Parallel Computing:
From Theory to Sound Practice
W. Joosen and E. Milgrom, Eds.
IOS Press, 1992

Millipede - A Programming Environment Providing Visual Support for Parallel Programming

M. Aspnäs R.J.R. Back T. Långbacka

Åbo Akademi University
Department of Computer Science
Lemminkäinengatan 14, SF-20520 Åbo, Finland

December 30, 1991

Abstract

This paper describes Millipede, a graphical programming environment for a Transputer-based MIMD multiprocessor system. Parallel programs are described as graphs, where the nodes denote parallel processes and the edges denote communication channels between processes. Graphs are constructed using a hierarchical graph editor which allows the user to group processes (nodes) together into hierarchical process structures. The highest level in the graph hierarchy, called the processor graph, also describes the processor network on which to execute the parallel program. Millipede contains tools for mapping processor graphs onto a reconfigurable transputer network and for configuring the target processor network accordingly. Monitoring data, produced and collected by a performance monitoring system, can also be presented upon the processor graph.

Keywords: Parallel programming, Programming environments, Visual support for programming, Transputer, MIMD multiprocessors, Performance monitoring

1 Introduction

Programming multiprocessor systems is often associated with great difficulty. One obvious reason to this is that there are less programming tools available for these types of systems than for traditional sequential systems. This is partly due to the fact that multiprocessors have not been widely available for as long a time as uniprocessor systems. Another reason is that there are several different categories of multiprocessor systems, each requiring their own set of programming tools. Furthermore, the implementation of many traditional programming tools, such as for example debuggers, becomes much more difficult in a multiprocessor environment. In fact, debugging of parallel programs is in itself an active research area.

Many of the existing programming tools reflect the structure of parallel programs very poorly and do not support the software development process very well. Parallel programs are often created using only a text editor. Clearly, a parallel program

organized as a set of textfiles is not very easy to understand and manage. Program representations used in programming tools for sequential systems are often not suitable for parallel systems.

System software for sequential computers completely hides the hardware architecture from the user. For multiprocessor systems, the programmer often has to be very much aware of the underlying hardware architecture and has to use this knowledge in the programs. For instance, logical entities (e.g. processes, communication channels) might have to be explicitly mapped onto physical ones (e.g. processors, physical communication links), using specific programming language constructs. Especially when the programmer wants to experiment with several different mappings while constructing the program, the amount of work can be considerable.

Overview of the work. The Millipede programming environment, which we present in this paper, is an attempt to solve the problems discussed above. The design goal has been to find a single program representation that can be used during the whole process of constructing efficient parallel programs. Graphs are commonly used to describe different aspects of parallelism. In the *process graph* representation we have chosen, the nodes of a directed graph denote processes and the edges denote unidirectional communication paths between processes. The design ideas behind Millipede were originally presented in [1].

Millipede addresses the problems of programming a traditional host/target processor network, where a programmer has a fixed amount of processing units at his disposal. Our ambition has not been to create an environment supporting multiple users which share a common processor network. With the development of new hardware and distributed operating systems, traditional host/target systems will to some extent be of less importance. However, in many application areas (e.g. real-time and embedded systems) single-user environments will still be important. Although we restrict ourselves to single-user systems, we believe that the concepts presented in this paper are generally applicable.

Millipede has been implemented on the transputer-based [14] reconfigurable multiprocessor system Hathi-2 [2]. Millipede integrates a set of programming tools that support the steps involved in constructing efficient parallel programs, from the initial high-level description in form of a process graph to a performance-tuned final executable program. All the tools are integrated under a common graphical user interface that implements the process graph representation of programs. The graphical user interface provides a *graph-based visual extension* to the CSP/Occam programming model [11, 13] and hides the underlying target architecture from the programmer.

The basis of the user interface to our programming environment is a graph editor. Using the editor, the user constructs a labelled process graph from which an executable parallel program automatically can be constructed. Each node in the graph is labelled with a process name and a parameter list, and contains a reference to the source code that describes the computational behaviour of the process. A process usually has a set of *input-* and *output-ports* associated with it. A logical communication channel connects an output-port on one process to an input-port on another process. Ports are labelled with a name and a message datatype.

Sets of processes can be grouped together into *compound processes*, thus forming a hierarchical process structure. In other words, a compound process contains a subgraph of processes, some of which can also be compound processes, and so on. The highest level in the graph hierarchy, which we call the *processor graph*, is interpreted as

a description of a logical processor network. The processor graph is used to automatically generate a process-to-processor mapping and to reconfigure the physical processor network accordingly. The environment builds all necessary source code files using the information in the process graph. Compilation, linking etc. is also administered by the environment.

Since the main reason for using parallel computers is to gain efficiency, the performance of programs is of crucial interest. Therefore, tools supporting performance analysis are needed. In Millipede, figures describing utilization of the CPU's and communication links of the physical processor network are shown on the corresponding nodes and edges in the processor graph. The user can then easily relate performance figures to the structure of the parallel program.

Related work. The need to simplify the task of writing efficient parallel programs has resulted in the implementation of a number of other graphically oriented, integrated programming environments. A majority of these systems are, however, designed for the shared memory model of computation, like for example PIE [15, 17] and FAUST [10].

TOPSYS [5] is an example of a programming environment for distributed memory systems. The environment includes tools for process to processor mapping, program animation, program debugging, performance tuning and dynamic load balancing. TOPSYS does not provide the user with the kind of visual support that Millipede offers. For example, the performance analysis tool [6] uses traditional data presentation techniques such as diagrams and histograms. We feel that these techniques do not sufficiently support performance analysis of parallel programs.

MARC [7] and TIPS [22] are integrated environments for transputer-based systems. Both possess features similar to the ones in Millipede, like automated process-to-processor mapping and support for performance analysis. Neither one, however, offers any visual support for the actual program construction process.

The possibility to visualize different aspects of parallel processing using graphs has also motivated other researchers. An early and often referenced graph-based programming environment for parallel programming is Poker [20]. It gives the programmer a possibility to specify the communication structure and assign processes to processors using a graph description, features which are typical for many of the more recent programming environments.

In [4] a powerful general purpose graph editor for parallel programs, called ParaGraph, is described. The authors advocate a graph based representation of parallel program similar to the one described in this paper. ParaGraph has advanced features for handling large process graphs. It offers scalable graph specifications, based on graph rewriting rules, and support for graph visualization. These are issues which have not been considered in depth in the Millipede project. Some work in this direction has been done, in form of support for regular replicated process structures. However, ParaGraph is not a complete programming environment. It is basically a building block for a programming environment user interface. ParaGraph does not support hierarchical graph specifications. We feel that graph hierarchies are very useful for structuring large graphs.

A somewhat different graph-representation is used in CODE/ROPE [8, 9]. The user can specify dependencies between program components using dependency graphs. The main idea behind CODE/ROPE is to support modular design of programs and the structuring of reusable program libraries.

2 Hierarchical process graphs

Parallel programs consist of a number of parallel processes which cooperate with each other in order to solve a given task. When constructing parallel programs, the programmer has to be able to organize the program in a way that reflects his/her understanding of the program. We feel that graphs are the most natural way of representing parallel programs. As pointed out in [4], graphs have been used for describing process structures, data dependencies, process to processor mapping, performance visualization etc., and they are often used for development and description of parallel algorithms. Therefore, in Millipede, we have chosen to represent parallel programs as hierarchical process graphs, in which the nodes denote parallel processes and the edges denote communication paths between processes.

The interpretation of such graphs depends on which programming model is assumed, i.e., whether communication between processes is implemented using shared memory or a communication network. Millipede is designed with the CSP/Occam-model in mind. Inter-process communication is assumed to be handled by the means of synchronous message passing over unidirectional channels. Even though we base our approach on the distributed memory model, we feel that the ideas on which Millipede is based can be employed also for other models of computation.

Process graphs in Millipede are constructed using four types of objects: processes, input- and output-ports and channels.

The *process objects* denote user defined processes, which are executed in parallel. Each process object is labelled with a name, an optional parameter list and source code that describes the computational behaviour of the process.

Processes can have an arbitrary number of *input- and output-ports* associated with them. Port objects are labelled with a name and a message datatype (the message protocol). A *channel*, represented by a directed arc, connects an output-port in one process to an input-port in another process. Processes can only refer to a channel via the port name to which the channel is connected. Thus the port name is used as a local name for the channel.

A set of processes can be grouped together into a *compound process*, which will contain the original set of processes as a subgraph. The component processes of a compound process can be either ordinary processes or compound processes themselves. In this way it is possible to construct arbitrarily large hierarchical process graph structures. Compound processes are interpreted as pseudo-parallel processes which are executed on the same processor (using a time-sharing scheduler). The highest level in the process hierarchy, which we call the *processor graph*, is interpreted as a description of the network configuration on which the process graph will be executed.

A labelled hierarchical process graph of this kind completely describes a parallel program: the parallel processes of the program, the source code of the procedures that constitute the parallel processes and the interconnections between the processes. The highest level in the graph also describes how processes are grouped onto processors and also how these processors are interconnected. This information can be used for extracting a textual representation of the program, which can be mapped onto a physical processor network topologically equivalent to the processor graph, assuming that such a connection is physically realizable on the target system.

Since the process graph describes the logical structure of the parallel program, it

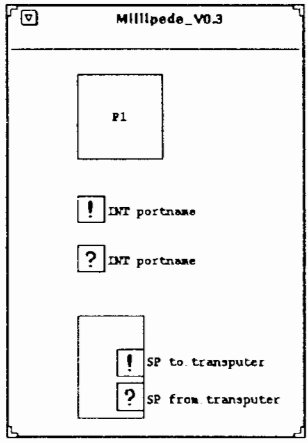

Figure 1: The palette

is also very well suited for both diagnostic and performance debugging as well as program animation. Performance figures, error messages and other kind of information to the user, can be presented in a way that relates well to the programmers own view of the program. Graphs have successfully been used for parallel debugging [12] and performance analysis [22].

3 The Millipede programming environment

The Millipede programming environment is implemented on a Sun SPARCstation, which acts as a host computer for the Hathi-2 multiprocessor system [2], a transputer-based reconfigurable general-purpose multiprocessor system. The menu-driven graphical user interface is based on X-Windows and is implemented using DesignML [16]. In the current implementation, Millipede supports parallel programs written in Occam-2. Part of the programming tools integrated into the environment are commercially available products (e.g. Occam Toolset), and some are designed especially for the Hathi-2 system.

When the Millipede programming environment is started, the user is presented with a worksheet on which he/she can draw the process graph. Graphs are constructed by selecting template objects from a palette (see Figure 1) and pasting them onto the worksheet, to the position indicated by the mouse. The environment does not contain any support for automatic visualization of graphs. The user is assumed to organize the graph according to his view of the structure. However, there is support for aligning the different objects to each other. The palette contains four types of objects (listed in order of appearance): A process object, an output-port, an input-port and a host computer object.

To construct a process, the user selects the process object from the palette and pastes this onto the worksheet. The process name and the optional parameter list can be edited by selecting the process and choosing an appropriate command from the **Process** menu. The code of the process is edited in a similar way by invoking the **Edit** command. The user is then presented with an editing window in which the source code

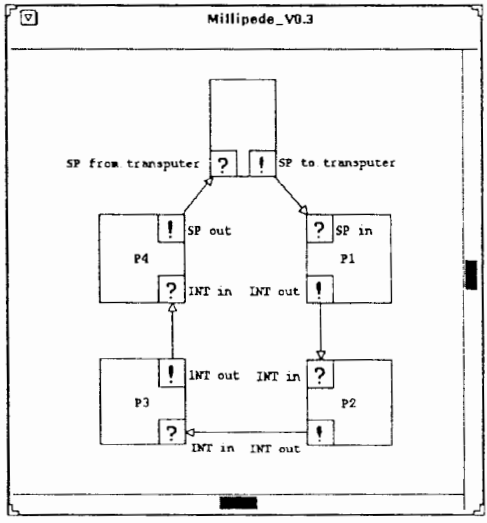

Figure 2: 4 processes connected in a ring

of the process is written.

Ports are created by selecting either the input- or the output port object from the palette and placing it onto a process. The port attributes (name and message protocol) can be edited by choosing a command from the **Port** menu. Channels between processes are introduced by connecting an input-port to an output-port. The environment checks that the connection is legal, i.e., that the channel connects an output-port to an input-port and that the two ports have a matching message protocol. In Figure 2 an example of a user constructed process graph is shown. The graph describes a parallel program consisting of four processes: `P1`, `P2`, `P3` and `P4`, connected to the host and forming a unidirectional ring.

Compound processes are created by selecting a set of processes and choosing the **Group** command from the **Task** menu. The selected processes are then grouped together into a single compound process, which is distinguished from a simple process by a thicker border line. Figure 3 shows the graph from Figure 2 where the processes `P2` and `P3` have been grouped together into a compound process by name `task41`. Figure 4 shows the subgraph of the compound process `task41`, which consists of the processes `P2` and `P3` from the original graph. The darkened port objects on the subgraph denote ports that connect the subgraph to the compound process. A compound process can be ungrouped with an **Ungroup** command, thus restoring the original process graph.

Using the primitive operations described above, the user can construct a hierarchical graph that represents a parallel program. In order to generate an executable program from this graph, the user chooses a **Make** command from the **Program** menu. This launches a sequence of actions, all using information extracted from the graph and carried out automatically by the environment. Below we briefly describe these actions.

Code generation. The processor graph is recursively parsed, and for each simple process that is found a source code file, in the form of an Occam-2 procedure, is created. The procedure head is constructed from the process name label giving the procedure

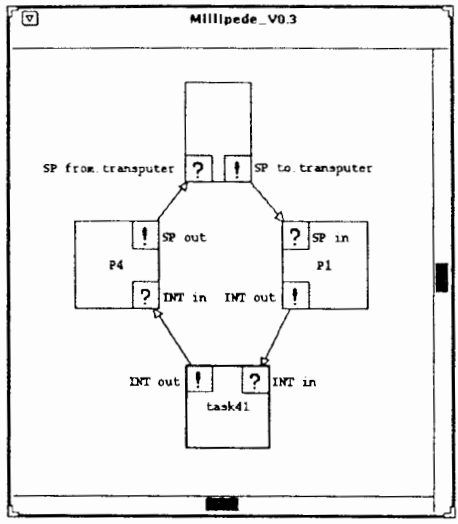

Figure 3: An example of a graph containing a compound process

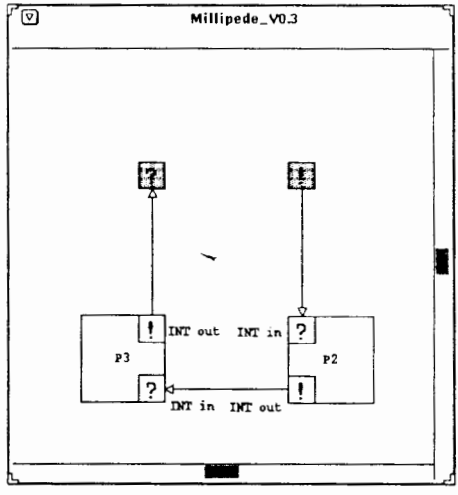

Figure 4: The subgraph of a compound process

name. The formal parameter list of the procedure is constructed using the name and datatype of the port objects associated with the process, and the optional parameter list label. The procedure body is simply constructed from the user-provided source code of the process.

For each compound process, a similar source code file is generated. The procedure head is constructed as above. The procedure body is constructed by parsing the subgraph and invoking all constituting processes in parallel with an Occam-2 PAR statement. To generate actual channel parameters for the procedure invocations, the environment has to parse the subgraph and extract information about how the constituting processes are interconnected.

Mapping. The processor graph describes a partitioning of the process graph onto a logical processor network. The process-to-processor mapping tool automatically allocates the logical processor network onto physical processors in a way that can be realized on the target multiprocessor. The mapping tool is based on a heuristic process-to-processor mapping algorithm called self-adjusting mapping [18, 19]. The mapping algorithm guarantees that a successful mapping is always found, by further combining processes into compound processes if a mapping otherwise can not be found.

The result of the mapping is a source code file that exactly describes how the logical processor network is mapped onto the target processor network. This file specifies exactly on which physical processors the logical processors are placed, and which physical communication channels are used to connect the processors to each other. When compiled, this produces an executable image that can be loaded onto the target multiprocessor system and executed.

Configuring. The mapping of the logical processor graph onto the physical processor network also defines the required configuration of the target network. This information is used to automatically reconfigure the target multiprocessor system. The environment builds a text file describing how the target network should be configured in order to match the processor graph given by the user. This description is sent to a configuration tool, that is part of the system software in Hathi-2 (see [2] for details).

After this, the user selects an **Execute** command from the **Program** menu, and the executable image of the program is loaded onto the target multiprocessor system by a network loader, and the execution starts. A separate execution window is created for interaction with the user.

The program can be debugged using the Occam Toolset debugger, in which case a similar execution window is created for interaction with the debugger. Currently, the environment offers no support for high level debugging and program animation, although some work in this area has been done [21].

4 Support for performance analysis

Parallel programs are designed with efficiency in mind. Sometimes, however, the speedup of the program is far less than expected. The reasons for the inefficiency is often due to imbalance in the usage of hardware resources in the multiprocessor system. In Millipede, it is possible to present performance data about the utilization of hardware resources in the target network. These figures are presented on the corresponding objects in the processor graph of the program. This way the user can associate the figures

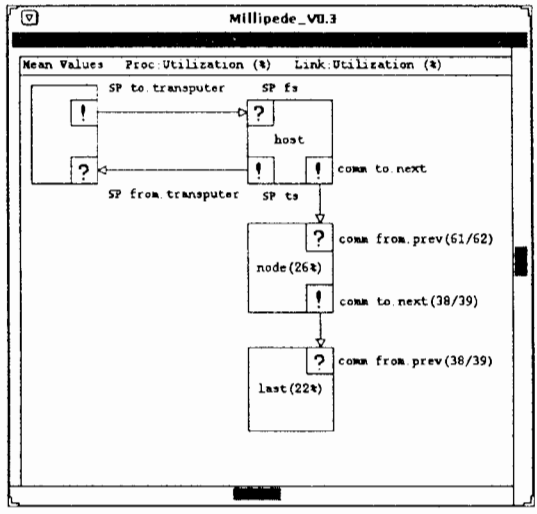

Figure 5: Monitoring data presented upon a processor graph

directly to the structure of the program. Many existing tools present performance data in form of tables, histograms or charts. This way the dependencies between monitored resources are hidden from the user, making the analysis unnecessary difficult.

The performance analysis tool is based on a monitoring system [3] that collects information about the utilization of the CPU's and communication links during program execution. The utilization degree of each hardware resource is measured in short timesteps (intervals), the length of which are user definable, thus forming a performance trace of the program execution.

The presentation system provides a set of pre-defined metrics which can be viewed in a number of different ways. CPU utilization is presented as percent of utilization during a time interval. The utilization of a link can be presented as the number of communications during an interval, or as the number of bytes transferred over the link. Furthermore, the data transmission time, the waiting time and the sum of these (the total communication time) can be presented, either as absolute values in milliseconds or as percent of the time interval. The user can step through the performance trace of the execution one interval at a time. The system also gives the user the possibility to view mean values and standard deviations over the whole or a part of the execution for all the metrics mentioned above.

Figure 5 gives an example of how mean values of CPU and link utilization are presented. The CPU utilization figures are printed after the process names of the processes called `node` and `last`. Similarly, the link utilization figures are printed after the port-names of these processes. The first value is the percentage of time spent waiting for communication synchronization (transputers communicate synchronously). The second value is the total percentage of time used for communication during the interval. There are no utilization figures neither for the process host nor for the ports associated with it. The reason for this is that the process object connected to the host computer object is automatically mapped to the host transputer of the transputer network, which currently can not be monitored.

5 Conclusions

The Millipede programming environment integrates a number of programming tools under a common graphical user interface, which allows the user to construct and manipulate parallel programs in the form of hierarchical process graphs. The programming environment hides the physical architecture of the target multiprocessor from the user, but still allows the user to control how logical processes are grouped together and placed onto physical processors. A prototype version of Millipede has been in use since spring 1991.

In order to make Millipede more useful a number of improvements could be made. At the moment, Millipede does not support top-down program development. This is due to how the commands for grouping of compound processes has been implemented in the current version. With some effort this problem could be solved. Another problem is that at the moment there is no support for graph visualization and graph scaling, i.e. the possibility to rewrite a simple graph into a larger and more complex one using rewriting rules. The debugger used by the environment has not yet been properly integrated under the user interface. The current implementation offers no support for high-level debugging and program animation.

Millipede is well suited for program development where the user wants to experiment with different process placement strategies in order to find the most efficient implementation. The ability to present monitoring information upon the processor graph of a program and step through a performance trace of an execution enables the user to compare different program versions. The graph-based user interface together with the automatic process-to-process mapping facility makes it very easy to modify the program and try out different implementations.

Acknowledgements

This work has been done in the FINSOFT III research program, which was financed by TEKES. We especially wish to thank Jens Granlund for implementing parts of the user interface. We also wish to thank Henrik Gullberg, Stefan Levander, Tor-Erik Malén, Hong Shen and Elena Trishina for their contributions.

References

[1] M. Aspnäs and R.J.R. Back. A programming environment for a transputer-based multiprocessor system. *Acta Cybernetica*, 9(3):291–301, 1990.

[2] M. Aspnäs, R.J.R. Back, and T-E. Malén. Hathi-2 multiprocessor system. *Microprocessors and Microsystems*, 14(7):457–466, September 1990.

[3] M. Aspnäs and T. Långbacka. A monitoring system for a transputer-based multiprocessor. In Peter Welch, Dyke Stiles, Tosiyasu L. Kunii, and Andre' Bakkers, editors, *Transputing '91 - Proceedings of the World Transputer User Group Conference, Volume 1*, pages 78–93. IOS, 1991.

[4] D.A. Bailey, J.E. Cuny, and C.P. Loomis. ParaGraph: Graph editor support for parallel programming environments. *International Journal of Parallel Programming*, 19(2):75–110, 1990.

[5] T. Bemmerl. The TOPSYS architecture. In H. Burkhart, editor, *Proceedings of the CONPAR-90 - VAPP IV (LNCS 457)*, pages 732–743. Springer-Verlag, 1990.

[6] T. Bemmerl, O. Hansen, and T. Ludwig. PATOP for performance tuning of parallel programs. In H. Burkhart, editor, *Proceedings of the CONPAR-90 - VAPP IV (LNCS 457)*, pages 840–851. Springer-Verlag, 1990.

[7] J.E. Boillat, N. Iselin, and P.G. Kropf. MARC: A tool for automatic configuration of parallel programs. In Peter Welch, Dyke Stiles, Tosiyasu L. Kunii, and Andre' Bakkers, editors, *Transputing '91 - Proceedings of the World Transputer User Group Conference, Volume 1*, pages 311–329. IOS, 1991.

[8] J.C. Browne, M. Azam, and S. Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, pages 10–18, July 1989.

[9] J.C. Browne, T. Lee, and J. Werth. Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE Transactions on Software Engineering*, 16(2):111-120, February 1990.

[10] V.A. Guarna Jr., D. Gannon, D. Jablonowski, A.D. Malony, and Y. Gaur. FAUST: An integrated environment for parallel programming. *IEEE Software*, pages 20–27, July 1989.

[11] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] A. Hough and J. Cuny. Initial experiences with a pattern-oriented parallel debugger. *SIGPLAN Notices*, 24(1):195–205, January 1989.

[13] Inmos Ltd. *Occam2 Reference Manual*. Prentice–Hall, 1988.

[14] Inmos Ltd. *Transputer Reference Manual*. Prentice–Hall, 1988.

[15] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. Visualizing performance debugging. *IEEE Computer*, pages 38–51, October 1989.

[16] Meta Software Corporation. *DesignML Reference Manual*, 1990.

[17] Z. Segall and L. Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, pages 22–37, November 1985.

[18] H. Shen. Self-adjusting mapping: a heuristic mapping algorithm for mapping parallel programs onto transputer networks. In J. Wexler, editor, *Developing Transputer Applications (Proc. 11th Occam User Group Technical Meeting)*, pages 89–98. IOS, 1989.

[19] H. Shen. Occam implementation of process-to-processor mapping on the Hathi-2 transputer system. In Peter Welch, Dyke Stiles, Tosiyasu L. Kunii, and Andre' Bakkers, editors, *Transputing '91 - Proceedings of the World Transputer User Group Conference, Volume 1*, pages 139–158. IOS, 1991.

[20] L. Snyder. Parallel programming and the Poker programming environment. *IEEE Computer*, 17(7):27–36, July 1984.

[21] U. Solin. Animation techniques for parallel algorithms. In E. Chiricorozzi and A. D'Amico, editors, *Proc. International Conference on Parallel Processing and Applications*, pages 437–445. Elsevier Science Publishers B.V. (North-Holland), 1988.

[22] A. Wagner, S. Chanson, N. Goldstein, J. Jiang, H. Larsen, and H. Sreekantaswamy. TIPS: Transputer-based interactive parallelizing system. In Peter Welch, Dyke Stiles, Tosiyasu L. Kunii, and Andre' Bakkers, editors, *Transputing '91 - Proceedings of the World Transputer User Group Conference, Volume 1*, pages 212–229. IOS, 1991.