GENERALISED COMPUTATIONAL MODELS FOR PARALLEL MULTILINGUAL PROGRAMMING

George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str.
Nicosia, T.T. 134, P.O.B. 537
CYPRUS

E-mail: george@jupiter.cca.ucy.cy

ABSTRACT

Recently, a number of proposals have appeared in the literature describing various computational models that allow multilingual programming in a parallel or distributed environment. In this paper we briefly review two of these models that seem very promising, namely Linda based on a shared memory model (Tuple Space) and the Program Composition Notation based on concurrent logic programming. We discuss the common points as well as their differences, and we argue the need for a thorough evaluation of the two models. We then propose the development of a similar formalism based on the generalised computational model of Term Graph Rewriting.

1. Introduction

Recently, a number of computational models have been developed aiming at providing a common basis for developing parallel programs written in the programmer's favourite language and able to execute on a variety of parallel architectures. These models are based on abstractions such as shared memory and message passing that are suitable for designing parallel programs and providing the necessary mechanisms for interprocess communication and synchronisation.

This decoupling of the language itself from the actual computational model employed, not only allows the programmer to use any language he desires, as stated above, but it also offers the opportunity to develop multilingual programs consisting of subprograms written in different languages, executing in parallel and communicating via the common abstraction formalism. There are many reasons why *multilingual programming* is useful: it allows the reuse of existing components written in some language without the need to reprogram the whole application in a brand new language; it also offers the programmer the convenience to use the most appropriate language for implementing each of the various components of the application and then "glue them together". In [1] it is argued that future programming environments should have to deal with applications requiring both symbolic and numeric computations and also possess reactive properties to support the development of user interfaces and processing of data received from sensors. Multilingual programming

formalisms should be particularly useful in providing unified frameworks that manage the concurrency in distributed control, core numeric and symbolic computation and user interfaces. In such an environment, an application may consist of a component written in Fortran responsible for numeric computation, another one written in, say, Lisp or Prolog responsible for symbolic evaluation, and yet another one written in C responsible for process management.

The formalisms for multilingual programming that have been developed so far are based either on the *shared memory* model or on *message passing*. Shared memory models are preferred because they offer a simpler programming paradigm and an easier communication and synchronisation abstraction between separate program components written, possibly, in different languages. They have also proved that can be implemented with reasonable efficiency on non shared memory architectures thus allowing the combination of the convenience inherent in shared memory parallel programming with the scalability of distributed architectures.

In this paper we will concentrate on shared memory models; in particular, we will consider Linda ([2]) based on a shared memory abstraction called *Tuple Space* and the Program Composition Notation (PCN) based on concurrent logic programming and the *logical variable*. The rest of the paper is organised as follows: the next section compares Linda and PCN and argues for the need to evaluate thoroughly the two models; section three discusses the relationship between PCN and the generalised computational model of Term Graph Rewriting; the paper ends with some conclusions.

2. Two Shared Memory Abstractions Supporting Multilingual Programming:

In order to compare two models based on a shared memory abstraction the following issues must be taken into consideration: i) what are the semantics of the synchronisation and communication primitives each model provides; ii) how are shared data addressed; iii) how is access to shared data synchronised; iv) what kind of programming paradigms are supported by the model; v) how can the primitives offered by each model be implemented efficiently in the absence of physical shared memory. In this section we will provide an initial comparison of Linda and PCN based on the above guidelines and we will argue for the need to evaluate properly the two models making use of real life applications.

Linda is a parallel model of computation based on the Tuple Space - a distributed data structure common to all processes comprising a parallel program and used for interprocess communication and synchronisation. The Tuple Space is accessed by means of a small number of primitives whose semantics is in fact orthogonal to the language that uses this model. Thus, Linda can enhance any sequential language transforming it into a parallel one; so far there are implementations of Linda for C, Fortran and Modula-2 from the camp of imperative languages, and Prolog and Lisp from the camp of symbolic evaluation languages.

Linda's primitives define the operations for accessing the Tuple Space; in particular `out` places a tuple in TS, `in` and `rd` removes and reads respectively a tuple, and `eval` creates and spawns a new process. If the required tuple is not available, `in` and `rd` will block, thus providing the required synchronisation between producer and

consumer processes. A tuple is a sequence of up to 16 typed fields separated by commas; some of them may hold actual data while others may be "variables", i.e. placeholders that can be filled with data values by means of the operations `in` and `rd`. If all the fields of a tuple are data values, then the tuple is a data tuple; if some of the fields play the role of formal parameters, then the tuple is a template which specifies what sort of tuple must be retrieved from the TS. In the latter case the data fields of the template play the role of indexes into the TS. For example, the tuple `("stream_a" 5 ?data1 ?data2)` is a template that can be used by, say, `rd` to retrieve a data tuple whose first and second fields have the values `"stream_a"` and `5` respectively. The fields `data1` and `data2` then will be filled with the corresponding data fields from the data tuple.

Linda encourages a master/slave computation strategy where the total work to be done by some program is broken into a number of separate tasks located in the TS. A master process is responsible for generating these tasks, place them in the TS, and at the end of the computation select the results. A number of worker processes are responsible for selecting these tasks and execute them. So the granularity of a program is determined by the task size rather than the number of active processes.

Two of the most important issues in implementing Linda in a distributed environment is support for the `eval` primitive, and mapping the conceptually memory shared Tuple Space onto a physically distributed architecture. Implementing `eval` poses some difficulties with respect to load balancing (new processes may be placed anywhere in the network), process migration overheads (the actual executable code size may be significant) and semantics (it is not clear whether a spawned process inherits the environment of its parent process). Implementing the Tuple Space itself requires the distribution of the data tuples across the local memory modules to be done in such a way to make it even but also optimise communication overheads. The techniques developed so far are based on hashing schemes (where some data fields are used as indexes) and function name schemes (where the function invoked by a tuple determines the tuple group in which the tuple resides). However, finding the best possible way to distribute the TS across the local memories of the processing elements involved in a computation is still a topic of active research.

The following example computes the infinite ordered list of hamming numbers. This example will be used to demonstrate the different programming styles and paradigms employed by each of the formalisms discussed in the paper. Unless stated otherwise, the C language is used for the versions of the hamming number generator presented in the paper (to simplify the presentation only the main pieces of code are shown in each case).

```
main()
{
 for (i=0; i<3; i++)
     eval("multiply", multiply()); /* spawn 3 processes for i,j,k */
 eval("filter_2_35", filter_2_35()); /* filtering 2i with the result of */
 eval("filter_3_5", filter_3_5());   /* filtering 3j and 5k */
 out("multiply", 2, 1);   /* send out the first value to all 3*/
 out("multiply", 3, 1);   /* processes involved in the multiplication */
```

```
 out("multiply", 5, 1);   /* to start the computation */
 out("result", 1); /* not really necessary */
 for (;;) {
     rd("result", ?number);
     printf("Next hamming number: %d\n",number);
 }

multiply()
{
 for (;;) {
     in("multiply", ?pivot, ?number);
     switch (pivot)
     {
      case 2: out("multiple_2", 2*number);
      case 3: out("multiple_3", 3*number);
      case 5: out("multiple_5", 5*number);
     }
 }
}

filter_2_35()
{
 for (;;) {
     in("multiple_2", ?number1);
     in("filter_3_5", ?number2);
     if (number1<number2) {
        out("multiply", 2, number1);
        out("multiply", 3, number1);   out("result", number1);
        out("multiply", 5, number1);   out("filter_3_5", number2);
     }
     else if (number1>number2) {
             out("multiply", 2, number2);
             out("multiply", 3, number2);   out("result", number2);
             out("multiply", 5, number2);   out("multiple_2", number2);
     }
     else
     {
      out("multiply", 2, number2);   out("multiply", 5, number2);
      out("multiply", 3, number2);   out("result", number2);
```

```
      }
     }
}

filter_3_5()
{
 for (;;) {
      in("multiple_3", ?number1);
      in("multiple_5", ?number2);
      if (number1<number2) {
         out("filter_3_5", number1);
         out("multiple_5", number2);
      }
      else if (number1>number2) {
              out("filter_3_5", number2);
              out("multiple_3", number1);
           }
      else out("filter_3_5", number2)
 }
}
```

The above coding reflects the pipelining and feedback behaviour of the process network established during the execution of the program. Note that there is just one `multiply` task that works for any number whereas in the case of the filtering processes we have distinguished their functionality to force ordering in the produced partial results. Note also that the final output from `filter_2_35` is effectively copied to all three `multiply` tasks; compare this with the versions that follow.

The Program Composition Notation (PCN) is another formalism based on a shared memory abstraction and in particular that of shared variables. PCN draws heavily on the expressiveness of Concurrent Logic Programming (on which the Japanese Fifth Generation Computer Systems project was based) and the experience gained in implementing Concurrent Logic Programming Languages (CLPLs,[4]). Unlike Linda, where the granularity of the program is defined by the set of active tasks, in PCN a program is composed of a number of active processes (possibly written in different languages), executing in parallel and communicating by means of shared variables referred to as `definitional`. What makes this formalism attractive is the fact that these variables are single-assignment: once a process instantiates such a variable to some value, no other process can change the value but only use it. This monotonic behaviour of definitional variables provides an elegant synchronisation and communication mechanism where some process has been designated as the producer of a shared definitional variable and all the other processes that share it are designated as consumers. If a consumer process attempts to

read the contents of a definitional variable before the corresponding producer process has instantiated it, the consumer process suspends. Thus, for each definitional variable a list of suspended processes is formed. Upon instantiation of a definitional variable, all the consumer processes suspended on it are activated. The benefit of such a formalism is twofold: on the one hand all the concurrent logic programming techniques such as stream parallelism, back communication, short-circuit, metaprogramming, etc. can be used in PCN; on the other hand the fact that PCN itself is orthogonal to any imperative language allows the use of a programmer's favourite language and renders the written programs more efficient than the corresponding ones written in a concurrent logic language.

PCN encourages a style of programming based on CSP: a number of components written in a variety of sequential imperative languages, executing in parallel and communicating by means of definitional variables. The model defines a number of *proper interfaces* which allow the composition of sequential components in such a way that the overall effect of executing them concurrently is equivalent to the result of a fair interleaving of the atomic actions of the individual components. This last feature allows the reuse of existing code.

PCN has been implemented on a variety of distributed architectures and many compilers have been developed that map imperative languages such as C, Fortran or Ada onto PCN. The use of single assignment variables incurs some copying overhead especially when large data structures are involved and the development of suitable compile-time optimisation techniques is an open topic for research. The above hamming numbers generator program is shown below written in "declarative" C adhering to PCN semantics.

```
hamming(res)
{|| res = [1|result],
    multiply(2,res,res2), multiply(3,res,res3), multiply(5,res,res5),
    merge(res2,res35,result), merge(res3,res5,res35)
}

multiply(number,list,newlist)
int number;
{ ? list ?= [head|tail] -> newlist = [number*head|newtail]
}

merge(list1,list2,list)
{? list1 ?= [head1|tail1], list2 ?= [head2|tail2],
   {? head1<head2 -> list = [head1|newtail], merge(tail1,list2,newtail],
      head1>head2 -> list = [head2|newtail], merge(list1,tail2,newtail],
      head1=head2 -> list = [head1|newtail], merge(tail1,tail2,newtail]
   }
}
```

Although the above program was written in PCN using C-like notation, it resembles the equivalent version that would be written in a concurrent logic language. Any undefined variable used in the body of a procedure is definitional. Its producer is that procedure that assigns it a value via the assignment operator `=` and its consumers are those procedures that access its value via the matching operator `?=`. Any group of statements enclosed in brackets and identified by the operator `||` is executed in parallel; a similar operator `?` is used to denote that the order of executing the statements in a block is immaterial and the underlying implementation should decide whether sequential or parallel execution will be employed. Note the feedback of results from the `merge` processes back into the `multiply` processes; note also the sharing of the partial list of results as they are computed by all the processes involved in the computation.

The above necessarily brief comparison of the two models shows some of their common points as well as their differences. They are both based on a shared memory abstraction and they are both orthogonal to the actual language used. They have shown their usefulness in implementing a number of real life applications. However, they encourage different programming styles. Also, PCN provides a finer grain of controlling the parallel tasks than Linda does which may enhance the efficiency of implementing some classes of problems at the expense of placing some extra burden of programming effort on the programmer. The debate on the merits and weaknesses of Linda compared with concurrent logic programming has been an ongoing argument in the scientific community ([5,6,7]). However, performing a proper evaluation of the two models w.r.t. each other was not possible, to a large extent due to the different languages employed - imperative languages for the case of Linda, logic languages based on the Warren's Abstract Machine (WAM) for the case of concurrent logic programming. The existence of PCN allows now a proper comparison of the two models since the same base language can be used for all applications developed and the same architecture (say, network of workstations) will be used for running them. We are currently setting up the two systems (Linda, PCN) to run on a network of IBM RS/6000 and we are identifying a suitable class of programs that will be encoded using both formalisms and will be used for comparison purposes. We are also interested in large scale real life applications that have been successfully developed using either of the two systems.

3. Graph Rewriting as a Generalised Computational Model

Rewriting theory is a powerful computational model for implementing a variety of declarative languages. Functional languages (FLs) based on equations can be viewed as specifying sets of rewrite rules (a *Term Rewriting System*) that determine the way a program (a tree of reducible subexpressions) will be transformed into a form where no more transformations are possible (the so called *normal form* or, more simply, the result). *Graph Rewriting* was developed as an extension and optimisation of Term Rewriting where common subexpressions are shared by means of pointers and are thus evaluated only once. This led to the development of *Graph Rewriting Systems* and, more recently, *Term Graph Rewriting Systems* ([8]) that manipulate graphs instead of trees. TGRSs had traditionally been used as implementation models for functional languages but we have shown ([9,10]) that CLPLs too can be viewed as instances of specialised graph rewriting systems; in [11] we extend this work by defining a subset of unrestricted TGRSs suitable for distributed implementation and we

show that this subset is adequate for modelling the behaviour of concurrent logic programs. Term Graph Rewriting can thus be seen as a generalised computational model able to accommodate the often divergent needs of the languages it models (lazy evaluation of modern FLs vs "eager" evaluation of CLPLs, determinism for FLs vs non-determinism of CLPLs, etc). Indeed, generalised TGRSs can also model imperative features such a destructive assignment. An intermediate (compiler target) language called Dactl has been developed ([12]) and a number of declarative languages were implemented on top of it. As an example of programming in a TGRS language consider again the hamming problem, this time written in Dactl:

```
h:Ham => Cons[1 Merge[Times[2 h] Merge[Times[3 h] Times[5 h]]]]

Times[num Cons[head tail]] => Cons[num*head Times[num tail]]

Merge[l1:Cons[hd1 tl1] l2:Cons[hd2 tl2]] =>
     If[hd1<hd2 Cons[hd1 Merge[tl1 l1]
                If[hd1>hd2 Cons[hd2 Merge[tl1 l2] Cons[hd2 Merge[tl1 tl2]]]
```

The symbol `=>` defines the way a subgraph matching the lhs of a rule will be transformed to the form specified by the corresponding rhs. Note that '`node_id:Name`' allows the sharing of the node `Name` by any process that uses its pointer `node_id`. Thus, the list of hamming numbers is fed back as it is constructed into the concurrently executing processes allowing full sharing.

We recall from the previous section that PCN is based on concurrent logic programming techniques; the reader can observe the resemblance in model behaviour of the graph rewriting version of the hamming problem with the one written in PCN. However, since concurrent logic programming can be seen as an instance of graph rewriting, we believe that TGRSs are better candidates for playing the role of generalised computational models. In particular, we are currently designing a notation similar to PCN based on generalised TGRSs. As in traditional TGRSs, a program consists of a set of nodes executing (being reduced) in parallel by means of transformations specified by sets of rewrite rules. However, a node now is an arbitrarily coarse grain sequential component written in any language. The only requirement is that communication between the nodes is done via special graph rewriting nodes called *stateholders* ([11]). Stateholders are a generalisation of PCN's definitional variables: they can serve as communication and synchronisation mechanism and can be redirected to another steteholder. The model will allow also full sharing of computation and mixture of lazy and eager evaluation.

4. Conclusions and Further Work

In this paper we have examined two models for parallel multilingual programming based on the shared memory abstraction: those of Linda and PCN. Both models have proved their usefulness when applied to real life problems. Initial attempts to compare the two models ([5,6,7]) where necessarily limited to a model's expressiveness and ease of programming due to differences in the language used and underlying architecture. However, the existence of PCN allows now a proper comparison of the two models to be made. We are

currently implementing a set of typical benchmarks in both models, using C as the common language and a network of RS/6000 as a common underlying architecture. We will be studying such parameters as speed of execution, scalability, ease of programming, program sizes, etc. In parallel, we are designing a model similar to PCN, based on Term Graph Rewriting that in addition to the concurrent logic programming techniques supported by PCN, will also support lazy evaluation, full sharing of computation and allow the programmer to make use of a *process graph oriented* programming style.

References

[1] Chandy K. M. and Kesselman C., "Parallel Programming in 2001", *IEEE Software*, November 1991, pp. 11-20.

[2] Ahuja S., Carriero N. and Gelernter D., "Linda and Friends", *IEEE Computer*, August 1986, pp. 26-34.

[3] Foster I., Olson R. and Tuecke S, "Productive Parallel Programming: The PCN Approach", *Scientific Programming*, 1992 (to appear).

[4] Shapiro E. Y., "The Family of Concurrent Logic Programming Languages", *Computing Surveys*, Vol. 21(3), 1989, pp. 412-510.

[5] Gelernter D., "A Note on Systems Programming in Concurrent Prolog", *IEEE ISLP*, 1984, pp. 76-82.

[6] Shapiro E. Y., "Systems Programming in Concurrent Prolog", *11th ACM SPPL*, 1984, 93-105.

[7] Carriero N. and Gelernter D., "Linda in Context", *CACM*, Vol. 32(4), April 1989, pp. 444-458.

[8] Barendregt H. P., van Eekelen M. C. J. D., Glauert J. R. W., Kennaway J. R., Plasmeijer M. J. and Sleep M. R., "Term Graph Rewriting", *PARLE*, 1987, LNCS 259, pp. 141-158.

[9] Glauert J. R. W. and Papadopoulos G. A., "A Parallel Implementation of GHC", *FGCS*, 1988, pp. 1051-1058.

[10] Papadopoulos G. A., "A Fine Grain Parallel Implementation of Parlog", *TAPSOFT*, 1989, LNCS 351, pp.313-327.

[11] Banach R. and Papadopoulos G. A., "Parallel TGRS and Concurrent Logic Programming", *WP&DP*, 1993.

[12] Glauert J. R. W., Kennaway J. R., Sleep M. R., Holt N. P., Reeve M. J. and Watson I., "Specification of Core Dactl 1", *Internal Report*, Univ. of East Anglia, 1987.