A System for Data-Driven Programming of Multicomputers

Jean-Luc Gaudiot

Department of Electrical Engineering - Systems, University of Southern California, Los Angeles, CA 90089-2562, USA

Giorgio Ventre

The Tenet Group, Computer Science Division, Department of EECS, University of California, Berkeley, and International Computer Science Institute , Berkeley, CA 94704-1105, USA

Abstract: Distributed systems are a promising solution to increase the computational power and fault-tolerance capabilities currently available in traditional computer architectures. While it is technologically possible to integrate large numbers of processors to form a single parallel machine, new approaches to the programming of such machines are needed. Indeed one of the major problems is to offer a programming model independent from the physical architecture and topology of parallel systems. The data-driven approach seems to be a good candidate for such a model, but requires an implementation able to hide the architectural complexity of a multicomputer. In this paper we show how we applied these principles to a Transputer-based parallel system and the characteristics of the resulting programming environment.

1 Introduction

Different solutions have been proposed in order to exploit parallelism and fault-tolerance capabilities of distributed systems. The proposed architectures range from shared memory multiprocessor systems to tightly coupled, distributed memory multicomputers and, more recently, clusters of computers connected by high speed communication networks [16]. Multicomputers appear a very promising approach toward parallelism since they offer very interesting performances combined with crucial features such as scalability and modularity of the architecture [3]. While existing technology enables system designers to increase computing power by integrating multiple processors in a parallel computer, a different software approach altogether must be taken in order to offer a *scalable* programming environment in which the programmer will not need to be concerned with the physical

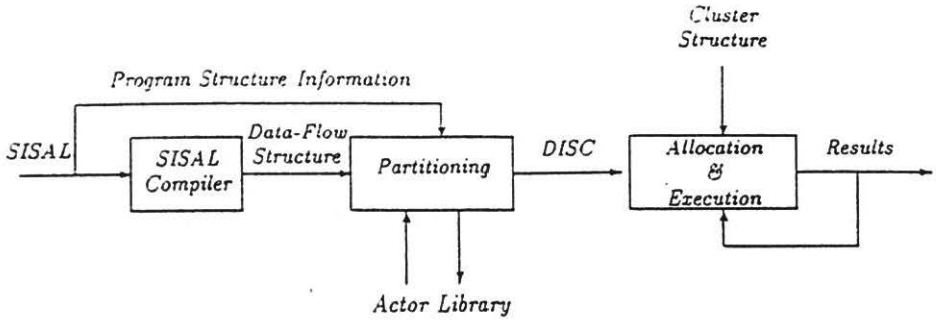


Fig. 1. The Programming Environment

configuration of the machine. We believe that for a distributed programming environment to be successful, an architecture independent programming interface should be offered. In addition to simplify the development of parallel programs, architectural independence of code is needed to assure portability of applications among different machines and support of heterogeneous architectures. The data-flow principles of execution [4] provide such an environment since they allow the distribution of the sequencing mechanism over all the instructions of the program. While many projects (see [6]) involve the design of special data-flow Processing Elements, we have, in this projects, applied data-flow principles of execution to a network of existing microprocessors (Inmos Transputers). In this paper, we briefly describe the programming environment and emphasize the performance results obtained on a number of numerical applications.

2 The Programming Environment

The system consists of up to 16 mesh-connected Transputers and an additional Transputer connected to the host computer. Each Processing Element is a TMS T800 with 4K bytes of on-chip memory and 2M bytes of off-chip memory. In order to achieve the high programmability of the system, we have designed a complex functional programming environment which automatically translates SISAL into DISC. Fig. 1 is the overview of the software environment. The output of the SISAL compiler, IF1 (Intermediate Form 1), is essentially a high-level data dependency graph which contains information concerning the original structure of the user's program. A high-level partitioning of the original program is made, based on the program structure as well as on heuristics [8].

In addition to the basic features of the system which have been developed in our previous work [8], [9], [7], we will present in this section the files which are generated during the translation process and several newly developed features of the system.

2.1 The SISAL Language

SISAL (Streams and Iterations in a Single Assignment Language) [15] is the high-level data-flow language which has been used in the course of this research. This language has also been chosen for many multiprocessor systems, such as the University of Manchester

```
type OneDim = array[integer];

function Aadd(A,B:OneDim;N:integer
returns  OneDim)

for i in 1,N
      c := A[i] + B[i];
      returns array of c
end for
end function
```

Fig. 2: A SISAL Function

data-flow machine [11]. Since SISAL is a single assignment language, it greatly facilitates the detection of parallelism in a program. A SISAL program comprises a set of functions. The input and the output of the program are passed through a main program which is one of these functions. Figure 2 is a SISAL function which adds two arrays.

Note that according to the SISAL grammar, the left-hand side of the assignment statement must be a variable name, either a simple variable or an array name. For an array name, e.g. statement `c := A[i] + B[i]` in the above example, the statement `returns array of c` is used to obtain the entire value of the array.

2.2 Translation from a data-flow language

A SISAL program can be translated to generate an IF1 (Intermediate Form 1) graph [18] by the SISAL compiler. Our translator then translates the IF1 graph into DISC code by creating the following intermediary files:

- PSG (Program Structure Graph) and DFG (Data-Flow Graph): This graph file contains a combined graph of PSG and DFG. The structure information is carried by the *compound nodes* while the dependency information is carried by *simple nodes*. A compound node can be considered a control point which affects a sequence of actors in its range. On the other hand, the simple node is the elementary processing actor; it consists of the input and output area.

- PDFG (Partitioned Data-Flow Graph): Based on the PSG and DFG, a basic partitioning process is performed to lump those simple nodes that have potentially high communication costs [9].

- Communication cost matrix: This file describes the communication costs between partitions. According to the number of available PEs, the interconnection network, and the communication cost matrix, a partitioning process is performed and a new communication cost matrix is generated.

- Allocation information: This file is generated after the optimization phase. It provides the information to indicate the location of the PE where a proper process should be allocated.

- Macro instruction: According to the PDFG, each simple node is translated into a macro instruction which contains the actor code, arc information and partition information. Applying a macro definition table, macro instructions can be expanded to an object program.
- DISC program: This is the final object code. Each process of the program corresponds to a basic data-flow actor.

2.3 Structure handling

In a pure data-flow system, data structures are viewed as single values which are defined and referenced as units. The entire structure must be passed to each referencing actor. Obviously, this can impose a large overhead. Therefore, several schemes have been developed in the past, in order to reduce the overhead of transmitting data structure values [1], [2], [4], [5].

The method adopted to handle arrays in this system is similar to that used in the Hughes Data-Flow Machine [10]. As opposed to the complex system of heaps [4] or I-structures [2], we have chosen the simplified option of von Neumann arrays which are never updated until it is determined that no more read accesses will be made to the current value of the array. Only then, can the array be modified and become a new array. This sequence of reads followed by one write is compiler-controlled. This method brings the very important advantage that no complex mechanisms are needed to ensure the safety of array operations. This comes at the expense of possible compiler-induced loss of parallelism.

2.4 Function calls

In the data-flow scheme, a *function call* can be considered as an actor which requires a function name and arguments on its input arcs to generate results.

When the function and the calling process are located on the same PE, the calling scheme in occam can be expressed as follows:

$$function\_name\,(argument1, argument2, \ldots, result1, result2, \ldots)$$

As in other languages, the *call* actor receives arguments and passes them to a procedure, named *function_name*, to generate results. This scheme can be implemented easily, but may have a lot of parallelism. In this scheme, to call a function, the calling process has to wait until all results have been completely generated. Moreover, the function cannot be called by processes which are located on remote PEs.

In order to allow the parallel execution of a function, the function and calling processes must be located on different PEs. In this scheme, the communication between calling process and function requires external channels. When a function call is made, i.e., a *call* actor is fired, the calling process just passes arguments to the specified function through an external channel, the next process which is not waiting for the results of the function can be executed continuously. On the *function* side, once the input arguments of the function have been received, the specified operations are executed, and the results are sent back to the calling process through an external channel. The major problem of this scheme is that a function cannot be executed in parallel when several calling processes are calling this function simultaneously. However, we can duplicate the function body to achieve a higher degree of parallelism.

2.5 Forall construct and loop unrolling

In IF1, *FORALL* is a compound node which contains a range-generator, a block which actually performs the operations, and a gather node. The body of the loop can be executed in parallel since this construct insures that there are no data dependencies between two iterations of the loop.

Our approach consists in using the concept of loop unrolling [17], a very efficient optimization approach for array operations, in which the data are split among the PEs, processed within these PEs and gathered by the main processor to form the result structure. The loop-unrolling controller controller sends the data through channels to all remote loop-bodies, and collects the partial result generated by each unrolled loop-body to form the final result.

3 The DISC Language

The DISC (DIStributed C) language [12] is a concurrent language that borrows mechanisms to manage concurrency and communication from the CSP model. According to CSP, in DISC each computation is described by a set of entities, called processes, each of which, in turn, represents a computation. Processes run in parallel and interact by means of message passing. The activity of each process can be accomplished by a set of jointly operating parallel processes, according to a hierarchical, recursive structure.

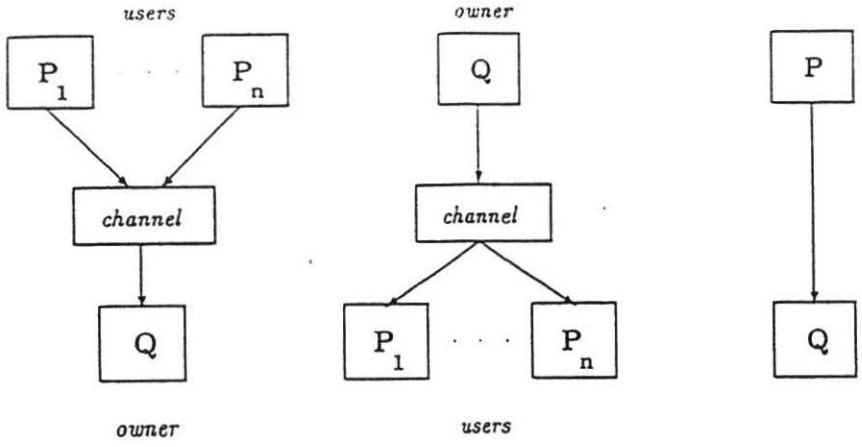
For the sequential part of the processes, the DISC concurrent language adopts syntax and semantics from the C language, as they are defined in [13]. As to the extensions for the management of concurrency, the original CSP constructs have been partially modified. The main changes involve the communication mechanisms. In CSP communication is between pair of processes, in one direction, synchronous (i.e. both the processes must be ready for the communication to take place) and it requires explicit specification of the partners' identities (fig. 3.a). In DISC many-to-one communication channels have been introduced. DISC channels link many processes (called the *users* of the channel) to a single process (called the *owner* of the channel), as is shown in fig. 3.b .

The mechanism is still monodirectional and synchronous, but channels might be referred to in both input and output commands. In the DISC syntax an input command is expressed as `channel ?? variable` while an output command is expressed as `channel !! expression`.

An interesting feature of the DISC implementation of the CSP model is the exception handling. In the original model, when a process requests a communication that cannot be executed since the partner process is no longer active (i.e. it has been already completed), that process is automatically terminated with an exception, by means of a mechanism called distributed termination. In DISC the programmer is allowed to explicitly manage such exceptions, by using a particular construct, called *on fail specification clause*. This feature can be used to cope with changes in the state of the processes composing a program. As an example, we consider the following output command:

```
channel !! variable
     on fail alternate_channel !! variable;
```

In this example, if none of the processes connected to `channel` is active, the exception clause is activated and an alternative solution is attempted using `alternate_channel`. The same mechanism can be used to manage exceptions in parallel commands.



a) DISC b) CSP

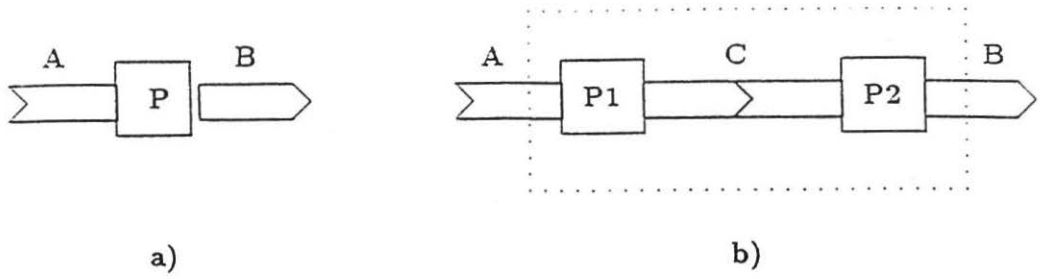
Fig. 3. DISC channels versus CSP channels

In CSP, the alternative command is constrained to contain only input commands in the guards to allow its efficient distributed implementation. In DISC, the distinction between users and owner of a channel has allowed the introduction of output commands in the guards without loss of efficiency. In the alternative command the `on fail` clause can also be used.

To control the activation and termination of processes, DISC uses the parallel command. According to the CSP model, the number of processes to be activated must be known at compile time and no kind of recursive activation is allowed. However, parallel commands may contain processes which contain in turn other parallel commands. This gives rise to the possibility of nesting parallel commands, as it is implicitly allowed in the original model, but not permitted in most of the existing implementations of the model. A DISC program can thus be viewed as a hierarchy of processes, that can be described formally as a digraph (called activation tree) in which the nodes can represent both processes and parallel commands.

The introduction of nested parallel commands has required a further extension to the semantic of the communication mechanisms, to allow message passing also among processes belonging to different parallel commands. The solution adopted, which is named channel inheritance, allows a parent process to pass its channels to the child processes, in a way transparent to its partners. In this way interaction is possible between processes at different levels in the program activation tree. An additional mechanism, called input/output variables, is provided by the language to permit data exchange between a process and its childs activated by means of a parallel command.

Channel inheritance is informally illustrated in fig. 4. In fig. 4.a is shown a simple process, P, which can communicate with other processes by means of two channels, A and B. Process P might actually hide a more complex software structure; for example it could



a) b)

Fig. 4. Parallelism Incapsulation in DISC

contain a a parallel command which activates two other processes *P1* and *P2*, connected by another channel, *C*.

Thanks to the possibility of inheriting channels from the parent process *P*, *P1* and *P2* can effectively interact with all the processes that are allowed to communicate with their parent. In our example, fig. 4.b, process *P1* inherits channel *A*, while process *P2* inherits channel *B*.

4 The Reasons for a Choice

The choice of DISC as the low-level language for the implementation of this prototype depends on linguistic and implementative characteristics that this language shows with respect to other available solutions, such as Occam.

Occam [14] is the native CSP-based programming environment for the Transputer family of components and has been used for a previous research [9]. It is very efficient due to the fact that it is the native language for this kind of processors. In fact Occam is available only on Transputer based parallel systems. This represents a major problem for what concerning portability of developed software on different multicomputers.

If compared to DISC, Occam does not offer some linguistic features which have improved the characteristics of our system. Indeed the availability of many-to-one channels, the possibility of nesting parallel commands and the channel inheritance mechanism have been shown very effective in simplifying the translation process we presented in a previous section.

4.1 An Example

We now illustrate the translation mechanism for an important kind of SISAL construct: a vector operation. In fig. 1 we have shown the SISAL code for the sum of two arrays. This code can be seen as a data-flow actor (see fig. 5.a), where the inputs are the two arrays and their size, and the output is the resulting sum array. The DISC code resulting from the translation procedure is shown hereafter [1]:

[1] For the sake of clearness the actual code has been slightly modified

```
process AADD(N,A,B,C)
chan int N, A[N], B[N];
chan int C[N];
::
par{
        local index, elem1, elem2,result;

        process RANGE(N,index);
        process ARRAY(A,index,elem1);
        process ARRAY(B,index,elem2);
        process ADD(elem1,elem2,result);
        process GATHER(result,C);
}
endprocess

process RANGE(N,index)
chan int N;
chan int index;
::
int counter;
N ?? counter;
for(counter=0;counter<index;counter++)
       index !! counter;
endprocess

process ARRAY(arr,index,out_array)
chan int arr[N],index;
chan int out_arr[N];
::
int _arr[N],_index;
arr ?? _arr;
while(TRUE)
{
        index ?? _index;
        out_arr !! _arr[_index];
}
endprocess

process ADD(elem1,elem2,result)
chan int elem1,elem2;
chan int result;
::
int _elem1,_elem2;
while(TRUE)
{
        elem1 ?? _elem1;
        elem2 ?? _elem2;
        result !! _elem1 + _elem2;
```

```
}
endprocess

process GATHER(result,C)
chan int result;
chan int C[N];
::
int _C[N],counter=0;
while(TRUE)
{
        result ?? _C[counter++]
              on fail C !! _C;
}
endprocess
```

Function Aadd has been translated in one macro-actor process and four elementary actor processes. Process `AADD` is the macro-actor. It activates five child processes by means of a parallel command: RANGE, ARRAY (two instances), ADD and GATHER.

The interface of AADD with the outside world is composed of four channels. Three channels (`N`, `A[N]` and `B[N]`) are used to receive the dimension and the arrays to sum, while channel `C[N]` is used to send out the result array. Channels `index`, `elem1`, `elem2`, `result`, used by processes activated by AADD to communicate locally among themselves, are defined in the `local` statement and declared in the parameter list passed to each process.

Process `RANGE` task is to produce the sequence of integers that will be used as index for the arrays by the two instances of the `ARRAY` process. Processes RANGE and ARRAYs inherit one channel each from AADD (`N`, `A[N]`, and `B[N]` respectively) by declaring the channel name in the process parameter list.

Process ADD is a simple *Plus* actor; it is very general, since it does not depend on any data specification existing in other processes. This result has been accomplished by using the distributed termination mechanism: in fact, once process ADD is activated, it starts to accept inputs values on channels `elem1` and `elem2`, and to send the computed sum out on channel `result`. This task continues indefinitely until anyone of the input/output commands in the process cannot be executed. In our example, this happens when the two ARRAY processes terminate, which, in turn, will happen when process RANGE will terminate. Process GATHER, instead, uses exception handling to wait for the completion of the entire program before communicating the result array on channel `C`.

The software architecture of this set of DISC processes is shown in fig. 5.b . Due to the incapsulation provided by the nesting of parallel commands, the internal structure of process AADD is totally hidden to other processes that might need to communicate with it.

4.2 An Actor Library

By using the abstraction mechanisms we presented above, a number of simple, albeit general, SISAL actors might be translated to form an actor library. Following is the DISC code for a *Times* actor.

```
process MULT(elem1,elem2,result)
```

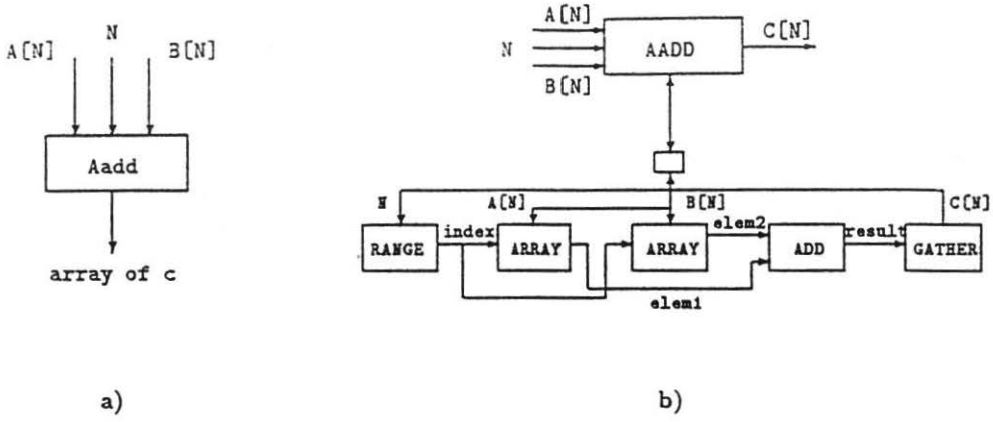


Fig. 5. Software Architecture for the Aadd function

```
chan int elem1,elem2;
chan int result;
::
int _elem1,_elem2;
while(TRUE)
{
        elem1 ?? _elem1;
        elem2 ?? _elem2;
        result !! _elem1 * _elem2;
}
endprocess
```

For example, assume we need to perform the function (a + b) X (c + d). A very simple DISC translation can be implemented by using library actors ADD and MULT, as is shown below:

```
process ADD_MULT(elem1,elem2,elem3,elem4,result)
chan int elem1,elem2,elem3,elem4;
chan int result;
::
while(TRUE)
{
        par{
                local local1, local2;

                process ADD(elem1,elem2,local1);
                process ADD(elem3,elem4,local2);
                process MULT(local1,local2,result);
```

```
        }
}
endprocess
```

Since also process ADD_MULT is totally independent from any input or output data specification (with the exception of data type), it is a good candidate for being included in the actor library.

However, the modularity and incapsulation capabilities owned by DISC are not sufficient to allow the creation of a library. Indeed two major problems must still be solved to fulfill this goal. The first problem is how to map the software architecture of channels and processes of a DISC program onto a hardware architecture composed of processing elements communicating through a network of physical links. In other words, the use of already developed actors could be dramatically limited if their code depends on a particular mapping scheme or network topology.

The second problem is efficiency. In multicomputers, the ratio between the parallelism achievable in the computation and the amount of required communication is an important parameter to evaluate the efficiency of the implementation of a distributed application on a particular architecture. Consequently, both code expressedly developed and the one available in form of libraries should allow the adaptation of this ratio to the characteristics of the software and hardware architecture of a parallel system. In the next section we show how we satisfied such demanding request for architectural independence of programs.

5 The Run-Time Environment Architecture

An important feature of the DISC language is the independence of a DISC program from the topology of the underlying hardware architecture. In Occam, the programmer has to specify in the source code the allocation of the processes on the processors composing the network. He must also directly cope with the routing of messages between processes allocated on non contiguous processors. Consequently, any modification to the software allocation scheme must be reflected in the code, resulting in a very limited flexibility and portability of Occam programs even on machines which differ only in the topology. On the contrary DISC requires no specification of process placement and explicit management of message routing within the communication network, since these tasks are accomplished by the run-time support architecture we implemented for the DISC language.

The run-time environment architecture (RTE) of our prototype is based on an abstraction we called virtual processors (VPs). A virtual processor is a special process associated to each DISC process (DP) defined at source level. The DISC compiler translates all the high-level concurrent constructs to be executed by a DISC process into requests to the RTE. The associated virtual processor receives these service requests and interprets them carrying on the corresponding actions. A VP maintains a data structure representing the state of the associated DISC process and interacts with the VPs associated to the other DISC processes composing the concurrent program.

To improve efficiency, virtual processors related to DISC processes allocated onto the same node, are actually implemented through a single sequential process (we name this process multi virtual processor, or, in short, MVP). This solution reduces the overhead generated by both context switching and message passing.

According to this structure, means have to be provided to MVPs for interacting with the DPs allocated onto the same node, and MVPs allocated onto different nodes. The

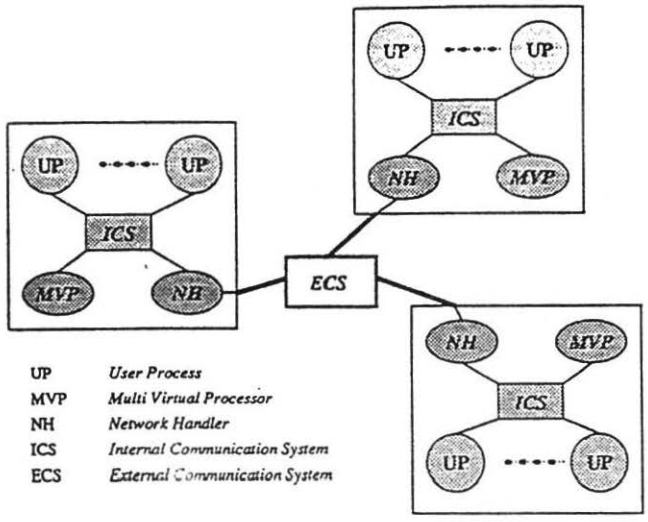


Fig. 6. The Run-Time Environment Architecture

two kind of interactions, named internal and external respectively, are kept well distinct both conceptually and in the implementation. They rely on different mechanisms called internal communication subsystem (ICS) and external communication subsystem (ECS).

Cooperation among DPs and corresponding MVP (i.e. through ICS), is achieved by means of simple asynchronous (bufferized) communication primitives. Cooperation among MVPs allocated onto different nodes (i.e. ECS) is achieved through a delivery system which consists of as many processes as the network nodes. These processes are named network handlers; their task is to hide the details of the communication media connecting the nodes and provide for the transmission, the receipt and the routing of the messages exchanged. The MVPs communicate with network handlers through ICS.

In fig. 6 the whole architecture is shown. The thin lines represent internal interactions, whereas thick lines represent external interactions.

It should be noted that, to let virtual processors communicate among themselves, informations are needed about the concurrent structure of the program and the allocation of processes. In fact, MVPs should be able to determine whether the virtual processor to which a message has to be send, is allocated onto a different node or not. In the former case, the message must be yielded to the NH for delivery through the internode communication system. On the contrary, in the latter case, no message has to be sent and only local actions must be undertaken. Allocation informations are read by the processes composing the run-time environment during an initialization phase, and can be supplied by the programming environment just before starting the execution of the application program.

The RTE has been coded in C, and most part of it is portable on different hardware. The modules that need to be changed are the ones that implement ICS and ECS. The former depends on the operating environment running on each node, whereas the latter

depends on the physical characteristics of the network connecting the nodes. Hence, generally, ECS depends on both the kind of communication media utilized and on the interconnection topology. Different ECS modules have been developed for a number of common topologies that can be created with Transputer processors (i.e. mesh, folded mesh, pipeline).

6 Performance Evaluation

We have chosen to directly evaluate the performance of our system by observing a certain number of test cases. This was done using our Transputer multiprocessor architecture.

6.1 Experiments and experimental results

In order to verify the correctness of the translator and to evaluate the performance of the optimization schemes, we measure the *speedup*, ratio of the execution time of a program on a single Transputer over the execution time of the same program on multiple Transputers. The unit of execution time in measuring is a tick, 64 μsec. The different data allocation methods described previously are also applied.

1. Livermore Loops: Two array sizes, 1000 and 50000, are used in each loop. The *locally distributed* data allocation method has been applied. Fig. 7, and Fig. 8 show the experimental results of loop1, and loop7.

2. Histogram (A program for histogramming): In this experiment two different sizes of digit, 1000 and 50000 are applied to 16 slots. Slots are evenly distributed to each Transputer. The data allocation method is *locally replicated*. Fig. 9 shows the experimental result.

3. MMULT (Matrix Multiplication): In this experiment, we compare two different sizes of matrices, 16 $\times$ 16 and 64 $\times$ 64. Data of one matrix is *locally distributed*, while data of the other matrix is *locally replicated*. (see Fig. 10).

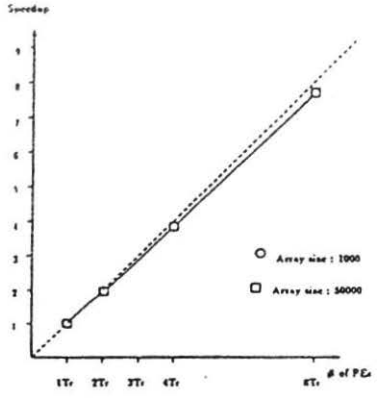


Fig. 7. Speed-up for Loop1

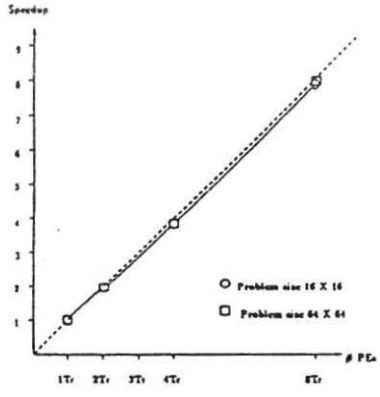


Fig. 8. Speed-up for Loop7

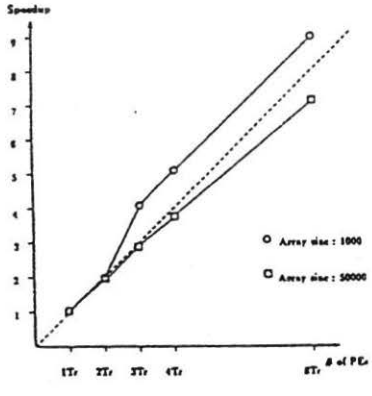


Fig. 9. Speed-up for histogramming

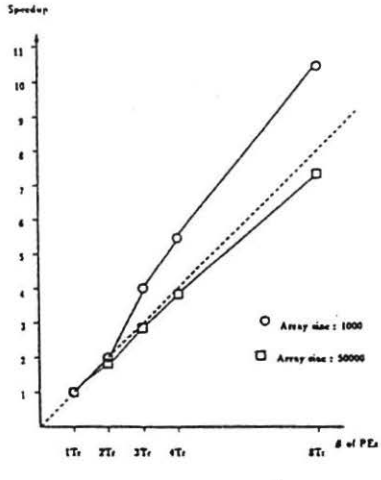


Fig. 10. Speed-up for matrix multiplication

6.2 Interpretation of results

Here we will describe the issues that are concerned with the result of the above experiments.

Topology of the network: The transputer network is mesh-connected. However, for loop unrolling implementation purposes, we look at a hierarchical of the network. The host processor will act as a dispatcher, splitting the data and sending it to several processors as explained in section 2.4.

Speedup: The measured speedups have to be analyzed separately for the different problem sizes. Note that the Transputer owns an on-chip memory of 4Kbytes which is three times faster than the off-chip memory. This feature caused some interesting results to occur.

- **Large problem size:** The required data size is too big to fit in the on-chip memory, even after the partitioning. In this case the speedups obtained are close to linear, since execution time is proportional to the amount of data to be processed.
- **Small or intermediate problem size:** The required data size is greater than the capacity of the on-chip memory, but it will fit in the on-chip memory after unrolling the loop. In this case a superlinear speedup may occur, since the operation in an unrolled loop needs less memory access time that will shorten the entire execution time.

Computation/communication equilibrium: Depending on the ratio of the computation time over the communication cost of a problem, one can observe good performance even if some of the data are required to reach a distant processor to be processed. As one can observe in the LOOP1 case, the system achieves a superlinear speedup when two Transputers have been used, the speedup remains superlinear even if data have to perform a second hop to reach their processor. There is a similar behavior in the LOOP7 case. When more computation time is needed, as in LOOP1 and LOOP7 cases, the better performances can be achieved when using 8 PEs, where the computation cost is much greater than the communication costs. On the other hand, LOOP12 shows a different behavior. Having less computation requirement, the system can achieve a superlinear speedup as long as the data fits in the on-chip memory and needs only one hop to reach its target. However, the performance degrades when some data needs two hops in order to reach its assigned processor, the fourth Transputer, since communication costs are now greater than computation costs.

6.3 Discussion

From the above experiments we have concluded the following:

- As mentioned in section 2.4., the actors of vector operations are under the control of a *forall* compound node in IF1. Since vector operations are easily detectable in IF1, improvement by loop unrolling came at low compiler cost.
- In order to decrease the communication overhead for array operations, according to properties of the application programs, different types of data allocation are needed. For the *locally replicated method*, it does not affect the ratio of data size stored in on-chip and off-chip memory, *i.e.*, the speedup is not affected by the memory access time. However, in the *locally distributed method* the data size distributed to each PE is proportionally decreased to match the increase of the number of available PEs. Thus the ratio of the data size stored in on-chip and off-chip memory is increased by increasing number of PEs.

- According to the experiments and the speedup analysis in the previous section, when the problem size is relatively large the actual speedup will approximate the linear speedup. In fact, it is worth processing in parallel only if the problem is large enough.

7 Conclusions

Our research efforts as described in this paper have focused on demonstrating a practical approach to provide high programmability to the user of a homogeneous, asynchronous MIMD architecture. The results we have shown point to the high *scalability* of the data-driven approach to multiprocessor programming. Indeed, the benchmarks we have used have all been shown to exhibit a linear speed-up as the size of the machine increases.

Our experiments have also shown how crucial are on multicomputers the adoption of efficient allocation schemes and the influence of the computation/communication equilibrium on the performances achievable from parallel implementation of applications. However, the characteristics of the run-time environment architecture of our prototype allow the development of modular and scalable code, independent from the allocation policy that will be adopted at run-time. We believe that these features can dramatically reduce the work and the time required for tuning applications to different network topologies and allocation schemes.

In the future, more sophisticated algorithms for efficient allocation and partitioning of the programs and more benchmark programs must be applied to evaluate the performance of the system.

References

[1] Arvind and R.A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the* 10^{th} *Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 1983.

[2] Arvind and R.E. Thomas. I-structures: An efficient data type for functional languages. Technical Report LCS/TM-178, MIT, Laboratory for Computer Science, June 1980.

[3] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 24(8), 1988.

[4] J. B. Dennis. First version of a data flow procedure language. In *Programming Symp.: Proc. Colloque sur la Programmation*, pages 362–376, Paris, France, April 1974. Springer-Verlag, New York. B. Robinet Lecture notes in Computer Science.

[5] J-L. Gaudiot. Structure handling in data-flow systems. *IEEE Transactions on Computers*, C-35(6):489–502, June 1986.

[6] J-L. Gaudiot and L. Bic. *Advanced Topics in Data-Flow Computing.* Prentice Hall, 1991.

[7] J-L. Gaudiot, L. T. Lee, and P. Aubrée. Data-driven approach for programming a transputer-based system. In *Proceedings of the 1990 Spring COMPCON*, pages 94–99, 1990.

[8] J-L. Gaudiot and L.T. Lee. Multiprocessor systems programming in a high-level data-flow language. In *Proceedings of the European Conference on Parallel Architectures and Languages, Eindhoven, The Netherlands*, June 1987.

[9] J-L. Gaudiot and L.T. Lee. Occamflow: A methodology for programming multiprocessor systems. *Journal of Parallel and Distributed Computing*, August 1989.

[10] J-L. Gaudiot, R. Vedder, G. Tucker, M. Campbell, and D. Finn. A distributed VLSI architecture for efficient signal and data processing. *IEEE Transactions on Computers*, C-34(12), December 1985.

[11] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Data-Flow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[12] G. Iannello, A. Mazzeo, C. Savy, and G. Ventre. Parallel software development in the disc programming environment. *Future Generation Computer Systems*, 5(4), 1990.

[13] B. W. Kerninghan and D. N. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, New York, 1988.

[14] INMOS Ltd. *Occam2 Reference Manual.* Prentice Hall, Cambridge, 1988.

[15] J.R. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J.R.W Glauert, I. Dobes, and P. Hohensee. SISAL-Streams and Iterations in a Single Assignment Language, Language Reference Manual, version 1.2. Technical Report TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.

[16] S. Mullender. In S. Mullender, editor, *Distributed Systems.* ACM Press, 1989.

[17] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Mass., 1988.

[18] S. K. Skedzielewski and John Glauert. IF1: An intermediate form for applicative languages reference manual, version 1. 0. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.