ParoC++: A Requirement-driven Parallel Object-oriented Programming Language

Tuan-Anh Nguyen, Pierre Kuonen

University of Applied Sciences Western Switzerland, EIA-FR
tuananh.nguyen@epfl.ch, pierre.kuonen@eif.ch

Abstract. Adaptive utilization of resources in a highly heterogeneous computational environment such as the Grid is a difficult question. In this paper, we address an object-oriented approach to the solution using requirement-driven parallel objects. Each parallel object is a self-described, shareable and passive object that resides in a separate memory address space. The allocation of the parallel object is driven by the constraints on the resource on which the object will live. A new parallel programming paradigm is presented in the context of ParoC++ - a new parallel object-oriented programming environment for high performance distributed computing. ParoC++ extends C++ for supporting requirement-driven parallel objects and a runtime system that provides services to run ParoC++ programs in distributed environments. An industrial application on real-time image processing is used as a test case to the system. The experimental results show that the ParoC++ model is efficient and scalable and that it makes easier to adapt parallel applications to dynamic environments.

1 Introduction

The emerging of computational grid [1, 2] and the rapid growth of the Internet technology have created new challenges for application programmers and system developers. Special purpose massively parallel systems are being replaced by loosely coupled or distributed general-purpose multiprocessor systems with high-speed network connections. Due to the natural difficulty of the new distributed environment, the methodology and the programming tools that have been used before need to be rethought.

While traditional distributed HPC applications usually view the performance as a function of processors and network resources, we will address the question: How to tailor the application with a desired performance to the distributed computational environment.

We developed an object-oriented model that enables the user to express high-level resource requirements for each object. This model is implemented in a parallel object-oriented programming system for HPC called ParoC++. ParoC++ is a programming language and a runtime system. We did not try to create a new language but we extended C++ to support our model. The runtime system

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2657, pp. 165–174, 2003.
© Springer-Verlag Berlin Heidelberg 2003

of ParoC++ is responsible for managing and monitoring distributed computational environment and is partially written using ParoC++ itself. The current prototype runtime system supports the ability to map an arbitrary object onto a resource in a heterogeneous environment. We have modelled a wide area environment as a dynamic graph of resources. The resource discovery process during parallel object allocation takes place on this graph by mechanism of request matching and forwarding.

In ParoC++, the user does not directly deal with processes. Instead, he handles the so-called "parallel objects" which encapsulate processes. A parallel object is a self-described object that specifies its resource requirements during the lifetime. Parallel objects can be computational objects, data objects or both. Each parallel object resides in a separate memory address space. Similar to CORBA, parallel objects are passive objects that communicate via method invocations. The selection of resource for a parallel object is driven by the object requirement and is transparent to the user.

This paper focuses on the programming language aspect of the ParoC++ and the requirement-driven parallel object. In section 2, we will explain our requirement-driven parallel object model. Parallel object is the central concept in ParoC++ which we describe in section 3. We also present in this section some experimental results on low-level performance of ParoC++. Next, we demonstrate using ParoC++ in an industrial real-time application in the field of image processing in section 4. Some related works are discussed in section 5 before the conclusions in section 6.

2 Requirement-driven parallel object

2.1 A parallel object model

We envision parallel object as the generalization of the traditional object such as in C++. One important support for parallelism is the transparent creation of parallel objects by dynamic assignments of suitable resources to objects. Another support is various mechanisms of method concurrency: parallel, sequential and mutex.

A parallel object, in our definition, has all properties of a traditional object plus the following ones:

- Parallel objects are shareable. This property is described in section 2.2.
- Parallel objects support various method invocation semantics: synchronous, asynchronous, sequential, mutex and concurrent. These semantics are discussed in section 2.3.
- Objects can be located on remote resources and in a separate address space. Parallel objects allocations are transparent to the user. The object allocation is presented in section 2.4.
- Each parallel object has the ability to dynamically describe its resource requirement during its lifetime. This feature will be discussed in detail in the section 2.5.

It has to be mentioned that as normal objects, parallel objects are passive objects that can only go into active mode upon executing a method invocation request. We believe that using the passive object is easier and more familiar to the traditional object-oriented programming paradigm. The passive object allows the user to fully control object execution, thus allowing a better integration into other software components and making the maintenance of components simple.

2.2 Shareable parallel objects

All parallel objects are shareable. Shared objects with encapsulated data provide a means for the user to implement global data sharing in distributed environments. Shared objects can be useful in many cases. For example, computational parallel objects can synthesize the output data simultaneously and automatically into a shared output parallel object.

2.3 Invocation semantics

Syntactically, invocations on parallel objects are identical to invocations on traditional sequential objects. However, the parallel object invocation supports various semantics. The semantics are defined by two parameters:
1. Interface semantics:

- Synchronous invocation: the caller waits until the execution of the requested method on the server side is finished and returned the results. This corresponds to the traditional way to invoke methods.
- Asynchronous invocation: the invocation return immediately after sending the request to the remote object. The results of the invocation can be actively returned to the caller object if the callee knows the "call back" interface of the caller.

2. Object-side semantics:

- Sequential invocation: the invocation is executed sequentially. The serializable consistency of sequentional invocations is guaranteed.
- Mutex invocation: the invocation request is blocked until all instances of methods are terminated.
- Concurrent invocation: the execution of method occurs in a new thread (multithreading) if no sequential or mutex method is currently invoked.

All invocation semantics are specified during the design phase of parallel objects.

2.4 Parallel object allocation

The allocation of parallel object is transparent to users. It consists of two phases: first, the system need to find a resource where the object will live; then the object code is transmitted and executed on that resource, the interface is connected to the object.

2.5 Requirement-driven parallel objects

We believe that the high performance in highly heterogeneous and dynamic environments such as the Grid can only be obtained if the two following conditions are satisfied:

- The application should be able to adapt to the environment.
- The programming environment should somehow enable application components to describe their resource requirements.

The first condition can be fulfilled by multi-level parallelism, dynamic utilization of resources or adaptive task size partitioning. One solution is to dynamically create parallel objects on demand that will be expressed in section 3 where we describe the ParoC++.

In the second condition, the requirements can be expressed in form of quality of services that components desire from the environment. Number of researches on the quality of service (QoS) has been performed [3–5]. Most of them consist in some low-level specific services such as network bandwidth reservation, real-time scheduling, etc.

Our approach integrates the user requirements into parallel objects in the form of high-level resource descriptions. Each parallel object is associated with an *object description* (OD) that depicts the needed resources to execute the object. The resource requirements in OD are expressed in terms of:

- Resource name (host name) (low level).
- The maximum computing power that the object needs (e.g. the number of Mflops needed).
- The amount of memory that the parallel object consumes.
- The communication bandwidth with its interfaces.

An OD can be `"power>= 150MFlops :100MFlops; memory=128MB"` which means the object requires a power of 150MFlops although 100MFlops is acceptable (non-strict item) and a memory storage of at least 128MB (strict item).

The construction of OD occurs during the parallel object creation. The user can initiate the OD for each object constructor. The OD can be parameterized by the input parameters of the constructor.

It can occur that, due to some changes on the object data or some increase of computation demand, the OD needs to be re-adjusted. If the new requirement exceeds some threshold, the adjustment can request for object migration. The migration process should be handled by the system transparently to the user.

3 ParoC++ programming language

ParoC++ is an extension of C++ that supports parallel objects. We try to keep this extension as close to C++ as possible so that the programmer can easily learn ParoC++ and the existing C++ libraries can be parallelized using ParoC++ without too much effort.

We claim that *all C++ classes with the following restrictions can be implemented as parallel object classes without any changes in semantic*:

– All data attributes of object are protected or private
– The object does not access any global variable
– There is no user-defined operator
– There is no method that returns the memory address references

In other word, to some extension, ParoC++ is a *superset* of C++. This is important if we want to construct mechanisms for coarse-grain auto-parallelism. In many case, the compiler can efficiently decide among objects which ones are parallel objects and which ones are sequential objects and thus automatically generates the codes for each kind of object. Auto-parallelism is not yet implemented in ParoC++.

In this section, we will refer our *parallel object* as *object*.

3.1 ParoC++ parallel class

Developing ParoC++ programs mainly consist of designing parallel classes. The declaration of a parallel class begins with the keyword **parclass** following the class name:

```
parclass myclass {...};
```

As sequential classes, parallel classes contain methods and attributes. Method accesses can be public, protected or private while attribute accesses must be protected or private. For each method, the user should define the invocation semantics by using ParoC++ keywords: **async**, **sync**, **conc**, **seq** and **mutex** for asynchronous, synchronous, concurrent, sequential and mutex invocations. These semantics are described in section 2.3.

The combination of the interface and object-side semantics defines the overall semantics of a method. For instance, the following declaration defines an asynchronous concurrent method that returns an integer number:

```
async conc int mymethod();
```

Two important properties of object-oriented programming: multiple inheritance and polymorphism are supported in ParoC++.

3.2 Object description

Object description (OD) is declared along with object constructor statement. Each constructor of a object associates with an OD that resides right after the argument declaration between "@{...}". An OD contains a set of expressions on the reserved keywords **power** (for the computing power), **network** (for the communication bandwidth between the object server and the interface), **memory** (for the memory) and **host** (user-specified resource). Each expression is separated by a semi-colon (";") and has the following format:

`[power | memory | network]` [>= | =] <number expression 1> [":" number expression 2];

or `host` = [string expression];

The existence of *host* expression will make all other expressions be ignored.

3.3 Parallel object creation and destruction

Syntactically, the creation and the destruction of a parallel object are identical to those of C++. The object creation process consists of locating a resource satisfying the OD, transmitting object code, remote executing object code, establishing communication, transmitting arguments and invoking the object constructor. Failures on object creation will raise an exception to the caller.

Each object has a counter that defines the number of reference to the object. A counter value of 0 makes the object be physically destroyed. The object counter is managed by the ParoC++ run-time system transparently to the user.

3.4 Inter-object communication: method invocation

The conventional way to communicate between distributed components in ParoC++ is through method invocations. The semantic of invocations is fixed during the class declaration.

The current prototype of ParoC++ implements the communication using TCP/IP socket and Sun XDR as its data representation. All data transmitted over the network conforms to XDR format.

3.5 Intra-object communication: shared data vs. event sub-system

In an object, there are two ways for concurrent operations to communicate: using *shared data attributes* or via *the event sub-system*. Using shared attributes is simple but it requires the programmer to manually verify and synchronize the data access.

Another method is communication via event sub-system. Each object has its own event queue. An event is a positive integer whose semantic is application dependent. An object can raise (**eventraise(n)**) or can wait (**eventwait(n)**) for an event "n" in its own queue. Raising an event in one object will not affect the waiting-for-event in other objects.

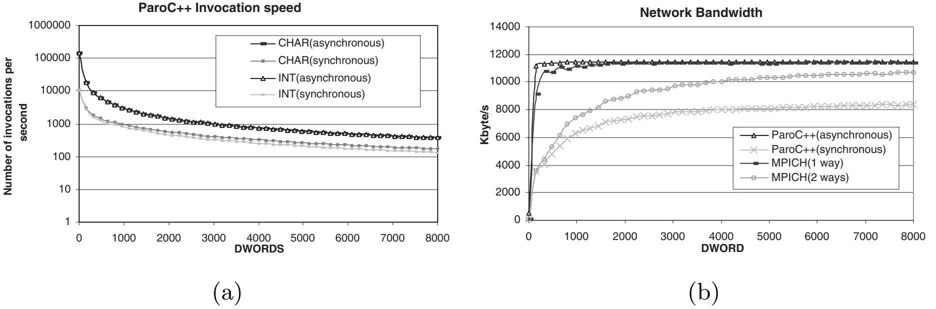
Event sub-system is a very powerful feature to deal with signalling and synchronization problems in distributed environments. For instance, it can be used in conjunction with the shared data attributes to notify the status of data during the concurrent invocations of read/write operations. It can also be used to tell the others about occurrence of failure or the changes in the environment.

3.6 Mutual exclusive execution

When concurrent invocations occur, some parts of executions might access an attribute concurrently. To deal with these situations, it is necessary to provide a mutual exclusive mechanism. ParoC++ supports this feature by using the keyword **mutex**. Inside a given object, all block of codes starting with the keyword **mutex** will be executed mutual exclusively.

3.7 Communication cost

We wrote a program containing two objects called "Ping" and "Pong" running on two different machines. Ping invokes methods of Pong with different arguments (size and type) and with two different invocation semantics: synchronous and asynchronous. Invocation speed and the communication bandwidth are measured.



(a) (b)

Fig. 1. Parallel object communication cost

Figure 1(a) shows the invocation speed of objects on 8-bit and 32-bit integer messages. Asynchronous invocations are more efficient than synchronous ones, especially for small messages due to message aggregation. The latency of asynchronous invocation is about 6.9 μsec (MPICH: 43 μsec) and of synchronous one is about 94 μsec (MPICH: 123 μsec).

The communication bandwidth, in Fig. 1(b), shows that asynchronous invocations, due to the overlapping, utilize better bandwidth than synchronous invocations. This bandwidth is slightly better than asynchronous send (one way) of MPICH. The bandwidth of asynchronous calls almost reaches the limit of the Fast Ethernet throughput (11.3 MB/s). For synchronous invocation, MPICH achieves somehow better bandwidth in our experiment (15-20% better for large messages). This is due to the extra cost for multiplexing remote method in ParoC++.

4 Example application

We present in this section the development of Pattern and Defect Detection System (PDDS) using ParoC++. PDDS is part of the European project Forall-1[1] in textile manufacturing. The main function of PDDS is to analyze continuous tissue images to find pattern positions and to discover defects on the tissue. This process should be in real-time with the capacity of analysis up to 3.3 Mpixels/s or about 10MBytes/s for 24bits RGB images.

Figure 2 demonstrates the parallel object diagram of PDDS using ParoC++. The main program create two objects `ImageBuf` and `OuputData` and several objects `Analyzer`. `ImageBuf` and `OutputData` are shared among `Analyzer` objects.

[1] European project E!1955/CTI 5130.1 financed by Swiss Government in the Eureka program

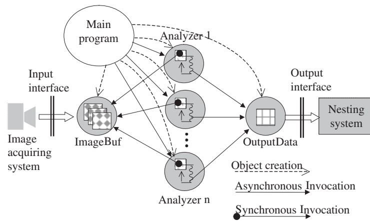


Fig. 2. ParoC++ implementation of PDDS

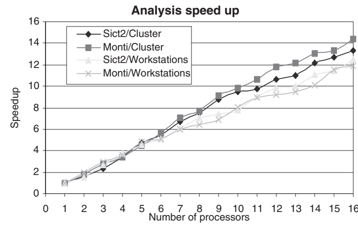


Fig. 3. Performance of PDDS/ParoC++

The `Analyzer` access `ImageBuf` to get the images, analyze them and then store the results in `OutputData`. The main program also plays the role of a monitoring agent. It monitors the real speed of `ImageBuf`. If the main program detects that the system overworks due to some increase on the computation demand or some external changes to the resources, it reacts by creating some more `Analyzer` objects (allocated more resources). Hence, in PDDS we also deal with the adaptation of the application to the user requirement and to the dynamic state of the environment.

We have performed two experiments. First, we run PDDS in homogeneous networks to measure the performance, the scalability and the efficiency in term of number of Analyzer objects. The second experiment is done in a heterogeneous network where we take into account the changes on the computation demand and on the environment.

The input image for the first experiment consists of 100 frames of size 2048x2048 pixels. `ImageBuf` splits the frame into several sub images of the size 512x512 pixels. No adaptation is considered in this test. Figure 3 shows the speedup of two types of tissues: small patterns (Sict2) and big patterns (Monti) on a network of Sun sparc workstations and on a cluster of Pentium 4. We see that in both environments, almost linear speedup is achieved. PDDS runs about 14 times faster on 16 processors.

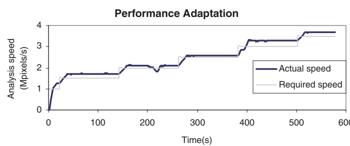


Fig. 4. Adaptation to the changes

In the second experiment, PDDS is launched in a heterogeneous environment of Solaris/sparc and Linux/Intel with the adaptation monitor turned on. Figure 4 shows the dependency between the analysis speed (in term of MPixel/s) and the time. The dash line presents the required power whereas the continuous line is the real power of PDDS. In the test, we dynamically change the requirement speed every 2 minutes. Due to these external changes, additional Analyzer objects (resources) are automatically allocated in order to satisfy the required performance. One interesting note is that at a certain time, the actual performance goes down (at the second of 220). The reason is that we have changed the load of a machine used by PDDS . The system reacts to this change and is soon recovered to the normal speed. By this experiment we want to show the two important points:

- ParoC++ application can efficiently deal with the computation on demand.
- ParoC++ can adaptively use the heterogeneous resources efficiently.

5 Related works

On the language aspect, Orca[6], MPL[7] and PO[8, 9] are some examples. Orca provides a new language based shared objects. The programming model that Orca used is Distributed Shared Memory (DSM)[1] for task parallelism. While Orca aims at using the objects as a mean to share data between processes, our approach combines the two concepts of shared data object and the process into a single parallel object.

MPL on the other hand, is an extension of C++ with some so-called metat classes for parallel execution. MPL follows the data-driven model. The parallelism is achieved by concurrent invocations on these objects. The Mentat runtime system is responsible for the instantiation of mentat objects, the invocation of method and keeping objects consistency. While the metat object supports only asynchronous invocation and is not shareable, ParoC++ provides a more general approach with various invocation types (synchronous, asynchronous, concurrent, sequential, mutex) and the capacity of sharing objects. Moreover, both Orca and MPL do not allow specifying the resource requirement within the object.

Our parallel object and PO both share the inter-object and intra-object parallelism. The difference is on the object model: PO follows *active object mode*[10] with capability of deciding when and which invocation requests to serve while our Parallel Object uses *passive object model* that is similar to C++. Abstract Configuration Language (ACL) in PO to specify high-level directives for the object allocation is similar to our Object Description (OD); however, the ACL directives are only expressed at the class-level and cannot be parameterized for specific instances whereas our OD deals directly with each object instance. Therefore, our OD can be customized based on the real input parameters of the object.

On the tool aspect, COBRA[11] and Parallel Data CORBA[12] extend CORBA by encapsulating several distributed components (object parts) within an object and by implementing the data parallelism based on data partitioning: data is automatically split and distributed to several object parts in difference memory address spaces. This differs from our approach in which each parallel object resides in a single memory address space and the parallelism is achieved by concurrent interaction of objects and concurrent invocations of methods on the same object. In addition, the specification of resource requirement is not defined in both Data Parallel CORBA and COBRA.

6 Conclusions

Adaptive utilization of the highly heterogeneous computational environment for high performance computing is a difficult question that we tried to answer in this paper. Such adaptation has two forms: or the application components should somehow decompose dynamically based on the available resources of the environment; or the components should allow the infrastructure to select suitable resources by providing descriptive information about the resource requirement.

We have addressed these two forms of adaptation by introducing our parallel object and ParoC++-a parallel object-oriented programming language. The integration of requirement driven by object-description into the shareable parallel object is a distinct feature or our approach. We have described ParoC++ that extends C++ to support the parallel object. ParoC++ also offers various mechanisms such as event sub-systems, synchronization, and mutual exclusive execution to support the concurrency within the parallel object. Programming in ParoC++ is rather easy since ParoC++ is very similar to C++.

Some primary experiments on ParoC++ have been performed. Low-level tests on different types of method invocations give a good latency and a good bandwidth compared to MPICH on the same architecture. An industrial application on real-time image analysis has also been demonstrated. The results have showed the efficiency, scalability, adaptability and the ease-to-use of ParoC++ in dealing with the computation on demand of the HPC application in heterogeneous and distributed environments.

References

1. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers (1998)
2. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. International J. Supercomputer Applications **15** (2001)
3. Foster, I., Roy, A., Sander, V.: A quality of service architecture that combines resource reservation and application adaptation. In: The 8th International Workshop on Quality of Service. (2000)
4. Hoo, G., Johnston, W., Foster, I., Roy, A.: Qos as middleware: Bandwidth reservation system design. In: Proc. of the 8th IEEE Symposium on High Performance Distributed Computing. (1999)
5. Gill, C., Kuhns, F., Schmidt, D.C., Cytron, R.: Empirical differences between cots middleware scheduling paradigms. In: The 8th IEEE Real-Time Technology and Applications Symposium. (2002)
6. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S.: Orca: A language for parallel programming of distributed systems. IEEE Transactions on Software Engineering **18** (1992) 190–205
7. Grimshaw, A., Ferrari, A., West, E. In: Parallel Programming Using C++. The MIT Press, Cambridge, Massachusetts (1996) 383–427
8. Corradi, A., Leonardi, L., Zambonelli, F.: Hpo: a programming environment for object-oriented metacomputing. In: Proc. of the 23rd EUROMICRO conference. (1997)
9. Corradi, A., Leonardi, L., Zambonelli, F.: Parallel object allocation via user-specified directives: A case study in traffic simulation. J. Parallel Computing (2001) 223–241
10. Chin, R., Chanson, S.: Distributed object-based programming system. ACM Computing Surveys **23** (1991)
11. Keahey, K., Gannon, D.: Pardis: A parallel approach to corba. In: The 6th IEEE International Symposium on High Performance Distributed Computing. (1997)
12. Priol, T., Rene, C.: Cobra: A corba-compliant programming environment for high-performance computing. In: Proc. of Europar'98, Southampton, UK (1998) 1114–1122