**Date**: January 2006

# Data Parallel CORBA Specification

OMG Available Specification
Version 1.0

formal/06-01-03

OBJECT MANAGEMENT GROUP

## USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

## LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

## PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

## GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

## DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO

WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

## RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

## TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

## COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

# OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

# Table of Contents

# Preface

## About the Object Management Group

### OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

## OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

*http://www.omg.org/technology/documents/spec_catalog.htm*

Specifications within the Catalog are organized by the following categories:

### OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

### OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

### Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

## Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

**Helvetica/Arial - 10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

**`Courier - 10 pt. Bold:`** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

**Note –** Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

## Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

# 1 Scope

This specification attempts to bring the requirements and advantages of parallel computing systems into the CORBA environment and standards. The two primary goals are:

- Enable scalability and performance similar to that found on scalable parallel systems.
- Preserve the object model, transparencies, and interoperability benefits of CORBA.

Interoperability is sought at a level where the client and server ORBs can collaborate efficiently in the parallel processing of requests, even when the ORBs are different.

Since the structure of the requirements is similar to Fault Tolerant CORBA, techniques and interfaces are shared when possible. This sharing is reinforced by using a **PortableGroup** module that is defined as a pure subset of the IDL from the Fault Tolerant CORBA specification. This module is used in other existing submissions (Multicast) and is intended to be suitable for submissions to other RFPs being considered (Load Balancing). Thus, such a module will likely be shared among four specifications.

The intention of this specification is to define a new compliance point similar to that of Real-time CORBA and Fault Tolerant CORBA, while not precluding any combination of these three optional ORB feature sets.

# 2 Conformance

This specification comprises a single optional compliance point for Data Parallel CORBA. Within this compliance point all interfaces are mandatory.

## 2.1 Compatibility

Existing code should be usable on parallel ORBs. No changes to the core are required.

## 2.2 Interoperability

Interoperability is achieved among various combinations:

- Clients on non-Parallel ORBs can use parallel object references and the POM.
- Parallel clients, part servers, and the POM, can all be implemented by different ORBs.
- An ORB can support singular clients, and the POM, without supporting part servers (if we choose to define these two compliance points).
- All parts of a parallel object must be on the same ORB.
- Load balancing (LEAST_BUSY) is only supported when the client and parts are on the same ORB.

# 3 Normative References

[MPI] Message Passing Interface Forum
http://www.mpi-forum.org/
MPI-2 and MPI-IO: http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2.html
MPI-IO home page: http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html

[DATAR] DARPA Data Reorganization Standard
http://www.data-re.org/
http://www.data-re.org/documents/dri-report-06082001.pdf

# 4 Terms and Definitions

There are no terms and definitions associated with this specification.

# 5 Symbols

There are no symbols associated with this specification.

# 6 Additional Information

## 6.1 Changes to Adopted OMG Specifications

This specification adds profile ids, component ids, and service context ids to the IOP module. It is anticipated that a future revision of the Fault Tolerant CORBA specification will share the **PortableGroup** module, which was carefully designed for such sharing.

## 6.2 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Los Alamos National Laboratory
- Mercury Computer Systems, Inc.
- Objective Interface Systems, Inc.
- MPI Software Technology, Inc.

# 7 Overview

This specification defines the architecture for data parallel programming in CORBA. The specification address data parallelism as opposed to other types of parallel processing that are already possible with distributed systems, namely pipeline parallelism and functional parallelism.

## 7.1 Data Parallel Programming

Parallel applications are characterized by a set of processes operating in parallel, usually on parts of a larger data set that is divided up among the participating processes. Data is typically redistributed between a set of sending processes and a set of receiving processes, which are sometimes the same single set. This pattern has been implemented in several CORBA-based environments by modeling the data distribution as aggregate data parameters to a CORBA invocation. These parameters are then divided up, collected, or redistributed between one or more clients and one or more servers.

The client/server model of CORBA has generally been considered unsuitable for parallel programming due to the lack of peer-to-peer semantics and difficulty in achieving distributed concurrency and/or data flow. A number of these issues have been mitigated by recent evolutions of CORBA such as AMI, multithreading, and reactive/recursive ORB implementations (which can process a request while waiting for a reply, all in a single thread). However, not all interactions can be described in today's CORBA model.

This specification enables the same basic patterns of computation and communication manifested by high performance, scalable, parallel applications and systems, but under a CORBA-based programming model.

### 7.1.1 Parallel Objects

This specification defines an additional approach for the implementation and use of CORBA objects that enables the object implementer to take advantage of parallel computing resources to achieve scalable, high performance. It enables clients to use such objects transparently and efficiently, whether or not the client ORB supports the new features. This is similar in spirit and structure to Fault Tolerant CORBA (orbos/00-01-19), which enables a replicated implementation alternative to achieve higher availability, which is also transparent to clients. This specification is limited to using the parallel computing resources in homogeneous, "data parallel" patterns, rather than some arbitrary work decomposition. Parallel objects embrace many of the concepts and techniques embodied in other course-grained parallel programming APIs and systems. Parallel objects are somewhat analogous to "process sets" or "process groups" in these other systems.

Parallel objects (whose implementation is a set of partial implementations executing in parallel) can be used by normal CORBA clients, and can also make requests of normal CORBA objects. Parallel objects can also make requests of other parallel objects and also of themselves. All these patterns are required for efficient and interoperable parallel systems. They are described in more detail below. Scalability of parallel objects in this specification is a run-time issue enabling reusability of scalable parallel implementations. This specification allows a parallel-capable ORB running a client to participate directly in making invocations on parallel objects. This pattern is key to parallel program performance.

### 7.1.2 Data partitioning

This specification defines interfaces and semantics for the partitioning and distribution of the data and requests involved in the use of parallel objects. Since the implementation of parallel objects is generally distributed in a homogeneous pattern across a set of "parallel" computing resources, this capability is necessary to support parallel implementations. These techniques embrace many of the concepts specifically defined in the Data Reorganization Effort (www.data-re.org) that has collected and consolidated best practices in this area.

## 7.2 Rationale

Parallel programming systems (APIs and middleware) use the same distributed computing platform technology as CORBA: multiple interconnected computers cooperating on an application. Such systems have been developed in a way generally disconnected from object-based and distributed object computing (e.g., [MPI]).

This specification brings the advantages of parallel computing to CORBA-based programming systems and brings the benefits of CORBA to problems requiring scalable, parallel computing implementations. These areas have been generally distinct, with only non-standard experiments and solutions developed with no interoperability. The approach defined here is to preserve the object model, transparencies and interoperability of CORBA while enabling parallel execution patterns necessary to applications with parallel computing requirements.

This approach was chosen (rather than any more direct mapping of parallel APIs into CORBA) to facilitate the use of parallel techniques by CORBA based systems and CORBA-trained application developers. Thus, it is conceptually more skewed toward existing CORBA users than users of other parallel programming systems. This is in the interest of avoiding fragmentation in the CORBA model.

## 7.3 Goals of this Specification

This specification attempts to address the objectives of the RFP, namely to bring the requirements and advantages of parallel computing systems into the CORBA environment and standards. The two primary goals are:

- Enable scalability and performance similar to that found on scalable parallel systems.
- Preserve the object model, transparencies, and interoperability benefits of CORBA.

Interoperability is sought at a level where the client and server ORBs can collaborate efficiently in the parallel processing of requests, even when the ORBs are different.

Since the structure of the requirements is similar to Fault Tolerant CORBA, techniques and interfaces are shared when possible. This sharing is reinforced by using a **PortableGroup** module that is defined as a pure subset of the IDL from the Fault Tolerant CORBA specification. This module is used in other existing submissions (Multicast) and is intended to be suitable for submissions to other RFPs being considered (Load Balancing). Thus, such a module will likely be shared among four specifications.

The intention of this specification is to define a new compliance point similar to that of Real-time CORBA and Fault Tolerant CORBA, while not precluding any combination of these three optional ORB feature sets.

# 8 Programming Model

This chapter presents the application programming model that is being defined. While the specification provides interfaces that exist only for ORB interoperability, this chapter only describes the interfaces visible to application programmers. A later section describes interoperability interfaces and semantics.

## 8.1 Concepts

### 8.1.1 Parallel ORBs

We use the term *parallel ORB* to mean an ORB that supports this specification, as opposed to *non-parallel ORBs*, which do not.

### 8.1.2 Three Types of Objects

For the purposes of this specification, we distinguish between three types of objects that relate to parallel objects.

#### 8.1.2.1 Singular objects

Singular objects are “normal” CORBA objects that are generally considered to have an implementation such that requests are processed in one execution context (thread or process), at a time, which carries out the entire job of processing requests from clients. While the implementation may internally divide the work among processing contexts somehow, this is invisible to the ORBs involved. Even with replicated services in Fault Tolerant CORBA, every request is nominally processed in one place.

#### 8.1.2.2 Parallel objects

Parallel objects are those whose requests may be carried out by one or more “parts,” probably, but not necessarily, running concurrently in different execution contexts. Thus, the work to process a request to a parallel object is carried out, in parallel, in multiple execution contexts. The implementation is such that different aspects of the work on a single request may be done piecemeal, in parallel.

#### 8.1.2.3 Part objects

The part objects are parts of a parallel object and work together to process requests for the parallel object. A parallel object is implemented by a set of part objects. The work performed by the parts is homogeneous in function; that is, interface but potentially heterogeneous in data or implementation. Any ad hoc or designed work breakdown other than homogeneous data parallelism is outside the scope of this specification. Part objects have no direct client/application-accessible interface. Parallel ORBs know about part objects, applications/clients do not. There is a simple hierarchical relationship between part objects and parallel objects. They are the parts and make up the whole, transparent to the client application. Throughout this document, “parts” and “part objects” are used interchangeably.

### 8.1.3 Two Types of Clients

For discussion purposes, we define two types of clients. As with normal CORBA clients, there is no specific representation or object that represents a client. A client is simply an entity issuing requests to objects.

#### 8.1.3.1 Parallel clients

Parallel clients are virtual CORBA clients that execute in a context that is associated with some parallel object. This execution context is normally inside a method of a servant of a *part object*. Parallel clients can only run on a parallel ORB. A parallel client is the collective behavior of part object servants when they act together to initiate a request on some object. It is the parts, acting collectively as the whole, which is acting as one client. An individual part object servant method invocation can of course act independently as a normal (singular) CORBA client.

#### 8.1.3.2 Singular clients

Singular CORBA clients have no implicit association with any parallel object, and thus are normal CORBA clients. Singular clients can receive references to, and invoke operations on, parallel objects, transparently. They do not have to run on a parallel ORB but there are significant performance benefits from parallel ORBs when they use parallel objects. This is similar to Fault Tolerant CORBA where there are availability benefits to clients running on Fault Tolerant ORBs, but it is not mandatory.

### 8.1.4 Data Partitioning

Data partitioning is how data entities, represented by parameters to an invoked operation on parallel objects, are in fact divided up to be distributed during invocations. An example is the typical two-dimensional tiling of images - an image processed by four parts that causes the image to be distributed with one quarter of the image to each part. There is a variety of commonly needed partitioning features defined here, mostly taken directly from the Data Reorganization Effort (www.data-re.org). They include:

- Block Distribution: Where similarly sized sub-pieces of data are distributed one to each part; for example, a set of four tiles of an image are distributed to four parts, where each part gets one tile.
- Cyclic distribution: Where similarly sized sub-pieces of data are distributed to parts in a round-robin fashion; for example, a set of 16 tiles of an image are distributed to four parts, where each part gets four tiles.
- Overlap: Where partitioned data is distributed with small amounts in common; for example, where each square tile of an image shares one row/column of pixels with its neighboring tile.
- Modular/Minimum constraints: Where an implementation requires a minimum or specific multiple of data; for example, minimum of 5 rows, or multiple of 3 rows.

Since client parameter data sizes are determined at run time; for example, the length of a sequence, partitioning details are only fixed when a request is created by the client. The partitioning details mentioned above are typically constraints of a part servant implementation that must be applied to client data sizes at runtime.

### 8.1.5 Request Distribution

This refers to how the request to a parallel object is in fact distributed to the part objects during invocation. Normally, this is simply “parallel,” meaning an invocation on the parallel object results in a set of simultaneous parallel invocations to the part objects (one for each). Other more dynamic distributions are possible; for example, load balancing, and sometimes part objects can be used more than once in a single invocation on a parallel object (cyclic). Request distribution builds on data partitioning to actually decide on the required communication pattern between client and parallel object.

### 8.1.6 Parallel Behavior

ParallelBehavior is the packaging of both *data partitioning* and *request distribution* that describes the implementation-defined (as opposed to interface-defined) behavior of a parallel object implementation (the parts). This information, expressed as data structure **ParallelBehavior**, can exist for each operation supported by the parallel object.

### 8.1.7 Collective Invocation

This is a type of invocation made by a *parallel client* that specifically (in the client source code) requests that the invocation is made collectively among the parts of the parallel object associated with the parallel client. Such invocations are evident in the specification by generated prefixed method names, starting with "collective_", similar to the AMI methods in the Messaging specification. This prefix would be after any sendc/sendp prefix needed by AMI. A collective invocation implies specific collaboration by parallel client ORB code executing in all the parts of a parallel object. Parallel clients can also make normal, non-collective invocations as defined in the standard CORBA language mappings.

## 8.2 Clients Using Parallel Objects

This specification defines no new interfaces or semantics that are visible to singular clients using parallel objects.

During normal operation on parallel objects, there is no need for an actual singular implementation with an exposed interface. The parallel object exists in its references and in the existence of its parts.

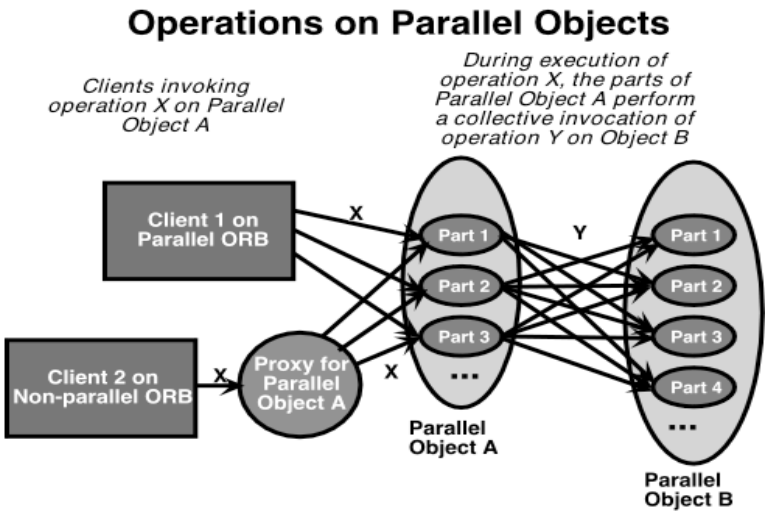


**Figure 8.1 - Operations on Parallel Objects**

## 8.3 Creating Parallel Objects

When applications create parallel objects, they are not necessarily acting as client or server, but as creator, using various factory capabilities. Being a creator has no mandatory relationship of locality with either clients or servants (parts). The creation and management of parallel objects closely follows the model established in the adopted and finalized Fault Tolerant CORBA specification (ptc/2000-04-04). This section can be thought of describing a small number of variations

from that specification's treatment of the same subject (the Replication Management section). The creator of a parallel object uses an interface called the **ParallelObjectManager** (POM), which inherits a number of interfaces from the **PortableGroup** module, including **GenericFactory**.

### 8.3.1 PortableGroup Module

This specification defines a **PortableGroup** module (identical to that proposed in the joint submission to the Unreliable Multicast submission). The ultimate intent of this module is to share it among at least these three technologies (and possibly future Reliable Multicast and Load Balancing ones). It is identical to a subset of the FT CORBA specification with a few changes to make it more generic to group management. The changes are as follows:

- Rename **TAG_FT_GROUP** to **TAG_GROUP**.
- Rename **FTDomainId** to **GroupDomainId**, **ft_domain_id** to **group_domain_id**.
- **RenameInitialNumberReplicasValue** to **InitialNumberMembersValue**.
- Rename **MinimumNumberReplicasValue** to **MinimumNumberMembersValue**.
- The **set_primary_member** operation is removed from the **ObjectGroupManager** interface.

The basic content of this shared module is to define these interfaces and their supporting types, structures, and exceptions:

- PropertyManager
- ObjectGroupManager
- GenericFactory

These interfaces and all their required data types are described in the Fault Tolerant CORBA specification, ptc/2000-04-04. The full **PortableGroup** IDL is in section 7.1. The following properties are also implicitly defined in the **PortableGroup** module and used by this specification:

- **MembershipStyle** - **org.omg.pg.MembershipStyle**, of type **MembershipStyleValue**.
- **Factories** - **org.omg.pg.Factories**, of type **FactoryInfos**
- **InitialNumberMembers** - **org.omg.pg.InitialNumberMembers**, of type **InitialNumberMembersValue**.
- **MinimumNumberMembers** - **org.omg.pg.MinimumNumberMembers**, of type **MinimumNumberMembersValue**.

The changes in properties from the Fault Tolerant CORBA specification are:

- The name scope in the included property names is renamed from "**ft**" to "**pg**."
- Rename property names that include "**replica**" to "**member**."

### 8.3.2 The Parallel Object Manager (POM)

Creating parallel objects requires that the creator make several decisions. Specifically the creator must specify, either explicitly or implicitly:

- Number of Parts - how many part objects should be created to jointly process requests for the parallel object. This is specified by the **org.omg.pg.InitialNumberMembers** property with an **InitialNumberMembersValue**.

- Creation style - whether the actual creation is done in a single operation on the POM, or whether the creation is done by the application creating parts, and communicated to the POM incrementally. This is specified by the **org.omg.pg.MembershipStyle** property with the value of type **MembershipStyleValue**. The choices are **MEMB_INF_CTRL**, for infrastructure-controlled membership style, or **MEMB_APP_CTRL**, for application-controlled membership.

- Location of parts - in which execution context should each part be created?

  With **MEMB_INF_CTRL**, this is specified by the locations in the **org.omg.pg.Factories** property with the value of type **FactoryInfos**. As stated in the FT CORBA specification: **FactoryInfos** is a sequence of **FactoryInfo**, where **FactoryInfo** contains the reference to the factory, the location at which the factory is to create a member of the object group and criteria (properties) that the factory is to use to create the member. With **MEMB_APP_CTRL**, the location is a parameter to the **add_member** or **create_member** POM operations.

- Parallel implementation type - which actual interface type should be used in the parallel object's reference?

  This is specified as the **type_id** parameter to the **create_object** operation of the **GenericFactory** interface as inherited by the **ParallelObectManager**. This type (which is not a "part interface type", but the interface of the parallel object as a whole) implies the "part" interface from which the implementation types supported by the part objects must be derived. The local factory for the part objects determines the actual type of the parts.

  The Parallel Object Manager is also responsible for checking that the ParallelBehaviors reported by the part objects are compatible and either creating a "least common denominator" parallel behavior acceptable to all parts, or reporting a BadComparison exception.

The **ParallelObjectManager** (POM) inherits the **GenericFactory**, **PropertyManager**, and **ObjectGroupManager** interfaces from the **PortableGroup** module. This is the same structure as in Fault Tolerant CORBA (the **ReplicationManager**).

```
module DP {
    interface ParallelObjectManager :
        PortableGroup::GenericFactory,
        PortableGroup::PropertyManager,
        PortableGroup::ObjectGroupManager {};
};
```

As in FT CORBA, the MembershipStyle property specifies whether the application will use the POM to create or add individual parts (via **create_member** or **add_member**) or whether the application uses the POM's **create_object** operation to indirectly create all the parts. In the application controlled membership style, the application can create the parts in a distributed and parallel fashion, and have each part server, where the individual parts are created, use the **add_member** operation on the POM to inform the POM of the existence of parts.

The POM has several capabilities not found in the Fault Tolerant specification. The POM must communicate the object group reference to all parts before they can be used. This is accomplished by a hidden operation that is part of the implied-IDL of the part interface (described in Section 8.4.3, "Implementing part operations," on page 15). The operation, **_DP_set_whole,** allows the POM to inform the part object's ORB of the whole object to support collective operations. This operation is used in the POM's implementation of **create_object** (after all parts are created, but before returning), as well as the implementation of **get_object_group_ref**, when, in application-controlled membership, the application is obtaining the final version of the references. The parallel ORB hosting the part objects processes this operation to support collective operations. It is not expected to be implemented by the part servant.

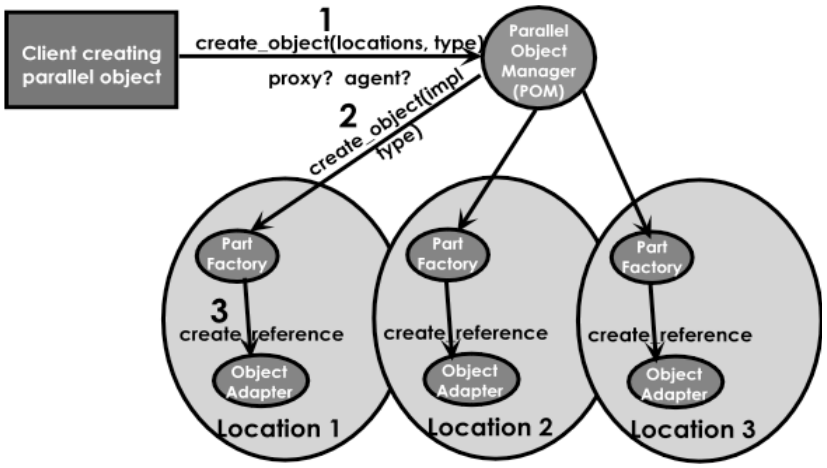


**Figure 8.2 - Creating Parallel Objects – top-down pattern**

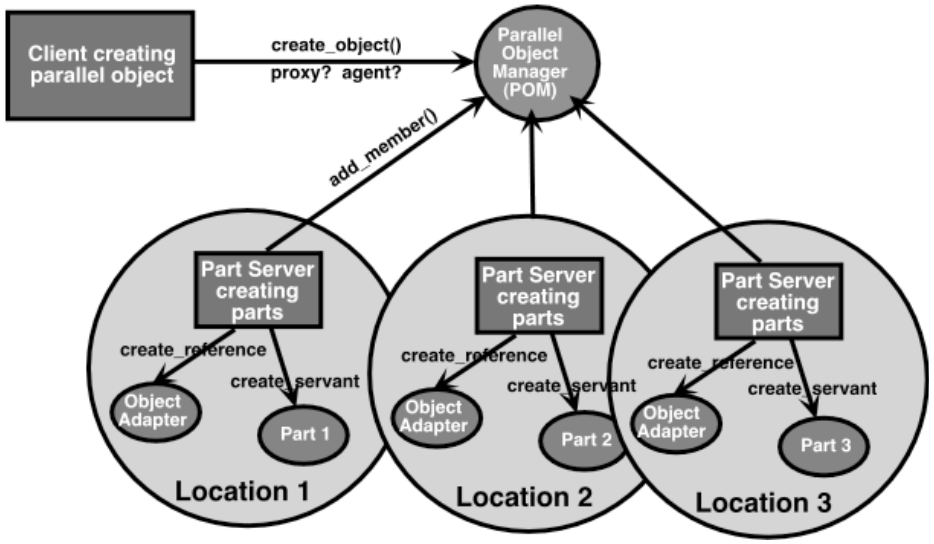


**Figure 8.3 - Creating Parallel Objects – bottom-up pattern**

## 8.4 Implementing Parallel Objects - “Part Servers”

Implementing parallel objects by implementing “parts,” involves special parallel application techniques to decompose the functionality in a way that can exploit execution contexts operating concurrently, usually on subsets of the data supplied to or returned from the operation. There are four aspects to this task that require support as addressed in this specification:

1. Creation of parts that the POM will collect together as a parallel object.

2. Specifying the **ParallelBehavior** that is expected of client ORBs when invoking operations on the parts.

3. Implementing versions of operations that are specialized to operate on part of the data conveyed in the operation

4. Possibly performing collective invocations among the parts during the processing of requests, whether on a different object or on the (self) same parallel object.

## 8.4.1 Creating Parts Objects

The creation of part objects mirrors the creation of replicas in the Fault Tolerant specification. The **GenericFactory** interface is used, on a location-specific factory, either by an application process that locally creates parts or by the POM. In either case, the actual reference for the part must be created with the **ParallelBehavior** information provided to the POA. As in FT CORBA, the local factory provides a **GenericFactory** interface for use by the POM. The local factory here is called the Parallel Part Factory, and implements the **GenericFactory** interface. The *application* supplies the Parallel Part Factory, which knows how to create the references and possibly servants for the part objects. The only local capability provided by the infrastructure is a Data-Parallel POA.

### 8.4.1.1 Data-Parallel POA

Data-Parallel CORBA defines a Data-Parallel CORBA Current interface to supply **ParallelBehavior** information to POA operations that create references (**create_reference** and **create_reference_with_id**).

A Data-Parallel POA will differ from a CORBA POA only in the ability to support **ParallelBehavior** settings on **DP::Current** that apply to future reference creation operations of that POA. Thus, there is no IDL for the Data-Parallel POA. The IDL for the **DP::Current** is:

```
module DP {
    local interface Current : PortableServer::Current {
        void set_parallel_behavior(in ParallelBehavior
                                   the_parallel_behavior);
        void clear_parallel_behavior();
    };
};
```

The initial state of a Data-Parallel POA is that no **ParallelBehavior** will be associated with any created references. After calling the **set_parallel_behavior** operation on **DP::Current** the supplied **ParallelBehavior** is used for all POA-created references in the current thread. The initial state can be achieved by calling the **clear_parallel_behavior** operation. The reference creation operations that this will apply to are: **create_reference**, **create_reference_with_id**, and, with RT CORBA, **create_reference_with_priority** and **create_reference_with_id_and_priority**.

See Section 9.2, “Interoperability Requirements: Potential System Decomposition,” on page 21 for the content of part references. The application can obtain an instance of **DP::Current** by narrowing a reference to **PortableServer::Current**.

#### 8.4.1.2 Part Factory

The Part Factory is analogous to the location-specific **GenericFactory** in the Fault Tolerant CORBA specification. It is the entity local to the execution context (Location) in which the part object will be implemented. This object is remotely accessible, by the POM, when the infrastructure-provided membership style is used. It is application supplied and inherently associates implementation types that are available along with associated **ParallelBehavior**. It uses the Data-Parallel POA to do this, as defined above.

There may be multiple Part Factories (thus multiple locations) in a process; that is, in an SMP system, where a thread is a location. Part Factories are registered as a "location" with the POM, by virtue of **Factories** property setting. There is no IDL for Part Factories beyond the **PortableGroup::GenericFactory**.

### 8.4.2 Specifying ParallelBehavior

This data structure contains descriptions of how the implementation can behave for each operation. It is:

- A description of what the part objects will do and what clients ORBs performing invocations must do.
- The information necessary for the client ORB to "do the right thing."
- A sequence of descriptions of operations, for those operations that do not have default parallel behavior.

The default parallel behavior is that all parameters are sent in their entirety to all parts and all results are assumed to be identical.

Operation descriptions are provided as a sequence in which, for each operation, there are two sections, request distribution and data partitioning. Note that all parts of a parallel object must have the same **ParallelBehavior,** even if the actual implementation types of parts differ. The information included defines what is different from the default. Thus, only operations that have non-default behavior are included, and only parameters (and result) that have non-default behavior are mentioned.

#### 8.4.2.1 Data Partitioning - What to do with the parameters?

Each in or inout parameter to a parallelized operation can either be delivered whole to the part objects or be divided into pieces (partitioned), if appropriate for the data type and the implementation. Data types that are appropriate for partitioning are those whose top-level representation is a sequence or array. Sequences of sequences are treated the same as multi-dimensioned arrays for partitioning purposes.

If an operation has more than one parameter (or return value) that can be partitioned, it is almost always necessary to make the partitioning consistent between them, since most data parallel processing requires it. Therefore, this specification only defines behavior in the useful case where all partitioned parameters are divided into (or combined from) the same number of pieces.

The partitioning possibilities used here are similar to those defined in the Data Reorganization Effort (www.data-re.org), namely *for each dimension* of the data parameter, we specify:

1. Whether data can be partitioned in this dimension
2. Constraints on the number of elements (minimum, modulus, maximum)
3. Required overlap between pieces

4. The position of this dimension in the piece of data at the part vs. the client; that is, the dimensions can be reordered between client and server.

These represent an expression of constraints, per dimension, without knowing the data sizes, which are only known by the client.

- A minimum constraint of zero means the dimension cannot be partitioned.
- A maximum constraint of zero means there is no maximum.
- A modulo constraint of zero is interpreted as one (1).

Thus a structure initialized to zero defines default behavior.

Overlap specifications can be specified to both left side (lower index) and right size (higher index). When an edge of a piece is also an edge of the whole (client-size) data value, the extra overlap data must be synthesized (padded) using one of four specified methods:

1. Use zero.
2. Replicate the edge value in the piece as the overlap area.
3. Truncate the piece and do not add the overlap at the edge of the whole.
4. Wrap the value from the other opposite edge of the whole.

IDL for data partitioning is:

```
module DP {
    enum OverlapPad {
        OVERLAP_PAD_ZEROS,
        OVERLAP_PAD_REPLICATED,
        OVERLAP_PAD_TRUNCATE,
        OVERLAP_PAD_WRAP
    };
    struct DimensionOverlap {
        unsigned long       amount
            OverlapPad      pad;
        };
        struct DimensionPartition {
            unsigned long       position;       // at part object
            unsigned long       minimum;        // 0 means unpartitionable
            unsigned long       maximum;        // 0 means no maximum
            unsigned long       modulo;         // 0 means 1
            DimensionOverlap    left_overlap;
            DimensionOverlap    right_overlap;
        };
        typedef sequence<DimensionPartition>DataPartition;
};
```

#### 8.4.2.2 Request distribution (and synchronization)

Different parallel implementations of operations require different patterns of request distributions and synchronization. To some extent this can be inferred from the Data Partitioning requirements. The most common is to send the request to all parts of the parallel object exactly once. Another essential pattern is when the appropriately partitioned data requires more requests than there are part objects. There are four basic types of request distribution, which describe, per invocation of an operation:

1. Send one request to some part (like normal CORBA or FT CORBA).
2. Send one request to each part.
3. Send zero or one request to each part.
4. Send zero, one or more requests to each part.

When we are not sending exactly one request to all parts, we further must specify how to distribute the requests among the parts:

1. Cycle through parts starting with the first and wrap back to the first if necessary (“round-robin”, or “cyclic”).
2. Cycle persistently (starting where previous request stopped).
3. Send to the least-busy part (with round-robin when equally busy).

When we are allowed to send multiple requests to a given part, we must specify how many requests can be outstanding, per part. We must also specify whether parts need to be informed when they will receive no more requests for a given client invocation. Finally, we must specify any required barrier synchronization among parts before and/or after the invocation. The client ORB does not perform the synchronization of the part objects; it is performed by the Data-Parallel POA as part of dispatching. The IDL for request distribution is:

```
module DP {
    enum DistributionType {
        DISTRIBUTION_ONE_REQUEST,
        DISTRIBUTION_ONE_FOR_EACH,
        DISTRIBUTION_ZERO_OR_ONE,
        DISTRIBUTION_VARIABLE_FOR_EACH
    };
    enum DistributionPattern {
        DISTRIBUTION_CYCLIC,
        DISTRIBUTION_PERSISTENT,
        DISTRIBUTION_LEAST_BUSY,
        DISTRIBUTION_RANDOM
    };
    struct RequestDistribution {
        DistributionType    type;
        DistributionPatternpattern;
        boolean             inform_at_end;
        boolean             sync_before;
        boolean             sync_after;
    };
};
```

The least busy distribution pattern requires cooperation between client and server ORBs that is not specified here, and thus such behavior is not interoperable. The precise definition of “least busy” is left to the implementers, but it meant to allow the part servant author to specify that the distribution pattern should be based on dynamic system loading to achieve good system utilization when the processing time of requests by part servants is variable. This could be refined or expanded based on future load balancing standardization.

#### 8.4.2.3 ParallelBehavior

The complete **ParallelBehavior** structure describes all parallel operations supported by the parts, by combining request distribution for each operation with the data partitioning specification for each partitionable parameter in that operation.

Partitioning, when used for output and return value parameters, must specify how the data from the parts should be combined. There are several ways to resolve returned or output values:

1. Combine the values according to the data partitioning specification.
2. Compare all values and provide an exception if different, otherwise return the common value. The exception is Bad-Comparison, which reports the name of the parameter that failed to compare. If more than one parameter fails the comparison, the first one is reported (the result is considered first if not void).
3. Take any returned/output value.

The IDL for **ParallelBehavior** is:

```
module DP {
    struct ParallelParameter {
        CORBA::Identifier  name;        // empty for return
        DataPartition      partition;
        boolean            compare;  // for inout/out/return
    };
    typedef sequence<ParallelParameter> ParameterList;
    struct ParallelOperation {
        CORBA::Identifier      name;
        ParameterList          parameters;
        RequestDistribution    distribution;
    };
    typedef sequence<ParallelOperation> ParallelBehavior;
    };
    exception BadComparison {
        CORBA::Identifiername;
    };
};
```

### 8.4.3 Implementing part operations

Although parallel parts are not directly known or used by clients, they exist and support operations that perform part of the work of the operations on a parallel object. The author of a parallel object implementation writes the implementation of part objects and specifies their **ParallelBehavior**. The actual interface supported by part object implementations is the “part version” of the IDL-defined interface of the parallel object. There are rules to generate the “part interface” mapping from the IDL. This is in fact an IDL-to-IDL mapping that creates implied-IDL. The implied-IDL interface name is **DP_<ifaceName>Part**, similar to the **AMI_<ifaceName>Handler** in the CORBA Messaging specification.

Implementers of parallel parts use skeletons that are generated by standard IDL compilation on this implied IDL. This is because the actual partitioning of data is determined by the client ORB, requiring extra partitioning information to be passed with the request. Since the IDL cannot (per the RFP) contain extra information about which parameters are partitioned and how requests are distributed, this information must be made available to the skeletons in a way not driven by IDL.

For partitioned parameters, the part servant must know:

1. What is the actual size of the piece being provided, excluding overlap?
2. What was/is the size of the whole parameter data at the client?
3. What is the position of the piece in the whole?
4. What is the position of the "owned" piece in the "local piece" that may include overlap?

The term "owned" refers to the data that is partitioned without considering overlap. Any element of the original data is only "owned" by one part, even though, due to overlap requirements, some elements may be sent to multiple parts. The information about the "whole" (size and position of piece) is required because many data partitioned computational algorithms are dependent on this, like human vision. The part servants must also know:

1. Is the method call the last (or only) one that will be received for a given client request?
2. What is the number of this part request among all those for a given client request?
3. What is the total number of part requests for this client request?

If the **DistributionType** for the method is **DISTRIBUTION_ZERO_OR_ONE** or **DISTRIBUTION_VARIABLE_FOR_EACH**, and the **inform_at_end** value is **TRUE**, an indication that no more (or none at all) part requests will be received by this part for a given client request. The "none at all" case is when data partitioning constraints preclude creating as many data pieces as there are parts.

The implied-IDL for all part servant methods changes the original IDL in two ways:

1. All array arguments become sequences of the same dimensionality and type. When this happens a new typedef is implied for this new type, whose name is **DP_<operation_name><parameter_name>**. For the result (return value) of an operation, the parameter name is "Result." If the interface name **DP_<iface-Name><parameter_name>** conflicts with an existing identifier, uniqueness is obtained by inserting additional "DP_" prefixes before the **ifaceName** until the generated identifier is unique.
2. An additional parameter is appended with the name **part_info**, of a generated data type **<operation_name>_PartInfo**. This parameter is **in**, **inout**, or **out** depending on the requirements of the other partitioned parameters. Outputs require that the implementation fill the corresponding structure members to inform the client ORB of the actual partitioning effected by the parts.

The generated data type **<operation_name>_PartInfo** always contains an initial member structure of type **DP::PartRequestInfo** under the member name of **request**, which contains information not specific to any parameter. Its IDL is:

```
module DP {
    struct PartRequestInfo {
        unsigned long part_ordinal;
        unsigned long num_parts;
        boolean     last;
```

```
        boolean     empty;
    };
};
```

The **part_ordinal** and **num_parts** are information basic to the entire parallel object and which part this is in the whole. It is redundantly present in this per-operation structure for convenience of the part servant method. The **last** structure member indicates, when the number of part requests is variable, that this part request is the last this part will receive for the associated client request. The **empty** member indicates that this is not a part request but simply an indicator that no more part requests will follow. The part method should always check this value first, if it has advertised the ability to receive variable numbers of part requests and also requested such notification. It is only valid if the **last** value is also TRUE.

The **<operation_name>_PartInfo** data type parameter contains the **PartRequestInfo** structure as well as one structure member per sequence parameter in the operation. This structure member (per sequence parameter) has the name that is the same as the parameter name, and is of the **PartParameterInfo** data type. If there is a result (not **void**) that is a sequence (or, originally an array), then a member whose name is **result** is added after the initial **part_info** member. The definition of this data type is:

```
module DP {
    typedef sequence<unsigned long> Sizes;
    typedef sequence<unsigned long> Positions;
    struct PartParameterInfo {
        Sizes       whole_size;
        Sizes       owned_size;
        Positions   position_in_whole;
        Positions   position_in_local;
    };
};
```

The members, per parameter or result, have the following meanings:

1. **whole_size** - the size of the client-size data (size in each dimension).
2. **owned_size** - the size of the local piece of data not including overlap.
3. **position_in_whole** - the position (per dimension) of the local owned data in the whole (sequence).
4. **position_in_local** - the position (per dimension) of the owned data in the local piece.

An example of this implied IDL follows. Assuming an original IDL of:

```
interface ParObj {
    typedef float Floats[400][400];
    Floats ParOperation(in long x, in sequence<float> arg1);
};
```

The implied IDL would be:

```
interface DP_ParObjPart {
    struct ParOperationPartInfo {
        PartRequestInfo         request;
        PartParameterInfo       arg1
        PartParameterInfo       result;
```

```
    };
    typedef sequence<sequence<float>> DP_ParOperationResult;
    DP_ParOperationResult
        ParOperation(in long x, in sequence<float>,
                     inout ParOperationPartInfo part_info);
};
```

### 8.4.4 Performing Collective Invocations

When the implementation of a part object operation (part servant method) requires that all parts collectively perform some operation on another object, whether singular or parallel, it uses a collective invocation. The part servant method code explicitly does this.

Collective invocations require a preparatory step to inform the local client ORB about the required client-side behavior of the invocation and to compute the required partitioning on the client (local) side of the invocation. Since there is a set of parts on the client side of the collective invocation, data partitioning information must be supplied, independent of any actual invocation. The "whole" size of any partitioned data parameters must also be supplied. This step is normally done once, when the part servant first obtains a reference to an object with the appropriate interface, and when it knows the size of the whole data to be partitioned. This is accomplished by using an implied-IDL operation, which the ORB processes locally. This implied operation provides the client stub (ORB) with enough information to correctly perform a collective operation later.

This special, locally processed operation has a name created by prepending a "**collective_setup_**" prefix. It has one **inout** parameter that is implied-IDL structures. This data structures describes the data partitioning and data sizes for each partitionable parameter. Note that this implied-IDL is added to the actual interface rather than in any new interface, but only on the client stubs.

An example of this implied IDL follows. Assume an original IDL of:

```
interface ParObj {
    typedef float Floats[400][400];
    Floats ParOperation(in long x, in sequence<float> arg1);
};
```

and the existing data structure definitions in the standard DP IDL of:

```
module DP {
    struct CollectiveParameterInfo {
        DataPartition   partition;
        Sizes           whole_size;
        Sizes           owned_size;
        Positions       position_in_whole
    };
};
```

The implied IDL would be:

```
interface ParObj {
    struct ParOperationCollectiveInfo {
        CollectiveParameterInfoarg1;
        CollectiveParameterInforesult;
```

```
    };
void collective_setup_ParOperation(
            inout ParOperationCollectiveInfo info
};
```

When the actual invocation is made, a different implied operation is used, whose name is formed by prepending the prefix "**collective_do_**" to the operation name. This operation has the same parameters as the original operation, with arrays changed to sequences (with generated/implied typedefs as in the part interface implied-IDL above), and an additional parameter, added at the end to supply the information returned from the (local) collective_setup operation. This operation is also only generated for the client stub. Continuing the example above, the implied IDL for this would be:

```
interface ParObj {
    typedef sequence<sequence<float>> DP_ParOperationResult;
    DP_ParOperationResult
        collective_do_ParOperation(in long x,
                                   in sequence<float> arg1,
                                   in ParOperationCollectiveInfo
                                   collective_info);
};
```

Collective invocations use the implicit association between the execution context and a specific parallel object since all collective invocations by definition execute in the context of a part operation on some parallel object.

A collective invocation on a parallel object that *is* the current parallel object of the invoking parts is a special case. The performance of this is important, although there is no special interface or behavior defined. It essentially means that the parts are performing an operation on themselves as a set. The parallel-behavior of the operation used can result in a redistribution of data among the parts.

## 8.5 Programming Model Summary

A Data Parallel CORBA application will generally consist of these types of code:

1. Standard CORBA application code that implements and/or uses standard CORBA objects.
2. Standard CORBA application code that uses (acts as clients to) parallel objects.
3. Application code that creates (deploys) parallel objects (using the POM).
4. Application code that implements part objects (implementing the "part version" of the interfaces).
5. Application code that locally creates part servants and objects.(implementing the local GenericFactory or simply creating objects and registering them with the POM).

# 9 Support for Interoperability and Scalability

## 9.1 Concepts

### 9.1.1 Parallel Proxy

A parallel proxy is an object that can be used by clients on non-parallel ORBs to make requests on parallel objects. Parallel proxies are provided by the infrastructure (POM) on request. They are only evident in the specification as a side effect of parallel object (and reference) creation and in IORs. They have no defined IDL. They enable interoperability with clients running on non-parallel ORBs. This behavior is similar to proxy objects and request-level bridging as described in the CORBA specification.

### 9.1.2 Parallel Agent

A parallel agent is an object used by parallel ORBs (as clients) to obtain the complete information about a parallel object. Parallel agents are supplied by POMs such that applications can optionally create them when parallel objects are created. They are evident in the specification by an IOR profile and IDL, but these are used only for ORB interoperability, not by applications. They exist for scalability (parallel objects with many parts).

## 9.2 Interoperability Requirements: Potential System Decomposition

In addition to the application programming model described above, this standard defines interfaces and behavior necessary to achieve interoperability between ORBs when different parts of the same application are on different ORBs. Interoperability is assured when the following parts of the application and infrastructure are all running on *independently developed* ORBs:

1. Singular CORBA clients and servers running on non-parallel ORBs.
2. Singular clients running on parallel ORBs making high performance invocations on parallel objects on other parallel ORBs.
3. Singular servers running on parallel ORBs being the target of collective invocations from part servers.
4. Part servers.
5. The Parallel Object Manager.

## 9.3 Bridging

The parallel proxy object is simply an object that can be used by non-parallel ORBs to invoke operations on a parallel object. When creating parallel objects that must be usable by non-parallel ORBs, a proxy must be created also (and the appropriate profile put in the object reference to the parallel object). The proxy has the same interface and lifecycle as the parallel object, although it does not need to be in any execution context or on any ORB that is running the part objects. Thus, the work of proxies can be distributed separately from the actual part objects.

POMs must be able to implement proxies and offer the option of creating a proxy at the time of creating the parallel object. Note that proxies are not involved in any way when the client code is running a parallel ORB. Proxies specifically and solely exist for purposes of interoperability when client code is running on non-parallel ORBs.

Applications that create parallel objects can choose to include proxy creation as a side effect or not. The only application-visible result is that the object reference works when given to clients on non-parallel ORBs. There is no object type or interface defined for proxies, and there is no (new) IOR profile for proxies.

## 9.4 Object References

References to Parallel Objects support three features:

1. Allow clients to use parallel object references on non-parallel ORBs.
2. Allow references to contain complete information to allow parallel ORBs to directly use part objects, for maximum performance at modest scale.
3. Allow client ORBs to use references to retrieve complete information about the parallel object at runtime for maximum flexibility and scaling.

### 9.4.1 IOR Profiles for Parallel Objects

#### 9.4.1.1 Parallel-Proxy Profile (not new)

A POM can optionally add a standard IIOP profile (with **TAG_INTERNET_IOP**) to parallel IORs that references a Parallel Proxy to allow clients running on non-parallel ORBs to invoke operations on parallel objects. The Parallel Proxy can forward the invocation to the actual parallel (set of parts) implementation of the parallel object. A non-parallel ORB will invoke on a parallel object using this profile (or fail if it is absent).

If the POM does not add a standard IIOP profile to a parallel IOR then only clients on parallel ORBs can use the IOR. The standard IIOP profile is optional because the creation of a Parallel Proxy may cause overhead that is unnecessary for the application. Also, for many aspects of parallel applications, the parallel object is internal to the parallel application and there is no requirement to expose a parallel object to non-parallel ORBs.

#### 9.4.1.2 ParallelRealization Profile

This profile, known only by parallel ORBs, is how a complete description of a **ParallelObject**'s behavior and parts, is embedded in an IOR. It allows for distribution of this information embedded within the IOR and thus does not require any further querying or traffic to start using the part objects. This embedding is not highly scalable but delivers best performance for small-scale parallelism. This technique is similar to that of Fault Tolerant CORBA, where replica profiles are all contained in the IOR. This profile contains a sequence of **PartProfile** structures each of which is a sequence of existing and/or standard profiles for each part object(allowing each part to in fact have multiple profiles). It also contains the **ParallelBahavior** of the parallel object as a whole.

```
module IOP {
    const ProfileId TAG_DP_REALIZATION = xx;
};
module DP {
    typedef sequence<IOP::TaggedProfile>Profiles;
    struct PartsProfileBody {
        GIOP::Version  dp_version;
            ParallelBehaviorbehavior;
        };
};
```

#### 9.4.1.3 ParallelAgent Profile

This profile, known only to parallel ORBs, references an object that will supply information about the parallel object, but does not contain the part profiles as embedded in the **ParallelRealization** profile described above. It is not the same as the proxy reference, although it conceivably could address the same object. The parallel agent interface is described below. This profile will contain a **TAG_DP_BEHAVIOR** component.

```
module IOP {
    const ProfileId TAG_DP_AGENT = xx;
};
module DP {
    struct AgentProfileBody {
        GIOP::Version  dp_version;
        Profiles       profiles;
        ParallelBehaviorbehavior;
    };
    interface Agent {
        readonly attribute Profiles realization;
    };
};
```

The **DP::Agent** interface has one readonly attribute which can be retrieved, which contains the sequence of **PartProfile** structures locating the parts.

### 9.4.2 Parallel Object References

Parallel object references thus contain either a **ParallelRealization** profile or a Parallel-Agent profile, or both. They may also optionally contain a Parallel-Proxy profile (the standard one).

### 9.4.3 Part Object References

References to parts of parallel objects must contain a standard profile that contains a **TAG_DP_BEHAVIOR** component, containing a **ParallelBehavior** value. The interface of a part object reference is not the same as the interface to the parallel object. It is rather the “part version” of that interface, with its associated implied IDL. The implied-IDL interface name is **DP_<ifaceName>Part**, similar to the **AMI_<ifaceName>Handler** in the CORBA Messaging specification.

```
module IOP {
        const ComponentId TAG_DP_BEHAVIOR = xx;
};
```

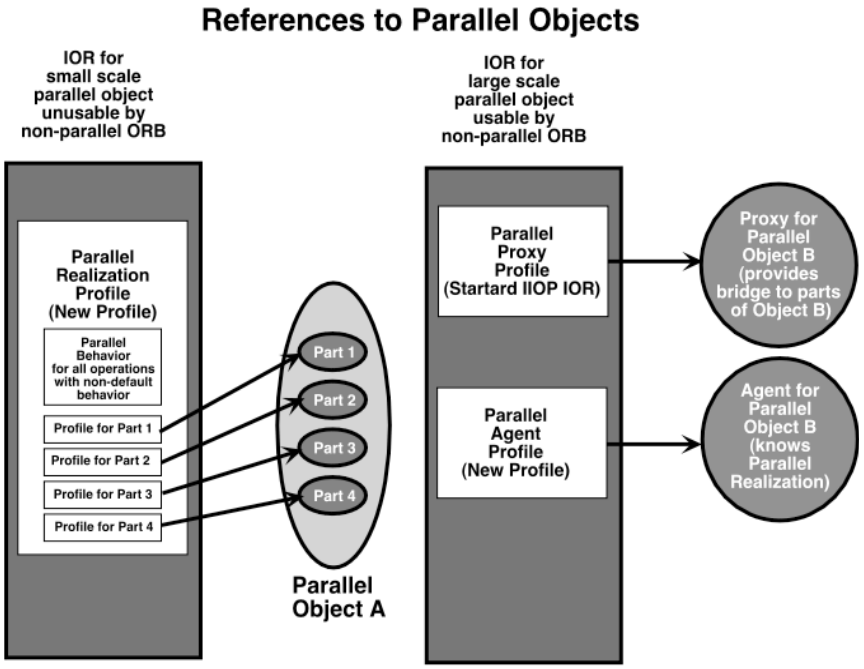


**Figure 9.1 - References to Parallel Objects**

## 9.5 CORBA::Object Operations

The interface **CORBA::Object** contains operations typically implemented by the ORB rather than by the object implementation.

Parts are considered homogeneous with respect to type, security, policies and lifecycle, thus any part can be used for the various operations in CORBA::Object interface that an ORB might do remotely (making requests to the actual implementation of the object). Examples are **get_interface**, **is_a**, **non_existent**, **is_equivalent**.

In general, the Parallel Agent, when present should be usable by the parallel ORB when it needs to make requests on the implementation object needed to implement these operations. Load balancing could conceivably be used across part objects when useful in implementing these operations.

## 9.6 Service Context

The client parallel ORB of a request issued to a parallel object must supply a unique identity for that request in order that the POAs for each part object have a common request identity across all parts. To achieve interoperability, this service context must be defined. This specification uses a structure very similar to that of FT CORBA (**FT_REQUEST** service context), although the goal is somewhat different. A **DP_REQUEST** service context is defined, which simply contains a unique request identity in two parts: a client identifier and a request identifier.

```
module IOP {
    const ServiceId DP_REQUEST = 13;
};
```

```
module DP {
    struct RequestServiceContext { // context_id = DP_REQUEST;
        string client_id;
        long request_id;
};
```

The **client_id** uniquely identifies the client, so that repeated requests from the same client can be recognized. No mechanisms are defined for generating this unique identifier. The **request_id** uniquely identifies the request within the scope of the client. The client ORB can reuse the **request_id** provided that it guarantees uniqueness.

# Annex A
(normative)

# Consolidated IDL

## A.1 PortableGroup IDL

```
#ifndef _PortableGroup_IDL_
#define _PortableGroup_IDL_


#include "CosNaming.idl" // 98-10-19.idl
#include "IOP.idl"       // from 98-03-01.idl
#include "GIOP.idl"      // from 98-03-01.idl
#include "CORBA.idl"     // from 98-03-01.idl

#pragma prefix "omg.org"

module IOP {
    const ComponentId TAG_GROUP = OMG_assigned;
    const ComponentId TAG_GROUP_IIOP = OMG_assigned;
};
module PortableGroup {
    // Specification for Interoperable Object Group References
    typedef GIOP::Version Version;
    typedef string GroupDomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;
    struct TagGroupTaggedComponent { // tag = TAG_GROUP;
        GIOP::Version group_version;
        GroupDomainId group_domain_id;
        ObjectGroupId object_group_id;
        ObjectGroupRefVersion object_group_ref_version;
    };
    typedef sequence<octet> GroupIIOPProfile; // tag = TAG_GROUP_IIOP
    // Specification of Common Types and Exceptions
    // for GroupManagement
    interface GenericFactory;
    typedef CORBA::RepositoryId TypeId;
    typedef Object ObjectGroup;
    typedef CosNaming::Name Name;
    typedef any Value;
    struct Property {
        Name nam;
        Value val;
    };
```

```
typedef sequence<Property> Properties;
typedef Name Location;
typedef sequence<Location> Locations;
typedef Properties Criteria;
struct FactoryInfo {
    GenericFactory the_factory;
    Location the_location;
    Criteria the_criteria;
};
typedef sequence<FactoryInfo> FactoryInfos;
typedef long MembershipStyleValue;
const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;
typedef unsigned short InitialNumberMembersValue;
typedef unsigned short MinimumNumberMembersValue;
exception InterfaceNotFound {};
exception ObjectGroupNotFound {};
exception MemberNotFound {};
exception ObjectNotFound {};
exception MemberAlreadyPresent {};
exception ObjectNotCreated {};
exception ObjectNotAdded {};
exception UnsupportedProperty {
    Name nam;
};
exception InvalidProperty {
    Name nam;
    Value val;
};
exception NoFactory {
    Location the_location;
    TypeId type_id;
};
exception InvalidCriteria {
    Criteria invalid_criteria;
};
exception CannotMeetCriteria {
    Criteria unmet_criteria;
};
// Specification of PropertyManager Interface
interface PropertyManager {
    void set_default_properties(in Properties props)
        raises (InvalidProperty, UnsupportedProperty);
    Properties get_default_properties();
    void remove_default_properties(in Properties props)
        raises (InvalidProperty, UnsupportedProperty);
    void set_type_properties(in TypeId type_id, in Properties overrides)
        raises (InvalidProperty, UnsupportedProperty);
    Properties get_type_properties(in TypeId type_id);
    void remove_type_properties(in TypeId type_id, in Properties props)
        raises (InvalidProperty, UnsupportedProperty);
```

```
    void set_properties_dynamically(in ObjectGroup object_group,
                                    in Properties overrides)
        raises(ObjectGroupNotFound,
               InvalidProperty,
               UnsupportedProperty);
    Properties get_properties(in ObjectGroup object_group)
        raises(ObjectGroupNotFound);
}; // endPropertyManager
// Specification of ObjectGroupManager Interface
interface ObjectGroupManager {
ObjectGroup create_member(in ObjectGroup object_group,
                         in Location the_location,
                         in TypeId type_id,
                         in Criteria the_criteria)
    raises(ObjectGroupNotFound,
           MemberAlreadyPresent,
           NoFactory,
           ObjectNotCreated,
           InvalidCriteria,
           CannotMeetCriteria);
ObjectGroup add_member(in ObjectGroup object_group,
                       in Location the_location,
                       in Object member)
    raises(ObjectGroupNotFound,
           CORBA::INV_OBJREF,
           MemberAlreadyPresent
           ObjectNotAdded);
ObjectGroup remove_member(in ObjectGroup object_group,
                          in Location the_location)
    raises(ObjectGroupNotFound, MemberNotFound);
Locations locations_of_members(in ObjectGroup object_group)
    raises(ObjectGroupNotFound);
ObjectGroupId get_object_group_id(in ObjectGroup object_group)
    raises(ObjectGroupNotFound);
ObjectGroup get_object_group_ref(in ObjectGroup object_group)
    raises(ObjectGroupNotFound);
Object get_member_ref(in ObjectGroup object_group, in Location loc)
    raises(ObjectGroupNotFound, MemberNotFound);
}; // end ObjectGroupManager
// Specification of GenericFactory Interface
interface GenericFactory {
    typedef any FactoryCreationId;
    Object create_object(in TypeId type_id,
                         in Criteria the_criteria,
                         out FactoryCreationId factory_creation_id)
        raises(NoFactory,
               ObjectNotCreated,
               InvalidCriteria,
               InvalidProperty,
               CannotMeetCriteria);
    void delete_object(in FactoryCreationId factory_creation_id)
```

```
            raises (ObjectNotFound);
    }; // end GenericFactory
}; // end PortableGroup

#endif // for #ifndef _PortableGroup_IDL_
```

## A.2 Data Parallel CORBA IDL

```
#ifndef _DP_IDL_

#define _DP_IDL_

#include “IOP.idlíí       // from 98-03-01.idl
#include “GIOP.idlíí     // from 98-03-01.idl
#include “CORBA.idlíí      // from 98-03-01.idl
#include “PortableGroup.idl”

#pragma prefix “omg.org”

module IOP {
    const ProfileId TAG_DP_REALIZATION = xx;
    const ProfileId TAG_DP_AGENT = xx;
    const ComponentId TAG_DP_BEHAVIOR = xx;
    const ServiceId DP_REQUEST = xx;
};
module DP {
    struct RequestServiceContext { // context_id = DP_REQUEST;
        string client_id;
        long request_id;
    };
    interface ParallelObjectManager :
        PortableGroup::GenericFactory,
        PortableGroup::PropertyManager,
        PortableGroup::ObjectGroupManager {};
    local interface Current : PortableServer::Current {
        void set_parallel_behavior(in ParallelBehavior
                                        the_parallel_behavior);
        void clear_parallel_behavior();
    };
    enum OverlapPad {
        OVERLAP_PAD_ZEROS,
        OVERLAP_PAD_REPLICATED,
        OVERLAP_PAD_TRUNCATE,
        OVERLAP_PAD_WRAP
    };
    struct DimensionOverlap {
        unsigned long       amount
        OverlapPad          pad;
    };
    struct DimensionPartition {
        unsigned long           position;    // at part object
```

```
    unsigned long       minimum;      // 0 means unpartionable
    unsigned long       maximum;      // 0 means no maximum
    unsigned long       modulo;       // 0 means 1
    DimensionOverlap    left_overlap;
    DimensionOverlap    right_overlap;
};
typedef sequence<DimensionPartition>    DataPartition;
enum DistributionType {
    DISTRIBUTION_ONE_REQUEST,
    DISTRIBUTION_ONE_FOR_EACH,
    DISTRIBUTION_ZERO_OR_ONE,
    DISTRIBUTION_VARIABLE_FOR_EACH
};
enum DistributionPattern {
    DISTRIBUTION_CYCLIC,
    DISTRIBUTION_PERSISTENT,
    DISTRIBUTION_LEAST_BUSY,
    DISTRIBUTION_RANDOM
};
struct RequestDistribution {
    DistributionType    type;
    DistributionPattern pattern;
    boolean             inform_at_end;
    boolean             sync_before;
    boolean             sync_after;
};
struct ParallelParameter {
    CORBA::Identifier  name;       // empty for return
    DataPartition      partition;
    boolean            compare;    // for inout/out/return
};
typedef sequence<ParallelParameter> ParameterList;
struct ParallelOperation {
    CORBA::Identifier      name;
    ParameterList          parameters;
    RequestDistribution    distribution;
};
typedef sequence<ParallelOperation> ParallelBehavior;
};
exception BadComparison {
    CORBA::Identifier  name;
};
struct PartRequestInfo {
    unsigned long      part_ordinal;
    unsigned long      num_parts;
    boolean            last;
    boolean            empty;
};
typedef sequence<unsigned long> Sizes;
typedef sequence<unsigned long> Positions;
struct PartParameterInfo {
```

```
        Sizes       whole_size;
        Sizes       owned_size;
        Positions   position_in_whole;
        Positions   position_in_local;
    };
    struct CollectiveParameterInput {
        DataPartition   partition;
        Sizes           whole_size;
    };
    struct CollectiveParameterOutput {
        Sizes           owned_size;
        Positions       position_in_whole
    };
    typedef sequence<IOP::TaggedProfile>Profiles;
    struct PartsProfileBody {
        GIOP::Version       dp_version;
        Profiles            parts;
        ParallelBehavior    behavior;
    };
    struct AgentProfileBody {
        GIOP::Version       dp_version;
        Profiles            profiles;
        ParallelBehavior    behavior;
    };
    interface Agent {
        readonly attribute Profiles realization;
    };
};

#endif // for #ifndef _DP_IDL_
```

# INDEX