Parallel implementation of a simulation system

Ing. Petr Přikryl
Doc. Ing. Tomáš Hruška, CSc. (supervisor)
Department of Computer Science and Engineering,
Faculty of Electrical Engineering and Computer Science,
Technical University of Brno, Božetěchova 2, 612 66 Brno, Czech Republic
tel. +420-5-7575 218 e-mail: prikryl@dcse.fee.vutbr.cz

Abstract: *The article continues in the approach to parallel programming and to simulation of systems which was developed and presented in [7, 8, 11, 9] and which was summarized in [10]. It shows general problems which may appear during implementation of a queuing simulation system when a general model for parallel programming is used.*

Key Words: Parallel programing, coordination, parallel simulation, Linda, Beauty.

1 Introduction

The article [10] observed the problem of parallelism and simulation from several distinct viewpoints. It concluded that the parallelism must be seriously taken into account in very near future. The introductory section summarized the basic reasons for the orientation towards parallel computing in general. Next section discussed the automatic versus human approaches to the building of a parallel program, and the following section compared the approach of building the specialized distributed (parallel) applications or systems with the approach based on an abstract general model for parallel programming. It briefly showed how such a model could look like.

This article is focused on some details of parallel implementation of parts of a *simulation system* using *the general object-oriented model for parallel programming.* It puts both topics together. A possible parallel implementation of parts of the simulator will be described in terms of the programming model.

2 General parallel programming model

This section presents some features of the general *model* for parallel programming called *Beauty.* The related ideas were firstly published in the article [6]. It presented basic features of the *language* Beauty which was derived in several steps from Linda [2]. However, the original tuples and processes were totally replaced by objects. The language was later converted into more general (i.e. language independent) form, into the *model for parallel* *programming*. The model got the same name as the language[1]. The goal was to design abstract classes of objects which create a uniform environment for parallel programming while hiding details about the implementation environment.

Application and system objects. The model is based on communicating objects. Their activities have to be coordinated, and the coordination must be intuitive and easy from the programmer's viewpoint, yet preserving theoretical correctness. The requirements are fulfilled via special properties and behaviour of objects. Beauty is based on two kinds of objects: *system objects* create an abstract environment for application objects, and the *application objects* are used as basic building blocks of applications.

The application objects are designed by an application programmer and they *always* communicate with other application objects via system objects. They encapsulate only one thread of activity thus making their programming easier. The system objects implement the desired behaviour with no other implementation restrictions. They are tailored to the hardware and the architecture. They are written by the programmer who implements the model in a real environment.

Object Space. The fundamental system object in Beauty is called *Object Space* (OSp). It is responsible for creation and destruction of application objects and for their mutual communication. It hides the architecture—the communication environment—and makes it very abstract. The following table shows the methods of OSp.

Access	Operations							
direct	create	destroy	send	call	lock	unlock	replicate	collapse
indirect	icreate	idestroy	isend	icall	ilock	iunlock	?	?
associative	—	?	asend	acall	?	?	?	?
multi-assoc.	select	?	msend	mcall	—	—	?	?

The four lines in the table correspond with the four types of the access to the destination objects. Each object has its own, unique, architecture-independent identification. When the *direct*-access operation of OSp is called, the explicit identification of the application object is used to find its placement. The operations for creation of objects do return the identification instead of using it for searching for them. When the *indirect* access is used, the operation is passed to the intermediate object identified directly. For example, the explicit identification of a *group* object is used to perform the operation in parallel with all objects which belong to the group. The *associative* access to an object is based on searching for candidates. If the associative version of an operation is used, the explicit identification is replaced by the class of the target object and by the specification of the object method (with arguments). The method must be non-blocking, and it returns a boolean result interpreted as "I may be the candidate". If more than one candidate is fond by OSp, the arbitrary candidate is chosen. The *multiple-associative* access works similarly, but the desired operation (the method) is called in parallel for all destination candidates. Their quantity limits are prescribed by arguments.

[1]There are several reasons why the model is called Beauty. One of them is that the Linda, originally named after the porno star Linda Lovelace [4], has also the meaning of *beautiful* as the word in Spanish or Portuguese. This way I wanted to express my homage to the Linda language (which inspired me) and to its authors.

The operations in the first group—*create*, *destroy*, *send*—are non-blocking which means that the caller do not wait for the result. When the *call* operation is used, the caller (the application object) is blocked until it receives the result from the called application object. The operations *lock* and *unlock* are used to implement complex, custom, atomic operations. The locked objects can be accessed only by the object which locked them. The last mentioned operations *replicate* and *collapse* are used for hardware and architecture independent optimization of algorithms. They prescribe replication of the identified object to more computing nodes or collapsing of the copies into one object, respectively. They may be implemented as empty operations without the impact on function of the implemented task. Implementations of languages based on the Beauty model can also refuse to replicate the object under some circumstances, or—on the other hand—they may replicate it even if it was not prescribed.

The versions of operations marked by '—' are not allowed because they do not make sense or they would complicate the semantics or the implementation. The versions of operations marked by '?' are not decided yet — the design of Beauty is still opened.

Other system objects. It is expected that several classes of system objects (other than OSp) will be considered as *general*. They can be viewed as extensions of the OSp. For example, the system class *group* is expected to be used for indirect operations with a set of application objects. Other system-object classes may be created for some *special purpose*. The distributed event-queue object class may be one of the examples when we think about usage of the Beauty model for and efficient implementation of a queuing simulation system. The system objects other than OSp have to fall into the category of system objects because the quality of their implementation (architecture dependent) has impact on efficiency of their abstract behaviour. And also, their encapsulated algorithms may be tightly bound to algorithms and data structures of OSp.

3 Simulation of systems

Simulation of systems belongs to disciplines with high demands on computational resources. Parallelism is inherited probably to all simulated systems. Moreover, the simulated systems often show high level of parallelism. We can always find many components which implement parts of a simulated system and which can be evaluated in parallel. Both simulators and simulated systems are often very complex. If we want to manage the complexity, we have to use *hierarchy* of *abstractions* (see e.g. [1]).

A simulation system which uses the aggregated power of many processors can be built as a specialized application, like the distributed simulation system presented in [5]. Such systems can be tailored and optimized to get the maximum effect for the projected purpose in a specific computing environment. I am personally interested in the alternative approach of using the general model for parallel programming.

Using more abstract model for parallel programming may not lead to that much efficient solutions in one particular environment; however, it may be more attractive if the system is to be ported to *various* computing environments — in the range from single-processor computer to massively parallel computers. The key point is to use an *abstract model* for parallel programming. The problem is how the model should look like to meet our needs in all thinkable cases. Trade-offs are inevitable.

Pitfalls of sequential thinking. It is very natural in the simulation branch to build simulation models of systems as hierarchies of subsystems. What is also very natural, however, is to think in categories of sequential programming during the implementation. Even when the programmer wants to use a parallel environment, he/she is often mislead to the sequential thinking just because he/she is used to follow the stereotypes. Some parallel versions of the solution do not come to his/her mind simply because he/she is not used to think this way. Searching for operations that may be done in parallel inside typical sequential algorithms may not be the correct way. Parallel solutions may be completely different in principle.

Implementation of the event queue. This paragraph illustrates the problem on implementation of the event queue. The event queue represents a typical object of queuing simulation systems. It is used for storing of events related to model time instants — it models the order of real time instants. In sequential simulators, the event queue takes usually the form of the linked list where each item is related to certain model-time instant and it contains information about events which are to happen in the time. The simulation algorithm usually takes the lowest model-time from the event queue, and the events planned for the time are evaluated. The events for one time-instant are considered independent, and they may be evaluated in parallel. Still, the event-queue represents a bottleneck in the parallel implementation because it plays the role of a centralized element in the system. It is a single abstraction, hence it cannot be parallelized easily.

If we cannot get rid of the event queue, we have to preserve as much parallelism inside the object as possible. But the parallelism must be hidden, encapsulated, in one abstract object unless we can use several independent event queues for independent parts of the system. We possibly may create a hierarchy where each subsystem can be modelled separately, using its own, local event-queue. Nevertheless, the event queue always look centralized on its level of abstraction.

The event queue stores only events planned for certain time window. The lowest time is usually equal to the present model-time, and the greatest time is also somehow related to the present or past model time. It hardly happens that some events are very early planned for time which can never be reached in the model. Usually, the present model-time events cause planning of future events with delay within some limited interval. The question is whether we cannot evaluate also events planned for later-than-present time. We often can, but there is always a risk that the event will be canceled or invalidated by an event planned for the time between the current time and the time evaluated in advance. The technique using the approach is called *time warp*, and it requires an implementation of roll-back of the system time in the above described cases (see [3], e.g.).

The operations with the event queue should be tightly bound to the *abstraction* represented by the event-queue interface-object. We should think about the event queue as about the central object responsible for coordination, but it need not to be necessarily an object with the only thread of activity. The opposite is the truth. We should implement it as one abstraction, but it must be parallel internally. In other words, the event-queue interface object must be able to serve more than one request in parallel, and then the requests must be dispatched to its internal objects.

In the terms of Beauty, the event-queue interface object have to be implemented as the system object, or it may be the application object which uses non-blocking operation to dispatch the requests to other event-queue related objects. The internal event-queue objects are not accessible directly from the other objects that work with the event queue, but they need not to be encapsulated inside the interface object. The superior/subordinate relation may be expressed only via communication relations, and it may be ensured by the interface object through the locking mechanisms. Even though the internal objects are not encapsulated in the OO terms of the implementation language, we may consider them encapsulated inside the interface object from the logical, abstract point of view.

The evaluation of one event. Depending on the implementation of the simulation system, the event-queue can play or active or passive role. In the first case, the simulation algorithms are built inside the event-queue object, and the evaluation objects—the application object of classes which implement simulated subsystems—are activated by the event queue. In the latter case, another object must be responsible for implementation of the simulation algorithm. The event-queue object is then responsible only for storing and retrieving the event records.

Preserving parallelism in subsystems. If we think about building a simulation model using the Beauty programming model, the subsystems of the simulated system will be implemented as application objects. The main reason is that an implementation of a model will be written by an application programmer. Therefore, the implemented objects may encapsulate only one thread of activity. However, the application programmer should preserve as much of the internal parallelism of the model as possible. The pitfalls of the sequential thinking are even more dangerous here (the application programmer is probably much less skilled in the "parallel thinking" in comparison with the programer of system objects). The Beauty model is being designed so that it lowers the chance of unintended usage of *bad practices*. However, it does not recognize a possible parallel solution where the programmer prescribed the sequential one. Even though designed for parallel programming, it still perfectly allows the sequential programming.

Reusability. Beauty fully supports building applications as hierarchy of objects. The application objects may contain or may create both system and application object. This way they may create internally levels of abstraction. The classes, which implement often used subsystems, may be stored in libraries to be reused later. The subsystems which implement typical evaluation and other objects used for the simulation can be prepared for application programmers. This way, a lot of the above mentioned pitfalls may be surpassed.

4 Conclusion

The object-oriented approach fits well with modelled objects in simulation. The OO approach can also be used for general parallel programming. The Beauty model for parallel programming seems to be promising also as the mean for parallel implementation of a simulation system. Purely object-oriented design of Beauty allows building the simulation model as a hierarchy of communicating objects. Difficult parts of the simulation system can be implemented as system and application objects by the designer of the simulation system thus removing a lot of pitfalls of parallel programming from the application programmer.

The application objects are used for implementation of subsystems of the model. They encapsulate only one thread of activity thus simplifying their programming. Non-blocking operations of OSp may still be used for working with other application objects in a parallel manner. The implementation of even a simple queuing simulation system proved to be a good case study for testing features of the abstract model for parallel programming — here the Beauty.

References

[1] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[2] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[3] Petr Hanáček. Problems of Implementation of Virtual Time for Parallel Simulation. In Štefan [12], pages 298–303.

[4] John Markoff. David Gelernter's Romance With Linda. *The New York Times*, page 1, January 19 (Sunday), 1992. In the Business Section (Section 3).

[5] Peter Meňhart and Jiří Šafařík. Development of the Distributed Simulation System. In Štefan [12], pages 253–258.

[6] Petr Přikryl. Beauty — The Language for Parallel Programming. In *Proceedings of WORKSHOP 97, Part1 I.*, pages 185–186, Prague, January 20–22, 1997. Czech Technical University in Prague. Extended abstract.

[7] Petr Přikryl. Discrete Simulation in the Linda Programming Language. In Jan Štefan, editor, *Proceedings of MOSIS '93*, pages 238–245, Olomouc, Czech republic, June 1–4, 1993.

[8] Petr Přikryl. Extension of Linda and the Discrete simulation. In Jan Štefan, editor, *6th Moravo-Silesian International Symposium on Modelling and Simulation of Systems MOSIS '94*, pages 188–195, Zábřeh na Moravě, Czech republic, May 31 – June 2, 1994.

[9] Petr Přikryl. Implementation of a simulation system in the PIN language. In Štefan [12], pages 235–240.

[10] Petr Přikryl. Parallel Simulation using Communicating Objects. In Jan Štefan, editor, *Proceedings of MOSIS '97, Vol.1*, pages 239–245, Hradec nad Moravicí, Czech Republic, April 28–30, 1997. MARQ.

[11] Petr Přikryl. PIN — the Language for Parallel/Distributed Environment. In Jan Štefan, editor, *Proceedings of MOSIS '96, Vol.1*, pages 187–192, Krnov, Czech Republic, April 23–25, 1996. MARQ.

[12] Jan Štefan, editor. *Proceedings of XVIIIth International Workshop on Advanced Simulation of Systems ASIS '96*, Zábřeh na Moravě, Czech Republic, September 17–19, 1996. MARQ. Acta MOSIS No. 62.