XXXII Неделя науки СПбГПУ. Материалы межвузовской научно-технической конференции. Ч.II : С.134-135

УДК 681.3.07

M.Kosby (postgraduate student, Intelligent Systems Program),
M.V.Khloudova, PhD, associate prof.

# PARALLEL FUNCTIONAL PROGRAMMING: ADVANTAGES AND CLASSIFICATIONS

Parallel programming is inherently harder than sequential programming. The exploitation of parallelism is a long pursued – and not yet convincingly met – goal in programming. It appears to be a contradiction between the efficient exploitation of parallelism and the simplicity of the corresponding programs: the more control the language has on process management, communication and synchronization aspects, the more complex and longer, and the less amenable for reasoning, are the resulting programs written by programmers must not only describe what to compute, but also how to organize the sub-computations on the target architecture. Imperative parallel programming is a good example of this assertion. Contemporary functional languages have three key properties that make them attractive for parallel programming:

a. Abstraction: Two important abstraction mechanisms are function composition and higher order function. Function composition allows complex problems to be decomposed into simpler sub-functions. Higher-order functions are ones that manipulate other functions, which may be used to form the basis of new parallel programming constructs. Thus more efforts can be devoted to improving parallel algorithms and experimenting with alternative parallelization.
b. Elimination of unnecessary dependencies: the absence of side effects makes it relatively straightforward to identify potential parallelism; the only source of sequential dependency is that the arguments to a function must be evaluated before they can be used. Since the values do not change once they have been computed, data flow analysis is not needed to determine usage patterns, even at inter-procedural level.
c. Architecture independence: Good parallel abstractions encourage high level portability by abstracting over lower level issues. At extreme case could even lead to implicit parallelism. By using standards like PVM or MPI at runtime system level, languages can abstract over architecture characteristics.

It is well known that functional languages offer, in principle, good opportunities for parallelism exploitation due to the freedom they present in the evaluation order of their sub expressions. In some sense, the implicit parallelism is too much. If we try to exploit all of it, then we get a big number of very low granularity parallel activities, in such a way that the benefits of parallelism are lost in creating and communicating processes. For this reason, most of the approaches rely on the programmer to decide which expressions deserve the effort of creating a parallel process for their evaluation. The differences between these approaches fall mainly in the degree of explicitness they consider to be the appropriate one to deliver this information. From less to more explicitness, we can classify the languages into the two following groups.

Transformational languages. In a parallel transformational system some inputs are transformed into some outputs functionally depending on them. The whole purpose of parallelism is to speed up the computation. The programmer supplements a purely functional program with special expressions, either written as annotations interspersed in the text or provided as specialized wiring functions, that directs the compiler about where and when processes should be created. The semantics of the program with these specialized expressions is (almost) the same as the semantics of the program without them. The only difference is the order of evaluation of the subexpressions. This group can be further classified into two subgroups: annotated languages – we classify here languages such as Concurrent Clean and Glasgow parallel Haskell and skeleton based languages and the language Caliban.

Reactive languages. Reactive systems are the opposite to transformational ones: usually for them there are no clear notions of inputs and outputs or even of termination and the whole purpose of parallelism is to maintain a set of separate tasks interacting with an external environment. Of course, reactive constructs can also be used for the programming of parallel transformational systems but the set of possible systems is wider than in the previous group. Non determinism unavoidably appears in these systems and the referential transparency of functional languages is lost.

Typically, languages in this group offer constructs not only for the creation of processes but also for communicating and synchronizing them. In some sense, the resulting languages appear to be a more or less successful combination of two languages: a functional one and a coordination one. languages such as FACILE, Concurrent ML, Erlang, and Concurrent Haskell fall in this category.

Another classification proposed is on the approach the languages implement parallelism.

Skeleton based approach: Skeleton based approach defines a set of parallel templates or skeletons. The programmer writes the program using these skeletons as appropriate. A parallelizing compiler can then exploit the rules provided for each skeleton in order to produce an efficient parallel implementation of the program on target machine. Languages falling in this category are Parallel ML with skeletons (PMLS), Caliban SCL and P3L.

Process/Thread based approach. Thread based approaches to parallelism allows threads to be created, but do not provide mechanisms to control those threads. Threads are thus managed entirely under runtime-system control. In process based approaches, the language expose parallel tasks at the language level. The programmer must manage the tasks using the control mechanisms provided in the language. For example in Eden is explicit about process creation and about the communication topology. The other languages that fall in the thread/ process based approaches are Glasgow parallel Haskell and Concurrent Clean.

A even more recent approach is the data parallel functional language. the most successful is NESL. Currently two data parallel extensions of Haskell have been partly implemented. Data field Haskell and Nepal.