Visual Programming for Message-Passing Systems

Nenad Stankovic Kang Zhang
Department of Computing, Macquarie University
Sydney, NSW 2109, Australia
[nstankov, kang]@ics.mq.edu.au

Abstract

The attractiveness of visual programming stems in large part from the direct programmers interact with program elements as if they were real objects, since people deal better with the concrete objects than with the abstract. This paper describes a new graph based software visualization tool for parallel message-passing programming named Visper that combines the levels of abstraction at which message-passing parallel programs are expressed and makes use of compositional programming. Central to the tool is the Process Communication Graph that correlates both the control and data flow graphs into a single graph formalism, without a need for complex textual annotation. The graph can express static and runtime communication and replication structures, as found in Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). It also forms the basis for visualizing parallel debugging and performance.

1. INTRODUCTION

Visualization is recognized as a powerful technique in understanding user requirements and development of software applications, since visual constructs and relationships are often easier to reason about than similar structures described in plain text. A number of visual programming tools have been developed to provide a mechanism for a computer to understand graphical representations and transform user interactions into components of source code. Visual programming relies on visual programming languages designed to handle visual information, enable programming with visual expressions and support visual interaction.

When developing a program, we find that there are four main stages: problem partitioning, program construction, debugging and performance tuning [17]. Visual programming tools for parallel systems facilitate primarily the first half of the software engineering design cycle, i.e. the creation, mapping and partitioning to the target environment of an application and its analysis. Since activities in parallel programs are distributed across space and time, these tools are graph based because a multidimensional directed graph closely resembles the structure and execution of parallel programs. In the broadest perspective, they usually do not provide single

graph formalism for all the activities. It can be observed that some of the more recent tools use a simplified and high level of abstraction to represent parallel programs [1, 2, 3, 4, 6]. They view parallel programming as a two step process, where the programmer first writes sequential components as text, and then utilizes graphics to organize the components into a graph, the so called large grain approach. Moreover, a system based exclusively on a control flow or data dependency graph formalism often fails to visually identify parallelism, but nevertheless provides some assistance to the programmer when compiling, mapping or partitioning, by providing automatic guidance or graphical interfaces. Since the leveraging existing code is often vital to the acceptance of a new tool, the advantage of such an approach is that parallel programs can be composed quickly out of the existing components. This approach also provides an easy solution to the objective that the visual formalism should be independent of the target machine architecture, configuration, programming language and operating system. On the other hand, due to a rather poor visual representation it provides little or no help when designing from scratch, because it appears as if the programmer already knows how to structure the program before the visual programming starts. Another problem is that in the mapping of tasks to processors, the explicit visual information on the data dependencies between processes is lost.

Depending on the stage, the purpose of using graphics and consequently the graph may change, because the level of abstraction at one stage may not be appropriate at another [11, 16]. It is, however, desirable to provide an environment that can handle program description and realization using the same graph formalism. This requirement may be difficult to meet, and as a consequence, many tools designed so far are applicable only to some of the four identified stages.

The other group of tools goes further, to actually build programs from completely or mostly visual programming languages, such that the visual representations implicitly or explicitly include the semantics of the programs. However, many of the proposed visual programming languages have been only a visual overlay to their textual counterparts on which their semantics depend [15]. While this allows certain patterns to become apparent (e.g. loops), it makes the requirement on the language independence hard to meet since not all languages share the same set of constructs or the same programming paradigm. Also, the program construction and reasoning behind it is just as, or even more difficult as it would have been in the underlying textual language because the programmer must be aware of the underlying semantics and the new visual syntax. This extra burden results in the lack of a compelling reason to adopt the visual version of an existing textual language.

This paper describes a new graph formalism that provides a true multidimensional environment for composition of parallel message-passing programs. A visual programming language for parallel processing must provide graphical symbols for parallelism, communication, and synchronization together with symbols for conventional logical and control flow [5]. The control and data flow graphs must be seamlessly combined into a unique graph formalism to provide a complete program description. Our approach originates in the space-time diagram [9] and the concurrency map [14]. It combines different levels of abstraction at which parallel programs are expressed into a single Process Communication Graph (PCG) [12]. The emphasis in PCG is on modeling the control (e.g. loops, conditionals, alternatives and synchronization) and communication structure (e.g. message-passing primitives) of the design. Therefore, a program is specified as a composition of asynchronously interacting processes and procedures. The visual programming language utilizes various

graphical shapes and constructs to present the semantics of a parallel program. It also supports explicit graphical representation of functional and physical parallelism and load balancing. These constructs identify primitive actions and their grouping or sequencing onto processes or into tasks. Another important issue in parallel programming is the scalability of the tool regarding the number of processes that can be displayed and provided information about in a single analysis. In a PCG, this is addressed by allowing the programmer to identify single processes and groups of processes when programming. The aim of PCG is to visualize program construction, debugging and performance tuning throughout the software development cycle and thus maintains a single mental image for the programmer [13]. Upon a programmer's request, our visual-programming tool, called Visper, automatically generates textual code of a program based on the graphical information. A graph of the same visual format can be used for debugging and performance tuning. This paper describes Visper in the context of the Message Passing Interface (MPI) [8], while it is important to notice that our approach is general enough to suit the development of any message-passing program.

2. PROCESS COMMUNICATION GRAPH

Parallel applications assume many processors running in parallel. Each processor can run one or more processes. For simplicity, we will assume that there is only one process running at each processor. These processes may be identified individually or grouped together into a structure that closely resembles the aggregation required by the algorithm. In a message-passing environment, a message can be communicated selectively from one process to another, or to a group of processes, or broadcast within a group as well as to all processes on a multiprocessor**.** The involved processes may have to be synchronized, or may perform asynchronously. Whatever the possible solution, a visual programming tool must be scalable and easy to use, providing the user with a hardware independent graphical representation of the problem.

2.1 Programming Space

The fundamental idea behind Visper is that programmers can create message-passing programs by drawing and annotating process communication graphs. A PCG provides the programmer with a multidimensional programming space (Figure 1). In a PCG, the X-axis runs horizontally and represents processes and groups. It also describes the direction in which data are communicated between processes. The Y-axis runs from the top to the bottom of the graph and identifies the control flow of the program.

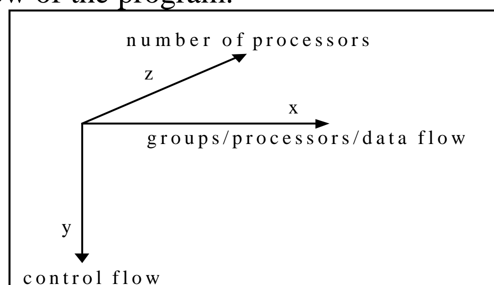

Figure 1 Programming Space

The concept of groups of processes has added another dimension to the domain of parallel programming. In Figure 1, the number of processes that make up each group is assigned to Z-

axis. To simplify graph rendering, the Z-axis is virtual. In a parallel environment that does not support grouping, the third dimension defaults to one and can be ignored.

PCG is hierarchical since each graph can reference many other graphs. Therefore, when developing a new program, components may be reused, rather than designed from scratch. This feature also provides scalability, when developing large programs.

2.2 Three-Tiered Visual Programming language

PCG provides a three-tiered visual programming language to express high level architecture and programming language independent design abstractions. The language is synthetic, since it combines generic components into complex message-passing constructs. It does not force the programmer to be familiar with all aspects of the message-passing paradigm, but rather encourages an incremental approach to program construction, by hiding the complexity of the paradigm.

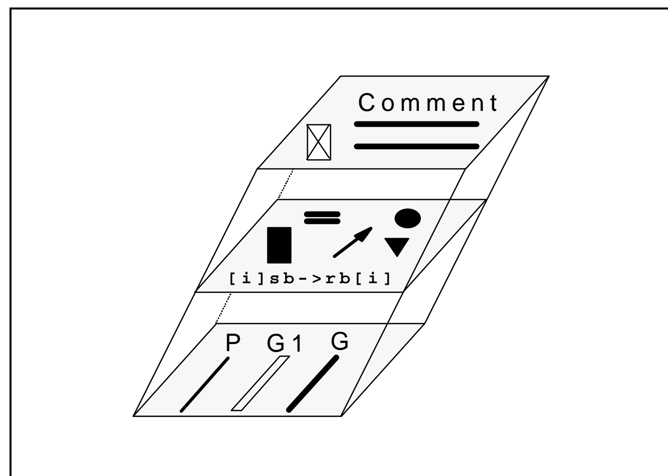

Figure 2 Three-Tiered Language

In Figure 2, the bottom tier consists of lines representing processes and groups, which are called *resource lines*, that are drawn first during visual programming. They can be thought of as the subjects in the language. On the top of each resource line there is a *name* that uniquely identifies the resource. The graphical symbols in the middle tier describe various aspects of a parallel program. They have to be combined with other graphical symbols into real message-passing primitives. In the graph grammar the annotation can be thought of as the attributes, and other graphical symbols in this tier as the predicates. We refer to the predicates as *control symbols* or just *controls*, for short. On the top, there is a collection of symbols whose purpose is to help with reading and understanding a graph, like: *comment*s, *task line*s and *include*.

2.3 Resources and Names

Resource lines are the backbone of a PCG, allowing other graphical symbols to be added later on, because they provide scope for the language primitives, found in the tier above. In Figure 3, the thin solid line represents a single process. Two types of groups are recognized. The solid thick line represents an unbounded group of processes. A group is unbounded if the number of processes in the group is determined at runtime. The hollow thick line represents a

bounded group of processes, whose size is determined at compile time. Resource lines also represent replication in the control flow graph.

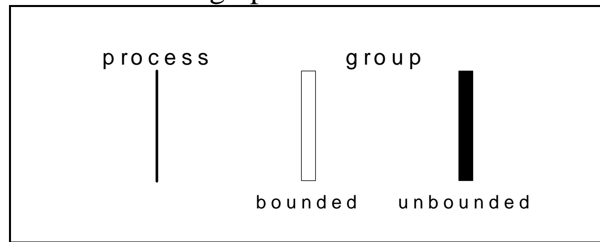

Figure 3 Resources

Names enable resource resolution within a PCG and across different PCGs. In a most general form a name is defined as:

ResourceName (GroupName)

where:

ResourceName is the name of the resource, and

GroupName is the name of the parent group (optional).

If no group name is specified, the resource belongs to the default group, the *World*.

2.4 Execute Blocks

Execute blocks are used to add code to a PCG, and to describe how the sequential components are composed into a parallel construct (Figure 4). They are the basic building blocks for assembling the code provided by the programmer into a program.

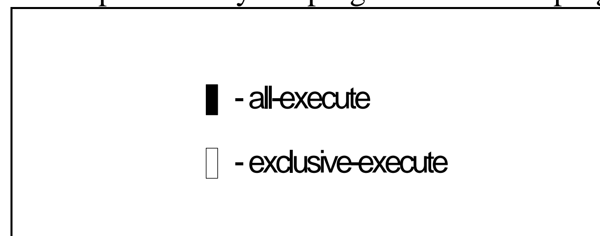

Figure 4 Execute Blocks

There are two types of execute blocks: the *all-execute* block (*AExec*) designates the part of a program that all processes in a World execute, and the *exclusive-execute* block (*XExec*) designates that only the resource it belongs to executes. Both execute blocks may be added to any process or group in a PCG.

2.5 Message Nodes And Directed Arcs

PCG provides two types of message nodes, one representing synchronous/blocking calls, and the other representing asynchronous/non-blocking calls (Figure 5). A node could be designated as conditional, meaning that it is part of a conditional construct (e.g. if, case).

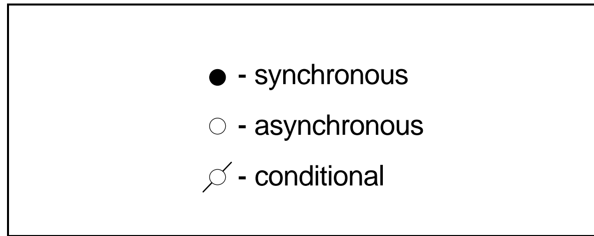

Figure 5 Message Nodes

Communication calls often come in pairs. For each send call there must be one receive that matches the destination and the tag specified in the send. In its simplest form, a pair of message nodes, connected by a directed arc (Figure 6) represents a pair of matched send and receive calls. In a PCG, a message node represents one end of a communication channel, and/or one message-passing call. Even though the graph is read from top to bottom, arcs can point up or down, to provide flexibility when rendering.

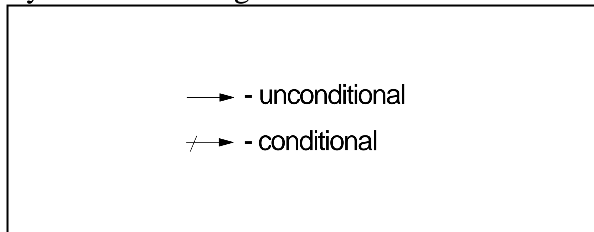

Figure 6 Directed Arcs

In a message-passing environment communication events can be either *deterministic* or *nondeterministic*. PCG identifies four sources of nondeterminism:

- a message node can receive a message from any other message node
- a message node can receive messages of any tag
- asynchronous communication calls
- a communication event will happen if some control flow condition is met (Section 4.5).

It is also possible to have the four sources in any combination, as required by the algorithm or for the sake of efficiency. Therefore, the directed arcs in Figure 6 are named *conditional* and *unconditional*. A directed arc is conditional if both its source and its sink are part of the same conditional construct.

2.6 Collective Messages

A collective message control (i.e. symbolically a *message loop*) consists of a message node and a curved directed arc (Figure 7).

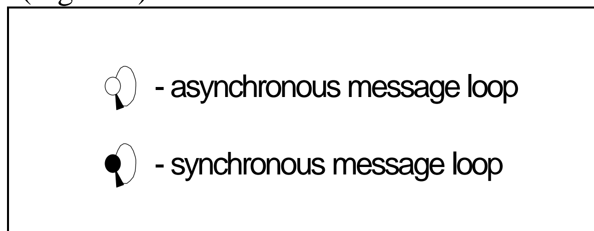

Figure 7 Collective Messages

An asynchronous message loop is used when a node sends a message to itself, because a non-blocking send is made first to avoid deadlock. A synchronous message loop is used to describe a collective communication event triggered by a many-to-many call, like a global reduction, a complete exchange, etc. Similar to message nodes and directed arcs, a message loop can also be designated as conditional or unconditional, depicted as a crossed arc line.

2.7 Synchronization

The synchronization is an important part of parallel programs. The most typical primitive used to synchronize running processes without exchanging data is the barrier (Figure 8).

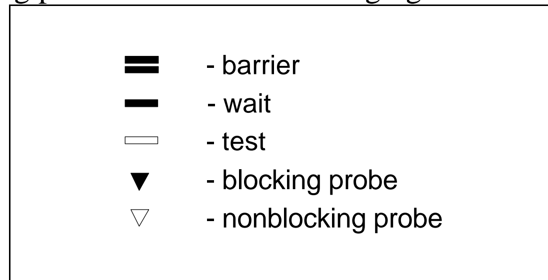

Figure 8 Synchronization

Messages between processes can be communicated by blocking or non-blocking calls. In the case of non-blocking calls there is a need to test for completion of the event. MPI provides the primitives like: wait, test and probe.

2.8 Annotations

Control symbols and *textual annotation* together provide the complete description of the message-passing function to which they belong. An annotation represents textual information added to a control, that follows the graph annotation syntax. It is displayed next to the graphical symbol. The annotation is designed to describe data flows regarding the type of the data being communicated, the data distribution onto individual processes, the variables that are used as data buffers, the scope, the protocols, etc. The information is displayed in stanzas. PCG defines five types of stanzas:

- type stanza: **{type; tag}**
- copy stanza: **[pR] sB [bI]→[pR] rB [bI]**
- wait stanza: **{request; count}**
- probe stanza: **{source; tag}**
- count stanza: **{count}**

where:

pR is the processor rank,

sB and rB are the send and the receive buffer names, respectively, and

bI is the buffer index.

In Appendix we provide the complete syntax of the stanzas in the BNF formalism.

2.9 Program Task

When solving a complex problem, it is natural to decompose it into a set of smaller problems, in the way the program is structured. We define a *program task* (or *task* for short) as a well-defined stream of sequential computation, together with *task topology*. We define task topology as a subset of *application topology*, where the application topology represents the pattern of communication between processes for the whole application [8].

The state of a program at any point in time is defined by the state of its tasks:

$$\Psi(t) = \tau_1(t) \cup \tau_2(t) \cup ... \cup \tau_n(t)$$

where:

τ_i is the state of the task i,

Ψ is the state of the program, and

t is the time.

The dependency of tasks on time is explained in Section 4.3. A task τ itself is a triple:

$$\tau\ (C, A, T)$$

where:

C is the set of controls,

A is the set of directed arcs, and

T is the set of stanzas.

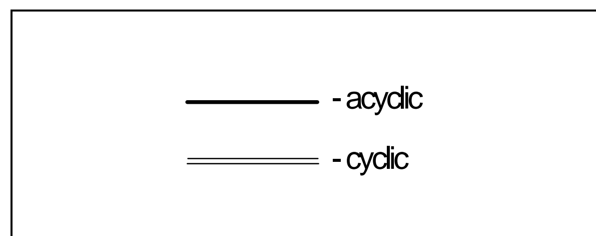

Figure 9 Task Lines

In a PCG, *task line*s serve two purposes, and consequently there are two types of task lines, drawn horizontally across the resource lines (Figure 9). One purpose is to identify (or separate) tasks and the other is to designate loops because the graph is acyclic. A pair of *cyclic* task lines explicitly designates a cyclic task. Tasks are also useful in a situation where one process belongs to multiple groups and has to perform different tasks at different time. The horizontal lines can easily identify that.

2.10 Include

The *include* symbol looks like an exclusive-execute block with two crossed lines inside the box (Figure 10), but the information it represents is graphical, rather than textual. It is used as an indicator that another PCG is included, by reference, in the current graph. Therefore, it encapsulates the computations performed by an entire graph. Include symbols are bound to resource lines just like other controls.

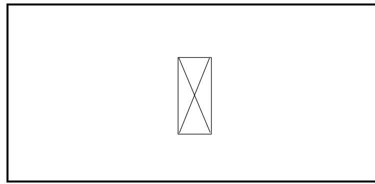

Figure 10 Include

There is no limit to the number of include symbols in a graph, as there is no limit to the level of nesting. For easier maintenance and reusability, each PCG has its own disk file.

2.11 Comments

The programmer can enter comments anywhere in a graph, for better understanding. Strictly speaking, a *comment* is a text that does not need to conform to the graph syntax.

3. PARALLELISM IN EXECUTION

We now show how various visual symbols are used or combined to represent parallelism in execution. The primary symbols are the resource lines that represent parallel processes, and the execute blocks that represent parallelism a construct level. Execute blocks do not take annotations, and only processes or groups they belong to modify their meanings. Each main PCG must begin with an all-execute block named Initialize (or Init) and end with an all-execute block named Finalize, for the programmer to know where the program starts and finishes. Spatial positioning of execute blocks describes the logic and the structure of a program. In a PCG, due to the fact that there are two types of blocks, there are different ways these blocks can be combined, and their combinations interpreted.

3.1 All-Execute Blocks

The left graph in Figure 11 represents the most trivial situation. There is only one all-execute block attached to a process (*P*). The block indicates that all processes in the application execute the given set of actions, i.e. as the name suggests, its scope is not determined by the resource it belongs to.

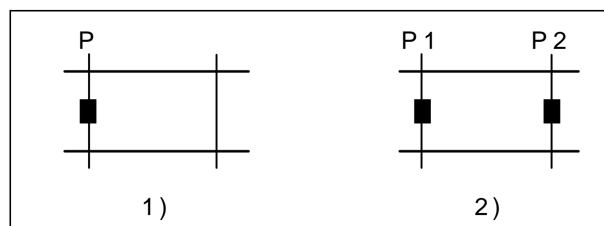

Figure 11 All-Execute Blocks

Graph 2, where two all-execute blocks are located at the same latitude in the graph, represents a syntax error. Generally, a situation where two all-execute blocks are horizontally placed next to each other at the same latitude is not allowed, because it is not clear which block (or set of actions) comes first. This is also true for a situation when an all-execute and an exclusive-execute blocks are positioned at the same graph latitude.

3.2 Exclusive-Execute Blocks

Graph 1 in Figure 12 presents a task with two exclusive-execute blocks attached to two processes, as designated by the thin vertical lines. Since both blocks are positioned at the same latitude, they execute in parallel. This is a classical *if-else if* situation that is described in C as:

```
if (pR == P1) {
        ...
} else if (pR == P2) {
        ...
}
```

where:

P1, P2 are the ranks of the processes.

Graph 2 represents a task with one exclusive-execute block attached to a bounded group G, where the C code to scope the actions looks like:

```
if (pR == GP1 || pR == GP2 || ... || pR == GPn) {
        ...
}
```

where:

GP1, GP2, GPn are the ranks of the processes that make up group G.

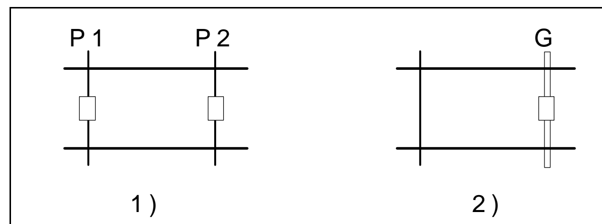

Figure 12 Exclusive-Execute Blocks

By looking at the code, this situation can be described as a generalization of the case with only one process being involved. This also complies with the definition of a bounded group as a (virtual) group of individually known processes.

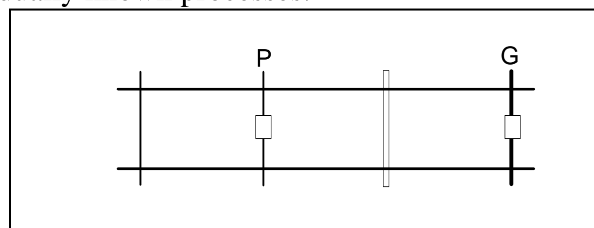

Figure 13 Unbounded Group

A third situation that is interesting to look at is a combination of an exclusive-execute block and an unbounded group of processes. Since the group is defined at runtime, it requires another exclusive-execute block at the same latitude attached to a process (Figure 13). This is an *if-else* situation that is described in C as:

```
if (pR == P) {
        ...
} else {
        ...
}
```

where:

P is the process rank.

4. VISUALLY CONVEYED INFORMATION

Visual programming languages let the programmer sketch, point at or demonstrate data relationships, structures or transformations. Their directness, immediacy and simplicity are appealing. Rather than translating them directly into sequences of commands the aim of such an analysis is to abstract away the low-level programming details of the program components and concentrate on the coupling between task-sized elements of programs.

Writing fast and efficient parallel programs requires full understanding of details such as what tasks make up the programs, the communications and their sizes, and the granularity of the computations that occur between communications. Heterogeneous environments further complicate the programming task because such systems are harder to model for performance. A graph based programming language like the PCG with its set of symbols helps to formalize the process.

4.1 Program Structure

Figure 14 shows a C-like message-passing program [7]. It consists of three functions and a *main*. The program is using three processes with rank numbers equal to *0*, *1* and *2*. Each process runs a function that reflects its name (e.g. process *0* runs the function *Proc0*).

```
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 14 Program Structure Example

All three functions are structured similarly making use of point-to-point calls. Each function has a loop with some sequential code first, than a communication call is made followed by

another sequential code. Finally, the 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 *0* receives data from process *2*.

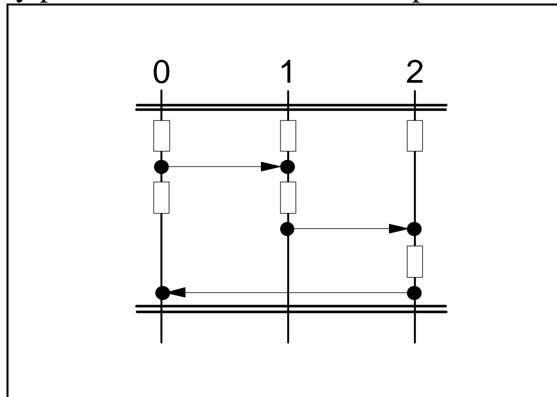

Figure 15 Program Structure

Figure 15 shows the PCG for the above example. The blocks at the top represent the sequential computation designated by the first row of ellipsis in each *Proc** function in Figure 14. The blocks at the bottom represent the second row of ellipsis. The presented PCG provides a direct graphical representation of sequential computation and communication dependencies between the processes, therefore helping the programmer to understand the structure.

4.2 Performance Issues

Together with the program structure, a PCG can also explicitly display other information that is crucial to understanding the performance of a parallel program. For example, in Figure 15, if the execution time of the sequential segments between communications is too short, the performance is dominated by the overhead of message passing.

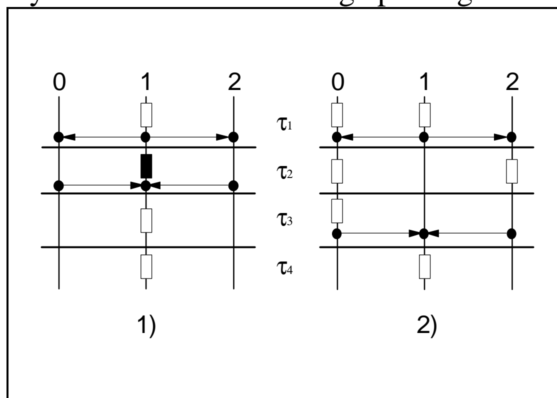

Figure 16 Performance Issues

Issues such as poor load balance or inadequate degrees of parallelism are obvious from the shape of the graph, as shown in Figure 16. Graph 1, on the left side, represents a situation where insufficient parallelism can be identified due to the fact that processes *0* and *2* are idle most of the time except for task t_2. On the other hand, graph 2 represents a situation where poor load balance can be identified, since process *0* appears to be busier than processes *1* and *2*.

4.3 Program State

As mentioned in Section 2.9 the state of a program at any point in time is defined by the state of its tasks. Figure 17 illustrates two parallel programs both having three processes and four tasks, but one using barrier synchronization at the end of each task. In a heterogeneous environment, where the available computers on a network may be made by different vendors or have different compilers, it is reasonable to expect that some nodes will run faster than others. The same is true for messages, no matter if the environment is heterogeneous or not. Therefore the performance of a program will very much depend on the balance in the assignment of tasks to computing nodes and the interconnecting network. If we neglect for a moment the impact of messaging on the performance we can concentrate on the example in Figure 17.

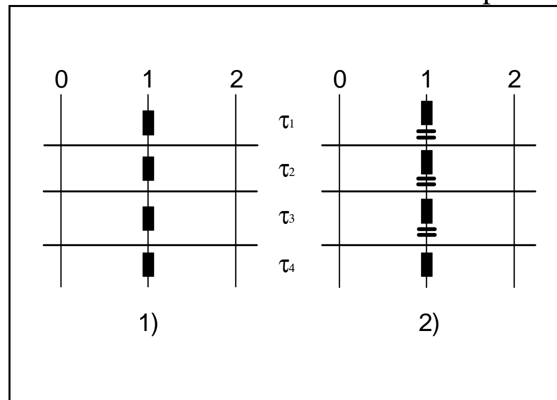

Figure 17 Program State Example

Based on the controls found in both graphs some general performance predictions can be drawn out. Since all processes run the same code, for graph 1 we can expect that tasks will overlap in time if the nodes do not run at the same speed (Figure 18).

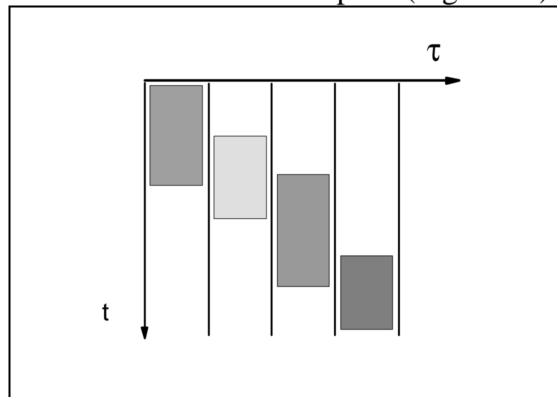

Figure 18 Time - Task Diagram

The same is not true for graph 2, because the barriers will force all the processes to be synchronized before they can proceed to the next task. This situation can also be expected in a homogeneous multiprocessor where all the processors are same in capability, resources, software and communication speed.

4.4 Data Flow Scalability

Multiple message nodes can be combined together to express a higher level of communication, like broadcast, scatter, etc. When combining multiple message nodes it is important to notice that node multiplexing is allowed only at one end of the communication channels (either at source or sink), which is a direct consequence of the definition of the one-to-many functions. For example, MPI_Sendrecv combines into one function the sending of a message to one process and the receiving of an another message from another process.

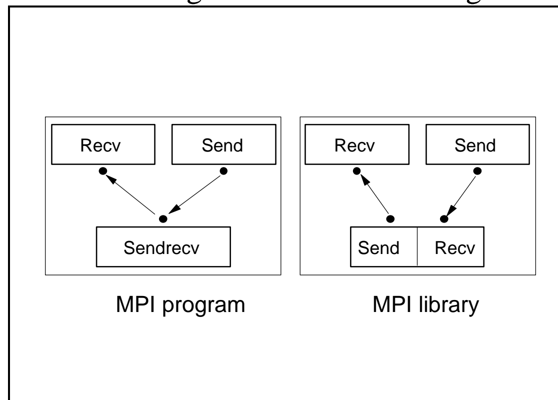

Figure 19 Construction of Convenience Functions

Figure 19 shows how a MPI_Sendrecv appears to the programmer when writing a program and how is the same function implemented in the library. There are three processes and functions involved together with two communication channels. We use rectangles to represent the involved processes together with the functions they invoke. Communication channels are represented by nodes and arcs. Since each point-to-point function is represented by one message node, we use only three nodes in PCG as presented in the left graph named *MPI program*. There is one node for *Send*, one for *Recv* and one for *Sendrecv*. On the other hand, in the *MPI library* the MPI_Sendrecv is implemented as a *Send* and a *Recv*. But the programmer does not need to know that, and he or she simply assumes that the two nodes of the *Sendrecv* have collapsed into one.

4.5 Nondeterminism

Besides the data flow, another important aspect in parallel programming is the control flow. Since parallel programs consist of many threads of execution, it is important to understand not just the data dependencies between the threads, but also the control flow and precedence relations. PCG does not have special symbols to describe control flow, except for the conditional arcs, but is nevertheless powerful in expressing various control flow constructs.

In Section 2, we have identified four basic sources of nondeterminism in message-passing programming. Beguelin [3] has pointed out that determinacy is important in parallel programming, although, sometimes, nondeterministic solutions may yield better performance. Debugging parallel programs is often difficult, and multiple runs of a nondeterministic program on the same input can produce different outputs, making the debugging process even more cumbersome. Various analysis techniques have been developed for finding sources of nondeterministic behavior. PCG, as a visual programming language, addresses this issue visually.

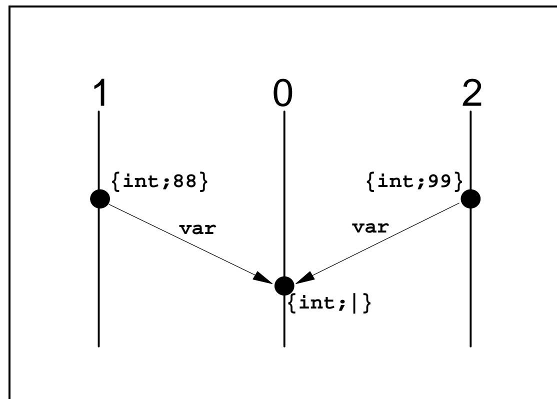

Figure 20 MPI_ANY_SOURCE and MPI_ANY_TAG

In Figure 20 we can identify two sources of nondeterminism at process *0*, due to the use of a MPI_ANY_SOURCE and MPI_ANY_TAG wildcards. The existence of the first wildcard is described by the two incoming arcs at the message node at process *0*. The copy stanza *var* at both arcs indicates that the whole construct represents two point-to-point communication channels. The second wildcard is designated by the vertical line in the type stanza at the same node. This is due to the fact that processes *1* and *2* are sending messages with different tags (i.e. 88 and 99, respectively). There are two paths of execution that can be taken in such a situation, which is easily perceivable by following the flow of the data. It is important to notice that this scenario will always generate at least one communication event, because there is no condition imposed on any of the arcs or nodes (but the receiver will handle only one message at each run). This scenario can be described by the following example:

```
if (pR == 1)
    MPI_Send (var, 1, MPI_INT, 0, 88, MPI_COMM_WORLD);
if (pR == 2)
    MPI_Send (var, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
if (pR == 0)
    MPI_Recv (var, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
        MPI_COMM_WORLD, &stat);
```

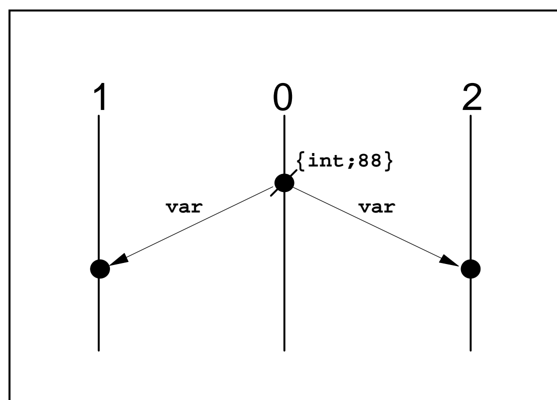

Figure 21 Conditional Node

In Figure 21, there are two processes receiving a message and one process is sending. Since the node at process *0* is conditional, this scenario will generate 0 or 1 send event at each run. It can be described by the following example:

```
if ((pR == 1) || (pR == 2))
        MPI_Recv (var, 1, MPI_INT, 0, 88, MPI_COMM_WORLD, &stat);
if ((pR == 0) && (<condition 1> == True)) {
        destination = <condition 2> == True ? 1 : 2;
        MPI_Send (var, 1, MPI_INT, destination, 88, MPI_COMM_WORLD);
}
```

The condition: *<condition 1> == True* requires the node at process *0* to be designated as conditional. If it is left out, this scenario will always generate only one send event. The condition: *<condition 2> == True* represents an alternative. If we compare this example to the previous, it is obvious that conditional nodes are not used to designate identification of processes, because that is taken care of by the resource lines.

4.6 Visual Debugging

There are two aspects to debugging in parallel programming. One is logical debugging and the other is performance debugging. PCG can help with both aspects, however in a different capacity. Since PCG is very detailed in presenting communication and synchronization events, it is easy for the programmer to detect parallelism, actual or potential deadlock situations, nondeterminism, etc. When debugging a program, the programmer is typically concentrating on a small segment of the program that is not performing as expected. In the programming model found in Visper, program tasks are the most natural segment of control flow to be considered when debugging. Inside a task, the programmer can either look at the execute blocks to understand the actions, or look at the task topology for the problems related to communication events.

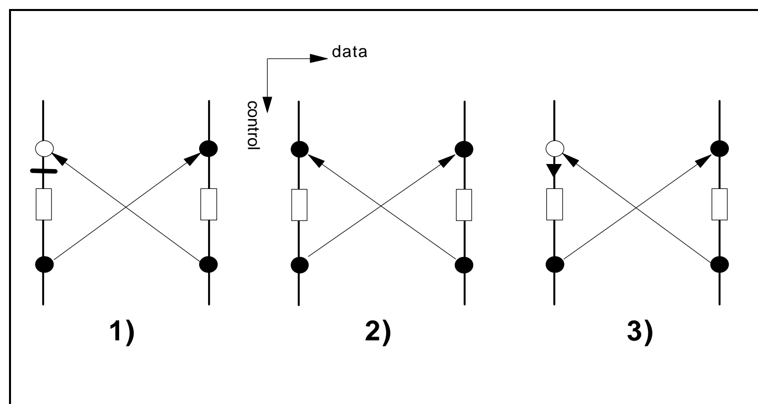

Figure 22 Deadlock

Figure 22 shows three deadlock situations, because both processes in each case are blocked waiting for each other to send a message. In the first case, a wait is blocking the left process to continue execution, and the right process is blocked by a receive call. In the second case both processes are using blocking receive functions. In the third case, the left process is blocked by a probe, and the right process is blocked by a receive call.

The PCG for program construction described above uses explicit graphical symbols to express the routines that trigger communication events at run-time. The vertical position of controls in a PCG describes the ordering of events in the control flow. When used for performance tuning, resource lines are represented as time lines such that the length of each line segment is proportional to the execution time spent on the corresponding activity. The graph then becomes a space-time diagram, and is constructed according to the trace data collected at run-time. It closely resembles the mental conceptualization of the program execution when programming, and therefore minimizes the disparity between the conceptualization of the solution and its implementation. The user can compare the PCG drawn at construction stage and the space-time diagram generated from the execution data to identify the program or execution errors and to tune the performance.

5. RELATED WORK

The approach of PCG to solving the complexity of parallel programming is similar to the approach taken by visual programming languages like HeNCE [2], Phred [3] and CODE [4]. PCG is, however, unique in its graph formalism that is designed to support program construction, debugging and performance tuning. Graphs in HeNCE and CODE can display only one type of information, either a control flow or a data flow, respectively. PCG is inherently a combination of the two. While a pure control flow PCG is possible, a pure data flow PCG is undefined because it is not known what control flow has triggered the data flow. This is best understood visually, by looking at some examples.

In Figure 23 we compare the visual representation of the replication construct in HeNCE and PCG. The HeNCE graph is read from the bottom upwards, as the arrows on the arcs indicate. To designate a replication in HeNCE, a pair of triangular icons is used. They are called fan icons. Circles are used to designate nodes or tasks that represent computation. The graph construct that comes between the icons is dynamically replicated at runtime. The right graph represents a similar situation in PCG. The replication here is absolute, since we use an all-execute block.

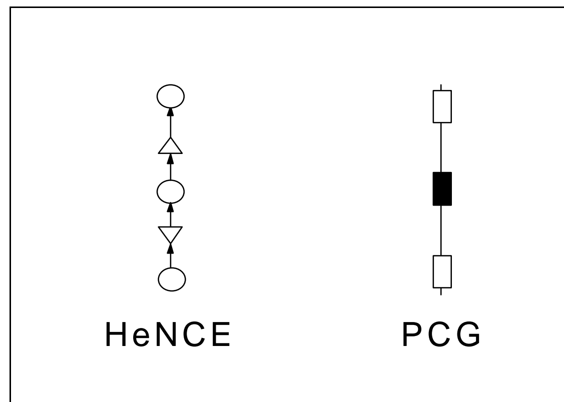

Figure 23 Replication in HeNCE and PCG

In HeNCE, arcs take no annotation, and computation nodes are named by the exactly one sequential procedure they call. Each procedure call is accompanied by declarations that specify the input and output variables, the scope of which is global. A node can execute whenever all of its predecessors have executed. The PCG in Figure 23 does not read like that, as PCG does not

pose such restrictions on the programming process. Our programming model is asynchronous , so are the execute blocks. Synchronization between execute blocks is a result of the data flow constructs, as presented in Figure 24.

While the PCG graph in Figure 23 is correct, it is not natural because there is no data flow component in it. In HeNCE, the programmer is only interested in the control flow aspects and the task precedence. It is assumed that data, if any, are somehow, somewhere communicated. In a PCG, on the contrary, if there is no data flow graph drawn out, there are no data exchanged between processes. Therefore, the situation in Figure 24 is more natural to PCG.

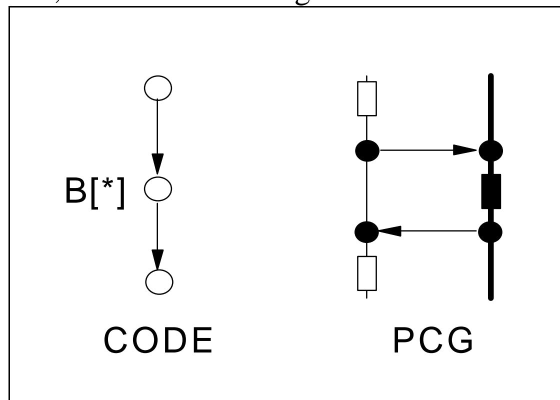

Figure 24 Replication in CODE and PCG

The CODE graph in Figure 24 represents a data flow graph with 3 nodes (circles) with the arcs showing the flow of the data between the nodes. Nodes in a CODE graph represent sequential computations, where data can come and leave only at the beginning and the end of a node. The node in the middle is dynamically replicated, which is designated by a star in the annotation. The PCG in Figure 24 is a proper version of the same graph in Figure 23, because it shows parallelism as well as data flow between the resources. The lack of graphical symbols in CODE has been compensated by a complex set of annotations. The programmer has to define the input and output ports, variables, firing and routing rules and computation in terms of stanzas. For example, a node can execute once its firing rules are satisfied. In PCG terms, that means that all the data are available on all the incoming arcs, or the barrier has been released on all the affected processes.

PCG does not support many direct control flow constructs except those that are fundamental for describing parallel programs. So far we have seen the replication, the sequential computation, the parallelism, the synchronization and the cycle. The execute blocks and the cycle, however, are defined as opaque constructs to allow greater flexibility when programming. PCG is unique in its use of conditionals that are in the data flow rather than control flow. The reason for this stems from the fact that PCG originates in the space-time diagram [9] that displays the occurrence of data flow events. A crossed arc (Figure 6) or a crossed message node (Figure 5 and Figure 21) designates a condition in PCG. It means that the event may or may not happen. If it happens it gets displayed in the space-time diagram. Another consequence of the data flow graph is that the objects used in the description of the data flow have also their meanings in the control flow. For example, an alternative is a consequence of a condition placed upon the data flow or the data flow construct itself, which is the case in Figure 25.

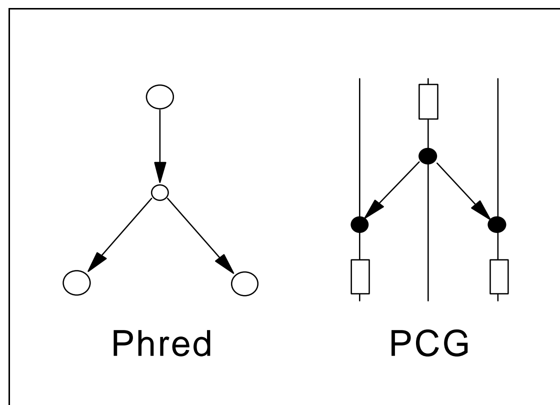

Figure 25 Alternative in Phred and PCG

In Figure 25, we can see a comparison between a Phred graph and a corresponding PCG. In the Phred, the control flow takes either the right or the left branch. The small hollow circle in the middle (the so-called OR node) designates that. In the PCG, the program makes a similar decision, but based on the data flow. We can read from the graph that all the three resource lines represent individual processes. Therefore, the two messages are of a point-to-point type. Since there are only three message nodes, only one message is sent, either to the left or to the right process. This can be described by the following MPI code:

```
if (pR == MIDDLE) {
        destination = <condition> == True ? LEFT : RIGHT;
        MPI_Send (…,destination,…);
}
if ((pR == LEFT) || (pR == RIGHT))
        MPI_Recv (…,MIDDLE,…);
```

The process that receives the data keeps on processing, while the process that does not get any data is blocked. This is due to the fact that the receiving nodes are blocking. We can say that the nodes represent a logical AND operation in the control flow graph. They serve, together with the arcs, as the link between the data flow and the control flow graphs. This example also shows that an alternative in PCG has an explicit spatial connotation.

6. IMPLEMENTATION AND EXAMPLE

Visper has been implemented for design and generation of message-passing programs. It provides a visual programming environment that is computer architecture independent, supports process-to-processor mapping and various message-passing and control flow constructs. It encourages reuse of software modules by moving to compositional programming and allowing hierarchical graph structure. The main goal was to build a system that would be intuitive and easy to use, but at the same time powerful enough to express graphically the issues such as: point-to-point and collective communications, non-determinacy, synchronization and replication. The same graph formalism must be useful not only for programming, but also for debugging and tuning.

6.1 Visper

Visper consists of a user-friendly visual editor, a database and a code generation module (Figure 26). It is built upon an object-oriented model and written in Java. There are several steps in creating a program by Visper. By using a mouse driven visual editor, the programmer first draws a PCG. The graph is then annotated by filling in a set of forms that describe properties of the resources, execute blocks, arcs, etc. For example, the programmer must specify the sequential computation of each block, or the type of the data sent in a message. Finally, the programmer selects the compile operation from the Graph pull-down menu (Figure 27) to automatically assemble the program. If no syntax errors were detected, the program is composed and saved to a text file.

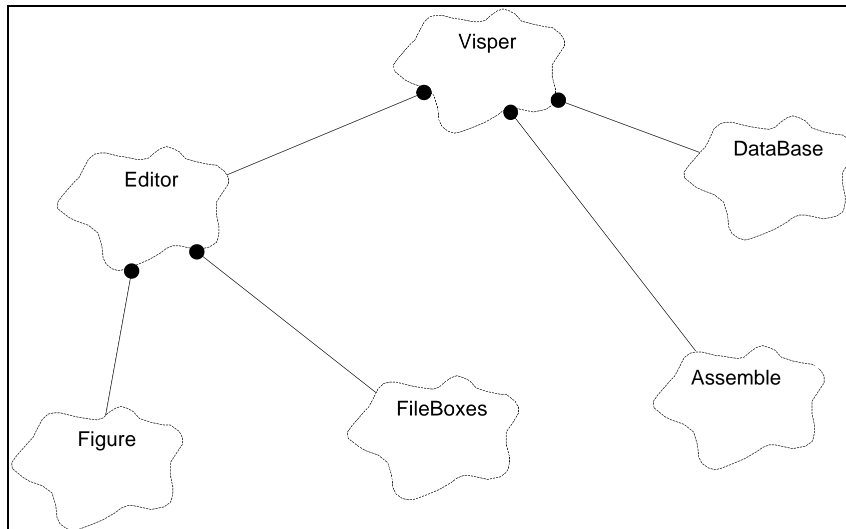

Figure 26 Main System Classes

6.2 Calculate π by the Monte Carlo Method

Figure 27 shows a main PCG drawn by a programmer to solve the problem of calculating the value of π by the Monte Carlo method of integration [8]. The method uses a circle with the radius equal to one, and a square around it with the surface equal to 4. Therefore, the ratio r of the area of the circle to the area of the square is $\pi/4$. The ratio is computed by generating random points (x, y) in the square and counting how many of them are in the circle. The user must provide the required precision in the calculation.

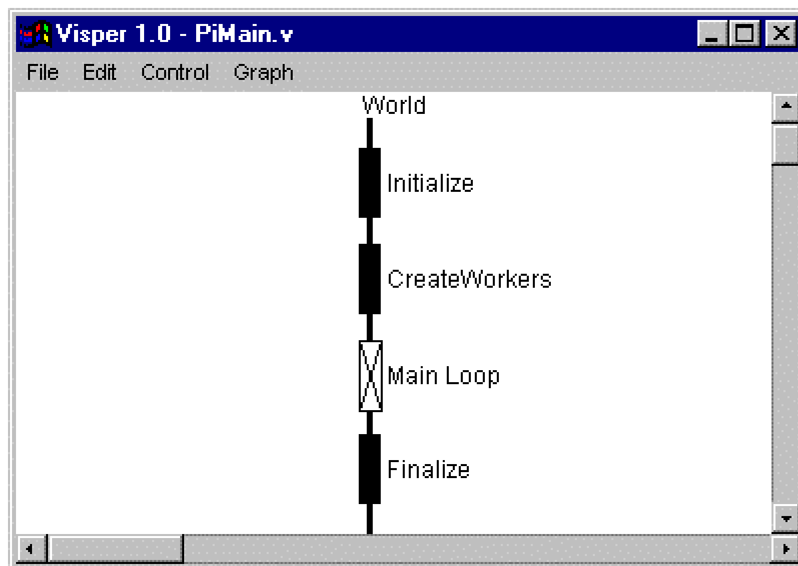

Figure 27 Main Graph

Figure 27 shows the main graph for the program. The program is divided into four stages that make use of an unbounded group named *World*. Firstly, the MPI environment is initialized, and then a new group of processes called *Workers* gets created. To emphasize the fact that all the resources must perform both stages, we use all-execute blocks. Then, the program enters the *Main Loop* that performs the calculation and finally, before it exits, the program must free up all the resources by calling a MPI_Finalize.

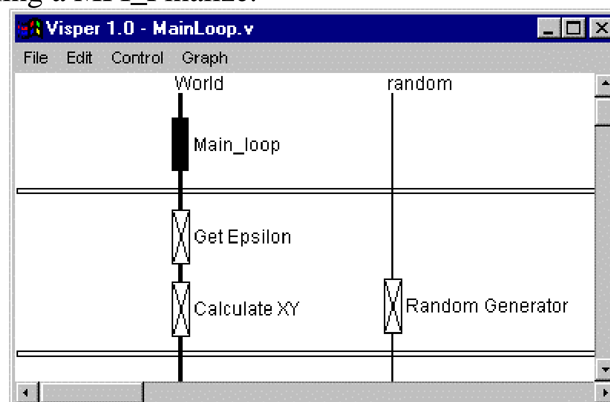

Figure 28 Main Loop Subgraph

The *Main Loop* subgraph in Figure 28 allows the user to test for convergence different values of precision *epsilon* without restarting the program. A process named *random* is used to generate random numbers, while other processes in the World perform the calculation. It can also be seen that the calculation and the random number generation are performed in parallel.

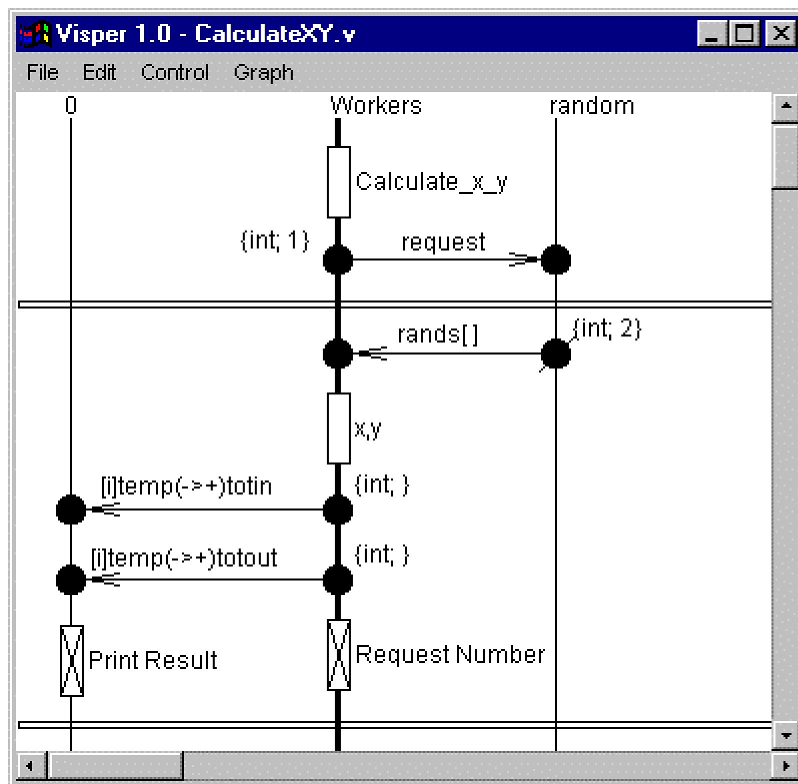

Figure 29 Calculate XY Subgraph

In Figure 29, each process in the *Workers* sends a *request* to the *random* for a new array of random numbers *rands[]*. Since a point-to-point call is used to make the request, the receiver uses a MPI_ANY_SOURCE wildcard. (This is implicitly defined by the group-process-point-to-point-call construct.) The server sends back an array to the calling process from the *Workers*. A pair of empty brackets designates the fact that the random sends back an array. Based on the *rands*, the *x* and *y* coordinates are calculated and the results are sent to process *0* by two MPI_Reduce calls. The syntax for the call is *[i]temp(*fi *+)totin*. If the required precision was met, process *0* prints the result out, or else the whole procedure repeats.

7. CONCLUSION

Visper is a new tool for visual parallel programming that extends the functionality of the concurrency map [14] and space-time diagram [9], to handle the composition of parallel programs for MIMD architectures. It is based on MPI [8] and PVM [7] for writing message-passing applications in a heterogeneous computing environment. Central to Visper is a graphical editor by which the programmer directly inputs a PCG. The graph is scalable and hierarchical with a set of graphical symbols and constructs for describing communication and synchronization routines together with replication and sequential computation. In a PCG, processes can be manipulated individually or as groups. Arcs in the graph denote the data flow that can be deterministic or nondeterministic. The control symbols represent the disjunctive (OR) and conjunctive (AND) operations in the control flow. Due to the explicit visual representation of parallelism and nondeterminism, the potential dependency and concurrency between processes that can lead to deadlocks is easily perceivable. Since the idea behind the

graph originates in the graphs that are used for performance analysis, a PCG can also help in understanding the feedback obtained from the program execution. In the future we plan to integrate the tool with other tools developed at Macquarie University that support online and offline performance monitoring [10].

ACKNOWLEDGEMENT

The authors would like to thank Dieter Kranzlmueller from GUP Linz, Johannes Kepler University Linz for his discussions and suggestions, and also the anonymous referees for their constructive comments.

REFERENCES

1. Babaoglu, Ö. Paralex: An Environment for Parallel Programming in Distribution Systems. *Proceedings of ACM International Conference on Supercomputing*. July 1992, pp.178-187.
2. Beguelin, A. L., et al. HeNCE: Graphical Development Tools for Network-Based Concurrent Computing. *SHPCC-92 Proceedings of Scalable High Performance Computing Conference*. Williamsburg, Virginia, April 1992, pp.129-136.
3. Beguelin, A. L. Deterministic Parallel Programming in Phred. PhD Thesis, University of Colorado, at Boulder, 1990.
4. Newton, P., and Browne, J. C. The CODE 2.0 Graphical Parallel Programming Language, *Proceedings of ACM International Conference on Supercomputing*. 1992, http://www.cs.utexas.edu/users/code
5. Chang, S. K. *Introduction: Visual Languages and Iconic Languages*. Visual Languages, Plenum Press, New York and London, 1989.
6. Cotronis, J. Y. Efficient Composition and Automatic Initialization of Arbitrarily Structured PVM Programs. In Jelly, I., Gorton, I., and Croll, P., (Eds.). *Software Engineering for Parallel and Distributed Systems, March 1996*. Chapman & Hall, London, 1996, pp.74-85.
7. Geist, G. A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. S. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.
8. Gropp, W., Lusk, E., and Skjellum, A. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.
9. Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communication of the ACM*. Vol. 21, No. 7, July 1978, pp.558-565.
10. Lei, S., and Zhang, K. Performance Visualization of Message-Passing Programs Using Relational Approach. *Proceedings of 7th International Conference on Parallel and Distributed Computing Systems*. Las Vegas, Nevada, 6-8 October 1994, pp.740-745.
11. Scheidler, C., and Schöfers, L. Trapper: A Graphical Programming Environment for Industrial High-Performance Applications. *PARLE'93, Parallel Architectures and Languages Europe*. München, June 1993, pp.403-413.
12. Stankovic, N., and Zhang, K. Towards Visual Development of Message-Passing Programs. *Proceedings of 13th IEEE International Symposium On Visual Languages*. Isle of Capri, Italy, 23-26 September 1997, pp.144-151.
13. Stankovic, N., and Zhang, K. Graphical Composition and Visualization of Message-Passing Programs. *Proceedings of 1997 Software Visualization Workshop*. Flinders University, Adelaide, South Australia, December 1997, pp.35-40.

14. Stone, J. M. Debugging Concurrent Processes: a Case Study. *Proceedings of ACM SIGPLAN 1988 on Programming Language Design and Implementation*. June 1988, pp.145-153.
15. Temple, D. M., and Guest, S. T. Diagrammatic Techniques for the Visualization of Object Oriented Programming. In Tauber, M. J., Mahling, D. E., and Arefi, F., (Eds.). *Cognitive Aspects of Visual Languages and Visual Interfaces*. Elsevier Science B. V., 1994, pp.259-294.
16. Turner, S. J., Cai, W., and Tan, H. K. Parallel Programming with VPE: a Case Study of an Integrated Visual Programming Environment. *High Performance Computing on the Information Superhighway*, *HPC'97 Asia*. Seoul, Korea, April 1997, IEEE Computer Society Press, pp.319-324.
17. Zhang, K., Ma, X., and Hintz, T. The Role of Graphics in Parallel Program Development. *Journal of Visual Languages and Computing*. Vol. 10, No. 3, Academic Press, June 1999, pp.215-243.

APPENDIX

The textual annotation syntax is simple and declarative. It is formulated in such a way, that all the relevant information in the signature of message-passing functions is displayed next to the graphical symbol, in a consistent manner. It is designed to describe data flows regarding the type of the data being communicated, the data distribution onto individual processes, the variable names that are used as data buffers, the scope, the protocols, etc. The information is displayed in stanzas. The most important is the copy stanza that is similar to the arc topology specification found in CODE.

<Annotation> ::= <Stanza>
<Stanza> ::= <Copy Stanza> | <Type Stanza> | <Probe Stanza> |
<Wait Stanza> | <Count Stanza>
<Copy Stanza> ::= <Variable> |
<Variable> [] |
[] <Variable> |
[<pR>] <Variable> |
<Variable> [<bI>] |
[<pR>] <Sb> [<bI>] → [<pR>] <Rb> [<bI>] |
[<pR>] <Sb> (→<func>) <Rb> |
(<Sb> [<bI>]) → [<pR>] <Rb> |
[<pR>] <Sb> → (<Rb> [<bI>]) |
[<pR>] <Sb> [<bI>] → [<pR>] <Rb> [<bI>] |
[<pR>] <Sb> (→<func>) [0..<pR>] <Rb> |
<Copy Stanza> , <Copy Stanza>
<Count Stanza> ::= {<numeral> | <name> | all | <all op>}
<Type Stanza> ::= {<Type>} |
{<Type>; <tag>} |
{<Type>; <Type>}
<Probe Stanza> ::= {<source>; <tag>}
<Wait Stanza> ::= {<request>;} |
{<request>; <wait count>}
<Sb> ::= <Variable>
<Rb> ::= <Variable>
<Type> ::= <name> |

<name>, <name>,…, <name>

<Variable> ::= <name> |
<name>, <name>, …, <name>

<tag> ::= <name> | <numeral> | any | <any op>

<count> ::= <name> | <numeral>

<request> ::= <name>

<source> ::= <numeral> | <name> | any | <any op>

<wait count> ::= <any op> | any | <some op> | some | <all op> | all

<bI> ::= <name> | <numeral>

<pR> ::= <name> | <numeral>

<func> ::= <name> | * | + | − | / | & | && | ^ |
<any op> | <any op> <any op> |
.<letter sequence>.

<name> ::= <letter> | <name> <letter> | <name> <digit>

<numeral> ::= <digit> | <numeral> <digit>

<letter sequence> ::= <letter> | <letter sequence> <letter>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= a | … | z | A | … | Z | _

<all op> ::= *

<any op> ::= |

<some op> ::= :