PIN — the language for parallel/distributed environment

Petr Přikryl
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. +42-(0)5-7575 216 Fax +42-(0)5-4121 1141
E-mail prikryl@dcse.fee.vutbr.cz

23.–25. April, MOSIS'96, Krnov, Czech Republic

Abstract: *PIN is a proposal of the language for parallel programming. This article further elaborates on ideas presented in [6] and introduces the object oriented approach to the language design. The motivation from articles [5] and [6]—to implement a simple general queuing simulation system in a parallel/distributed environment—still raises many questions. However, the PIN language is not intended to be designed as a specialized simulation language. PIN is based on the idea of the generative communication mechanism which was firstly implemented in the system named* Linda *[2].*

Keywords: Event-driven simulation, parallel programming, generative communication, Linda.

1 Introduction

Simulation systems are designed to simulate behaviour of real systems that are essentially parallel. A simple thought leads to the conclusion that a parallel implementation of the simulation model is more natural. But multiprocessor computers are still not very common, and their commercial usage is restricted to highly specialized cases. As a consequence, a parallel programming is not as spread as the sequential one. However, new programming models and methodologies are being developed, and new programming languages are being designed to fill the gap.

The Linda language was used in [5] to show how the parts of a simple, general queuing simulation system can be implemented in parallel. The article focused on questions of how such a simulation system can be naturally parallelized even in a non-expensive computer environment. The article [6] had shown that the original Linda operations are not powerful enough (from the practical programmer's point of view) to express some ideas efficiently, and it focused on enhancements of the operations.

Further research led to the conclusion that this approach cannot be used to reach the desired functionality of a language for parallel/distributed environment. At least, the functionality cannot be expressed easily and elegantly this way; and also, the solution would not be clean and simple from the theoretical point of view. The result of the research is the proposal of PIN.

The language design is always a trade-off between a strict mathematical purity, potentials of current hardware and software technology, and subjective demands of programmers. The language should be clean, simple, expressive, and intuitive. Extremes in some direction lead to inefficiency. If this holds for sequential programming languages then it is even more important for parallel programming tools.

The book [1] shows three typical ways of building parallel programs: using distributed data structures, using live data structures, and using message passing approach. One cannot say what approach is the best. It depends on the kind of problems being solved. The first two approaches are *usually* more intuitive, the third one is *usually* the most efficient when thinking about a real hardware. However, the message passing approach usually leads to more difficult programming. A programmer has to keep in mind a mental picture of many processes and of their explicit cooperation.

The *generative communication* paradigm (GC)—appeared at Yale University in the 80's [2]—supports the first two earlier mentioned methods of building parallel programs. The GC separates processes by removing the need of the explicit communication. Therefore, a programmer can develop processes separately — the programming becomes mentally easier. The GC paradigm allows the design of a parallel programming tool built from two separable means: a computing language and the coordination language. As the PIN coordination language is derived from Linda, the originally proposed operations will be described first.

The *Linda* coordination language uses an abstract, globally shared, persistent data space called *tuple space*. *Tuples*—basic data elements stored in tuple space—consist of one or more typed elements. Tuples are content addressable (an associative memory), and they can be used by processes as the communication and synchronization means. Tuple space is also an environment for launching of processes that are executed in parallel. Every process can access all tuples in tuple space, but tuples are not bound to processes.

Four[1] basic operations, implemented as function calls in a computing language (for example C), act as a glue that makes otherwise independent processes working together. The operation `out(`*`tuple`*`)` places the tuple into tuple space. The operation `in(`*`tuple-template`*`)` searches for a tuple that matches the tuple template. If the tuple is found, the requested values from the tuple are returned, and the tuple is withdrawn from tuple space. Otherwise, the operation blocks until suitable tuple appears in tuple space. The operation `rd(`*`tuple-template`*`)` works like `in()`, but the tuple is not removed from tuple space. The operation `eval(`*`tuple`*`)` creates concurrent processes to evaluate fields of the tuple, and the results form a tuple that is placed to tuple space.

2 Associative versus direct access

The Linda language defines the associative access to data as the only possibility. From the philosophical or theoretical point of view, the direct access is only a special case of the associative one. The address (or identification, handle) of a data object can be interpreted as an associative key for accessing the data. However, there *are* differences between associative and direct accesses to data if we think about implementation details.

Firstly, more than one data element can be found during searching for data when an associative key is used. If only one data element should be returned then the associative access introduces *non-determinism* related to the data choice. The direct access is based on a unique identification of each object thus making the data choice implicitly deterministic.

Secondly, the direct access to data introduces a notion of the *existence/non-existence* of data. Unlike in the associative access case, it makes sense to think about *undefined* values of data when they are accessed directly. In other words, similarity with physical locations of data

[1]Six operations were originally defined by authors, but the predicate versions of `in()` and `rd()` operations introduce the notion of real time into the parallel programming thus raising more problems than advantages [4].

exists. In Linda, the process that called the operation has no notion of time, it does not know that the data did not exist for a while because it is blocked until it can be satisfied.

The next difference between associative and direct access to data is *efficiency*. Using current hardware and software technology, the direct access is more efficient. Moreover, some programming abstractions can be implemented better, more easily when the direct access to data can be used. The truth is that some problems intuitively lead to the associative access; therefore, it is a nice idea to support both kinds of the access to the data.

The direct access to data in PIN is based on a unique identification of objects. If we think about the original Linda operations then the identification can be viewed as an extra element added to each tuple by the Linda-kernel when a new tuple is formed and placed in tuple space. The natural extension of the original operations are new operations where the first parameter defines the identification. The extension was proposed in [6] where so called *id*-operations were defined (`rdid()`, `inid()`, `modid()` – atomic modification of a tuple). Syntax of the original operations was slightly changed. The operations return identifications of tuples that were accessed. Therefore, the operation could appear also at the right side of the assignment operator (think about C language). The extension was inspired by the need for implementation of graph data structures in tuple space. The extension proposal was revised mainly because of replacing tuples by more promising and more general *objects* (in the OOP[2] sense). The extension also raised more questions about other possible operations.

3 From tuples to objects

If the associative access is the only method for accessing data stored in a shared memory then a tuple seems to be natural kind of an object because one or more elements of a tuple can form an associative key. The *searching engine* of tuple space uses implicit knowledge of a tuple structure and of a tuple template thus interpreting their elements correctly.

A tuple seems to be a good abstract data object here, but only from the theoretical point of view. While it is not difficult to change a theory very quickly (it is purely abstract), it can be very difficult to change the real implementation of the theory. Just to give you an example. In [1], the authors mentioned that their Linda implements the dynamic array as one of the possible type of elements. The reason is that they needed to store references to other tuples in the sense of arcs of graph structures (for some intended applications). The number of references could not be decided in the time of programming. They introduced the dynamic arrays not because of theoretical reasons, but because of solving of practical problems. What happens, however, when the mean like a dynamic array is not enough for certain new class of applications? This is the argument for introducing much more flexible type of data. We can use *objects* instead of tuples. The objects emphasize the fact of being the entity and hide their functionality. They have to have some mandatory methods, related to globally shared data space, but they can have many methods that are application dependent and that interpret the encapsulated information correctly. This way also operations with globally share data space can be more abstract, because the data space (the former tuple space) need not to interpret the information from data elements — the separable data elements of some type simply do not exist.

If tuple space is viewed as an object then the original operations can be viewed as its methods. Let's call the globally shared structure of PIN the *object space* (OSp[3]) instead of tuple space. If the object space is implemented as the object itself, we can simply name it `OSp`. The original

[2]Object Oriented Paradigm/Programming.

[3]The 'p' is used inside the acronym to distinguish it from the OS which is usually used for abbreviation of *Operating System*.

Linda-operations become the methods of `OSp`. For example `OSp.out(`*`class, arguments`*`)` generates new object of the *`class`* inside the object space. The *`arguments`* are passed directly to the constructor of the object.

When we think about replacing tuples by objects then also the associative searching for an object must be implemented differently because data of the object are encapsulated inside, and. obviously, they can be interpreted correctly only via object's methods. Therefore, the searching engine has to invoke the mandatory method `obj.match(`*`ta`*`)` of each potential candidate object `obj`. The *`ta`* stands for template arguments. The arguments can be interpreted correctly only by objects of certain class, therefore only objects of that class have to be considered. The template arguments are taken directly from the method of OSp that requested the associative searching. Syntax of the call could look like `OSp.in(`*`class, ta`*`)`.

The naive testing of all objects of the class in the OSp could be rather inefficient; therefore, the searching engine can narrow the quantity of potential objects using additional, auxiliary information related to object entities — the *object signatures.* The signatures are collected inside OSp using the mandatory object method `obj.signature()` and they are updated whenever an object changes its state (the object must decide). The signature—if it can be calculated from the template arguments—can also be passed to the `match()` method along with template arguments to increase the speed of the process of matching. However, there is not a lot of things that could be said about signatures as they can be derived only from the encapsulated information from inside of the object. We can simply think about a signature as about fixed data structure that depends on the class and the content of the object, but the method of its calculation can be heavily application-dependent.

Objects stored in OSp can be accessed directly via their identifiers. An identifier belongs to the instance of the object class. The importance of the object identification is essential because it is one of few information that can be interpreted by OSp methods correctly[4]. The other information have to be interpreted by the candidate-object methods.

As OSp may be implemented in a distributed and heterogeneous environment, the object methods must be context independent. In other words, all values used by an object methods must reside inside the object or they have to be retrieved from OSp. Nor global constants, nor global or friend functions are allowed[5]. Code of object methods must be also distributable hence stored in a machine independent form[6]. Naturally, the code of methods belongs to classes, not to instances of objects. Therefore, it can be stored separately from object data. Data and methods can be linked via unique class identifications.

Notice: We can use the object oriented design also for implementation of the original Linda-like system. The internal object oriented implementation of PIN need not to be visible for a programmer. Syntax of the coordination language operations can be preserved thus obtaining pure Linda-like system. It can be preprocessed to OSp method calls. Tuples can be implemented by appropriate objects. This can be helpful when the PIN system is being developed. The original and representative Linda programs can be used for testing the PIN implementation. Moreover, a better syntax of the PIN coordination language can be developed partly independently on the PIN-kernel development.

[4]Compare it with extended tuples where usage of identifications was optional [6].

[5]This problem is not as simple as it can look in the first time. E.g., think about dynamic data structures built by an object itself.

[6]E.g., in interpreted or on-line compiled byte-code. Implementation languages like Smalltalk, Java, or (just being developped) Inferno are considered as a base language for PIN.

4 Why PIN?

Section 1 discussed basics of the generative communication paradigm and of the Linda language (the implementation of GC). Next section (2) described how the PIN coordination language extends the original operations defining the direct access to the object space—the counterpart of the tuple space. The previous section (3) showed how more general objects than tuples can be introduced into the language. The question stated in this section heading can be interpreted as "*why the language is called PIN*" or "*why the PIN was designed*". You can find answers to both questions further in the text.

As the generative communication paradigm does not use the explicit communication between processes (shared data are not contained inside processes; there is no need for explicit send/receive operations) a programmer need not to think so hard about mutual dependencies of processes. Parallel programming becomes easier. But data are not bound do processes which means that, just after creation, a process does not know direct access to any data. An associative access is essential here to establish the communication.

It is not difficult to find out that there is an antagonism brought by GC paradigm. Data can be accessed by any process, but the process should be able to access only the data that belong to it from the application point of view. However, many Linda applications tend to use very simple hence very similar tuples, e.g. `(string,integer)`. Moreover, two instances of the same application can be launched in the same system. Then the program should be designed so that it anticipates such cases. This makes parallel programming more difficult again. Programmer can do mistakes easily. Therefore, some possibilities for restriction of the access to only some data should exist.

The problem of coexistence of applications that use similar tuples, or the problem similar to existence of local variables is solved in [3] by creating multiple tuple spaces — new operations are defined for the purpose. However, adopting that view, a newly created tuple space becomes dependent on the application. It is not globally shared more. In other words, a persistency of such tuple space is unclear.

Another implementation of the GC mechanism [4] works with user defined types of elements. Tuples which use elements of user defined types cannot be accessed by processes that do not know the type. This way, they are effectively separated from the other application processes while they still resides in the globally shared and process independent tuple space.

PIN solves the restricted access to objects via *access rights*. The access rights handle is stored in a system part of a object together with the unique object identification and class identification. Processes inherit the access rights handle from the process that created them. The father process can also decide to modify the access rights hierarchy and to assign newly created process another handle obtained from the PIN-kernel. The strategy of access rights can be implemented easily. Moreover, it need not to be hard-coded in the PIN-kernel. It can be even customizable by overloading of the `match()` method of the object.

PIN introduces also another protection against the *associative access confusion* error. Each object can be *pinned*[7] down (as a butterfly). Pinned objects are ignored by the associative operations. In other words, their `match()` method always return *false* if the object is pinned. They can be accessed only directly, via their identification. To find out entry points to pinned data structures (e.g. to many objects forming the graph structure), new, mandatory, and the only operation based on associative search that can return the identification of pinned objects was defined: `OSp.getid(`*`class, ta`*`)`[8]. It works also for unpinned (free) objects. Development of the PIN language is in its early stages, therefore the definition of other operations for using the

[7] The importance of this feature gave the name to the PIN coordination language.

[8] See section 3 for meaning of the arguments.

feature was postponed to avoid mistakes in the design. However, methods like `OSp.outpin()`, `OSp.pinid()`, `OSp.unpinid()`, etc. can be predicted.

Thinking about efficient implementations of some programming abstraction, one more feature was defined in PIN. The object in OSp can also be locked. Then it is temporarily unaccessible by all the above mentioned operations. The key can be spread to more processes (via OSp) so that more processes has the exclusive access right to access the locked object. Unlike the access rights, the locking of the object is temporary, and it is bound to certain processes. It is expected that the locking scheme is valuable only in combination with direct access. Operations like `lockid()` and `unlockid()` can be anticipated, at least. The operations can be used for implementation of atomic operations when more than one method of an object should be invoked. The locking mechanism resembles the usage of Dijkstra's *semaphores*. The key to the locked object represents the right to use the object's semaphore.

5 Conclusion

The PIN coordination language is still under development. Practical examples (case studies) showed that using exclusively the associative access to data is not always the best solution. Therefore, PIN introduces also the direct access to data. Searching for the explicitly referred objects can be implemented more efficiently than the associative searching. It can also help a programmer to write more efficient programs or to build algorithms where the explicit access to data is more natural.

Basic elements, stored in a globally shared space, are objects in the OOP sense. Replacing tuples by objects promises better features from the programmer's point of view, and also the higher language purity.

To avoid *stupid* programmer's errors, the PIN language introduces several means for restricting the access to objects in the object space. In addition to direct access to data (mentioned earlier), also the access rights and a locking strategy is proposed. All mentioned enhancements of PIN, in comparison with Linda, were designed to be easily implementable. Some special functionality—like application dependent, more efficient process of matching—need not to be implemented on the system level.

References

[1] Nicholas Carriero and David Gelernter. *How to write parallel programs: a first course.* MIT Press, 1990.

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

[3] David Gelernter. Multiple tuple spaces in Linda. In G. Goos and J. Hartmanis, editors, *PARLE'89: Parallel Architectures and Languages Europe, Vol. II: Parallel Languages*, number 366 in Lecture Notes in Computer Science, pages 20–27. Berlin: Springer-Verlag, June 1989.

[4] Jerrold S. Leichter. Shared tuple memories, shared memories, buses and LAN's — Linda implementations across the spectrum of connectivity. Technical Report TR-714, Yale University Department of Computer Science, July 1989.

[5] Petr Přikryl. Discrete simulation in the Linda programming language. In Jan Štefan, editor, *Proceedings of 5th Moravo-Silesian International Symposium on Modelling and Simulation of Systems*, pages 238–245, Olomouc, Czech republic, June 1–4 1993.

[6] Petr Přikryl. Extension of Linda and the discrete simulation. In Jan Štefan, editor, *Proceedings of 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.