NAUTA: A Network Administration Utility for Transputer Architectures

G. Marino, G. Succi, G. Levo and R. Pavesio

DIST – Università di Genova, via Opera Pia 11a, I-16145 Genova, Italia

Abstract


MIMD reconfigurable architectures including transputer arrays, are becoming very popular because of their natural approach to parallelism and the fact that they grant large flexibility. However, they leave a crucial task to the programmer: he must divide the body of the whole program in many parallel modules, and define the network’s topology in order to best adapt it to the communication needs arising when the various modules must communicate with each other. The computation and communication loads should be well distributed over the network available resources, in order to ensure good performances. Maybe this goal cannot be accomplished at the very first attempt, thus it would be useful to measure the traffic along the various links of the network in search for ‘bottlenecks’ which cause delays. This would suggest which changes to the distribution of computation duties and to the routing of messages which would lead to an improvement of the global traffic’s fluency.

NAUTA was developed to cope with the problems of administrating the topology of the network and analyzing its subsequent influences upon the general behaviour, when running an application on a transputer architecture. It offers the user a graphical interface to set up the network configuration and to load the processes constituting the application onto the processors.

Finally, NAUTA gives an overview upon the runtime behaviour of the processors and the traffic between them, in order to highlight possible bottlenecks. This overview is presented in the form of a set of runtime-evolving ‘Strip charts’, providing a good degree of comprehension to a user’s glance.

The present version of NAUTA was designed to operate inside the environment of a transputer computing system with CSTools, the application development toolset for multiprocessor systems distributed by Meiko Ltd. However, since NAUTA is essentially an interface between the application programmer and the computing environment; not many details of the latter need to be known, perhaps apart from some small constraints on the realization of the topology which could be removed when handling a different system to Meiko’s. The NAUTA approach is quite general and portable to other kinds of MIMD machines. The graphical world where NAUTA was conceived is that of XWindows, and in particular massive use was made of the XtIntrinsics [1] and the Athena Widgets [2]: in this way the development of NAUTA’s interface proceeds with the aid of the great modularity supported by the Widgets and it is very easy and intuitive for a user to learn how to use it [3].

Keywords. Parallel programming tools; transputer; MIMD architectures.

1. Configuring a MIMD architecture

The great versatility of MIMD machines is due to the way they approach parallelism: given a task to solve, it is often easy to split it into simpler pieces, elementary tasks that can be evaluated in parallel. Therefore, when the problem is such that it can be split in an easy way, it is immediate to think of assigning each smaller task to a different processor, and provide some means of communication between them, to ensure data-passing. This is of course only a simplification of the real scenario offered by a MIMD system at work
Fig. 1. Splitting a global task into a network of nodes and connectors.

Fig. 2. Mapping grouped processes onto a network with limited resources.
on a complex problem, but it is already enough to point out which are the salient features of these machines.

Figure 1 shows a diagram recalling this approach of MIMD architectures to parallelism, and it can already be observed that the job of assigning the program segments to the several processors building up the architecture is also not really trivial. In particular, an application programmer must manage this job using a limited set of resources. That is, it is not always the case that there are at least as many processors as code modules, and for each node as many communication links as the local process would need to be connected to all its neighbours. Therefore, processes will have to be divided into groups to be assigned to a processor, and some connections between groups will not be direct, so the messages will have to be routed along the links of the network, going beyond intermediate nodes. An example of such a situation is shown in Fig. 2.

From these remarks the importance of network administration is clear, i.e. managing the available resources (nodes and links) to achieve the best possible performances. It is the responsibility of the programmer, therefore, to decide how to split the global task into smaller modules, how to group them together for mapping them onto the processors, and also how to connect them using the available links in the most efficient way.

2. Exploiting the resources

Creating a customized environment for running an application may have a great effect upon its runtime behaviour. Probably, the user would need several attempts to find the optimal architecture configuration to run his application. Therefore, it was decided to provide a tool to make this phase of work in some way less time-consuming. NAUTA was conceived to permit, through a graphical interface, the straightforward configuration of networks, and also to provide a runtime overview of the load on the processors and links due to the various communicating processes scattered on the network. The aim of this traffic monitoring is to point out the ‘bottlenecks’ in the present configuration, enabling the user to targeted modifications of the network topology in order to distribute more homogeneously the computation and communication duties over the available processors and links. The idea is to try running the application on an initial network, monitoring the balancing of loads, thus discovering possible hindrances, and possibly changing the network configuration, maybe several times.

3. Architecture configuration with NAUTA

As regards the job of network topology definition, NAUTA is, in its present implementation, merely a graphic interface to edit ‘.par’ files, which are a method of architecture configuration used under CSTools, the program development toolset for multiprocessor computer systems distributed by Meiko Ltd [4]. In other words, the result of the configuration phase will be a file obeying to the syntax rules of .par files, ready to be run. Nevertheless, the front-end of the interface provided by NAUTA is oriented to any user, even those who have no familiarity with that particular syntax. In some way, this facility pro-
Fig. 4. A simple example of a network layout generated by NAUTA.

Fig. 5. The placement of processes is simply an editing operation.
vided by NAUTA is analogous to the paper sheet where application programmers usually sketch the architecture layouts devoted to accommodate their programs. These programmers do not have to translate their sketches into .par files: it is done by NAUTA's configuration interface.

3.1 Using NAUTA’s Interface for network topology definition

Let us first describe how to define a completely new topology. The interface window displays a stack of available processors: they can be picked up and moved across the window to obtain the desired layout (see Figs. 3 and 4).

Each of these processors, representing a transputer, has four ports (also called links). Once the processors have been placed to form the nodes of the network, some links belonging to different processors can be connected in couples to establish the communication paths, thus realizing a complete network.

For those particular kinds of problems for which standard topologies (e.g. pipeline, binary tree, etc.) are suitable, the .par file method provides shortcut keywords to carry out such regular structures. NAUTA also does this, providing the proper command buttons on its interface.

3.2 Loading the processes onto the processors

NAUTA associates with each processor a list of the processes that run on it. This list is in practice a text editor where the user types in the names of the processes (see Fig. 5). In case the same group of processes has to be loaded onto more than one processor, a repeat button is provided which will load the processes onto other nodes. Thus, the current process list will be appended to the ones already existing for each of those other processors.

The process list of every processor can be popped up for editing as often as desired, to enable possible additions or corrections to its current content.

3.3 Retrieving and modifying a previously defined network

The user’s .par files, including those not created with the help of NAUTA, can be submitted to NAUTA, which will show on the screen the corresponding topology, and will load the process lists onto the respective processors. At this point, the user can run the .par file or modify the structure, either to obtain a better view on the layout, or to change the process lists of the pro-
cessors and the connections between them. This second possibility is related mainly with the theme we introduced before, namely looking for the network configuration which best fits a given application.

The NAUTA .par file interpreter must of course be designed so that, when an existing structure is retrieved, the layout which is initially exposed on the screen is already quite comprehensible, with the nodes placed in regular structures and few crossing communication lines.

3.4 Setting up load monitoring

To perform runtime evaluation of the computation loads on the nodes and communication loads on the links of the network, NAUTA uses of a set of test processes. Therefore, these processes themselves must be loaded on the processors involved in the benchmarking; so, as an extension to the phase of process loading, the user decides which nodes and links are to be monitored during the execution of his application program. NAUTA provides several options to specify to what extent benchmarking should be carried out:

- complete monitoring over the loads on every processor and every link in the network;
- monitoring of the computation loads on all the processors only;
- monitoring of the communication loads on all the links only;
- selective monitoring over the loads only on the specified processors and links.

The reason for providing several partial methods (see Fig. 6), giving up the complete monitoring, is not the performance enhancement, reducing the work to be done. It comes rather from the intent of keeping the reports on the screen at a good level of comprehension for a human glance, limiting the number of subwindows needed for the output.

3.5 Running the application

The preceding subsections were devoted to describing how an application programmer can use NAUTA’s interface to prepare the proper environment for executing his program. Placing the processors-nodes, establishing connections between them, loading processes onto processors and eventually setting up a benchmarking system, are thus the phases that properly allow the task of configuring the multiprocessor architecture. It must be noted still, that NAUTA does not put any sequence constraint upon these phases. That is, they can be performed in any possible order, in the way the user finds most natural. Every move can be reversed until the user decides to execute his application with the current architecture configuration.

When this happens, the present situation is saved in a temporary .par file, which is then launched by NAUTA using a normal CSTools ‘mrun’ command. This causes actual execution of the application program, and also creation of the output subwindows needed to issue the reports of load measures upon the nodes and connectors of the network.

4. Benchmarking on a MIMD system

4.1 The reasons for analyzing the performances

As we already pointed out it is very important, when dealing with a MIMD reconfigurable architecture, to provide a good level of efficiency to communications between processes, and to distribute the computation duties homogeneously over the processor set. The various existing MIMD architectures can differentiated with regard to the strategies they adopt to carry out these tasks, and probably, though MIMD machines are oriented to be ‘general-purpose’, there are some which are more efficient than others when solving particular problems. However, once an application programmer has chosen the machine on which to run his programs, there is still a lot he can do to exploit the resources in the best way. The quantities he can control are the code of the parallel modules which compose the application program, and the network configuration. This second aspect is considered here.

The network is composed by the processors ‘Pi’, each loaded with its local processes, some of which need to communicate with each other. For messages whose sender and receiver reside on the same node, there is probably no need to access the routing mechanism provided by the system, the communication being performed through a simple local memory access. However,
there are also messages that must be sent around along the network’s connections, and perhaps even passing through intermediate nodes, when no direct communication link exists between source and destination. In this last case, the amount of traffic they find on their path to the destination influences the global communication speed; in fact, even in those cases in which messages are so short that they can be sent through the link in a ‘one-shot’ mode, they must first reckon with the other requests made by the processes that use the same link.

In order to minimize the communication delays, the user should pay a lot of attention when placing the processes on the processors and creating the connections, because if the computation and message-routing duties are well distributed, there will be an improvement in the global traffic’s fluency. Therefore, although he should already have an a priori idea of which paths and nodes are busier, it would be beneficial to have a real measure of the actual load on processors and links.

4.2 The problems to solve

From the very beginning of the NAUTA project, it was pointed out that there are two main features which the benchmarking system should have. First, the job of taking the measurements should be as unintrusive as possible. Always, when trying to analyze a dynamic quantity, there is a corruption of its autonomous behaviour, and therefore measures must be developed which take care to ‘disturb’ the system as little as possible. Second, the reports of the measured loads should be at the user’s disposal at runtime. The reason for this is that during the execution of the application program, some processes can be created and some can terminate, there can be long phases of massive computation with almost no need of communication and vice versa. Therefore, the plots of the loads on the processors and links are not in general almost flat: rather they have ‘stairway’ shapes, and it would be interesting to relate modifications happening in different parts of the network.

Keeping these remarks in mind, NAUTA’s benchmarking system was designed as a sequence of instant measurements over the loads, resembling a sampling process, the sampled quantity being the runtime-evolving load on a processor or a link. This grants simultaneous progress of both the application program and the reports of the related loads. On the other hand, to prevent the benchmarking system from itself being a cause of increased work and poor performances for the MIMD architecture, the time interval between two subsequent instant load measures should be kept long, but also short enough to obtain a nearly unbroken report. In the next section, the strategy which was followed in developing NAUTA’s load evaluation system will be introduced. The duration of the ‘sampling interval’ is a very important parameter, and influences the sensitivity of the whole benchmarking system.

5. NAUTA’s benchmarking system

5.1 Choosing a strategy

The approach to runtime load evaluation used by NAUTA is very simple: the time that a process takes to be executed is a measure of how much the resources needed by that process are loaded. That is, if there is more than a single process to serve, the resources will not be able to grant the minimum execution time, and the delay is proportional to the amount of requests waiting for execution. Therefore, a load measurement event can be composed of three phases: reading the current time from an internal clock, launching a test instruction, and finally calculating the elapsed time and issuing it to the output. Two different test instructions are used: one that uses the CPU for calculation and thus evaluates the computational load on the processor, the other that accesses the communication link facilities and measures communication traffic upon a link connecting two processors.

Always using the same instruction for each purpose provides a constant landmark to compare the various execution times. On the other hand, the frequency of the measurement events should be kept reasonably fixed and it should be possible to adjust it (for the sake of sensitivity, as mentioned above).

Because of this, it is difficult to entrust the processes themselves, which compose the application program, with the task of performing the load measures: in which points of their code
should the test instruction be placed? How often? Moreover, two versions of the same program should exist. One copy of each process containing the extra code needed for benchmarking, the other copy lacking it and used when performance evaluations are not requested any more.

Thus, another strategy seems to be more promising: benchmarking is carried out by a set of elementary `shadow' processes that are placed on every node and at both ends of each communication link of the network whose load is requested to report. It could be observed that this would bring an extra quantity of processes scattered on the machine, inducing a heavy decrease in the global performances. However, the current version of NAUTA, implemented on a Meiko transputer architecture, adopts just this strategy, because that machine allowed the `shadow' processes to be designed so that they could provide a reliable runtime benchmark report without competing too hard with the application program for using the network resources.

5.2 The structure of the measuring processes

At most, three `shadow' processes must be placed on every node of the network, in addition to those belonging to the application program. One of them takes care of measuring the computational load over the processor where it resides. The other two, which can be called `writer' and `reader' for the sake of clearness, are devoted to evaluate the communication traffic along a link connecting two processors. In truth, the `writer' and its respective `reader' are to be placed on the two different nodes which are the ends of the connection (see Fig. 7). However, a processor which is used as both a destination for incoming message traffic and source for outgoing traffic, will have both a `writer' and a `reader' together, although not related with each other.

The first important characteristic of the `shadow' processes is their low memory occupation: their body is quite small, since they perform very few operations. This strategy of using dedicated processes to evaluate the network traffic, enables the system described above to be realized: the load measurements can be obtained by calculation of the time elapsed during the execution of an atomic instruction or communication. This duration can be related with the number of pending requests that are executed in the meantime, i.e. the load found on the processor or link in that moment. In particular, the shadow processes execute always the same instruction, thus providing a fixed term of comparison for the execution time. Moreover, after having terminated the instruction, measured and issued the execution time as a load report, these processes suspend themselves for a desired time: this achieves an almost constant `sampling interval', and grants low interference of the `shadow' processes with the application's ones. In fact, when they are suspended, they free all the resources.

![Fig. 7. Load measurement by mean of the shadow processes.](image)
back to the normal application's traffic. Then, after having waited for the fixed interval, they wake up again and repeat a new cycle of measurement.

These aspects of the shadow processes strategy seem to overcome the disadvantages pointed out in the preceding subsection. Indeed, some proofs were made using test processes simulating a normal application program, and their behaviour was scarcely affected by the presence of the 'shadow' processes.

Finally, another feature of the writer-reader mechanism is worth mentioning. A single reader placed on a processor carries on the duties of replying to the messages coming from all the neighbouring writers, and a single writer sends messages to all the neighbouring readers. In particular, the writer processes are those that carry on the task of measuring the load on the links and of transmitting it for the report on the screen. The readers are only present on the processors to receive the messages sent by the various writers. It is also important to note that each reader provides a single letter-box for the messages coming from all the neighbours, so that there is no ordered sequence to comply to, and an almost immediate reply can be given to any incoming call. In this way, there is no risk of deadlock among the writer processes.

6. Conclusions

The mechanism used by NAUTA to evaluate the performances of a MIMD architecture is quite simple and relies upon the internal scheduling system provided by the processors [5,6]. It seems to give a reasonably reliable report of the runtime behaviour of computation and communication loads. For instance, using test processes simulating a typical application program, it was possible to observe a linear increase of loads with respect to the number of running test processes.

No particular assumption was made over the scheduling mechanism, since its details were not known. However, NAUTA pays for the lack of precision that maybe derives from this hardware-independence with a port ability of the benchmarking system to other kinds of MIMD machines than the transputer ones.

The output is presented on the screen in form of 'Stripcharts', resembling the well-known 'Xload' plots issued by XWindows. Of course, when the number of processors and links whose load is requested to report is large, the screen can be filled with a lot of 'load' subwindows, and it can become difficult for the user to follow the runtime evolution with a glance.

Obviously, the report issued by NAUTA gives only a qualitative idea of the traffic on the network. It would be very naive to think that the complexity of real application programs were always as low as in our examples. Therefore, and also because NAUTA has not been used to develop any application yet, it is still impossible to predict whether it will give a real help or not.

NAUTA is still far from being an intelligent system that automatically extracts from the structure of an application the network topology best suited to running it: all of this work is still left to the user.

On the other hand, NAUTA's graphical interface for network configuration is really handy and should appear very useful to those application writers used to work with .par files or similar methods. The processors can be moved, connected, loaded in the most natural way, and connections are preserved during motion.

Of course, when the architecture provides a great number of processors, some problems can arise in presenting on the screen a very close network, which can appear a little puzzling to the user.

References

Gianpaolo Marino is currently Associate Professor at DIST-Università di Genova. He has got his Laurea Degree in 1977 at DIST. He became Associate Researcher in 1978, Assistant Professor in 1981 and Associate Professor in 1988. His research interests concern parallel and distributed processing, open systems and networks.

Giancarlo Succi is currently a post-doctoral fellow at DIST-Università di Genova. He has got his Laurea Degree in 1988 at DIST, the MS in 1991 at SUNY-Buffalo and the PhD in 1993 at DIST. His research interests concern parallel and distributed processing and the compilation of declarative languages for parallel architectures.