Abstract. Most parallel object oriented languages (OOL) are currently using a general parallelism model based on communicating sequential processes. This approach makes it difficult to program massively parallel systems in an easy and efficient way. So we propose to use another form of parallelism, known as data parallelism. We describe how a pure sequential OOL can embed data parallelism in a clean and elegant fashion to exploit the potential power of massively parallel systems. Then we present EPEE (an Eiffel Parallel Execution Environment) at work with a well known parallel paradigm (matrix computations), along with experimental performance results. We draw some conclusions on the generality of this approach.

Keywords: Massively Parallel Architectures, Data Parallelism, Parallel Object Oriented Programming, Eiffel

1 Introduction

It is now widely agreed that the road leading to TeraFlops machines (i.e. machines performing $10^{12}$ floating point operations per second) must go by massively parallel architectures. Only modular and distributed memory multi-processor architectures can be scaled up— theoretically— at will. Precursors of these architectures already exist today, as illustrated for example by the Intel iPSC hypercubes or the transputer based machines. More powerful ones are about to come as a result of research projects led in Europe (GP-MIMD machines), in America (Touchstone project) or in Japan.

But these Distributed Memory Parallel Computers (DMPC) presently suffer from a lack of programming environments: the user must have a wide experience in parallel programming and a good understanding on his machine architecture and operating system. This is probably the main factor hampering their wider diffusion. So, it is our strong feeling that the Object Oriented powerful concepts should be made available to program DMPC.

Parallelism is often introduced in OOL through the idea that objects can be made active, i.e. can be seen like concurrent activities communicating by sending messages: the object oriented message passing paradigm is mapped onto the communication structure of a parallel system. In that case, object creation can involve a process creation. Such notion of parallelism is implemented in POOL-T [1], where an object may have a “body” (kind of background activity). Communications between objects
is based on the Remote Procedure Call paradigm, and the POOL-T user has to control explicitly the concurrency structure.

Another way to introduce parallelism is by means of asynchronous operation calls: an object calling a method of another object may continue its activities in parallel with the object executing the called operation. This is implemented for instance in ABCL/1 [15], ELLIE [2], ConcurrentSmalltalk [14].

Both ideas of active object and asynchronous operation calls are used to make OOL parallel either by integrating this parallelism directly into the design of a new concurrent programming language, or simply by extending existing sequential languages to handle parallelism. The former approach leads to clear and unified support of conceptual models. ELLIE, POOL-T, ABCL/1, etc. are OOL where a significant number of structures are devoted to the deal with parallelism. Using the latter approach, we can find for example DistributedSmalltalk [4] which extends Smalltalk or COOL [6] which extends C++.

So, near all parallel object oriented languages are based on a MIMD programming model, which seems to match rather well some kind of problems (operating systems, industrial process control...). This model of parallelism is typically dedicated to handle functional parallelism: a given function is divided in some sub-functions that can be processed by new computational threads. So the definition of processes is determined by the subtask decomposition, and as this is application dependent, it necessitates a strong involvement from the user. Furthermore, the processes are of an heterogeneous nature, leading to difficult load balancing problems.

One of the most difficult obstacle to be worked around with this approach is to ensure that well known problems in the protocol engineering community —such as deadlocks, livelocks, unspecified receptions and so on— are properly dealt with. Because the cooperation between processes is explicitly coded, it is usually difficult to manage the overall communication structure.

When programming DMPC that way, we must face another major issue: efficiency. We have to remember that getting scalable performances is the main reason for using DMPC with large number of processors. However functional parallelism usually lacks of compile-time defined process and communication structures. Few regular structures may be derived from the original program at compile time, and most of the work must be done at runtime. But the dynamic nature of this process definition implies the use of general and costly mechanisms such as objects naming servers, object migration, and general purpose operating system functions to implement load balancing, scheduling, etc... Above all, this kind of parallelism suffers from a lack of scalability: the subtasks decomposition does not allow an efficient handling of the very high number of processors available on future DMPC.

To override these problems we propose to use a different model of parallelism, known as data parallelism. This approach seems to be rather natural in an OO context, as object oriented programming usually focuses on data rather than on functions. Some principles to build an OOL aimed at data parallelism have already been introduced in [10], and basic ideas to extend C++ in the same way are presented in [7]. With EPEE we go one step beyond: data distribution and parallelism is totally embedded in an OOL using only already existing language constructions. EPEE is based on Eiffel [11] because it features all the concepts that we need, using a clear syntax and semantic. However, our approach is not strongly dependent on Eiffel; it