A NEW LANGUAGE FOR MULTICOMPUTER PROGRAMMING

José M. García Carrasco
Dpto. de Informática
Campus Universitario s/n
Universidad de Castilla-La Mancha
02071-Albacete (Spain)

Abstract

In this paper we present an extension of the Pascal language: the Distributed Pascal, suitable for MIMD computers with message-passing. This extension is simple, elegant, flexible and powerful. This language is very suited for numerical algorithms. A complex mathematical algorithm is shown as an example of this. Up to date, we have developed the compiler's front-end and we are running a program in this language by means of simulation.

1. Introduction.

Distributed-Memory machines are playing an increasingly important role in high-performance computing as parallelism is being exploited at a large scale. Since the early attempts to use parallel machines, one of the most critical problems has always been how to program such machines.

A way to solve this problem is automatic parallelization, i.e., from a program written in a sequential language the compiler translates such language into a parallel language. In fact, the compiler must carry out the partitioning, mapping and communication of the different parallel processes. Up to now, this technique is not very developed and serial algorithms parallelizing doesn't work as well as algorithms genuinely conceived for parallel implementations.

The other way is to use a parallel language. In this paper we present a new parallel language for multicomputers: the Distributed Pascal. This language allows an easy and elegant programming of parallel algorithms, specially numerical algorithms. The language is based on a lot of processes which communicate by means of message-passing.

Moreover, a compiler has been developed for this language. This compiler generates an intermediate code, which is executed (by simulation) in the FDP environment [1] that we have also developed.

In the remainder of this paper, we detail the Distributed Pascal, its main concepts and the process communication. We also explain the computer model which the language is based on, and show a complex numerical algorithm programmed in this language. Finally, we draw some conclusions.

2. The Distributed Pascal.

The Distributed Pascal is a new language for MIMD computers with distributed memory suited for numerical algorithms.

This language allows the creation of and communication between parallel processes for a multicomputer. It is independent of underlying architecture, allowing the portability from one multicomputer to another as well as a more flexible environment for work.

The Distributed Pascal is a parallel variant of the original Pascal language [2]. On the one hand Pascal is a very educational language and it is used as a basis for the teaching of programming in many universities. On the other hand, it has enough power and versatility to allow the treatment of a large number of problems. Therefore the Distributed Pascal can be a very suitable language for the teaching and programming of parallel algorithms.

The Distributed Pascal is a parallel language with an imperative style. The parallel-programming languages can be classified into three categories: Algol-based languages, logic languages (parallel Lisp and its dialects) and parallel functional languages. The Distributed Pascal is a language inside the first group, i.e., with an imperative style. We believe imperative languages are more familiar to most programmers. Likewise, these languages are so well established that they will continue to be used, even if the interesting work being done in the areas of logic programming and functional programming languages leads to efficient implementations on parallel machines.

Another feature of the language is that it shows the parallelism explicitly. We think that the programmer and compiler must work as a team to produce good parallel code. The parallelism is handled by means of some extensions which have been added to the standard Pascal.

3. Computer model.

The Distributed Pascal is a language developed for multicomputers. Therefore the only way of communicating and synchronizing the processes is by means of message-passing.

In order to make the language independent from the computer, the number of processes is not limited to the number of processors. With regard to the programming of an algorithm the concept of virtual processor is used, i.e., we assume that there are as many processors as declared processes. It is a function of the run-time kernel to determine which processes are loaded in each processor (mapping). This mapping is static, i.e. once decided which processor each process is executed in, there is no change during the execution of the program.

The communication model is based on message-passing. For the programming the concept of virtual network is employed, supplying a totally connected topology to the user. This concept permits him to communicate a process directly with any other, without taking into account the interconnection network found in a particular multicomputer. It is even possible to apply the concept of virtual network on a shared memory multiprocessor. Again it is a function of the kernel to determine if the source and destination processes of a message are in the same processor and the communication is internal, or execute the router in each processor through which a message passes if the source and destination processes are in different processors, time multiplexing the physical channels of a processor to provide service for all the processes which are being executed in it. This can be done because the mapping is static and known throughout the entire system

The programming style we have adopted is the SPMD one [3], i.e., all the processes execute the same program. In the case several processes are sharing a processor, there is only one code copy in memory, each process having its own variables. This approach reduces memory occupation significantly. However, it does not imply that all the processes execute the same instruccions. In order to be able to execute some code zones only by some processes, the language possesses the variable *process* which identifies each process. So for the process 0 the variable process will equal 0, for the process 1 it will be 1, etc... For instance, if we wished for a part of the program to be carried out only by the process 0 it would look like:

```
(...)
if process=0 then begin
      (...)  (* executed only by the process 0 *)
end;
(...)
```

It is worth comment on two details. Firstly, we could equally use the CASE sentence to discriminate the code between processes as well as IF. Secondly, we can also calculate functions where one of their arguments is *process*. In this case all the processes execute the same instructions but the result will be different to each process.

4. Main features of the language.

4.1 Process creation and termination.

As stated above, the virtual processor abstraction provides a process model which is independent of the number of physical processors in the multicomputer. Each process is identified by its *processid*. The creation of processes is carried out statically (at the beginning of the execution), the number being defined at the program header by means of the constant *processes*. This constant declaration takes the following format:

processes number

where *number* is an unsigned integer indicating how many processes are to be executed in parallel.

The statement of the number of processes is placed inmediately after the statement of the name of the program. It should not appear, it is assumed by default that there is only one process.

No explicit mechanism is offered to handle process termination. In this respect it is like Occam and different from CSP. The processes terminate when the corresponding code has been executed up to the end.

4.2 Process communication.

The communication between processes is done via some primitives: *send*, *receive*, *received* and *broadcast*.

The send primitive is used for sending a message from one process to another. It takes the following format:

send (dest_node, message)

where *dest_node* is the destination process of the message and *message* is the message to be sent, which may be any simple type, a structured one or defined by the user.

Its implementation is asynchronous and unblocking, i.e., the message is sent to a buffer, and it is therefore unnecessary for the sending process to wait until the receiving process can accept the message. This behaviour has been chosen (unlike Occam for example) to give greater freedom to the process and thereby achieve a higher level of parallelism. If in some algorithms synchronization was needed, the user could implement it by means of acknowledgement messages, i.e., with the send-receive pair each time a process sends a message it remains waiting to receive a sign confirming that this message has arrived.

The receive primitive is used to receive a message from another process. It takes the following format:

receive (dest_var)

where *dest_var* is the variable where the received message is kept, which must be of the same type as the variable *message* in the corresponding *send* (the check is made by the compiler).

It is implemented with blocking, i.e., if a process executes the previous statement and still has not received a message, this process is blocked waiting for the arrival of a message. The buffer for message reception has been implemented by means of a FIFO queue. As this primitive does not allow the conditional acceptance of messages depending upon which process is sending the message, it is necessary to ensure that the messages are sent to a process in the same order in which this process expects to receive them. Any implementation of the language must ensure that all the messages transferred between a given part of processes carried in the same order they are sent. However, nothing is ensured about different processes sending messages to the same destination. Then the algorithm behaviour must be independent from the order messages arrive when they come from different sources. Otherwise, there will usually be a run_time error (the data type of a received message is compared with the type of dest_var) or wrong results. Alternatively, the programmer can add a heading to each message showing the source process and the order number if necessary.

The language also includes the Boolean function *received*. When executed by a process this function will give a TRUE result if a message has arrived, or FALSE if not. By means of this function, a process can choose between taking a message from its FIFO input queue or execute other sentences, thus avoiding the blocking produced when *receive* is executed and there are no messages in the input queue (provided there is an alternative code to execute). Likewise, the Occam ALT construction can be simulated.

Finally, in some algorithms it is necessary for a process to send a message to all the other processes. Therefore another pimitive called *broadcast* has been implemented. It takes the following format:

broadcast (message)

where *message* is the message sent to all the other processes.

For exactly the same reasons as for *send*, it is implemented without blocking.

Thanks to the development of the language-compiler binomial the primitives assume a compact format and irksome calls to the system libraries are avoided where the user must add the length and address of the message, its type, etc. (an example of this is the iPSC and its communication primitives for FORTRAN and C [4]).

It should be underlined that in these primitives the communications are established between processes, without taking into account the localization of these processes in the physical processors. In this way one of the requisites of the language is achieved, namely its **independence** with respect to a specific architecture.

4.3 Sequential characteristics of the language.

Currently, The Distributed Pascal adds some small differences to the sequential characteristics of the standard Pascal. With respect to the data types, the Distributed Pascal has the integer, real, Boolean and chart types, also allowing data handling at a bit level by means of *shift, and, or, xor* and *not* operations, although it does not have the enumerated, subrange or pointer types yet.

With respect to the structured data types, it does not support either the variant record or the set. It does, however, the file type with its basic operations (limited at this moment to text file): open, close, read and write. Finally, it does not allow either procedures or functions to be passed as parameters to another procedure.

5. An example: the matrix triangularization.

As an example we are going to present an algorithm to triangularize a sparse matrix. It is based on Givens rotations, which are very suitable for parallel machines thanks to their inherent parallelism. For a deep study of this algorithm see [5].

In synthesis the algorithm is as follows. Given a mxn sparse matrix, Givens rotations are applied to row pairs which have the first nonzero element in the same column, in such a way that the elements below the main diagonal become zero. To achieve this, as many processes as columns in the matrix are executed. Each process has those rows for which the first nonzero element appears in the column corresponding to that process. If we define the type of a row as the column in which the first nonzero element of this row is situated, then type 1 rows will belong to process 1, type 2 ones to process 2, etc.

Given two rows assigned to the same process via Givens rotations, we get one of them to cancel out the first nonzero element that it had, thereby shifting its first nonzero element to a later column. To continue processing this row, the process must send it to the corresponding process according to its new type. By repeating this for all the rows of a process and executing all the parallel processes we achieve the triangularization of the matrix.

Each process either receives the rows sent by other processes or applies the previous calculation to a pair of rows. Since a process can only send rows to later processes (i.e. they work with rows of a higher type), a process finishes when it has processed all its rows and receives a sign token indicating that all the previous processes have finished.

The main program turns out as follows:

```
begin  (* main program *)
load_matrix;
if process<>0 then begin
   if process=1 then endtoken:= true else endtoken:= false;
   repeat
      while received() do begin
         receive(message);
         unpack(n);
         if n=0 then endtoken:= true
         else begin
              index:= index+1;
              matrix[index]:= mat_aux1;
              end;
      end;
      if index>1 then begin
         mat_aux1:= matrix[index];
         index:= index-1;
         rotation(matrix[1], mat_aux1);
         j:= typ(mat_aux1);
         if j<>0 then begin
                          pack(1);
                          send (j, message);
         end;
      end;
   until (endtoken) and (index=1);
   if process<>processes-1 then begin
                                pack(0);
                                send(process+1, message);
                                end;
end;
end.
```

The following procedures and variables are employed.

Load-matrix: The process 0 loads the matrix from a file and distributes the rows according to their type among the other processes.

Pack: Packs the messages sent to other processes.

Unpack: Unpacks the messages received from other processes.

Rotation: Applies Givens rotations to the given rows.

Type: Determines the type of a row.

Index: Gives in each moment the number of rows that a process has.

End-token: Used to indicate end of process. When a process receives this value it knows that all previous processes have finished and therefore is not going to receive any other message.

Message: Variable used to send and receive messages, and may be either a row or the end-token variable.

Finally note that if the matrix has n columns, at the begining of the program we have to add

processes n+1.

as there are as many processes as columns plus a process which takes charge of loading the matrix and initiating the remaining processes.

7. Conclusions and future work.

The typical parallel programming language is at too low-level and is machine dependent. In this article, we have presented a new programming language, Distributed Pascal, an extension of the Pascal, which makes it suitable for the programming of distributed memory MIMD computers, such as iPSC or transputer networks. This language is machine architecture independent and allows us a simple, elegant, flexible and powerful programming on multicomputers.

This language uses the concepts of virtual processor and virtual interconnection network, which allow us a great independence of the developed code with regards to the architecture on which it will be executed later.

The Distributed Pascal has the SPMD programming style. For this reason, it is very suited for programming algorithms where all processes execute the same code aproximately. This language has been used for parallelizing numerical algorithms successfully. In the paper, we have presented the source code for the triangularization of a sparse matrix via the Givens rotations.

As for future work, we are finishing the development of the Distributed Pascal so that it can support all the Standard Pascal structures and data types, as well as the posibility of breaking down the code into modules. Moreover, we are currently about to generate codes for several commercial machines.

References

[1] García, J.M. and Duato, J. "FDP: An environment for a MIMD programming with message-passing". Technical Report GCP #1/91, Departamento de Ingeniería de Sistemas, Computadores y Automática. Univ. Politécnica de Valencia, 1991.

[2] Jensen K. and Wirth, N. *Pascal User Manual and Report.* Springer-Verlag, 1975.

[3] Karp, A. Programming for Parallelism. *IEEE Computer*, pp. 43-57, May 1987.

[4] Moler, C. and Scott, D.S. *Communication Utilities for the iPSC.* iPSC Technical Report nº2. Intel Scientific Computers, Aug. 1986.

[5] García Carrasco, José M. *Desarrollo de herramientas para una programación eficiente de las redes de transputers: estudio de la reconfiguración dinámica de la red de interconexión.* PhD thesis. Universidad Politécnica de Valencia. December 1991.