Abstract

Production system programs are being routinely used in diverse tasks. With the increasing acceptance, demands on production system implementations is growing. Currently several research efforts are trying to use production system programs for pattern-directed processing on large amounts of data. These efforts are pushing at the limits of the scope of applicability envisioned by the researchers who worked on the design and efficient implementation of production languages. A common feature in these applications is that they perform a large number of independent operations which could be performed in parallel. Considerable work has been done on parallelizing compilers for production languages but the results are inadequate for tasks with large scale parallelism. This paper argues that explicitly parallel production languages are necessary for such tasks and presents one such language, PPL.

1 Introduction

Currently several research efforts are trying to use production system programs for pattern-directed processing on large amounts of data. Examples include maintaining consistency in large databases and triggering actions for unusual and deviant conditions [5], expert databases [1, 6], image understanding [4], simulation [3]. A common feature in these applications is that they perform a large number of independent operations which could be performed in parallel. Considerable work has been done on parallelizing compilers for production languages but the results are inadequate for tasks with large scale parallelism. This paper argues that explicitly parallel production languages are necessary for such tasks and presents one such language, PPL.

2 Description of PPL

For more than one operation to be performed in parallel in a production language program, semantics of the language must allow more than one instantiation to be fired in a single match-resolve-act cycle. Two instantiations firing in parallel can either correspond to the same production or to two different productions. Consider the case when two instantiations of the same production fire in parallel. This will happen when there are multiple similar data items which are independent and can be processed in parallel. On the other hand, parallel firing of two instantiations corresponding to two different productions indicates that there are two different sequences of computation (which may or may not be concerned with the same data) proceeding in parallel.

PPL provides constructs to support both these cases. Situations in which multiple instantiations of the same production fire in parallel can be handled with parallel productions. All active instantiations1 of a parallel production fire together. For the purpose of conflict resolution, all instantiations of a parallel production are considered equal. This means that the instantiations of such a production are selected as a single group.

Situations in which instantiations of different productions fire in parallel can be handled with production-sets. Production-sets consist of disjoint sets of productions (which may or may not be parallel productions) which fire independent of each other. That means that if the conflict

---

1Active instantiations are those that match the current data but have not yet been fired.
set contains instantiations from a production set, at least one of them will be fired irrespective of any instantiations from other production sets that might be present in the conflict set. During conflict resolution, instantiations of all productions in a particular production set are compared only with each other and not with instantiations of productions from other production-sets. A PPL program may contain an arbitrary number of production-sets.

The production system computational model is inherently synchronous, i.e. matching of all instantiations must be completed before the selection of the dominant instantiations and once the dominant instantiations are selected, they are fired together. PPL extends the computational model to permit a structured form of asynchronous execution. A PPL program consists of one or more modules. Each module consists of one or more production-sets and has associated with it a private tuple space and a private conflict set. Instantiations of productions in the same module are fired synchronously whereas instantiations of productions from different modules are fired asynchronously. As discussed previously, instantiations from one production-set are fired in parallel with instantiations from other production-sets in the same module.

PPL extends the standard OPS5 tuple-creation action, make, to provide message-passing communication between the different modules that make up a program. Message-passing communication fits in seamlessly with the production-system computational model as individual tuples constitute messages and the different tuple spaces are message destinations.

The computation in a PPL program proceeds in the following fashion.

1. Each module has its own match-resolve-act cycle. At the beginning of the match phase, a module checks if any tuples have been received from other modules. If so, it incorporates these tuples into its tuple space. Thereafter, all productions in the module are matched against contents of the tuple space, the generated instantiations being placed in the conflict set belonging to the module.

2. The resolve phase determines the dominant subset of the conflict set and fires all instantiations that belong to it. The set of operations performed is atomic with respect to the tuple space. Conflict in the act phase could occur if more than one instantiation tries to modify a particular tuple. These conflicts are runtime errors and are flagged as such.

3. If there are no instantiations in the conflict set after the match phase of a cycle, the module waits for a message on the inbound message queue.

4. Different modules execute asynchronously to each other. Synchronization is achieved incorporating the tuples from other modules at the beginning of the match-resolve-act cycle.

Acknowledgements

I would like to thank Aarti Gupta for detailed comments on an earlier version of this paper and for the patience with which she explained them. I would also like to thank Allen Newell for many fruitful discussions on parallel production languages and for his comments on an earlier version of this paper. Thanks are also due to Puneet Kumar and Bob Doorenbos for providing comments at a short notice.

References


