Synchronizing Operations on Multiple Objects*

Tim Rühl and Henri E. Bal
Dept. of Mathematics and Computer Science
Vrije Universiteit, Amsterdam, The Netherlands

Abstract. Parallel programming on distributed memory systems is one of the most challenging research areas in parallel computing today. Object-based parallel programming languages are an important class of languages for such systems. Shared objects allow the programmer to deal with data partitioning, communication, and synchronization in a high-level manner. Synchronizing operations on a single object is well understood. Dealing with synchronization on multiple objects distributed over the available processors, however, is still an open issue. In this paper, we will present an abstraction, called weavers, that is used to synchronize operations on multiple objects, and show how weavers are applied in a runtime support system for atomic functions on multiple objects.

1 Introduction

Programming parallel computers with distributed memory is one of the most challenging research areas in parallel computing today. The interest in these systems is caused by the fact that such systems are relatively easy to build and cost less than parallel architectures that support shared memory in hardware. In particular, clusters of workstations receive much attention [1].

Writing parallel applications for distributed memory systems, however, is more difficult than for shared memory. To make distributed memory systems more attractive to program, some parallel programming systems provide the illusion of shared memory on top of a distributed memory system [2, 3, 6, 7, 11, 12, 17]. These systems differ in the programming interface they offer. Systems like Shasta and TreadMarks provide the same programming interface as a true shared memory (i.e., a flat address space). Other systems, such as Orca, DiSOM, and CRL, let the programmer specify objects, which contain associated data. The programmer defines operations that can be executed on these objects.

This paper discusses how to synchronize operations on multiple objects. This work is based on an extension to Orca, a parallel programming language that provides a shared memory abstraction based on shared objects. Shared objects are instances of an abstract data type, which defines the data that each object contains and the operations that can be invoked on this data. Processes can communicate and synchronize with each other by performing operations on shared objects.

* This research is supported by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O)
For efficiency, the Orca runtime system replicates objects that have a high read/write ratio on all processors that can access this object. Operations that change an object are sent to all processors that have a copy. Each of these processors then performs the operation on its local copy. A totally-ordered broadcast message is used to send this operation. The total ordering guarantees that all processors receive all messages in the same order, and therefore perform the updates in the same order. We have efficient implementations of totally-ordered group communication on a large number of systems [4]. By using a total ordering on the updates sequential consistency is preserved [13]. Read operations on replicated objects are performed on the local copy. Operations on remote single-copy objects are performed using point-to-point communication [9].

The original Orca system supports atomic operations only on a single object. This design decision was made to be able to build an efficient implementation. We have proposed an extension to Orca, called atomic functions, that allows the programmer to specify a sequence of statements that is executed atomically on a set of objects [16]. In this paper, we will discuss the problems this extension imposes on synchronization. We propose a generic abstraction, called weavers, that we have used to implement runtime support for atomic functions. This abstraction handles all synchronization requirements for performing operations on multiple objects.

In Section 2, we describe the synchronization requirements for performing operations on single Orca objects. Atomic functions and their synchronization problems are presented in Section 3. The weaver abstraction is presented in Section 4. In Section 5, we describe how the runtime support systems for Orca objects and atomic functions are integrated on top of weavers. In Section 7 we discuss related work, and in Section 8 we draw our conclusions.

2 Synchronization on Orca Objects

In parallel programming, two types of synchronization are important. The first type, mutual exclusion, guarantees that only a single process can be in a critical section. Other processes that want to access this critical section will be blocked until the first process has left it. Mutual exclusion on shared memory systems is supported by locks. By associating a lock with data and requiring that this lock must be granted before the data may be accessed, mutual exclusion on data access can be enforced.

The other type of synchronization is condition synchronization. With condition synchronization, processes depend on each other, i.e., the result of the computation of one process is needed for the computation of the other process to continue. A typical example is a bounded buffer, in which one process puts elements into the buffer and another process gets elements from this buffer. The generating process has to wait if the buffer is full, while the acquiring process has to wait if the buffer is empty. Therefore, each get and put operation can trigger the other process to continue. Condition synchronization on shared memory systems is supported by constructs such as semaphores, event counters, and condition variables.