MODELING INTERPROCESSOR COMMUNICATION IN POLYMORPHIC PROCESSOR ARRAYS

P. Baglietto, M. Maresca, A. L. Frisiani
DIST - University of Genoa
Via Opera Pia 11a - 16145 Genova
Tel. (+39) 10 353.2983 - Fax (+39)10 353.2948
E-mail: prp@dist.unige.it

Abstract - A high level model for interprocessor communication operations in Polymorphic Processor Arrays is presented. The interconnection network topology and the connection autonomy features of a SIMD massively parallel computer have great influence on its computational capabilities. The PPA architecture allows to dynamically reconfigure its mesh interconnection network in order to optimally map onto it any topology required by a parallel algorithm. We have developed a high level model for this interconnection network and expressed the model in terms of procedures available in a function library of the PPC (Polymorphic Parallel C) language, a high level parallel language we have developed for SIMD massively parallel computer programming. We present the PPA architecture, the PPC language, the interprocessor communication model and an example of its use.

Introduction

The interconnection network topology and autonomy is a very, if not the most, important parameter for performance characterization of SIMD massively parallel architectures. For this reason the basic SIMD computational model, which demands an identical behaviour of all PEs (processing elements) via the instructions issued by a central controller, can be enhanced by providing each PE with a certain degree of connection autonomy, which is very useful in parallel computing applications. The connection autonomy can be obtained by means of packet switching or circuit switching. The PPAs (Polymorphic Processor Arrays) are the best known example of implementation of the circuit switching paradigm based on a mesh interconnection network, reconfigurable via a switching device added to each PE and programmed by combining instructions broadcast by the central controller and local conditions.

We present a high level model for the interprocessor communication strategies available on the PPA. This model is expressed in terms of procedures of the PPC (Polymorphic Parallel C) language. The PPC, a parallel language we have developed for SIMD massively parallel computers programming, aims both to be architecture independent and to exploit the specific features of each SIMD architecture, above all its own interprocessor communication capabilities based on the interconnection network topology. This goal has been reached by providing a few extensions to the C language in order to support the distributed allocation of data arrays and the parallel flow control, and a library of functions for each SIMD architecture considered. This library models the specific interprocessor communication functions supported on the architecture. PPC language and interconnection library make-up a high level framework for the development of parallel algorithms and performance evaluation of the interconnection network.

We first briefly describe the PPA features, its peculiar interconnection network and the main PPC extensions to the C language (needed to understand the parallel algorithms presented); then we present the library of functions for interprocessor communication that provide a high level model for the interprocessor communication strategies available with a reconfigurable interconnection network. Finally we give an example of use of the model and some concluding remarks.

Polymorphic Processor Arrays Architecture

Like most massively parallel computers [1][2] Polymorphic Processor Arrays (PPAs) are processor arrays with a mesh interconnection network. In addition to the standard mesh
network, they are provided with some switching devices in order to support connection autonomy [3]. We can describe a PPA, shown in figure 1, as a stack of three planes, respectively called processor plane, or P-plane, memory plane, or M-plane, and switch plane, or S-plane.

The P-plane is a two-dimension array of processors called \( P_{E_{ij}} \), the M-plane is a two-dimension array of memories called \( M_{E_{ij}} \) and the S-plane is a two-dimension array of switch boxes called \( S_{E_{ij}} \) (we assume \( i,j = 0, 1, \ldots, \sqrt{N} - 1 \) where \( N \) is the number of PEs making the mesh). Each set \( \{ P_{E_{ij}}, M_{E_{ij}}, S_{E_{ij}} \} \) is referred to as a PPA node and is called \( N_{E_{ij}} \); in each node \( N_{E_{ij}} \), \( P_{E_{ij}} \) controls the operation of both \( M_{E_{ij}} \) and \( S_{E_{ij}} \) in the sense that it is able to enable/disable their operation, depending on a local condition.

The PPA nodes are connected by means of two sets of busses, traversing the S-plane and connected at the extremes. The first set consists of \( \sqrt{N} \) busses, one for each row of nodes, crossing the switch-boxes along the horizontal direction; analogously, the second set consists of busses, one for each column of nodes, crossing the switch-boxes along the vertical direction.

As usual in SIMD computer, a central program controller broadcasts instructions and data to the P-plane and the memory addresses to the M-plane; in addition to the P-plane instructions and M-plane addresses, in PPAs the program controller broadcasts instructions to the S-plane, to control the data movement in the processor array and the configuration of each switch-box.

The data movement direction is the same for all the PPA nodes, meaning that, at any given time, all the nodes send data in the same direction (North, East, West or South), selected by the SIMD program controller. On the contrary, the switch-box configuration can be different in each PPA node. A different switch-box configuration at each node is programmed by combining the S-plane instruction, broadcast by the SIMD program controller, with a local compute condition, different in each node; the result of such a combination is used to set up the switch-box configuration.

Two switch-box configurations are allowed, called Open and Short: in the Open configuration, a switch-box disconnects the two busses by which it is traversed, and allows its corresponding processor to inject data into one of them; in the Short configuration, on the contrary, a switch box lets the data propagate on the busses, according to the current data movement direction, and prevents its processor from injecting data into the bus system. The cooperative action of the switch-boxes leads to the possibility of dynamically partitioning the two sets of busses of a PPA in a number of independent sub-busses, that group cluster of adjacent nodes.

Polymorphic Parallel C Language

Like many parallel languages [4][5], the PPC language is an extension of a widely used sequential language, in this case the C language.

The main features of the PPC language are two extensions to the standard C language: a new memorization class called parallel and a new control structure called where.

The new memorization class, denoted by the keyword parallel, allows to specify that a variable must be allocated in multiple copies in the local memory of each PE instead of in the memory of the central controller. So a parallel data declaration, as every data declaration in C, consists of a memorization class (usually optional, but required for parallel data declaration), and a type identifier followed by at least one variable name. Available parallel data types are the standard C data type. For example:

\[
\text{parallel float } A;
\]

assigns in the local memory of each PE a memory location for the variable \( A \). As usual in SIMD computers each PE executes instructions involving the variable \( A \) on its local data. In this way the dimension of variable \( A \) coincide with the dimension of the architecture. Adding one or more dimensions in such a declaration we can also define an array in one or more dimensions for each PE. For example:

\[
\text{parallel float } A[4][4];
\]

assigns 16 memory locations in each PE for the variable \( A \). In this way we can manage data arrays at PE level and better exploit parallelism in many data processing problems.

Parallel variables can be managed with functions provided in the interprocessor communication model in the sense that values allocated in the local memories can be exchanged among PEs.

The new control structure is denoted by the syntax:

\[
\text{where (expression) }
\text{instruction group 1;}
\]
and allows to define the PE activity states and to cluster them into two sets. The first set is composed by PEs that verify the expression: the PEs belonging to this set execute only the first instruction group. The second set is composed by all remaining PEs that execute only the second instruction group.

**The Interprocessor Communication Model for Polymorphic Processor Arrays**

The model proposed exploits the PPA reconfigurable network to implement two classes of communication primitives:

- Data transmission among PEs;
- Data processing (namely data reduction via computation to be carried out on arrays of data allocated on the PE array).

The two classes reflect the necessity not only to exchange data among PEs, but also to introduce some mechanisms in order to perform operations on sets of data allocated in multiple copies in the local memory of each PE. This kind of data allocation in the PPC language is carried out through the use of the parallel memorization class for variables. So the communication primitives proposed will be expressed in terms of PPC procedures managing variables of the parallel memorization class.

The first class of communication primitives is concerned with parallel data transmission among PEs, that is with the transmission of a copy of a variable belonging to the parallel memorization class allocated in the local memory of a PE to one or more other PEs. Data transmitted may be stored in another variable of the parallel memorization class.

Three data transmission primitives are proposed: the first primitive is named `shift` and transfers data from a PE to one of its adjacent PEs; the second primitive is named `broadcast` and is a PPA specific communication primitive since it exploits the PPA interconnection network reconfigurability; the third is named `s-broadcast` and is a combination of the two previous primitives.

The `shift` primitive syntax is:

```
shift (A, Dir)
```

where `A` is a variable of the parallel memorization class and `Dir` is one of the four directions allowed for data transmission on PPA (North, South, East, West). Note that in the torus configuration the PE adjacent to $P_{E0,0}$ in the West direction is $P_{E0,\sqrt{2}-1}$.

The `shift` primitive returns a variable, of the same type of variable `A`, which contains the same data contained in variable `A` moved one position in the specified direction; that is each PE receives the value of `A` allocated on the PE adjacent to it in the opposite direction with respect to the `Dir` direction.

The `broadcast` primitive syntax is:

```
broadcast (A, Cond, Dir);
```

where `A` and `Cond` are variables of the parallel memorization class and `Dir` is one of the four directions already defined.

The `broadcast` primitive returns a variable of the same type of variable `A` containing selected data of `A` broadcast along the specified direction `Dir`. The data transmission is carried out according to a configuration of the SEs set by the values of parallel variable `Cond` on each PE, namely a SE is set to the Short configuration if the value of `Cond` is 0 and to the Open configuration if the value is 1 (see example in figure 2). PEs in the Open configuration are named source PEs since they return as value of the broadcast primitive their own value of variable `A`. PEs in the Short configuration are named destination PEs and for them the returned value is the value of variable `A` of the nearest source PE in the direction opposite to the `Dir` direction.

![East data broadcasting with torus configuration](image)

**Fig. 2 - East data broadcasting with torus configuration**

The `broadcast` primitive is architecture specific since it exploits the network reconfigurability in order to complete the transmission operation in one step.

The `s-broadcast` primitive syntax (`s` for shift) is:

```
s_bcast (A, Cond, Dir)
```

The operation carried out is a combination of those carried out by `shift` and `broadcast` primitives. Differently from the `broadcast` primitive the source PEs do not return their own value of variable `A` but that of the nearest source PE in the direction opposite to the `Dir` direction.

The second class of communication primitives is concerned with parallel data processing. Operations on sets of variables of the parallel memorization class are carried out by implementing a communication binary tree for each column or row of the mesh interconnection network, again exploiting the network reconfigurability. The type of data returned depends on the kind of operation requested and can be either parallel or
Three data processing primitives are proposed: \texttt{v\_reduce}, \texttt{p\_reduce} and \texttt{s\_reduce}.

The \texttt{v\_reduce} primitive syntax (\texttt{v} for vector) is:
\begin{verbatim}
v\_reduce (A, Dir, Op);
\end{verbatim}
where \texttt{A} is a variable of the \texttt{parallel} memorization class, \texttt{Dir} is one of the four directions allowed for data transmission and \texttt{Op} is the operation to be carried out on variable \texttt{A}.

The \texttt{v\_reduce} primitive returns a parallel variable of the same type of variable \texttt{A} but with significant values only in the last (along the direction \texttt{Dir}) column or row. The returned value for the PE of every column or row is the result of the computation of the \texttt{Op} operation on all the data associated to variable \texttt{A} in that column or row of PEs. Such a computation is carried out by implementing a binary tree in each column or row of PEs, again using the interconnection network reconfigurability. Figure 3 gives an example of a binary tree summation in the South direction, completed in 3 steps for an \texttt{8x8} mesh. For each step we can distinguish three groups of PE: source PEs that contain data to be summed in couples, destination PEs that receive the temporary results of the computation and inactive PEs corresponding to data already used. For a \texttt{N x N} mesh the number of steps required is \(\log_2 N\).

Available operations are:
\begin{itemize}
  \item \texttt{sum} for data summation on each row or column;
  \item \texttt{max, min} for the search of the maximum/minimum element in each row or column;
  \item \texttt{and, or} boolean operators
\end{itemize}
Other operations may be added to the list.

The \texttt{p\_reduce} primitive syntax (\texttt{p} for polymorphic) is:
\begin{verbatim}
p\_reduce (A, Activ, Dest, Dir, Op);
\end{verbatim}
where \texttt{A}, \texttt{Activ} and \texttt{Dest} are variables of the \texttt{parallel} memorization class, \texttt{Dir} is one of the four directions allowed and \texttt{Op} is the operation to be carried out on variable \texttt{A}.

The \texttt{p\_reduce} primitive returns a parallel variable of the same type of variable \texttt{A} and can be considered as a combination of features of the \texttt{broadcast} and \texttt{v\_reduce} primitives since any PE may contain a significant value. The PEs corresponding to significant values are selected with the variable \texttt{Dest} that allows to define the clusters of PEs that must be reduced in a way similar to that defined by variable \texttt{Cond} in the broadcast primitive. In addition the user can exclude some PEs, and consequently their data, from the computation using variable \texttt{Activ}: a PE is excluded if its value of \texttt{Activ} is 0. An example of maximum search in the west direction is presented in figure 4.

The \texttt{s\_reduce} primitive syntax (\texttt{s} for scalar) is:
\begin{verbatim}
s\_reduce (A);
\end{verbatim}
where \texttt{A} is a variable of the \texttt{parallel} memorization class. The returned value is a scalar value to be stored in the central controller memory. The operation is accomplished through two \texttt{v\_reduce} operations, the first on rows and the second on columns.

\textbf{Application example}

The algorithm presented as a simple example of use of our

\begin{verbatim}
source PE in max_search operation
\end{verbatim}
\begin{verbatim}
destination PE for max_search operation result
\end{verbatim}
\begin{verbatim}
inactive PE in max_search operation
\end{verbatim}
\begin{verbatim}
cluster of PEs from which to extract a maximum element
\end{verbatim}

Fig. 4 - Searching the maximum element for clusters of PEs with torus configuration, West data direction.
model is a procedure that executes the matrix product of two matrices A and B. Such a computation requires, in a sequential environment, a time proportional to $N^3$ because three nested cycles are necessary: two to scan the entire resulting matrix C and the third to compute each element $C[i,j]$ calculating


for all values of the index $k$.

To realize the same operation on the PPA we start from the hypothesis that each processor PE$_{ijkl}$ has in its memory the A$_{ijkl}$ and B$_{ijkl}$ elements of the matrices to be multiplied at the beginning, and the C$_{ijkl}$ element of the resulting matrix at the end of the computation.

At each step $k$ the network executes a shift operation in the south direction of matrix B and then a broadcast operation on the $i$th row of the element A$_{i,j-k}$. In this way, because of the ranging of $k$ from 1 to $N$ and of the shift operations made on B, each processor sees the $j$th column of matrix B in its memory. Besides, on the 'broadcast bus', each PE can read one after the other all the elements of the $i$th row of matrix A. The PPC code of the algorithm described is:

```c
parallel double mat_prod(A, B)
parallel double A, B;
parallel double A-tmp, C = 0;
int i;
for (i=0; i<righe(A); i++) {
  A-tmp = broadcast(/, east, ((Col - Row + i) % righe(A) == 0));
  C += A-tmp * B;
  B = shift(B, south);
}
return (C);
```

From the algorithm written in the previous lines we can see that the steps expected are $N$; therefore in a time proportional to $N$ the computation is completed thanks to the reconfiguration of the network topology available on the PPAs.

### Discussion and concluding remarks

The complexity of the `shift`, `broadcast` and `s_broadcast` primitives is $O(1)$, assuming the constant time propagation model. In such a model the time required for a signal to flow through a chain of switches in the SHORT state is considered negligible with respect to the system clock cycle; the experience done with the Polymorphic Torus chip [6] supports such an hypothesis.

The complexity of the `v_reduce` and `p_reduce` primitives is $O(\log(\sqrt{n})$, again assuming the constant time propagation model. $V_{reduce}$ and $p_{reduce}$ primitives are implemented by executing $\log(\sqrt{n})$ steps, in such a way that at each step $i$ ($i = 0, 1, \ldots, \log(\sqrt{n})-1$) pairs of data are managed together (summed or-ed, etc.) in parallel. Such a method corresponds to the simultaneous embedding of $\sqrt{n}$ trees on the $\sqrt{n}$ rows or columns of a PPA.

Considering that the above primitives are optimum in terms of computational complexity we can state that the interprocessor communication capabilities available on a PPA allow to design and implement optimum algorithms for a number of computational problems as described in [7][8][9][10]. Moreover the high level model of these communication operations allows to design parallel algorithms easy to understand and very concise without additional costs in terms of computational complexity.

### Bibliography