

Transparent Distributed Processing For Rendering

Peter Kipfer*

Computer Graphics Group
University of Erlangen

Philipp Slusallek †

Computer Graphics Group
Stanford University

Abstract

Rendering, in particular the computation of global illumination, uses computationally very demanding algorithms. As a consequence many researchers have looked into speeding up the computation by distributing it over a number of computational units. However, in almost all cases did they completely redesign the relevant algorithms in order to achieve high efficiency for the particular distributed or parallel environment.

At the same time global illumination algorithms have gotten more and more sophisticated and complex. Often several basic algorithms are combined in multi-pass arrangements to achieve the desired lighting effects. As a result, it is becoming increasingly difficult to analyze and adapt the algorithms for optimal parallel execution at the lower levels. Furthermore, these bottom-up approaches destroy the basic design of an algorithm by polluting it with distribution logic and thus easily make it unmaintainable.

In this paper we present a top-down approach for designing distributed applications based on their existing object-oriented decomposition. Distribution logic, in our case based on the CORBA middleware standard, is introduced transparently to the existing application logic. The design approach is demonstrated using several examples of multi-pass global illumination computation and raytracing. The results show that a good speedup can usually be obtained even with minimal intervention into existing applications.

CR Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: Distributed programming, Parallel programming; D.1.5 [Object-oriented Programming]; I.3.3 [Computer Graphics]: Picture/Image generation, Viewing Algorithms; I.3.7 [Three-Dimensional Graphics and Realism]: Radiosity, Raytracing

Additional Keywords: Distributed Processing, Parallel Processing, Object-oriented Design, Design Pattern, Global Illumination, Lighting Networks

1 INTRODUCTION

Usually, distributed algorithms differ considerably from their non-distributed versions. In order to achieve optimal performance, care-

ful attention has been paid to issues such as load-balancing, communication patterns, and data and task management. These issues can easily dominate the core application logic of distributed algorithms. This is in particular true for systems that allow for the flexible combination of different distributed algorithms at run-time.

The loss of application logic in a sea of complex distribution issues is a severe and growing problem for reasons, such as increased application complexity, increased maintenance cost, or simply educational purposes. In particular maintenance and portability to different hardware architectures has always been a major issue with distributed applications. Also, for development and debugging purposes, it is often desirable to run the same code in a non-distributed serial fashion. This is often impossible with code designed for distributed applications where distribution logic is deeply embedded into the code.

Finally, probably the most important reason for keeping distribution issues transparent to the application programmer is the need to add distributed computation to an existing application. Here we need to add new features with as little impact on the existing application as possible.

Creating a transparent distribution infrastructure avoids many options for optimization and thus will very likely offer inferior performance than distribution code that is deeply integrated with the application. Thus, our work partly relies on the fact, that the increased availability of cheap but high-performance computers allows us to trade non-optimal efficiency for simpler, cleaner, and more maintainable application code, of course within limits.

The object-oriented design of an application is the main starting point for achieving transparent distribution. The basic idea of object-orientation, the encapsulation of data and algorithms in units that communicate via messages, carries over nicely to distributed systems where objects now live in separate address spaces. All that needs to be changed, is the way these objects communicate with each other, so they do not need to be aware of the fact that a peer object may actually be located on a different computational unit.

Object-oriented middleware like CORBA [OMG98a] already provides much of the required distribution infrastructure, such as location, communication, and network transparency. However, from a programmers perspective, CORBA is still highly visible due to CORBA-specific types in interface definitions and the requirements that distributed objects and their proxies derive from CORBA-specific classes. Furthermore, interfaces that work well with colocated objects can result in high communication costs if these objects get separated across a network. This raises the need to transparently adapt the interfaces for objects that may be distributed.

In the remainder of this paper we present several design patterns for hiding the distribution infrastructure in distributed object-oriented systems. These patterns emerged from our work on speeding-up an existing large system for rendering and global illumination [SS95] by distributing it across a network of computers. For educational purposes, we required the distribution infrastructure to be highly invisible to the normal programmer. For practical reasons we could not afford to redesign the whole system around some intrusive distribution framework.

* Am Weichselgarten 9, 91058 Erlangen, Germany
Email: kipfer@informatik.uni-erlangen.de

† Gates Building 364-3B, Stanford, CA, 94306, USA
Email: slusallek@graphics.stanford.edu

Thus, we concentrated on encapsulating distributed and non-distributed modules, and on providing interface adaptors that take care of distribution issues. The result is a system with a highly configurable distribution infrastructure that is mostly invisible to the programmer and the user, but still achieves good parallel performance. Although we concentrate on distributed processing across a network of computers in this paper, the same design patterns are also being used for parallel execution of modules within the same address space on computers with multiple CPUs (see Section 4).

1.1 Previous Work

There have been a large number of papers on parallelization and distribution of rendering and lighting simulation algorithms. Good surveys are available in [RCJ98, CR98, Cro98]. Most of the papers concentrate on low-level distribution for achieving high performance (e.g. using such tools as PVM [GBD⁺94] or MPI [GLS94]). One of the few exceptions is the paper by Heirich and Arvo [HA97] describing an object-oriented approach based on the Actor model. Although this system provides for location and communication transparency, the distribution infrastructure is still highly visible to the programmer.

Several object-oriented frameworks for supporting parallel or distributed programming have been suggested (e.g. POET [MA] or EPEE [Jez93]). POET is a C++ toolkit that separates the algorithms from the details of distributed computing. User code is written as callbacks that operate on data. This data is distributed transparently and user code is called on the particular nodes on which the data is available. Although POET as well as all other frameworks abstracts from the underlying message passing details, it requires to adapt the algorithms to the given structure of the framework and is thus not transparent to the programmer.

Other approaches view local resources only as a part of a possibly world-wide, distributed system (“computational grids”, “world-wide virtual computer”), for instance Globus [FK97] or Legion [GLFK98]. While these are certainly a vital contribution to distributed computing, the demands on the code are significant and by no means transparent to the programmer, which is the main goal of our efforts.

2 DESIGN PATTERNS FOR TRANSPARENT DISTRIBUTION

In the following we present an integrated approach to parallelization and distribution of application modules. It is based on the fact, that object-oriented systems should be and usually are composed of several quite independent subsystems. In contrast to addressing parallelization at the level of individual objects, larger subsystems of objects usually offer a better suited granularity for distributing computation across computers. These subsystems are often accessed through the interface of a single object using the “facade” design pattern [GHJV95].

In an application based on this common design approach, these few facade classes can easily be mapped to CORBA interfaces [OMG97], providing the basis for distributing the application. However, this initial step does not solve our problem, as the CORBA-specific code would be introduced at the heart of our application and we do not want the details of distribution to be visible to a developer. Ideally developers should be able to concentrate on their problem instead of being unnecessarily forced to consider distribution-specific issues, like network latencies, CORBA-types, request-bundling for optimized transport, marshaling and object serialization, mapping of class creation requests to factory methods, and the handling of communicating threads for asynchronous operations.

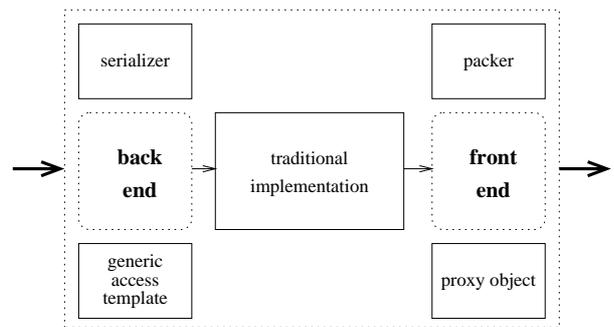


Figure 1: Wrapping existing implementations promotes code reuse by enabling traditional classes to communicate with the distributed system through the services provided by the wrapper. Because these services emulate the traditional interfaces to the contained class, and with the help of templates, this requires almost no manual coding.

We have chosen to build a new distribution interface that completely hides the CORBA distribution infrastructure from the application. This new interface provides the illusion of traditional, non-distributed classes to the outside, while internally implementing optimized distributed object invocations. It is based on asynchronous communication with a multi-threaded request-callback scheme to enable a maximum of parallelism. Additionally, the framework performs load balancing and bundling of requests to avoid network latencies. These are the key concepts that allow us to optimally make use of CORBA and its current synchronous method invocation paradigm (the new CORBA Messaging specification [OMG98b] add asynchronous method invocation, but is only now becoming available).

For encapsulating existing interfaces, our framework provides base classes that provide management services for object creation, communication transport control and synchronization and many other services (see below). Our wrapper for the subsystems that contain the rendering and illumination algorithms use and inherit from these base classes.

For example, our main management class, which controls the overall execution of the rendering task, must be able to define certain synchronization points to ensure that all distributed objects have the same view on the whole system. This occurs for example when waiting for all distributed rendering objects to finish their setup and scene parsing routines before invoking rendering commands. Additionally, these management classes provide host machine information, a scripting engine for configuring the distribution of objects, resource locking, and access facades for the managed subsystem while hiding the use of CORBA completely. In the next three subsections, we address the basic patterns used to implement this approach.

2.1 Wrapping for Distribution

In order to actually reuse the existing object implementations within a distributed environment, our distribution framework provides wrappers for entire subsystems. A wrapper actually consists of two half-wrappers that encapsulate the subsystem as a CORBA client (calling) and as a server (called). We assume that a subsystem is represented by at least one abstract C++ facade class, that defines the interface of the subsystem. We also assume that the subsystem communicates with the outside through interfaces defined by similar facade classes.

We replicate each of these interfaces in CORBA IDL using struc-

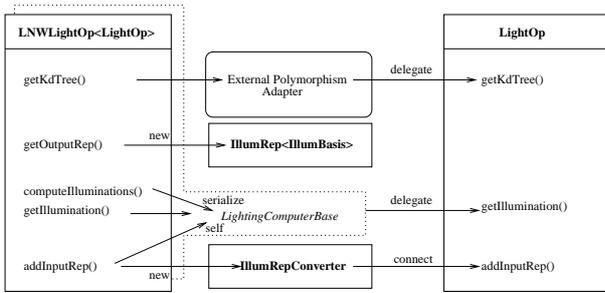


Figure 2: Specific method calls can be forwarded to the implementation in a pseudo-polymorphic way, while general functions like serialization of request packets are inherited from template base classes which in turn implement the abstract interface declaration (see also Figure 6).

tures to pack relevant object data that needs to be transferred (the object by value extension of CORBA has not been available until very recently). Most often we also define new methods that allow for the bundling of multiple requests on the calling side. We then implement the server side by forwarding the requests to the wrapped facade object in a pseudo-polymorphic way [CS98], serializing any bundled messages that arrive, and managing asynchronous calls (see Figure 1).

For the client role of a wrapped subsystem, we need to instantiate C++ classes that derive from a distributed C++ proxy template. They translate the calls from the old C++ interface to calls, that use the CORBA object references. This layer is also responsible for bundling individual calls and using new asynchronous interface methods for bundled requests within the CORBA interface.

Although this wrapping seems complicated and does require some small amount manual coding, most of the work can be delegated to generalized template abstract base classes (see Figure 2). When viewed from the outside, the encapsulated subsystem looks just like a distributed CORBA object using the equivalent CORBA IDL interface. To the contained object, the wrapper looks exactly like any other part of the traditional system using the old C++ interfaces.

The biggest benefit of using this kind of wrappers is the possibility of reusing existing code. While this does not take advantage of parallelization within a subsystem, it enables the distribution and parallelization of different subsystems. This can be of great value, in particular when multiple memory-intensive algorithms have to be separated across multiple machines. The interfaces, provided by the wrappers, finally allow wrapped traditional objects to transparently cooperate with other distributed objects as they are introduced in Section 2.3.

2.2 Replication and Request-Multiplexing

In order for old code to use distributed subsystems, we need an additional wrapper. Its interface is derived from the old C++ facade interface, but it translates the messages to corresponding calls to distributed CORBA objects, e.g. those from Section 2.1. As mentioned before, this translation has several aspects. For one, it translates between traditional and CORBA types where object data needs to be copied into IDL structures. Second, small individual requests may be accumulated and sent across the network in bundles, thus avoiding network traffic overhead.

In addition, we take the opportunity of the wrapper to perform multiplexing and re-packeting of requests across a pool of functionally identical CORBA servers. This enables us to distribute

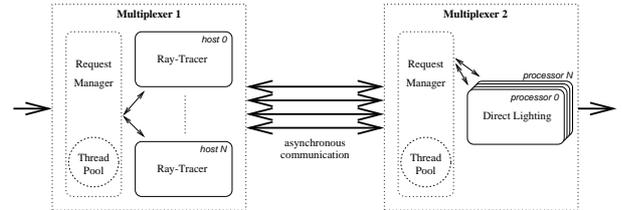


Figure 3: Multiplexers distribute requests equally to functionally equivalent objects either distributed across a network (data-parallel ray-tracers) or running in different threads (colocated lighting objects). Note that the multiplexers do not contain the computation classes, rather they supply the requests and manage the transport of the responses. The embedded request managers use a request/callback model and a thread pool to achieve asynchronous communication.

the computational load evenly using load balancing performed by the wrapper. However, because of the current synchronous nature of CORBA method calls, multiplexing needs to use the request-callback scheme [SV96] provided by our base classes.

Load balancing is performed by sending requests to the server with the lowest load. To this end, the servers maintain FIFOs of requests to balance network latencies. The fill-level of those FIFOs is communicated back to the wrappers piggy-packed on data returned in the callbacks.

Using this scheme, the multiplexed classes look to the outside like a single, more powerful instance of the same subsystem. The benefit of this approach is that by using wrappers and multiplexers, existing code can fairly easily be wrapped, replicated, and thereby sped up. While multiplexers fan out requests, the wrappers in Section 2.1 automatically combine and concentrate asynchronous requests from multiple clients. Note that both patterns perfectly meet our goal of distribution transparency and do not alter the application logic of the remaining system at all.

The following pseudo-code shows how a multiplexer for lighting computations inherits the interface of the lighting base class and overloads the computation request method by implementing some scheduling strategy (see also Figure 6).

```
IDL:
interface LightOp {
    void computeIlluminations(in sequence<Request> req);
};

interface Multiplexer : LightOp {
    void addLightOp(in LightOp op);
};

C++:
class Multiplexer : public IDLMultiplexerInterface {
    virtual void addLightOp(LightOp op)
    { lightOpList_.push_back(op); }
    virtual void computeIlluminations(Request req[]) {
        int idx= determineBestServer()
        lightOpList_[idx]->computeIlluminations(req);
    }
protected:
    vector<LightOp> lightOpList_;
}
```

2.3 Transparent Services

Some subsystems are computational bottlenecks and promise to offer substantial speed-up when they are completely re-implemented

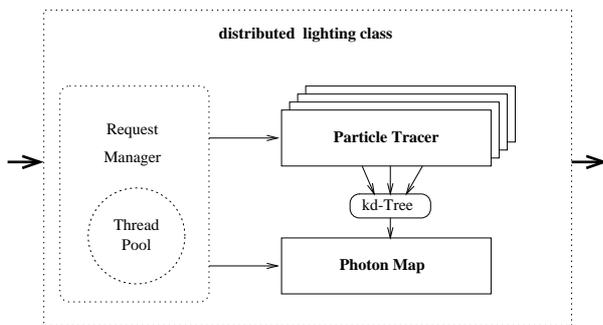


Figure 4: Distribution and parallelization services provide support for implementing advanced computation algorithms.

to take advantage of distribution. Our framework provides distribution and parallelization services within the wrapper classes that go beyond plain data transportation and interface adaption, such as thread-pool handling, mutexes, factories for one-to-many and many-to-one operating threads and their synchronization, runtime system state and type information.

This pattern is the most powerful form of creating a new computation object for the distributed system. It does however require knowledge about the design and behavior of the distribution services. Because the wrapper classes provide the CORBA interface to the other traditional subsystems of the framework, a distributed or parallel implementation of a subsystem can easily access them directly.

A good example is a class that performs distributed lighting computation using the PhotonMap algorithms [Jen96] (see Figure 4 shows our implementation). We reuse existing code for tracing of photons from the light sources and for reconstructing illumination information. Both reused object implementations are wrapped with the patterns described above. Because the algorithm is aware of its distributed or parallel nature, it can steer and adapt to the computational requirements, e.g. by adding new particle tracer threads on a multi-processor machine or adding new instances of distributed objects. This scheme allows the programmer to gradually make selected subsystems aware of the distribution infrastructure without compromising the remaining system on the way.

The possibility of reusing existing classes simplifies the creation of new distributed subsystems in a straightforward building-block manner. However, a drawback of this approach is the dedication to distributed computing, making the new subsystem more difficult to use when running the application in a serial, single-threaded fashion.

2.4 Discussion

The patterns introduced above offer several benefits:

- New developments within the traditional framework are immediately distributable through the wrapper pattern, which offers speedup through replication and multiplexing.
- There is no need for developers of algorithms to bother with distribution and parallelization issues because the distribution framework does not alter or interfere with the application logic.
- The distribution and parallelization services offered by the framework provide the developer of advanced computation classes with basic functionality that is guaranteed to conform to the overall design.

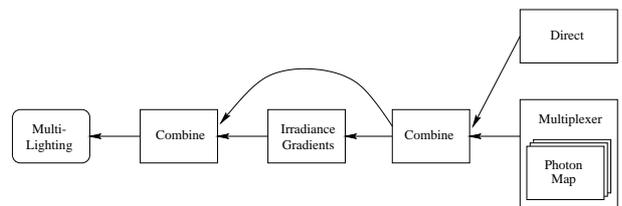


Figure 5: Logical data flow within an example distributed lighting network performing direct, indirect, and caustic illumination through different LightOps, some of which are replicated and use a multiplexer for speed-up.

- The learning effort for beginners can be reduced dramatically by a transparent distribution infrastructure — in particular if compared to other distribution frameworks and the large number of new software concepts introduced by them.
- Our distribution framework transparently supports modularization and helps to structure the framework into toolkits with well defined interfaces. This can help to reduce the overall programming effort, and promotes a better understanding of the big picture.

For each of the above pattern, there is a typical case of application. Like a modular object-oriented program can be viewed at various levels of granularity, the patterns support this building-block design strategy. Because the distribution infrastructure uses consistent interfaces, the patterns can be combined with each other or be applied to traditional class implementations by a configuration script. Especially for research and development purposes, this offers a tremendous flexibility. Note, that the multiplexer can be used to easily handle a new parallel implementation of a computation class, which in turn can be constructed using wrappers, other distributed classes, or multiplexers.

3 IMPLEMENTATION

The Vision rendering architecture [SS95] is an object-oriented system for physically-based realistic image synthesis. The Lighting Network [SSH⁺98, SSS98] technology within the Vision framework provides an object-oriented way of dealing with functional decomposition for lighting calculations. It implements the lighting subsystem for Vision by decomposing the global illumination computations into a set of lighting operators that each perform a partial lighting simulation. Conceptually, these “LightOps” take a representation of the light distribution in the environment as input and generate a new representation as output. By connecting these LightOps in the right way, the lighting simulation can be configured flexibly by simulating any light-paths in a multi-pass fashion [CRMT91].

The Lighting Network acts as a data flow network much in the spirit of AVS [UFK⁺89] or similar systems. Figure 5 shows an example of a very simple distributed Lighting Network. It uses two basic LightOps to perform direct lighting, adds their individual contributions, and then performs indirect lighting computations. The result is the sum of the direct and the indirect illumination (also see Figure 8). Direct illumination from light sources is obtained through ray-tracing, the PhotonMap algorithm [Jen96] computes caustic light paths, and indirect illumination is computed with the irradiance gradients algorithm [WH92]. The whole lighting network is managed by a special object called MultiLighting that implements the lighting subsystem interface towards other Vi-

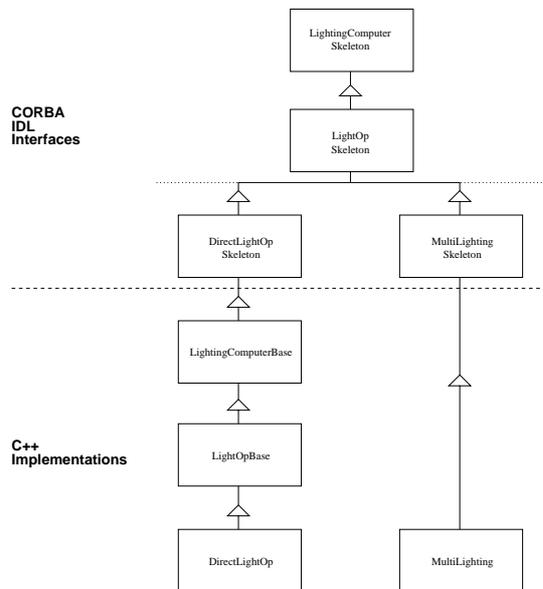


Figure 6: Multiple layers of abstract interface declarations are complemented by C++ definitions, to give consistent interfaces to all components of the lighting subsystem.

sion subsystems and behaving according to the facade design pattern [GHJV95].

The Renderer subsystem of the Vision framework encapsulates various screen sampling techniques. It computes intersections with visible objects of the scene and queries the lighting subsystem for the incident illumination at that point. This illustrates the clear separation of independent computation within the Vision rendering framework.

We have applied the presented distribution framework to the Rendering and Lighting Network subsystem in Vision in that we allow individual Renderer and LightOp objects to be distributed across a network or to be run in parallel through the use of threads. Figure 6 shows the inheritance relations between the interfaces of the LightOps and the MultiLighting facade. The asynchronous communication patterns and services are implemented within the C++ base classes. Note that for wrapping traditional code, the C++ class on the lower left is a pseudo-polymorphic wrapper template¹, which requires no manual coding.

Figure 7 shows a running distributed Vision system. Note that hosts 1 and 2 contain multiple concurrent LightOps within a lighting network. They should therefore have multiple processors to enable functional parallelism.

The basic operating system functions are accessed via the portable operating system adaption layer interface of the ACE library [Sch94]. The communication and remote object creation is done using the CORBA implementation VisiBroker of Inprise Corp. [VG98]. To facilitate further development and maintenance, the design of the base classes follows the guidelines of several design patterns [GHJV95, CS98, LS96, SHP97, McK95].

¹The external polymorphism pattern [CS98] allows treating non-polymorphic classes as if they have the proper inheritance relationship, by providing all classes with a method that simply delegates the calls to a sufficiently global template signature adapter (that's why it's called external) who in turn calls the method that performs the task.

4 RESULTS

This section demonstrates the flexibility of the presented distribution and parallelization framework as applied to the Vision rendering system. Several distributed LightOps have been implemented using the design patterns from Section 2 and we discuss some of their typical configurations. In order to reuse the traditional LightOp implementations efficiently, several multiplexer classes are available along with different scheduling strategies. This allows building distributed lighting networks, that functionally distribute lighting calculations. The configuration of the distributed objects is usually specified in a TCL configuration file using the existing scripting engine of the traditional Vision system, avoiding the introduction of a second tier of abstraction for configuring the distributed system (compare [Phi99]).

4.1 Efficiency of Asynchronous Communication

In the first example, we show the benefits of the asynchronous communication pattern used throughout the CORBA implementation of the base classes at the heart of the distribution infrastructure. Table 1 compares the packeted data transfer within a small lighting network using asynchronous requests with an equivalent network using the original interface with fine granularity. Both cases use wrapped traditional LightOps and the same host configuration:

SGI	Onyx	Onyx	O2
# processors	4	2	1
R10k @ MHz	196	195	195
Renderer			×
Lighting	Irr. Grad.	Direct	Combine

The main reason for the speedup of 33% is the low number of 210 CORBA method calls to transfer requests over the 100 MBit/s network in the case of asynchronous communication, compared to 128,070 synchronous invocations in the second case. Both networks transfer identical 22.7 MB of request data through CORBA marshaling. It is the synchronous protocol of CORBA that blocks the client until the server has completed the method call which is responsible for the poor performance in the second case. This shows clearly the important fact, that latency can be almost entirely hidden using the asynchronous protocol provided by our distribution base classes.

4.2 Distributed Rendering

To optimize rendering times in the case of calculating previews or testing new computation class implementations, we pick up the example from Section 2.2 (see Figure 3). The following configuration of a distributed Vision system shows the best achievable speedup we have found using our framework. It uses 4 hosts with a total of 8 processors. There are 8 ray-tracers to work in data-parallel mode and 6 lighting modules. Each group is controlled by a multiplexer. The distribution framework ensures that all communication between the two multiplexers is done asynchronously.

SGI	Onyx	Onyx	O2	O2
# processors	4	2	1	1
R10k @ MHz	196	195	195	195
Renderer	4	2	1	1
Lighting	4	2	-	-

The lighting hosts execute a traditional implementation of a Irradiance Gradients [WH92] LightOp, which is wrapped for distribution. Additionally, the wrappers on the multiprocessing machines also include a multiplexer that executes the incoming requests in parallel using a thread pool. Because there are multiple threads per

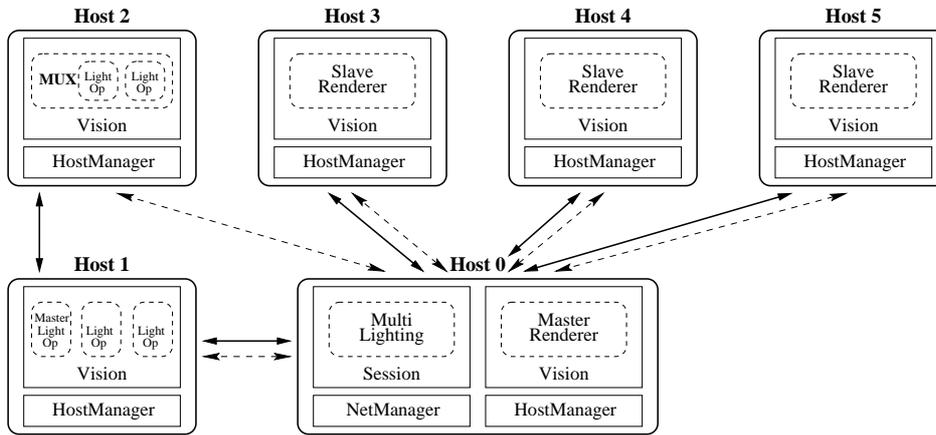


Figure 7: Example of a running distributed Vision system. The master renderer controls the data-parallel activity of the slave renderers on hosts 3, 4 and 5. The MultiLighting on host 0 is the facade of the lighting subsystem, which is a lighting network residing on hosts 1 and 2 (it can also be further distributed as shown in later examples). Its entry point is the MasterLightOp, which controls the other LightOps. Note that this functional parallelization also communicates asynchronously in a pipeline fashion (indicated by the solid arrows), enabling parallel execution if a host has multiple processors. A single NetManager and a HostManager on each host are responsible for bootstrapping the system onto the network by providing initial object factory methods (dashed arrows).

CPU, the multiplexer synchronizes them in order not to overload the machine. Configuring this system, required just to name the hosts and the LightOp with its parameters in a configuration file. The TCL scripts for system setup take care of distributing the objects using the Net- and HostManager of Figure 7. This distributed system is compared to the traditional Vision system with a single thread of control, running on the fastest machine in a single address space and calculating lighting with the very same LightOp implementation.

As Table 2 shows, the speedup obtained is near the theoretical maximum of 12.5%. The overhead of ≈ 90 seconds consists of 30 seconds session setup, 5 seconds of additional parsing on the CORBA startup client and another 5 seconds delay for allowing the hosts to clean up the CORBA objects before the main CORBA startup client shuts down all Vision instances. After subtracting this overhead, we obtain a penalty of $\approx 13\%$ during the rendering phase for the distributed system. We believe this is a very good result, given such a general and unintrusive distribution infrastructure.

4.3 Distributing Complex Lighting Computations

The functional decomposition of a lighting network offers the biggest potential for distribution and parallelization, at the risk of high communication costs. As shown in Section 4.1, the asynchronous request-callback communication paradigm is able to provide a partial solution for that problem. In the following example we make heavy use of the patterns from Sections 2.3 and 2.1. This configuration uses 3 hosts with a total of 7 processors:

SGI	# proc. R10k	@ MHz	Renderer	Lighting
Onyx	196	4		Photon Map, Direct, Combine
Onyx	195	2	×	Photon Map, Irrad. Grad.
Octane	175	1	×	Photon Map

In this setup, the reconstruction method of the Photon Map LightOp takes much more time to process a request, than any of the other LightOps in the lighting network. Consequently, a multiplexer is used to distribute this LightOp onto 3 hosts. In contrast,

the three other LightOps are executed on multi-processor machines, because their reconstruction method is fast and the communication between them can be optimized, if the CORBA implementation supports object collocation. In order to drive this complex lighting subsystem, two hosts execute rendering objects controlled by a multiplexer in a data-parallel way.

As one can see from Table 3, the speedup obtained by this setup is not as good as in the first example. But even the advantage of the non-distributed version of running in a single address space, does not outweigh the communication overhead of the distributed system. Our profiling shows that the performance difference to the theoretical maximum of 14.3% is mainly due to process idle times. This occurs for example, if the calculation of one upstream LightOp is sufficiently delayed. Since the underlying Lighting Network is entirely pull-driven, the pipeline is blocked. We try to cope with that problem to some extent by allowing the asynchronous interface to drive three parallel streams at a time. Additionally, the resource handling within the base classes allows running the rendering computation concurrently with a lighting computation, resulting in a kind of interleaving CPU-usage scheme, if the lighting pipeline on the host is stall.

This example shows that there are cases where the full transparency of the distribution infrastructure cannot hide inherent limitations due to coarse grained communication patterns of existing subsystems. Note however, that this behavior is mostly a problem of the non-distribution aware algorithms of the lighting network and not so much a general drawback of the distribution framework. However, even with the very limited success, we still get some speed-up without any change to the application logic.

Apart from that, one has also to take into account, that while a traditional system performs quite well in this case in terms of execution speed, it is severely limited by the host's memory resources. Especially the PhotonMap LightOp needs to store many photons that have been shot into the scene when working with large scene descriptions. The distributed PhotonMap LightOps in this example have the memory of three hosts to their disposition. Furthermore, the initial shooting of particles is done in parallel, reducing the Lighting setup time needed to one seventh (there are 7 processors on the three hosts), which is of great value when simulating high quality caustics.

<i>wallclock seconds for</i>		asynchronous LightOps	wrapped-only LightOps
Session Setup		22.26	23.37
Parsing Scene		5.80	5.67
Lighting Setup		1.56	1.68
Renderer Setup		0.30	0.34
Render Frame		1,922.06	2,916.95
<i>Total</i>		1,977.36 66 %	2,974.92 100 %

Table 1: Packeted asynchronous data transfer within a lighting network compared to LightOps using CORBA’s synchronous request invocation.

<i>wallclock seconds for</i>		distributed System	traditional Vision
Session Setup		31.91	-
Parsing Scene		5.61	-
Lighting Setup		0.14	-
Renderer Setup		0.36	-
Render Frame		317.03	2,359.20
<i>Total</i>		387.41 16 %	2,380.15 100 %

Table 2: A distributed system using two multiplexers, controlling the data-parallel renderers and the lighting objects on the left side, are compared to the traditional single-threaded system.

Although there certainly is a price to pay for the flexibility of our distribution strategy, we obtain high degrees of freedom in configuring the distributed system and adapting it to the challenges of a specific lighting network.

5 CONCLUSIONS

We presented a general approach on providing a transparent infrastructure for distributing object-oriented systems and used this infrastructure for the distribution and parallelization of rendering and lighting computations. While we created several design patterns to hide CORBA and the distribution infrastructure from the average system programmer, our system provides distribution services to the advanced application programmer and still offers access to all basic distribution classes for sophisticated tuning if necessary.

The use of the CORBA middleware allowed us to abstract from much of the underlying communication infrastructure. Contrary to popular believe, the runtime overhead of using CORBA has been minimal. However, the synchronous nature of CORBA messages was a major problem that we had to work around using a non-trivial request-callback scheme based on multi-threading. Here, the addition of asynchronous messaging to CORBA should help tremendously.

The implementation of distribution functionality within a few base classes makes distribution issues totally transparent to an application programmer. We demonstrated the approach with examples for the Vision rendering framework, to which it provides transparent data-parallelism and distribution of the existing object structure. Developers of new computation classes are free to use the distribution infrastructure to add distribution aware modules or to wrap existing algorithms and distribute them across a network of computers.

The distribution infrastructure has proven to be practical and stable. It offers well-defined interfaces without imposing any limitations on the remaining parts of the Vision system. Distributed lighting networks simply can be constructed and configured by scripts

that specify the location and parameterization of specific modules in a network. Figure 8 gives an impression, of how the flexible structure allows the configuration of the whole distributed system for different purposes, ranging from speeding up preview renderings to experimenting with complex lighting networks consisting of many different distributed lighting simulation algorithms.

Future work on the distribution infrastructure will concentrate on recovering some of the efficiency that we lost in the process. In particular, it would be useful if the system would take care of the distribution of modules across a network automatically and perform better load-balancing. However, due to the dynamic nature of our application, this requires some knowledge about the computational characteristics of the different modules. Making this available at the wrapping level or during run-time would allow us to statically allocate and maybe dynamically move modules across a network.

6 ACKNOWLEDGEMENTS

We would like to thank Thomas Peuker who developed the initial idea of this distribution framework. Also Marc Stamminger provided considerable support for integrating the new scheme into the existing Vision system. We would like to thank the anonymous reviewers who helped to improve the initial version of this document.

References

- [CR98] A. G. Chalmers and E. Reinhard. Parallel and distributed photo-realistic rendering. In *SIGGRAPH Course Notes – Course 3*, July 1998.
- [CRMT91] Shenchang Eric Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner. A progressive multi-pass method for global illumination. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):165–174, July 1991.
- [Cro98] Thomas W. Crockett. Parallel rendering. In *SIGGRAPH '98 "Parallel Graphics and Visualization Technology" course #42 notes*, pages 157–207. ACM, July 1998.

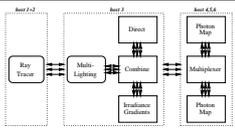
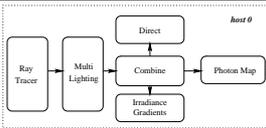
wallclock seconds for		distributed System		traditional Vision
Session Setup		79.14		-
Parsing Scene		10.43		-
Lighting Setup		4.58		-
Renderer Setup		0.28		-
Render Frame		1,498.39		6,988.45
Total		1,665.28		8,174.22
		20 %		100 %

Table 3: On the left, the lighting network is distributed among 3 hosts. On the right, the same computations are done with the traditional single-threaded system.

- [CS98] Chris Cleeland and Douglas C. Schmidt. External Polymorphism — An Object Structural Pattern for Transparently Extending C++ Concrete Data Types. *C++ Report Magazine*, September 1998. Also available at <http://www.cs.wustl.edu/~schmidt/C++-EP.ps.gz>.
- [FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.
- [GBD⁺94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine; A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, first edition, 1995.
- [GLFK98] Andrew S. Grimshaw, Michael J. Lewis, Adam J. Ferrari, and John F. Karpovich. Architectural support for extensibility and autonomy in wide-area distributed object systems. Technical Report CS-98-12, University of Virginia, June 1998.
- [GLS94] William Gropp, Ewing Lusk, and Anthony Skjelum. *Using MPI – Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.
- [HA97] Alan Heirich and James Arvo. Parallel rendering with an actor model. In Fhrad Arbab and Philipp Slusallek, editors, *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, pages 115–125, September 1997.
- [Jen96] Henrik Wann Jensen. Global illumination using photon maps. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996.
- [Jez93] J.-M. Jezequel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.
- [LS96] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In James O. Coplien, John Vlissides, and Norm Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA, 1996. Also available at <http://www.cs.wustl.edu/~schmidt/ActiveObjects.ps.gz>.
- [MA] Jane F. Macfarlane and Rob Armstrong. POET: A parallel object-oriented environment and toolkit for enabling high-performance scientific computing. Also available at <http://omega.lbl.gov/poet/Poet.html>.
- [MB98] D. Meneveaux and K. Bouatouch. Parallel hierarchical radiosity for complex building interiors. Rapport de Recherche 3425, INRIA, May 1998.
- [McK95] Paul E. McKenney. Selecting locking primitives for parallel programs. Technical report, Sequent Computer Systems, Inc., 1995. Also available at <http://c2.com/ppr/mutex/mutexpat.html>.
- [OMG97] Object Management Group. *OMG Home Page*, 1997. Available at <http://www.omg.org/>.
- [OMG98a] Object Management Group. *CORBA/IIOP 2.2 Specification*, 2.2 edition, 1998. OMG Document 97-07-01.
- [OMG98b] OMG. *CORBA Messaging*, omg tc document orbos/98-05-05 edition, May 1998.
- [Phi99] Philippe Merle, Université des Sciences et Technologies de Lille. *The CorbaScript Language*, 1999. Available at <http://corbaweb.lifl.fr/CorbaScript/>.
- [RCJ98] E. Reinhard, A. G. Chalmers, and F. W. Jansen. Overview of parallel photo-realistic graphics. In *EUROGRAPHICS'98 – State-of-the-Art Reports*, pages 1–25, August 1998.
- [Sch94] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the 12th Annual Sun Users Group Conference*, pages 214–225, San Francisco, CA, June 1994. SUG. Also available at <http://www.cs.wustl.edu/~schmidt/SUG-94.ps.gz>.
- [SHP97] Douglas C. Schmidt, Timothy H. Harrison, and Nat Pryce. Thread-specific storage for C/C++. In *Pattern Languages of Programming '97 conference proceedings*, September 1997. Also available at <http://www.cs.wustl.edu/~schmidt/TSS-pattern.ps.gz>.
- [SS95] Philipp Slusallek and Hans-Peter Seidel. Vision: An architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1:77–96, 1995.
- [SSH⁺98] Philipp Slusallek, Marc Stamminger, Wolfgang Heidrich, Jan-Christian Popp, and Hans-Peter Seidel. Composite lighting simulations with lighting network. *IEEE Computer Graphics and Applications*, 18(2):22–31, March/April 1998.
- [SSS98] Philipp Slusallek, Marc Stamminger, and Hans-Peter Seidel. Lighting networks — A new approach for designing lighting algorithms. In *Graphics Interface*, pages 17–25, June 1998.
- [SV96] Douglas C. Schmidt and Steve Vinoski. Distributed callbacks and decoupled communication in CORBA. *SIGS C++ Report*, October 1996. Also available at <http://www.cs.wustl.edu/~schmidt/C++report-col8.ps.gz>.
- [UFK⁺89] Craig Upson, Thomas A. Faulhaber, Jr. David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, July 1989.
- [VG98] Inprise, Corp. *Visigenic Products*, 1998. Available at <http://www.inprise.com/visibroker>.
- [WH92] Gregory J. Ward and Paul Heckbert. Irradiance gradients. *Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.

Distributed Lighting Network

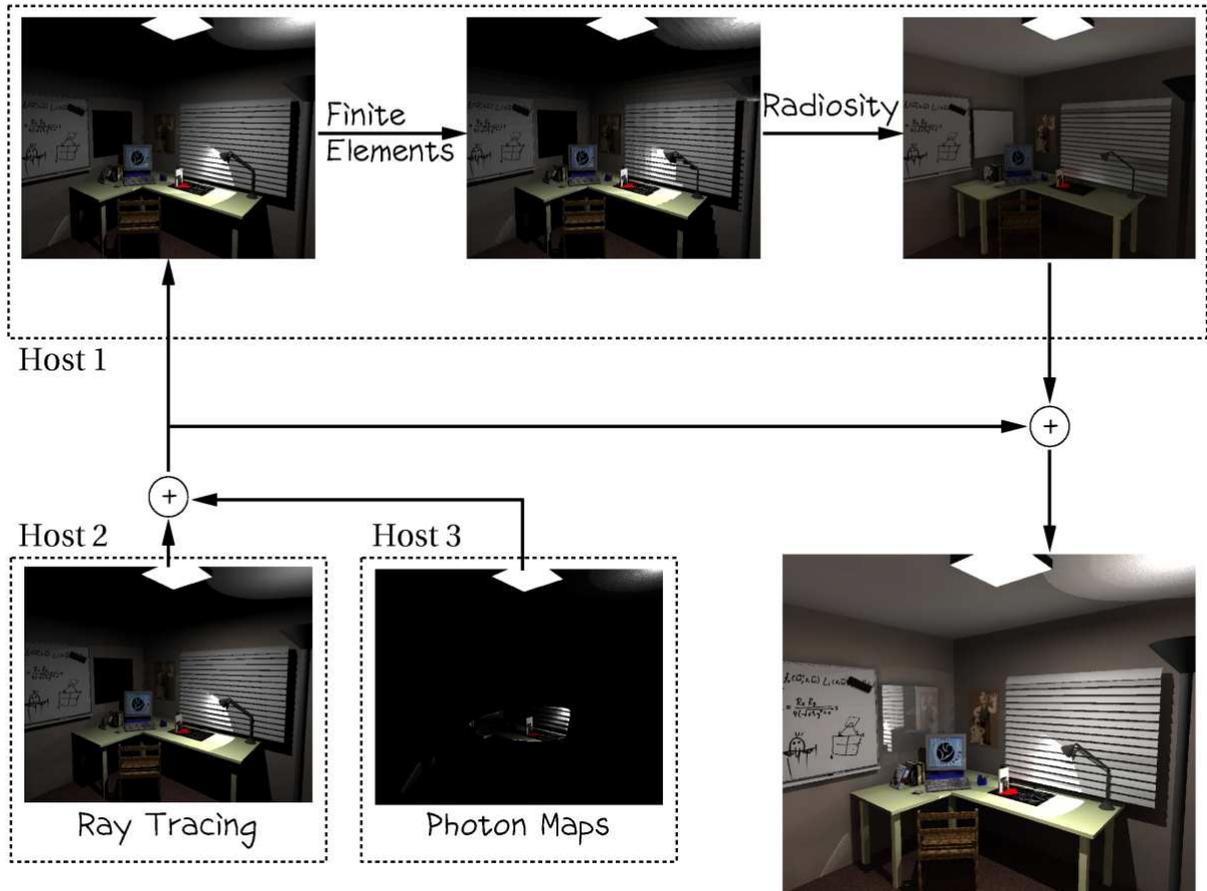


Figure 8: Data flow and intermediate results computed by a distributed lighting network performing direct, indirect, and caustic illumination through different LightOps.