A Framework for Exploiting Object Parallelism in Distributed Systems*

Chen Wang and Yong Meng Teo

Department of Computer Science
National University of Singapore
3 Science Drive 2
Singapore 117543
teoym@comp.nus.edu.sg

Abstract. To support parallel computing in a distributed object-based computing platform, a uniform high performance distributed object architecture layer is necessary. In this paper, we propose a distributed object-based framework called DoHPC to support parallel computing on distributed object architectures. We present the use of dependence analysis technique to exploit *intra-object parallelism* and an *interoperability model* for supporting distributed parallel objects. Experimental results on a Fujitsu AP3000 UltraSPARC workstation cluster computer show that with intra-object parallel computation speedup efficiency is greater than 90% and with overhead of less than 10% for large problems. In addition, the interoperability model improves speedup by 20%.

1 Introduction

Distributed object technology has attracted much attention recently from high performance computing researchers. The compositional and interoperable features embodied in distributed object technology allow the development of high performance computing software that meets the requirements of open systems [9]. However, performance results reported show that the efficiency of applications encapsulated with CORBA is always far lower than programmed directly in C/C++ [4, 5]. To achieve high performance in distributed object computing, one key issue is to optimize the distributed object architecture. Current distributed object architectures contain three main layers. A *component framework layer* such as OMG's Component Model, Sun's Java Beans and Microsoft's ActiveX. A *distributed architecture layer* such as OMG's CORBA, Sun's Java RMI, and Microsoft's DCOM. The *process communication layer* such as OMG's GIOP/IIOP and OSF's DCE. Current research concentrates mainly on optimizing the event delivery mechanism of ORB and in improving the efficiency of event service in CORBA. This includes various implementations such as multi-threaded ORB [6], high performance and real-time ORB [11, 14], optimized IDL

* *Proceedings of the 8th International Conference on High Performance Computing and Networks*, pp. 668-677, LNCS 1823, Springer-Verlag, Amsterdam, May, 2000

(Interface Definition Language) compiler [2], and environment sensitive event delivery mechanism [13].

Besides, there is some research that focus on making the component framework layer suitable for parallel computing to facilitate extending parallel computing over wide-area networks. In general, this is achieved by implementing the component framework layer based on parallel programming languages or mechanisms. Examples include LSA [3] that is based on HPC++, and in POOMA [10] the key mechanism to ensure reusability is the field class in which data-parallel operations are encapsulated. The main reasons for using parallel languages or mechanisms are: (a) Current distributed object systems do not support these parallel programming languages and the underlying parallel mechanism; (b) The efficiency of current distributed architecture layer does not meet the requirement for many parallel applications. To support parallel computing more efficiently on distributed architectures, we propose a uniform distributed architecture layer called DoHPC. Difference in remote invocation models for parallel computing and in distributed computing hinders efficiency when their components must interoperate. For example, HPC++ based on IIOP in LSA must be efficiently implemented to facilitate interoperation with external components that are defined using the OMG's Component Model.

In this paper, we propose a distributed architecture layer framework that aims to achieve high performance while keeping the interoperability with other components. Our major contributions include a method to exploit implicit parallelism in distributed objects through dependence analysis at compile-time, and an interoperability model to support distributed parallel objects that reduce internal message transfer. Section 2 presents the architecture of DoHPC. Sections 3 and 4 discuss DoHPC supports for intra-object parallelism and the interoperability model for distributed parallel objects respectively. Performance analysis is discussed in section 5. In section 6, we discuss related works on optimizing distributed architecture and comparing them with DoHPC. Section 7 summarizes our conclusion.

2 Overview of DoHPC

In DoHPC, the active object concept is fully supported. A distributed object is an active object in DoHPC, which is a service-providing entity as well as an autonomous entity that could also run its own program. DoHPC supports two categories of parallelism in distributed objects: **inter-object parallelism** and **intra-object parallelism** using the following mechanisms:

1. To exploit **inter-object parallelism**, a mechanism is provided for client to invoke methods of supplier asynchronously.
2. To exploit **intra-object parallelism**, we provide a mechanism for the supplier to serve more than one request in parallel and to dispatch some of these requests to idle processors.
3. A mechanism to support the **interoperability of distributed parallel objects** efficiently. Current interoperability model of distributed object cannot

harness the parallelism among distributed parallel object very well [7]. This can be regarded as a special case of inter-object parallelism.

As shown in figure 1, DoHPC consists of *three* parts: client, supplier and a runtime system. A *client* consists of client applications and several distributed objects' implementation stubs. The *supplier* consists of a *Method Queue*, a *Deciding Controller*, an *Event Service mechanism*, a *Transaction Controller*, *Port Information* and *Runtime System.* The method invocation queue (Method Queue) stores all client requests for the supplier. The *Deciding Controller* accesses the Method Queue and dispatches methods to processors with lighter workload. The *Event Service* is used to "push" message from supplier to clients and the *Transaction Controller* executes methods of a supplier object. To improve the efficiency of a parallel supplier, the *Port Information* mechanism ensures that all parallel executing methods can interoperate with clients and other parallel objects directly. The *runtime system* provides common services such as synchronous/asynchronous message passing, name service, distributed shared-memory service and object schedule service, etc.

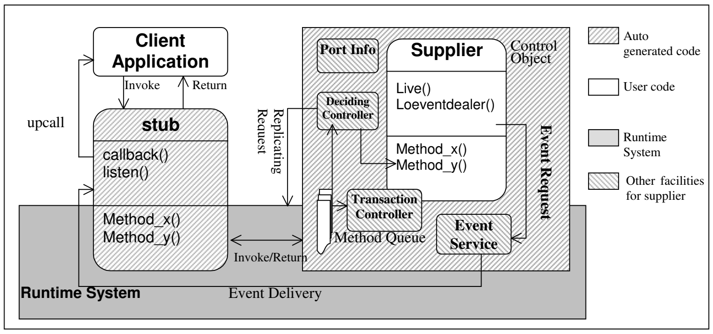

Fig. 1. Architecture of DoHPC

3 Support for Intra-Object Parallelism

Intra-object parallelism is exploited implicitly by automatically dispatching clients' requests to different processors. Methods belonging to an object that have no dependence are executed in parallel. Inter-method dependence in DoHPC is determined during compilation.

3.1 Inter-Method Dependence Analysis

Figure 2 shows the inter-method dependence analysis process. Firstly, the *dependence analyzer* obtains the method names from an *IDL file*, starts the analysis by invoking the analyzer for a specific programming language and generates a new IDL file. In the new IDL file, a method declaration consists of its method name, return type and parameter list but also a list of all dependent methods.

Secondly, an *enhanced IDL compiler* automatically generates the *dependence table*, *supplier skeletons* and *client stubs*. The dependence table together with the implementing skeleton of a distributed object is used by the Deciding Controller to schedule methods for execution. To obtain the inter-method dependence, we require the summary information for each method, namely **flow sensitive information** and **flow insensitive information**. Computation of the flow sensitive information is a NP problem when variables are permitted to have alias [1].

Our Dependence Analyzer currently computes only the flow insensitive information. Let p and q denote methods in the interface declaration of a distributed

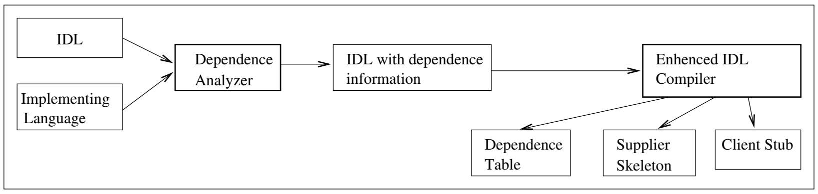

Fig. 2. Inter-Method Dependence Analysis

object. W_p, W_q and R_p, R_q denote sets of all fields p, q write and read in the superclass, superinterface and implementing class of this interface. In general, if $(R_p \cap W_q) \cup (W_p \cap R_q) \neq \Phi$, then p depends on q or q depends on p, otherwise no dependence exists between p and q. This is called **strong dependence**. It is necessary to relax this condition to increase parallelism. Assume p is ahead of q in the Method Queue, then

1. If $W_p \cap R_q \neq \Phi$, the dependence is called **true dependence**. Thus, p, q must be executed in sequence to ensure the correctness of the result and these two methods cannot be parallelized.
2. If $R_p \cap W_q \neq \Phi$ and $W_p \cap R_q = \Phi$, the dependence is called **pseudo-dependence** and the two methods can be executed in parallel. To ensure correct computation, shared variables must be localized in the processes running p and q. After p completes, these variables should be merged. In DoHPC, the localization is achieved through replication of an object.
3. If $W_p \cap W_q \neq \Phi$ and $W_p \cap R_q = \Phi$, the dependence is called **pseudo-dependence**, also called **output dependence**. The two methods p, q can be executed in parallel and those shared variables q holds should be merged after the execution of p is completed.

For illustration, we discuss DoHPC dependence analyzer for the Java programming language. The read/write set, S, for a Java Interface comprises of three parts: *all fields inherited from its superclass*, *all fields inherited from its super interfaces*, and *all fields of itself.* Private fields of superclass or super interface are excluded in the read/write set. Methods of a distributed object that are accessible to its clients include all public methods defined in its interface and all public methods it inherited. For each method p, the analyzer generates two sets R_p, W_p containing the flow insensitive information. There are twelve assignment operators in Java. If v, where $v \in S$, appears on the left of these operators in method p then $W_p = W_p \cup \{v\}$ else v is on the right and $R_p = R_p \cup \{v\}$. Assume that the alias of v exists, such as,

```
SOMECLASS obj = new SOMECLASS();
SOMECLASS another_obj = obj;
```

Operations on *another_obj* are in fact performed on *obj*. In this case, *another_obj* is called an alias of *obj*. The DoHPC analyzer maps aliases to the original variable first, then merge all read/write sets of these aliases to the flow insensitive information of their original variables. For cases whereby there are function calls in the source code of method p, such as, *foo(arg1,arg2,...,argn)*, the flow insensitive information of p can be obtained with the algorithm presented in [12].

Client requests for a supplier are stored in a method queue. For each *method_i*, a corresponding entry generated at compilation time is placed in the Dependence Table. The entry consists of three fields:*method_i*, W_{method_i} and R_{method_i}. The deciding controller schedules methods in the method queue for execution. The scheduling algorithm can be found in [12].

4 Support for Interoperability of Distributed Parallel Objects

The efficient interoperation among objects and with particular client's remains a problem. Since the IDL is unable to describe how its methods are scheduled, each call has to be forwarded to those processors on which the object is running. This increases the communication overhead. We propose an interoperability model that allows distributed parallel objects to interoperate more efficiently.

To generalize the characteristics of *data parallelism* and *task parallelism*, we introduce task parallel and data parallel object in DoHPC. Two classes *PObj* and *DSeq*, define the templates for these two kinds of objects. *PObj* provides an abstracted interface of distributed object's interoperation. *DSeq* describes the common interface of distributed data in a data parallel program, and can be used as the base class for implementing data parallel object.

4.1 Supporting Task Parallel Distributed Objects

Figure 3(a) shows the structure of *PObj*. Darken arrows represent normal channels that a distributed object used to map network message to its methods, while hollow arrows represent channels for interoperation between parallel programs. A port class defined in *PObj* contains the mapping information of methods already dispatched by *PObj*'s scheduler (see [12] for details of the *port* and *PObj* classes).

The address portion consists of three parts: global process communicating address (host name and process_id), object name, and method name. This definition contains input as well as return value, and is implemented as a duplex port. It can be used to represent the call and reply message when performing

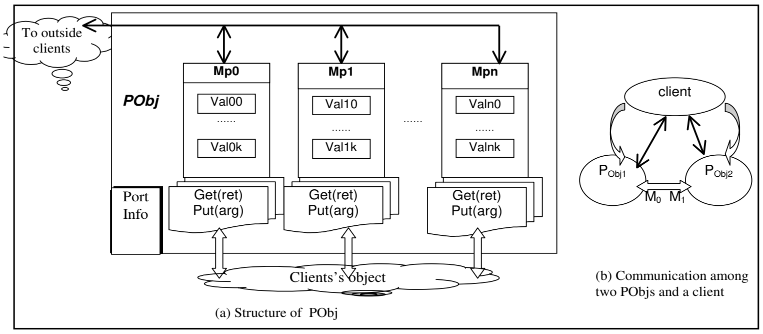

Fig. 3. Interoperability Model for Task Parallel Distributed Objects

remote method invocation. To minimize extra synchronization overhead, we restrict concurrent execution on one processor by closing the port to all other requests before the current active client release it. Only the scheduler can spawn parallel executing object entities.

Client requests without port operation is treated as common object method invocations. The client stub automatically converts client requests with port operation into two phases: **port query** and **method invocation**. The former fetches the runtime information of the requested method while the latter send request directly to the corresponding port.

4.2 Supporting Data Parallel Distributed Objects

The distributed object implementation using the data parallel model provides various transportation methods to support distributed data interoperation. However, we can consider distributed data as an object too. Distributed data objects can be viewed as task parallel objects that are distributed among several nodes and provide data services to client applications. A major difference is that a *DSeq* object provides many array operation and data distribution functions. In addition, methods of *DSeq* are not dynamically dispatched by the scheduler, but dispatched to different processors during object construction (through dispatch) and provide services to data parallel applications. *DSeq* provides data distribution information through *get_part_info()*. The method called Partition in *DSeq* allows client to define data partition ratio, for example, prt.strsplit="1:2:2:4" denotes partitioning these data into four blocks as 1:2:2:4 and distributing these blocks into four processors. Data partition by row or column is supported, for example, part.rowsplit = BLOCK denotes putting data sequentially on each nodes by row, with the length of each row defined by BLOCK. Each data block is assigned a port for listening to the data request of applications and send related data to them. It also incorporates an interface to the scheduler, so that the client can dispatch data to it through the scheduler. The declaration of *DSeq*'s interface can be found in [12]. This class overrides the subindex operator, so that parallel statements can get the port number of a data portion transparently when accessing the data resource, and complete data operations through port access.

5 Performance Analysis

The test environment is a Fujitsu AP3000 workstation cluster consisting of 143MHz SUN UltraSPARC processors running the Solaris operating system. Processors are connected via a 200MB/sec bi-directional Torus network. The Fujitsu PVM/AP message-passing library is used.

5.1 Intra-Object Parallel Computation

Detail results of inter-object parallelism and remote method invocation can be found in [12]. To analyze the trade-off between transparency of obtaining parallelism and the performance of the intra-object scheduler, we implemented the Mandelbrot set example.

In the implementation of the Mandelbrot class, we use a public method to compute the vertical lines with fix horizontal distance in a Mandelbrot set. The method places the computed result into a matrix. Obviously, the method has access dependence with itself because two simultaneous invocations write their results to the same matrix. Let l denotes the fix distance and assume a client invokes the method l times with different starting vertical line to get the whole Mandelbrot set, and the scheduler dispatches all these requests to execute in parallel on idle processors. Since the supplier is duplicated during dispatching, the access dependence will not affect the achieving of parallelism. The maximum iteration number is 1024.

We implemented the Mandelbrot set as a parallel program and as a DoHPC program, and collected the elapsed times for different plane sizes and varying the number of processors from 2(P2) to 16(P16). The percentage differences in

Plane Size		Speedup				Efficiency			
		P2	P4	P8	P16	P2	P4	P8	P16
Parallel	200×200	1.92	3.74	7.59	13.32	0.96	0.94	0.95	0.83
	400×400	1.92	3.89	7.70	15.05	0.96	0.97	0.96	0.94
	800×800	1.96	3.98	7.71	15.59	0.98	0.99	0.96	0.97
DoHPC	200×200	1.88	3.65	7.28	12.17	0.94	0.91	0.91	0.76
	400×400	1.90	3.82	7.38	13.89	0.95	0.96	0.92	0.87
	800×800	1.94	3.91	7.50	14.65	0.97	0.98	0.94	0.92

Table 1. Mandelbrot Set – Speedup and Efficiency

elapsed time show that DoHPC is problem size scalable, e.g., for 16 processors, the overhead reduces from 9.3%(200×200) to 6.4%(800×800). Table 1 shows that speedup efficiency of DoHPC for the Mandelbrot set with large problem size is greater than 90%. DoHPC's overheads for achieving transparent intra-object parallelism include scheduling, object replication, state recovery and in the merging of results. The results indicate that the intra-object parallelism mechanism is efficient and is scalable.

5.2 Interoperability of Distributed Parallel Objects

To evaluate the performance of interoperability among distributed parallel objects, we use a 2D-FFT (2 Dimension Fast Fourier Transform) parallel algorithm.

We compare three different implementations of the 2D-FFT algorithm. Assume we have eight images with each image represented by a 512×512 matrix, and using eight processors as the execution platform. In the *first case*, we use one distributed parallel object and dispatch the execution processes of its 1D-FFT method belonging to this object to the eight processors for execution. In the *second case*, we use two distributed parallel objects and divide the processors into two groups, each containing four processors. Computation starts with the first object computing the 1D-FFT of rows in parallel, and on completion, the object transposes the matrix and transfer it to the second object. Subsequently, the second object completes the computation of the matrix in parallel. In the meantime, the first object starts computation on the next image (matrix). For the two distributed parallel objects to interoperate, we adapted the CORBA interoperability model of communicating through the scheduler. The *third case* is similar to the second but implements the DoHPC interoperability model distributed parallel objects. The interoperation of two distributed objects is analogous to a 2-stage pipeline, and overlaps communication with computation. Table 2 shows

Number of Processors	Schemes	Elapsed Time(ms)		Speedup	
		8 images	16 images	8 images	16 images
4	one object	157051	308606	3.14	3.23
	two distributed objects – CORBA	156549	301333	3.15	3.31
	two distributed objects – DoHPC	152333	291136	3.24	3.43
8	one object	84394	174513	5.85	5.72
	two distributed objects – CORBA	83002	166163	5.94	6.01
	two distributed objects – DoHPC	77925	148925	6.33	6.70
16	one object	48293	90294	10.23	11.05
	two distributed objects – CORBA	47486	92682	10.40	10.77
	two distributed objects – DoHPC	41876	84030	11.78	11.87

Table 2. 2D-FFT – Elapsed Time and Speedup

the elapsed time and speedup for 4 to 16 (divided into 2 groups of 8 processors each) processors.

As the dispatching of rows of data to processors can incur a high communication overhead, the performance result of the 2D-FFT is better with two distributed objects (overlaps in computation and communication) than using one distributed object. However, when the number of processors reaches 16, speedup with CORBA style interoperability model is even lower than with one object (16 images). In this case, processors used by the second object is blocked waiting for data from the scheduler, and hinders the smooth operation of the pipelined 2D-FFT algorithm. In DoHPC, the interoperability model achieves higher speedup in this situation by ensuring that these processors receive data from the first object directly.

One of the factors that influence speedup is the degree of problem parallelism relative to the available machine parallelism. For a fixed problem size and increasing machine parallelism, the speedup varies from 79-86% on 4 processors, 71-84% on 8 processors and 64-74% on 16 processors. However, DoHPC's interoperability model improves speedup efficiency by up to 20% over the one object implementation. Data-dispatching overhead is reduced by communicating through port in DoHPC.

6 Related Works

Much research has been carried out to optimize the underlying layer of distributed object architecture, such as improving the efficiency of current distributed object's communication [6, 2], extending the real-time and QoS features [11, 14] and supporting the mobile environment in a distributed object framework [13]. Some frameworks support the composition of parallel component, such as LSA [3], POOMA [10], PARDIS [8], etc.

DoHPC differs Multi-threaded ORB in two aspects. Firstly, DoHPC provided partial concurrency control through inter-method analysis, which makes the achieving of parallelism in a distributed object more transparent to its clients. Secondly, the implementation of multi-threaded ORB is based on parallel computers with shared memory, such as SMP parallel computer. DoHPC is designed for both shared- and distributed-memory systems.

PARDIS is a distributed system that aims to provide interoperability between distributed, heterogeneous, and data-parallel objects [7]. DoHPC differs from PARDIS in supporting the interoperability of both SPMD and MPMD parallel objects through a uniform interoperating model.

LSA provided a visual programming environment to compose parallel components. Components transfer data from one to another through ports. The port object in LSA could establish a data channel differing from the stub/skeleton channel in common distributed object architecture. However, the port of LSA cannot transfer data from one parallel thread in an object to another object directly. This function is supported in DoHPC.

7 Conclusions

In this paper, we present a distributed object-based framework for parallel computing called DoHPC to exploit both intra- and inter-object parallelism in distributed object architecture. The framework emphasizes both performance and compositional features in distributed object architecture. The implicit intra-object parallelism is exploited through inter-method dependence analysis. The proposed interoperability model exploits both task and data parallelism. Performance results obtained on a cluster of UltraSPARC workstations show that implicit intra-object parallelism can be efficiently exploited and are scalable. Support for interoperability of distributed parallel objects improves speedup by 20%.

The functionality of DoHPC can be extended in the following directions. Firstly, the dependence analyzer can be enhanced to estimate the computing complexity of programs to improve on load balancing. Secondly, we can use parallel compilation techniques to partition compute- intensive methods to exploit intra-method parallelism for parallel execution. Thirdly, more efficient method of distributing shared data among processors is necessary for data that has to be replicated many times. This reduces the scheduling cost and enhances the scalability of the scheduler. Finally, the data parallel mechanism can be optimized for various network environments. For example, in a wide-area network with different types of connection mode, how to make use of them in the data parallel mechanism efficiently is important for the overall efficiency of a parallel application.

References

1. J. Barth, A Practical Inter-procedural Data flow Analysis Algorithm, *Communications of the ACM*, Vol. 21, No. 9, pp. 721-736, 1978.
2. E. Eide, K. Frei et.al, Flick: A Flexible, Optimizing IDL Compiler, *ACM SIGPLAN Notices*, Vol. 32, No. 5, May 1997.
3. D. Gannon, R. Bramley et.al, Developing Component Architectures for Distributed Scientific Problem Solving, *IEEE Computational Science & Engineering*, Vol. 5, No. 2, April-June 1998.
4. A. Gokhale and D. C. Schmidt, Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceeding of SIGCOMM'96*, pp. 306-317, Stanford, CA, August 1996.
5. A. Gokhale and D. C. Schmidt, The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In *Proceedings of GLOBECOM'96*, pp. 50-56, London, England, November 1996.
6. A. Gokhale and D. C. Schmidt, Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks., In *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997.
7. K.Keahey, A Model of Interaction for Parallel Objects in a Heterogeneous Distributed Environment, *Technical report*, `ftp://ftp.cs.indiana.edu/pub/techreports/TR467.ps.Z`, Sept, 1996
8. K. Keahey, Dennis Gannon, PARDIS: A Parallel Approach to CORBA, Technical report, `ftp://ftp.cs.indiana.edu/pub/techreports/TR475.ps.Z`, February 1997.
9. O. Nierstrasz, A Tour of Hybrid-A Language for Programming with Active Objects, *Advances in Object-Oriented Software Engineering*, ed. D. Mandrioli and B. Meyer, Prentice-Hall, 1992.
10. J. Reynders, P. J. Hinker et.al, POOMA: A Framework for Scientific Simulation on Parallel Architectures, available in: `http://www.acl.lanl.gov/pooma/documentation`.
11. D.C. Schmidt, S. Mungee et.al, Alleviating Priority Inversion and Non-determinism in Real-Time CORBA ORB Core Architectures, In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, San Francisco, December 1997.

12. C. Wang and Y. M. Teo, Supporting Parallel Computing on a Distributed Object Architecture, *Technical Report*, School of Computing, National University of Singapore, February 2000.
13. G. Welling, B.R. Badinath, Mobjects: Programming Support for Environment Directed Application Policies in Mobile Computing, *ECOOP95*,1995.
14. V.F. Wolfe, L.C. DiPippo, et.al, Real-time CORBA, In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montréal, Canada, IEEE, June 1997.