runtime systems (Chapters 7-9)
Generic runtime system (This chapter)
High-level communication primitives (Chapter 5)
Panda (Chapter 4)

Figure 6.1: The generic runtime system in the architecture overview.

We can identify the following requirements for the generic runtime system:

- Provide support for the parallel language constructs; in particular, the runtime system has to support objects (local, remote, or replicated) and lightweight processes (local process creation and remote process creation.)
- Provide support for operations on a single and on multiple objects. This support consists of updating replicated objects and of two kinds of synchronization: mutual exclusion and condition synchronization.
- Resolve the data dependencies between operations. This implies that the compiler has to pass data dependency information to the runtime system (see Section 3.6).
- Deal with dynamic object configuration with respect to the number, location, and replication of objects that are accessed within an operation or an atomic function. Generating optimized communication patterns to resolve data dependencies implies that processors must share knowledge of object locations to generate the same communication schedule.

- Finally, the generic runtime system also has to manage the cooperation between model-specific runtime systems, which ensures that additional model-specific runtime systems can easily be integrated into the system. For example, an operation on a nested object must be able to invoke an operation on a subobject that is a normal Orca object. Also, the atomic function module has to invoke both Orca objects and nested objects.

In Chapter 3, we showed that the collective computation model is a useful implementation technique to implement advanced parallel programming systems. In this chapter, we will present *weavers*, which implement the collective computation model. First, we will present the process and object abstractions, and then discuss how weavers can be used to resolve the requirements for the generic runtime system.

Light-weight processes are single threads running on a single processor, and are meant to do the main work of a parallel computation. Each process has its own stack, but shares the address space with the other processes running on the same processor. When an application is started, a single process (the main process) will be running on processor 0. A process can create another process (a child process) on any processor that participates in the execution of the application. The parent process can pass call-by-value arguments to a child process. After creating the child process, the parent process continues its execution and the child process starts executing a user-specified process function.

Objects encapsulate shared data. Objects are the only means by which running processes can interact. The state of a shared object can only be accessed and manipulated through operations. Mutual exclusion synchronization is guaranteed by allowing each processor to execute only one invocation on an object at a time. Condition synchronization is handled by allowing operations to block on guarded statements; as long as none of the guard expressions is true, the operation is blocked. Objects are stored either on a single processor, or can be replicated on a set of processors. The generic runtime system provides support for global object identifiers. On all processors, this object identifier can be used to uniquely identify the object and to find out which processors have a copy of the data. In addition, the generic runtime system provides functions to invoke operations on the local copy of an object.

The implementation of collective computation is provided in an abstraction called *weavers*. A weaver is an invocation of a single function on a set of processors in a total order (i.e., all processors execute the they are involved in in the same order; the weaver invocations on different processors operate in parallel). The total ordering property makes weavers a useful tool to implement various runtime support routines. A weaver is created by a single process; this process is blocked until the weaver is finished. Therefore, executing a weaver can be regarded as a process that gives up its single processor and instead temporarliy runs on a set of processors. In the implementation of the generic runtime system, weavers are used to create and destroy objects and to create and terminate processes (as described in Section 3.5, we assume that applications do not create excessive numbers of objects or processes). The total ordering guarantees all processors can keep track of all objects and processes that are in use in a consistent manner.

The most important use of weavers is that each runtime system creates a weaver to invoke operations on objects. The generic runtime system provides support to perform an operation on only a local instance of an object. By using weavers and the ability to perform operations on the local copy of an object, specific runtime systems can implement operations on replicated objects. Since weavers are invoked in a total order, sequential consistency is preserved. Also, since all runtime modules always use a weaver to invoke operations on objects, a runtime module can use the weaver even if it is created by a different runtime module. For example, the weaver created by the atomic function module to perform the computations of the atomic function can be used by the nested object module when a nested object is invoked within the atomic function. This separation between object management and invocation facilitates the integration of different runtime modules.

Operations on objects can block by using guards (i.e., condition synchronization). Since an operation invocation is always associated with a weaver, the generic runtime system has to provide support to block weavers. When another operation updates an object, the guard expression of a blocked invocation may become true. If so, the generic runtime system will wake up the blocked weavers. If a weaver wakes up other weavers, these weavers will be resumed in the order in which they were originally executed.

Operations on Orca objects can be easily expressed in terms of weavers. For example, an operation on a remote object is similar to running a weaver on the remote processor that invokes the operation locally. Performing a write operation on a replicated object is similar to running a weaver on all processors that have a copy of the object. Weavers are not restricted to invocations on a single object, however, but can be applied more generally. A single weaver function can perform operations on multiple objects, as is needed to implement atomic functions. As described in Chapter 3, this can lead to data dependencies, which result in communication between instances of weavers. This communication is implemented using the high-level communication primitives presented in Chapter 5.

Section 6.1 describes the creation and destruction of processes and objects. In Section 6.2, the interaction between the specific runtime systems and the generic runtime system is described. Section 6.3 describes how weavers are implemented and how synchronization between weavers takes place. Finally, Section 6.4 describes how data dependency information can be passed to the generic runtime system and Section 6.5 presents how these data dependencies are translated into communication schedules during the execution of a weaver.

6.1 Creating processes and objects

A process is a single thread of computation that runs on a single processor. Processes are used by the application to express parallelism. Each process can fork other processes by calling the process create function. Parameters passed to the new process are copied (i.e., only call-by-value parameters), so that the parent process and the child process can continue their execution independently. Processes are implemented on top of the Panda threads interface. A single processor can run multiple processes. The thread scheduler of the underlying Panda implementation decides how these processes will be scheduled (i.e., preemptively scheduled, prioritized threads on most systems).

To share an object between processes, the object's global identifier can be passed as parameter to the child process. A process can perform operations on this object by using the global object identifier. Therefore, it is important that object creation and process creation be ordered. Suppose that a process creates an object and passes the new object identifier as argument to a new process. The new process may not invoke an operation on the object until the information associated with the global object identifier is available. To prevent this situation, both processes and objects are created using weavers. The total ordering of the execution of the weavers guarantees that the local information that corresponds to a global object identifier is available before the new process is started.

The generic runtime system creates processes by creating a weaver on all processors. The weaver function gets as arguments the process function to execute, a copy of the argument vector, and the processor identifier on which the new process should be created. On this processor, a new process is created that calls the process function with the argument vector. On all processors, a process structure is created that contains information about each process, such as the processor on which a process runs. In addition, each processor maintains the number of running processes; if no more processes are running, the runtime system terminates. To detect this, a process that finishes its execution creates a weaver on all processors that clears the process state and decrements the number of running processes.

Creating a process on a single processor by running a weaver on all processors is expensive. We assume, however, that only a few processes will be created (typically only one process per processor). In the original Orca system, creating a process is also expensive, because it also uses a broadcast message [12]. Most (existing) Orca applications use one process per processor, and therefore comply with this assumption. We will make the same assumption for objects, i.e., we assume that applications do not create large numbers of shared objects during their execution.

Object creation is handled in a similar way as process creation. To create an object, the programmer has to specify the *object type* and the replication set (i.e., the set of processors that will maintain a copy of this object). The object type contains the information to create and destroy the object data and contains pointers to the operations that can be applied to an object of this type. Again, a weaver is created on all processors, but now with the object type identifier and the replication set as arguments. When the weaver function is called, a sequence number is incremented and used as the global object identifier. Since the weavers are executed in total order, the object identifiers will be the same on all processors. All processors create a runtime data structure for this new object, which contains the global object identifier and some information used by the specific runtime system that is responsible for this object. Finally, the processors that are in the replication set call the object initialization function that creates the local data part of the object (see Figure 6.2).

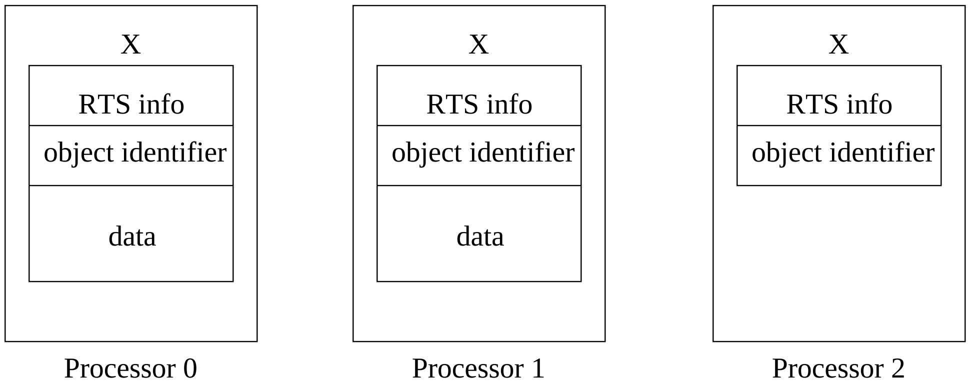


Figure 6.2: Creating object X on replication set {0, 1}. All processors maintain some state for the object, but only the processors in the replication set have a copy of the data.

A similar problem as the ordering problem between the creation of objects and processes occurs when an object has to be destroyed. If the process that created the object can destroy the object at any time, child processes (created by this process) can still have references to this object through the global object identifier. To solve this problem, the weaver that creates a new process increments a reference counter for each shared object that is passed as parameter. This reference counter is stored in the object information. Since this weaver is executed on all processors, the reference counters can be maintained locally for all objects. The reference counter is decremented whenever a process that shared this object terminates or when the process that created the object tries to destroy it (which is also implemented with a weaver). The actual destruction of an object takes place only when the reference counter reaches zero.

6.2 Model-specific runtime systems

The generic runtime system is intended to support (various) specific runtime systems, each implementing their own object model or extension. It is important to keep in mind that these specific runtime systems can only interact by performing operations on objects. For example, if an atomic function invokes an operation on a subobject that is a shared data object, the interaction between the two runtime systems is managed by the generic runtime system.

In Section 6.2.1 we will present the interface between a model-specific runtime system and the generic runtime system. Section 6.2.2 describes the object interface presented by the generic runtime system. Finally, Section 6.2.3 describes how operations on objects are handled.

6.2.1 Interface

The generic runtime system associates a specific runtime system with each object type. The functions of this runtime system are registered as the object type's *hook* functions. When a process or weaver accesses an object, the generic runtime system forwards these calls to the hook functions in the specific runtime system that is associated with the objects.

Table 6.1 presents the hook functions that provide the interface between the generic runtime system and the specific runtime systems. During initialization, each specific runtime system registers this information to the generic runtime system. Each object type contains a pointer to the associated specific runtime system. Since every object instance contains a pointer to its object type, the correct hook function can always be found for each object.

Meta_info_size, *meta_object_create*, and *meta_object_destroy* are used during the initialization and destruction of objects, as described in the next section. *Meta_object_operset* and *meta_object_oper* are used to perform operations on objects. This is described in Section 6.2.3.

6.2.2 Objects

An object is a reference to some user-defined data. All objects are managed by the generic runtime system. Figure 6.3 presents the object data structure that is used in the generic runtime system. The first part consists of runtime information

Function	**Description**
meta_info_size( )	Size of the specific runtime system information.
meta_object_create( )	Initializes the object's runtime-specific information.
meta_object_destroy( )	Destroys the object's runtime-specific information.
meta_object_operset( )	Returns the processor set on which the operation invocation weaver has to be executed.
meta_object_oper( )	Performs an operation on the local state of the object.

Table 6.1: Interface between the generic runtime system and the specific runtime systems.

used by the generic runtime system, such as the global object identifier (obtained during object creation), the reference counter, and the object type. The specific runtime system associated with the object can add some state information to each object, which will be pointed to by the `info` pointer. This data is reserved by the generic runtime system, based on the size returned by *meta_info_size*. Finally, on those processors that contain the object data, the `data` pointer points to this object state.

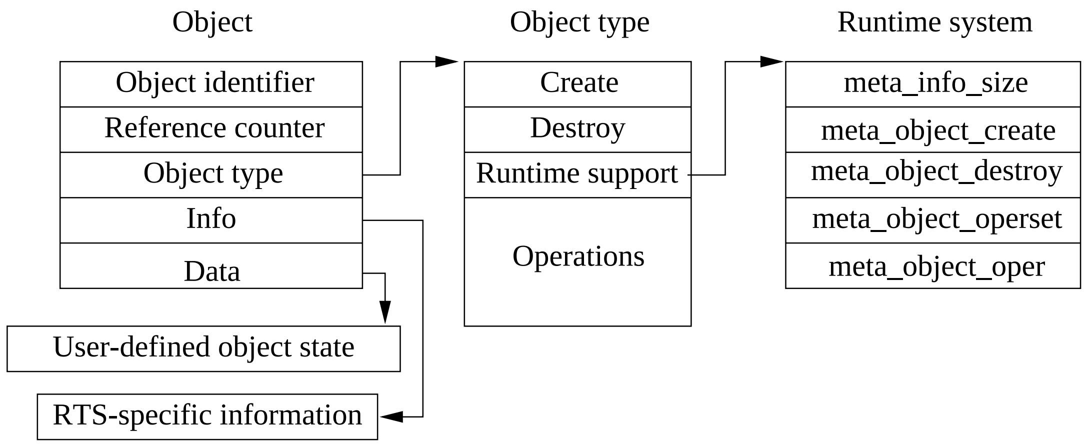


Figure 6.3: Runtime system object data structure.

The object type contains pointers to all operations that are allowed on the object. In addition, the object type information contains a function to create and destroy the user data of an object on a processor. Finally, each object type contains a pointer to the associated model-specific runtime system.

When an application creates a new object, it calls the *rts_object_create* function in the generic runtime system, passing the object type descriptor and the set of processors that have to receive a copy of the object as arguments. The generic runtime system creates a weaver that runs on all processors (see also Section 6.1). This weaver creates an object data structure on all processors. When the object data structure is created, a call is made to the *meta_object_create* function of the associated specific runtime module. This call initializes the runtime-specific information associated with the object. For example, the shared data object runtime system places the replication set of the object in this runtime system information.

After the runtime-specific information is initialized, the user-defined object data will be created on all processors in the processor set passed as argument to *rts_object_create*. These processors call the *create* function that is registered in the object type descriptor.

For the nested object runtime system, the *create* function also can create subobjects. Since a subobject may only be created on a processor that already contains a copy of the parent object, the existing weaver can be used to build the object, to fill in the runtime-specific information, and to call the *create* function for this object.

6.2.3 Operations on objects

To perform an operation on an object, the set of processors on which the weaver has to run must be known. For nonreplicated objects this is trivial, namely only the processor that has a copy of the data. For replicated objects, however, the set depends on the operation that has to be executed and the state of the object. For example, consider operation *min* in Figure 6.4. If the value of parameter v is less than the object value stored in x, the operation is a write operation; otherwise is is just a read operation. Read operations require fewer resources than write operations if the object is replicated, since only one processor that has a copy of the object needs to be accessed (preferably the local processor, if it has a copy of the object.) For a write operation, all processors that have a copy of the object must perform the operation.

To perform an operation, a processor set has to be computed. The object, however, may be managed by a different runtime system than the runtime system that invokes the operation. For example, the atomic function runtime system may invoke an operation on a shared data object. Each runtime system can also make additional distinctions apart from read and write operations. For example, the nested object runtime system distinguishes between operations that access at most one subobject and operations that can access multiple subobjects, because accessing multiple subobjects requires stronger synchronization.

To acquire a processor set, we use the concept of an *operation capability*. An operation capability is a bitmap that specifies at run-time what an operation invo-

```
object implementation IntObject;
    x: integer;

    operation min(v: integer);
    begin
        if v < x then
            x := v;
        fi;
    end;
end;
```

Figure 6.4: Minimum operation on an integer object in Orca.

cation is allowed to do to an object, based on the set of processors that participate in the operation. Each specific runtime system can provide its own set of capabilities. For example, the runtime system for shared data objects provides two capability values: `READ_CAP` and `WRITE_CAP`. If an operation only reads the object, it suffices to pass a capability with value `READ_CAP`. However, before an operation can update the state of an object, it first has to check whether the capability allows this. If the capability is insufficient, the required capability value is stored in the parameter that was used to pass the original capability, and the operation returns with an error code that indicates that the given capability was insufficient.

Figure 6.5 presents a simplified implementation of the *min* operation (as it would be generated by the compiler). The shared data object runtime system first tries to perform the operation with only a read capability. If the operation detects that it has to update the object, because the parameter has a lower value than the object state, a call to *CHECK_CAP* is inserted that checks whether the current execution of the operation has write permission. Since this is not the case, the operation sets the required capability to “read and write” and returns `FAIL_CAP`. The specific runtime system that handles this object now knows that the operation should be retried using the new capability.

Each combination of capability and object instance has its own processor set on which the operation should be executed. For a replicated integer object, an operation with a read capability would be executed locally (or on a single remote processor if no local copy is available), while an operation with a write capability would need to be executed on all processors that have a copy. Capabilities are strictly ordered by the processor set that they require: each capability requires a processor set that is a superset of the processor sets required by weaker capabili-

```
#define READ_CAP  0x01
#define WRITE_CAP 0x02

#define CHECK_CAP(cap, need) \
        if ((∗(cap) & (need)) ≠ (need)) {\
          ∗(cap) |= (need); \
          return FAIL_CAP; \
        }

int
oper_min(IntObject ∗obj, int ∗cap, int v)
{
    CHECK_CAP(cap, READ_CAP);
    if (v < obj→x) {
        CHECK_CAP(cap, READ_CAP | WRITE_CAP);
        obj→x = v;
    }
    return OK;
}
```

Figure 6.5: Implementation of the *min* operation in C.

ties. In the *min* example, the `FAIL_CAP` return value triggers the specific runtime ssystem to compute the processor set that belongs to the updated capability. This processor set is then used to re-invoke the operation.

To determine the initial processor set, each operation knows about its minimum capability. The minimum capability is the capability that needs the fewest processors to execute the operations. For the *min* operation, this is a read capability. The *assign* operation, which always updates the object, would require the write capability. Whenever an object type is registered, its operations and their minimum capabilities are also registered (either by the compiler or by the object writer).

To determine the initial processor operation set, the generic runtime system provides the *rts_object_operset* hook function. This function gets as parameters the object and the capability. This hook calls the associated *meta_object_operset* function in the specific runtime system, which returns the processor set needed to perform the operation. For example, if a process performs the *min* operation, it first determines the minimum capability needed (i.e., a read capability). The process then asks the generic runtime system for a processor set that is needed to execute an operation with a read capability on the integer object. This request is forwarded to the shared data object runtime system, which will return a processor set that contains only one processor, since that is sufficient for a read operation. Now a weaver is created on only this processor (which may be the local processor), and the operation is invoked. If the operation succeeds, the result is returned to the invoking process. If the operation fails, a new processor set is returned that is sufficient for the stronger capability.

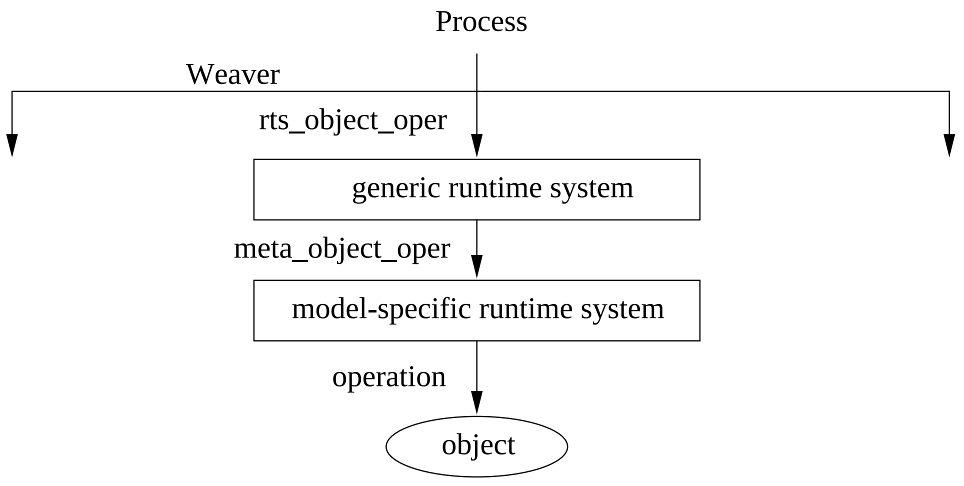


Figure 6.6: Invoking an operation.

Figure 6.6 presents an overview of the invocation of an operation on an object. First, the invoking process requests the initial processor set for the operation and starts an invocation weaver on each processor in this processor set. All weaver threads invoke *rts_object_oper* with the object, the operation identifier, and the operation arguments. The generic runtime system finds the model-specific runtime system associated with the object and looks up the operation descriptor that belongs to the operation identifier. The generic runtime system then calls *meta_object_oper*, passing the object, the operation descriptor, and the operation arguments. First, the model-specific runtime system computes the capability of the operation, based on the runtime specific object information and the processor set on which the invocation weaver is running. This capability may be stronger than the capability that the invoking process used to compute the initial processor set, because this processor set may also be sufficient for the stronger capability. For example, if a process invokes a read operation on a single-copy shared object, the processor set contains all processors that have a copy of the object. Therefore, the shared objects module will invoke the operation with a capability that allows both reads and writes on the object. After computing the capability, *meta_object_oper* calls the actual operation function.

When the operation function is finished, it returns with an operation status value to the model-specific runtime system. Four operation status values are used:

OK The operation succeeded. The model-specific runtime system returns this value to the generic runtime system, which in turn returns it to the invoking weaver threads.

BLOCKED None of the guard conditions evaluated to true. This value is returned to the generic runtime system, which blocks the weaver. Blocking weavers is described in Section 6.3.2.

FAIL_CAP The operation failed because the capability did not include the necessary rights. The model-specific runtime system computes a suitable processor set based on the runtime specific information associated with the object (e.g., the replication processor set) and the required capability that is returned by the operation. This processor set is returned to the generic runtime system with a *PROCSET* return status.

PROCSET The operation failed because the processor set was not sufficient to execute the operation, even though the capability included all required rights. This can occur, for example, in an operation on a nested object. Although the weaver processor set may be sufficient to perform an operation on the root object (because it is a read operation), it may not be sufficient for an operation on a subobject. The operation on the subobject returned a suitable processor set, which is forwarded to the generic runtime system. The generic runtime system forwards this processor set to the weaver, which will return it to the invoking process.

The invocation weaver can receive only three of these operation status return values, since the `FAIL_CAP` return status is always converted to `PROCSET` by the model-specific runtime system. If the call to *rts_object_oper* returns `BLOCKED`, the invocation weaver can block itself. This is explained in Section 6.3.

What is important is that an operation on an object can be performed by any weaver. The runtime system makes a clear distinction between the control of the operation on an object (which is always handled by the associated runtime system) and the invocation of an operation on an object (the weaver can be created by any runtime system). This distinction allows us to integrate different runtime systems easily. For example, a weaver that is created by the atomic functions module can perform operations on shared data objects.

6.3 Weavers

In this section, we present the *weaver* abstraction. A weaver is a *single* function (e.g., the code of an atomic function) that is executed on a *set* of processors. On each of these processors, an *instance* (i.e., a thread) is started that will execute this weaver function. Weavers serve three purposes. First, they support the collective computation execution model presented in Chapter 3. Second, weavers provide facilities to implement the synchronization requirements (condition synchronization and mutual exclusion) that are needed to perform operations on multiple objects. Finally, weaver instances can communicate with each other to interlace their computations.

6.3.1 Interface

A process creates a weaver by calling *weaver_create*, passing the destination processor set as parameter (see Table 6.2). During the execution of the weaver, the invoking process is blocked. Both input and output parameters can be passed to the weaver, and the results of the weaver invocation will be returned to the process after the weaver has finished. If one of the weaver instances runs on the same processor as the process that invoked the weaver, this thread will copy all output parameters of the weaver function to the argument variables that the process provided. Otherwise, the processors that execute the weaver select a processor that will marshal the output parameters and will send the result message to the invoking process. Figure 6.7 shows an example situation in which a single weaver function (f) is executed on three processors (0, 1, and 3).

Weaver functions can contain arbitrary code. In particular, weavers are not restricted to invocations on a single object, but can also be used to perform operations on multiple objects.

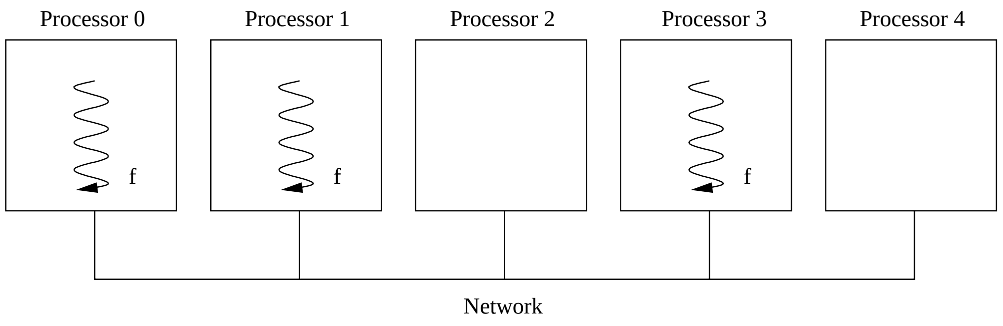


Figure 6.7: A weaver with instances running on three processors (0, 1, and 3) executing the same function (f).

Weavers are always executed in a total order. To guarantee this, all weavers that execute on multiple processors are created using a totally-ordered group message. These weavers receive a globally unique identifier, so that they can be identified by other processors. A weaver that has to run only on a remote processor (i.e., its processor set contains one processor) is created with a unicast message, since this does not violate the total ordering. If the weaver only has to run locally (i.e., its processor set contains only the local processor), no message is sent. The total ordering on the execution of weavers together with program order (since the process that creates a weaver is blocked until the weaver is finished) guarantees that weavers can be used to implement sequential consistency [48].

Since weavers are used throughout the runtime system, we tried to implement them as efficiently as possible. First, we avoid using a broadcast message if only a single processor is in the processor set of a weaver. Therefore, operations on a single-copy object on a remote processor use only a request and a reply message. Second, when a model-specific runtime system registers a weaver function, a flag can be set which tells the generic runtime system whether the weaver function needs a real thread for each instance. A weaver function needs a real thread if it can block while there is state on the stack of the weaver instance. If no such state is required, the weaver can specify a continuation function when it blocks [26,54]. This continuation function will be called to resume the weaver.

6.3.2 Synchronization

As with the shared data object model, atomically executing operations on multiple objects imposes two synchronization requirements. The first requirement is that

Function	**Description**
weaver_create( )	Creates a new weaver with instances on all processors in the processor set.
weaver_wait( )	Blocks the current weaver.
weaver_signal( )	Wakes up another weaver.
weaver_register( )	Registers which processors have a consistent value of a local variable.
weaver_depend( )	Registers which processors need a consistent value of a local variable.
weaver_synchronize( )	Synchronizes the instances of a weaver. After this call, all processors that have registered a dependency for one or more local variables have received a consistent copy.

Table 6.2: Weaver interface functions.

the execution is atomic: no other operations may be performed that access any of the objects that the atomic execution can access. The runtime support system must enforce this requirement.

The second synchronization requirement is condition synchronization. For this synchronization to be useful, the programmer must be able to specify synchronization conditions that depend on the results of operations on objects. Therefore, a blocked atomic execution needs to be continued whenever one or more of these objects is updated. For example, consider the atomic function *Terminate* in Figure 2.9 on page 31. *Terminate* must be re-evaluated after either the work queue is updated (since it may contain work) or the active worker counter is updated (to detect termination).

To provide mutual exclusion, all processors execute a single weaver function at a time. When a *weaver_create* message arrives, a new weaver thread is created and enqueued in the *weaver queue*. (For efficiency, weaver threads that finished the execution of a weaver are not destroyed, but cached in a pool to serve other weavers in the future.) All processors manage their own weaver queues, which contains entries for all weavers that a processor has to participate in (i.e., a instance of the weaver will be executed on the processor). Only the *first* runnable weaver in the weaver queue is executed. Other weaver invocations that arrive during the execution of a weaver are enqueued in the weaver queue. An enqueued weaver function will only be executed if all earlier weavers in the queue stopped running. Since the *weaver_create* messages arrive in a total order, all weaver queues will contain enqueued weavers in the same order. Since only a single weaver function at a time is allowed to be active on a processor, mutual exclusion is guaranteed.

To implement condition synchronization on weavers, weavers can block by calling *weaver_wait*. When a weaver blocks, it remains in the weaver queue, but its state is changed from *running* to *blocked*. Another weaver can execute while earlier weavers in the weaver queue are blocked. A weaver can wake up another blocked weaver by calling the *weaver_signal* function, which changes the state of the other weaver from *blocked* to *runnable*.

Figure 6.8 shows the state transition diagram for each weaver. When the weaver is created, its initial state is *start*, a special case of *runnable* used for initialization. When this weaver is the first runnable in the weaver queue, its state is changed to *running*. A running weaver that blocks itself changes state to *blocked*. This state can only be changed by another weaver to *runnable* again. Finally, a weaver that is finished calls *weaver_exit*, which changes its state to *exit*. These weavers are removed from the queue by the *weaver queue scheduler*, which is invoked by a weaver when the weaver blocks or exits.

All instances of a weaver have to make the same calls to *weaver_wait* and *weaver_signal* to ensure the consistency of the weaver queues. Recall that all instances of a weaver execute the same function. However, this is not sufficient to ensure consistency of the queues. To handle condition synchronization on objects, all weaver instances must have the values of all variables used to compute the synchronization condition. If the value of such a variable is the result of an operation on an object, it might be necessary to send this value to the processors that have no local copy of this object. The communication interface between weaver instances will be presented in Section 6.4.

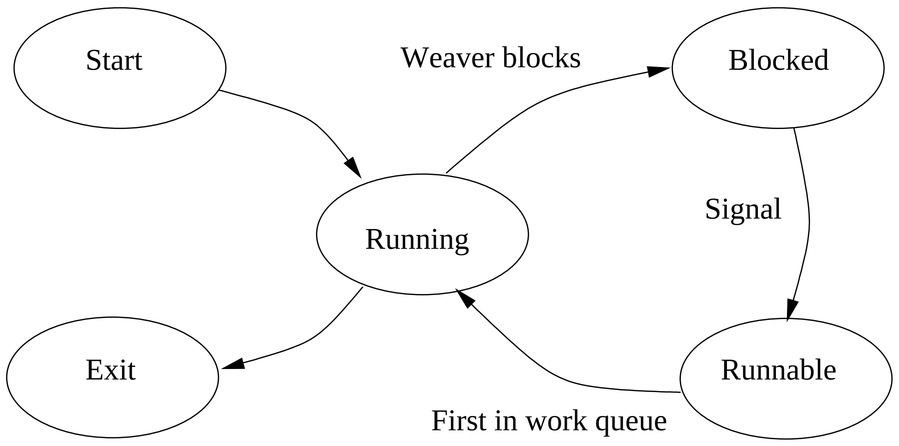


Figure 6.8: State transition diagram for the lifetime of a weaver. All instances of a weaver (i.e., on different processors) must perform the same state changes.

A major problem with the integration of mutual exclusion and condition synchronization is that copies of an object may only be available on a subset of the processors. Mutual exclusion requires that all processors that have a copy of any of the objects accessed in the atomic execution synchronize. Condition synchronization, on the other hand, uses object state changes to synchronize weavers. The state change of an object, however, is only known on those processors that have a copy of the object, since only those processors perform the write operation.

We consider the two cases that can occur when a weaver is blocked on an object: either all processors that execute a weaver instance have a copy of the object, or some processors execute a weaver instance, but do not have a copy of the object. To illustrate these two cases, we use the termination detection algorithm presented in Section 2.4, and assume that the `workers` object is replicated, while the work queue object is stored only on processor 0 (see Figure 6.9).

If the workers counter is updated, all processors are aware of it, since they all have to update their local copy. After the update, all processors know that the atomic function *Terminate* has to be re-evaluated. Therefore, all processors locally change the state of the weaver from blocked to runnable.

After a processor has changed the state of some of the weavers to runnable, it has to determine which weaver to execute next. Since the signalling weaver function is executed in a total order, the signals are also handled in total order. When the signalling weaver exits, the weaver scheduler executes the first runnable weaver in the weaver queue. This guarantees that the total order of the execution of the weavers is preserved. Whenever the weaver that executes *Terminate* becomes the first runnable weaver in the weaver queue, it will continue its execution and the termination condition will be re-evaluated.

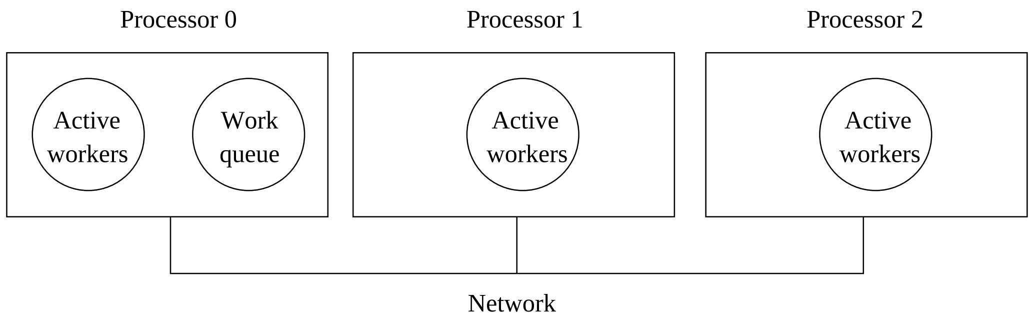


Figure 6.9: Object distribution of the `workers` object and work queue object on three processors.

To illustrate the second case (not all processors have a copy of the object), we consider the situation in which the atomic function *Terminate* is blocked and the work queue object is updated. In that case, only processor 0 performs the update, since it is the only processor with a local copy of the object state. Since only processor 0 knows about this state change, a *signal message* is required to inform the other processors that the atomic function *Terminate* has to be re-evaluated.

In order to preserve the total order of the execution of the weavers, all updates to the state of the weavers in the weaver queue must be kept consistent. There are two ways to handle signal messages so that this property is maintained. The first solution is to block all weaver instances until all signal messages generated by this weaver have arrived. This is inefficient, because it requires all weaver instances to communicate with each other before each processor can continue with the other weavers.

The second solution is to select one of the processors that have a local copy of the object (the *master* processor) to send a *totally-ordered* signal message. Since different objects can have different master processors, multiple signal messages can arrive. In addition, signal messages can arrive before or after a processor started the execution of the signaling weaver, because weaver instances do not necessarily synchronize their execution. To preserve the order of the weavers, signal messages are not executed directly when they are received, but instead are also enqueued in the weaver queue. When no runnable weavers precede the signal message in the weaver queue, all weavers in the signal message that still exist are signaled locally (see Figure 6.10). The weaver scheduler then selects the first runnable weaver to be executed. Since the signal broadcast uses the same total ordering as the broadcast that is used to create weavers, the weavers are still executed in order.

To reduce the number of messages a master processor has to send, a signal message is not sent immediately, but instead a single message is sent when the weaver queue does not contain any runnable weavers. The master processor then collects the weaver identifiers of the weavers that it has to signal, and broadcasts these identifiers in a single signal message.

In the current implementation, the processor with the lowest processor identifier is selected as master when an object is created. Alternatives, such as selection the master processor randomly, are also possible, as long as all involved processors come up with the same master. The overhead on the master processor is very low, though, so our simple scheme suffices.

Figure 6.11a presents a situation in which weaver A is blocked while performing operations on objects X and Y. Object X and object Y are single-copy objects on processor 0 and processor 1, respectively. Since weaver A was blocked, its weaver identifier is stored in the block queue of both objects. Weaver B now performs an update operation on objects X and Y. Since both objects are on different processors, and weaver A runs on processor 0 and 1, both processors mark weaver A. Note that both processors are not aware of the weavers that the other processor needs to signal. After the execution of weaver B is finished, both processor 0 and 1 broadcast a signal message, which results in the situation in Fig-

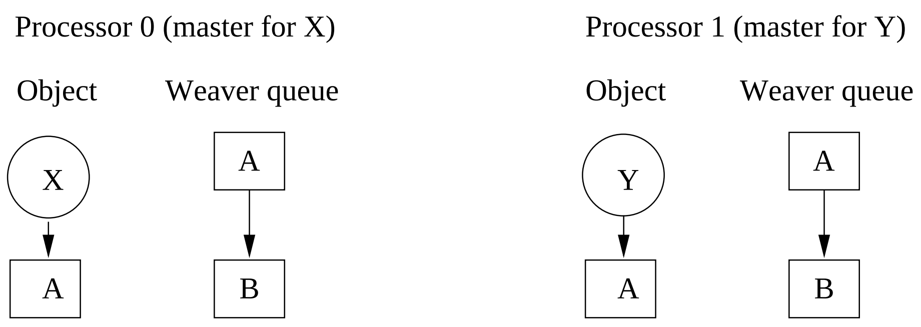
```
proc weaver_queue_scheduler( ) ≡
  while runnable weavers do
        entry := first runnable entry
        if entry = signal message then
          perform signals locally
        else
            execute weaver in entry
        fi
  od
  if marked weavers then
    send signal message
  fi
end
```

Figure 6.10: Weaver queue scheduler algorithm in pseudo code.

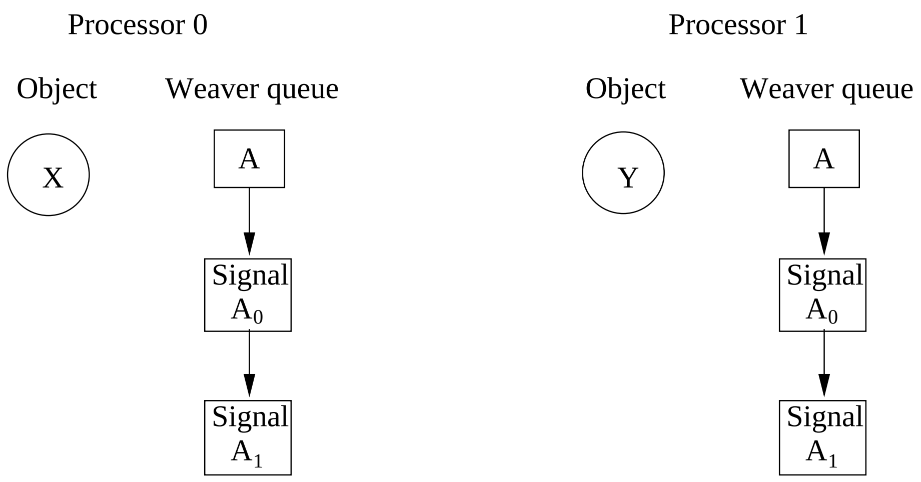
ure 6.11b. After the first signal message, weaver A will be re-executed.

Spurious signal messages and weaver wakeups can occur if different objects have different master processors. If weaver A in Figure 6.11b completes its execution after the first signal message, the second signal message will find that weaver A no longer exists, and therefore does not do anything. If weaver A blocks again after the first signal message, the second signal message will again signal weaver A, and weaver A will be re-executed. Since neither object X nor object Y has changed in the meantime (since no other weavers have been executed), the weaver will block again.

To reduce the number of spurious signal messages and local wake-ups, we use the following optimization. When a master processor determines that it needs to signal a weaver using a signal message, it marks this weaver with a *wakeup marker*. When all weavers are blocked, the identifiers of all marked weavers are collected in one signal message, and this signal message is broadcast. When the scheduler handles a signal message and wakes up a weaver, the mark is removed from this weaver. Therefore, a master processor only sends a signal message if it has a marked weaver that has not been signaled by another master processor yet (with respect to the logical time as represented by the weaver queue). In the example, if the signal message from processor 0 (A_0) is received by processor 1 before the local instance of weaver A finished its execution, the signal message will be enqueued and handled before processor 1 sends its signal message. Since the signal message removes the wakeup marker from weaver A, the signal message from processor 1 (A_1) will not be sent.



a) Weaver A is blocked on objects X and Y, while weaver B is running.



b) Weaver B finished its execution. Two signal messages have arrived and are placed in the work queue.

Figure 6.11: Waking up weaver A after weaver B performed update operations on objects X and Y.

6.3.3 Blocking operations

The weaver signal and wakeup functions provide a generic mechanism for one weaver to wake up another. For the extended shared objects system, however, weavers only wake up each other by performing operations on objects. Therefore, support has been added to the generic runtime system to facilitate this kind of synchronization between weavers.

Condition synchronization in operations is specified by guard conditions. With normal Orca objects, the guard condition has to be re-evaluated whenever the object is changed. With nested objects and atomic functions, however, the guard condition must also be re-evaluated if any of the objects that were accessed by the operation before it blocked is changed. For example, the guards in atomic function *Terminate* (see Figure 2.9 on page 31) need to be re-evaluated when either the queue object or the workers object are updated. To manage this information, the generic runtime system maintains for each weaver an *accessed objects set* that contains the identifiers of accessed objects. Every time an object is accessed for the first time by a weaver, its object identifier is added to this set.

Whenever a weaver blocks, the generic runtime system traverses the accessed objects set and adds the weaver identifier to a *pending weavers set* that is locally maintained per object. After that, the accessed objects set is cleared and the state of the entry of this weaver in the weaver queue is set to blocked. When another weaver updates an object, the generic runtime system signals all weavers in the pending weavers set of this object and the pending weavers set is cleared.

After a weaver is signaled, it starts with a re-evaluation of the guard conditions. If the guards fail again, the same process is repeated. If the guard succeeds, however, the weaver will finish its execution and terminate. Since only one object needs to be updated to wake up a weaver, the weaver identifier of a terminated weaver may still be in the pending weavers sets of objects that are not updated yet. Therefore, the *weaver_signal* function first checks whether the weaver is still blocked before it performs the actual state change. To prevent pending weavers sets from growing without bounds, a garbage collection phase is started whenever the length of the set exceeds a certain threshold. This garbage collection phase removes all weavers that do not exist anymore (this can be checked on the local processor) from the set.

Re-evaluating the guard conditions guarantees that the operation will succeed if one of the guard becomes true. If the guard conditions depend on the results of operations on objects, all those operations will be re-executed. This is not necessary, since only the operations on objects that have changed need to be executed again. As described earlier, we do not allow operations used for the guard condition to change the state of the invoked object.

To solve this problem, we extended the wakeup message mechanism and added a per-weaver object set (the *recheck* set) of objects on which the operation has to be re-evaluated when the weaver is woken up. The signal message now not only contains the weaver identifier, but also the set of objects that have been changed. For nested objects, the object identifier of the root object is used. If an object has a copy on all processors on which the weaver is executed (i.e., the update does not require a signal message), the recheck set can be updated locally.

When the weaver performs an operation that can block on a guard condition, it maintains on the stack of the weaver instances the results of the operation. (We can only use this optimization for weaver functions that associate a stack with the execution.) Instead of returning `BLOCKED` to the generic runtime system, the weaver function calls *weaver_wait* directly, so the results are maintained on the stack until a signal arrives. Now, the weaver function re-executes only operations on objects that are in the recheck queue of the weaver. This information can be retrieved using *weaver_cached*, which returns true if the weaver is re-executed and the object is not in the recheck set of the weaver. For the cached objects, the results of the operation are still available on the stack.

By handling all weaver synchronization on objects in the generic layer, we ensure that the specific runtime modules are orthogonal. For example, suppose that the work queue that atomic function *Terminate* accesses is a nested object with normal Orca queue objects as subobjects. All three runtime modules are involved during the execution of this atomic function. The *Empty* operation on this nested object will check all Orca queue objects before it returns true, so the accessed objects set will contain the object identifiers of all these Orca queue objects. When the atomic function blocks, its weaver identifier is added to the pending weavers sets of all those objects. When another weaver updates any of these subobjects, the atomic function will be signaled and eventually be re-evaluated.

6.4 Resolving data dependencies

During the execution of a weaver, communication may be required between the instances in order to propagate results of operations on objects to those processors that do not have a local copy. At compile time, however, it is not known on which processors these objects will be located. Therefore, the weaver abstraction provides a communication interface that allows weaver instances to specify data movement by associating consistent variables (i.e., variables that contain the result of the operation on a local copy of an object) with processors. This communication interface provides three functions, one to express which processors contain consistent (i.e., up-to-date) values of the data, a function to express which instances need a consistent value, and a function to perform the actual data transfer.

The first communication function, *weaver_register*, is used to register which weaver instances have a valid copy of a local variable. For example, the result of a read operation on an integer object is only available on weaver instances that run on processors that have a local copy of this object, since only those processors performed the operation. After the statement that invoked the operation, a call to *weaver_register* can be added by the programmer (or the compiler) that registers that the local variable containing the result is only consistent on those processors that actually performed the operation. A pointer to the local variable, its type descriptor, and the processor set are passed as parameter to *weaver_register*. Each call to *weaver_register* returns a *variable identifier*, which can be used to address this local variable. The function *weaver_object_register* invokes *weaver_register* with the processor set of the object as argument.

The second communication function, *weaver_depend* is used to specify which processors need a consistent value of a local variable before they may continue their execution. This function gets as arguments the variable identifier returned by *weaver_register* and a processor set.

To be able to aggregate data messages, no communication takes place during the execution of *weaver_depend*. Instead, a separate function *weaver_synchronize* has to be called. This function guarantees that all local variables that have to be consistent on the local processor will be made consistent. Since all processors that execute the weaver make the same calls to *weaver_register* and *weaver_depend*, they all can determine locally what the communication pattern is [110]. Section 6.5 describes the generation of a communication schedule for resolving data dependencies.

For operations on multiple objects, we distinguish three types of data dependencies. The first type deals with variables that are used to synchronize the execution of normal statements, such as flow control conditions and guard expressions. All instances have to execute this code, and therefore the destination processor set contains all processors that execute the weaver.

Figure 6.12 gives an example of a data dependency for all weaver instances. On the left side of the figure is part of an atomic function and on the right side the extended version with the calls to the generic runtime system. First, all weaver instances invoke the *value* operation on object X. The runtime system enforces that only invocations on local objects are actually executed. After the operation on object X, all weaver instances register the local variable `r`, which contains (if the processor has a copy of object X) or should contain the result of the *value* operation. Because the control flow depends on the value of `r`, all weaver instances should get this result. Therefore, before the expression `r < 5` is computed, all weaver instances call *weaver_data_depend*, with `r_id` as parameter. Function *weaver_data_depend* calls *weaver_depend*, passing the variable identifier and the processor set containing all processors that are involved in the weaver as argu-

```
r := X$value();
....
if r < 5 then
    ....
```

a) Atomic function code.

```
r = X$value( );
r_id = weaver_object_register(&r, X);
....
weaver_data_depend(r_id);
weaver_synchronize( );
if (r < 5) {
   ....
}
```

b) Calls to the runtime system.

Figure 6.12: Data dependency.

ments.

After all data dependencies have been specified (one in this example), a call to *weaver_synchronize* is made. This call guarantees that after it returns, all specified data dependencies have been resolved, i.e., all weaver instances that need a certain result have it in their local variables. In this example, the *weaver_synchronize* call guarantees that in all weaver instances the value of `r` is the result of the *value* operation on object X.

The second data dependency type deals with local variables that are used as argument to another object operation, or are used in an expression that is only used to compute an argument for an object operation. Only the processors that have a local copy of this object need this result. Therefore, the processor set passed to *weaver_depend* contains those processors that have a local copy of this object.

```
atomic function copy(from, to:
        shared IntObject);
    r: integer;
begin
    r := from$value();
    to$assign(r);
end;
```

a) Atomic function code.

```
r = from$value( );
r_id = weaver_object_register(&r, from);

weaver_object_depend(to, r_id);
weaver_synchronize( );
to$assign(r);
```

b) Calls to the runtime system.

Figure 6.13: Object dependency.

Figure 6.13 gives an example of an object dependency. Before the operation on object `to` can be called, its input arguments must be available. In this example, this implies that `r` is consistent. Therefore, the compiler inserts a call to *weaver_object_depend* with parameters `to` and `r_id`. Function *weaver_object_depend* calls *weaver_depend* with a processor set that contains all processors that have a copy of object `to`. After the *weaver_synchronize* call returns, each processor that has a copy of object `to` has a consistent value of `r`. The result is only accessed if a local copy of the object is available.

The last type considers local variables that are used to compute the output parameters of the weaver. Only the processors that are designated to compute this result need consistent copies of those variables. The processor set passed to *weaver_depend* contains those processors.

```
atomic function sum(X, Y:
        shared IntObject): integer;
        r1, r2: integer;
begin
        r1 := X$value();
        r2 := Y$value();
        return r1 + r2;
end;
```

a) Atomic function code.

```
r1 = X$value( );
r1_id = weaver_object_register(&r1, X);

r2 = Y$value( );
r2_id = weaver_object_register(&r2, Y);

weaver_return_depend(r1_id);
weaver_return_depend(r2_id);
weaver_synchronize( );
if (weaver_invoker( )) {
   res = r1 + r2;
}
```

b) Calls to the runtime system.

Figure 6.14: Return dependency.

Figure 6.14 shows an atomic function that computes the sum of two integer objects, and the associated code. Only the processor that computes the final result of the atomic function needs the results of the operations on objects X and Y. Therefore, calls to *weaver_return_depend* are inserted, with the variable identifiers of `r1` and `r2` as argument. The function *weaver_return_depend* calls *weaver_depend*, with a processor set that contains the processor that will return the result to the invoking process. The *weaver_synchronize* call guarantees that the invoker gets the actual results in the variables `r1` and `r2`. On the other processors, no guarantees are given on the contents of these variables, so they should not be referenced. Finally, the invoker computes the sum of the two results.

Figure 6.15 illustrates the use of caching object invocation results for guard synchronization. The weaver function performs the read operations on objects X and Y, and checks whether the guard condition (i.e., `X$value() = Y$value()`) is true. If so, the code of the guard block is executed. Otherwise, *weaver_wait* is called and the weaver is blocked until one of the objects

```
atomic function compare(X, Y:
        shared IntObject);
begin
        guard X$value() = Y$value()
        do
                .....
        od;
end;
```

a) Atomic function code.

```
while(true) {
  if (!weaver_cached(X)) {
    r1 = X$value( );
    r1_id = weaver_object_register(&r1, X);
  }

  if (!weaver_cached(Y)) {
    r2 = Y$value( );
    r2_id = weaver_object_register(&r2, Y);
  }

  weaver_data_depend(r1_id);
  weaver_data_depend(r2_id);
  weaver_synchronize( );
  if (r1 == r2) break;
  weaver_wait( );
}
....
```

b) Calls to the runtime system.

Figure 6.15: Caching the results of earlier object invocations.

is updated by another weaver. For example, if object X is updated, the weaver function will re-execute the *value* operation and register the new result. The *value* operation on object Y, though, is not invoked again. Since the variable identifier is also not changed, all data dependencies on variable `r2` are still resolved, so communication (if required) only has to take place for `r1`.

6.5 Generating communication schedules

In the previous section, we have illustrated how the compiler can pass data dependency information to the runtime system in a way that is independent of the location and replication strategy of the objects involved. In this section, we will present mechanisms to translate such data dependencies into communication schedules. Since the compiler also inserts synchronization points, all communication can be postponed until synchronization.

When a new local variable is registered, an entry for this variable is added to the weaver data structure. This entry consists of a pointer to the real variable, the type of the variable, and two sets. The first set, the *has* set, contains all processor identifiers of processors that have a valid copy of the variable. The second set, the *need* set, describes the processors that need a copy of the variable.

Initially, the *has* set contains all processors that have the object on which the variable depends and the *need* set is empty. When *weaver_depend* is called, the set of processors that is passed as argument is added to the *need* set. After synchronization, the *need* set is added to the *has* set, and the *need* set is emptied.

During synchronization, the information in the *has* and *need* sets is used to generate a communication schedule. A basic algorithm would simply consider every variable separately. If the *need* set is a subset of the *has* set, the variable is already available on all processors that need it, so it can be skipped. Otherwise, the number of processors that need the variable and do not have it (i.e., $\mid need \backslash has \mid$) determines whether it is better to broadcast the variable or to send a point-to-point message. Based on this information, the appropriate actions are registered in the communication schedule. The decision which processor is going to send the variable is taken deterministically by all involved processors, so the resulting communication schedule is consistent.

Instead of treating each variable separately, it is better to take a decision based on the *has* and *need* sets of all variables. Our first extension exploits this information. First, it determines all variables that need to be transmitted. Next, the processor that has the largest number of variables available is selected as the source, using the processor identifier as tie breaker. The total number of destination processors (i.e., $\bigcup_{vars}(need \backslash has)$) again determines whether a unicast or a broadcast message is used.

This algorithm guarantees that all dependencies will be resolved, but for special cases it might not be the most optimal communication pattern. An example is when a global minimum has to be computed. Instead of performing the minimum computation on all weaver instances, which requires that all weaver instances have all the data, it is better to use an optimized reduction operation, as described in Chapter 5. Therefore, a set of specialized communication routines are provided which use optimized communication patterns for such special data dependency cases. The generic runtime system provides a function to annotate a variable identifier with a tag. This tag implies that the variable need not be resolved with the default mechanism, but instead will be resolved by the optimized communication function that corresponds to this tag. When *weaver_synchronize* is called, this communication function fills the communication schedule with an optimized communication pattern for those variables.

Summary

In this chapter, we described the generic runtime system, which provides a layer on which specific runtime systems can be implemented. The generic runtime system provides three major abstractions: processes, objects, and weavers. A process is a single thread running on one processor. Processes are meant to do the main work of a parallel application.

Objects are used for the interaction between processes, even if the processes are running on different processors. The actual implementation and semantics of an object are not known to the generic runtime system, but instead depend on the specific object implementation and on the associated runtime module. To allow different model-specific runtime systems to interact, the generic runtime system provides support for naming and accessing objects. Model-specific runtime systems are invoked by the generic runtime system through hook functions, which are registered when a new object type is created.

Weavers provide an implementation of the collective computation execution model. Weavers are used to invoke operations on objects. In addition, weavers are used within the generic runtime system for process and object creation. The separation of object management (by the associated runtime module) and object invocation (by weavers) allows us to integrate different runtime modules.

Performing operations on multiple objects imposes two synchronization requirements: mutual exclusion and condition synchronization. These synchronization requirements can be mapped to the signal and wakeup functions provided by the weaver abstraction. In addition, the generic runtime system provides support for weavers that perform blocking operations on objects.

Finally, we discussed the communication primitives that are provided by the weaver abstraction. These communication primitives allow the weaver instances to interlace their computations. Communication is not expressed in terms of message passing operations, but instead is described in data dependencies that need to be resolved. Based on these data dependencies an optimized communication schedule is generated which performs the actual communication.

Chapter 7

Shared Data Objects

This chapter describes the shared data object runtime module. As discussed in Section 3.6, each model-specific runtime system is implemented on top of the generic runtime system. The specific runtime modules are independent of each other, since they only interact through operations of the generic runtime system. Therefore, the shared data object module does not have to be aware of the other specific runtime modules, but only has to deal with performing operations on shared data objects. (We will call these objects Orca objects, to avoid confusion with nested objects that are shared between processes.) The application, on the other hand, has to be aware of the runtime modules that it is using (see Figure 7.1).

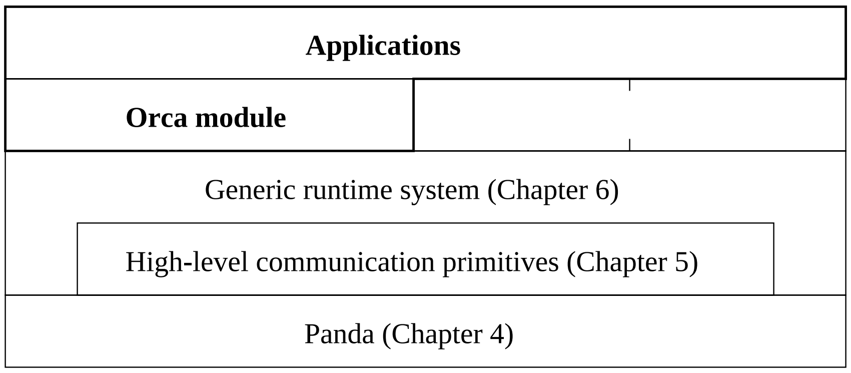

Applications
Orca module
Generic runtime system (Chapter 6)
High-level communication primitives (Chapter 5)
Panda (Chapter 4)

Figure 7.1: The Orca module in the architecture overview.

The Orca objects module has two interfaces. The first one is the interface to the application. This interface allows processes to perform operations on Orca objects. The second interface is for the interaction between the Orca object module and the generic runtime system. This interface is registered at the generic runtime system during initialization (see also Section 6.2).

Each object type is supported by a specific runtime module. When a new object type is registered, its object type descriptor contains a pointer to the specific runtime module that is associated with it (see Figure 6.3). For Orca objects, the object type points to a structure that contains the hook functions for the Orca objects runtime module.

Section 7.1 describes the implementation of the runtime support for Orca objects. In Section 7.2, we present performance results for this implementation of Orca objects, and compare it with the normal Orca runtime system.

7.1 Implementation

The implementation of runtime support for operations on Orca objects can directly be mapped on the weaver abstraction. To perform a write operation, a weaver, called the *invocation weaver*, has to run on all processors that have a copy of the object. For a read operation the invocation weaver can run on any (one or more) of the processors that have a copy.

If a process performs an operation on an Orca object, it has to create an invocation weaver. This invocation weaver is created by the part of the Orca objects runtime module that interfaces to the application. It is important to note, however, that it is not required that the invocation weaver that performs an operation on an Orca object is created by the Orca objects runtime module. Weavers created by other runtime modules can perform operations on Orca objects as well, as long as they run on the correct processor set. This property is used to integrate the different runtime modules.

In this section, we will describe the implementation of the two interfaces of the runtime module for Orca objects. We first describe the interface to the application, and then discuss the interface to the generic runtime system.

7.1.1 Interface to the application

The Orca objects runtime module exports one function to the application, *orca_do_oper*, which allows the application to perform operations on Orca objects. This function first marshals the object identifier, the operation identifier, and the operation arguments. Second, it determines the initial operation processor set on which the operation execution weaver must be executed.

To optimize read and write operations, we use the operation capability mechanism described in Section 6.2.3. Each operation descriptor for an Orca object type contains the minimum capability that is required to perform the operation. The initial operation processor set is determined by calling *rts_object_operset* with the minimum capability that is required by the operation as argument.

After the operation processor set is determined, an invocation weaver is created that runs on this processor set (see Figure 7.2). All instances of this invocation weaver unmarshal the object identifier and the operation arguments, and call *rts_object_oper*. The result parameters of this call (the operation return status and possibly the required processor set) are then returned to *orca_do_oper*.

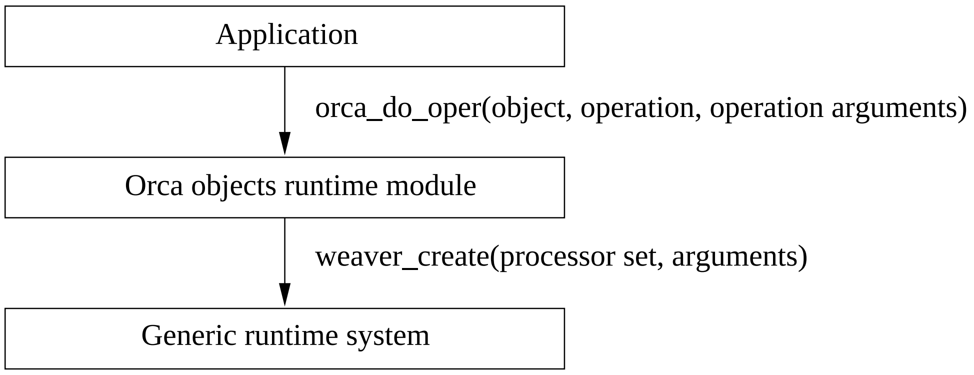

Application
orca_do_oper(object, operation, operation arguments)
Orca objects runtime module
weaver_create(processor set, arguments)
Generic runtime system

Figure 7.2: Invoking an operation on an Orca object.

The Orca objects module knows that the invocation weaver will only perform operations on Orca objects. Since such operations do not block half-way, the weaver instances do not need a separate stack (see Section 6.3.1). Instead, the weaver function can be called from any other thread, in particular the upcall thread that delivers the weaver create message. If the invocation weaver that is created is the first runnable weaver in the weaver queue, it will be executed immediately by the upcall thread. This saves an expensive context switch for most Orca operations. A similar optimization is applied in the normal Orca runtime system [26].

After the invocation weaver is finished, *orca_do_oper* checks the operation return status. If the return status is *OK*, the operation has successfully been executed, and the invoking process can continue. If the operation returned *PROCSET*, a new weaver will be created that runs on the new processor set (see Figure 7.3).

Even if the processor set is equal to the replication set of an object, it is not guaranteed that the operation can succeed. Between the computation of the processor set and the moment that the weaver starts running, other weavers may have changed the replication of an object. Therefore, the invoking process will create a new weaver every time the operation failed until the operation succeeds.

As Figure 7.3 shows, *orca_do_oper* does not make direct calls to the hook functions of the Orca objects module. Therefore, the same function can be used to perform operations on an object that is associated with another runtime module. The only assumption that *orca_do_oper* makes is that the weaver that is created to perform the operation invocation does not need a stack for its instances.

7.1.2 Interface to the generic runtime system

In Section 6.2, we described the interface between the generic runtime system and a model-specific runtime system. For the Orca objects module, these functions are

```
proc orca_do_oper( ) ≡
  marshal operation and arguments
  procset := rts_oper_procset(object, minimum operation capability)
  repeat
    ret := weaver_create(procset, invocation_weaver)
  until ret = OK
end
proc invocation_weaver( ) ≡
  unmarshal operation and arguments
  repeat
    ret := rts_object_oper(object, operation, arguments, procset)
    if ret = BLOCKED then
      weaver_wait( )
    else
        return ret, procset
    fi
  endrep
end
```

Figure 7.3: Implementation of *orca_do_oper* in pseudo code.

implemented as follows. All functions correspond to the hook functions, except that the prefix *meta* is replaced by *orca*.

Orca_info_size returns the size of the runtime specific data that is associated with each object. For Orca objects, this data contains the replication processor set (i.e., the processor set that was passed as parameter to *rts_object_create*). Therefore, *orca_info_size* returns the size of a data structure that contains one processor set.

Orca_object_create initializes the replication processor set that is associated with the object (see Figure 7.4). Space for this data has already been reserved by the generic runtime system, based on the return value of *orca_info_size*. The processor set that is passed as parameter to *rts_object_create* is passed as one of the arguments, so *orca_object_create* only has to make a copy of this processor set. *Orca_object_destroy* destroys the replication processor set.

Orca_object_operset determines the minimum set of processors on which the weaver must be executed to perform the operation. This operation processor set is determined by the replication processor set and the capability of the operation that has to be performed. For read operations, one of the processors from the replication processor set is selected and inserted in the processor set that is returned. If the local processor is in the replication processor set, it is selected; otherwise, an

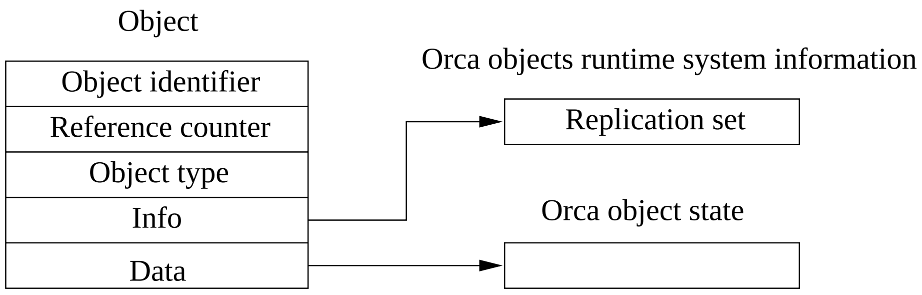


Figure 7.4: Orca objects runtime module information per object.

arbitrary processor is selected. For write operations, the replication processor set is returned.

Orca_object_oper is used to perform the actual operation. All instances of the invocation weaver make a call to the generic runtime system function *rts_object_oper*. The generic runtime system forwards this invocation to *orca_object_oper*, passing the operation descriptor as argument. *Orca_object_oper* performs three tasks: it determines the capability of the invocation weaver, it invokes the operation on the local copy of the object, and it determines the new processor set if the operation failed due to an insufficient capability.

The capability of the invocation weaver is determined by the replication processor set R of the object and the processor set W of the weaver (i.e., on which processors a weaver is running). If $W = R$, the weaver has "read and write" capability. If $W \subsetneq R$, the weaver has only read capability. Finally, if $W \supsetneq R$, some of the weaver instances cannot perform the operation, since there is no local copy of the object. In this case, a processor set is computed that contains only processors that have a copy of the object, and this processor set is returned to the invoking process. The invoking process has to create a new invocation weaver on this processor set.

There are two situations in which the weaver processor set can be a superset of the object processor set. The first case occurs when some of the replicas of an object are removed. The replication set of each object is maintained on all processors, but the system does not guarantee that an object does not change its replication between the creation of a weaver and the moment the weaver starts running. Therefore, when the weaver starts its execution, the object may have lost some of its replicas. The second case occurs when an Orca object is a subobject of a nested object and the operation on the nested object invokes an operation on this subobject. This case is discussed in Chapter 9.

After *orca_object_oper* has determined the capability of the invocation weaver, it invokes the actual operation function on the local copy of the object. This function returns an operation result status. As described in Section 6.2.3, the model-specific runtime system only has to deal with operations that fail due to insufficient capability. For Orca objects, this implies that the operation tried to perform a read operation but detected that it has to update the object. In this case, the replication processor set is returned to the generic runtime system, together with the operation return status *PROCSET*. All other return values are forwarded directly to the generic runtime system. Note that the function that performs an operation on an Orca object will never return *PROCSET*, since the processor set that is based on the capability is always sufficient to perform the operation.

7.1.3 Synchronization

Orca operations that contain guarded statements can block when all guard conditions fail. Since no stack is associated with the invocation weaver instances, the operation implementation will only return the status *BLOCKED* to the generic runtime system. The generic runtime system then calls *weaver_wait*, which changes the state of the weaver from running to blocked. In addition, the accessed objects set of the invocation weaver contains the object on which the operation was executed, so therefore the weaver is added to the pending weavers set of this object.

When another operation updates the object, the invocation weaver is signaled and the weaver's state is changed to runnable. The scheduler will at some point in time determine that this is the first runnable weaver in its weaver queue, and change its state to running. Since no thread (i.e., no stack) is associated with this weaver, the thread that executes the scheduler code immediately invokes the operation. This way, we never have to create a thread to invoke an Orca operation.

7.2 Performance

In this section, we will present the performance of the Orca objects runtime module and compare it to the performance of the normal Orca runtime system. The goal of this section is to show that the performance of our prototype runtime system is close enough to the performance of the original Orca system. This allows us to use the prototype runtime system for making performance comparisons between applications that use only Orca objects and applications that also use the extensions. These comparisons will be presented in the following chapters.

In Section 7.2.1 we present the performance figures for some micro-benchmarks. Section 7.2.2 presents a comparison between applications.

7.2.1 Micro-benchmarks

We have evaluated the three communication patterns that are used to perform operations on Orca objects: local object invocation, object invocation on a remote (single-copy) object, and a write operation on a replicated object. We use three benchmarks to evaluate the performance of these communication patterns: LOI, ROI, and GOI.

LOI (Local Object Invocation) measures the overhead of performing a local read operation on an Orca object. ROI (Remote Object Invocation) uses a single process that repeatedly performs operations on a remote object. GOI (Group Object Invocation) uses two processes that in turn perform an increment operation on a replicated integer object. After a process performed the increment, it calls operation *AwaitValue* with an integer parameter that contains the value that the replicated integer object will have after the other process performed the increment operation. The *AwaitValue* operation contains a guard statement that blocks the operation until the object has the same value as the parameter. This enforces that processes perform the increment operations in order. Measurements are performed for all pairs of processors, and the average latency of one invocation is computed.

The normal Orca runtime system is implemented on top of Panda, like the runtime system presented in this thesis [12]. Therefore, performance differences between the two systems are due to the overhead in the runtime system and in the interface between the runtime system and the object implementation. No compiler support is available for the extended shared object model, so all micro-benchmarks and applications are hand-written in ANSI-C [80]. For the normal Orca runtime system, the Orca compiler is used, which also generates ANSI-C. The performance of Orca applications compiled by the Orca compiler and running on one processor is close to the performance of C versions of the same applications. (In [12], the sequential performance of three applications written in C and in Orca is compared; one application is even faster in Orca than in C, whereas the two others are 2 and 17 percent slower.)

Table 7.1 shows the performance results of the micro-benchmarks on the DAS system (see Section 4.5). The GOI benchmark is executed on eight processors.

The LOI benchmark clearly shows the overhead of the weaver queue compared to the normal Orca runtime system. The Orca compiler generates a special test to see whether the object can be accessed directly. If so, the object is locked, and a function call to the operation is made without going through the runtime system. After the operation is finished, the lock on the object is released.

The extended object runtime system, on the other hand, handles a local invocation as all other invocations. A weaver is created and added to the weaver queue of the local processor (this does not require communication). The invoking pro-

	LOI	ROI	GOI
Myrinet			
Original Orca system	0.68	42.66	72.12
Extended object runtime system	6.45	66.14	95.54
FastEthernet			
Original Orca system	0.64	247.76	266.44
Extended object runtime system	6.48	288.41	286.56

Table 7.1: Micro-benchmark performance of Orca objects (latencies in μsec).

cess has to wait for the weaver to finish, and therefore makes a call to the weaver scheduler. The weaver scheduler finds the weaver and executes it. When the operation execution weaver is finished, the invoking process can continue. Since we use a dummy operation (i.e., the operation does not execute any code), this number presents only the runtime system overhead. For normal operations, the relative performance would be closer, because the execution time of the operation code is the same for both systems.

To evaluate the results of the ROI and the GOI benchmarks we first show the latencies of two Panda benchmarks that have the same communication behavior as the ROI and GOI benchmarks. The remote procedure call test directly implemented on top of Panda shows a round-trip latency of 35.8 μs on Myrinet. The group communication test achieves a group message latency of 64.6 μs. The performance of the normal Orca runtime system is very close to the corresponding communication latency, which indicates that the Orca runtime system is highly tuned [13].

Part of the difference in performance between the Orca runtime system and the extended object runtime system is the overhead introduced by the weaver queue and by the support of the generic runtime system layer for multiple runtime modules. Another part is caused by the fact that our prototype implementation needs more performance tuning. For example, the normal Orca compiler generates specialized marshaling routines, whereas the extended object runtime system uses a generalized marshal function that always traverses the argument list. Earlier experiments using the Amoeba port (i.e., using the slower MicroSPARC processors) connected by Myrinet (see Section 4.5), showed a performance improvement of 18 percent (about 60 μs performance gain, resulting in a ROI latency of 328 μs [13]).

In conclusion, the performance of our implementation on Myrinet is up to 35 percent slower for operations that require communication than the original Orca version. The communication latency on Myrinet, however, is very low in comparison with other communication layers. On FastEthernet, for example, the communication latency is about 250 μs, and the relative overhead of the extended object runtime system is less than 10 percent.

7.2.2 Application performance

The micro-benchmarks presented in the previous section measure the performance when processes perform only operations. Typically, application processes perform a certain amount of computation in between operations on objects. The ratio between the amount of computation and the amount of communication determines the *grain size* of the application. *Fine-grained* applications perform only a small amount of computation, and are therefore more influenced by the operation performance than *coarse-grained* applications.

Originally, Orca was more suited for medium to coarse-grained applications, because the first implementations of Orca (i.e., on Amoeba and Unix) had an operation latency in the order of a few milliseconds. With the arrival of faster network architectures for clusters of workstations, however, operation performance has improved up to the point that many fine-grained applications implemented in Orca can be executed efficiently.

To study the impact of the runtime system performance on applications, we implemented an extended version of the traveling salesman problem (presented in Chapter 2) and compared the performance to the same implementation for the normal Orca system. To alleviate the problem of search overhead, the initial bound is fixed to the final result of the current problem. This guarantees that all runs of the application will evaluate the same tree.

In order to study the effect of the grain size, we ran the application for different lengths of the partial path that is generates by the main process (the `hops` argument of function *tsp* in Figure 2.2 on page 17). For short initial paths, the worker processes have to perform a large amount of computation (resulting in a course-grained program,) whereas a long initial path implies that worker processes only have to perform a small amount of computation (resulting in a smaller grain size.)

A second effect that influences the performance is the number of operations that is performed locally to obtain the current minimum path length. The example implementation checks this bound in every invocation of the *tsp* function. An alternative is to get the current value in the function *worker* and pass this value to *tsp*. This results in a much lower number of local object invocations, but can also lead to a larger search overhead (i.e., more nodes to evaluate), because the bound is less strict. For our test situation, however, the bound is set to the final result, so there is no difference in the search tree. For our measurements, we use this optimization.

Figure 7.5 presents the performance figures for the traveling salesman problem application with different grain sizes. As the figure illustrates, the default grain

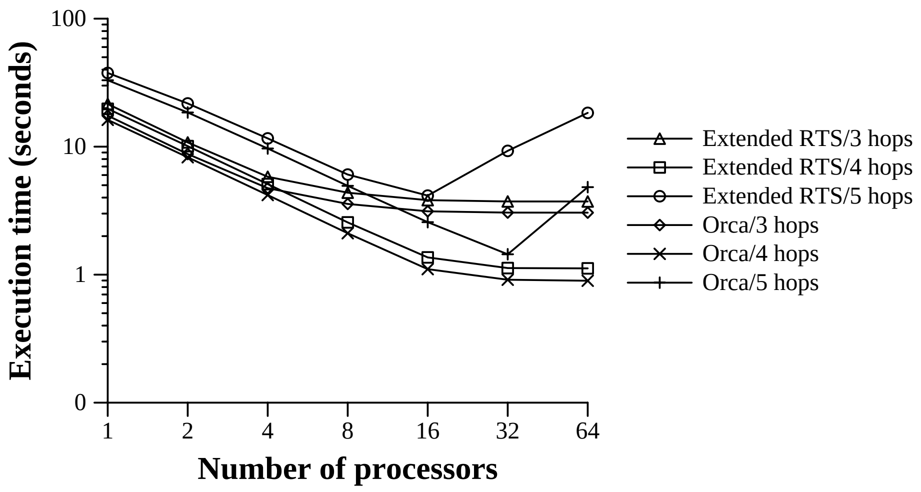


Figure 7.5: Performance of the traveling salesman problem application (18 cities) with different grain sizes.

size (i.e., split off when the partial path has length 4) obtains the best performance for both architectures. In addition, the original Orca system and the extended runtime system show similar performance. For the larger grain size (split after 3 hops), performance starts to degrade due to load imbalance.

For smaller grain size (split after 5 hops), the Orca runtime system is performing better than the extended runtime system for larger numbers of processors. This is due to contention on the job queue. Each worker process continuously retrieves jobs from the job queue, but since the jobs are small, a new job will be requested before the main process was able to generate a new one. Therefore, the worker processes queue up blocking on the job queue.

When the master process adds a new job to the job queue, the job queue object is updated. Therefore, all operations blocked on this job queue object will be retried, even though only one of these operations will succeed (i.e., the operation that retrieves the new job from the job queue.) This slows down the rate with which the master process can generate new jobs, which intensifies the problem. For example, with the split after 5 hops and using 64 processors, processor 0 performs about 3.6 million wakeups in the extended object runtime system. Since the Orca runtime system is faster in its handling of blocked operations (performance of a blocked operation is in the same order as the local object invocation) it suffers less from this performance problem.

One solution for this problem would be to have some advanced compile-time and run-time analysis to detect that all guard conditions are equal. In this case, after the first guard condition fails again, all other operations will not be retried, since it is already known that they will fail. Neither the Orca runtime system nor the extended objects runtime system implement such analysis. In addition, the single job queue would still become a bottleneck for larger numbers of processors.

Another solution was already pointed out in Section 2.2. Instead of having a single process generating jobs in a single queue, a number of processes generate jobs in multiple, independent queues. A flexible mechanism to handle these independent queues is to encapsulate them in a nested object. All processes access the independent job queues through this nested object. The performance implications of this solution are worked out in Chapter 9.

Summary

Implementing runtime support for Orca objects on top of the generic runtime system is relatively easy. The weaver abstraction provides the basic functionality to implement operations on local, remote, and replicated objects. The hook functions for Orca objects contain a small amount of code and are easy to understand. In addition, making a distinction between the invoker of an operation and the control of an operation allows us to implement runtime support for Orca objects without any knowledge of the other runtime modules.

There is some performance overhead compared to the performance of the Orca runtime system. Especially local object invocations are not optimized. The weaver queue and weaver scheduler add additional overhead for nonlocal invocations. However, the performance difference is not that large that it severely degrades application performance. Therefore, we can use the extended runtime system implementation to compare applications that only use Orca objects with applications that also use the extensions.

Chapter 8

Atomic Functions

This chapter describes the implementation of the runtime support module for atomic functions (see Figure 8.1). In contrast to the Orca objects module (see Chapter 7) and the nested objects module (which will be described in Chapter 9), the atomic functions module does not add a new object type. Instead, it only provides a mechanism to synchronize operations on an arbitrary set of objects. These objects may be supported by any other runtime module; only the application writer has to know which runtime module is associated with each object.

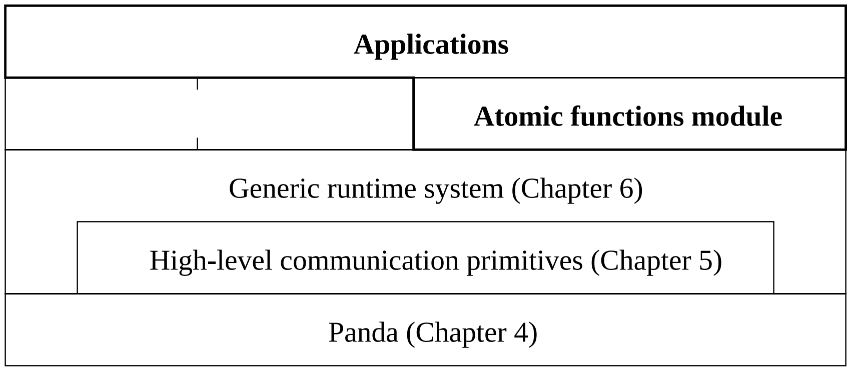

Applications
Atomic functions module
Generic runtime system (Chapter 6)
High-level communication primitives (Chapter 5)
Panda (Chapter 4)

Figure 8.1: The atomic functions module in the architecture overview.

The atomic functions module has only a single interface, namely the interface to the application. This interface provides the functions to register and create atomic functions, and to call operations on objects from within the atomic function. Also, this interface provides calls that can be used by the programmer or a compiler to specify data dependencies.

Section 8.1 describes the implementation of the atomic functions module. In Section 8.2, we present the performance of the atomic functions implementation.

8.1 Implementation

The implementation of atomic functions is based on the support provided by the generic runtime system. A single invocation weaver is used per atomic function.

This invocation weaver calls the code of the atomic function. Since only a single weaver will be running on a processor at a time, mutual exclusion is guaranteed. For condition synchronization, the synchronization mechanisms provided by the weaver abstraction and the generic runtime system are used.

During the execution of the atomic function, operations can be performed on the objects that are passed as shared parameter. The atomic functions module ensures that the operation is invoked on all processors that have a copy of the object, and that the other processors skip the operation invocation. Finally, the atomic functions module takes care of data dependencies that arise during the execution of the atomic function.

8.1.1 Interface to the application

Like process functions, atomic functions need to be registered at startup time. Therefore, an application that desires to use an atomic function calls *atomic_register* to register the associated code. A process can use the atomic function identifier that is returned by *atomic_register* to invoke an atomic function.

Objects that are passed to the atomic function as shared arguments can be invoked from within the atomic function. Since these objects are handled by other runtime modules, the atomic functions module has to guarantee that the processor set of the invocation weaver is sufficient to perform all operations within the atomic function. To meet this requirement, the invocation weaver of the atomic function is created on all processors that have a copy of *any* of the objects passed as shared argument. Since the processor set of these objects is known to the invoking processor, it can compute the union of these sets and use this as the processor set for the weaver. This ensures that the atomic function can perform all operations, since the maximum processor set that is required for an operation on an object is the processor set that contains all processors that have a copy of the object.

8.1.2 Object invocations

During the execution of the atomic function, operations on shared objects are invoked. In contrast to the Orca objects module, however, we do not try to minimize the processor set of the operation (see Section 7.1.1), but instead always perform operations on all copies of an object. The application interface of the Orca objects module uses the smallest processor set to minimize communication. For atomic functions, however, we assume that the result of an operation (i.e., the output parameters and/or the return value) will be used later on in the atomic function. Combined with the assumption that it is better to compute the result locally than to send the result later in the execution, we chose to perform the operation on all copies, even if it is a read operation. Note that this is a heuristic: it is possible that the result of an operation will be used on only some processors. In that case, we do too much computation, but it does not lead to additional communication.

```
atomic function copy(x, y: shared IntObject);
   r: integer;
begin
   r := x$Value();
   y$Assign(r);
end;
```

Figure 8.2: Atomic function that copies the integer value of object x to object y.

In the atomic function *copy* (see Figure 8.2), for example, the processor set of the weaver contains all processors that have a copy of either object x or object y. During the atomic function execution, the read operation on object x is performed on all processors that have a copy of object x. Figure 8.3 illustrates an example situation, in which object x has copies on processors 0 and 1, and object y on processors 1 and 2. After the read operation on object x has been performed, processor 1 has all the state to invoke the write operation on object y. Only processor 2 needs to receive the result from the read operation, either from processor 0 or processor 1.

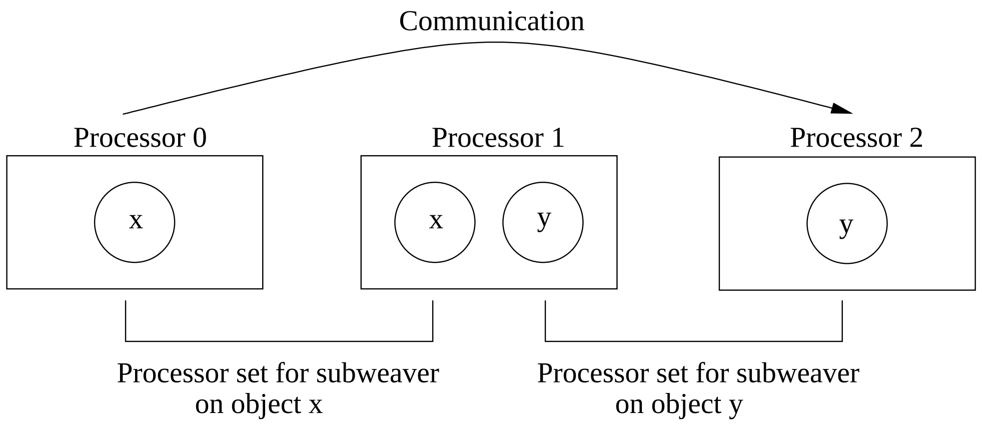

Communication
Processor 0
Processor 1
Processor 2
x
x
y
y
Processor set for subweaver on object x
Processor set for subweaver on object y

Figure 8.3: Processor sets for the subweavers that perform operations on object A and object B.

The processor set of an operation on an object within an atomic function is computed by calling *rts_oper_procset* with a maximum operation capability (defined by the generic runtime system) as argument. This processor set always contains the processors that are required to perform an arbitrary operation. The atomic function invocation weaver, however, can be running on a larger processor set. As described in Section 7.1.2 a runtime module does not have to deal with a weaver that runs on processors that do not have a copy of the object. Therefore, we have to restrict the invocation weaver to a subset of the processors that it is running on.

Figure 8.3 shows the processor sets that need to perform the read and write operations on object `x` and `y`, respectively. The weaver that performs the atomic function runs on all three processors. To restrict the invocation weaver to a subset of the processors that it is running on, we extended the generic runtime system with a function that creates a *subweaver*. A subweaver is a restriction of the current weaver to a subset of the processors that the weaver is running on. Since no new processors are involved in the subweaver and all the restricted processor set of the subweaver is determined locally by all involved processors, the subweaver can be created without communication. After a weaver instance determines the processor set of the subweaver, it checks whether its processor identifier is in this set. If so, the weaver encloses the code of the subweaver within calls to *subweaver_start* and *subweaver_end* (see Figure 8.4). Otherwise, the weaver simply skips the subweaver code and continues its execution.

```
set := subweaver processor set
if my processor identifier in set then
   subweaver_start(set, state)
   Perform subweaver code.
   subweaver_end(state)
fi
Continue execution of normal weaver code.
```

Figure 8.4: Implementation of subweaver creation in pseudo code.

Subweaver_start saves the current processor set and the current invoker set of the weaver. Space to store these processor sets is provided by the weaver (the `state` argument). Since *subweaver_start* and *subweaver_end* need to match, this state variable can be put on the stack of the function that creates the subweaver. This also allows subweavers to nest, i.e., a subweaver can create another subweaver. After saving the current processor set and the current invoker set, the subweaver processor set is copied into both. This has two purposes:

- Whenever the subweaver performs an operation on an object, the generic runtime system has the correct processor set to determine whether communication is necessary to wake up other weavers.

- If the code that the subweaver invokes registers a return dependency using *weaver_return_depend*, the generic runtime system will return the final result to *all* invoking processors. For Orca objects, this is not an issue, since all processors already computed the result on their local copy. For nested objects, however, this is not always the case, since the value that an operation returns can be the result of an operation on a subobject. By setting the invoker set to contain all invoking processors, this result is automatically propagated.

Subweaver_end restores the original processor set and invoker set. All processors that are not involved in the subweaver immediately continue with the normal weaver code.

For the atomic functions module, each object operation is the same. The only assumption is that all processors that have a copy of the object return the results of an operation, since the invoker set contains all those processors. Therefore, the atomic functions module can be integrated with all object runtime modules that apply to this assumption.

8.1.3 Synchronization

Two synchronization requirements have to be met by the atomic functions system: mutual exclusion and condition synchronization. Both can be directly mapped onto the weaver abstraction (see Section 6.3.2). Weavers already provide mutual exclusion by running only one weaver instance on a processor at a time.

The weaver abstraction supports condition synchronization in the form of the signal and wait functions. The problem we have to solve is that the atomic function has to be re-executed if any of the objects that the guard conditions depend on is changed (see for example Figure 2.9 on page 31). The atomic functions module uses the object synchronization support that the generic runtime system provides (see Section 6.3.3).

During the evaluation of the guard conditions, the atomic function performs non-blocking read operations on its shared object arguments. During this part of the execution, the generic runtime system builds the accessed objects set, which contains a list of objects that this weaver has performed operations on. If all guard conditions evaluate to false, the atomic function blocks. Only the objects that the atomic function has accessed up to then can change the result of the guard conditions. Therefore, the invocation weaver has to block until one of these objects is updated.

When the atomic function code determines that all guard conditions fail, it calls *atomic_wait*. This function calls *weaver_wait*, which places the weaver identifier in the pending weavers set that is associated with each object. This guarantees that the generic runtime system will wake up the invocation weaver whenever another weaver has updated any of these objects.

8.1.4 Resolving data dependencies

During the execution of the atomic function, data dependencies can occur between the result of one operation and the invocation of another operation. As described earlier, we try to minimize communication by invoking an operation on all processors that have a copy of the object, so that all those processors have the correct values in the result variables (i.e., the *OUTPUT* and *SHARED* parameters of the operation and the variable that contains the result of the operation). This does not guarantee, however, that all data dependencies can be resolved without communication, because objects can be on different processors.

Operations on objects generate values in the result variables. *Atomic_register* is used in the atomic function code to register these result variables. It therefore calls *weaver_register*, which does the registration in the generic runtime system.

To resolve all data dependencies, we use the *weaver_depend* and *weaver_synchronize* functions provided by the generic runtime system. As described in Section 6.4, the function *weaver_depend* can be used for three different data dependencies: from result variable to object invocation, from result variable to all weaver instances, and from result variable to all invoking processors. Therefore, the atomic functions module provides the following functions: *atomic_object_depend*, *atomic_data_depend*, and *atomic_return_depend* (see also Section 6.4).

When consistent results are required, the atomic function code calls *atomic_synchronize*. This function only calls *weaver_synchronize*, which takes care of all communication that is required to guarantee that all result variables are consistent on the processors that need them.

8.1.5 Caching operation results

When the atomic function weaver is signaled, it has to re-evaluate the guard expressions. During this re-evaluation, only those objects that have actually changed while the weaver was blocked can influence the new result of a guard expression. As described in Section 6.3.3, the generic runtime system maintains a set of objects per weaver (the *recheck* set) that have changed while the weaver was blocked. Since the weaver that is used to execute the atomic function is a real thread (instead of performing the atomic function code from the upcall thread), the results of the operations can be stored on the stack. During re-evaluation, only the operations on objects that have changed are executed. To implement this cache optimization, the atomic functions module provides two additional functions: *atomic_cached* and *atomic_wait*. These functions are mapped directly to the corresponding weaver functions *weaver_cached* and *weaver_wait*.

Figure 8.5 presents the caching implementation of the code that corresponds to the atomic function *terminate* of Figure 2.9. Before an operation on an object is executed on all processors that have a copy of the object, is called to check whether the object has changed. If the atomic function is executed for the first time, *atomic_cached* returns false. If the function has already evaluated the guard conditions before, however, the result variables `value` and `res` still contain the values from the previous invocation. For those objects that did not change in the meantime, the operation is not invoked again and no new result variables are registered. Therefore, the synchronize call detects that the result variables are still available on all processors that need a consistent copy, so no communication is required for these result variables. Note that operations that change the state of an object are not allowed in the guard conditions, so it is always safe to cache the results.

8.1.6 Multiple synchronization points

Atomic functions follow the same structure for condition synchronization as Orca, namely that the guard statements may occur only at the beginning of an operation. This is consistent with the model that we have in mind, namely to provide an atomic set of statements. One of the consequences of this design, though, is that it is not possible for a blocking operation to leave some state behind.

In Figure 2.9 on page 31, for example, the decrement operation on the `workers` object cannot be performed inside the `Terminate` function (before the guard statements), although that would be the most logical place. Instead, the decrement must be performed before `Terminate` is called. Technically, there is no reason not to allow an atomic function to perform a set of operations and then block, although the entire operation would no longer be atomic. Figure 8.6 shows an implementation of the atomic function *Terminate* in which the update of the active workers counter is also performed within the atomic function. When this function is invoked, it first performs a decrement operation on the active workers counter. (This implementation assumes that the active workers counter has already been initialized to the number of worker processes.) After the update is finished, the block of guard statements is evaluated. When all guard conditions fail, the function blocks until any of the objects accessed in the guard evaluation has changed. When the operation is resumed, only the guards are executed again; the decrement operation on the active workers object is never re-executed.

We have defined the semantics of an atomic function such as presented in Figure 8.6 as follows. All operations from the first statement to the first block

```
for(;;) {
    int value, val_id, res, res_id;

    if (!atomic_cached(q)) { /* Perform operation if not cached */
        if (atomic_local(q)) { /* Perform operation on local copy */
            args[0] = &res;
            ret = atomic_do_oper(q, JOBQUEUE_EMPTY, args);
        }
        /* Register result variable */
        res_id = atomic_register_result(&res, q);
    }
    if (!atomic_cached(workers)) {
        if (atomic_local(workers)) {
            args[0] = &value;
            ret = atomic_do_oper(workers, INTOBJ_VALUE, args);
        }
        val_id = atomic_register_result(&value, workers);
    }

    /* Register the data dependencies for both result variables,
     * and get the values of all result variables to all processors
     * involved in the atomic function.  */
    atomic_data_depend(res_id);
    atomic_data_depend(val_id);
    atomic_synchronize( );

    /* Evaluate termination condition */
    if (res == 0) {
        *af_res = 0; return OK;
    } else if (value == 0) {
        *af_res = 1; return OK;
    }

    /* Block the atomic function invocation. The guards will be
     * re-evaluated when either object q or object workers has
     * changed. All results from the previous guard evaluation are
     * preserved on the stack. */
    atomic_wait( );
}
```

Figure 8.5: Simplified C code for the caching implementation of the atomic function *terminate*.

```
atomic function Terminate(q: shared JobQueue;
                workers: shared IntObject): boolean;
begin
    workers$dec();
    guard not q$Empty() do workers$inc(); return false; od;
    guard q$Empty() and workers$value() = 0 do return true; od;
end;

process Worker(q: shared JobQueue; workers:  shared IntObject);
    job: JobType;
begin
    do
        while q$GetJob(job) do
            # handle job
        od;
    while not Terminate(q, workers);
end:
```

Figure 8.6: Implementation of the atomic function *terminate* with multiple synchronization points.

of guard statements are executed atomically. Then, all statements starting with a block of guard statements to the next block of guard statements are also an atomic unit. This allows us to add additional statements between sequences of guard statements. The start of the atomic function and each block of guard statements now represent multiple *synchronization points* within one atomic function. For the user, the behavior of an atomic function with multiple synchronization points is the same as invoking a sequence of atomic functions, where each atomic function contains the statements of the original atomic function between two consecutive synchronization points.

For performance, the main benefit of using multiple synchronization points within one atomic function is that only one invocation weaver has to be created. For example, the original implementation for the function *terminate* required additional invocations to increment and decrement the active workers counter. In the new implementation, only a single invocation weaver can perform the operation. This allows the programmer to trade off the benefits of collective computation (fewer invocation messages) with the drawbacks (more communication required for flow control.)

The primitives provided by the generic runtime system and the atomic functions module already allow atomic functions to have multiple synchronization points. Blocks of guard statements can be coded as a loop that is exited when any of the guard conditions has succeeded and the associated statements have been executed. Compared to the code in Figure 8.5, the statement `return OK;` would be replaced by `break;`. When none of the guard conditions evaluates to true, *atomic_wait* is called.

One implementation aspect of using multiple synchronization points within one atomic function is the condition on which the invocation weaver has to be resumed. In the original definition of an atomic function, all objects that have been accessed by the weaver can change the result of a guard condition. The generic runtime system therefore resumes the invocation weaver when any of these objects has changed. With multiple synchronization points, only those objects that are accessed during the evaluation of the guard conditions matter. Therefore, we have extended the generic runtime system with the function *weaver_restart*, which clears the accessed objects set of the weaver. By inserting this function as the first statement of the for loop that is used to evaluate the guard conditions (see Figure 8.5), only changes to the objects that are accessed from this point will resume a blocked invocation weaver.

8.2 Performance

In this section, we will evaluate the performance of the atomic function implementation. The main goal of atomic functions is to provide a higher-level programming model in which stronger semantics are enforced than in the original Orca model. Still, performance should be close to what can be achieved using Orca objects, otherwise application writers will not use atomic functions.

In Section 8.2.1 we will look at the performance of some micro-benchmarks. In Section 8.2.2 some applications that use atomic functions are compared to similar applications that use only Orca objects.

8.2.1 Micro-benchmarks

Although atomic functions are mainly intended to ease parallel programming, they also offer the application programmer the opportunity to exploit the collective computation model. An example that illustrates the potential performance benefits of the collective computation model is the following micro-benchmark. In this benchmark, an increment operation is performed on a set of integer objects that are distributed over the processors.

In the Orca version, the invoking process performs each increment operation consecutively (see Figure 8.7). In the atomic function version, the process invokes a single atomic function that performs all increment operations. Figure 8.8 shows the performance results for these two versions. As expected, the performance

```
function increment(a: shared array[integer 1 .. N] of IntObject);
begin
    for i in 1 .. N do
        a[i]$inc();
    od;
end;
```

Figure 8.7: Implementation of the function *increment*. For the atomic function version, the keyword `atomic` is added.

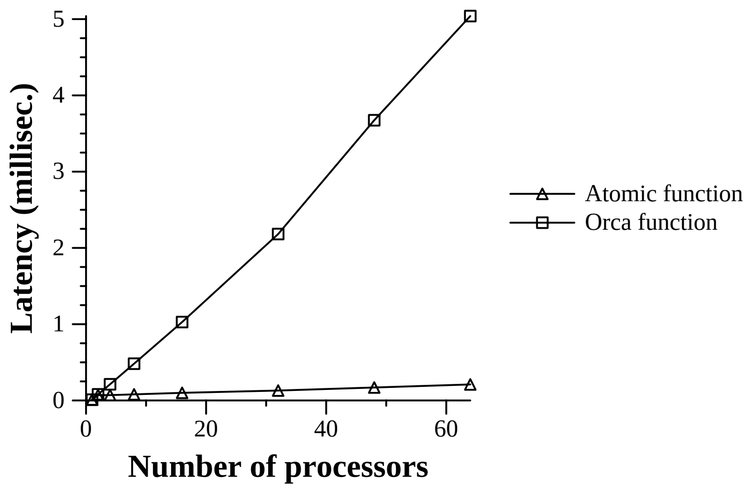


Figure 8.8: Updating single-copy distributed objects using Orca operations and an atomic function.

achieved by exploiting the collective computation execution model is much better than the Orca performance. The atomic function runtime system only has to start a weaver on all processors (which takes a single broadcast message), and then performs all increments in parallel. The Orca version, on the other hand, has to invoke each operation separately.

Our second micro-benchmark computes the minimum value of a set of objects, and assigns this minimum to all objects (see Figure 8.9). The communication pattern of this benchmark is typical for implementing weak consistency and load balancing strategies (see Chapter 2). Again, we compare the Orca version, which does not enforce atomicity, against the atomic function version.

For the atomic function, a number of possible implementations exist. In particular, the number of calls to *atomic_synchronize* influences the performance of the atomic function. In the *basic* implementation, we follow the simple rule that all

```
function minimize(a: shared array[integer 1 .. N] of IntObject);
min, t: integer;
begin
    min := a[1]$value();
    for i in 2 .. N do
        t := a[i]$value();
        if t < min then min := t; fi;
    od;
    for i in 1 .. N do
        a[i]$assign(min);
    od;
end;
```

Figure 8.9: Implementation of the function *minimize*. For the atomic function version, the keyword `atomic` is added.

involved processors compute the current minimum in each iteration. Therefore, a data dependency to all processors and a call to synchronize is generated after each *value* operation. In the second loop, each processor assigns this minimum to the object if the processor has a local copy of the object, so no communication is required.

A more advanced implementation of the atomic function *minimize* exploits the fact that the communication of all values can be postponed until after the first loop. For each object, space is reserved to store the result of the value operation. During the loop, each result variable is registered together with a data dependency to all involved processors, but the call to synchronize is postponed until after the first loop. After synchronization, each processor has all the result values and can compute the minimum value. The second loop is handled as in the basic version. We call this version the *gather-all* version.

Instead of computing the minimum value on all processors, the *reduce-all* version uses a reduction tree to compute the minimum value. When the root of the reduction tree has received the minima of its children and has computed the global minimum, it broadcasts this minimum to all involved processors. We use the annotation system that is described in Section 6.5 to inform the runtime system that this global minimum can be computed with the optimized reduction communication pattern. Since this communication pattern is based on the architecture-dependent parameters obtained from the LogGP performance benchmarks, we achieve very good performance for this operation.

Figure 8.10 shows the performance results of the Orca version and the different atomic function implementations. First, the figure illustrates the benefit

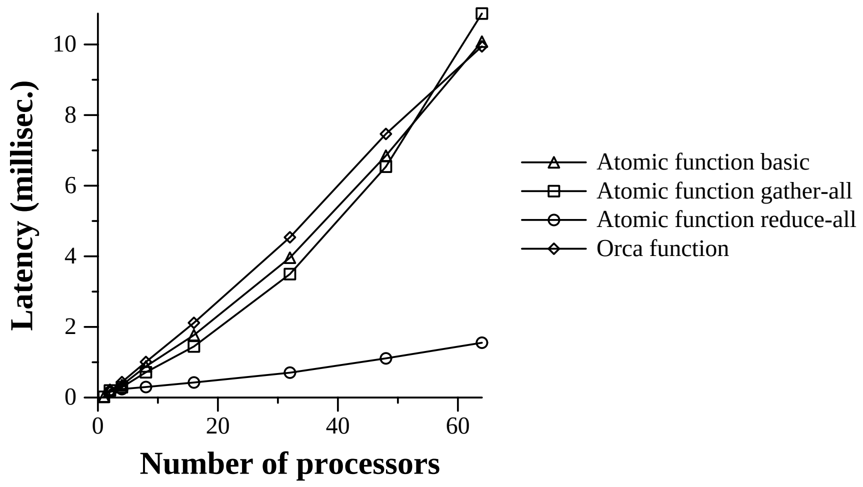


Figure 8.10: Performance results of the minimize benchmark.

of reducing the number of synchronization points. In the basic implementation, each *value* operation invocation waits until the result of the previous *value* operation has arrived. Therefore, the total execution time of the atomic function is the number of processors times the time to construct and execute the communication schedule and to send the broadcast message. In the gather-all implementation, only a single communication schedule is constructed, and all broadcasts occur when this schedule is executed. For smaller numbers of processors, the gather-all version gives a performance improvement of up to 20 percent. For larger numbers of processors, however, the concurrent broadcasts have a negative influence on the performance, due to network congestion and interrupt overhead. Therefore, the performance of the basic implementation is better than the gather-all implementation on 64 processors.

The basic atomic function implementation has about the same performance as the Orca implementation. Since the Orca implementation has to invoke each operation separately, all *value* and *assign* operations on a remote object require a remote procedure call. The atomic function implementation performs all assignments in parallel after the minimum value has been computed, which compensates the performance loss in the first phase.

Finally, Figure 8.10 shows the performance gain that is obtained by using the reduction communication pattern for computing the minimum value. For 64 processors, the reduce-all implementation is seven times faster than the Orca version.

Our final benchmark illustrates a worst-case scenario for the collective computation model. An array of objects is accessed to find the first object that fulfills a given condition (e.g., a job queue that is not empty). When such an object is found, the loop is terminated (see Figure 8.11). Since the loop condition depends on the result of the operation, all processors need to obtain this result before the next iteration. Therefore, no parallelism is possible, and each loop iteration involves a synchronization phase.

```
function global_dequeue(a: shared array[integer 1 .. N] of JobQueue,
                  j: out JobType) : boolean;
begin
    for i in 1 .. N do
        if a[i]$GetJob(j) then
            return true;
        fi
    od;
    return false;
end;
```

Figure 8.11: Implementation of the function *global_dequeue*. For the atomic function version, the keyword `atomic` is added.

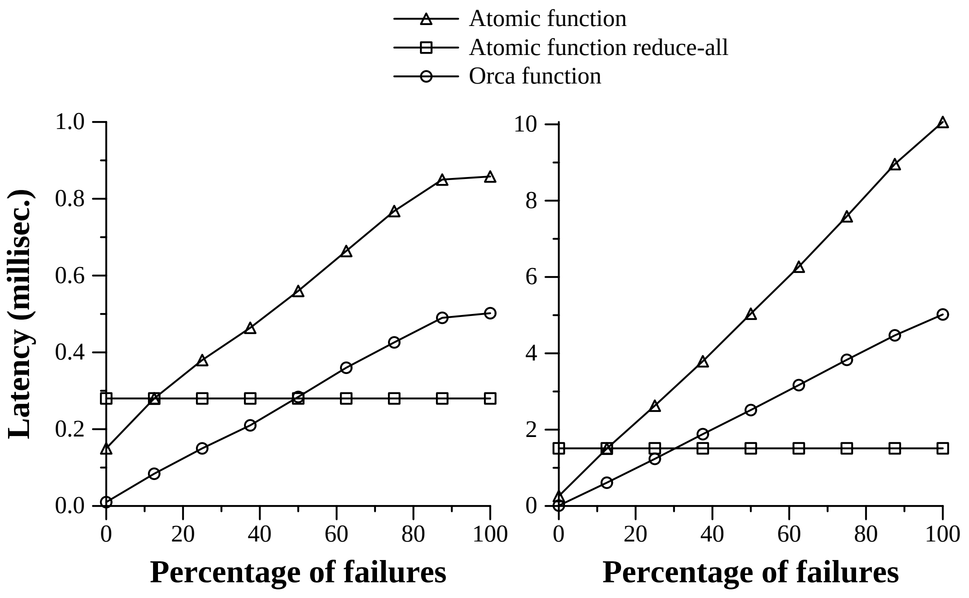
If the objects that are accessed in the loop are not replicated, this benchmark is especially suited for computation migration. During the execution of the loop, the execution migrates to the next object that is going to be accessed (see Figure 3.2 on page 39). If an object is found that fulfills the loop termination condition, only the currently active processor needs to know about this.

An interesting extension to our system, which has not been implemented, is to integrate computation migration phases within collective computation. Invocation of such a combined function would start as a normal collective computation invocation. Within this function, computation migration phases are indicated by the start computation migration and stop computation migration statements.

During the computation migration phase, execution is migrated when an object is accessed to the set of processors that have a copy of this object, passing along the state of the execution. These processors can perform all operations on this object, even operations that involve collective computation. Since only the active set of processors knows which object is going to be accessed next, however, no collective communication can be used for the migration messages. The initial set of active processors (i.e., before the first object access) can consist of all involved processors. After the first object access, only the active processors continue.

If a processor becomes inactive, it blocks until either the computation migration phase is finished, or until it receives a computation migration message (which contains the state of the atomic function) and it has to perform an operation on a local object. The computation migration phase finishes when the active set of processors reaches the stop migration phase statement. At the end of the computation migration phase, a single message is sent to wake up all nonactive processors, after which the collective computation continues as normally. This way, we have the benefits of computation migration (i.e., only those processors that need the results get it) together with the ability to deal with replicated objects.

Another implementation of the global dequeue benchmark using only collective computation is to first determine the state of all queues that can be accessed in the loop, and then to perform the dequeue operation. First, the loop determines which job queues are empty. The results of these queries are reduced to a single value which specifies whether a nonempty queue exists, and if so, which queue is the first that the original loop would have accessed. As in the second benchmark, this reduction can be implemented using a gather-all or a reduce-all communication pattern. Based on this reduced information, the job can be dequeued from this job queue or the global dequeue can fail. Note that this transformation needs an additional job queue method that returns whether the dequeue operation would succeed, or an advanced compiler that performs inter-procedural optimizations.



a) On 8 processors. a) On 64 processors.

Figure 8.12: Performance results of the global dequeue benchmark.

Figure 8.12 shows the performance results for the different versions of the global dequeue benchmark. We show the performance numbers for 8 and 64 processors, while we change the percentage of *GetJob* invocations that fail. In contrast to the previous benchmark, this benchmark does not have a global assignment phase that is well suited for the atomic function implementations. Therefore, the performance difference between the basic atomic function implementation and the Orca implementation clearly illustrates the worst-case overhead of the collective computation approach. Note, however, that the atomic function implementation provides stronger semantics (i.e., atomicity) than the Orca implementation. Although this property is not used in the benchmark code, it can be very important in more complex operations.

The reduce-all implementation, which uses the reduction communication pattern, shows an interesting tradeoff with respect to the Orca implementation. For 8 processors, more than half of the *GetJob* invocations must fail before the atomic function implementation performs better. For 64 processors, however, this break-even point is already around 30 percent. Since the reduction operation scales logarithmically while the Orca version scales linearly with respect to the number of processors, this tendency will continue for even larger numbers of processors.

8.2.2 Application performance

In this section, we will look at the performance of the atomic function implementation for two applications: the traveling salesman problem described in Chapter 2, and the 15-puzzle application. We chose the problem sizes small enough to stress the applications on larger numbers of processors, because this illustrates more clearly the performance differences between the Orca version and the atomic function versions. In addition, we present the performance improvements of the result-caching optimization and discuss some other optimizations.

The difference in performance of atomic functions and Orca objects of the traveling salesman problem application are due only to the difference in the way termination is handled. Therefore, we use a small problem size, namely 14 cities. This implies that the application quickly reaches the termination detection phase, and that the cost to determine whether a worker can finish has a profound effect on the application performance.

Figure 8.13 shows the performance results of the TSP application on Myrinet. The slowdown is due to the larger number of processors that have to synchronize at the end of the application. As the figure shows, the atomic function version performs only slightly worse than the Orca version.

As illustrated in Section 7.2.2, the scalability of the TSP application is limited by the single work queue. For large numbers of processors, this single queue becomes a bottleneck, and the master process is no longer able to generate jobs fast enough to keep all other processors occupied. One solution is to split up the single job queue in a job queue per processor. Each processor now maintains its own job queue, and only accesses another processor's job queue if the local job queue is empty. A worker process now needs to check whether all job queues are

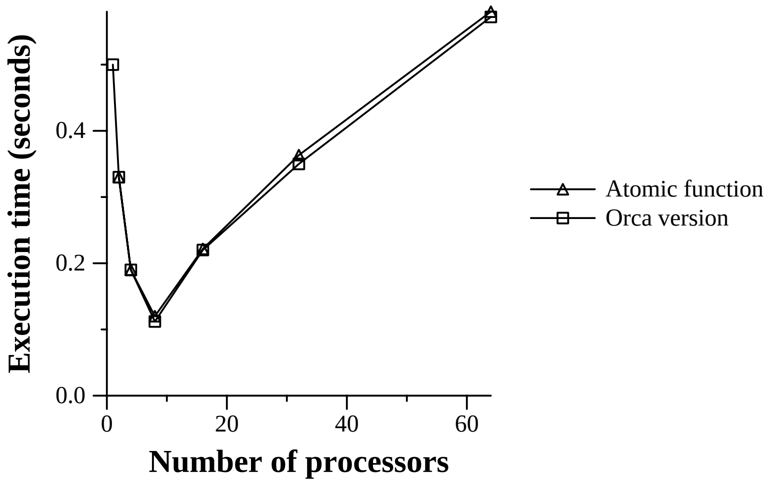


Figure 8.13: Performance of the traveling salesman problem application (14 cities) using only Orca objects or using an atomic function for termination detection.

empty and no more processes are active before it may terminate.

To investigate the performance of splitting up the job queue, we use another application that has already been implemented in Orca. This application, a 15-puzzle solver, uses an iterative deepening approach to find all shortest solutions to solve a given 15-puzzle (see Figure 8.14.) The 15-puzzle application is used in a graduate course on parallel programming, and is therefore already tuned carefully to illustrate the performance issues in parallel programming.

Each job in the 15-puzzle implementation consists of a description of all squares and the number of moves that have been made so far. When a worker fetches a job, it generates all following positions by moving the empty square up, down, left, or right, if possible. These movements result in new jobs, which are added to the local job queue. Then the worker process fetches the next job from the local job queue. A job is not expanded if it cannot result in a solution in the number of steps that is currently searched for. If no solutions are found for a given number of steps, the limit is increased and the complete search is started again.

When the local job queue becomes empty, either a new job has to be fetched from another processor's job queue or the process has to terminate. The decision which job queues to query for new jobs has a profound impact on the performance of this application. Querying too few remote job queues causes load imbalance, because it takes a while before each processor has enough work to perform. Querying too many remote job queues causes performance problems at the end of an iteration, when only a few jobs are still available. An alternative is

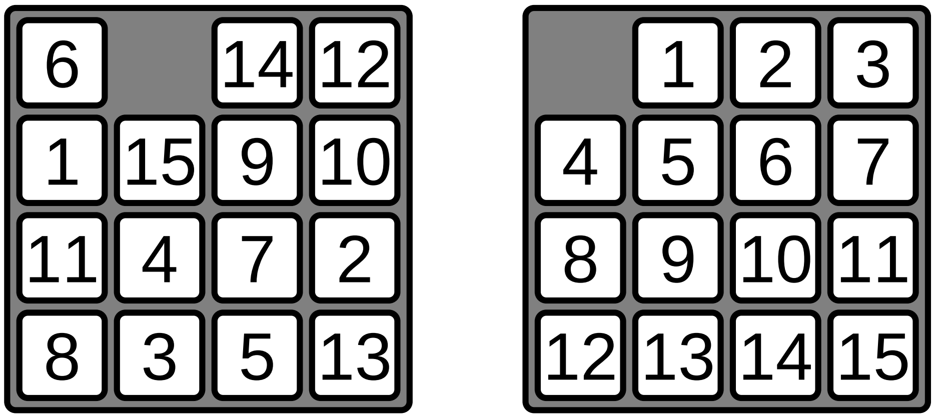


Figure 8.14: Example of the 15-puzzle. The first picture shows the start position used in the performance measurements. The second picture shows the final result that needs to be achieved.

not to steal jobs, but to register idle processors and pass new jobs directly to those processors.

For this evaluation, we use the Orca implementation, which uses job stealing from those processors that are numbered as $pid + 2^i \pmod{nrprocs}$, where $0 \leq i < \log nrprocs$. This way, at most $\log nrprocs$ remote queues will be accessed for each local dequeue operation that fails. For termination detection, the Orca implementation uses a specialized *idle* object implementation, which contains state for the number of idle workers, whether the iteration is finished, and an estimate of the number of jobs that is globally available for this iteration. To prevent the system from updating this (replicated) object after each job expansion, the estimate of the number of jobs in the system is only updated if any of the processors is idle. If a worker cannot find a job on any of the job queues, it blocks until either the iteration is finished (i.e., all workers are idle) or until another worker announces some new work.

The atomic function implementation is based on the same load balancing scheme as the Orca implementation. The complex idle object, however, has been replaced with the atomic function terminate (see Figure 8.15). This is basically the same implementation as presented in Section 8.1.6, only the nested object has been replaced by an array of normal Orca objects, and the terminate function returns not only the boolean value, but also the job if the dequeue operation succeeds.

The implementation of the atomic function *terminate* offers ample opportunity to exploit the cache optimization. By inlining the code of the function *GetJob*, the *terminate* function can keep a cached state of each dequeue operation on the job

```
function GetJob(arr: array of shared JobQueue;
        j: out JobType): boolean;
    index, step: integer;
begin
    step := 1;
    while step $<$ NCPUS() do
        index := (MYCPU() + step) mod NCPUS();
        if arr[index]$GetJob(j) then return true; fi;
        step *:= 2;
    od;
    return false;
end;

atomic function Terminate(arr: array of shared JobQueue;
            workers: shared IntObject;
            j: out JobType): boolean;
begin
    workers$dec();
    guard GetJob(arr, j) do workers$inc(); return false; od;
    guard not GetJob(arr, j) and workers$value() = 0 do return true; od;
end;

process Worker(arr: array of shared JobQueue; workers:  shared IntObject);
    job: JobType;
begin
    while not Terminate(arr, workers, job) do
        do
            # handle job
        while arr[MYCPU()]$GetJob(job);
    od;
end:
```

Figure 8.15: Termination detection in the atomic function implementation of the 15-puzzle.

queues.

A final implementation involves a specialized *synchronizer object* for synchronizing all processors. In the normal IntObject, each increment and decrement operation invokes the function *rts_object_updated* to inform the generic runtime system that the object has changed. This function triggers the wakeup of all weavers that are pending on the pending weavers set for this object. For our synchronizer object, we have adapted the decrement operation to only call *rts_object_updated* if the object state reaches zero. Therefore, only the last invocation of the atomic function *terminate* after the job queue has become empty triggers the wakeup of all pending weavers that are blocked on the guard expressions.

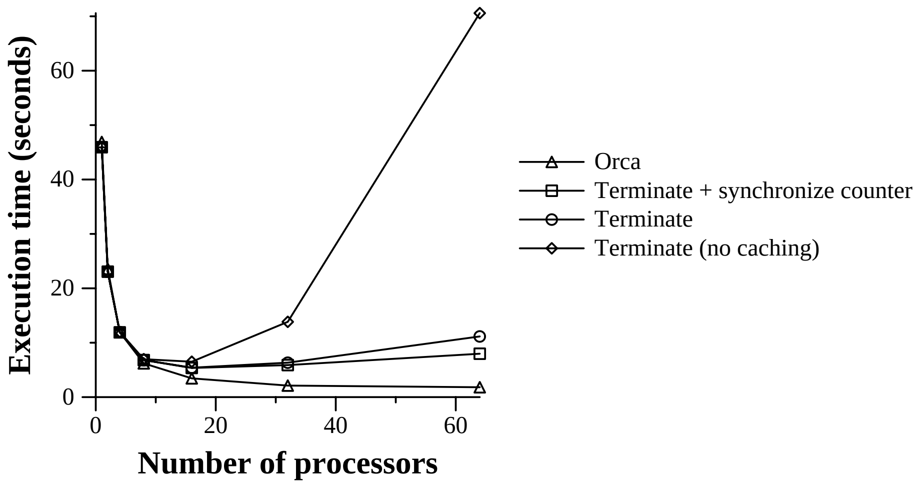


Figure 8.16: Performance numbers for the different implementations of the 15-puzzle on the test position given in Figure 8.14.

Figure 8.16 shows the performance of the 15-puzzle implementations on Myrinet. For up to 16 processors, the atomic function implementations can keep up with the Orca version. However, for 32 processors and more, the atomic function implementations suffer from the stronger semantics that the atomic function provides. Since the atomic function ensures that the state of the `worker` object does not change after the *dec* operation, all invocations of the *terminate* become serialized, whereas in the Orca implementation termination detection can occur in parallel. Although the atomic function implementation is easier, its atomicity guarantee is not required for this application. This illustrates the trade-off between programmability and performance.

If we look at the atomic function versions, it is clear that the caching optimization improves performance significantly. Each change of the `worker` object causes the pending *terminate* invocations to be re-evaluated. Instead of performing the *GetJob* operation after every update on the `worker` object, which in this case requires a data dependency synchronization of the boolean result to all processors, only the new value of the `worker` object needs to be obtained, which does not require communication since the worker object is replicated. Therefore, waking up a pending *terminate* invocation only causes communication if the job queue object is changed. On 64 processors, this gives a performance improvement of 85 percent.

Finally, the synchronizer object reduces the number of wakeups on pending *terminate* invocations. Instead of waking up the pending *terminate* invocations after every state change, a wakeup only occurs if the new value can change the result of the evaluation of the guard condition. On 64 processors, the performance improvement is almost 30 percent.

Summary

The atomic functions module implements an application interface that allows the programmer to write atomic operations on an arbitrary set of objects. Since the atomic functions module only uses functions provided by the generic runtime system, it is independent of the actual object type. Therefore, Orca objects and nested objects can be used in atomic functions.

The synchronization requirements of the atomic functions module can easily be implemented using the functionality provided by the generic runtime system. The only extension to the generic runtime system that does not follow directly from the collective communication model is the ability to create a subweaver. This extra functionality to the generic runtime system is a simple and efficient extension.

We have extended the atomic function as defined in Chapter 2 with the notion of *multiple synchronization points*. This extension allows the programmer to perform state changes to the objects passed as parameter before blocking on the guard conditions. Multiple synchronization points can be used both as a performance optimization as well as for clarity.

The performance of our atomic functions implementation is good. For some situations, the benefits of using collective computation gives even better performance than the original Orca system. Applications that benefit from the stricter semantics that is provided by atomic functions perform well compared to the versions in which only Orca objects are used. For applications that can be implemented with only Orca objects, performance differences can be larger, but the Orca versions are typically harder to implement correctly.

Chapter 9

Nested Objects

This chapter describes the runtime module for nested objects. Like the Orca object module, this module adds a new object type to the extended object runtime system. Nested objects can contain arbitrary objects as subobjects. The nested objects module is not aware of the object type (i.e., the runtime module that is associated with the object) of a subobject. All interaction between the nested objects module and the subobject is handled by the generic runtime system (see Figure 9.1).

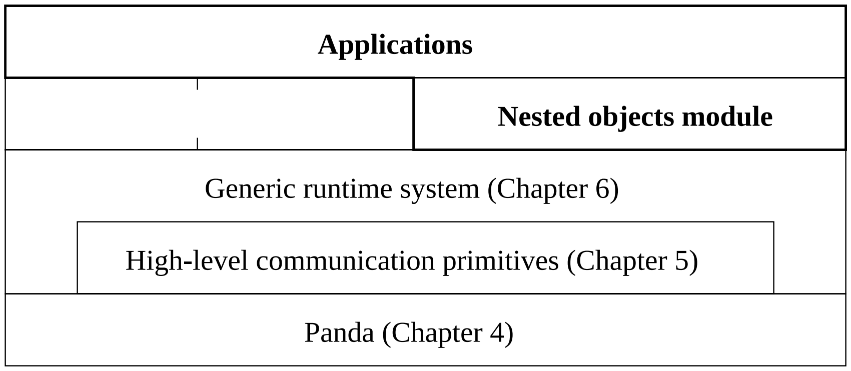

Applications
Nested objects module
Generic runtime system (Chapter 6)
High-level communication primitives (Chapter 5)
Panda (Chapter 4)

Figure 9.1: The nested objects module in the architecture overview.

Since the nested objects module implements a new object type, it has two interfaces: one to the application to build and invoke operations on nested objects, and one to the generic runtime system to implement nested objects. These interfaces will be described in Section 9.1. In Section 9.2, we present the performance of the implementation of the nested objects module and show some applications.

9.1 Implementation

Similar to the atomic functions module, the nested objects module uses a single weaver invocation to invoke an operation on a nested object. This weaver invokes the operation on the root object. Within this operation, other operations may be invoked on subobjects of the root object using the same weaver. If a subobject is also a nested object, its operation may also invoke operations on its subobjects.

This way, a tree of operation invocations is invoked using a single weaver invocation.

The major difference between nested objects and atomic functions is that the exact set of objects that is going to be accessed is not known when the operation is invoked. In an atomic function, the objects are passed as parameters. In a nested object operation, on the other hand, the only known object is the root object; there is no information which subobjects are going to be accessed. In addition, the performance requirements for nested objects are more stringent, especially for operations that update only local objects.

The implementation of nested objects can be regarded of a mixture of an optimistic and a pessimistic approach. If the root operation does not require global synchronization (i.e., the state is not updated and only a single subobject is accessed), the invocation weaver only runs on a single processor, preferably the local one. Since this might not be the right processor or the right set of processors, this invocation may fail. Since no state changes have occured yet, rolling back is easy. On the other hand, if global synchronization is required, the invocation weaver is executed on a set of processors that is sufficient to perform the operation safely, like the invocation weaver of an atomic function.

9.1.1 Interface to the application

To perform an operation on a nested object, the application calls *nested_do_oper*, with the object and the operation arguments as parameters. This function is implemented exactly as *orca_do_oper* (see Figure 7.3); the only difference is that the invocation weaver needs a stack, because the weaver may need to communicate to resolve data dependencies due to operations on subobjects.

Like the Orca module, the nested objects module also supports the `READ_CAP` and `WRITE_CAP` capabilities. To perform an operation that only reads the status of a nested object and accesses (i.e., reads or writes) at most one subobject, a read operation capability is required. For an operation with read capability, it suffices to run the invocation weaver on one of the processors that has an instance of the object (i.e., the optimistic case). For an operation with write capability, all processors with a copy of the object need to participate in the weaver invocation (i.e., the pessimistic case).

The nested object runtime system also introduces a third capability, the `SYNC_CAP` capability. This capability, which is ordered between the read and the write capability, captures the fact that an operation that reads a nested object and accesses more than one subobject only needs to synchronize on those processors that have a copy of these subobjects. To estimate this set, the function *nested_object_operset* returns at least the union of the processor sets of all subobjects for the `SYNC_CAP` capability. Only those processors need to participate in the weaver invocation to guarantee a sequentially consistent execution. Since the model restricts subobject instances to only those processors that have a copy of the parent object, the resulting processor set is a subset of the set that would be returned for a write capability (see Section 2.3).

For a read operation (i.e., the optimistic case), the capability does not represent enough information to always determine the correct processor set in advance, because the required processor set also depends on the subobject that is going to be accessed. To illustrate this, consider a read operation on a nested object that accesses a single subobject (see Figure 9.2 for an example situation). When the operation is invoked, the invoking process may reside on a different processor than the subobject that is going to be accessed. Since the invocation weaver is initially invoked on a single processor (preferably the local processor if it has a local copy of the nested object), this may be the wrong processor. In the example, processor 0 performs an operation on the root object that accesses subobject Y, which only has a copy on processor 1.

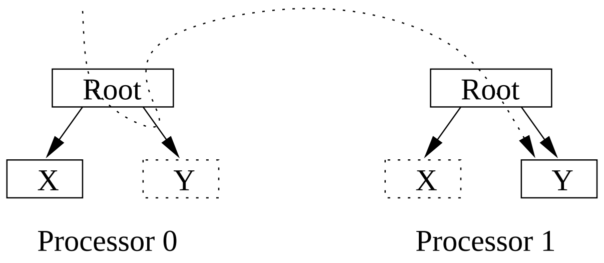


Figure 9.2: Invoking an operation on subobject `Y` from processor 0.

Before invoking the operation on the subobject, the actual processor set of the weaver is compared to the processor set that is required. If the weaver processor set does not overlap the required processor set, the required processor set is returned together with return status `PROCSET` (see Section 6.2.3). Like *orca_do_oper*, *nested_do_oper* detects that the operation invocation failed, and starts a new weaver on the returned processor set. In the example, a new weaver is created on processor 1, which invokes the operation on the root object and can access subobject Y.

This mechanism allows us to achieve our goal for nested objects, namely that we can access subobjects efficiently without executing operations on all processors that have a copy of the root object. As long as an operation consists of a chain of operations that do not update their object and that only access a single subobject, only one processor needs to be involved. When an operation on a subobject does not apply to this condition, the invocation processor set is compared with the processor set that is required. If the invocation processor set contains the required processor set, the operation can be completed. If the invocation processor set is insufficient, the required processor set is returned to the invoker, which will create a new invocation weaver on this processor set. This invocation weaver is executed on enough processors to obtain a `SYNC_CAP` or `WRITE_CAP` capability for the operation that failed. From then on, the operation cannot fail anymore due to an insufficient capability.

The most prominent example of an efficient nested object operation is a nested object with a subobject per processor. Accessing the local subobject without changing the root object requires no communication, while the atomicity of all operations on the root object is preserved.

9.1.2 Interface to the generic runtime system

In Section 6.2, we described the interface between the generic runtime system and a model-specific runtime system. For the nested objects module, these functions are implemented as follows. As for the Orca objects module, all functions correspond to the hook functions, except that the prefix *meta* is replaced by *nested*.

Nested_info_size returns the size of the runtime information that is associated with each nested object. As for Orca objects, this state consists of only a replication processor set. This processor set is initialized and destroyed by *nested_object_create* and *nested_object_destroy*, respectively.

Nested_object_operset returns the initial processor set that is used to create the invocation weaver, based on the runtime system information (i.e., the replication set) and the capability of the operation that is to be performed. As *orca_object_operset*, read operations are invoked on one processor (preferably the local processor) and write operations on all processors that have a copy of the object (i.e., the processors in the replication set.) For operations with the `SYNC_CAP` capability, we do not have to invoke the operation on all processors, but only on those processors that have any of the subobjects. (There must be more than one subobject that the operation can access, otherwise it would not need the `SYNC_CAP` capability.) The current implementation simply uses the replication set, but it is trivial to maintain a second set (the *subobject set*) in the object runtime system state that contains all processors that have a copy of a subobject. If this information would be maintained, this subobject set can be returned for a `SYNC_CAP` operation.

When *rts_object_oper* is invoked on a nested object by an invocation weaver instance, the generic runtime system calls *nested_object_oper*. Like *orca_object_oper*, this function performs three tasks: it determines the capability of the invocation weaver, it creates the weaver to invoke the operation, and it determines the new processor set if the operation fails due to an insufficient capability.

Determining the capability of the invocation weaver is similar as in the Orca objects module. If the `SYNC_CAP` would be treated separately using the subobjects set, it would be taken into account as follows. If the invocation weaver set contains the subobjects set, the weaver has at least `SYNC_CAP` capability. If the invocation weaver set also contains all replicas, the capability is `WRITE_CAP`.

During the invocation of the actual nested object operation, the operation may invoke operations on its subobjects. Since the subobjects that are going to be accessed are only known at runtime, there is no way to determine the correct processor set in advance. As explained in the previous section, the invocation on the subobject may fail. When this occurs, the operation returns `PROCSET` and a processor set that suffices to perform the operation on the subobject (see Section 7.1.2). This return status and the processor set are passed to the generic runtime system, which passes this information to the invoking processor. The invoking processor (i.e., the *nested_object_oper* function) checks the return status and determines that the operation failed. The processor set is then used to create a new weaver invocation (see Figure 7.3).

9.1.3 Synchronization

As in the atomic functions module, mutual exclusion and condition synchronization are both handled by the weaver abstraction. Mutual exclusion is guaranteed by running only a single weaver on a processor at a time and the total ordering in which weavers are executed.

For condition synchronization, a problem occurs if an operation on a subobject blocks. If this operation is part of a guard evaluation, it is not allowed to block, since this would prevent the evaluation of the other guard expressions. If the operation is not part of a guard expression, there are two solutions. The first solution is to actually block the operation on this subobject. This solution is hard for the programmer to understand, because only updates on this subobject will continue the operation on its root object. A consequence of this approach is that a state change of another subobject that causes another guard expression to succeed will not be detected.

A second solution is to abort the operation and try another guard expression, if available. This requires that all state changes are rolled back to the state in which the operation started. If all guard alternatives fail or block during their execution, the operation as a whole blocks. In the existing Orca runtime system, the same problem occurs for objects that are implemented using other objects. In this system, the roll back is implemented using compiler support for saving and restoring the state [8]. A consequence of this approach is that all state changes to any of the subobjects that enable a guarded expression are detected.

The semantics introduced by the first solution is confusing, and since we do not have compiler support for the extended object runtime system to implement the second solution, our prototype implementation does not allow a subobject operation to block; only the root object may contain guard expressions. This does not restrict the class of applications that we can write, but it requires that the programmer of a nested object knows which operations can block and avoids these operations within a nested object. The guard expression in an operation on a subobject can always be moved to the root object, by returning a boolean that indicates whether the original operation would have blocked.

If an operation on a root object contains guard expressions, these conditions are evaluated until a guard condition succeeds or all conditions have failed. If a guard condition succeeds, the corresponding guard statements are executed. When all guards failed, the operation calls *nested_wait*, which calls *weaver_wait*. *Weaver_wait* suspends the invocation weaver, and also adds the weaver to the pending weavers set of each object that the weaver has accessed. When any of these objects is updated, the generic runtime system will signal the invocation weaver. When the weaver becomes the running weaver (i.e., is not preceded by any other runnable weavers in the weaver queue) the guard conditions are re-evaluated.

Similar to atomic functions, there is no need for the nested objects implementation to restrict synchronization to a single block of guard statements at the beginning of the operation. By allowing multiple synchronization points within a single operation, such operations can be implemented more efficiently, because only a single invocation weaver needs to be created.

9.1.4 Collective computation issues

In Chapter 8, we discussed two implementation issues of atomic functions that are related to the collective computation nature of weavers: resolving data dependencies and caching operation results. For the execution of a nested object operation that accesses multiple subobjects, the same issues arise. These issues are solved in the same way as for atomic functions.

The interface to define data dependencies within a nested object operation is similar to the interface for atomic functions (see Section 8.1.4). Result variables are registered to the runtime system using *nested_register*. Using the variable identifier that is returned, data dependencies can be registered using *nested_object_depend*, *nested_data_depend*, and *nested_return_depend*. *Nested_synchronize* resolves these data dependencies.

Caching operation results is implemented similarly to the implementation in atomic functions. Each invocation on a subobject within a guard condition evaluation is encapsulated with an if statement which checks (using the function *nested_cached*) whether the operation has already been executed before and if so, whether the results of the previous invocation are still valid.

9.1.5 Partitioned objects

One of the problems of Orca that was identified in Chapter 2.2 was that Orca lacks the ability to express the partitioning of an object. Two solutions were described for this problem: partitioned objects and nested objects. Nested objects solve this problem in a way that gives the programmer full control over how the partitioning takes place by using subobjects. For regular applications, however, the partitioned object model is more suitable, because it removes the burden of managing communication from the programmer.

Below, we will show that the nested object model can be used to implement partitioned objects. We do not claim that this is the most optimal mapping, but we want to illustrate that the collective computation execution model provides all the support to implement partitioned objects.

To illustrate the mapping of partitioned objects to nested objects, we use an example application, successive overrelaxation (SOR). SOR is an iterative method for solving discretized Laplace equations on a grid. The main data structure is a 2-dimensional array that contains all the grid points. During each iteration, each nonboundary grid point is updated using the average value of its four neighbors scaled by a weight factor (the *relaxation parameter*). This process terminates if, during the last iteration, none of the grid points has been changed by more than a certain quantity.

The parallel version is based on the parallel Red/Black SOR algorithm. The algorithm treats the grid as a checkerboard and alternately updates all black points and all red points. Since each point only has neighbors of the opposite color, there are no data dependencies within one iteration. Therefore, the algorithm can easily be parallelized.

In the Orca version, the grid is partitioned among the available processors by assigning to each processor a set of consecutive rows. Each grid partition is contained in a local two-dimensional array in the corresponding worker process. During execution, some of the grid points are available on one processor, but are required on another processor for the computation of the average of the neighbors. Therefore, a worker process has two objects to exchange boundary grid points with its neighbor (see Figure 9.3). Basically, the Orca implementation uses the exchange objects to implement a specialized form of message passing.

In the partitioned object version, the whole grid is contained within a single object. The programmer uses annotations to specify how this object is partitioned and how these partitions are distributed over the processors [21]. In this version, each partition consists of a row of the matrix. Based on these annotations, the compiler would generate code that operates on a single partition of the grid. In

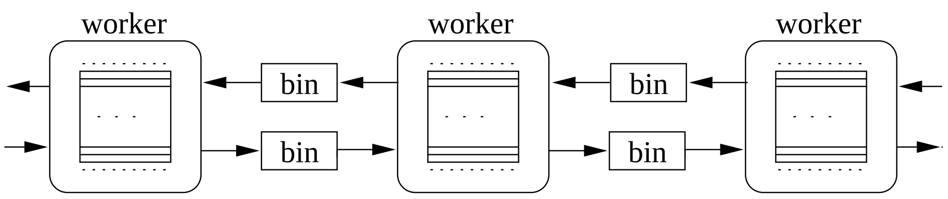


Figure 9.3: Objects used in the Orca implementation of Successive Overrelaxation (taken from [8]).

addition, the compiler computes the data dependencies for each partition update and generates code to send and receive the neighboring partitions that are required. In short, the partitioned object version removes the responsibility of resolving data dependencies from the programmer.

To use nested objects as a target language for this application, we use the following mapping. For each partition, a *data object* is stored as subobject of the nested object. This data object is assigned to the processor that is responsible for the computation of the elements in this partition. Since all partitions have the same elements and the same dimensions, only a single object type has to be defined for data objects.

To resolve data dependencies, each data object has operations to fetch and put the elements of a partition. Before performing the data manipulations, each processor executes the fetch operation on its local partitions and stores the result in a array of partitions within the operation. For each result, a data dependency is registered to the subobject that contains the corresponding neighbor partition. Then, the operation calls *synchronize*, which resolves all data dependencies. This causes all partitions to move to the corresponding neighbor partition. When a processor has received all data, it can perform the local update operation on its partitions. No explicit message passing emulation is required outside the nested object.

At the end of each iteration, termination detection needs to take place. Each update operation also returns whether this part of the grid fulfills the stop condition or not. These boolean results are combined to a global boolean condition which indicates whether all partitions fulfilled the stop condition. Due to the regular nature, an optimized reduction function can be used (see Section 6.3.3).

To summarize, it is possible to translate parallel operations on partitioned objects to an operation on a nested object. The original runtime system for partitioned objects, Hawk, provides several optimizations to reduce synchronization overhead by starting the computation on a partition when its boundary elements have arrived [21–23]. Since one processor can have multiple partitions, this allows overlap between computation and communication. Due to the synchronous nature of the communication primitives that are defined for weavers, such optimizations are not possible in our model.

9.2 Performance

In this section, we will evaluate the performance of the nested objects implementation. In Section 9.2.1, we give micro-benchmark performance results for an optimized accumulator object. In Section 9.2.2, we will look at the performance of some applications that use nested objects.

9.2.1 Micro-benchmarks

To illustrate the use and performance of nested objects, we use the accumulator object (see Figure 9.4). The accumulator object has operations to increment and to read a counter. This object is implemented using one subobject per processor, in which a local counter is maintained. The increment operation on the accumulator increments only the local counter subobject (specified by the parameter `index`), without doing any communication. The *value* operation atomically sums all counters.

This accumulator object is intended for applications that access a counter with a low read/write ratios (i.e, the counter object is incremented more often than read). Such a counter can be used, for example, to maintain statistics for dynamic load balancing. Note that in the original Orca model, writes are always at least as expensive as reads (for single-copy objects) or more expensive than reads (for replicated objects). Therefore, the Orca model does not provide suitable support for objects with a low read/write ratio.

The root object of the accumulator consists of an array of subobjects, one for each processor. Each subobject contains a single integer. An increment operation reads the root object to find the local subobject. Since the root object is not updated and only a single subobject is accessed, it is not necessary to globally synchronize this operation. Therefore, the increment operation on the local subobject can be performed without communication.

To read the value of the accumulator object, the sum of the values of all subobjects needs to be computed in an atomic way. Therefore, the value operation has the `SYNC_CAP` capability, and the resulting processor set includes all processors. Within the *value* operation, the value of all subobjects are summed up and returned to the invoking process. During the execution of the value operation on the root object, no local increment operations are permitted, since this would violate the atomicity property of the nested object model.

```
object implementation accumulator;
    import IntObject;
    c: array[integer 1 .. N] of IntObject; #subobjects

    operation inc(index: integer);
    begin
        c[index]$inc();  # increment local counter
    end;

    operation value(): integer;
        sum: integer;
    begin
        sum := 0;
        # Add all counters in one atomic operation
        for i in 1 .. N do
            sum +:= c[i]$value();
        od;
        return sum;
    end;
end;
```

Figure 9.4: Accumulator object with a subobject on each processor.

The accumulator object performs well for low read/write ratios, but very poorly for high read/write ratios. To improve the performance for high read/write ratios, we extended the accumulator object with an integer attribute that caches the sum, together with a boolean flag that indicates whether the cached value is still valid. The read operation first checks this boolean, and if the cache is still valid, the operation returns the cached value directly. Since this operation only requires read access on the root object, it does not require communication. If the cache is invalid, the local invocation fails due to an insufficient capability, and a global invocation is used to compute the sum. Before returning the sum, the cache is updated and the boolean is set to true.

The write operation also checks the boolean flag. If the cache is valid, the root operation fails due to an insufficient capability. The global invocation not only updates the counter object of the invoker, but also invalidates the boolean flag. If the cache was already invalid, the write operation proceeds as before, without any communication.

Figure 9.5 shows the performance of using a single-copy Orca object, a replicated Orca object, and a nested object with and without caching of the sum value. On all processors a process is executed that performs 1000 read or write operations with the given read/write ratio. For high read/write ratios, the overhead of the accumulator object is large. Each read operation requires communication to read the values of all the subobjects, since they are distributed over all processors. However, when the read/write ratio drops below 1/20, the accumulator object becomes much more efficient than the Orca objects, since all increments can be performed locally and in parallel. Caching the sum reduces the overhead of the read operation, making the nested object better suited for applications in which the access pattern is not regular.

9.2.2 Application performance

In this section, we will look at the performance of three applications: All-pairs Shortest Path, the 15-puzzle, and Successive Overrelaxation.

All-pairs shortest path

One of the most commonly used communication patterns that cannot be implemented efficiently using a single Orca object is message passing. If all messages are stored within the state of a single object, all processors have to write this object to insert and retrieve their messages. Both the send and receive operation have to change the state of the object to indicate that a message has been delivered to or removed from the mailbox. Since only write operations occur, the object will be located on one processor. This makes it inefficient to communicate between two

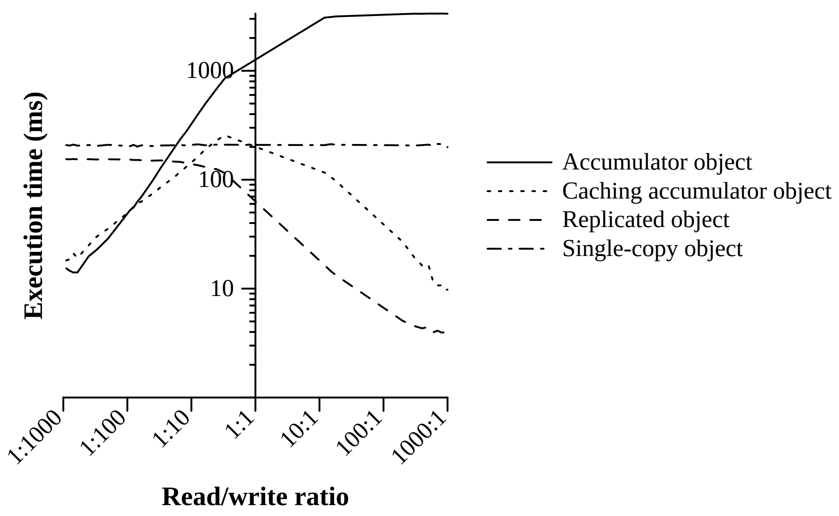


Figure 9.5: Performance comparison between a single-copy integer object, a replicated integer object, and an accumulator object for different read/write ratios running on 16 processors.

processors that both do not contain the object. To solve this problem, an object (a *mailbox* object) per destination processor needs to be allocated, and all processes need to have access to all those mailboxes. This setup, however, makes it inefficient to implement a broadcast or multicast primitive, because for each destination an operation is required to insert the message in the destination's mailbox.

Figure 9.6 shows the mailbox implementation using a single nested object. The mailbox object now consists of a collection of *port* objects, one per destination processor. Each port object is located on the destination processor that it belongs to (so the mailbox object is replicated on all processors.) Sending a point-to-point message now only requires communication with the destination processor, because the `send` operation only accesses a single remote subobject. The `receive` operation only accesses the local subobject, and therefore does not require any communication. To wake up a blocked `receive` operation no extra communication (apart from the communication to insert the message) is required, because the invocation weaver that executed the receive operation only runs locally.

We illustrate the usage of the mailbox nested object with the All-pairs Shortest Path (ASP) application. The ASP application finds the length of the shortest path from any node *i* to any other node *j* in a given weighted graph. The parallel algorithm is based on the standard sequential algorithm due to Floyd [3].

The sequential solution to the ASP problem uses an iterative algorithm. Dur-

```
object  implementation Port;
        q: list of messages;

        operation send(m: msg);
        begin
                add m to end of q
        end;

        operation receive(): msg;
        begin
                guard q not empty  do
                    return first message from q
                od;
        end;
end;

object implementation Mailbox;
        A: array[1..NCPUS()] of Port;

        operation send(m: msg; dest: integer);
        begin
                A[dest]$send(m);
        end;

        operation broadcast(m: msg);
        begin
                for i in 1 .. NCPUS() do
                        A[i]$send(m);
                od;
        end;

        operation receive() : msg;
        begin
                return A[MYCPU()]$receive();
        end;
end;
```

Figure 9.6: Implementation of the partitioned mailbox object.

ing iteration k, it finds the shortest path from every node i to every node j that only visits intermediate nodes in the set $1..k$. The algorithm checks if the current best path from i to k concatenated with the current best path from k to j is shorter than the best path from i to j found during the first $k-1$ iterations. Initially, only direct connections are registered as path; all other paths are initialized to infinity. After N iterations, where N is the number of nodes in the graph, all nodes may have been used as intermediate nodes. Therefore, the resulting path is the shortest path from node i to node j.

In the parallel Orca version [8], the matrix that contains the lengths of all paths is distributed over all processors. Each worker process receives a consecutive set of rows. All worker processes execute the iterative algorithm, updating their part of the matrix. For iteration k, however, all processes need to have access to row k before they can update their part of the matrix, because the current best length from node k to node j is required to compute the shortest length that can contain node k. Therefore, the processes execute as follows. In iteration k, the process that owns row k broadcasts this row to the other processes. The other processes wait for this broadcast message before they perform their local computations.

The Orca implementation uses a single object, the `RowCollection` object, to broadcast the rows. This object provides two operations: *AddRow* and *AwaitRow*. In iteration k, the process that owns row k invokes *AddRow*, passing the iteration number and the value of the row as parameters. The other processes block until the row is available by calling *AwaitRow* with the iteration number as parameter.

The `RowCollection` object suffers from the same problem as the message passing object in Orca: the *AwaitRow* operation would have to update the object to detect whether a row has been delivered to all destinations. In the actual implementation, no such state is maintained. Therefore, the `RowCollection` object has to keep *all* rows, which causes it to grow as large as the complete matrix. In the nested object implementation, the `RowCollection` object is replaced with the mailbox object. Now, each destination processor gets its own copy of the row in its local mailbox, and therefore the row can be removed when it is delivered.

Figure 9.7 shows the performance results for the Orca implementation and the nested object implementation. The sequential version of the nested object implementation is almost 50 percent faster than the sequential Orca version. This performance difference is mainly due to the difference between the hand-written C code and the code that is generated by the Orca compiler; especially replacing array indexing with pointer arithmetic to access the row elements makes a large difference.

More interesting, however, are the results of the scalability analysis (see figure on the right). Both implementations need to store their local partition, which takes $O(N^2/p)$ of memory, where N is the number of nodes and p the number of pro-

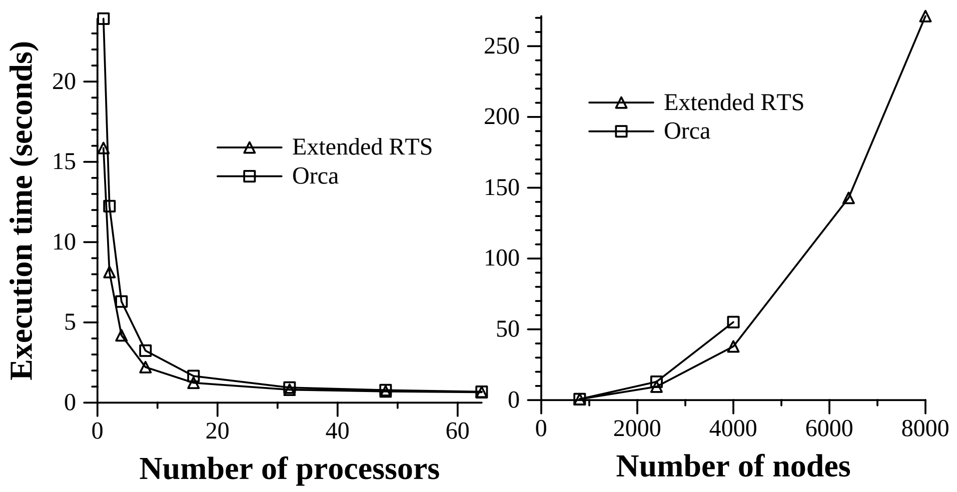


Figure 9.7: Performance comparison of the Orca implementation and the nested object implementation of the All-pairs Shortest Path application. The figure on the left shows the performance for 800 nodes; the figure on the right shows the performance on 64 processors.

cessors. The Orca version, however, also needs to store all the rows that are sent, which consumes another $O(N^2)$ of memory. Therefore, the Orca version can only handle problem sizes of up to 4000 nodes; for larger problems memory becomes exhausted (storing all rows for 4000 nodes already requires 64 MB). The nested object implementation, on the other hand, performs well even for 8000 nodes. In the nested object implementation, each processor only has to store the local partition of the matrix and enough memory for rows that arrive early. Note that the current (block-wise) partitioning can still cause memory exhaustion with the nested object implementation if a processor cannot cope with the rate it receives rows. This can be solved by switching to a cyclic partitioning, because in this case at most one row per processor needs to be queued in the local mailbox.

The 15-puzzle

In Section 8.2.2, we have studied the performance of the 15-puzzle implementation using only Orca objects and using Orca objects in combination with an atomic function. Here, we will look at the performance of two nested object implementations.

The first implementation uses a nested object to implement a distributed job queue and the termination detection function given in Figure 8.6 on page 153. This implementation is similar to the Orca version (see Section 8.2.2), except that the array of `JobQueue` objects is now replaced by a single nested object, which contains the real queues as subobjects. The root object provides operations to enqueue and dequeue jobs in the local queue or a remote queue. The same work stealing algorithm is used, so if the local queue is empty, a selection of remote queues is queried explicitly. For termination detection, the *idle* object from the Orca implementation is used.

The performance of this implementation is similar to the original Orca implementation, because it follows the same communication pattern and load balancing strategy. If we try to hide the complexity of the dequeue operation within the nested object, a number of problems occur. First, the dequeue operation is atomic, as any other operation. This implies that all queue objects will be locked if the local dequeue fails, causing the dequeue operations to serialize the execution of the worker processes. Second, the dequeue operation follows the same worst-case scenario as the atomic function *global_dequeue* presented in Section 8.2.1. The global dequeue operation runs on all processors, so after each queue has been checked, all processors need to know whether the queue was empty or not.

In our second implementation of the 15-puzzle using nested objects, we apply a different load balancing scheme and termination detection scheme that allows us to integrate these functionalities within the distributed job queue object. The root object not only contains a subobject for each jobqueue, it also contains a boolean array to indicate which processors are blocked on their empty local queue. Each dequeue operation that fails updates his entry in this `blocked` array (see Figure 9.8). If a dequeue operation succeeds, it also checks if there are more jobs available in the local queue and if other processors are blocked. In this case, jobs are migrated to the other job queues. The enqueue operation behaves in a similar fashion; if the local queue is not empty and other processors are blocked, jobs are migrated.

Figure 9.8 shows the implementation of the dequeue operation. There are several design issues that cause this implementation to perform efficiently. First, by allowing multiple synchronization points, the dequeue operation can change the entry of the `blocked` array that belongs to the invoking processor without leaving the operation. The guard block is within the while loop, so the operation only finishes if a *return* statement is reached. Second, our operation capability mechanism, as described in Section 6.2.3, allows us to perform the operation without communication if possible. Global synchronization is enforced only within the bodies of the `if` statements at line 13 and 16-23. In the default case (i.e., the local queue has a job and no other processors are blocked), the operation does not go into these blocks, and therefore only read access is required on the root object.

Figure 9.9 shows the performance results of the two nested object implementations and the Orca implementation. The nested object implementation uses the `idle` object for synchronization has the same performance as the Orca version, therefore only the nested object variant is shown. As already discussed in Sec-

```
VAR blocked: array[] of boolean;
    jobqueue: array[] of JobQueue;

operation dequeue(job: out JobType) : boolean;
    len, nr: integer;
begin
    while true do
        guard nr_blocked(blocked) = NCPUS() do
            return false;
        guard not blocked[MYCPU()] do
            len := jobqueue[MYCPU()]$length();
            if len = 0 then
                blocked[MYCPU()] := true;
            else
                if len > 1 and nr_blocked(blocked) > 0 then
                    nr := MIN(len - 1, nr_blocked(blocked));
                    for i in 1 .. nr do
                        dest := next_blocked(blocked);

                        jobqueue[MYCPU()]$dequeue(job);
                        jobqueue[dest]$enqueue(job);
                        blocked[dest] := false;
                    od;
                fi;

                jobqueue[MYCPU()]$dequeue(job);
                return true;
            fi;
        od;
    od;
end;
```

Figure 9.8: Distributed job queue with job migration.

tion 8.2.2, the performance of the Orca implementation of the 15-puzzle is very good and therefore hard to beat by implementations that use different synchronization schemes. The nested object implementation that is based on the blocking dequeue operation performs less than the Orca version, but at least it can longer keep up with the Orca implementation than the most optimized atomic function version (the atomic function with the optimized synchronizer object that is described in Section 8.2.2.)

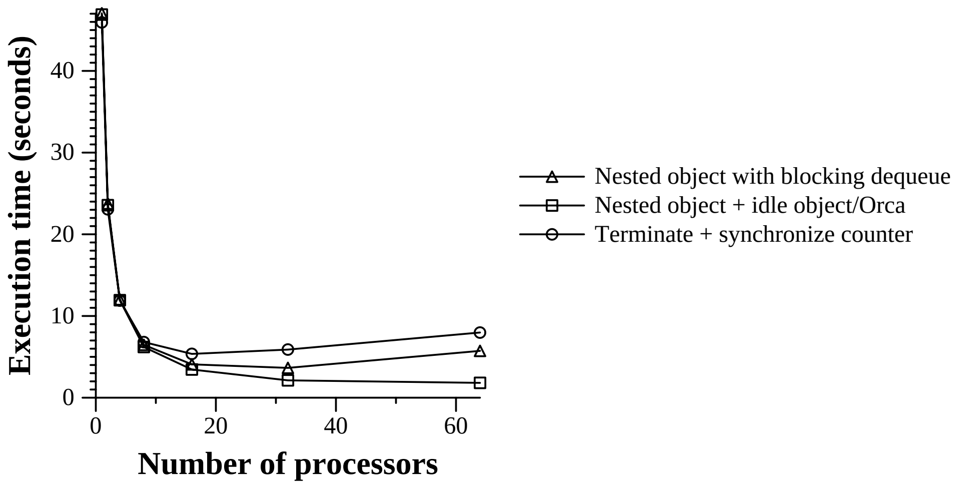


Figure 9.9: Performance comparison of the different implementations of the 15-puzzle.

The main benefit of the nested object implementation with a blocking dequeue operation, however, is that it hides all the complexity within the nested object. It is no longer necessary to use a complex *idle* object implementation, and also termination detection is handled by the nested object. This makes this nested object ideal for a library of useful objects; other applications that follow the same replicated workers structure as the 15-puzzle implementation only have to use the job queue object from the library to achieve reasonable performance.

Successive overrelaxation

In Section 9.1.5, we described the implementation of the successive overrelaxation using Orca objects and nested objects. Figure 9.10 shows the performance results for those implementations. As discussed in Section 9.1.5, the nested object version is not aimed to achieve the best performance, but only to show that partitioned objects can be mapped on top of weavers. As the partitioned object version, the nested object version manages to hide all the communication within the object, whereas the Orca version has to emulate message passing. The main reason for achieving a lower performance on larger numbers of processors is that the exchange of boundary buffers now occurs at the same time, whereas the Orca implementation can send boundary rows whenever a processor has finished its computation.

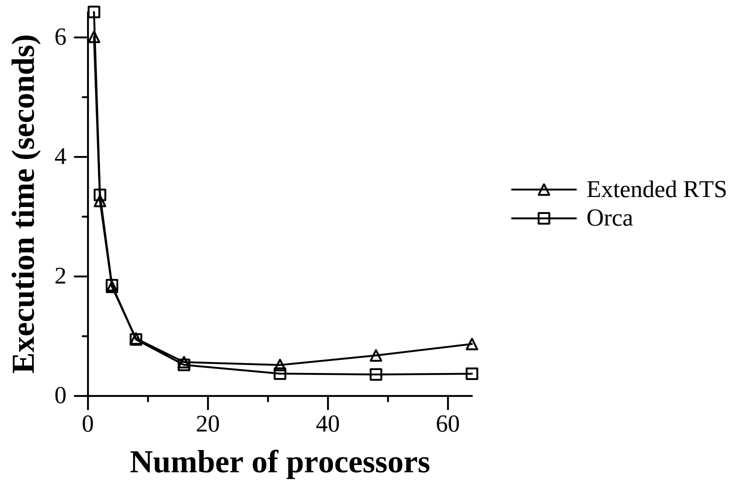


Figure 9.10: Performance comparison of the Orca and nested objects implementations of the Successive Overrelaxation application.

Summary

The nested objects runtime system adds a new object type to the extended runtime system, the nested object. To implement this object type, an application and a generic runtime system interface are provided. Operations on subobjects are invoked using the generic runtime system, so nested objects can be implemented using arbitrary object types as subobjects.

The restriction imposed by the model to limit the replication of subobjects to those processors that already contain a copy of the parent object makes it possible to implement support for nested objects efficiently. As long as no updates are performed and at most a single subobject is invoked, only a single processor needs to execute the invocation weaver. If an operation requires other or more processors to achieve the required capability, a new weaver is created on this set of processors. When the new invocation weaver reaches the failing operation, the processor set now contains all required processors to perform the operation.

If the invocation weaver for a nested object operation requires more than one processor to perform the operation, it behaves similar to an atomic function.

Therefore, we can use the same mechanisms as we used in the atomic functions runtime system to resolve data dependencies and to handle condition synchronization.

The performance analysis illustrates that nested objects can be applied very well to implement objects with a low read/write ratio. As the accumulator object illustrates, additional caching techniques can be implemented within the nested objects that cause the object to perform reasonably well even for high read/write ratios. Therefore, nested objects are also suited for applications in which the access ratio differs over time.

We have achieved our goal of efficient and consistent operations on the local subobject without suffering from severe performance losses in the more complex operations. The partitioned mailbox object, as used in the ASP application, illustrates the benefits of fast access to the local partition while maintaining the atomicity property of operations. Using the mailbox object, both unicast and multicast primitives can be implemented efficiently without causing memory exhaustion.

Finally, the performance results on the other applications illustrate that the nested object implementations can hide a lot of the application complexity within the object without suffering from large performance losses. This allows the creation of libraries of useful object implementations that can be used in other applications while still achieving good performance.

Chapter 10

Conclusions

In this thesis, we have looked at high-level language extensions for programming parallel computer systems. In particular, we have focused our attention on the most common parallel architecture today: essentially complete computers connected with a high-performance, scalable interconnect. Each computer contains one or more processors, memory, and a communication interface. The communication interface determines how the hardware behaves: either as shared memory or as a message passing architecture.

This research is concentrated on one particular programming model, shared data objects, as used in the Orca language. In Orca, the programmer uses operations on shared data objects for communication between processes. Processors logically share objects, even when there is no physical shared memory present in the system. Each operation is atomic, so complex state changes within an operation are only visible after the operation is finished. In addition, processes can synchronize by using condition synchronization (i.e., guarded expressions) within operations.

Orca has been used for a wide range of applications. For small applications, the language is easy to use. Larger applications, however, are harder to implement due to restrictions in the shared object model. The original shared data object model only supports atomic operations on a single object. We have shown that this is a severe restriction of Orca. Often an Orca application programmer needs to be able to specify atomic operations on *multiple* objects. In addition, Orca lacks the possibility to distribute the state of an object over different processors. Note that both these issues also arise in other object-based and object-oriented parallel programming languages, such as the various parallel extensions to C++ [124] and Java [118]. Within the context of Orca, we have investigated how these problems can be solved in a useful and efficient manner.

10.1 New concepts

We have generalized the shared data object model by defining two extensions: atomic functions and nested objects. Both extensions are primarily intended to facilitate parallel programming, but sometimes also performance improvements can be obtained.

Nested objects allow the programmer to partition the state of an object over the processors. From the outside, this object behaves exactly as a normal Orca object, but internally the object can take care of data partitioning and locality. The nested object is especially suited for implementing distributed data structures such as work queues, but it can also be used to encapsulate (multidimensional) arrays. Since a nested object behaves as a normal Orca object, nested objects are well suited for high-performance libraries that implement commonly-used behavior.

Operations on a nested object are always atomic and allow the same condition synchronization mechanism as shared objects. Although the runtime system tries to allow as much concurrency as possible, there is still some overhead involved. In some applications, atomicity is only required during some specific phases of the execution, for example a termination detection phase. Atomic functions extend the Orca model with the ability to perform such atomic and synchronized operations on arbitrary sets of objects. Like nested objects, both atomicity and condition synchronization can be expressed using an atomic function.

These extensions share a number of implementation properties with each other and with the implementation of normal shared objects. First, the implementation must deal with replicated objects, which is necessary to achieve good performance for several applications. Second, mutual exclusion and condition synchronization have to be integrated. Finally, data dependencies between operations need to be resolved efficiently.

A simple execution model to perform an atomic function is to let the invoking processor perform all the operations. This requires a protocol to lock all objects in advance (e.g., a synchronization broadcast message). The processor then invokes operations as in the Orca runtime system. Whenever the atomic function blocks or finishes, the locks have to be released (another synchronization message).

In Chapter 3 we discussed several problems of this simple execution model. The simple model locks objects for a long time, thereby serializing the execution of the parallel application. Furthermore, it does not exploit data locality, since all operation attributes are always passed by the processor that executes the atomic function to the processors that perform the operation. Finally, it is not clear how to incorporate nested objects in this scheme, since operations on a nested object may change the state of this object and also invoke operations on the subobjects.

To implement these two extensions efficiently, we investigated the use of *collective computation* within the context of object-based parallel programming languages. Collective computation is the execution of the same function on a set of processors. Although the idea of collective computation has already been used in parallel programming, this is the first study in the context of object-based languages. Object-based parallel programming imposes extra requirements for incorporating collective computation, such as object replication, dynamic object placement, and condition synchronization.

To evaluate the collective computation model, we looked at other techniques that have been used to implement runtime support for parallel programming. Collective computation integrates the ideas behind computation migration (perform the action where the data is), data replication (allow data to be available on multiple processors for fast access), collective communication (processors coordinate their invocations to optimize the communication pattern), and data parallelism (all processors determine the data dependencies locally). Our conclusion is that collective computation:

- allows the programmer to trade off data locality versus load balancing;
- can handle data replication efficiently;
- can generate optimized communication patterns for efficient communication; and
- is flexible enough to be used in an object-based parallel programming system.

10.2 Architecture

In Section 3.6, we presented an overview of the runtime system architecture that we developed to evaluate the collective computation model for implementing object-based parallel programming languages, in particular Orca and the two extensions (see Figure 10.1). By splitting the architecture in several layers, we are able to focus on the exact functionality and performance that each layer provides. Implementing a complete runtime system also allows us to show that the interface of each layer is suitable to implement on and does not sacrifice performance.

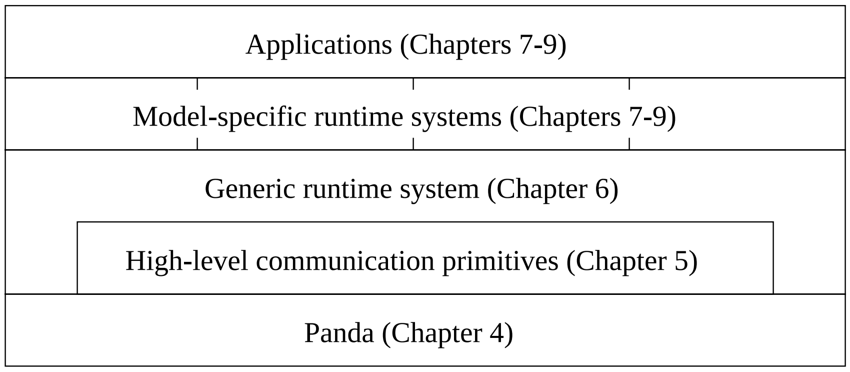

Applications (Chapters 7-9)
Model-specific runtime systems (Chapters 7-9)
Generic runtime system (Chapter 6)
High-level communication primitives (Chapter 5)
Panda (Chapter 4)

Figure 10.1: Overview of the prototype implementation.

The lowest layer, Panda, is a portable virtual machine designed to support implementations of parallel programming systems. Panda is an integrated system that provides threads, message passing, and group communication. To obtain high efficiency, Panda is designed as a flexible system, in which the communication modules can be adapted statically (i.e., when the system is compiled) so that they benefit from the properties provided by the underlying operating system and hardware. Also, the programming interface that Panda provides has evolved over the years to achieve both good performance and ease of use.

Panda provides two basic communication mechanisms: point-to-point message passing and group communication. These two communication mechanisms are sufficient to implement the original Orca runtime system. In combination with collective computation, however, we can exploit the fact that a set of processors will perform a certain communication pattern by optimizing how messages are handled and when acknowledgment messages are sent. We have developed a layer that provides the functionality to build efficient and flexible collective communication operations. The high-level communication primitives are based on the concept of a *communication schedule*, a description of the communication pattern that needs to take place.

Describing the communication pattern using operations on an abstract data type not only allows the low-level optimizations described earlier, it also provides the opportunity to generate optimized schedules for communication patterns that occur often. We have used the LogGP benchmarks to obtain architecture-specific performance parameters, and use these parameters to generate optimized schedules for gather and reduce operations. We have shown that using the LogGP parameters is especially useful for small messages, which are common in object-based parallel applications. The performance of this communication layer is evaluated by implementing a set of MPI primitives.

The next layer presents the generic runtime system. This layer implements the basic functionality that is needed for implementing shared objects, atomic functions, and nested objects. We identified the following requirements for the generic runtime system:

- Provide support for the parallel language constructs, such as objects and processes.
- Provide support for operations on a local instance of an object.
- Resolve data dependencies between operations.
- Deal with dynamic object configuration.
- Manage the cooperation between model-specific runtime systems.

The generic runtime system also provides the *weaver* abstraction, which implements the collective computation model. A weaver is an invocation of a single function on a set of processors in a total order. In the implementation of the generic runtime system, weavers are used to perform operations on objects, create and destroy objects, and to create and terminate processes. The total ordering guarantees that all processors can keep track of all objects and processes that are in use in a consistent manner.

Each runtime system creates a weaver to invoke operations on objects. By using weavers and the ability to perform operations on the local copy of an object, specific runtime systems can implement operations on replicated objects. Since weavers are invoked in a total order, sequential consistency is preserved. Also, since all runtime modules always use a weaver to invoke operations on objects, a runtime module can invoke a local operation on an object even if the invocation weaver is created by a different runtime module. This separation between object management and invocation allows the seamless integration of different runtime modules.

Performing operations on multiple objects imposes two synchronization requirements: mutual exclusion and condition synchronization. These requirements can be mapped to the signal and wakeup functions provided by the weaver abstraction. In addition, the generic runtime system provides support for weavers that perform blocking operations on objects by keeping track of which objects a weaver has accessed and which weavers are (potentially) blocked on an object.

Within an invocation of a weaver, processes involved in the weaver can communicate with the communication primitives of the weaver abstraction. These primitives allow the weaver threads to interlace their computations. Communication is not expressed in terms of message passing operations, but instead is described in data dependencies that need to be resolved. Based on these data dependencies, an optimized communication schedule is generated which performs the actual communication.

The top two layers present the specific runtime modules for shared objects, nested objects, atomic functions, and their applications. The specific runtime modules are independent of each other; they only interact through functions of the generic runtime system. Only the application has to be aware of the runtime modules that it is using.

Implementing runtime support for Orca objects on top of the generic runtime system is relatively easy. The weaver abstraction provides the basic functionality to implement operations on local, remote, and replicated objects. The hook functions for Orca objects contain a small amount of code and are easy to understand. In addition, making a distinction between the invoker of an operation and the control of an operation allows us to implement runtime support for Orca objects without any knowledge of the other runtime modules.

Like the Orca object module, the nested object module adds a new object type to the extended object runtime system. Nested objects can contain arbitrary objects as subobjects. The nested objects module is not aware of the object type (i.e., the runtime module that is associated with the object) of a subobject. All interaction between the nested objects module and the subobject is handled by the generic runtime system

The nested object model restricts the replication of subobjects to those processors that already contain a copy of the parent object. This restriction makes it possible to implement nested objects efficiently. As long as no updates are performed and at most a single subobject is invoked, only a single processor needs to execute the invocation weaver. If an operation requires other or more processors to achieve the required capability, a new weaver is created on this set of processors. When the new invocation weaver reaches the failing operation, the processor set now contains all required processors to perform the operation.

The atomic functions module implements an application interface that allows the programmer to write atomic operations on an arbitrary set of objects. The module does not add a new object type to the system, but only extends the way in which objects can be used. Since the module only uses functions provided by the generic runtime system, it is independent of the actual object type. Therefore, Orca objects and nested objects can be used in atomic functions.

The synchronization requirements of the atomic functions module can easily be implemented using the generic runtime system. The only extension to the generic runtime system that does not follow directly from the collective communication model is the ability to create a subweaver. This extra functionality to the generic runtime system is a simple and efficient extension.

We have extended the atomic function as defined in Chapter 2 with the notion of *multiple synchronization points*. This allows the programmer to perform state changes to the objects passed as parameter before blocking on the guard conditions. Multiple synchronization points can be used both as a performance optimization as well as for clarity.

The performance of our extended runtime system implementation is good. For some situations, the benefits of using collective computation gives even better performance than the original Orca system. Applications that benefit from the stricter semantics provided by atomic functions or nested objects perform well compared to the versions in which only Orca objects are used. For applications that can be implemented with only Orca objects, performance differences can be larger, but the Orca versions are typically harder to implement correctly. The nested object implementations can hide most of the application complexity within the object. This allows the creation of libraries of useful object implementations that can be used in other applications while still achieving good performance.

We have achieved our goal of efficient and consistent operations on the local subobject without suffering from severe performance losses in the more complex operations. This allows the developer of a nested object to deal with data locality while the application programmer is only aware of the performance and the resource consumption of the object's implementation. The partitioned mailbox object, as used in the ASP application, illustrates the benefits of fast access to the local partition while maintaining the atomicity property of operations. Using the mailbox object, both unicast and multicast primitives can be implemented efficiently without causing memory exhaustion.

Nested object are also very suitable to implement caching policies within the object. For example, the accumulator object allows efficient write access (as all processors can update their local instance of the subobject), but it can also perform reasonably well for high read/write ratios by caching the sum at the root object. Therefore, nested objects are also suited for applications in which the access ratio changes over time.

There is some performance overhead compared to the Orca runtime system. Especially local object invocations have not yet been optimized as much as in the original Orca system. The weaver queue and weaver scheduler add additional overhead for nonlocal invocations. However, the performance difference is not that large that it severely degrades application performance.

10.3 Evaluation

In this thesis, we have discussed the design and implementation of an object-based programming model that is substantially more general and flexible than that of Orca. The model still (deliberately) is more restrictive than that of object-oriented languages like Java, which allow operations on arbitrary graphs of objects. The advantages of our model are an easier, higher-level condition synchronization mechanism (based on guarded expressions) and the possibility to automatically replicate shared objects. Object replication in less restricted languages like Java is a hard problem, and often is solved by introducing object clustering mechanisms [59, 90] that also impose restrictions on the programming model. Thus, we believe our approach achieves a good balance between ease of programming and efficiency. The techniques that we have introduced to implement our model are also applicable for other object clustering mechanisms.

The experience and performance results from this implementation show that the collective computation model is a feasible model to implement high-level operations on multiple objects. The extensions that we have investigated show that the performance of such high-level operations does not need to be slower than the corresponding implementations that only use Orca objects, while the stronger semantics ease parallel programming.

The extensions to the shared data object model that we have presented can be implemented efficiently using collective computation. In addition, these extensions ease parallel application development, since they allow the programmer to reason about data locality and multi-object synchronization conditions without having to deal with concurrency issues. The nested object model also provides a flexible mechanism to implement advanced application-level cache policies.

We have implemented a complete runtime system based on the collective computation model. The central component of this runtime system is the weaver. A weaver can be regarded as an advanced active message that can be sent to multiple processors in a total order and has associated with it a function that will be executed by all involved processors. In addition, a weaver has synchronization primitives to block and wake up weavers while preserving the total ordering. The weaver mechanism turned out to be an easy and flexible building block in our runtime system. Essentially, it generalized the idea of a totally-ordered multicast message to a totally-ordered active message that can be blocked and rescheduled.

Our runtime system consists of multiple layers that allow the system to be extended with additional runtime modules. To enable this interaction, we developed a generic runtime component that provides the basic functionality for runtime module registration, object and process management, local object invocation, and data dependency resolution. In combination with weavers, this allows the runtime modules for shared data objects, nested objects, and atomic functions to be implemented efficiently with a minimum amount of code.

We have focused this thesis on the collective computation concept. In some situations, other execution models might be more efficient to implement the extensions that we have proposed. For example, if many flow control statements are used within a nested object operation or an atomic function, it would be more efficient to switch to computation migration instead of using collective computation. We believe that the framework allows such optimizations, but further research will be needed to investigate this interaction in more detail. Also, the compiler analysis to resolve the data dependencies requires further research. I hope that this thesis serves as a base for further research in this area.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Krantz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita, Italy, June 22–24 1995.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[4] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, Aug. 1986.

[5] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the 1995 Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 16–18 1995.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[7] T. E. Anderson, D. Culler, D. A. Patterson, and the NOW team. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.

[8] H. E. Bal. *Programming Distributed Systems*. Prentice Hall International, Hemel Hempstead, UK, 1991.

[9] H. E. Bal. *Report on the Programming Language Orca*. Vrije Universiteit, Amsterdam, May 1994.

[10] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Orca: a portable user-level shared object system. Technical Report IR-408, Vrije Universiteit, Amsterdam, June 1996.

[11] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Portability in the Orca shared object system. Technical report, Vrije Universiteit, Amsterdam, 1997.

[12] 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):1–40, Feb. 1998.

[13] 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*, 40(1):49–64, Jan. 1997.

[14] H. E. Bal, R. Hofman, and K. Verstoep. A comparison of three high speed networks for parallel cluster computing. In *Proceedings of the Workshop on Communication and Architectural Support for Network-based Parallel Computing*, pages 184–197, San Antonio, TX, feb 1997.

[15] H. E. Bal and M. F. Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 162–177, Washington D.C., Sept. 1993.

[16] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed system. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[17] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir. CCL: A portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, Feb. 1995.

[18] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, 1968.

[19] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of the Scalable High Performance Computing Conference*, pages 357–364, Knoxville, TN, May 23–25 1994.

[20] S. Ben Hassen. Implementation of nonblocking collective communication on a large-scale switched network. In *Proceedings of the 1st Annual Conference of the Advanced School for Computing and Imaging*, pages 1–10, Heijen, The Netherlands, May 16–18 1995.

[21] S. Ben Hassen, I. Athanasiu, and H. E. Bal. A flexible operation execution model for shared distributed objects. In *Proceedings of the 11th International Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 30–50, San Jose, CA, Oct. 8–10 1996.

[22] S. Ben Hassen and H. Bal. Integrating task and data parallelism using shared objects. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 317–324, Philadelphia, PA, May 25-28 1996.

[23] S. Ben Hassen, H. E. Bal, and C. Jacobs. A task and data parallel programming language based on shared objects. *ACM Trans. Prog. Lang. Syst.*, 20(6):1131–1170, Nov. 1998.

[24] M. Bernaschi and G. Iannello. Collective communication operations: Experimental results vs. theory. *Concurrency – Practice and Experience*, 10(5):359–386, Apr. 1998.

[25] B. Bershad, M. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th Annual IEEE Computer Society International Conference*, pages 528–537, Feb. 1993.

[26] R. Bhoedjang and K. Langendoen. Friendly and efficient message handling. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, pages 121–130, Wailea, HI, Jan. 3–6 1996.

[27] R. Bhoedjang, T. Rühl, and H. E. Bal. Design issues for network interface protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998. Special issue on network interfaces.

[28] R. Bhoedjang, T. Rühl, and H. E. Bal. Efficient multicast on Myrinet using link-level flow control. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 381–390, Minneapolis, MN, Aug. 1998. Best paper award.

[29] R. Bhoedjang, T. Rühl, and H. E. Bal. LFC: A communication substrate for Myrinet. In *Proceedings of the 4th Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, 1998.

[30] R. A. F. Bhoedjang, K. V. an Tim Rühl, and H. E. Bal. Reducing data and control transfer overhead through network-interface support. In *Proceedings of the 1st Myrinet User Group Conference*, Lyon, France, 2000.

[31] R. A. F. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. E. Bal, and M. F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, San Diego, CA, Sept. 22-23 1993.

[32] R. A. F. Bhoedjang, K. Verstoep, T. Rühl, and H. E. Bal. Evaluating design alternatives for reliable communication on high-speed networks. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[33] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[34] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[35] R. S. d. Boer. Parallel thinning and skeletonization using Orca. Master's thesis, Vrije Universiteit Amsterdam, 1995.

[36] P. Boncz. Parallelizing the crossword generation game in Orca. Technical report, Vrije Universiteit, Amsterdam, May 1994.

[37] P. A. Boncz, T. Rühl, and F. Kwakkel. The drill down benchmark. In *Proceedings of the 24th International Conference on Very Large Databases*, New York City, NY, Aug. 1998.

[38] D. S. Bouman. Parallelizing a skyline matrix solver in Orca. Master's thesis, Vrije Universiteit, Amsterdam, Aug. 1995.

[39] F. Breg. Porting Panda to the SP-2. Master's thesis, Vrije Universiteit, Amsterdam, Jan. 1997.

[40] F. Breg, T. Rühl, and H. Bal. Implementing the Panda portability layer on the SP2 using MPI. In *Proceedings of the 3rd Annual Conference of the Advanced School for Computing and Imaging*, pages 106–110, Heijen, The Netherlands, June 2–4 1997.

[41] J. Bruck, D. Dolev, C.-T. Ho, M.-C. Roşu, and R. Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. *Journal of Parallel and Distributed Computing*, 40(1):19–34, Jan. 1997.

[42] J. Carreira, J. G. Silva, K. Langendoen, and H. Bal. Implementing tuple space with threads. In *Proceedings of the 3rd Annual Conference of the Advanced School for Computing and Imaging*, pages 137–143, Heijen, The Netherlands, June 2–4 1997.

[43] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.

[44] M. Castro, P. Guerdes, M. Sequeira, and M. Costa. Efficient and flexible object sharing. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages I–128–I–137, Bloomingdale, IL, Aug. 1996.

[45] J. S. Chase, F. G. Amador, E. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, Dec. 3–6 1989.

[46] F. T. Chong, B.-H. Lim, R. Bianchini, J. Kubiatowicz, and A. Agarwal. Application performance on the MIT Alewife machine. *IEEE Computer*, 29(12):57–64, Dec. 1996.

[47] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, WA, Dec. 1-4 1985.

[48] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, 1992.

[49] D. Cronk, M. Haines, and P. Mehrotra. Thread migration in the presence of pointers. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, Wailea, HI, Jan. 7–10 1997.

[50] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 19–22 1993.

[51] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[52] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), Aug. 1975.

[53] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software – Practice and Experience*, 21(8):757–786, Aug. 1989.

[54] R. Draves, B. Bershad, R. Rashid, and R. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, CA, Oct. 13-16 1991.

[55] A. Fekete, M. F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 439–449, Vancouver, Canada, May 30–June 2 1995.

[56] J. Gray and A. Reuter. *Transaction Processing: Techniques and Concepts*. Morgan Kaufmann, San Mateo, CA, 1992.

[57] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[58] D. Grunwald, B. Calder, S. Vajracharya, and H. Srinivasan. Heaps o' stacks: Combined heap-based activation allocation for parallel programs. Technical report, University at Colorado, Boulder, 1994.

[59] D. Hagimont and D. Louvegnies. Javanaise: Distributed shared objects for internet cooperative applications. In *Proceedings of the Middleware Conference*, The Lake District, England, Nov. 1998.

[60] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Supercomputing '94*, pages 350–359, Washington D.C., Nov. 1994.

[61] M. Haines and K. Langendoen. Platform-independent runtime optimizations using OpenThreads. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 460–466, Geneva, Switzerland, Apr.1–5 1997.

[62] M. Haines, P. Mehrotra, and D. Cronk. Ropes: Support for collective operations among distributed threads. Technical Report ICASE95-36, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, April 1 1995.

[63] S. Hamilton and L. Garber. Deep Blue's hardware-software synergy. *IEEE Computer*, 30(10):29–35, oct 1997.

[64] H.-P. Heinzle, H. E. Bal, and K. Langendoen. Implementing object-based distributed shared memory on transputers. In *Proceedings of the Transputer Applications and Systems Conference*, pages 390–405, Como, Italy, Sept. 5-7 1994.

[65] W. Hsieh, P. Wang, and W. E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, San Diego, CA, May 19–22 1993.

[66] W. C.-Y. Hsieh. *Dynamic Computation Migration in Distributed Shared Memory Systems*. PhD thesis, Massachusetts Institute of Technology, Laboratory of Computer Science, Sept. 1995.

[67] Y. Huang and P. K. McKinley. Efficient collective operations with ATM network interface support. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 34–43, Bloomingdale, IL, Aug. 1996.

[68] K. Hwang, C. Wang, and C.-L. Wang. Evaluating MPI collective communication on the SP2, T3D, and Paragon multicomputers. In *Proceedings of the 3rd Conference on High Performance Computer Architecture*, pages 106–115, San Antonio, TX, Feb. 1997.

[69] G. Iannello, M. Lauria, and S. Mercolino. LogP performance characterization of Fast Messages atop Myrinet. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, Madrid, Spain, Jan. 21-23 1998.

[70] IEEE. *Threads Extensions for Portable Operating Systems*, P1003.4a/D6 edition, Feb. 1992.

[71] N. Islam and R. H. Campbell. Techniques for global optimization of message passing communication on unreliable networks. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 246–253, Vancouver, Canada, May 30–June 2 1995.

[72] K. L. Johnson. *High-Performance All-Software Distributed Shared memory*. PhD thesis, Massachusetts Institute of Technology, Laboratory of Computer Science, Dec. 1995.

[73] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–228, Copper Mountain Resort, CO, Dec. 3–6 1995.

[74] E. Jul, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[75] M. F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1992.

[76] M. F. Kaashoek, R. van Renesse, H. van Staveren, and A. S. Tanenbaum. FLIP: an internetwork protocol for supporting distributed systems. *ACM Transactions on Computer Systems*, 11(1):73–106, Feb. 1993.

[77] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the 1993 Symposium on Parallel Algorithms and Architectures*, pages 142–153, Velen, Germany, June 30–July 2 1993.

[78] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Rice University, Dec. 1994.

[79] D. Keppel. Tools and techniques for building fast portable threads packages. UWCSE 93-05-06, University of Washington, 1993.

[80] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[81] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[82] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[83] H. F. Langendoen. Parallelizing the polygon overlay problem using Orca. Master's thesis, Vrije Universiteit, Amsterdam, Aug. 1995.

[84] K. Langendoen, R. Bhoedjang, and H. Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2):28–38, 1997.

[85] K. Langendoen, R. A. F. Bhoedjang, and H. E. Bal. Automatic distribution of shared data objects. In *Proceedings of the 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 287–290, Troy, NY, May 22–24 1995.

[86] K. Langendoen, R. Hofman, and H. E. Bal. Challenging applications on fast networks. In *Proceedings of the 4th Conference on High Performance Computer Architecture*, Las Vegas, NV, Feb. 1998.

[87] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[88] K. Li. IVY: A shared memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, St. Charles, IL, Aug. 1988.

[89] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–41, Spring 1993.

[90] J. Maassen, T. Kielmann, and H. E. Bal. Efficient replicated method invocation in Java. In *Proceedings of the JavaGrande Conference*, San Fransisco, CA, Aug. 2000. To be published.

[91] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson. Collective communication in wormhole-routed massively parallel computers. *IEEE Computer*, 28(12):39–50, Dec. 1995.

[92] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 12 1995.

[93] P. Mitra, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast collective communication libraries, please. Technical Report CS-TR-95-22, University of Texas at Austin, June 6 1995.

[94] D. Mosberger. Memory consistency models. *Operating Systems Review*, pages 18–26, Jan. 1993.

[95] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, May 22–24 1996.

[96] National Coordination Office for Computing, Information, and Communications. *High Performance Computing and Communications: Foundation for America's Information Future*. NCO, 1996.

[97] D. Nicolaas. Parallelising the Turing ring using Orca. Master's thesis, Vrije Universiteit, Amsterdam, Aug. 31 1994.

[98] N. Nupairoj and L. M. Ni. Performance metrics and measurement techniques of collective communication services. In *Proceedings of the Workshop on Communication and Architectural Support for Network-based Parallel Computing*, pages 212–226, San Antonio, TX, feb 1997.

[99] M. Oey, K. G. Langendoen, and H. E. Bal. Comparing kernel-space and user-space communication protocols on Amoeba. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 238–245, Vancouver, Canada, May 30–June 2 1995.

[100] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, Dec. 3-8 1995.

[101] R. Ponnusamy, Y.-S. Hwang, R. Das, J. H. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions using data-parallel languages. *IEEE Parallel and Distributed Technology*, 3(1):12–24, Spring 1995.

[102] M. Raghavachari and A. Rogers. Ace: A programming language for application-specific protocols. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[103] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, Chicago, IL, Apr. 1994.

[104] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the power of parallelism in a Pile-of-PCs. In *Proceedings of the IEEE Aerospace Conference*, Feb. 1997.

[105] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[106] D. L. Rozendaal. Orca for Linux/PC. Master's thesis, Vrije Universiteit, Amsterdam, Sept. 1996.

[107] T. Rühl. Panda: A portable platform to support parallel programming languages. Master's thesis, Vrije Universiteit, Amsterdam, June 1993.

[108] T. Rühl, H. Bal, R. Bhoedjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, Aug. 9-11 1996.

[109] T. Rühl and H. E. Bal. The nested object model. In *Proceedings of the 6th ACM SIGOPS European Workshop*, pages 235–249, Dagstuhl Castle, Wadern, Germany, Sept. 12–14 1994.

[110] T. Rühl and H. E. Bal. Optimizing atomic functions using compile-time information. In *Proceedings of the 1995 Conference on Programming Models for Massively Parallel Computers*, pages 68–75, Berlin, Germany, Oct. 1995.

[111] T. Rühl and H. E. Bal. A portable collective communication library using communication schedules. In *Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 297–306, London, United Kingdom, Jan. 22-24 1997.

[112] T. Rühl and H. E. Bal. Synchronizing operations on multiple objects. In *Proceedings of the 2nd Workshop on Runtime Systems for Parallel Programming*, Orlando, FL, Mar. 1998.

[113] D. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, Oct. 1-5 1996.

[114] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 101–114, Monterey, CA, Nov. 14–17 1994.

[115] J. Schaeffer and A. Plaat. Kasparov versus Deep Blue: The rematch. *International Computer Chess Association Journal*, 20(2):95–101, June 1997.

[116] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing '94*, pages 97–106, Washington D.C., Nov. 1994.

[117] A. R. Sukul. Parallel implementation of an active chart parser in Orca. Technical report, Vrije Universiteit, Amsterdam, Feb. 1995.

[118] SUN. *Java Remote Method Invocation Specification*, 1.4 edition, Feb. 1997.

[119] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, second edition edition, 1989.

[120] K. Verstoep, K. Langendoen, and H. E. Bal. Efficient reliable multicast on Myrinet. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume Vol. III, pages 156–165, Bloomingdale, IL, Aug. 1996.

[121] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active message: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 19–21 1992.

[122] G. V. Wilson. Assessing the usability of parallel programming systems: The Cowichan problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, Ancona, Switzerland, Apr. 1994.

[123] G. V. Wilson and H. E. Bal. Using the Cowichan problems to assess the usability of Orca. *IEEE Parallel and Distributed Technology*, 4(3):36–44, fall 1996.

[124] G. V. Wilson and P. Lu, editors. *Parallel Programming Using C++*. MIT Press, 1996.

[125] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, Germany, 3rd corrected edition, 1985.

[126] M. J. Zekauskas, W. A. Sawdown, and B. N. Bershad. Software write detection for a distributed shared memory. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 87–100, Monterey, CA, Nov. 14–17 1994.

Index

Curriculum Vitae

Name:	Tim Rühl
Born:	14 March 1970, Amsterdam, The Netherlands
Nationality:	Dutch

Aug. 1982 - Jun. 1988:	Gymnasium, Amstellyceum, Amsterdam
Aug. 1988 - Jun. 1993:	Master's degree in Computer Science, Vrije Universiteit, Dept. of Mathematics and Computer Science
Jul. 1993 - Jun. 1997:	Ph.D. student in Computer Science, N.W.O., Pionier project, Vrije Universiteit, Dept. of Mathematics and Computer Science
Jul. 1997 - Dec. 1997:	Research assistant, Vrije Universiteit, Dept. of Mathematics and Computer Science
Jan. 1998 - present:	Software developer, Data Distilleries, Amsterdam

Author's current address:

Tim Rühl
Data Distilleries
Kruislaan 402
1098 SM Amsterdam
The Netherlands

email: t.ruhl@datadistilleries.com

Publications

1. 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):1–40, Feb. 1998.

2. 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*, 40(1):49–64, Jan. 1997.

3. R. Bhoedjang, T. Rühl, and H. E. Bal. Design issues for network interface protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998. Special issue on network interfaces.

4. R. Bhoedjang, T. Rühl, and H. E. Bal. Efficient multicast on Myrinet using link-level flow control. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 381–390, Minneapolis, MN, Aug. 1998. Best paper award.

5. R. A. F. Bhoedjang, K. V. an Tim Rühl, and H. E. Bal. Reducing data and control transfer overhead through network-interface support. In *Proceedings of the 1st Myrinet User Group Conference*, Lyon, France, 2000.

6. R. A. F. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. E. Bal, and M. F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, San Diego, CA, Sept. 22-23 1993.

7. R. A. F. Bhoedjang, K. Verstoep, T. Rühl, and H. E. Bal. Evaluating design alternatives for reliable communication on high-speed networks. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

8. P. A. Boncz, T. Rühl, and F. Kwakkel. The drill down benchmark. In *Proceedings of the 24th International Conference on Very Large Databases*, New York City, NY, Aug. 1998.

9. T. Rühl. Panda: A portable platform to support parallel programming languages. Master's thesis, Vrije Universiteit, Amsterdam, June 1993.

10. T. Rühl, H. Bal, R. Bhoedjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, Aug. 9-11 1996.

11. T. Rühl and H. E. Bal. The nested object model. In *Proceedings of the 6th ACM SIGOPS European Workshop*, pages 235–249, Dagstuhl Castle, Wadern, Germany, Sept. 12–14 1994.

12. T. Rühl and H. E. Bal. Optimizing atomic functions using compile-time information. In *Proceedings of the 1995 Conference on Programming Models for Massively Parallel Computers*, pages 68–75, Berlin, Germany, Oct. 1995.

13. T. Rühl and H. E. Bal. A portable collective communication library using communication schedules. In *Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 297–306, London, United Kingdom, Jan. 22-24 1997.

14. T. Rühl and H. E. Bal. Synchronizing operations on multiple objects. In *Proceedings of the 2nd Workshop on Runtime Systems for Parallel Programming*, Orlando, FL, Mar. 1998.