A procedure for the efficient programming of a distributed memory message-passing hypercube multicomputer is presented by E L Zapala*, O G Plata† and F F Rivera†

That is, if we denote as PE(r) the processor numbered (or with the address) r, then the processors PE(r) and PE(s) are neighbours if r ⊕ s = 2^b, for a b such that 0 < b < q, where ⊕ is the bitwise exclusive-OR logical operation. We then say that PE(r) and PE(s) are neighbours along dimension b. Usually, we use the term hypercube computer to refer to a MIMD (multiple-instruction/multiple-data), loosely coupled, distributed memory, message-passing concurrent computer.

The rest of the paper is structured as follows. In the next section we present the problem of the design of parallel programs in multiprocessors with distributed memory and we expound on the procedure we use for programming hypercubes. We then present the parallel language used in order to express the programs developed and the ACLE (Array C Language Emulator) software package for the simulation of the parallel programs in sequential computers.

MAPPING SEQUENTIAL ALGORITHMS ON DISTRIBUTED MEMORY HYPERCUBES

When we address the problem of obtaining a parallel version of a sequential algorithm for its execution in a distributed memory MIMD multicomputer, we can choose between a model using structured or unstructured parallelism. In the unstructured model, or MPMD (multiple-program/multiple-data), each processor will have its own local data and its own local program that will process this data. The programmer is in charge of distributing the data in the local memories apart from designing a different program for each processor. On the other hand, the structured model, or SPMD model (single-program/multiple-data), attempts to mimic the simplicity in the programming of synchronous parallel systems. The SPMD model considers the same local program stored in the local memory of each processor. Each local program will be executed in an asynchronous manner over its own
set of local data. With this restriction of the SPMD model we can approach the design of parallel programs from two points of view. On one hand, we start by distributing the data to be processed on the local memories of the processors following some regular scheme (for example, cyclic or block distribution) and then we construct the program that is going to be executed in each node. The design of this program will be guided by the optimization of some performance function (for example, on the minimization of the total computation time, the minimization of interprocessor communications, the optimization of computational load balancing and/or the minimization of the information redundancy in the local memories). Obviously, the scheme for the data distribution in the local memories strongly influences the performance of the local program. This dependence can force us to execute both steps several times before finding an efficient parallel solution.

The other possibility in the design of a parallel SPMD program consists of exchanging the two previous steps. That is, we start by designing the local program, following, for example, some scheme for the partitioning of the logical structures (procedures, loops, etc) in a previously chosen sequential program. We then proceed with the data distribution on the local memories so that some performance function is optimized. In order to simplify the implementation, it is interesting to restrict the data distribution to regular schemes. In this case we also find an interdependence between the two steps, so that it might be necessary to execute the process several times during the search for a parallel program which is sufficiently efficient.

For each model, we must choose the level of parallelism we want to exploit. We can distinguish between a coarse grain level (procedure level parallelism), a medium grain level (loop level parallelism) and a fine grain level (statement or operation level parallelism). The first level can be used in large sequential programs which can be partitioned into a set of (semi-) independent tasks. It is obviously possible to parallelize each task using a finer level. The exploitation of the fine grain level has a great overhead in asynchronous multiprocessors associated with it, and, therefore is generally used within each processor by means of multiple or different functional units (pipelining). Due to the fact that most programs spend most of their time in loops, it is the exploitation of this parallelism level which is going to produce the largest reductions in processing time.

Independent from the parallelism model and level, the task of designing a parallel program can be the job of a human programmer or an advanced compiler. From this point of view, we can distinguish three approaches. The first considers that a human programmer completely designs the parallel program. Therefore, a language which permits the expression of the parallelism inherent to the program in an explicit way is necessary. Most multiprocessors have some conventional language implemented in them, such as Fortran or C, to which a machine-dependent specific library containing functions that allow the loading of programs in the processors and the host-processor and interprocessor communications have been added. However, the programming of these machines with these languages is tedious and error prone. Programming is easier and more reliable if we use a truly parallel language, which permits expressing the parallelism in a natural way.

In the literature we can find diverse parallel languages which verify these requirements. The most interesting ones include medium and high level structures which facilitate the specification of certain operations, such as the distribution of data in the local memories or the interprocessor communications. Some examples of these languages are DINO\(^1\), C\(^2\) and ACLAN\(^3\,\,^4\) among others. DINO is a data-parallel high level language oriented to SPMD computation, whereas C\(^2\) is a data-parallel extension of the C programming language designed for programming the Connection Machine. With DINO the programmer does not need to explicitly specify interprocessor communications, but he must indicate non-local accesses. ACLAN, on the other hand, is also a superset of C, although the new structures which permit expressing the parallelism are flexible enough to be able, for example, to directly access internal registers of the processors or specify that a given communication is going to be carried out using a specific interconnection link. Despite this possibility of low level programming, ACLAN is a machine independent language. This is achieved because the programmer can define a virtual multiprocessor over which the ACLAN statements act.

The second approach consists of transferring part of the task of designing the program from the human programmer to the compiler. This way we can free the programmer from some of the most complex and tedious tasks associated with the design of a parallel program. This will only be possible if the compiler can be automated and we have efficient techniques to include them in a compiler. Following this line, in order to express our program we need some parallel language which allows us to specify part of the program and an advanced compiler which is sophisticated enough to be able to add the rest. Some languages with these characteristics are Kali\(^5\), Vienna Fortran\(^6\) and Fortran D\(^7\). With these languages the programmer specifies the scheme for the data distribution and the local program. The interprocessor communication statements are added to the local program during the compilation stage.

We would find the third approach at the opposite end to the first one, that is, the parallelization task is carried out completely by the compiler. In this case we would speak of a vectorizing or parallelizing compiler. With this solution it is not necessary to use a parallel language. We simply write the program in some conventional language and then use the compiler in order to translate it to a parallel solution. Clearly, this approach is the most complex from the point of view of the implementation in the compiler. Some examples of compilers (environments) with these characteristics are Parafase\(^8\), PED\(^9\), SUPERB\(^10\) and AL compiler\(^11\).

We have recently developed a procedure for the extraction of the inherent parallelism of a sequential algorithm using the SIMD/SPMD model and the loop level (medium grain)\(^12\). The procedure assumes that the parallel architecture is a distributed memory, message-passing hypercube computer (synchronous or asynchronous). That is, we have combined the extraction of parallelism with the stage of mapping the parallel algorithm on the computer. We only extract the parallelism at the loop level which can be used in a hypercube architecture. Our procedure uses the second design possibility in the SPMD model, that is, we partition the code and then distribute the data as a function of this partitioning. This procedure can be carried out by a
human programmer and, in this way, utilize a parallel language in order to express the resulting program. We have followed this approach, obtaining efficient parallel algorithms in the fields of matrix algebra and image and signal processing. We are currently developing automatic techniques in order to include this procedure in a compiler. In the following sections we describe the steps of this procedure.

Loop-level analysis of the sequential algorithm

This analysis tests the data and control dependences permitting, in this way, detection of the loops that can be parallelized. In order to carry out this step tools such as those included in Parafrase-2 or PED can be used.

Partitioning the hypercube dimension and mapping the code

Once the nested loops which can be totally or partially parallelized have been found, we establish a partition of the dimension of the hypercube computer. The number of sets into which the dimension of the hypercube will be partitioned will coincide with the maximum depth of the parallelizable nested loops. That is, we define a map of the index space associated with the loops onto the dimensional space of the hypercube. With this partition, a given set of processors will execute specific pieces of the parallelizable loops. It is possible, however, to choose a partition of the dimension of the hypercube into a number of sets smaller than the depth of the parallelizable loops. With this selection we will increase the redundancy of the data, but we will possibly reduce the interprocessor communications. In order to describe the partitioning procedure, suppose that our original sequential algorithm has the following nested loops:

$$\text{for } (I_0 = 0; I_0 < N_{I_0}; I_0++)$$
$$\text{for } (I_1 = 0; I_1 < N_{I_1}; I_1++)$$
$$\quad \ldots$$
$$\text{for } (I_{k-1} = 0; I_{k-1} < N_{I_{k-1}}; I_{k-1}++)$$
$$s (I_k-1, \ldots, I_0)$$

where $$s (I_k-1, \ldots, I_0)$$ is a block of assignment sentences. Each iteration of the nested loop can be considered as a point $$(I_k-1, \ldots, I_0)$$ in a k-dimensional N-element loop-index space S, where $$N = N_{I_0} \cdots N_{I_{k-1}}$$. Our task is to distribute the $$(I_k-1, \ldots, I_0)$$ (and the data structures they involve) among the $${Q = 2^k}$$ PE of our q-dimensional hypercube. We assume for the moment that $$q > k$$, and that there is no data dependence among the $$(I_k-1, \ldots, I_0)$$ (though each block may have internal data dependence).

Faced with the above problem, we partition the q dimensions of the hypercube in k sets of q dimensions ($$i = 0, \ldots, k-1; q = Q_{k-1} + \ldots + Q_i$$) that are consecutive according to the binary PE indexation scheme. The index i of PE($$r$$) can thus be represented by a k-dimensional vector$$r = (r_0, r_1, \ldots, r_k)$$ in which $$r_j$$ is the value of bits $$q_j$$ to $$q_{j-1}$$ of the binary representation of $$r$$; in other words, $$r = r_0 + r_1 2^{q_0} + \ldots + r_{k-1} 2^{q_0} + \ldots + q_0r_1$$. Each PE($$r$$) will now be assigned the $$(I_k-1, \ldots, I_0)$$ corresponding to the D points of a k-dimensional loop-index space L$$_S$$ constructed from a subset of S, where $$D = d_k-1 \ldots d_0 d_i = \lfloor N_i/2^k \rfloor$$ is the size of the ith dimension of L$$_S$$. For a complexity-wise optimal partition $$(q_k, \ldots, q_0, D)$$ is minimum. With block processing distribution, $$s (I_k-1, \ldots, I_0)$$ is executed as sentence block $$(P_k-1, \ldots, P_0)$$ of the L$$_S$$ whose indexing vector $$(r_k, \ldots, r_0)$$ is given by the expressions $$r_i = \lfloor I_i/d_i \rfloor$$, where $$p_i = I_i \mod d_i$$ ($$i = 0, \ldots, k-1$$). With cyclic processing distribution, $$s (I_k-1, \ldots, I_0)$$ is executed as sentence block $$(P_k-1, \ldots, P_0)$$ of the L$$_S$$ whose indexing vector $$(r_k, \ldots, r_0)$$ is given by the expressions $$r_i = I_i \mod 2^k$$, where $$p_i = I_i/2^k$$ ($$i = 0, \ldots, k-1$$). If $$q < k$$, the situation is slightly complicated by our having to associate more than one dimension of S with some of the subsets of the hypercube dimensions.

If the effects of the $$(I_k-1, \ldots, I_0)$$ are not independent of those of previous iterations of the original nested loops, i.e. if the $$(I_k-1, \ldots, I_0)$$ are logically or semi-logically ordered, then the appropriate levels of the PE nested loops will have to include collateral data transfer loops in which the relevant information is transmitted to and received from the other PEs involved in obtaining the results. The logical order refers to data dependence itself, and the semi-logical order is associated with the nature of the electronic processors, i.e., the accumulation of a sum of many numbers must take place addend by addend, but without there being any logical priority among the addends. With a suitable choice of the processing distribution scheme, semi-logical ordering has no influence on the degree of concurrency that can be achieved. In such cases, the original sequential algorithm can be re-arranged by removing the semi- logically ordered sentences from the $$(I_k-1, \ldots, I_0)$$ and placing them all together in a second loop nest executed after the main nest. If we assume, for simplicity, that the hypercube is as big as the set of loop indices, then the non-accumulating, unlogically ordered remains of the $$(I_k-1, \ldots, I_0)$$ can be made totally concurrent; and since the size of the semi-logically ordered accumulation loop (M, say, in the sequential algorithm) can be reduced to log$$_2$$M on the hypercube, the performance of the algorithm is only marginally worse than in the case of global total concurrency. However, if there is total logical ordering of the iterations of any of the loops of the original sequential loop nest, then it is impossible to make these iterations concurrent, and this loop must be retained in the hypercube program.

In the literature we can find other schemes for partitioning nested loops. Two of the most efficient ones are OPTAL$$_{11}$$ and SECAL$$_{16}$$ (SECAL is an optimized OPTAL), which can generate optimal processor assignments if the loop bounds are known at compile time and there are no data accessibility problems. This situation can be reached in the ideal case of a computer with shared memory and without any interference in data access. OPTAL obtains partitions for any number of processors between 1 and a reference value of P. In the specific case of a hypercube, it would be sufficient to obtain the optimal partitioning for a processor number which is a power of 2. In this sense, we could use an algorithm such as FINAL$$_{16}$$. The problem with these algorithms is that the performance function they minimize is the total net computation time, that is, they do not consider delays in data accessibility. For a multiprocessor with distributed memory this function is not valid, except in the ideal case where there are no communications. We must add to the computation time the time spent in interprocessor communications. Also, we must consider other factors such as the regularity of
the scheme for the data distribution on the local memories and the redundancy of the information.

**Data distribution on local memories**

The previous step distributes the code among the nodes in a regular manner, so that all the processors execute the same code (partitioned) but over disjoint sets of data. The next step will consist in the data distribution on the local memories of the processors. This distribution should be carried out so that the processors find most of the data they need in their local memories, minimizing, therefore, the number of interprocessor communications necessary for transferring the data a processor needs and which is located in the local memory of another processor. The most usual schemes for the distribution of the data are the block and cyclic schemes or variations of these. The distribution of sparse matrices is a special case. Furthermore, in order to reduce interprocessor communications, an appropriate indexing scheme for the PEs must be chosen (pure binary, gray etc). The most usual indexing scheme is the pure binary one.

Taking as a reference the nested loop mentioned above, in general, the data structures involved in the nested loop will not each have as many dimensions as the space $S$. In the matrix multiplication $A[U][V] * B[V][W]$, for instance, $S$ is three-dimensional but the product matrix and the two multiplicands are each affected by only two of these dimensions. In such cases, parallel processing involves data storage redundancy: if matrix $A$ involves dimensions $0$ to $t-1$ but dimensions $t$ to $k-1$, for example, then with cyclic processing distribution the $(l_0, \ldots, l_{t-1})$th element of $A$ will have to be stored in the $(\lfloor l_0 2^{k-t} \rfloor, \ldots, l_{t-1} 2^{k-t})$th position of matrices $A(t_0, \ldots, t_{t-1})$ in the local memories of all the $Q_0, \ldots, Q_{t-1}$ PEs for which $t_i = l_i \mod 2^k$ ($i = 0, \ldots, t-1$), where $Q_i = 2^k$.

**Parallel program design**

Construction of the complete parallel program. This algorithm is the result of steps 2 and 3, adding the communication messages for the interprocessor transference of the non-local data and for the necessary synchronization of the processors in certain key points. We will also have to add the communications for the host-processors transferences (introduction of initial data and extraction of the final results). The design of the parallel program can be automated.

**Optimization**

Optimization of the parallel algorithm obtained in the previous step. By means of the measurement of performance parameters, such as speedup, efficiency, data redundancy, load balancing, among others, we establish the optimization factor of the parallel algorithm. If it is not optimum enough according to our criteria, we modify the partition of the hypercube chosen and/or the data distribution scheme.

We have satisfactorily tried this procedure for the SPMD/SIMD programming of hypercubes with a multitude of algorithms. In Table 1, we present some of these algorithms together with their characteristics such as the partitioning chosen, the level of nesting of the loops, the data distribution scheme chosen, the indexing of the nodes and the algorithmic complexity, both of the sequential algorithm and the parallel algorithm with the optimum number of PEs.

With this procedure we have designed a complete parallel software package for detection, classification and 3D reconstruction of biological macromolecules.

**PROGRAMMING MESSAGE-PASSING, DISTRIBUTED MEMORY HYPERCUBES**

The procedure of mapping algorithms onto message-passing, distributed memory hypercube computers described in the previous section permits the design, in a systematic manner, of parallel versions of sequential algorithms. Clearly, we need a language which permits expressing the parallel algorithm obtained in a way in which the parallelism extracted is explicit. The language we use for this task is an extension of sequential C language called ACLA (Array C Language). ACLA is a machine-independent SIMD/SPMD parallel programming language, which permits the explicit expression of the parallelism of the problem. Among its most important characteristics are the possibility of specifying internal operations of each processor at a low, medium and high level (from transfers between processors to operations with more complex data structures). It is also possible to specify an interprocessor transference through a specific interconnection link. The possibility of the programmer defining a virtual multiprocessor on which the ACLA statements act permits the independence of the ACLA language from the specific architecture of a real computer. In fact, by adequately defining this virtual multiprocessor, we have developed an implementation of the language on the NCUBE/10 hypercube computer, using a SPMD programming model.

ACLAN allows the programmer to work with the storage elements contained in the PEs and in the host of the hypercube, both internal registers and the local memory. This is available by giving a name to each storage element. If the storage element is composed of several locations, each location is specified by dimensioning the name associated with it. For example, if $R$ is the name of a 128-location RAM memory, $R[1][2]$ and $R[1][4]$ represent two locations of the RAM memory. In the first case we are accessing the memory as if it were a linear array, whereas the second case assumes that the memory is structured as a two-dimensional matrix. We can refer to a group of locations of a storage element by underdimensioning its name and/or by specifying a range of values in one or more of its dimensions. For instance, if we consider the above-mentioned 16*8-location RAM memory, $R[1][1:2], R[1:4][2:8]$ and $R$ represent the second and third columns (the first one is number 0) of $R$, the fifth, seventh and ninth rows of $R$, and the complete RAM memory, respectively. There is a special register, named #, included in each PE which stores the logical index identifying it. In addition to most of the C operators, ACLA includes the bit operator (.:) and the set operator (in:). The first allows the extraction of a bit or a range of bits from an integer value. For example, #.0 returns 1 for odd PEs and 0 for even ones. The second operator tests if a value is included in a given range. For example, $R$ in 4:8
Table 1. Characteristics of parallel algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Partition (nesting loop level)</th>
<th>Data distribution</th>
<th>Indexing</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCM\textsuperscript{17}</td>
<td>3 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 \text{ NPC}</td>
</tr>
<tr>
<td>kD-FFT\textsuperscript{10}</td>
<td>k (k)</td>
<td>Block</td>
<td>Pure binary</td>
<td>1 \text{ NPC} n + p + c</td>
</tr>
<tr>
<td>MMR\textsuperscript{19}</td>
<td>3 (3)</td>
<td>Block</td>
<td>Pure binary</td>
<td>1 N\textsuperscript{k}</td>
</tr>
<tr>
<td>MM\textsuperscript{14}</td>
<td>3 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>Cholesky\textsuperscript{14}</td>
<td>2 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 N\textsuperscript{3}</td>
</tr>
<tr>
<td>HKM\textsuperscript{20}</td>
<td>3 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>LU\textsuperscript{21}</td>
<td>2 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>kD-FHT\textsuperscript{22}</td>
<td>k (k)</td>
<td>Block</td>
<td>Pure binary</td>
<td>1 \text{ NPC} n + p + c</td>
</tr>
<tr>
<td>FBP\textsuperscript{23}</td>
<td>3 (3)</td>
<td>Block</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>WZ\textsuperscript{24}</td>
<td>2 (3)</td>
<td>Folded block</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>CCC\textsuperscript{25}</td>
<td>4 (4)</td>
<td>Shifted block</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>HT\textsuperscript{26}</td>
<td>2 (3)</td>
<td>Balanced cyclic</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>QR\textsuperscript{27}</td>
<td>2 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>2D-FCT\textsuperscript{28}</td>
<td>2 (3)</td>
<td>Cyclic</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>LT\textsuperscript{29}</td>
<td>2 (3)</td>
<td>Block</td>
<td>Pure binary</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>FDP\textsuperscript{30}</td>
<td>3 (5)</td>
<td>Block</td>
<td>Gray</td>
<td>1 N\text{LM}</td>
</tr>
<tr>
<td>FEP\textsuperscript{31}</td>
<td>1 (4)</td>
<td>1D way stripes</td>
<td>Gray</td>
<td>1 N\text{LM}</td>
</tr>
</tbody>
</table>

\(x = \log_2 X\)

FCM: Fuzzy c-means; N, P, C: number of input data, features, clusters.

kD-FFT: k-dimensional fast Fourier transform; \(N_i\): number of data along dimension \(i\).

MMR: Markovian model reliability; N, T: number of states and time.


Cholesky: Cholesky decomposition; N: dimension of the input square matrix.

HKM: Hard k-means; N, P, C: number of input data, features, clusters.

LU: LU decomposition; N: dimension of the input square matrix.

kD-FHT: k-dimensional fast Hartley transform; \(N_i\): number of data along dimension \(i\).

FBP: Filtered back-projection; N: dimension of the 3D volume.

(*) The complexity \(O(1)\) is obtained with \(N^3\) PEs if all of the PEs contain a copy of the projection.

returns 1 if the content of \(R\) is greater than or equal to 4 or lower than or equal to 8, and 0 in any other case, whereas \(R\text{in 4:8:2 returns 1 if the content of } R \text{ is 4, 6 or 8, and 0 in any other case.}

In ACLAN we can distinguish two types of executable sentences, scalar and parallel. The former, written directly in C, are executed by the control unit of the SIMD computer, or by the host in the case of a MIMD computer. Their function is to process global data and to control the program flow. The latter are in charge of processing the data distributed in the processing elements or performing the distribution of this data. The parallel processing of the data is carried out in the PEs, whereas their distribution is performed in the control unit (SIMD computer) or host.
The simulation package ACLE (Array C Language Emulator) tries to distribute the content of H on the process parameterized by the virtual array processor. It consists basically of a translator, a library of simulation routines and a virtual multiprocessor. Depending on the physical position of these memory locations, there are local actions (where the data transference is restricted to each PE), remote actions (or routing transfrences between different PEs) and central actions (or data transfrences between the host, or control unit, and the PEs in either direction). The first two actions are executed in the PEs, whereas the last one is executed in the host or control unit.

We will now discuss the differences among the various types of parallel actions in the following three examples:

\[
R[3][1] := R[1][2] || #0;
R[3][1](\text{neigh}[0]) < = = H I[1]^l!#.01; \nn R[3][1] < = = H[1]^l!#.0; 
\]

where we consider the 16*8-locations of local RAM memory called R. Because of the mask all the parallel actions are executed only by the PEs with even index. The first action expresses the copy of the first half of the third column of R on the fourth row of R (the second half of that column is ignored because there is no room for it). This action is executed by all the even PEs on their own local memory. The next sentence expresses the same action but affecting the local memories of different PEs. Each even PE sends a copy of the first half of the third column of R to one of its neighbouring PEs. There the copy is stored in the fourth row of R. The receiving PE is specified by the predefined vector \text{neigh}[1]. \text{neigh}[0] indicates that the routing transference uses the links associated with the interconnection function number 0. The definition of the interconnection functions is a part of the complete description of the virtual multiprocessor. In the last sentence H is the name of a storage element in the host. This sentence tries to distribute the content of H on the non-masked PEs. If H is a linear array, H[0] is sent to PE(0) and stored in R[3][0]. Then H[1] is sent to PE(0) and stored in R[3][1], and so on. When the column R[3] is completed, the next element, H[8], is sent to PE(2) (PE(1) is masked) and stored in R[3][0]. If the operands R[3] and H are interchanged, the central action expresses the reverse operation.

**EMULATING HYPERCUBE PARALLEL PROGRAMS**

The simulation package ACLE (Array C Language Emulator) allows a program written in ACLAN to be executed on a sequential computer. It consists basically of a translator, a library of simulation routines and a virtual multiprocessor block (see Figure 1). A source program written in ACLAN is converted by the translator into a standard C program, a process parameterized by the virtual array processor.

![Figure 1. Structure of ACLE](image)

During translation, scalar executable statements are left intact, while parallel executable statements are replaced by calls to the library sequential routines simulating the particular action involved in the statement. While the ACLAN parallel statement expresses an action that is executed by means of a certain group of PEs, the above mentioned sequential routine expresses the very same action, this time executed by the same group of PEs but one after the other. This is to say, the implicit parallelism associated with the ACLAN statement is transformed into a loop whose index moves along the logical identifications of the group of PEs that execute the action.

For efficient parallel programming, the programmer must have the possibility of direct control over local memory and inter-PE communications. The programmer will work with a conceptual scheme of the inter-PE communication network and the intra-PE structures that are accessible to him. This conceptual scheme constitutes a virtual multiprocessor. In order to allow programs to be written which can be run on any machine, the programmer can define the virtual multiprocessor he has in mind. Intra-PE definitions include definitions of local memory and internal registers of the PEs. Local memory is defined by specifying, for each required memory component, an identifier associated with a data type indicating the kind of values to be held in the component. The local memory definitions conform to the syntactic rules of C, including the new data type 'bit', that allow a memory element of a given number of bits to be defined. This data type is useful for defining mask and flag registers.

An inter-PE communication definition begins by declaring the maximum number of PEs required. ACLAN programs will be able to work with any number of nodes, but always less than the number declared in this section. The required inter-PE communication network is specified by defining a number of interconnection functions (IFs). An IF assigns to each PE one of its neighbours. Figure 2 shows an example of a virtual multiprocessor definition composed of a hypercube interconnection network with a maximum of 1024 PEs (Figure 2a) and a local memory and registers definition corresponding to the Cholesky L* L decomposition algorithm (Figure 2b).

**ACKNOWLEDGEMENTS**

This work has been partially supported by grants TIC88-0094, MIC88-0549, MIC90-1264-E and TIC90-0407 of the CICYT and XUGA20604A90 of the Xunta de Galicia.

Microprocessors and Microsystems
#16

#1F (~node) = hypercube( i, node ),
   i in 0:3;

long hypercube( i, pe )
short ~,
long pe;
{
   return (pe ^ (1<<i));
}

double R1, R2, R3;
double LL[N][N];
bit MASK, RM1, RM2;

Figure 2. Example of definitions for an: a, interconnection network; b, intra-PE

REFERENCES

1 Seitz, C L ‘The cosmic cube’ Comm. ACM Vol 28 No 1 (January 1985) pp 22–33
5 Plata, O G ‘An array processing language for multiprocessor systems’ PhD Thesis, Department of Electronics, Faculty of Physics, Univ. of Santiago de Compostela (1989) in Spanish
8 Zima, H, Brezany, P, Chapman, B, Mehrtra, P and Schaudl, A Vienna Fortran – A Language Specification Austrian Center for Parallel Computation, University of Vienna, Austria (1992)
28 Buján, J R, Plata, O, Rivera, E L, Pena, T F and Zapata, E L 'Fast cosine transform on hypercube computers' Proc. Int. Conf. on Parallel Computing and Transputers Applications (PACTA), Barcelona, Spain (1992)
29 García, L, Garzón, E M, Cabaleiro, J C and Zapata, E L 'Parallel tridiagonalization of symmetric matrices based on Lanczos method' Proc. Int. Conf. on Parallel Computing and Transputers Applications (PACTA), Barcelona, Spain (1992)
31 Zapata, E L and Pena, T F 'Finite-element solution of the non-linear Poisson equation on hypercube computers', to be published

Emilio L Zapata is Professor in the Department of Computer Architecture at the University of Málaga, Spain. He received his PhD degree in physics from the University of Santiago in 1983. His major interests are in the area of parallel computer architectures, parallel algorithms, VLSI and fault-tolerant computing.

Oscar Plata is an Associate Professor in the Department of Electronics and Computation at the University of Santiago de Compostela, Spain. He received BS and PhD degrees in physics from the University of Santiago de Compostela in 1985 and 1989, respectively. His research interests include supercomputing architectures, parallel processing, parallelizing compilers and parallel languages.

Francisco F Rivera is an Associate Professor in the Department of Electronics and Computation at the University of Santiago de Compostela, Spain. He received his BS degree in 1986 and his PhD degree in 1990 in physics from the University of Santiago. His research interests include supercomputer architectures and parallel processing.