Parallel Simulation using Communicating Objects

Petr Přikryl
Department of Computer Science and Engineering,
Faculty of Electrical Engineering and Computer Science,
Technical University of Brno, Boetchova 2, 612 66 Brno, Czech Republic
tel. +420-(0)5-7575 218 e-mail: prikryl@dcse.fee.vutbr.cz

Abstract: *The article puts together simulation systems and parallel computing environment. It shows what can be parallelized and suggests the parallel object-oriented approach to the implementation of such systems. The information may be valuable for people who are deeply interested in simulation but who do not know very much about parallel programming.*

The implementation of parts of an event-driven simulation system was used to test features of selected languages for parallel programming in my previous papers [9, 7, 8, 10]. The experience helped me to design the object-oriented model for parallel programming. Its basic features are briefly presented at the end of the article.

Key Words: Parallel programing, coordination, parallel simulation.

1 Introduction

Why parallelism? Doing things in parallel is a natural way how to do them faster. The term *parallelism* is used sometimes as a magic word related to the future of programming. However, parallelism is used in real-life situations as common as cooking the food, building a house, assembling a car, etc.

Parallelism is inherited to all real systems, and the parallel solution is always based on very simple observation: we try to split a bigger problem to smaller ones which are mutually independent, and which therefore can be solved by different *agents* at the same time, in parallel. In the computer terminology, the agents are called *processors*. The more processors we have and the more independent subproblems can be formulated, the more we gain from the parallel approach.

However, there are limitations of the maximum level of parallelism in all solutions. Firstly, the cost of decomposition grows with the quantity of subproblems. Secondly, the solutions of subproblems have to be put together to obtain the solution of the whole problem. The more subproblems is solved, the bigger is the cost of the composition of results. Thirdly, all parallel activities must be effectively *coordinated* when using shared resources or when the results of one activity is to be passed to another activity. We are always searching for trade-offs. We are weighting the overall cost of the solution, considering implementation environment, keeping in mind natural, inherited parallelism of the task, and applying other criteria.

Parallelism and simulated 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. If we think about a process as about the purely *sequential* activity—the basic building block of parallel computation—then we can distinguish between the systems where a process implements behaviour of a real physical or logical object (like devices and activities in queuing systems, messages in communication protocols, or logical elements in digital circuits), and the system where a process implements the computation of a part of the theoretical model (like a differential equation in the problem of heat conduction or of fluid dynamics). The border between the mentioned kinds of systems is often fuzzy — it depends on a viewpoint of the observer and on abstractions used for modelling. Anyway, we can always find many components that implement parts of the simulated system and that can be evaluated in parallel.

Parallelism and simulation models. Building a simulation model, we want to express and model the behaviour of the simulated system. We always state some restrictions. It is hardly possible to create a model whose behaviour is indistinguishable from its original. This would yield solutions like "the most accurate simulation model of a cat is another cat". However, we should preserve the main features of the simulated system.

The simulation of systems is difficult also because the real systems often work *more efficiently* than the model. One of the main reasons is that they really work *in parallel*. Claiming this to be one of the basic features of real systems, we should preserve it also in the simulation model if it is possible. And we do so. We are thinking in categories of hierarchical structures of submodels that *run in parallel*, and we tend to think about parts of the model as about *objects*. To be short, simulation models naturally work with communicating objects and with hierarchy. See [5] for more details on Hierarchical modelling based on communicating objects.

Implementation of a simulation model, which is inherently parallel, depends on hardware and software we can use. If we do not have a parallel computer, then the real parallelism of hardware have to be replaced by a sequential, software solution. In our case, the software is called a *simulation engine* or simply a *simulator*. It is used to serialize the run of the model. A potentially parallel program is converted into the sequential one.

Parallelism and simulation experiments. When building a simulation model in different implementation environments, we cannot get rid of *model time* which simulates the real time of the system. It is used for *coordination* of activities of the model. On the other hand, we may think about replacing the the sequential simulator by its parallel counterpart. The parallel engine should be designed on principles that gain from real parallelism. As speed is our goal to achieve, we do not want to parallelize *only* inherently parallel activities—like parallel evaluation of model components—but we *also* want to use new techniques which could not be used on a single-processor hardware. Just to give you an example, we can use the *time-warp* technique to evaluate the model components in parallel even for different model-time instants (see [2] for details).

If we have *more than enough* processors for the run of one simulation experiment—massive-parallel computers can have tens of thousands of processors—we can run more separate experiments with the same model but with different initial conditions. Then we can build new hierarchical level of the simulator. We can implement various optimization strategies and run many experiments (in parallel) to find optimal parameters of the model, or to get acquainted with the model's behaviour.

2 How to build a parallel program

Parallelization of a code versus parallelization of algorithms. Considering procedural programming languages, two basic approaches to parallelization are used. The first approach is based on the *automatic analysis and parallelization of the code.* The analyzer (a program) searches for typical patterns in a sequential code and tries to convert it into a parallel equivalent. For example, mutually independent instructions or blocks of instructions can be processed in parallel; iteration loops can be replaced by many parametrized parallel processes, etc.

The second approach suggests new category of programming languages with commands and constructions or with library functions which can be used for explicit description of a parallel computation. The approach can be called the *explicit parallelization* or human based approach, because the human is responsible for writing the algorithms in the form where the parallelism is expressed explicitly.

Both approaches have advantages and disadvantages. The automatic parallelization can be used to rebuild older applications without deep knowledge of the solved problem. And also, sequential algorithms are sometimes written in such a way that it would be very difficult (for a human) to find all possible sources of parallelism. The automatic parallelization can also be tailored for special purpose algorithms where some operations are inherently parallel. This way, for example, research community of people who frequently use matrix operations need not to be interested in learning details about parallel programming, yet using the advantages of parallel processing.

The explicit parallelism seems to be better for building new, general applications assuming that the programmer is familiar with the parallel programming language. He or she can use special programming techniques which require a deeper knowledge of the problem but which yield better results than the automatic approach.

Coordination and the mechanisms. Coordination mechanisms appeared in programing far before the true parallel programming was used. However, solutions related to the coordination in sequential languages are so natural that we do not think about them. For example, let one process (program) generates data and writes them to a file. After that we run another program which reads the data from the file and process them further. In the terms of coordination, the two processes share a common resource — the file. But they are separated in time therefore we need not to worry about their mutual exclusion.

The problems of coordination become more apparent when the processes are not separated in time. This can happen when the processes runs in the same time, even though they need not to run really in parallel. This is the case of multitasking operating systems which are often built as multiple-user systems. Even when only one processor is used to create more *virtual computers* (i.e. only one process can run in one time instant), the processes are assigned the processor for only short time-slices. Consequently, the processes may compete for shared resources. Using the previous example of communication via a file, we need locking of the file in some situations. The problem is well known from database systems where the centralized database is shared via network.

Similarly to the case of programming languages, the coordination mechanisms have developed from usage of low-level operations (like test-and-set instruction) through higher level operations (operations of semaphores, monitors, basics of message passing) to more abstract operations like *remote procedure calls* or built-in constructions implemented in high-level programming languages. Thinking about coordination mechanisms, we can distinguish between the mechanisms for communication and for the synchronization of processes; however, we cannot say that one mechanism can only be used for solving a communication problem and not for the synchronization (and vice versa). For example, the semaphore can be used to synchronize the run of two processes (one of them waits for the other), but it can also be used to control the access to a shared communication media (like a block of shared memory). We also need some operations and constructs for expressing the parallelism. Also here, the well known but low-level operations (like *fork-join*) developed into higher level constructions built in high-level parallel programming languages (like *parbegin-parend*).

To summarize, new languages for parallel programming, based on explicitly expressed parallelism, are designed for humans. The abstract thinking of a programmer must be supported by powerful (but theoretically correct) constructs for expressing parallelism and coordination. Event though the basic mechanisms for parallel computing already exist—they are proved to be correct and strong enough in the theoretical sense—they do not protect the human programmer from doing *stupid* mistakes. As programming has also psychological aspects, we are still searching for better mechanisms and constructions — the ones which better reflect the process of human thinking.

Communicating objects. The working and research area of simulation of systems is tightly bound to reality where the notion of *object* is very natural. This is probably the reason why the first language which introduced objects was the simulation language — Simula. What are the reasons for using objects?

Objects clearly define parts of any system. One of the most important features of objects is called *encapsulation* — an object encapsulate its status and behaviour which can be observed only through clearly defined interface. The object *class* represents the behaviour of many objects that belong to the same abstraction. The object is responsible for interpretation of its internal data. There is nothing like implicit interpretation of some portion of data placed somewhere outside of all objects in purely object-oriented approach. Data are always encapsulated together with the methods.

The objects interact each with other via mutual calling of methods — we usually think about it as about sending messages. In a parallel environment, the objects may "live" in parallel. The object can be simple or complex to meet our needs still having the same interface. We can think about an object as about implemented abstraction of a subsystem. When the behaviour of the object is inadequate to the real subsystem, we can enhance the implementation of internals of the object. As the object represents some abstraction and hides details, it is easier to analyze theoretically the system based on objects than the system where data are separated from methods that are used for interpretation. For example, the article [3], which is interested in the behavioral equivalency of simulation models, introduces new formal model of discrete-event systems because the existing formal models cannot be successfully applied in real cases. The object-oriented approach makes implemented systems more clear and it also allows to enforce some theoretical restrictions easily and elegantly. We cannot expect miracles, however. But the objects allows us to define the interface and the behaviour very clearly, thus creating good abstractions hence a good fit to formal descriptions.

3 Parallel programming model

Specialized distributed applications. A simulating system which uses the aggregated power of many processors can be built in many ways. For example, when we are interested in usage of local area network computers, when we know advantages and disadvantages of the environment, and when we are oriented toward building a specialized application—of the simulator—then it is reasonable to think about specialized solution like the distributed simulation system in [4]. Such systems can be tailored and optimized to get the maximum effect for the projected purpose.

General model for parallel programming. Personally, I am interested in the area of general parallel programming, but I have also some simulation background. The problems of simulation seem to be a challenge for many parallel programming systems. Implementation of a simple event-driven simulation system can be used as a good case study to test the potentialities of the parallel programming language or system.

The article [6] presented basic features of the language Beauty. The language was derived in several steps from Linda [1]. However, the original tuples and processes were totally replaced by objects. These days, the language is being converted into more general *model for parallel programming* (i.e. language independent; the model is still unnamed). The goal is to design abstract classes of objects which create a uniform environment for parallel programming while hiding details about the computing environment.

When we think about the structured programming in sequential programming languages, we can see that it is not enough to write *some* procedures and functions. We have to use a recommended methodology and programming techniques to get good results. Similarly, usage of the object-oriented or object-based approach does not lead automatically to the success. Also the OO design must follow some rules that makes the system implementable efficiently. This holds namely in a parallel environment. We must not forget that objects live in parallel and they can send messages while also other messages are just passed. The activities of objects have to be coordinated, and the coordination must be intuitive and easy from the programmer's viewpoint, yet preserving correctness. The corresponding methodology should also be developed.

The programming model is based on two kinds of objects. *System objects* create an abstract environment for application objects. The *application objects*—basic building blocks of applications—communicate together via system objects. They are designed by an application programmer. They encapsulate always 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 architecture. The fundamental system object is called *Object Space* (OSp). It is responsible for creation and destruction of application objects and for their mutual communication. The other system objects can be viewed as extensions of the OSp. For example, the system class *group* can be used for indirect operations with a set of application objects. The following table shows the methods of OSp which are used by application objects. The OSp is used similarly like a telephone switch. The four lines in the table correspond to the four types of the access to the application objects. The operations will be implemented as 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 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 last mentioned operations *replicate* and *collapse* are used for hardware independent optimization of some parallel algorithms.

4 Conclusion

The object-oriented approach fits well with modelled objects in simulation. The OO approach can also be used for general parallel programming. Both areas define some restrictions and requirements. We can build a simulation system as a specialized application, or we can use some general model, system, and implementation language for parallel programming. The result of my work in both areas led to the design of new, object-oriented, hardware independent model for parallel programming which can be used effectively also for implementation of a simulating system. Details of the programming model will be presented in my future papers.

References

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

[2] Petr Hanáček. Problems of Implementation of Virtual Time for Parallel Simulation. In Štefan [11], pages 298–303.

[3] Peter Meňhart. Ekvivalencia sprvania sa modelov diskrtnych systmov. In Štefan [11], pages 227–234. In Slovak.

[4] Peter Meňhart and Jiří Šafařík. Development of the Distributed Simulation System. In Štefan [11], pages 253–258.

[5] Petr Peringer. *Hierarchical Modelling Based on Communicating Objects*. PhD thesis, Technical University of Brno, Czech Republic, December 1996. In Czech.

[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 [11], pages 235–240.

[10] 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.

[11] 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.