Journal of Visual Languages and Computing (2001) **12,** 203–216
doi:10:1006/jvlc.2000.0189, available online at http://www.idealibrary.com on IDEAL®

The PCG: An Empirical Study

NENAD STANKOVIC*, DIETER KRANZLMÜLLER† AND KANG ZHANG‡

**Nokia, 5 Wayside Road, Burlington, MA 01803, U.S.A., ext-nenad.stankovic@nokia.com, †GUP Linz, J. Kepler University Linz, Altenbergerstr. 69, A-4040 Linz, Austria., kranzlmueller@gup.uni-linz.ac.at and ‡Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688, U.S.A., kzhang@utdallas.edu*

Received 9 June 2000; revised 1 September 2000; accepted 18 September 2000

Process communication graph (PCG) is the visual formalism used in a graph-based visual language (VL) for parallel programming. It combines control flow and data flow graphs into a single visual formalism, and supports different levels of abstraction at which parallel programs are expressed and moves to compositional programming. Empirical studies allow designers to put their designs to test in a direct and intentional interaction with users. For research projects this may be the only way to assess if their goals have been met. The case study presented here was conducted on programmers (students) solving parallel programming problems using the PCG formalism to construct parallel programs. The results of this evaluation indicate that users benefit from visual programming, even at the beginning of the learning curve.
© 2001 Academic Press

Keywords: VPL: visual programming languages, VPL-II.A.1: concurrent languages, VPL-VI.D.1: usability studies.

1. Introduction

THE IDEA OF USING VISUAL rather than textual languages for programming is not new, and we have witnessed numerous efforts. Since the complexity of computer programs is steadily increasing, presenting them in graphic form facilitates their comprehension. By capturing the strategy and the algorithm of a program, the graphics abstracts away implementation details, which makes the program code more understandable. Scanlan [1] found that structured flowcharts had an advantage over pseudo-code in algorithm comprehension even in simple cases. Graphics as windows is an integral part of the modern desktop making it user friendlier [2]. Yet, in the world of computing visual programming is often attacked by skeptical critics inclined to dismiss the idea completely. Usability of visual languages is addressed by empirical studies, which attempt to evaluate and compare them to textual languages in terms of effectiveness and appropriateness [3]. However, case studies about solving problems through parallel computation using visual methods have rarely been conducted. The lack of empirical evidence has been regarded as the biggest open problem in research into and design of visual programming languages [4].

1045-926X/01/040203+14 $35.00/0 © 2001 Academic Press

Even though the idea to use graphics for programming is compelling, the practical experience so far has taught us that graphics suit better program design phase than program implementation phase. As a matter of fact, the presence and appeal of graphics during design is overwhelming. Graphical symbols convey easily and clearly program structure, action and event sequences, entity relationships, and similar. However, they scale badly when used for actual coding.

Through everyday practice, problems, challenges and advantages of visual programming have been identified:

- Graphics take up space and the information density can be lower than in text, which can make a visual program less readable. This problem is often referred to as the Deutsch Limit of 50 [5].
- In general, graphics can convey structural information more effectively than text, while text is usually more precise than graphics. Therefore, any effective visual languages must use both text and graphics.
- It is in fact not practical or feasible to have a pure graphical language, since programming language semantics and programming construct semantics do not map one to one. Implementation complexity far exceeds the programming language complexity. A programming language is not just a small set of keywords, but also requires a functional library.
- Graphical representation of a parallel program is very useful since it can closely resemble program execution on a multiprocessor or a network of workstations.
- Practically, a visual approach is very effective when solving visual problems such as building a dialogue box or a menu. Rather than imagining the right color, size and position of an entity, the visual technique materializes the mental image.
- Defining a GUI entity is a simple mechanical process of setting up a set of attribute-vale pairs by using, for example, JavaBuilder, Visual Basic, Visual Works.
- Generating code from graphics follows the same pattern of reasoning. It is easy to generate GUI code, since the domain is very constrained, as it implies no logic or alternative traversal paths.

So why do we suggest directed graphs for parallel programming? The reason is that graphs excel at displaying certain information that is important for better understanding of parallel programs and obtaining good performance from them. Graphs show parallelism in execution, locality and communication topology, which are among the most important aspects of parallel programming.

The success of a visual language often depends on the editor used for graphics rendering and program composition. The editor enforces consistency and performs syntax checking. The ease of use has been recognized as a more important goal of visual programming environments. Newton conducted a case study of the CODE [6] parallel programming environment, and concluded that though important, ease of use is difficult to evaluate due to its subjective nature [7]. It requires a comparative study of multiple languages and environments, such as the one reported in [8], conducted on many subjects. Based on these and similar works we have decided, when designing our visual language for parallel programming the approach should be pragmatic. Rather than insisting on a pure graphical environment, we decided to take advantage of

different language and programming forms that better meet their goals and serve the purposes.

In the next section, we present the PCG visual parallel-programming environment within the Visper system, a Java-based parallel programming prototype [9]. Section 3 presents an example of the program composition using PCG, and Section 4 describes our empirical study. Section 5 concludes the paper.

2. Parallel Programming in PCG

Parallel programs may be seen as three-dimensional structures in which programmers must manage control and data flow between distributed processes that can form groups. The idea behind PCG is that programmers create message-passing programs by drawing and annotating graphs. The goal was to build a system that would be intuitive and easy to use, but at the same time powerful enough to express graphically parallel programming features such as: point to point and collective communications, replication, non-determinacy, and synchronization. The graph formalism should not only be useful for design and programming, but also for debugging and performance tuning [10].

When programming in a PCG environment, we first create a PCG. A PCG provides the programmer with a hardware independent, three-dimensional programming space. In the graph (Figure 4), the *X*-axis runs horizontally and is used to add resource lines to the programming space. A resource is either a process or a group of processes. The *X*-axis also defines the flow of data between the resources. The *Y*-axis points downwards as does the control flow. The *Z*-axis is virtual, and defines the number of processes assigned to process groups. A process is a group with only one member.

2.1. PCG Language

The PCG visual programming language consists of graphical and textual symbols that represent the compositional constructs found in parallel programming. Figure 1 presents a subset of the symbols that were used in the empirical study. The *all-execute* and the *exclusive-execute* blocks represent sequential computations, which are entered as text. The message node, the probe and the directed arc are used for defining message-passing primitives, and the loop represents an iteration. Loop lines come in pairs, to designate the loop body. The stanza defines the data, the data type and the tag of a message-passing primitive.

The PCG visual language does not force the programmer to be familiar with every aspect of the message-passing paradigm, but rather allows an incremental approach to program construction, since the language itself hides the complexity of the paradigm.

Programming in PCG is not visual in all aspects, but rather exploits visual formalisms wherever data flow or parallel constructs are found. The PCG is hierarchical since an execute block in a graph may represent another PCG. Therefore, when developing a new program, some components may be reused, rather than designed from scratch. This feature also provides the scalability, needed when developing large parallel applications.

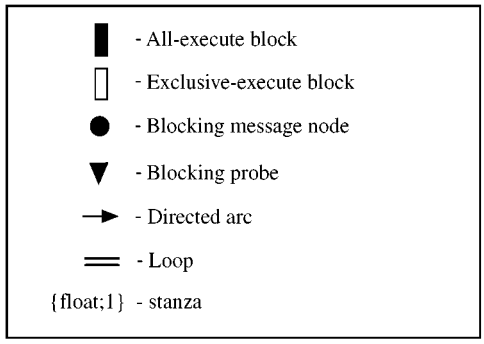

Figure 1. Visual symbols in PCG

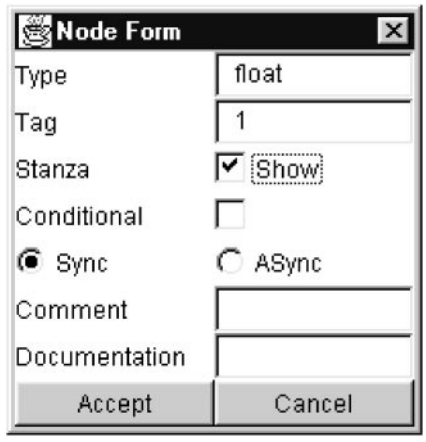

Figure 2. Message node form

2.2. PCG Editor

The PCG editor consists of a user-friendly visual editor, a database and a code composition module. It is built upon an object-oriented model and written in Java.

There are several steps when creating a program in the PCG editor. The programmer first draws a graph in the editor window that shows a parallel program or a module. Multiple windows can be opened concurrently. The graph is then annotated by filling in a set of forms (Figure 2) that describe the properties of the resources, execute blocks, arcs, etc. For example, the programmer must specify the sequential computation that an execute-block represents, and what type of data is sent by a message. Finally, the programmer selects the compile operation from the Graph pull-down menu (Figure 4) to automatically compose the program. If no syntax errors were detected, the program is saved to a text file that is ready to be compiled by any C, Java or Fortran compiler for execution, on an appropriate MPI [11] platform.

```
main () {
   procRank = getid ()

   if (procRank == 0) Proc0 ();
   if (procRank == 1) Proc1 ();
   if (procRank == 2) Proc2 ();
}
```

```
Proc0 () {
   while (!done) {
         ...
         send (1, data);
         ...
         recv (2, data);
   }
}
```

```
Proc1 () {
   while (!done) {
         ...
         recv (0, data);
         ...
         send (2, data);
   }
}
```

```
Proc2 () {
   while (!done) {
         ...
         recv (1, data);
         ...
         send (0, data);
   }
}
```

Figure 3. Program structure example

The PCG editor used in this study was an earlier version with some implementation and Java-related bugs.

3. Program Structure Visualized

Understanding the structure of a large-scale parallel program is very important due to a fact that large-scale structure can have dramatic impact on the execution performance. The aim of analyzing such a structure is to abstract away the low-level programming details of program components and concentrate on the coupling relationships and interactivity among task-sized elements of programs.

When writing a parallel program, a number of issues must be understood, such as what tasks make up the programs, the communication between tasks and the granularity of the computations that occur between communications, together with the communications size. Heterogeneous distributed environments in which MIMD programs execute further complicate the programming task because such systems are harder to model for performance and behavior.

Figure 3 shows a C-like message-passing program [12]. It consists of three functions and a *main*. The program uses three processes with rank numbers equal to 0, 1 and 2. Each process executes a function that reflects its name (e.g. process 0 executes the function *Proc0*). All three functions are structured similarly, making use of point-to-point calls. Each function has a loop with some sequential code first, then a communication call is made followed by another sequential code. Finally, a second communication call is made and the whole process repeats. Communication starts when process 0 sends data to process 1, then process 1 sends data to process 2 and finally process 2 send data back to process 0.

Figure 4 shows a PCG that graphically represents Figure 3. The three blocks at the top represent the sequential computation designated by the first row of ellipsis in each *Proc** function in Figure 3. The three blocks below them represent the second row of

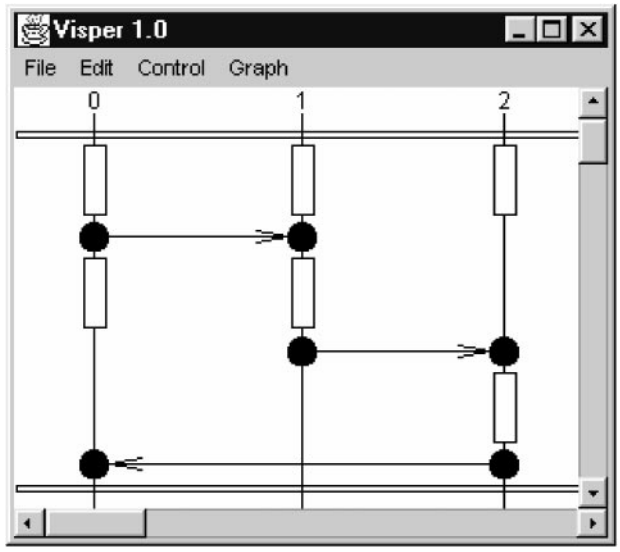

Figure 4. Program structure and PCG

ellipsis. The presented PCG provides a direct graphical representation of the sequential computation and communication dependencies between the processes, therefore helping the programmer to understand the structure.

More details of the PCG formalism have been presented elsewhere [9]. The following section presents an empirical study conducted on the PCG and reports some results.

4. An Empirical Study

The Department of Graphics and Parallel Processing (GUP) at the University of Linz, Austria, offers courses in parallel programming for graduate and undergraduate Computer Science students. These students are taught to design parallel programs by defining problem partitioning and communication structure, and to implement them in C/MPI, and HPF. During the spring semester of 1999, an empirical study was conducted where the participating students were introduced to do some practical work in PCG. Having students as the subjects is interesting as they were not experts in parallel programming, and therefore it is easier to assess how useful are the tool and the language for the beginner. A common problem in this kind of study is the small number of participants, which makes it hard to average the observation results and draw firm conclusions. We believe the findings are, nevertheless, quite indicative and could lead to further empirical studies on the PCG approach.

4.1. The Approach

In the second part of that semester, a group of 16 computer science students participated in an empirical study. At that stage, they were all familiar with the basics of

message-passing parallel programming that were taught during the first part of the semester, and in related courses they might have attended before. The study consisted of a presentation, two case studies, a questionnaire, and a same-different test [13]. Two hypotheses were under investigation:

- PCG helps program development by highlighting control flow and data flow information in parallel programs.
- Using the same (or similar) formalism throughout the whole software lifecycle facilitates comprehension.

Due to the nature of PCG environment, the empirical study can be described as comparative, by putting C/MPI against PCG. The subjects were divided in two groups: a group that researched PCG in more detail and a group that just tried to use it to do parallel programming. The workload was distributed as follows:

Group 1 (four subjects):

- Studies of visual programming in general and PCG in particular.
- One subject gave a 20-min presentation on these basics to all subjects of Groups 1 and 2,
- Three subjects gave a 20-min presentation on their implementations of a Poisson equation solver in PCG to the subjects of Group 2.

Group 2 (12 subjects):

- Implementing a parallel matrix multiplication in PCG.

Groups 1 and 2 (16 subjects):

- Experimental study on visual programming in general and experiences with PCG in particular.

The group of four subjects was selected to do some more in-depth research as well as some practical work based on the knowledge acquired during the coursework. Afterwards, they presented their work to the 12 subjects who had participated in this study. We believe that doing this will make it easier for the 12 subjects to understand and relate to the presented material, and we were able to study subjects with different exposure to PCG. Being acquainted with the problems experienced previously by other users may alleviate the negative impact of a first-time encounter. We also hoped to avoid an impression with the subjects that we want to influence the outcome. We assumed that all the 16 subjects had on average the same knowledge on parallel programming, since they attended the same basic course. The student presentation (which we will call *lecture* from here on) covered both the visual and the message-passing aspects of parallel programming. In the lecture, handouts were given to the subjects who covered these topics. In addition, the subjects could take notes and use the material in the practical part. After the lecture, the subjects were asked to implement a solution to a matrix multiplication problem.

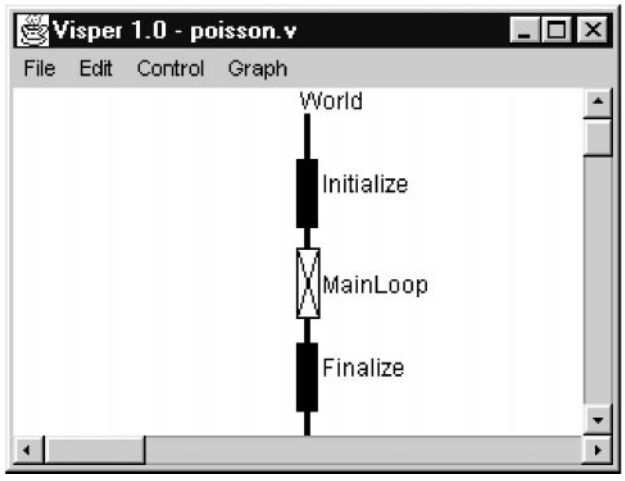

Figure 5. Poisson main program

4.2. The Lecture

The lecture consisted of two presentations. In the first presentation, one subject talked about visual languages for parallel programming and PCG, and briefly compared PCG–CODE [6] and HeNCE [14]. This was his second presentation on the topic, since he gave one before to another group of students, with the idea to explain the PCG approach to other possibly interested students in the future. That presentation was also attended by the remaining three subjects who were given a task of implementing a solution to the Poisson equation in PCG. The comments made and the observations at that presentation helped them to improve it for the case study.

The second presentation was focused on the issues that require understanding to implement the Poisson equation solver, avoiding unnecessary details. In the presentation, the three subjects analyzed their work and experience with PCG, and commented on the strong and weak points of the PCG environment, as they have been perceived or encountered. They explained the MPI routines (seven in total) and the PCG symbols used in the program. Their solution consisted of two graphs. The PCG in Figure 5 represents the main program that initializes MPI and calls (via the *MainLoop* include block) the Poisson solver in Figure 6. They also presented a space–time diagram [15] generated from trace data of their program, to compare the visual representation, explain the correlation and advantage, if any, of using a similar visual representation for program development, debugging and tuning, i.e. throughout the whole lifecycle. The generated space–time diagram was similar to the one in Figure 7 [16].

The practical relevance of Poisson equation stems from applications in weather forecasting, material analysis and other common problems. Therefore, it is often used as an example in teaching parallel computing, which is emphasized by its simplicity in its most basic form. Practically, the Poisson equation in its simplest form looks as follows:

$$f(x,y) = \frac{f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1)}{4}.$$

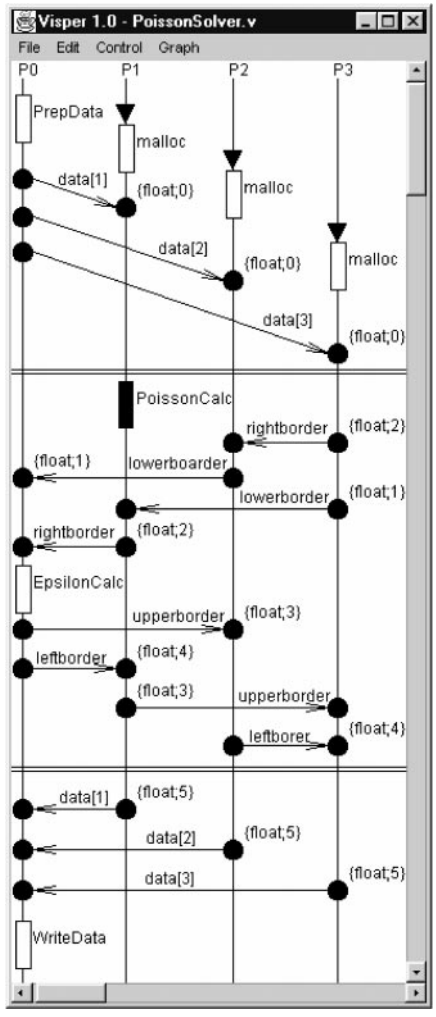

Figure 6. Poisson solver

This equation is solved iteratively, which means that during each iteration a new value of $f(x, y)$ is computed based on the values of its neighbors during the previous iteration. A PCG corresponding to this Poisson solver can be seen in Figure 6. It is divided into three parts: an initialization part, a computation part and a finalization part. During the initialization part the data values are sent from the master (*P0*) to the slave processes, which perform the computation. The computation part then performs the calculation of the Poisson equation, which requires to exchange the data values at the borders with neighboring processes. As soon as the data have been exchanged, a next iteration of Poisson's equation is computed until finally all desired iterations have been performed. Afterwards, it is necessary to return the computed data back to the master process, which is performed during the finalization part.

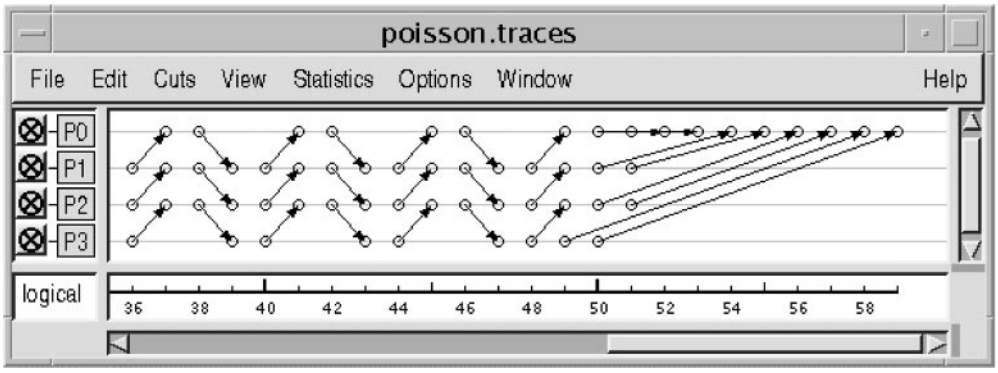

Figure 7. Space–time diagram

4.3. The Observations

The subject who gave the first presentation concluded with the following remarks: Many of PCG's ideas are without denial basic requirements to programming in different areas of computing. Hardware independence, usability, and visualization are very important and therefore included in new software products. This is especially the case in the domain of software engineering for parallel systems, because there are fewer standards available and the complexity of the program may easily explode. The advantage of PCG is its simple and systematic approach, and its feasible and usable graphical representation, which is even enhanced with the hierarchical approach. However, a final conclusion about the system may only be possible, if PCG can be compared directly to other systems and if the basic ideas are really achieved with PCG. Yet, this was not part of this project.

The three subjects who gave the second presentation were able to implement the Poisson solver in the PCG editor. Because they were dealing with practical issues, their comments were very much focused on the problems related to the PCG editor and the PCG syntax. They believe the idea behind PCG is nice and may be very useful in education. Implementing the Poisson solver was possible and not too difficult, as the algorithm is rather regular. A question was raised about irregular problems, as they believed that using PCG for real programming might be a source of overhead. Using PCG for people without knowledge about parallel programming may be difficult, because the way of thinking differs from the traditional method of programming (e.g. C) and may be problematic in the future. One subject recommended visual representation of the sequential execution (i.e. control-flow-diagram), rather than being hidden behind the execute blocks. These subjects, however, did not have much knowledge about related tools. The deficiencies of the PCG editor, as perceived, are summarized below:

- One subject mentioned that the number of forms might be confusing. However, there is no nesting of forms, as the tool consistently uses only one level of forms for each active editor window.

- At present, only the graphical presentation of the program can be made, since the code is only assembled, but not generated for the defined primitives. As a consequence, the visual representation may be different from the textual (e.g. loops).
- A comprehensive tutorial is needed and a step-by-step introduction to programming with PCG. For example, working with groups does not appear easy, and more clarification is needed here.

4.4. Solving the Matrix Multiply Problem

The group of 12 subjects were those who first heard both presentations. All of them had at least some knowledge on parallel programming and could compare PCG to other programming approaches such as C, MPI, PVM [12], or HPF. They were given the goal of implementing a matrix multiplication algorithm [17] where on each process $p(i, j)$:

```
recv(&a,p(i,j, − 1));
recv(&b,p(i − 1, j));
c = c + a ∗ b;
send(&a, p(i, j + 1));
send(&b, p(i + 1, j));
```

in PCG, C and MPI, in a reasonable time relative to the first three subjects who complete it. The subjects could refer to their notes and handouts. At the end of the test, the following results have been achieved:

- Five subjects managed to construct correct programs.
- Three subjects failed to produce anything close to a solution.
- Four subjects produced conceptually correct programs, but with programming errors.

The number of completely correct programs does not exceed by much the course average, but the number of conceptually correct does. While most PCGs were conceptually correct, the programming errors included incorrectly constructed matrices, mismatch between the PCG and the C/MPI code, incomplete initialization (e.g. MPI) and communication errors. Based on these results we conclude that the first hypothesis holds for PCG, making it a good match for parallel programming.

After the presentation and the practical work, the whole experience was discussed with the complete group of subjects. The subjects who managed to construct completely correct programs had mainly positive comments about PCG and commented about the potential they experienced in PCG. Their doubts were mostly due to the problems related to the PCG editor's reliability (e.g. it was suggested to frequently save the edits). Some expressed a concern that the functionality, as currently offered by the PCG editor, is not sufficient for real-life applications. Among those who did not come up with acceptable solutions, one commented that it was hard to switch over to the approach and practice of PCG since it differs substantially from the text editor routine. Some of the subjects mentioned that the given problem needed more thoughts than originally anticipated, and that it was hard to divide the program into components that were functionally complete, but at the same time would reflect the parallelism as dictated by the execute blocks. The graph helped conceptualizing the solution, but time was spent on correlating the graph with the code.

Table 1. Yes–No–Don't Know

Question	Yes	No	?
Previous exposure to visual programming	6	8	
Previous exposure to parallel programming	8	8	
Is visual programming useful	9	5	2
PCG helps in design	7	3	1
PCG editor is easy to use	7	6	2
PCG is understandable	9	3	
PCG for large problems	3	9	
Never use PCG	5	8	2
Understand groups in PCG	3	10	3
Code generation is needed	12		2
Prefer text over PCG	10	1	5
Use PCG for education	7	5	4

4.5. The Experiment

First, a 5-min questionnaire was given to all the 16 subjects. The questions ranged from subjects' background and familiarity with other tools and languages to visual and parallel programming. Although the subjects have been exposed to basic parallel programming, they had never used any tools for visual parallel programming. To avoid confusion, we quietly allowed them to assume that any software labeled as 'visual' was indeed a visual programming language or tool. We have summarized the responses from the subjects in Table 1. The subjects were given freedom not to answer all the questions, and could also add comments if desired.

About one-third of the subjects have come across visual programming tools, such as jBuilder, Visual Basic and Visual Java. Many subjects believed that visual programming is useful, few skeptical and few others did not believe. The experience of using PCG has convinced many subjects that visual parallel programming facilitates design. Many subjects suggested that it might be easier to approach the code by looking at the PCG first, but still preferred text. Generally, the PCG editor functionality was understood and found easy or relatively easy to use. Although most subjects would not like to use the PCG editor in a large parallel-programming project, half of them would still use it to implement parallel programs. Most of them found that working with the concept of groups was difficult, but we have not verified the knowledge of those who stated the opposite. This probably explains why none of the subjects used groups or attempted to implement a more generic solution, rather than a discrete. Many subjects hoped that the PCG editor was capable of generating automatically all the code, and one questioned its scalability.

Finally, we conducted an experiment based on the same-different judgement test, to test the second hypotheses. The experiment took 12 min to complete. The subjects were presented with three pairs of figures such as Figures 6 and 7, and three problems with text programming language (TPL) and space–time diagram (STD) pairs. In all the six cases the question was whether the given pair represents the same program or not. Obviously, Figure 6 does not match Figure 7 since *P2* never sends to *P1*, and *vice versa*,

P0 never sends messages to itself, and *P0* collects only one message from each other process at the end. With the PCG-STD pairs 48% of the answers were correct, while with the TPL-STD pairs only 26% were correct, even though the subjects' experience was in favor of TPL; therefore, our second hypothesis holds. This finding stands in contrast to some of the conclusions drawn from the questionnaire (e.g. prefer text to PCG).

5. Conclusion

This paper presents an empirical study conducted to acquire a direct feedback on PCG. The subjects were the students who have been exposed to parallel programming before participating in the study. Therefore, the experience in using a visual programming tool could be compared to the text editor practice without a bias (positive or negative) towards the new technology. While the number of subjects was not large, it was comparable to similar undertaken studies. In summary, it was generally agreed that integrating visual features into any software system was very important, particularly for parallel programming tools. The PCG editor has been found as simple and relatively easy to use. It adopts a systematic approach, and intuitive graphical representation. Although the PCG editor is not mature enough to be used for large-scale programming, using it for teaching parallel programming is expected to be useful.

This empirical study has encouraged us that a visual parallel programming formalism like the PCG can be useful if scalable and easy to use. It also provides us with a first-hand knowledge on what are the most needed features and what are the most common difficulties in visual parallel programming. Although the results are not conclusive, the findings do provide a useful guidance that will help us in improving usability of the PCG environment. While in a study like this one can only skim through relevant issues, it gives us a first-hand experience with the basic aspects regarding our approach. A more comprehensive study such as the one recommended in [18] would require subjects with a relevant background and experience. It would allow us to test more complex constructs in PCG, such as the conditional data flow, the secondary notation, collective communications, etc. Our future work is towards that direction.

Acknowledgements

We would like to express our appreciation to the students at the University of Linz for their participation, dedication, and useful feedback in our empirical study.

References

1. D. A. Scanlan (1989) Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Software* **6,** 28–36.
2. E. Kandogan & B. Shneiderman (1997) Elastic windows: evaluation of multi-window operations. *ACM Proceedings CHI'97: Human Factors in Computing Systems*, Atlanta, GA.

3. M. M. Burnett & H. J. Gottfried (1998) Graphical definitions: expanding spreadsheets languages through direct manipulation and gestrure. *ACM Transactions on Computer–Human Interaction* **5,** 1–33.
4. K. N. Whitley (1997) Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing* **8,** 109–142.
5. G. Wirtz (1994) Modularization and process replication in a visual parallel processing language. *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, Missouri, pp. 72–79.
6. P. W. Newton & J. C. Browne (1992) The CODE 2.0 graphical parallel programming language. *Proceedings of the ACM International Conference on Supercomputing*, July.
7. P. W. Newton (1993) A graphical retargetable parallel programming environment and its efficient implementation. Ph.D. dissertation, The University of Texas at Austin.
8. R. Pandey & M. Burnett (1993) Is it easier to write matrix manipulation programs visually or textually? An empirical study. *Proceedings of the IEEE Symposium on Visual Languages*, Bergen, Norway, pp. 344–351.
9. N. Stankovic & K. Zhang (1999) Visual programming for message-passing systems. *International Journal of Software Engineering and Knowledge Engineering* **9,** 397–423.
10. K. Zhang, X. Ma & T. Hintz (1999) The role of graphics in parallel program development. *Journal of Visual Languages and Computing* **10,** 215–243.
11. W Gropp, E. Lusk & A. Skjellum (1994) *Using MPI*, *Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA.
12. A. G. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek & V. S. Sunderam (1993) *PVM 3 User's Guide and Reference Manual*. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory.
13. T. R. G. Green, M. Petre & R. K. E, Bellamy (1991) Comprehensibility of visual and textual programs: a test of superlativism against the 'Match–Mismatch' conjecture. *Empirical Studies of Programmers: 4th Workshop*, Ablex, Norwood, NJ, pp. 121–146.
14. A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek & V. S. Sunderam (1992) HeNCE: graphical development tools for network-based concurrent computing. *SHPCC-92 Proceedings on Scalable High Performance Computing Conference*, Williamsburg, VA, pp. 129–136.
15. L. Lamport (1978) Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM* **21,** 558–565.
16. D. Kranzlmüller, S. Grabner & J. Volkert (1996) Event graph visualization for debugging large applications. *Proceedings of SPDT'96*, *ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, PA, May, pp. 108–117.
17. I. T. Foster (1995) *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Publishing Company, Reading, MA.
18. T. R. G. Green & M. Petre (1996) Usability analysis of visual programming environments: a 'Cognitive Dimensions' framework. *Journal of Visual Languages and Computing* **7,** 131–174.