PEPE: A Trace-Driven Simulator to Evaluate Reconfigurable Multicomputer Architectures*

José M. García[1], José L. Sánchez[2], Pascual González[2]

[1] Universidad de Murcia, Facultad de Informática
Campus de Espinardo, 30071 Murcia, Spain
jmgarcia@dif.um.es
[2] Universidad de Castilla-La Mancha, Escuela Politécnica
Campus Universitario, 02071 Albacete, Spain
{jsanchez,pgonzalez}@info-ab.uclm.es

Abstract. Recent research on parallel systems with distributed memory has shown that the most difficult problem for system designers and users is related with the interconnection network. In this paper, we describe a programming and evaluating tool for multicomputers, named PEPE. It allows the execution of parallel programs and the evaluation of the network architecture. PEPE takes a parallel program as input and generates a communication trace obtained from this program. Next, PEPE simulates and evaluates the behaviour of a multicomputer architecture for this trace. The most important parameters of the multicomputer can be adjusted by the user. PEPE generates performance estimates and quality measures for the interconnection network. Another important feature of this tool is that it allows us to evaluate networks whose topology is reconfigurable. Reconfigurable networks are good alternatives to the classical approach. However, only recently this idea became the focus of much interest, due to technological developments (optical interconnection) that made it more viable. A reconfigurable network yields a variety of possible topologies for the network and enables the program to exploit this topological variety to speed up the computation.

1 Introduction

The growing demand for high processing power in various scientific and engineering applications has made multiprocessor architectures increasingly popular. This is exemplified by the proliferation of a variety of parallel machines with some diverse design philosophies. This diversity in architectural design has created a need for developing performance models and simulators for multiprocessors, not only to analyze the effectiveness of a design, but also to reduce the design time.

Distributed memory multiprocessors (often called multicomputers) are increasingly being used for providing high levels of performance for scientific applications. The distributed memory machines offer significant advantages over their shared memory counterparts in terms of cost and scalability, but it is a

* This work was supported in part by CICYT under Grant TIC94-0510-C02-02

widely accepted fact that they are much more difficult to program than shared memory machines. As a result, the programmer has to distribute code and data on processors himself, and manage communication among processes (or tasks) explicitly.

Simulators provide many advantages over running directly on a multiprocessor, including the cost effectiveness of workstations, the ability to exploit powerful sequential debuggers, the support for non-intrusive data collection and invariant checking, and the versatility of simulation. Several simulators as Pie [12] or Paret [11] have been developed. Usually these projects are aimed at developing programming environments and tools for the programmer, rather than tools designed to evaluate the architecture of the system. Newer high-performance simulators such as TangoLite [4] or Proteus [2] consider the architectural support of the system showing several results about the performance of parallel machines.

A very interesting way of simulation is to evaluate the behaviour of an architecture using a trace taken from adequated algorithms. Trace-driven simulations, which evaluate network performance on actual communication streams taken from characteristic programs, are the most reliable way for network design evaluation. These simulations require a great computation power, because of the many different design possibilities that must be simulated, and because of the length of the communication traces that drive the simulation. We have developed a simulator to evaluate the main features of the interconnection network, because in this way it can more faithfully represent the hardware implementation, taking into account details like channel multiplexing, partial buffering and delays in blocked messages. Furthermore, we are very interested in evaluating some new features in parallel machines, mainly reconfigurable architectures, that is, multicomputers whose networks can change their topology dynamically.

In this paper we describe our environment called PEPE (this acronym stands for **P**rogramming **E**nvironment for **P**arallel **E**xecution). PEPE provides a user-friendly visual interface for all phases of parallel program development, i.e. parallel algorithm design, coding, debugging, task mapping, execution control and evaluation of some architecture parameters.

Our environment is different from previous work because of two major features. First, PEPE allows us to evaluate the network for a communication trace taken from proper scientific problems we have previously coded in the environment. That is, we can evaluate and modify the behaviour of the interconnection network for real problems and not only for predetermined workloads. Second, it allows us to evaluate the performance of a multicomputer with a reconfigurable interconnection network. A completely connected interconnection network can match the communication requirements of any application, but it is too expensive to build even for a moderate number of nodes. Reconfigurable interconnection networks are alternatives to complete connections. This paradigm is suitable with either electronic or optical interconnections, which are applicable to a large class of networks. Some researchers believe that the immediate goal in the development of computer networks should be hybrid optical-electronic systems which combine the advantages of both electronic and optical technologies while avoiding their

disadvantages. Reconfigurable networks are especially suitable when these technologies are combined.

Up to now, we only know another related environment for parallel programming on reconfigurable multicomputers. It is described in [1]. This environment is devoted to phase-reconfigurable programs, that is, programs must be implemented as series of phases, and each phase is assumed to be separated from another one by synchronization - reconfiguration points. These points select the adequated topology for each phase. Additionally, this tool is language-oriented. Our environment focuses on testing the different parameters of dynamically reconfigurable networks. A dynamically reconfigurable network means that a network can vary its topology arbitrarily at runtime. In this approach, any arbitrary topology is allowed, so that the interconnection network can easily match the communication requirements of a given algorithm. In our environment, we can study in depth the main concepts and options about dynamically reconfigurable networks, their limitations and tradeoffs.

The rest of the paper is arranged as follows. In section 2 the overall environment is described and its different parts are shown. The programming style and several issues related to it are detailed in section 3. The structure of the traces is discussed in section 4. In section 5, we present the network simulator and explain the main results we can obtain with it. Finally, we outline some conclusions.

2 An Overview of PEPE

In this section we are going to present PEPE. The key features of this environment are simplicity and completeness. Our main goal has been to obtain a flexible system which allows us an easy and efficient way to evaluate the multicomputer reconfigurable architecture.

Our environment has been developed on a workstation using C-language. PEPE provides a user-friendly visual interface for all phases of parallel program development and tunning. This graphical interface (figure 1) has been designed with the aim of keeping it really comfortable to the user, following the styles adopted nowadays by most of the human-oriented interfaces [9].

In our environment, the user gets tools for easy experimentation with both, different parallelization possibilities and different network parameters. With this methodology, the programmer can analyze very quickly several parallelization strategies and evaluate these strategies with tools for performance analysis.

PEPE simulates the execution of a parallel algorithm at two levels. At the first level, we are interested in verifying the behaviour of the parallel algorithm and studying the different strategies of parallelization, as well as the problems that arise when the parallel algorithms are coded and executed, such as deadlock, livelock, etc. For this, the execution of the parallel algorithm is simulated over a virtual architecture. At the second level, the behaviour of the reconfigurable network is studied. For this, we start with the communication pattern produced by the parallel algorithm and the performing of the network for this communication pattern is simulated. At this level, PEPE allows us to use different parameters

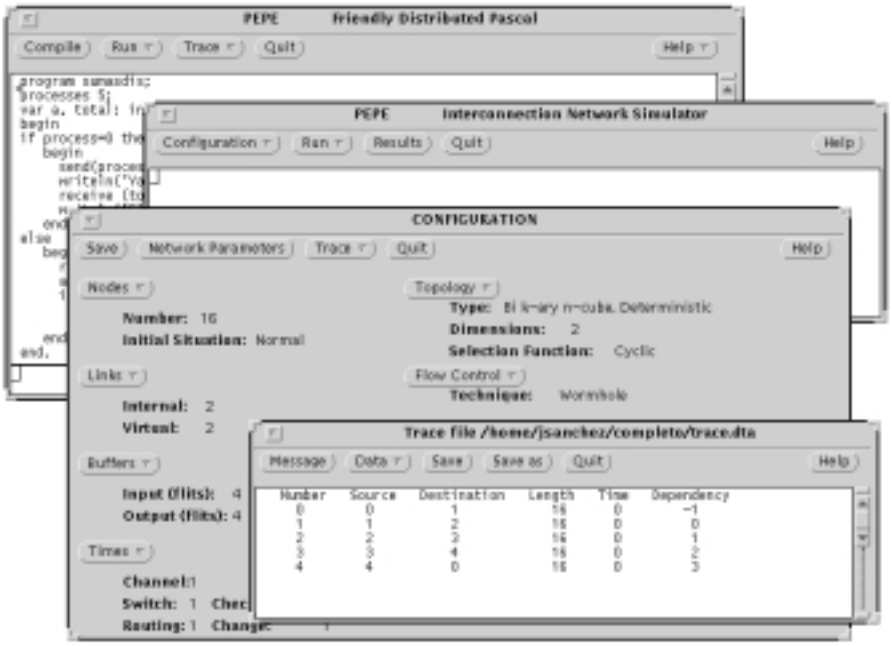

Fig. 1. PEPE's graphical interface

for the reconfigurable network. These levels give rise to the two phases of the simulator.

PEPE has two main phases and several modules within it. The first phase is more language-oriented, and it allows us to code, simulate and optimize a parallel program. In this phase, interactive tools for specification, coding, compiling, debugging and testing were developed. This phase is architecture independent. The second phase has several tools for mapping and evaluating the reconfigurable architecture. We can vary several network parameters such as interconnection topology and routing algorithm. In figure 2 we show the modules of PEPE. The link between the phases is an intermediate code that is generated as an optional result of the first phase. This intermediate code is a trace of the communication pattern of the source program. This allows the user to use the environment as a whole or each phase singly. For example, we can execute only the first phase for testing the parallel behaviour of an algorithm on an ideal multicomputer. We can also obtain an intermediate code from a key parallel algorithm. Then, we can execute several times the second phase from this trace with different network parameters to evaluate and tune the network for this key algorithm.

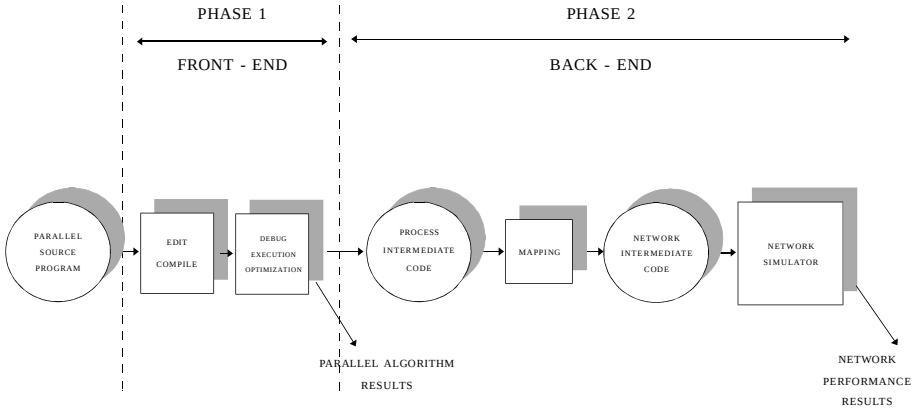

Fig. 2. Modules of PEPE

3 The Programming Tool

In this section we outline some important features of the first phase of PEPE. In this module we try to overcome the difficulty of programming multicomputers with an integrated approach and virtual concepts. The programming tool aims at increasing programming productivity. It takes a user parallel program as its input and tries to simulate its parallel execution on a virtual multicomputer. In this way, the user can test his parallel algorithm and, in case (s)he is not satisfied, come back to code a new parallel version.

In this phase it is important to abstract from specific architectural details such as topology, number of processors, etc. This module supports the concepts of virtual processor and virtual network. The user must take this into account, since the correct execution of a program must not depend upon the topology of the interconnection network.

Usually, the parallel programming style for most of these systems corresponds to the SPMD model [10], in which each processor asynchronously executes the same program but operates on distinct data items. PEPE uses the SPMD model and an extension of Pascal for coding parallel algorithms. This parallel language [7] which we have developed, is based on standard Pascal with some extensions to allow an easy and elegant programming of parallel algorithms, consisting of processes which communicate by means of message-passing. Finally, PEPE uses static scheduling. All processes must be created before starting execution.

For debugging parallel programs, PEPE's environment offers a parallel debugger. With this debugger the programmer gets a global view of the parallel system. Some features are breakpoints, the inspection of program states, displaying the

contents of data structures and the state of each process or the modification of the contents of data structures.

4 The Communication Trace

At the end of the first phase, we can optionally generate an intermediate code (or trace) to evaluate the network performance for the parallel algorithm. That is, the network performance can be evaluated from the communication pattern obtained from a parallel algorithm. The trace records the set of messages that must be sent through the network. As we will see, this trace is independent of the timing parameters of the network.

The trace contain a complete information for each message. It consists of five fields whose meaning is detailed below. Additionally, each message has a message identifier not included in the trace. It is equal to the row number where it appears in the trace.

a) Source Process. Indicates the process that sent this message.
b) Destination Process. Indicates the process for which the message is destined.
c) Message Length. This field indicates the number of data bytes. It does not include control information.
d) Injection Time. Indicates the instant at which the message was injected into the network. This value is given in clock cycles (clock frecuency is a simulator parameter). This value can be absolute or relative as detailed below.
e) Dependency. This field indicates if a message is dependent on the reception of another message or independent. If it is independent, the value of this field is -1; otherwise, its value indicates the identifier of the message on which it depends.

Next, we are going to explain the last two fields in detail. When a parallel program starts execution, some processes send messages. Upon reception of those messages, some processes perform some computations, eventually sending more messages. Thus, there are two types of messages in a parallel algorithm: dependent and independent. A message is independent of any other message if a process can send this message without having to wait for the arrival of another message. On the opposite side, a message m is dependent on another message m' when a process p receives message m', performs some computations and sends message m. This dependency arises either because message m makes use of the information contained in message m', or because process p has no way to reach the instruction that sends message m without receiving message m' before, even if m does not use the information in m'.

Please, note that in many cases this dependency cannot be statically resolved by the compiler, and it is necessary to wait until execution time to know which ones are the dependencies between messages. Thus, communication traces must be generated during the simulation of the execution of the parallel algorithm.

Figure 3 shows the code for a process in three different cases. In the first case, the message is sent independently of any other message, because the process does

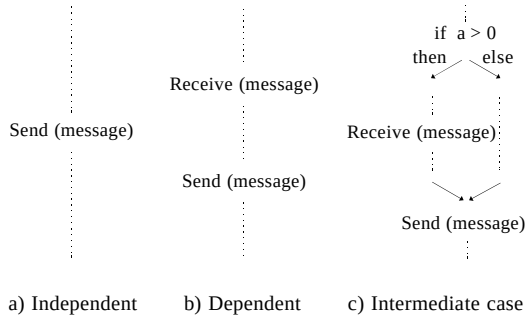

Fig. 3. Types of messages: dependent and independent messages

not need to receive any other message to execute the send instruction. Case b) shows a dependent message; the process has to receive a message. After processing it, the send instruction is executed. In case c), the dependency between messages is determined at run time. In this case, depending on the value of the variable a, the message to send will be dependent or independent. As the communication trace of the algorithm is generated during the simulated execution, this trace will always be correctly generated for the different input values.

The value of the field that contains the time at which a message is sent can be absolute or relative. An absolute value indicates the moment at which a message is sent. At this time, the network simulator will inject the message into the network. Obviously, for independent messages, this field will always contain an absolute value. On the other hand, a relative value indicates the time since the arrival of a certain message until the departure of the message on which it depends. Therefore, the network simulator will spend a time equal to this value between the arrival of a message at a node and the injection of the dependent message. For dependent messages, the injection time is always taken as a relative value.

The injection time is obtain by computing the time that the simulated processor needs to execute the instructions before the instruction send (absolute value), or between a pair of dependent instructions send and receive (relative value). Our environment allows us to choose among the execution times of some commercial processors like transputers and others. The use of dependent messages allows us to use the same traces to simulate different network parameters.

5 The Network Simulator

Next, we are going to describe the second phase of our environment, the interconnection network simulator. The unique feature included in our simulator is that it permits the network to be dynamically reconfigured. A reconfigurable network presents some advantages, the most interesting one being that it can

easily match the network topology to the communication requirements of a given program, properly exploiting the locality in communications; moreover, programming a parallel application becomes more independent of the target architecture because the interconnection network adapts to the application dynamically.

In our simulator we can evaluate the performance of the interconnection network for parallel applications and not only for synthetic workloads. This allows us to vary the parameters of the reconfigurable network and study how to improve its behaviour in real cases. This phase of the simulator consists of two modules, the mapping module and the network simulator module. We are going to detail the features of each one of them.

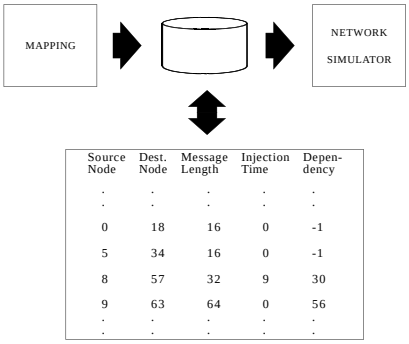

Fig. 4. Intermediate code after the mapping

The mapping module is the first one of the second phase and is responsible for translating the process-oriented communication trace to a processor-oriented intermediate code according to some pre-defined mapping functions.

The intermediate code output by this module is slightly different from the communication trace. Now, the first two fields are related to processors (source and destination processor) instaed of processes. The remaining code is unchanged. Figure 4 shows an example of this intermediate code. With this module, we can evaluate several mapping algorithms for a given parallel algorithm.

The last module is properly the network simulator. It is an improved version of a previous simulator [6] that supports network reconfiguration. It can simulate at the flit level different topologies and network sizes up to 16K nodes. The topology of the network is definable by the user. Each node consists of a processor, its local memory, a router, a crossbar and several channels. Message reception is buffered, allowing the storage of messages independently of the processes that must receive them. The simulator takes into account memory contention, limiting the number of messages that can be sent or received simultaneously. Also, messages crossing a node do not consume any memory bandwidth.

Wormhole routing is used. In wormhole routing [5] a message is descomposed

into small data units (called flits). The header flit governs the route. As the header advances along the specified route, the remaining flits follow in a pipeline fashion. The pipelined nature of wormhole routing makes the message latency largely insensitive to the distance in the message-passing network.

The crossbar allows multiple messages to traverse a node simultaneously without interference. It is configured by the router each time a successful routing is made. It takes one clock cycle to transfer a flit from an input queue to an output queue. Physical channels have a bandwidth equal to one flit per clock cycle and can be split into up to four virtual channels. Each virtual channel has queues of equal size at both ends. The total queue size associated with each physical channel is held constant. Virtual channels are assigned the physical channel cyclically, only if they can transfer a flit. So, channel bandwidth is shared among the virtual channels requesting it. It should be noted that blocked messages and messages waiting for the router do not consume any channel bandwidth.

The most important performance measures obtained with our environment are delay, latency and throughput. Delay is the additional latency required to transfer a message with respect to an idle network. It is measured in clock cycles. The message latency lasts since the message is injected into the network until the last flit is received at the destination node. An idle network means a network without message traffic and, thus, without channel multiplexing. Throughput is usually defined as the maximum amount of information delivered per time unit.

The network reconfiguration is transparent to the user, being handled by several reconfiguration algorithms that are executed as part of the run-time kernel of each node. This class of reconfiguration is not restricted to a particular application, being very well suited for parallel applications whose communication pattern varies over time. The goal of the network reconfiguration is to reduce the congestion of the network. For this, when the traffic between a pair of nodes is intense, the reconfiguration algorithm will try to put the source node close to the destination node to reduce the traffic through the network and, therefore, to reduce the congestion that may have been produced. The reconfiguration algorithm decides when a change must be carried out by means of a cost function. The network reconfiguration is carried out in a decentralized way, that is, each node is responsible for trying to find its best position in the network depending on the model of communication. Also, reconfiguration is limited, preserving the original topology. There are several different parameters [8] that can be varied to adjust how the reconfiguration is performed.

With reconfigurable networks, we want to reduce the latency and delay and to increase the throughput. Also, we want to have a small number of changes to keep the reconfiguration cost low. The quality of each reconfiguration is measured by the simulator.

6 Conclusions

Application development via high-performance simulation offers many advantages. Simulators can provide a flexible, cost-effective, interactive debugging en-

vironment that combines traditional debugging features with completely nonintrusive data collection.

We have presented an environment for evaluating the performance of multicomputers. Our environment, unlike most of the earlier work, captures both the communication pattern of a given algorithm and the most important features of the interconnection network, allowing even the dynamic reconfiguration of the network. This feature is a valid alternative to solve the communication bottleneck problem. By means of using a reconfigurable topology, the principle of locality in communications is exploited, leading to an improvement in network latency and throughput.

Until now, the study of these features was difficult because there were no tools that permited varying the different parameters of reconfigurable networks. In this paper, a system that solves this problem and opens a way for studying reconfigurable networks has been presented. A reconfigurable network yields a variety of possible topologies for the network and enables the program to exploit this topological variety in order to speed up the computation [3].

References

1. Adamo, J.M., Trejo, L.: Programming environment for phase-reconfigurable parallel programming on supernode. Journal of Parallel and Distributed Computing. **23** (1994) 278–292
2. Brewer, E.A., Dellarocas, C.N., Colbrook, A., Weihl, W.E.: Proteus: A high-performance parallel-architecture simulator. In Proc. 1992 ACM Sigmetrics and Performance '92 Conference, (1992) 247–248
3. Ben-Asher, Y., Peleg, D., Ramaswami, R., Schuster, A.: The power of reconfiguration. Journal of Parallel and Distributed Computing, **13** (1991) 139–153
4. Davis, H., Goldschmidt, S.R., Hennesy, J.: Multiprocessor simulation and tracing using Tango. In Proc. of the Int. Conf. on Parallel Processing, (1991) II99–II107
5. Dally, W.J., Seitz, C.L.: Deadlock-free message-routing in multiprocessor interconnection networks. IEEE Transactions on Computers, **C-36**, No. 5 (1987) 547–553
6. Duato, J.: A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Transactions on Parallel and Distributed Systems, **4**, No. 11 (1993) 1–12
7. García, J.M.: A new language for multicomputer programming. SIGPLAN Notices, **6** (1992) 47–53
8. García, J.M., Duato, J.: Dynamic reconfiguration of multicomputer networks: Limitations and tradeoffs. Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, (1993) 317-323
9. Hartson, H.R., Hix, D.: Human-computer interface development: Concepts and systems. ACM Computing Surveys, **21**, No. 1 (1989)
10. Karp, A.: Programming for parallelism. IEEE Computer, (1987) 43–57
11. Nichols, K.M., Edmark, J.T.: Modeling multicomputer systems with PARET. IEEE Computer, (1988) 39–48
12. Segall, Z., Rudolph, L.: PIE: A programming and instrumentation environment for parallel processing. IEEE Software, (1985) 22–37

This article was processed using the LaTeX macro package with LLNCS style