Journal of Visual Languages and Computing (1999) 10, 215–243
Article No. jvlc.1998.0109, available online at http://www.idealibrary.com on IDEAL®

The Role of Graphics in Parallel Program Development

KANG ZHANG*†, TOM HINTZ‡ AND XIANWU MA§

*Computing Department, Macquarie University, Sydney, NSW 2109, Australia,
E-mail: kang.zhang@mq.edu.au
‡School of Computer Sciences, University of Technology, Sydney, NSW 2007, Australia
§Fujitsu Australia Software Technology, French Forest, NSW 2150, Australia

Accepted 16 November 1998

Graphical visualisation plays an important role in parallel program development. Researchers have proposed and developed many visualisation tools that assist the development of parallel programs. A number of graph formalisms or notations have been used to visualise various aspects of parallel programs and their executions. This paper attempts to classify and compare these graph formalisms and notations which provide different information at different stages of parallel program development.

Keywords: Parallel programming, visual programming, program visualisation, graph models, debugging.

1. Introduction

VISUAL STRUCTURES AND RELATIONSHIPS are much easier to reason about than similar linguistically described structures [1–3]. Consequently, there have been many visualisation systems developed in dealing with parallel systems. There have been several interesting taxonomies and surveys of systems that use computer graphics to assist program development [4–10]. Yet limited work has been done on classifying or summarising the role of computer graphics in parallel program development [11], although an increasing number of parallel programming environments that support graphical visualisation have been developed. The aim of this review is to systematically examine the role of computer graphics in different stages of parallel program development. We restrict ourselves to the use of graphics to aid the reasoning and understanding of parallel programs and their executions. As Miller puts it: ''visualisation should guide, not rationalise. 'Guide' means that the visualisation leads you to discover things that you did not already know. 'Rationalise' means that it lets you illustrate things that you already know'' [12].

We propose a model that classifies parallel program visualisation systems according to the purpose of using graphics at different stages of parallel program development. This

† Corresponding author.

1045-926X/99/030215+29 $30.00/0

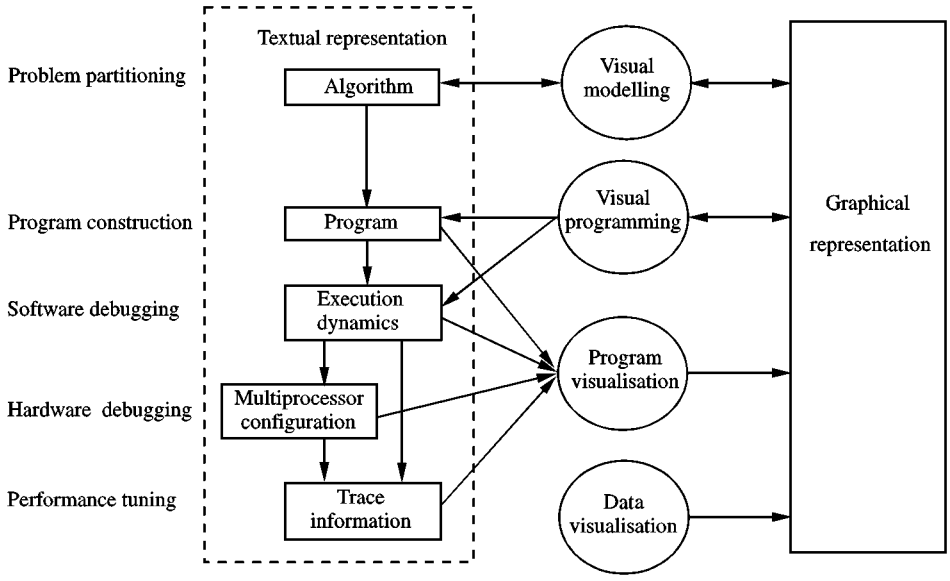

Figure 1. A classification model of graphics used to aid parallel program development

classification method relates to the definition of 'scope' in Myers' taxonomy [8], and 'aspect' in that of Stasko and Patterson [10], but tailored for the parallel program development cycle. We find that there are four main stages where computer graphics plays a guiding role: problem partitioning, program construction, debugging, and performance tuning. As illustrated in Figure 1, at different stages, graphics plays different roles and may have different notations. Any of the four stages may be entered more than once during the development life cycle. The purpose of using graphics in different iterations of a cycle of the same stage may vary depending on the progress with the program development.

In the first stage, the user may use a graphical editor to build one or more graphs to simulate or decompose an application problem. A variety of system properties can be analysed through simulation or prototyping of a given problem. This type of system is called a *visual modelling system*. In the next stage, a visual formalism is used to represent and construct a graphical program which is directly executable with its operational semantics; or to draw a diagram and then generate an intermediate textual program of an existing language syntax for execution. The program may be optimised during this design stage with the aid of graphics. This type of system is called a *visual programming system* [8]. Another type of system, a *program visualisation system*, allows program graphs to be generated from textual programs. At the debugging stage, the execution dynamics at the program level and its reflection on the multiprocessor configuration at the machine level may be visualised. Program animation techniques and some structure- or process-oriented diagrams may be used to help debugging. The final stage of parallel program development is usually concerned with performance tuning. The trace information recorded during the program execution can be visualised using various graphical notations which meaningfully

depict the program behaviour. We will call the visual formalisms and graphical notations *graph models*.

The graphical assistance at both the debugging and tuning stages is also classified as program visualisation. Using different conventional visual formats, such as bar charts, to visualise the pure statistical data that profiles the performance of a program is called *data visualisation*. We will call this class of visual forms *graph charts*. A major difference between a graph model and a graph chart is that every visual notation in the former depicts some aspect of parallel computation, while the visual notations in the latter simply reflect statistical quantities.

The next section will identify the main activities in a typical parallel program development cycle, and briefly mention how graphics has been involved in these activities. Section 3 examines several graph models for their roles in different stages of parallel program development. We will focus on some of the models that have been most widely accepted and used, and we describe some example systems that implement these graph models with specific visual notations. Section 4 proposes a set of criteria for evaluation and compares the graph models. Since many parallel program visualisation systems use graphical charts to show different aspects of program performance and profiling, Section 5 is devoted to the general overview of various ways graph charts are presented. Section 6 discusses the roles of animation, colour and three-dimensional effects in developing parallel programs. The final section concludes the paper by discussing the future role of graphics in developing parallel programs.

2. Graphics in the Development of Parallel Programs

Parallel programming refers to the programming of multiprocessor systems. The development of parallel programs is much more difficult than that of sequential programs. In addition to the use of parallel programming languages, programmers require design or modelling tools to help them partition parallel activities and evaluate design alternatives. Ideally, program design tools also help in generating parallel source codes from specifications. However, parallel programs are typically generated through traditional editing tools. During the execution of a parallel program, monitoring tools can be applied to collect execution information, which is then analysed for debugging and performance tuning. It is desirable to integrate these various tools to provide a unified parallel programming environment so that the user does not need to switch from one tool to another when developing a single program [13]. An important aspect of parallel programming tools is the use of graphics to visualise the behaviour and performance of programs and to make the development of parallel programs easier and more productive.

2.1. Problem Partitioning

Graphics introduced at the problem partitioning stage are used to decompose or parallelise a problem at a high level of abstraction. Several key characteristics of a given problem can be identified from users' requirements using the functionality model, the behavioural model, the structural model and the property model. The user may use a graphical editor to build one or more such models to simulate or decompose an

application problem. A variety of system properties can be analysed through simulation or prototyping of a given problem. This type of system is called a visual modelling system. The primary purpose of this stage is to model a given problem, find a feasible solution or algorithm to the problem, and partition the algorithm for multiprocessing. The generation of executable code is not usually required at this stage. Graph notations used at this stage are typically simple with a small number of graphical primitives that represent high-level concepts, typically algorithmic constructs. Several popular graph models are used in problem partitioning or decomposition, such as Petri nets [14], and Statecharts [15].

2.2. Program Construction

The use of graphics for program construction may serve two reciprocal purposes, for instance, using a graphical editor supporting a pre-defined notation to generate visual programs to perform desired functions, or displaying the structure of the program graphically to show either data dependencies or control flow. The former purpose is classified as visual programming and the latter program visualisation. Visual programming techniques can be used to build a graphical program to be directly executable according to its operational semantics; or to draw a diagram and then generate an intermediate textual program of an existing language syntax for execution. The program may be optimised during this stage with the aid of graphics. There are several widely used graph models for constructing programs, such as program dependence graphs [16], process graphs [17], and form-based notations [18], to be discussed in Section 3. Many systems use more than one type of graph model, each of which may represent a different conceptual model of the problem.

2.3. Debugging

Parallel programs may be debugged at both software and hardware levels. At the software level, debugger displays provide various views of parallel program states. These include inter-process communication, procedure call order, etc. At the hardware level, debugger displays show graphically run-time characteristics of a multiprocessor system when executing a particular program. These characteristics include the access pattern of memory, inter-processor communication, processor utilisation, etc. The execution dynamics at the program level and its reflection on the multiprocessor configuration at the machine level may be visualised. Program animation techniques and some structure- or process-oriented diagrams may be used to help debugging. The trace information recorded during the program execution can be visualised using various graphical notations which meaningfully depict the program behaviour. Graph models which have been used to assist debugging include dependence graphs, process graphs, causality graphs, and space–time diagrams.

2.4. Performance Tuning

The final stage of parallel program development is usually concerned with performance tuning. The graphical assistance at both the debugging and tuning stages is also classified as program visualisation. Using different conventional visual formats, such as bar charts,

to visualise the pure statistical data that profiles the performance of a program is called data visualisation. At the performance tuning stage, there are two major approaches to displaying performance data. One is event-oriented display, and the other is system-oriented display. Event-oriented display depicts all the interesting events of a program, including types of events and when they happen. This kind of display has a strong relation to the original program and mainly serves the purpose of debugging, as will be described in Section 3. System-oriented display visualises the execution details of a given parallel program in terms of the behaviour of the system components that support the execution. The displays may show the hardware status or operating system activities. The graphical notations and the data displayed are not directly related to the original program. To be discussed in Section 5, various graph charts have been used to visualise performance-related statistical data.

3. Graph Models

This section discusses several important graph models, in the order of their roles in visual modelling, visual programming and program visualisation. Some graph models may be used in more than one development stage. We regard a graph model as having a higher information bandwidth and thus being more efficient than another if it is useful in several stages. An efficient graph model serves as a single reference for multiple aspects of the program characteristics, and hence avoids forcing the programmer to switch from one mental image to another during development stages.

3.1. Petri Net

Petri nets have been one of the best-known visual modelling tools for concurrent systems [14]. A Petri net is a particular kind of directed graph, consisting of two types of nodes, called places and transitions, connected by arcs. In graphical representation, places are drawn as circles, transitions as bars or boxes. A place can be assigned with a non-negative integer through marking. If a marking assigns to place p a non-negative integer k, we say that p is marked with k tokens. Tokens are depicted as black dots at the marked place. In order to simulate the dynamic behaviour of a system, a state or marking is changed according to the firing rule. An example of the Petri net representing parallel or concurrent activities is shown in Figure 2, where transitions t2 and t3 begin at the firing of transition t1 and end with the firing of transition t4. Transitions t2 and t3 are concurrent as they are causally independent, i.e. one may fire before or after or in parallel with the other.

High-level extensions to Petri nets have been proposed to handle complex systems [19, 20]. For example, places may be able to hold structured objects [21]. Miriyala *et al.* use *predicate transition nets* (*PrT-nets*) as a visual specification language for observing and visualising the behaviour of actor programs. The *actor* model abstracts concurrent computations in distributed systems. The distributed computational agents, known as actors, communicate with each other by sending messages. The communication is asynchronous. An actor has a mail address and a behaviour. Actors' mail addresses can be communicated within messages, which implies that the inter-connection topology of actors is dynamic.

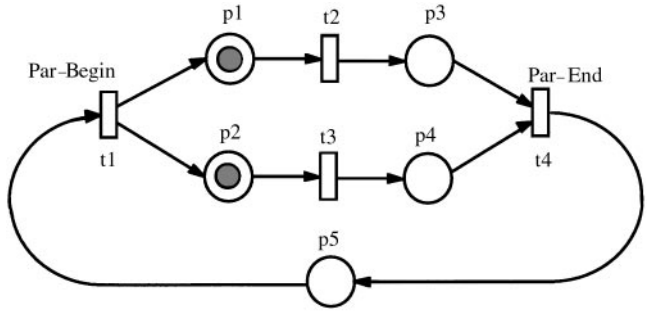

Figure 2. A Petri net representing parallel computation

A PrT-net is a high-level Petri net. It differs from a Petri net in that the tokens in a PrT-net may carry data. A token is defined as a tuple that contains 0 or *n* data fields. These fields may carry data elements which identify the token. A place in a PrT-net specifies a condition, called *predicate*. A transition changes the current predicate that is applicable to tokens. The firing of a transition indicates a change of condition.

Figure 3 shows the canonical bank account example adapted from Miriyala *et al.* [29]. There are two managers in the bank, represented as tokens M1 and M2 located at the place bank-manager. There are three tellers as tokens T1, T2, and T3 which mark the place bank-teller(T). A bank manager services customer requests for opening checking and savings accounts by informing the tellers the account numbers of the newly created accounts. In Figure 3, customer A hands in a request to manager M1 for opening an account (create-acc(A, M1)). Manager M1 creates a checking account a1 and a savings account b1. Upon receiving the tokens a1 and b1, the tellers fire a transition that moves the tokens to the part of the net representing the checking and savings accounts. A teller may receive transition requests from the customer once an account has been created (T1 receives a request to deposit some amount into the checking account a1 in Figure 3).

Places are OR-nodes and transitions are AND-nodes. Two types of tokens may exist in a net: actor token (e.g. a1, b1, M1, M2, T1, T2, T3), and message tokens (create-acc and checking requests). Each place in a net represents the behaviour of an actor. A transition is fired when an actor token's mail address matches a message token's mail address on each of its input arcs. This is like the firing rule in a dataflow graph (Section 3.2). Synchronisation constraints in actor programs are visualised by moving a token from one part of the net to another. It is suggested that using PrT-nets to visualise actor programs may help debugging parallel systems.

Petri nets specify system behaviour based on the observation of causal independence, and have been widely used to model concurrent systems and verify their communication and synchronisation requirements. The places and transitions that participate in a Petri net all have to be predefined. Therefore, the configuration of a Petri net is static. Nets can handle process distribution as they treat both state and action equally and thus enable mapping from a Petri net's elements onto processes and data, and further onto processors. The graphical notation of Petri nets can be easily implemented in a concurrent program visualisation tool. The main disadvantage of Petri nets and their extensions

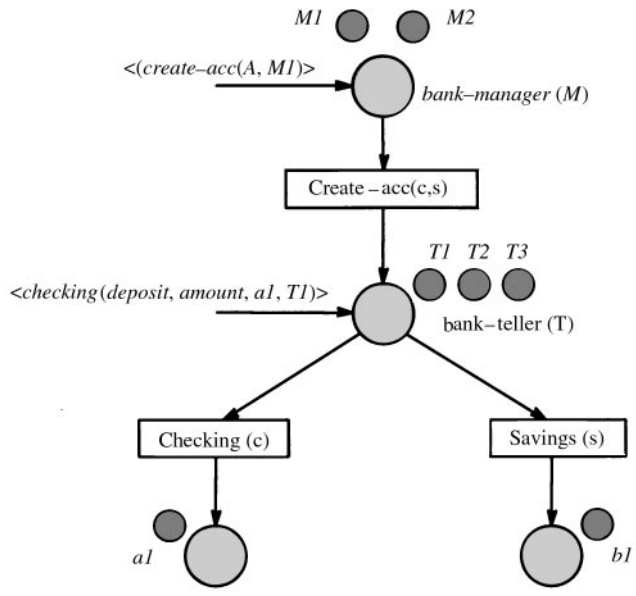

Figure 3. A bank account example in predicate transition net

is that they do not support hierarchical specification for programs of large scale. Therefore, they usually have lower scalability than other graph models.

3.2. Dependence Graph

Graph models discussed in this subsection all explicitly visualise dependence relationships (control, data or hybrid) among program components. We call this class of models *dependence graphs*. A program dependence graph (PDG) makes explicit both the data and control dependencies for each operation in a program [16]. A node in a PDG may be an operation, a statement, or a higher-level grouping. An arc pointing to a node indicates that the node's operation depends on an input data from the source node of the arc, or on the control condition for the execution of the operation to proceed (see Figure 4). The PDG model is well-suited for program optimisation and program transformations (e.g. vectorisation) that require interaction of different dependence types. Dataflow graphs, from which the PDG notation is extended by adding control arcs, have long been used to exploit parallelism independent of computer architectures [22, 23]. A dependence graph, like a Dataflow graph, exposes potential parallelism. A visual program dependence graph (VPDG) [24], an extension of a program dependence graph (PDG), supports both code transformations when parallelising programs and mapping when visualising the code. Common features of dependence graphs include high scalability, direct correspondence with programs and semantic support for parallelism.

Graph nodes in PDGs may be specialised for some specific purposes when they are used for program construction. For example, heterogeneous network computing environment (HeNCE) [25] has different notations for conditional, loop, pipe and fan-in/out nodes, apart from normal nodes which are drawn as circles representing

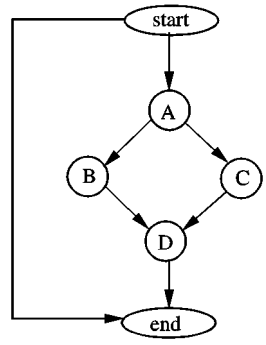

Figure 4. Program dependence graph

user-supplied subroutines. In computationally oriented display environment (CODE) [26], normal computation nodes (represented as circles) obey the dataflow firing rule, i.e., a node is executed once data is present on all inputs to the node. These nodes, known as SUC nodes, can also be encapsulated into 'subgraph' nodes (as boxes). The switch nodes in CODE are similar to the conditional nodes in HeNCE. CODE introduces an additional type of arcs, 'hyperarcs', to annotate shared data among computation nodes while having a semantics of preventing race conditions for accessing the data. A similar system, known as Phred [27], supports a graph model based on control-flow and data-flow graphs. A major feature of Phred is its capability of analysing the deterministic behaviour of a parallel program through the Phred graph grammar.

Phred is developed in an environment where the user can construct a Phred program expressed in control and dataflow graphs. A Phred dataflow graph represents data sharing among procedures, while a control flow graph shows the sequential flow of control, alternation, parallelism, and synchronisation. Figure 5 illustrates an example Phred program graph adapted from [27], which computes the values for the vector x in $Ax = b$ where A and b are matrices. The example program uses the successive overrelaxation technique. The two triangle nodes in the graph represent the start and end of the program, and square nodes are data repositories that contain shared variables. Access to the repositories is indicated by arcs connected to them and the connections form the dataflow graph. The filled circles marked input are the diverging conjunctive nodes and those marked output are converging conjunctive nodes. The disjunctive construct is similarly represented by a pair of nodes which are unfilled. The ellipsis node in the conjunctive construct indicates replications of the construct body. The large circle embedded with SOR is a task node, which reads the A, b and x repositories and computes a new value for its element of the x vector. The part of graph containing SOR is replicated multiple times, controlled by the test node. The replicated copies can be run in parallel.

Parallelism is implicitly represented in this example. But, in general, different branches of a common conjunctive node may be run in parallel. Therefore, Phred graphs can also visualise parallelism explicitly. A converging conjunctive node following a parallel computation will execute only after all the conjunctive branches terminate. It

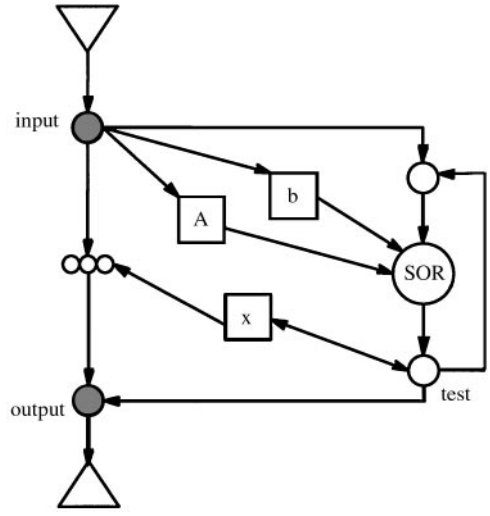

Figure 5. A Phred program graph which computes the vector x values in $Ax = b$

therefore plays a role of barrier synchronisation. Apart from visualising parallelism, Phred defines a formal way of finding parallelism through a straightforward marking scheme. Phred's static analyser is able to parse the graphs to determine whether they are syntactically correct, and to analyse syntactically correct graphs for determinacy.

Apart from the graph nodes shown in Figure 5, there are also nodes annotating pipelined processes. Determinacy is checked by analysing parallel writes and reads to the shared repositories. Phred graphs are scalable because they allow task nodes to be further defined as conjunctive, disjunctive, pipe or loop constructs.

In general, dependence graphs are scalable as their nodes can be program components of any granularity and arc-connecting nodes specify data or control flows that are not constrained by granularity. Parallelism is naturally shown through dependence relationships among computational nodes. Code generation can be easily achieved if graph nodes support fine granularity [28]. Although several dependence graph models have been defined on some graph formalism, such as the Phred graph grammar, there has been no commonly accepted formalism for general dependence graphs. Visual vocabulary, as well as graphical notations, vary with different dependence graph models.

3.3. Space–Time Diagram

Space-time diagrams, first proposed by Lamport [29], overcome the disadvantage of animation that does not clearly show patterns of behaviour occurring across time. A space–time diagram presents the execution of a parallel program in a two-dimensional display with time along one direction (vertical or horizontal) and individual processes along the other (see Figure 6). It provides a compact view of the event history by showing the temporal relationships and even race conditions among processes [30, 31]. A variation of a space–time diagram, called an event–time diagram [32], can display the

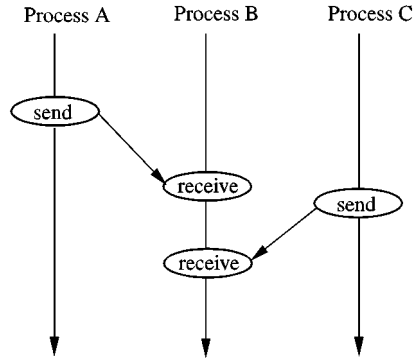

Figure 6. Space–time diagram with three processes

proportionality of the synchronisation and communication costs. The passive space and time view, extended from space–time notation, can show not only the inter-process concurrency but also potential intra-process concurrency [33]. In a passive space and time view, a process is seen as a partially ordered set of events such that a point in space is a passive entity which does not order events that occur at it, and the order is determined by the interaction among events.

A concurrency map is another type of space–time diagram as it also emphasises concurrency and interprocess dependencies through time and space [34]. In the example concurrency map shown in Figure 7, three processes, `A`, `B`, and `C`, are executed in parallel and event `A2` of process `A` sends a message to event `B2` of process `B`. It can be seen that the block containing `A1` and `A2` must occur before the block `B2`, while it may be executed concurrently with the block containing `B1`. Horizontal grid lines represent intervals of time. The events that appear in different columns but in the same row can occur concurrently. A communication between two events of different processes is shown as an arrow from the sending event to the receiving event. The concurrency map is useful in analysing the partial ordering of communication events which determine a program's behaviour.

Space–time type of diagrams can also be used to reveal the communication/computation ratio for performance tuning [35, 36]. A performance tuning example using space–time diagrams is shown in Figure 8, which visualises the execution of a CMMD program on the Connection Machine CM-5 [30]. Each vertical line, called activity line, under a process number in the display represents the process active time. Its length indicates the execution time of the process. Slanted lines represent communication events between the processes to which they connect and are called communication lines. Since the communication is synchronous in CMMD, the starting and ending points of a communication line correspond to the times of 'ready-to-receive' and 'ready-to-send', respectively. The slope of a communication line indicates a synchronous delay when one of the two communicating processes is ready before the other. The dark strip at the top portion of the activity line under process 0 represents an input/output event. The length of the strip indicates the I/O usage time.

Figure 8 is a simple example demonstrating the use of space–time diagrams for performance tuning. The program being tuned is the 'all pairs shortest path' (APSP)

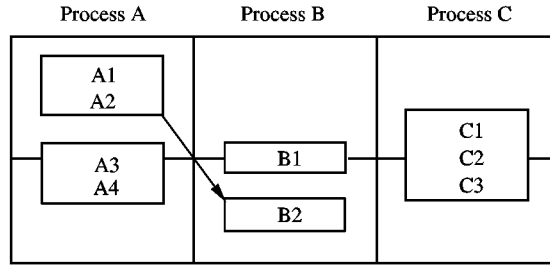

Figure 7. Concurrency map

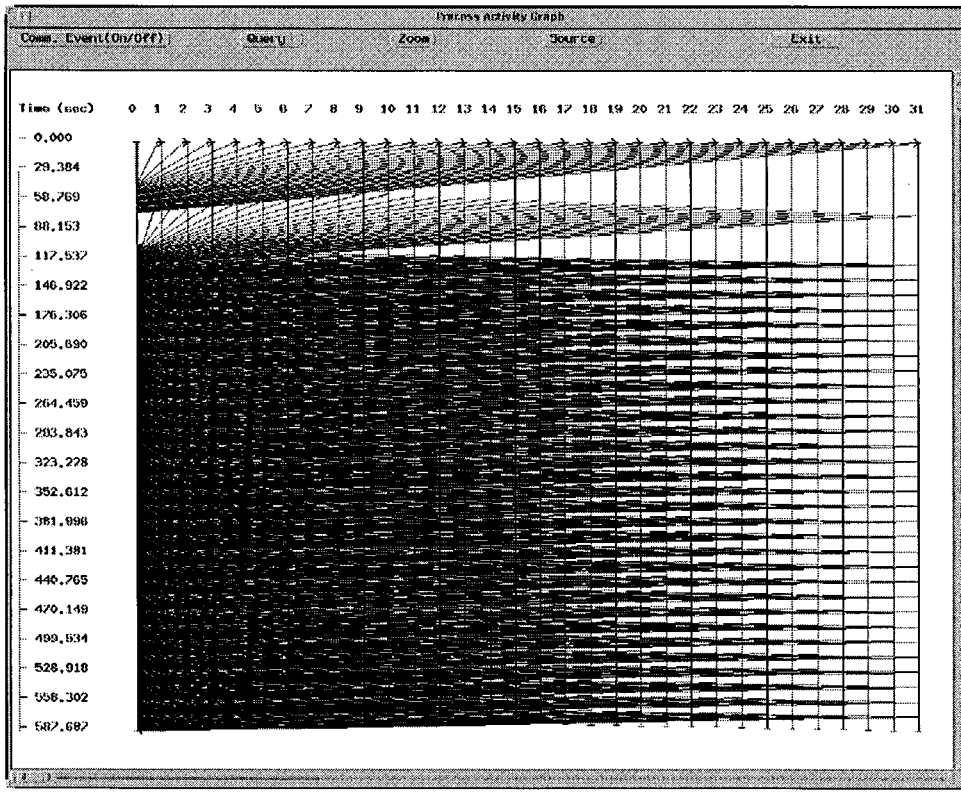

Figure 8. A space–time diagram for performance tuning of a CMMD program

problem, which is to find the shortest path from a source to a destination for all pairs of vertices. The example program solves 1024 vertices on a CM-5 with 32 processors. Each processor is allocated with a worker process that solves 32 ($= 1024/32$) pairs of vertices using a sequential algorithm. A master process performs the I/O operation by reading the problem matrix from a data file and distributing it to the workers. Each worker solves its own portion of the problem. Whenever it obtains a solution for a pair of vertices, it sends the solution to the master process.

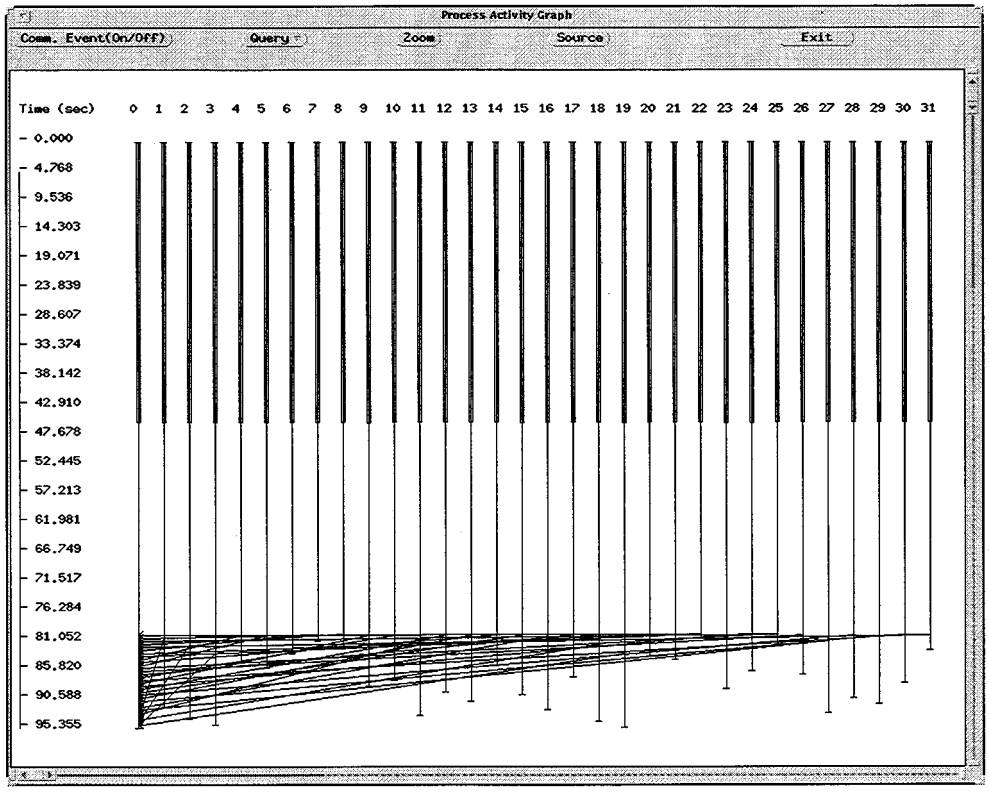

Figure 9. Space–time diagram showing the improved performance

The figure shows that process 0, i.e. the master process, delays the communication for certain amount of time due to the I/O operation. It is also clear that the communication overheads among the processes are high. Each process performs little computation but much communication. Therefore, the processing power is very wasteful. One improvement is to let each process send the solutions to the master process when it finds the shortest paths for all the 32 pairs of vertices, rather than send one solution at a time for each pair. This will substantially reduce the communication overheads. Another improvement is to reduce the I/O delay by letting each process read the problem matrix independently. The improved performance using the same space–time diagram is illustrated in Figure 9, where the computation is more concentrated and the total execution time is significantly reduced.

One disadvantage of this type of diagrams is that the information displayed for each process at any given time is limited. The concept of multiple threads communicating along a time line is not well suited to other parallel computation models, notably shared memory model.

3.4. Other Models

We consider the Nassi–Shneiderman type of diagrams [18] as form-based notations as they are essentially forms with characteristic layouts filled with corresponding program

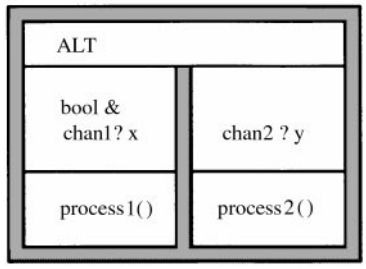

Figure 10. A form-based notation showing non-deterministic choices

constructs. The original motivation of designing Nassi–Shneiderman diagrams was to facilitate structured programming. GRAIL shows the parallel program structure written in the language Occam through form-based diagrams [37], where the vertical dimension represents sequential execution and deterministic choice (IF), and the horizontal dimension indicates parallelism and non-deterministic choices (ALT). A non-deterministic choice construct defines that the first arrived input will activate the process with which the input is associated. Figure 10 illustrates a notation adopted in GRAIL for an ALT process. If the input on chan1 arrives first and also bool is true, the input will be assigned to *x* and process1() will proceed for execution. If the input on chan2 arrives first, it will be assigned to *y* and process2() will proceed for execution. Communications are shown through additional arrows.

BLOX diagrams [38], based on strict-spatial representations, can also be classified as a type of form-based notation since a BLOX diagram is constructed from a constrained layout of building blocks that look like 'jigsaw puzzle' pieces. Each building block represents a language construct and visual appearance of a block suggests to which other blocks it may be connected. Extending the BLOX notation to support parallel programming, VPE introduces communication and synchronisation related blocks, such as send, receive, and barrier synchronise [39]. As shown in Figure 11, a VPE diagram closely resembles its textual counterpart, like a GRAIL diagram. The 'lock and key' metaphors play the same role as the form layout in GRAIL that facilitate the contruction of programs. VPE supports scalability through its four levels of abstraction. When the four hierarchical levels are flattened to a single level and the communication blocks are used to establish the time-dependent lines, a VPE diagram can be easily converted to a concurrency map (as seen in Figure 7).

In general, form-based graphs are useful for showing the program structure but do not convey computational activities. They are usually scalable but more textual than diagrammatical. Obviously, an advantage of being textual and structural is that code generation is very easy and sometimes straightforward. Therefore, form-based graphs can be used at program construction stage. Since they directly visualise language constructs, their graphical notations support small granularity at the low level and the number of such notations (i.e. visual vocabulary) corresponds to the number of language constructs. The visual semantics of a form-based model typically corresponds to the operational semantics of the language being visualised, and so the formalism of this model is only as strong as the underlying language.

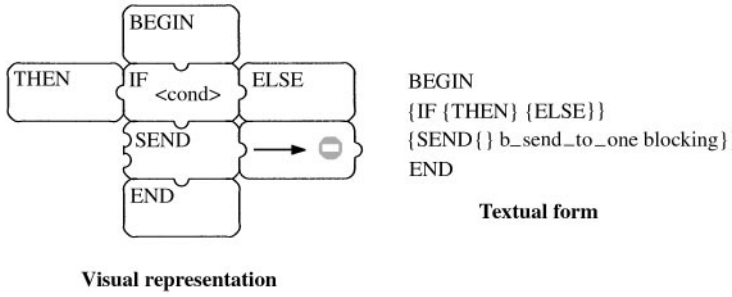

Figure 11. A VPE diagram and the textual program it represents

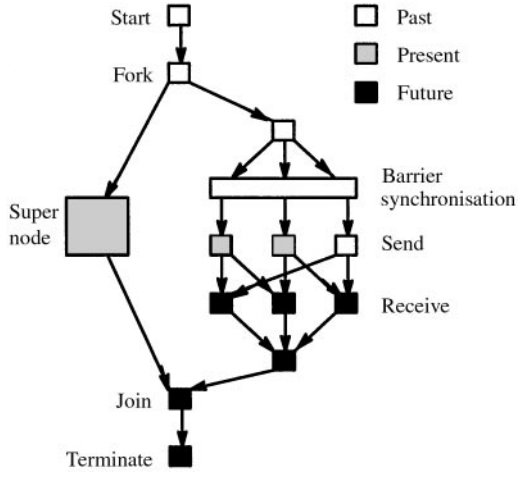

Figure 12. Causality graph

The notion of causality graph, proposed by Zernik [40], provides a logical view of a program's execution, including communication and process creation. It is suitable for a language that makes communication and synchronisation explicit. Causality graphs use time to indicate the so-called *causality order*, that is, which operation can affect or cause another operation. They can be collapsed into so-called Supernodes which can be recursively collapsed, as shown in a simple example in Figure 12. New threads are generated by Fork nodes and terminated by Join nodes. Other types of nodes include Message send, Message receive, Barrier synchronisation, User-defined and Processing (for sequential computations). Any two nodes can be connected by either a Control arc which specifies the execution sequence of the two nodes, or a Message arc which shows the communication relationship of the two nodes. The execution sequence of Past, Present and Future is shown through different colours or grey levels.

Causality graphs achieve high scalability through collapsing using an algorithm for directed-acyclic-graph abstraction [40]. They explicitly reveal coarse-grain parallelism through causality relationships among computation nodes. The model provides a useful program visualisation means for debugging multi-threaded parallel programs.

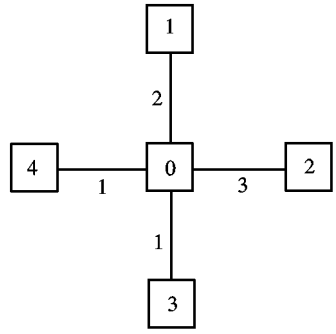

Figure 13. Process graph

Process graphs are for visualisation of coarse grain parallel programs, typically with nodes representing processes, and arcs representing inter-process communication links. In the process graph shown in Figure 13, numbers inside the nodes are process identifiers and those next to the links are communication costs. The process graphs used in Millipede [17] are constructed of four types of objects: processes, input- and output-ports, and channels which connect processes. Lewis *et al.* developed Task Grapher that can model a parallel program as a task graph [41] (a variation of process graph) and schedule tasks onto processors using scheduling heuristics after choosing a desired multiprocessor topology. A task graph has nodes representing tasks, each marked with an estimated execution time, and arcs representing communication channels, each marked with the size of the message to be transmitted. Process nodes may be iconised by meaningfully depicting them as pictures, as shown in VISTOP [42] and GILT [43]. An alternative view of the process graph is a tree structure of processes, whose hierarchy shows from which process a given process was created or forked [43].

All the systems using the notation of process graphs, such as the one in Figure 13, display multiprocessor topologies in a similar way as processor graphs [44, 45], which depict processors as nodes and interprocessor links as arcs. This, indeed, reflects the common approach to the design process of multiprocessors, in which the topological configurations of processors are visual by nature. The highest level of the notation describes the mapping of processes onto processors and interprocessor connections. The systems using process graphs, therefore, support hierarchical design and are thus highly scalable. They usually animate dynamic activities directly on processor graphs (see Section 6).

4. Comparison

Although many graph models have been proposed to aid parallel program design, development and understanding, previous research has not focused on defining criteria for evaluating the usefulness of these graphs in the development life cycle. We propose a set of criteria, which we believe to be useful in evaluating specific roles of the graph models. The criteria are independent and thus orthogonal to each other. These include the theoretical foundation (formalism), the exposure to parallel computation (parallelism

and functionality), the semantic gap between the graphs and programs (code generation), the measurements of graphical properties (scalability, vocabulary, intuitiveness and animation) and the level at which parallel programs can be visualised (granularity). There is no conflict between the criteria as they all can be applied to any single graph model.

- *Formalism*—whether a graph model is built on a formal basis so that certain program properties are provable or derivable from the graphical syntax and underlying semantics.
- *Parallelism*—whether a graph model supports parallelisation directly or indirectly when partitioning a system into multiple subsystems for multiprocessing; or how easily the viewer can identify parallelism from a given graph and whether parallel execution is supported by the graphical semantics.
- *Functionality*—what is the primary purpose of using a particular model at different stages of program development, which major property of a concurrent or parallel system a graph model can simulate and analyse, or which aspect(s) of the program behaviour a graph model can explicitly provide for debugging purposes.
- *Code generation*—how easily the intermediate artifact, such as formal description code, can be generated from graphical specifications, or whether it is easy (or possible at all) to generate the textual form of the program from the graphical layout constructed in the first stage. This criterion does not apply to the parallel visual languages which are directly executable on multiprocessor machines without the need to generate textual programs. The parallel languages of the latter category are rare at present and are not covered in this review.
- *Scalability*—whether a graph model is capable of describing a parallel computation at different levels of detail and/or through different observations while still capturing major characteristics at corresponding levels; whether it provides any mechanism for packing abstractions into computational units. (This means that specifications should provide certain operational definitions; or whether a graph model supports hierarchical constructions or aggregations so that different levels of program details can be viewed or constructed through 'zoom-in' and 'zoom-out' effects.) Ideally, a graph model should allow a given display space to visualise a parallel program of any size.
- *Vocabulary*—how many types of graphical primitives are required to construct a graph. The vocabulary is regarded as small if the primitives can be easily remembered (typically less than five), medium if it is necessary to remind the user through a menu or legend of the primitives, and large if the number of primitives is dependent on the number of language constructs or dependent on other factors.
- *Intuitiveness*—whether a graph model and its animation are intuitive and their implied meanings of program behaviour are straightforward to comprehend without textual interpretation. This in general relates to graphical perception, defined by cognitive scientists as 'the visual decoding of the quantitative and qualitative information which is encoded on graphs' [46]. No matter how impressive and beautiful a graph model may be, if the viewer cannot easily decode it then the decoding method has failed. (Our judgement of various graph models against this criterion is based on our own experience and thus may be biased.)
- *Animation*—which particular characteristic is best presented with animation in a graph model for debugging, optimisation, or tuning purpose. This criterion distinguishes

one graph model from another by focusing on a dynamic characteristic that can be visually described through animation.

- *Granularity*—how the primitive graphical components correspond to the complexity of operations. In other words, whether a graph node represents an atomic operation, program statement, a subroutine, a process of computation, or a processor. Graphs of large granularity are suitable for program design and partitioning, or for high level (e.g. communication) debugging; and those of small granularity are suited for code generation or code level debugging.

According to our classification discussed earlier, the general nature of a graph model in assisting parallel program development may be classified as visual modelling (VM), visual programming (VP) or program visualisation (PV). A graph model may be used for both visual modelling and visual programming, or both visual programming and program visualisation. Various forms of graphical charts play the role of data visualisation (DV). We have not identified any single graph model or chart which can support more than two of VM, VP, PV and DV.

Table 1 lists various graph models and compares them against the above criteria. The first two rows are not strictly criteria. They serve the purpose of our classification.

The first stage of parallel program development, problem partitioning, can be assisted by visual modelling, for example using Petri nets. Petri nets have a strong theoretical foundation which allows various concurrent properties to be modelled at the program design and partitioning stage. They can, however, hardly be useful at any later stage of program development. There is a trend towards using object-oriented techniques to model and design parallel computations. When an object includes not only its graphical appearance, but also transition statements defining behaviour towards other objects, it becomes an active object [47]. Although this approach supports structured modelling of parallel computations and integrated user interface, it suffers from the same problems as Petri nets, such as the difficulty of code generation.

Visual programming techniques can be used for program construction. When constructing parallel programs, process graphs and form-based notations support coarse and fine granularities of visual programming, respectively. Program dependence graphs support visual programming at different granularities as their nodes can depict processes as well as atomic operations. All the three graph models are easily scalable. The arcs of program dependence graphs and process graphs maintain the same visual semantics for different levels of abstraction, while form-based Notations may only show the execution order, and thus partial control flow of the program at the level above the language constructs. Therefore, in terms of supporting various levels of abstraction with high scalability as well as easy code generation, program dependence graphs are the best for constructing parallel programs.

Dependence graphs, process graphs, causality graphs and space–time diagrams have been used for program visualisation to help debugging. Program dependence graphs and causality graphs support the formulation of parallel debuggers in which a single notation can express the program structure (as for visual programming), the expected program behaviour, as well as the actual execution behaviour [13]. They provide intuitive animation facilities that visualise the mapping of the actual program execution to the expected behaviour. Process Graphs are useful for building and debugging coarse grain parallel programs, and particularly helpful for mapping processes onto multiprocessors.

Table 1. Comparison of graph models supporting parallel program

Graph model	Petri net	Form-based	Dependence graph	Process graph	Causality graph	Space-time diagram
Visual class	VM	VP	VPıPV	VPıPV	PV	PV
Stage	Modeling	Construction	Construction & debugging	Construction & debugging	Debugging	Debugging & tuning
Formalism	Strong	Weak	Weak	Weak	Weak	Weak
Parallelism	Indiredt	Direct (low level)	Direct	Direct	Direct	Direct
Functionality	Design & verification	Structured design	Optimisation & transformation	Design & mapping	Communication & causality	Communication & synchronisation
Code generation	Very difficult	Easy	Easy	Difficult	Easy	Impossible
Scalability	Low	High	High	High	High	Low
Vocabulary	Small	Large	Medium	Small	Medium	Small
Intuitiveness	Low	High	High	Medium	High	Medium
Animation	State transition	Execution sequence	Critical path	Load balancing	Time distribution	Race-condition & communication
Granularity	Medium	Small	Flexible	Large	Medium to large	Large

A space–time diagram, on the other hand, can only be used at the debugging stage to show the communication behaviour of a parallel program. Its advantage over the other graphs is that it shows the history of the execution through a time line so that the user can compare different phases of the program execution.

Space–time diagrams have also been widely adopted for performance tuning, where the time line plays a critical role for showing the performance losses, such as synchronisation delays. However, they only visualise communication- or synchronisation-related program behaviour. Therefore, many performance tuning tools also use data visualisation techniques, through various graphical charts, to reveal other aspects of the program performance.

5. Graphical Charts

Without any explicit computational meaning, graphical charts can be used to visualise statistical data. The relationship between the performance of a parallel program and the meaning of a graph chart can be established through their semantic context and reinforced by revealing the correspondence of individual visual components to the program components [36]. This is often referred to as *parallel performance visualisation* [48]. Heath, Malony and Rover have given a systematic overview on the issues in performance visualisation, and compared the similarities and differences of performance visualisation with scientific visualisation [49].

Many parallel program visualisation systems provide a set of charts for profiling program characteristics and hardware utilisations. They form system-oriented displays using various types of statistical information. Their primary purpose is to illustrate statistics in various characteristic formats. These include pie charts, bar charts, dials, meters, lights (analog to the graphical equaliser in an audio recorder), scales, plots, histograms, synchronisation graphs, etc. In general, line charts are best for illustrating trends and interactions, whereas bar charts are best at illustrating either differences between dependent variables for particular independent variable values or just one specific dependent variable value [50]. Therefore, line charts are typically used to show scalability and speed-up of a parallel program or system, and bar charts are mainly used for comparison of, for example, various optimisations of a program, different systems implementations, execution time of individual program components or different phases of a program execution.

Graph charts give quantitative measurements of a system's performance, such as cache misses, memory and interconnection-network traffic, page faults, entering and exiting parallel sections, calling and returning from procedures, communications, acquiring system resources, synchronising, message passing, etc. There are numerous example systems in this category. These include the so-called link message load display where the darkness of the elements in a processor matrix indicate the communication load between the two processors involved [51]; processor utilisation displays using Gantt charts [41] and Kiviat diagrams [35]; and task-processor charts which display events of tasks occurring in different processors across a time axis [52, 53].

ParaGraph uses a comprehensive range of diagrams to display the program performance from various angles [35]. Figure 14(a)shows a Kiviat diagram that depicts an

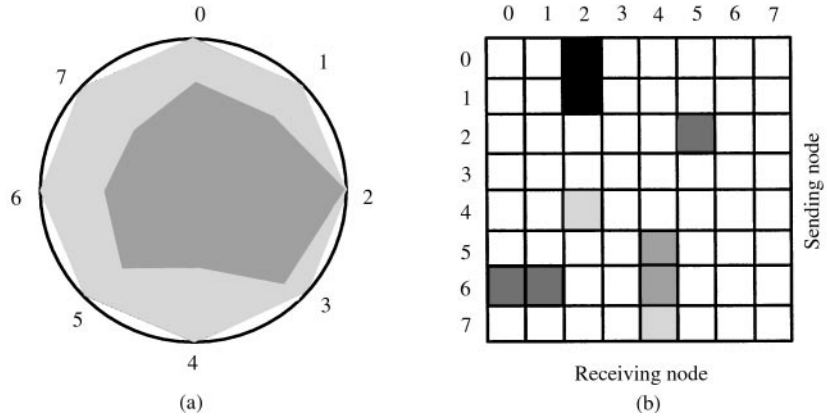

Figure 14. Some graph charts used in ParaGraph. (a) A Kiviat diagram showing processor utilization; (b) a communication matrix

individual processor's utilisation and the overall load balance across processors. Eight processors are represented as spokes on a wheel. The average fractional utilisation of each processor determines a point on its spoke. The centre of the wheel denotes zero, meaning that the processor is completely idle; and the outer rim denotes one, meaning completely busy. Current utilisation is shown in dark shading. The overall polygon shape reveals load balance. Low usage concentrates the polygon near the centre, while high utilisation results in a larger polygon whose edges are close to the perimeter.

We take another example of graphical charts used in ParaGraph, known as a *communication matrix*, similar to Lo's link message load display [51]. Figure 14(b) illustrates the communication pattern of eight processors. Messages are represented by squares in a two-dimensional array, whose rows and columns correspond to the sending and receiving processors of a message, respectively. A square is filled with a colour when a message is sent between the corresponding processors. The colour indicates the message size.

Time plays an important role in parallel performance visualisation, since computations unfold over time. How time is represented in a graphical notation usually determines the presentation style. There are two common methods of representing time, namely, explicit and implicit representations. When the chronological order of events is explicitly displayed along a time axis, the user can view the historical data and compare the variations of the data across a period of time [54, 55]. Other visualisation systems use time as an implicit parameter. The display space only shows the performance data without a time axis, while the changing of data is shown through animation. The display space can be linear, in two or three dimensions. At any instant in time, the display is a snapshot of the system's performance, and is therefore static. Animation can be conducted in the same manner as with video, in which the user may participate by issuing commands such as play, pause, fast-forward and rewind [56]. This technique has been successfully applied to algorithm animation.

6. Animation, Colour and Three-dimensional Effects

Unlike textual representations, which rely solely on the symbolic association of words, graphical displays allow arbitrary combinations of different graphical effects, including animated displays, multiple dimensions, colour hue and intensity, etc. Proper use of such combinations can result in a much more effective and fast understanding of a program's structure, behaviour and performance. This section discusses the role of animation, colour and dimensionality in developing parallel programs.

6.1. Animation

Program animation plays an important role in debugging parallel programs. Strictly speaking, animation means the display of a sequence of pictures, known as frames, over a period of time. If the change between any pair of neighbouring frames is small enough, the illusion of continuous motion is achieved. We loosely define program animation as any display of changing graphical objects that depict program states or data flow over a period of time. The most important aspect of animation is to choose a graphical notation that presents the collected data clearly and concisely. Animation can be performed on the existing graphical program structures built in the program construction stage. Almost all the models mentioned in Section 3 can support animation in one way or another. For example, in HeNCE's dependence graphs [25], nodes (for computations) drawn as circles are filled with different patterns when they are executed at different stages. During animation, HeNCE provides a legend to illustrate nodes of different patterns and their run-time meanings, such as 'start', 'executed', 'error', etc.

As another example, Visputer introduces two levels of animation: the software network shows the behaviour of processes and the hardware network shows the communication between processors (transputers in this case) [32]. The user sees communication events on channels between processes in the exploited view of the processor. An input and an output channel map onto a processor link in the hardware network. The effects of these events are also depicted on the hardware diagram. Link utilisation and possible contention on links can be identified during animation.

Figure 15 lists the graphical notation for communication animation. A filled circle appearing near the connection to a channel represents a ready-to-send (RTS) event, while an empty square indicates a ready-to-receive (RTR) event. A thick channel line symbolises that communication is taking place. All communication events shown in the software network are mirrored on the hardware network diagram. Each transputer may be linked to a maximum of four other transputers. This is depicted as four small blobs, called link connectors, on the inside edges of a processor node. A filled link connector indicates an RTS, while an empty triangular link connector represents an RTR event. Communication on a link is shown by making the link line thicker and filling the RTR link connector. Figure 16 shows a Visputer animation snapshot.

Many program animation systems support two animation modes, i.e. step mode and continuous mode. In step mode, the user has to click on a button to progress to the next event. This shows ordering of events and is particularly useful for debugging purposes, such as locating deadlocks. In continuous mode, there is a relationship between event

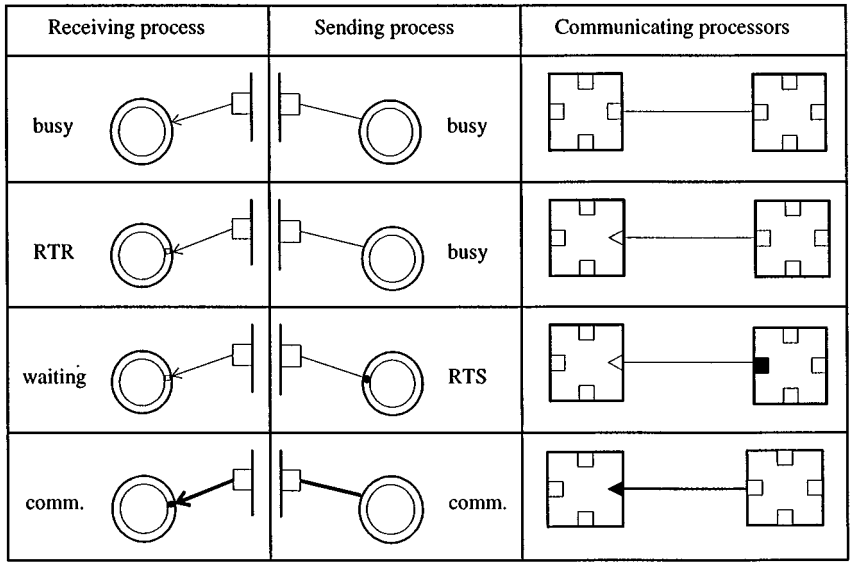

Figure 15. Graphical notation for animating communications in Visputer

time and the duration the event is displayed, allowing events to be observed 'as they happened'. The interval between any two consecutive display frames can be chosen to be equal or directly proportional to the actual time recorded for the program event involved. But such simple scaling of time intervals results in displays with long periods of inactivity punctuated by irregular bursts of complex updates [57]. Thus many animation systems allow the speed of frame sequencing to be adjusted according to user's visual needs. Continuous mode is useful for revealing time-critical information, such as when a particular event occurs, how long a communication event takes and how long the synchronisation delay of a communication is.

6.2. Colour Effects

Colour not only adds visual interest to a display but also helps the display provider and viewer to communicate effectively. Colour can be broken down into three aspects:

- *qualitative aspect:* the *hue* is what we usually mean by colour and is decided by the wavelength of the light;
- *quantitative aspect:* the *intensity* is the amount of light that is reflected or omitted, depending on the the grey that is added; and
- *depth:* the *saturation* measures the deepness of the colour and varies according to the amount of white that is added.

In parallel program visualisation systems, colour can be used to

- label a series of elements in a comparison group for distinguishing the elements or highlighting specific elements, such as a performance bottleneck in an execution period;

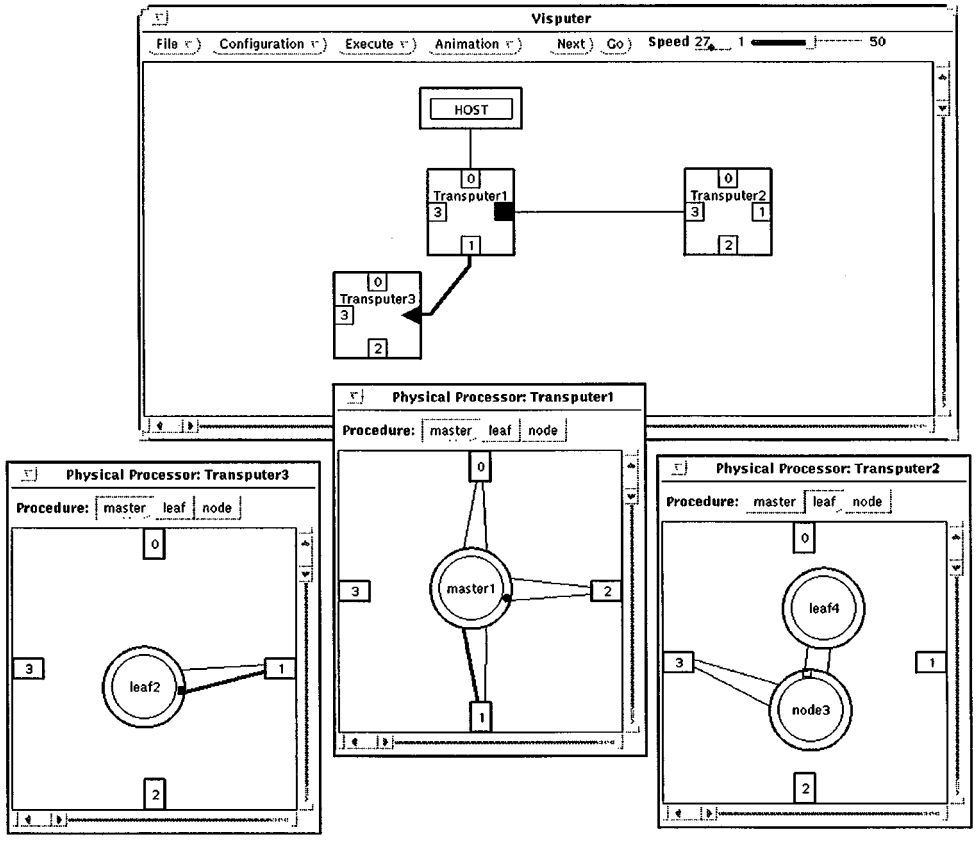

Figure 16. An animation snapshot in Visputer's user interface

- group elements to allow the viewer to compare two or more elements in different places, such as the execution behaviour of a program segment in several runs;
- measure quantity or rate of change through varying intensities or saturations, such as the work load in a processor and a number of calls to a particular function; and
- symbolise or imitate reality by associating colours with real-life meanings. For example, red could be used to indicate stop, warning or immediate attention.

For example, in ParaGraph's displays of processor utilisation and communication, a communication line's colour indicates whether a given message corresponds to a physical link in the topology, or must be routed through intermediate processors [48]. The use of colour and shape helps highlighting whether the program's communication pattern matches well with the underlying topology. The experiences with Avatar [58] suggest that colour coding the cubes in a scattercube (discussed above) provides rough guidance about the user's position, and is best used in conjunction with an opacity control for the cube faces to focus the user's attention.

In general, visualisation display must be looked at as a whole, because the perception of hue depends partially on the surrounding colours. Also, people's perception of colour

varies greatly and a sizable percentage of the population have difficulty in distinguishing certain colours, typically red and green. If used improperly, colour variations can become an impediment to clear communication. Psychological studies [2] suggest that we should use colours that are well separated in the spectrum, adjacent colours (red and blue should not be adjacent) in a graph should be of varying brightnesses, i.e. different intensities, the background should be in a warm colour, and colour hue should not be used to represent quantitative information.

6.3. Three-dimensional Effects

The cues the brain uses to assign depth allow us to see the world as it is—in three dimensions. When the brain, relying on the same cues, interprets a two-dimensional surface, it can be fooled into seeing a third dimension which is not present [2]. Three-dimensional displays typically use a *Z*-axis to indicate depth. They can be useful if they convey additional information for comparison for such as different versions of a program and different moments in time.

For example, the Gantt chart notation may be extended to a three-dimensional representation, with the extra dimension showing the durations of events that are visualised [59]. Three-dimensional graphs can be displayed in various styles to suit different situations. For example, a multiprocessor topology can be visualised intuitively in a three-dimensional display, as adopted in the parallel program visualisation tool MUCH [60]. MUCH visualises the message blocking possibilities and their solutions when using some routing schemes. Figure 17 is a MUCH display that shows the communication paths when an adaptive routing scheme is used in a three-dimensional nCube topology. The nodes in the graph represent processors and are connected via bi-directional communication links. Grey nodes are sending messages while the black node is receiving. The figure shows that when starting the transmission of two messages, one from node 0 to node 7 and the other from node 1 to node 7, the mutual blocking of the two messages can be avoided by using the routing scheme.

When visualising a large number of processes, a three-dimensional surface graph can be useful to emphasise the general trend and important peaks. It hides small variations of data. To visualise the variation between data points, line and bar styles serve better. Another style of three-dimensional display uses scattered points in a three-dimensional space to visualise the data distribution. The main purpose of using the scattered display is to view groups of data points. The groupings may be on a single line, in an area, or in a volume to manifest the behaviour of a single process or the interactions among a group of processes. A major drawback of a three-dimensional graph of line, bar, or point style is the interference and overlapping of lines, bars, or points, resulting from displaying a three-dimensional graph on a two-dimensional screen. Three-dimensional displays can also be harmful if they obscure the visual information or offer no added information.

Therefore, the designer of a program visualisation tool must decide whether adding the third dimension is worth the risk of losing some useful information. Three different approaches may be used to address this problem.

- Using colours to contrast the items in the least obvious dimension.
- Providing the user with the facility of flattening the 3-D view to 2-D views from various angles.

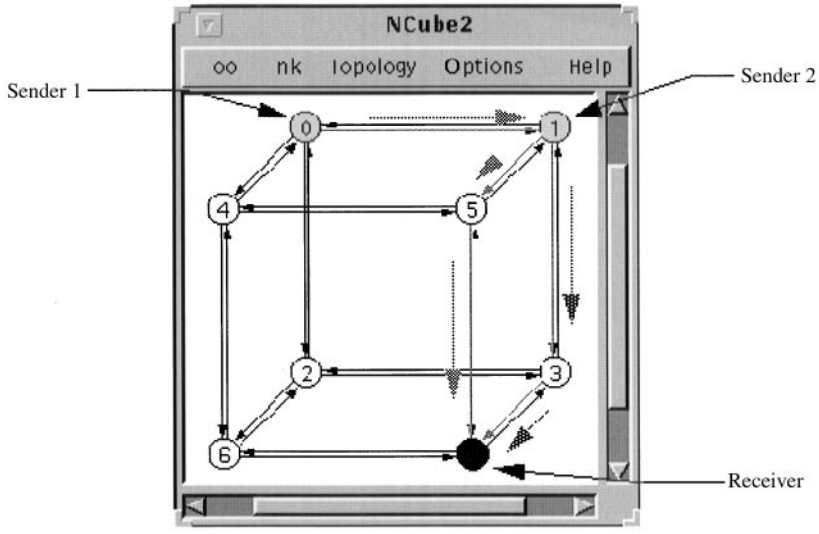

Figure 17. A three-dimensional nCube topology visualising communication in MUCH (courtesy of Dieter Kranzlmüller and Siegi Grabner)

- Allowing rotation operations on the 3-D display so that the user can view the display from any angle.
- Providing a virtual reality so that the user can virtually stand in the centre of the whole 3-D world and view any side of the display by simply turning around his/her viewpoint.

A virtual reality exploits humans' capability of changing the viewing perspective naturally through head and body movements in a three-dimensional world. Avatar [59] uses scattercubes, a three-dimensional generalisation of scatterplot matrices typically used in statistical software, to represent the parallel performance data in a virtual world. Each cube in a scattercube can be used to visualise a specific performance metric. The user can experience a stereoscopic view of the scattercube performance environment by wearing a pair of LCD shutter glasses or through a head-mounted display. The ability of highlighting multi-variate correlations in a scattercube assists the user in understanding and analysing typically high dimensional, irregular and sparse performance data.

7. Conclusion

Visualisation has been shown to be a useful way to understand parallel programs and their run-time behaviour. A graphical view delivers higher information content than the textual presentation [3]. Interactive graphics provides much better human–computer communication when developing parallel programs by using a judicious combination of text and static and dynamic pictures than is possible with text alone. The result is a significant improvement in our ability to understand parallel and concurrent programs. This is not to say that graphical forms are always better than textual forms. In fact,

graphical forms are strongly suitable for visualising quantitative relations, but not appropriate for conveying precise values. Tables are better than graphs for illustrating precise values. To convey relations among both conceptual information and absolute values, it is advisable to put a few numbers or words in critical places on the graph. Small or primitive program components or concepts can be understood more quickly in a textual form than in a graphical form.

In summary, this paper has discussed the features of various graph models used at different stages of parallel program development. Although there are many other graph models that have been used for parallel program development, the ones discussed in this paper are quite representative. We hope our comparison and summary information are useful in guiding the selection of appropriate graph models in designing parallel programming tools. There is clearly a need for more intuitive graph models which are easily understood by parallel programmers, rather than by parallel tool designers. A major challenge is to find a graph model which is not only generally intuitive and appeals to the scientific and engineering community, but also theoretically sound in terms of its expressiveness of parallel properties. Such a graph model should also be supportive across multiple development stages [61].

As program visualisation systems become more versatile, they may produce many types of graphical views, from which it may not always be easy to draw a conclusion about the program behaviour and performance. The main difficulty arises from the versatility itself and how a programmer can associate the meaning of one graph with that of another. The end-user does not want to digest the meanings of all the graphs and relate them to each other in order to understand a program's performance. For example, the graphics used at the tuning stage for data visualisation may not be directly related to the program. It is up to the user to interpret and associate the performance data to the original program. It is therefore highly desirable for the visualisation tool to provide automatic data analysing capability in order to give users, especially non-computer scientists, a meaningful interpretation of the data in terms of the program performance. The visualisation system should be able to reason about the program behaviour and performance to be able to derive a concise summary about them. We therefore believe that a program visualisation system should provide a final debugging summary at the end of the debugging stage, and a performance summary at the end of the tuning stage. Such a summary should be concise and easily understood by application programmers. In other words, it starts with text and finishes with text too, as

$$TEXT \rightarrow GRAPHICS \rightarrow TEXT$$

rather than finishing with many graphical displays. The graphical visualisation provides a high bandwidth of information interpreting the program execution and performance from various aspects, while the final textual summary focuses on the key problem with the program behaviour and the performance bottleneck.

Acknowledgements

We are very grateful to Margaret Burnett and anonymous reviewers for their invaluable comments and suggestions, which have greatly helped in improving the presentation.

References

1. R. Arnheim (1969) *Visual Thinking.* University of California Press, Berkeley.
2. S. M. Kosslyn (1994) *Elements of Graph Design.* W. H. Freeman and Company, New York.
3. J. Larkin & H. Simon (1987) Why a diagram is (sometimes) worth 10,000 words. *Cognitive Science* 11.
4. A. L. Ambler & M. M. Burnett (1989) Influence of visual technology on the evolution of language environments. *IEEE Computer* 22, 9–22.
5. S. K. Chang (1987) Visual languages: a tutorial and survey. *IEEE Software*, 29–39.
6. E. P. Glinert (ed.) (1990) *Visual Programming Environments: Applications & Issues.* IEEE CS Press, Los Alamitos.
7. E. P. Glinert (ed.) (1990) *Visual Programming Environments; Paradigms & Systems.* IEEE CS Press, Los Alamitos.
8. B. A. Myers (1990) Taxonomies of visual programming and program visualisation. *Journal of Visual Languages and Computing* 1, 97–123.
9. B. A. Price *et al.* (1993) A principle taxonomy of software visualisation. *Journal of Visual Languages and Computing* 4, 211–266.
10. J. T. Stasko & C. Patterson (1992) Understanding and characterizing software visualization systems. In: *Proceedings of the IEEE Workshop on Visual Languages*, Seattle, USA, 15–18 September, pp. 3–10.
11. C. Pancake & R. B. Netzer (1993) A bibliography of parallel debuggers. *ftp site cs.orst.edu*.
12. B. P. Miller (1993) What to draw? when to draw? an essay on parallel program visualisation. *Journal of Parallel and Distributed Computing* 18, 265–269.
13. J. C. Browne *et al.* (1995) Visual programming and debugging for parallel computing. *IEEE Parallel & Distributed Technology* 3, 75–83.
14. W. Reisig (1985) *Petri Nets, An Introduction.* Springer, Berlin.
15. D. Harel (1987) Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 231–274.
16. J. Ferrante, K. J. Ottenstein & J. D. Warren (1987) The program dependence graphs and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 319–349.
17. M. Aspnas, R. J. R. Back & T. Langbacka (1992) Millipede—a programming environment providing visual support for parallel programming. In: *Parallel Computing: From Theory to Practice* (W. Joosen & E. Milgrom, eds). IOS Press, Amsterdam, pp. 237–247.
18. I. Nassi & B. Shneiderman (1973) Flowchart techniques for structured programming. *SIG-PLAN Notices* 8, 12–26.
19. H. J. Genrich & K. Llautenbach (1988) System modelling with high-level petri nets. *Theoretical Computer Science* 13, 109–136.
20. J. Vautherin (1987) Parallel systems specifications with coloured petri nets and algebraic abstraction data types. In: *Advances in Petri Nets* (G. Rozenbery, ed.), Springer, Berlin, pp. 291–308.
21. S. Miriyala *et al.* (1992) Visualising actor programs using predicate transition nets. *Journal of Visual Languages and Computing* 3, 195–220.
22. J. B. Dennis (1975) First version of a data flow procedure language. Revised Comp. Struc. Group Memo 93, MIT LCS.
23. A. L. Davis & R. M. Keller (1982) Data flow program graphs. *IEEE Computer* 15, 26–41.
24. C. R. Dow, S. K. Chang & M. L. Soffa (1992) A visualisation system for parallelising programs. In: *Proceedings of Supercomputing '92*, Minneapolis, USA, November, pp. 194–203.
25. A. Beguelin *et al.* (1992) Hence: graphical development tools for network-based concurrent computing. *Proceedings of Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992. IEEE Computer Society Press, Silver Spring, MD, pp. 129–136.
26. J. Werth *et al.* (1991) The integration of the formal and the practical in parallel programming environment development: Code. In: *Languages and Compilers for Parallel Computing* (U. Banerjee *et al.*, eds), Springer, Berlin, pp. 35–49.

27. A. Beguelin & G. Nutt (1994) Visual parallel programming and determinancy: a language specification. *Journal of Parallel and Distributed Computing* 22, 235–250.
28. M. Girkar & C. D. Polychronopoulos (1994) The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming* 22, 519–551.
29. L. Lamport (1978) Time, clocks and the ordering of events in a distributed system. *Communications of the ACM* 21, 558–565.
30. S. Lei & K. Zhang (1994) Performance tuning of message-passing programs through visual analysis. In: *Proceedings of International Conference on Parallel and Distributed Systems*, Taiwan, 19–21 December, pp. 153–162.
31. R. B. Netzer & B. P. Miller (1992) Optimal tracing and replay for debugging message-passing parallel programs. In: *Proceedings of Supercomputing '92*, Minneapolis, MN, November.
32. K. Zhang & G. Marwaha (1995) Visputer—a gaphical visualisation tool for parallel programming. *The Computer Journal* 38, 658–669.
33. M. Ahuja, T. Carlson & A. Gahlot (1993) Passive-space and time view: vector clocks for achieving higher performance, program correction, and distributed computing. *IEEE Transactions on Software Engineering*, 19, 845–855.
34. S. Turner & W. Cai (1993) The 'logical clocks' approach to the visualisation of parallel programs. In: *Performance Measurement and Visualisation of Parallel Systems* (G. Haring & G. Kotsis, eds), Elsevier, Amsterdam, pp. 45–66.
35. M. T. Heath & J. A. Etheridge (1991) Visualising the performance of parallel programs. *IEEE Software*, 29–39.
36. J. C. Yan, S. Sarukkai & P. Mehra (1995) Performance measurement, visualisation and modeling of parallel and distributed programs using the aims toolkit. *Software – Practice and Experience* 25, 429–461.
37. S. Stepney (1989) Pictorial representations of parallel programs. In: *Graphics Tools for Software Engineers* (A. Kilgour & R. Earnshaw, eds). Cambridge University Press, Cambridge, pp. 46–57.
38. E. P. Glinert (1986) Towards 'second generation' interactive, graphical programming environments. In: *Proceedings of IEEE Workshop on Visual Languages*, Dallas, pp. 61–70.
39. W. Cai, H. K. Tan & S. J. Turner (1996) Visual programming for parallel processing. In: *Software Visualisation* (P. Eades & K. Zhang, eds.), Software Engineering and Knowledge Engineering, vol. 7. World Scientific, Singapore, pp. 119–140.
40. D. Zernik *et al.* (1992) Using visualisation tools to understand concurrency. *IEEE Software*, 87–92.
41. T. G. Lewis *et al.* (1990) Task grapher: A tool for scheduling parallel program tasks. In: *Proceedings of 5th Distributed Memory Computing Conference*, Charleston, USA, 8–12 April, pp. 1171–1178.
42. T. Bemmerl & P. Braun (1993) Visualisation of message passing parallel programs with the topsys parallel programming environment. *Journal of Parallel and Distributed Computing* 18, 118–128.
43. M. Roberts (1990) *Visual programming for transputer systems.* Ph.D. thesis, City University, UK.
44. W. Cai, W. J. Milne & S. J. Turner (1993) Graphical views of the behaviour of parallel programs. *Journal of Parallel and Distributed Computing* 18, 223–230.
45. O. Vornberger & K. Zeppenfeld (1990) Graphical visualisation of distributed algorithms. In: *Proceedings of the 3rd Conference North American Transfer User Group*, Seattle, USA, 26–27 April, pp. 223–234.
46. W. S. Cleveland & R. McGill (1985) Graphical perception and graphical methods for analysing scientific data. *Science*, 828–833.
47. T. Minoura, S. S. Pargaonkar & K. Rehfuss (1993) Structural active object systems for simulation. In: *Proceedings of OOPSLA '93*, Washington, DC, 26 September–1 October, pp. 338–355.
48. M. T. Heath, A. D. Malony & D. T. Rover (1995) Parallel performance visualisation: from practice to theory. *IEEE Parallel & Distributed Technology*, 44–60.
49. M. T. Heath, A. D. Malony & D. T. Rover (1995) The visual display of parallel performance data. *IEEE Computer*, 21–28.

50. S. Pinker (1990) A theory of graph comprehension. In: *Artificial Intelligence and the Future of Testing* (R. Freedle, ed.). Erlbaum Associates, Hillsdale, NJ, 1990, pp. 73–126.
51. V. M. Lo *et al.* (1992) Metrics: a tool for the display and analysis of mappings in message-passing multiprocessors. In: *Proceedings of Scalable High Performance Computing Conference*, Williamsburg, Virginia, April, pp. 195–199.
52. V. A. Guarna *et al.* (1989) Faust: an integrated environment for parallel programming. *IEEE Software*, 20–27.
53. S. Sharma (1990) Real-time visualisation of concurrent processes. In: *Proceedings of CONPAR'90-VAPP IV*, Zurich, Switzerland, 10–13 September, pp. 852–862.
54. V. Natarajan, D. Chiou & B. S. Ang (1993) Performance visualization on 'monsoon'. *Journal of Parallel and Distributed Computing* 18, 169–180.
55. S. R. Sarukkai *et al.* (1993) A methodology for visualising performance of loosely synchronous programs. *Journal of Parallel and Distributed Computing* 18, 242–251.
56. D. N. Kimelman & T. A. Ngo (1991) The rp3 program visualisation environment. *IBM Journal of Research and Development* 35, 635–651.
57. C. M. Pancake (1992) Graphical support for parallel debugging. In: *Proceedings of the NATO Advanced Workshop on Software for Parallel Computation* (J. S. Kowalik & L. Grandinetti, eds). Cosenza, Italy, 22–26 June. Springer, Berlin, pp. 216–228.
58. D. A. Reed *et al.* (1995) Virtual reality and parallel systems performance analysis. *IEEE Computer*, 37–46.
59. K. Imre (1993) Experiences with monitoring and visualising the performance of parallel programs. In: *Performance Measurement and Visualisation of Parallel Systems* (G. Haring & G. Kotsis, eds). Elsevier, Amsterdam, pp. 19–44.
60. D. Kranzlmüller *et al.* (1996) Parallel program visualisation with MUCH. In: *Proceedings of International Conference of Austrian Centre for Parallel Computation*, Klagenfurt, Austria, September, pp. 148–160.
61. N. Stankovic & K. Zhang (1997) Towards visual development of message-passing programs. In: *Proceedings of the IEEE Symposium on Visual Languages*, Capri, Italy, 23–26 September. IEEE Computer Society Press, Los Almitos, pp. 144–151.