

A Visual Development Environment for Parallel Applications

Philip T. Cox¹

Hugh Glaser²

Stuart Maclean²

*Faculty of Computer Science, Dalhousie University, Halifax, Canada¹
Dept of Electronics and Computer Science, University of Southampton, UK²*

Abstract

We report on the development of a visual programming environment for building applications for execution on a range of parallel computing platforms. This work exploits the dataflow and list-processing parallelism naturally exposed in the Prograph language, by providing annotations to indicate that operations can be remotely executed, supported by a task-pooling model for parallel execution that preserves the sequential semantics of the language. The goal is a practical system that builds on the comprehensive tools in the Prograph CPX environment in a consistent manner. This will be achieved via high-level editors for hardware configurations, annotations and program analysis, and an underlying kernel that implements the pooling model.

1 Introduction

Effective exploitation of parallel and distributed computers has for some time been a major goal of the computing industry and researchers. Recently, microcomputer manufacturers have begun to release symmetric multiprocessor (SMP) or shared-memory architectures aimed at computing-intensive applications such as rendering 3D models. There are, however, few tools for exploiting the parallel processing potential of such machines, so application developers must arrange the distribution of computing tasks among processors using very low level facilities.

The work reported here forms part of the GraphIcsla project at the University of Southampton, in which the use of a graphical programming environment for parallel and distributed systems is being investigated. Its primary objective is to address the shortcoming outlined above, by providing a programming language that allows the natural expression of parallelism, embedded in an application development environment that provides the necessary support for developing, debugging and deploying applications on parallel machines. The programming environment which forms the basis for this work is Prograph CPX [17]. Since Prograph employs a data flow evaluation model, it would appear to have potential for programming target machines offering fine-grain parallelism, such as the new SMP microcomputers. It would also be useful, however, to exploit the potential of the idle cycles of networked workstations by programming such networks as parallel machines with particularly coarse-grain architectures,

This work was supported by EPSRC (UK) grant GR/K41526, NSERC (Canada) grant OGP0000124, and a grant from AUCC (Canada).

such as the SP2 from IBM, which can be viewed as a network of workstations. Here we show how minor additions can make the Prograph language suitable for exploiting parallelism on a range of architectures, propose extensions to the Prograph CPX environment to support development of parallel applications, and discuss a development methodology.


Section 2 reviews aspects of the Prograph language, in particular those that relate to expressing parallel computation. In section 3 we introduce the “bless” annotation that can be applied to operations in a program to achieve parallel execution. The implementation model underlying Parallel Prograph is presented in section 4, followed in section 5 by an outline of the components of the system under construction. Sections 6 and 7 review related work and summarise the project.


2 Expressing parallelism in Prograph

We assume here that the reader has a working knowledge of Prograph. Brief descriptions of the language are given in [4, 5], while a complete description of the language, its sequential semantics, and the Prograph CPX development environment can be found in [17].

In Prograph, computations are expressed by data flow diagrams which expose the independence of operations. The language also provides lists as a primitive datatype, together with constructs that indicate potentially parallel list processing. As a result, expressing algorithms in a parallel fashion becomes the natural default, rather than a special effort requiring an extra layer of syntax. So in this sense, the language is well-suited to the aims of the project. To illustrate the various sources of parallelism in the language, we consider the implementation of the Quicksort algorithm shown in Figure 1.

The most obvious source of parallelism, called *data flow parallelism*, lies in the non-sequential arrangement of operations in a data flow diagram. To execute, an operation depends only on the availability of the input data it needs.

For example, once the operation  in the first case of the quicksort method has been executed, data is available to both quicksort operations which are then free to execute simultaneously.

The operation  in the first case of the method is a partition multiplex, an example of a general class called *list multiplexes*. The three-dimensional nature of this icon indicates that the enclosed \geq operation will be repeatedly executed, each time consuming one of the items from the list

that arrives on the list annotated terminal $\text{O} \rightarrow$. This is similar to “map” operations available in functional languages. Under certain conditions, as in this example, the operation could be applied to each element of the incoming list simultaneously. We refer to this as *multiplex parallelism*. Note that Prograph also has loop multiplexes, which, although they may have list-annotated terminals, also have loop-annotated terminals indicating outputs from one iteration that are fed as inputs to the next. Clearly such multiplexes cannot be parallelised.

Each method in Prograph consists of a sequence of cases, analogous to a set of clauses defining a predicate in Prolog. According to the sequential semantics of Prograph, the cases are tried in order until one of them executes to completion. The switching from one case to the next is accomplished by controls: for example, the first case of the `quicksort` method has two match operations O each with a *next-case-on-success* control \checkmark . Whenever one of these match operations succeeds, the associated control causes execution of the case to stop, and execution of the second case to begin. Clearly, all cases in a method could be concurrently executed, obtaining *OR-parallelism*. Note, however, that assumptions about input data that hold within a case of a method being sequentially executed are not necessarily valid with this kind of speculative execution. Consequently programs would have to be written specifically to take advantage of it.

Having identified these sources of parallelism in the language, we need to answer two issues. First, what should be the nature of the parallel programming model offered by our target environment? Second, how should the model be implemented to achieve parallel execution? We address the first of these questions in the next section, and the second in sections 4 and 5.

3 Annotating for parallelism

To achieve a practical environment for developing parallel applications for the various kinds of architectures mentioned in our introductory remarks, we are interested in building on the facilities already offered in Prograph CPX. In line with this goal, we wish to preserve the sequential semantics of the language so that applications can be developed on a sequential processor then easily

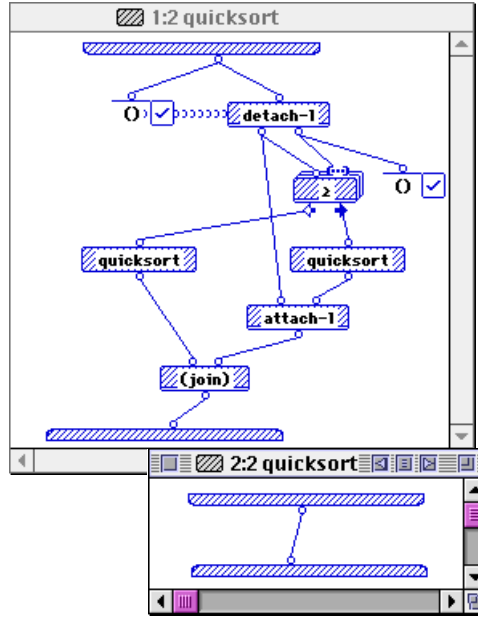


Figure 1: Quicksort in Prograph

guarantee a reduction in execution time since overheads due to synchronisation, data copying and task or process management may result in parallel execution with worse performance than that of equivalent sequential execution. For example, dispatching an operation for execution on another processor is unlikely to improve performance if the operation is a trivial one, such as applying an arithmetic primitive. Clearly, operations selected for dispatch should be carefully selected, taking into account various factors such as the size of the computation involved, the amount of data that must be copied and sent to the target processor, or whether the local processor will be gainfully employed while the remote processor is at work. Answers to these and other questions might be

deployed on a parallel platform. Therefore we have not attempted to implement OR-parallelism, since it is more suited to very large numbers of closely coupled processors and would require changes to established Prograph programming style. We focus therefore on exploiting data flow and multiplex parallelism.

Given the range of architectures we are considering, in the remainder of this paper we use the word “processor” to refer to processing entities. We also use the words “local” and “remote” to distinguish between the process or execution that is our current focus and one that occurs elsewhere. Hence the terms “local processor” and “remote processor” can mean threads within the same SMP machine or processors communicating at a distance over a network.

The availability of multiple processing units does not immediately

guarantee a reduction in execution time since overheads due to synchronisation, data copying and task or process management may result in parallel execution with worse performance than that of equivalent sequential execution. For example, dispatching an operation for execution on another processor is unlikely to improve performance if the operation is a trivial one, such as applying an arithmetic primitive. Clearly, operations selected for dispatch should be carefully selected, taking into account various factors such as the size of the computation involved, the amount of data that must be copied and sent to the target processor, or whether the local processor will be gainfully employed while the remote processor is at work. Answers to these and other questions might be obtained by analysis of the program in order to automate dispatching decisions: however, such analysis is likely to be expensive. We will return to this issue later. The programmer, on the other hand, is likely in most situations to have very good intuitions about the value of remotely executing a particular operation. Initially, therefore, we will leave it to the programmer to annotate operations for parallel execution, as described below. In the longer term we hope to be able to automate the annotation process to some extent.

3.1 Blessing

Annotation-based parallelism was pioneered by Hewitt and Baker [1] and Halstead [10]. We define an annotation called a *bles* which can be applied to Prograph operations to indi-

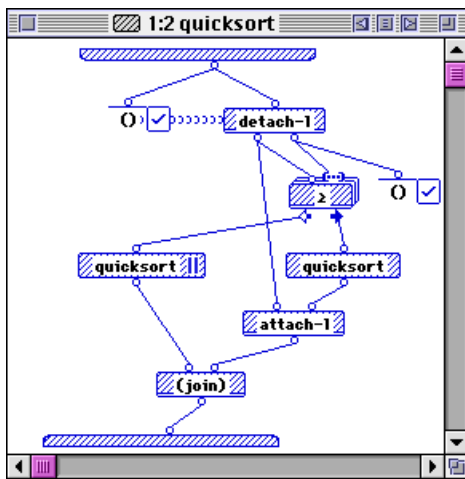


Figure 2: First case of quicksort method with one blessed quicksort operation

cate that they may be executed remotely. From a human-computer interface point of view, blessing is consistent with other features of the Prograph language, where operations, terminals and roots are adorned in various ways by annotation. Blessing differs from these other adornments, however, in that it has no semantic significance, as we shall see.

Figure 2 shows the bless `||` applied to the left hand `quicksort` operation in the first case of the `quicksort` method indicating that the operation is a candidate for parallel execution by a remote processor.

As noted above, list multiplexes promote concurrency owing to the absence of data dependencies between list elements, so under certain conditions the applications of the multiplexed operation to individual list elements may proceed in parallel. For example, blessing the operation `gather` enclosed in the multiplex as shown in Figure 3(a), indicates that each of the executions of that operation can be performed on a (different) remote processor. We call this *internally blessing* the multiplex. This does not imply that the multiplex as a whole is dispatched to a remote processor. The execution of the multiplex, that is, the management of the constituent executions and the assembly of returned results, is performed locally. To indicate that the multiplex itself is a candidate for remote execution, the bless must be applied to the multiplex rather than the enclosed operation as shown in Figure 3(b), called an *external blessing*. Blessing the multiplex both internally and externally as shown in Figure 3(c) indicates that the multiplex may be executed on a processor different from that of the enclosing case, and that the individual executions of the enclosed operation may be further dispatched.

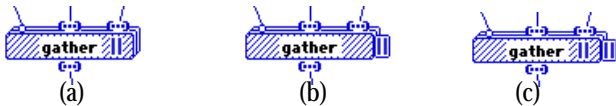


Figure 3: Combinations of blessings on a multiplex

3.2 Factors affecting annotation decisions

In the introduction to this section, we mentioned the need for caution in selecting an operation as a candidate for remote execution. Some of the factors affecting the decision to annotate an operation are as follows. These factors are not necessarily independent.

1. *Size of computation.* If faster execution is the aim then the amount of computation required to execute the operation must be large enough that benefit of doing this computation on a remote processor outweighs the overhead cost of dispatching the task. Assessing this overhead cost depends on 2 and 5 below. Estimating the benefit of remote execution must take 6 into account.
2. *Quantity of data to be copied and transmitted.* Within the range of hardware models we consider are platforms consisting of networks of processors with no shared memory. In such architectures the data required for a computation to be executed remotely must be copied to the remote

processor.

3. *Side-effects.* So far our discussion has avoided an important issue. An industrial-grade programming language must support abstract data types and supply a library of ADTs on which applications can be built. Prograph accomplishes this via object-orientation. As a consequence, Prograph is not a “pure” data flow language in that data passing through its diagrams can consist of class instances subject to side-effects. Clearly the integrity of data will be compromised if parallel processes are allowed to return altered copies of an instance. The issue of side-effects is discussed in [6], and in [13] algorithms are presented for analysing Prograph programs to infer datatypes and potential side-effects. To execute these algorithms on any program large enough to be useful is unlikely to be practical. However, in many cases the programmer will know whether blessing an operation is “safe” with respect to side-effects.
4. *Load balancing.* Dispatching a task to another processor makes sense only if the local processor will be gainfully employed while the remote process executes. For example, if both `quicksort` operations in the first case of the `quicksort` method in Figure 2 were blessed, then since the data for both is available at the same time, both would be dispatched leaving the current processor idle.
5. *Communication speed.* The speed of communication between processors affects the overhead associated with dispatching tasks.
6. *Processor speeds and characteristics.* Processors in a target hardware configuration may be different. For example, if the network contains a node consisting of a high-speed array processor, then dispatching a list multiplex to that processor may be worthwhile even if this leaves the current processor with little computation to do in the interim.

4 The Pool Model for Parallel Execution

So far we have discussed remote execution of tasks and annotation for parallelism from the programmer’s point of view, which we consider gives an appropriate but pragmatic level of abstraction. In this section we present the model we have adopted for dealing with the mechanics of the underlying process.

A possible meaning for the bless annotation is that it denotes a *future* in the sense of the mechanism employed in Multilisp [10]. Under this interpretation, the operation and its input data would be packaged as a future, possibly to be exported to another processor. When the return value of the operation is required, the future is inspected (“touched”), and if the result is available, the calling process uses it and continues its computation. If the result is not available and execution of it has not yet begun, the calling process may elect to execute it itself as a conventional subroutine, or may wait.

We have chosen not to use the futures model for several reasons. First, the mechanism requires the runtime system, or even the programmer, to explicitly test the current state of the future. Since futures are first class objects, that is, they can be

passed to and from functions and stored in data structures, such testing is non-trivial. This mechanism imposes an execution overhead on the runtime system of up to 50%, even for a sequential program. Second, the program development environment, and in particular the set of primitives supplied in the language, would require changes to accommodate unevaluated futures. Finally, side-effects which are predictable when all operations in a case must be evaluated before the case finishes become considerably less so if cases are allowed to return unevaluated futures.

We use a conservative model in which executes a blessed operation by creating a task which may be executed remotely, and must always be evaluated before execution of the case where it is created is finished. This containment keeps the programmer's view of the model simple, and promotes reuse of program components which would not be easily achievable were unexecuted tasks, like futures, allowed to flow out of cases.

Like the futures model, the pool model is based on packaging operations and associated data into a task data structure. When a process encounters a blessed operation, it places a task data structure into a central pool of tasks, and later investigates this pooled task in the hope that it has been executed.

A prototype implementation in Prograph of the pool model provides a clear description of how this mechanism works [6]. We will use a modified version of that implementation as the basis for our explanation. Since the Prograph CPX editor and compiler are not easily customisable, annotation cannot be directly incorporated. Instead, operations that would be annotated are expanded as illustrated in Figure 4, where the blessed `quicksort` operation in Figure 2 has been replaced by several operations and synchros as indicated by the grey region in the diagram which arrange for `quicksort` to be executed, as follows. First the inputs to the blessed operation are packed into a list by the primitive `pack`. Next, the universal method `poolit` is invoked with three inputs, the name of

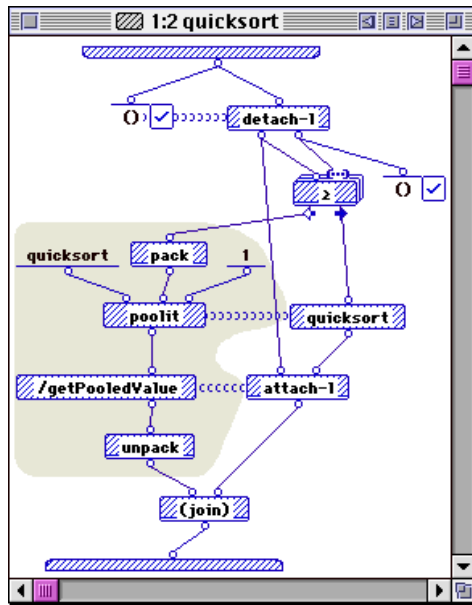


Figure 4: Expansion of blessed quicksort operation

the blessed operation, the list of inputs to the blessed operation, and an integer specifying the number of outputs the blessed operation produces. `poolit` returns an instance of an appropriate class to represent the task, and may also add this instance to a pool of tasks to be performed. The operation `/getPooledValue` receives the task instance, extracts from it the list of results computed by the execution of the task, removes the task from the task pool if necessary, and returns the list of values. Finally, `unpack` extracts the individual values from this list.

A task which has been pooled by `poolit` may be picked up for processing by another processor so that by the time `/getPooledValue` executes, the result is available. Clearly we want to maximise the opportunity for this to happen, which is why the two synchros from `poolit` to `quicksort` and from `attach-1` to `/getPooledValue` have

been introduced in the expansion of the blessed operation. The first ensures that the task is pooled before any other computation takes place in the current case, while the second ensures that as much computation as possible is done in the current case before the value of the task is requested.

In general, a blessed operation is expanded as follows. The operation is replaced by six operations as in the example, where the `pack` and `unpack` operations have the same number of terminals and roots respectively as the blessed operation. The datalinks connected to the terminals and roots of the blessed operation are replaced by datalinks connected to the terminals of `pack` and roots of `unpack` respectively.

In order to describe how synchros are introduced by the expansion, we note that a case represents a partial order on the operations occurring in it, determined by the graph of datalinks and synchros. Prograph determines a legitimate execution order by finding a linear order that satisfies this partial order, subject to the extra conditions that the input and output bars are always first and last respectively, and operations with controls occur as early as possible. Let x be some operation in the case before expanding the bless such that x occurs before the blessed operation in some

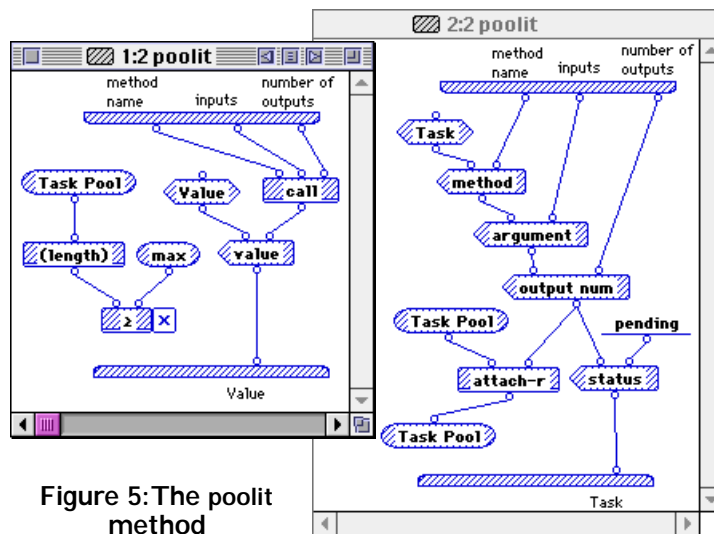


Figure 5: The poolit method

legitimate linear orderings and after in others. Let X be the set of all such operations, and let Y be the set of elements of X which are minimal with respect to the partial order. We introduce a new synchro from `poolit` to y for each operation y in Y . Synchros from other operations in the case to `/getPooledValue` are introduced analogously.

The method `poolit` is shown in Figure 5. The first case checks if the pool, maintained as a list of instances in the persistent `Task Pool`, already holds some maximum number of tasks, in which case the task is executed and the result placed in an instance of class `Value` which is returned to the caller. Putting a ceiling on the pool size provides a primitive throttling mechanism to control runaway pooling. The primitive `call` executes the method named by its first input, passing this method the list of values specified by its second input. The called method produces the number of outputs specified by the third input. These values are returned by `call` packed into a list. The second case of `poolit` creates an instance of class `Task` to record the information about the task, adds the instance to the pool and returns the task to the caller.

Recall that the first case of `poolit` executes a task immediately and creates an instance of `Value` to contain the result of the computation. The method `getPooledValue` of class `Value` simply retrieves this result as shown in Figure 6.

Dealing with a pooled task is somewhat more complicated since it may be pending, executing or executed. The method `getPooledValue` of class `Task` is also shown in Figure 6. If the task is pending, it is removed from the pool and executed as shown in the first case. Otherwise, if the task is executing, the local operation `wait` is repeatedly executed. As shown in Figure 6, in each iteration `wait` checks the status of the task and yields the processor if the task is still executing; otherwise the task is removed from the pool, its result extracted and the iteration of the multiplex stopped. Note that if the task is done when `getPooledValue` is executed, the first iteration of the multiplexed `wait` will execute the second case of `wait` returning the result. “Yielding the processor” here refers to the fact that in the Prograph CPX prototype, management of the task pool and communication with other processors is dealt with in separate threads. Note that a practical system would use a more sophisticated waiting mechanism, such as waiting on an event signifying task completion.

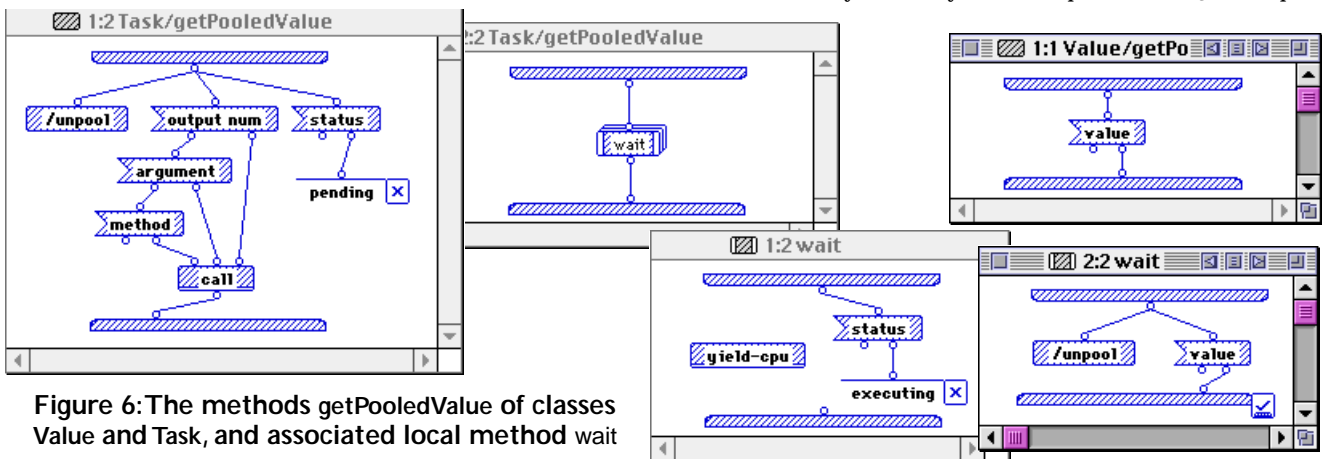


Figure 6: The methods `getPooledValue` of classes `Value` and `Task`, and associated local method `wait`

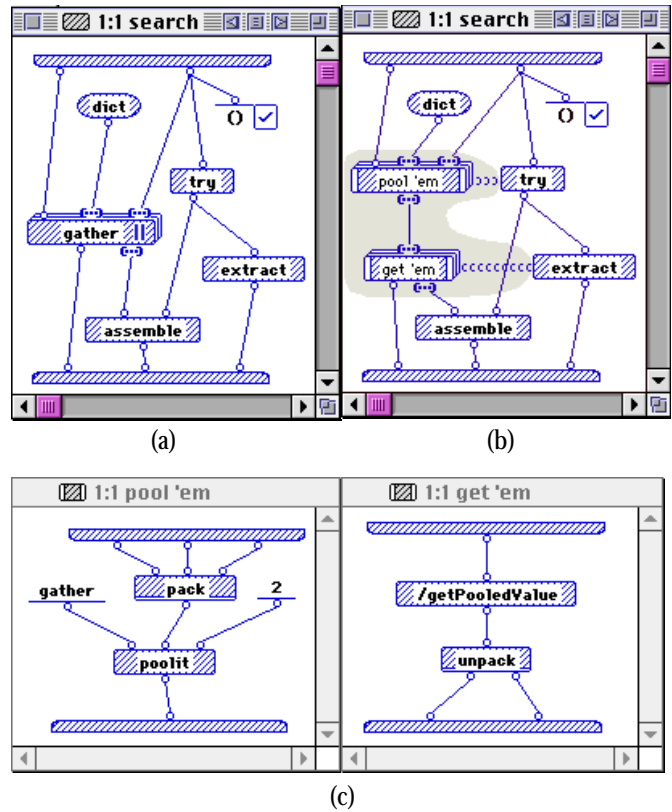


Figure 7: Expanding an internally blessed multiplex

To deal with an externally blessed multiplex, we introduce a new method consisting of a single case containing the multiplex without a `bless` annotation, replace the original externally blessed multiplex with a simple blessed operation referring to this new method, then expand this operation as described above.

Expansion of an internally blessed multiplex is illustrated by the example in Figure 7. The multiplex in the case in Figure 7(a) is replaced by the multiplexes and synchros indicated by the grey region in 7(b). The multiplexed local operation `pool 'em` has the same arrangement of terminals (simple or list) as the original multiplex, connected to roots in the case in the same way. Similarly, the multiplexed local `get 'em` pre-

serves the roots and associated datalinks of the original multiplex. pool 'em has a single list root connected to the single list terminal of get 'em. The methods for the two introduced locals are shown in Figure 7(c). Finally, synchros from pool 'em to other operations in the case, and from other operations to get 'em are added according to the rule defined above for expanding a simple blessed operation.

Each execution of the multiplexed pool 'em operation deals with one element of each of the list inputs to the original multiplex, packing them together with any simple inputs into a list, then invoking the method `poolit` as described earlier. This either creates and pools a `Task`, or evaluates the method in question and creates an instance of `Value`. The effect, therefore, is to do the same "packaging and pooling" for each set of input values as is done for a simple blessed operation. Similarly, the multiplexed get 'em performs the same "retrieving and unpacking" on each of the created `Tasks` as is performed on the single task corresponding to a simple blessed operation.

The result of this expansion is that as early as possible in the execution of the case, one task will be pooled for each application of the operation enclosed in the multiplex, and values returned by these tasks are retrieved as late as possible.

The above description of the pool model is slightly deficient in two respects related to controls. Recall that in Prograph one outcome of executing an operation is *failure*, which must be intercepted by a control on the operation. The above description of the pool model cannot deal with blessed operation that can fail. Neither can it deal with an internally blessed list multiplex that, when executed sequentially, stops before all elements of the shortest input list are consumed. This occurs if the invoked method fires one of the controls *finish*, *terminate* or *fail*. Neither of these issues can be handled by the above Prograph simulation of pooling since control messages are not accessible to the programmer. They are addressed, however, in the implementation described in the next section.

5 Implementation

In this section we describe the implementation of Parallel Prograph currently in progress. This consists of an implementation in Java of the pool model, called the Parallel Prograph Kernel (PPGKERNEL), together with Parallel Prograph Client (PPGCLIENT), a version of Prograph CPX modified to support blessing and employ the services of PPGKERNEL. Although the goal of PPGKERNEL is to support PPGCLIENT, client programs could be written in any language. Currently, a preliminary version of PPGKERNEL exists: PPGCLIENT is still in the design stage.

5.1 PPGKernel

PPGKERNEL consists of a task pool manager together with three thread sub-systems which select tasks from the pool to be executed locally or to be exported to a remote processor. *User* threads execute the client program. An *idler* thread periodically examines the task pool and spawns new user threads to evaluate pooled tasks. A *communications* thread selects tasks and dispatches them to other processors in

the distributed machine, and receives tasks dispatched from other processors, which it passes to the pool manager for inserting into the task pool.

In a loosely coupled architecture, each processor runs PPGKERNEL, providing services to a client program. Processors in such a machine are not necessarily identical, so each PPGKERNEL is configured according to the characteristics of its host processor. Configuration parameters include pool capacity, whether tasks should be actively sought or passively received, number of user threads, and whether or not a processor should competitively execute tasks which are under execution by other processors (racing). Note that allowing multiple user threads provides the model with fault tolerance, since a processor can decide to evaluate an exported but unfinished task.

In SMP machines, the implementation of threads provided by the operating system deals with farming out processes to individual processors, making a communications thread in our kernel unnecessary.

The design of the PPGKERNEL API follows the description of the pool model in section 4. At its highest level, the API provides methods `PoolIt`, `PoolAll` and `PoolItAll`. `PoolIt` implements blessing of simple operations and external blessing of multiplex operations (Figures 2 and 3(a)); `PoolAll` implements internal blessing (Figure 3(b)); and `PoolItAll` implements combined external and internal blessing (Figure 3(c)). Each of these operations takes a single `BlessedOp` object as argument, places a `Task` object in the pool and returns a `Proxy` object referring to the `Task`. The method `GetPooledValue` can be applied to the `Proxy` to obtain the list of outputs.

At lower levels the API lets the client become more involved in particular task-to-processor matching decisions. At the lowest level of abstraction, the client can bypass the task pool altogether, sending an operation plus arguments to a named processor for evaluation.

In the Prograph prototype described in section 4, `Task` objects have a `status` attribute which can have any of the values *pending*, *executing*, or *completed*. To accommodate blessed operations which can fail the `status` attribute of `Task` objects in PPGKERNEL can also have the value *fail*. This flag is returned to the user thread when the result is retrieved via the associated `Proxy`. In effect, the execution message is stored by the runtime system until the blessed operation's output values are requested.

To deal with early termination of internally blessed multiplexes, we add the values *finish*, *terminate* and *fail* to the repertoire of the `status` attribute. The `PoolAll` operation creates an instance of `MultiplexProxy`, a subclass of `Proxy`, which maintains a vector of the `Tasks` corresponding to each of the executions of the multiplexed operation. When `GetPooledValue` is applied to it, the `MultiplexProxy` inspects its task vector. If the `status` of one of the `Tasks` is *finish*, *terminate* or *fail* and the `status` of each of the `Tasks` to its left is *completed*, `MultiplexProxy` composes appropriate values to return to the user thread, and withdraws from the pool all remaining `Tasks` in

its task vector which are *pending*. This behaviour preserves the sequential semantics of multiplex execution. Note that when `MultiplexProxy` determines that the multiplex execution is complete, it does not kill any of the remaining `Tasks` still *executing*. This will be implemented in future versions of `PPGKERNEL`.

5.2 PPGClient

`PPGCLIENT` will consist of modified versions of the Prograph `CPX` editor, incremental and optimising compilers, together with a `PPG` class library that extends the application framework.

The Prograph `CPX` application framework consists of Application Building Classes (ABCs) which deliver standard functionality such as user interface management and file management, and Application Building Editors (ABEs) which provide editors for the ABCs. To add new functionality, one extends the ABCs by subclassing or adding new, independent classes, and by adding classes to the ABEs to implement editors for the new ABC classes. The `PPG` library consists of two such extensions.

The Platform Specification Classes incorporate information about the target computing platform into the project that defines an application. Using the associated Platform Specification Editor, the programmer defines such things as the communication network topology, communication speeds, processor speeds, and parameter settings for the `PPGKERNEL` on each processor. Some defaults are assumed if the programmer chooses not to define any of these characteristics.

The Annotation Classes define various types of blessing. The Annotation Editor is used to create new classes of blessing and to customise blessings on operations. For example, the programmer can make remote execution conditional on such things as the size of the input data, or direct the `PPGKERNEL` to use a specific processor for execution. The latter customisation relies on structures built with the Platform Specification Classes. If the programmer chooses not to customise a blessing, a default will be assumed.

Since blessings are defined by classes in Prograph, the Prograph `CPX` editor will need to be modified so that blessing information, expressed in terms of these classes, can be attached to the internal representation of an operation. The incremental and optimising compilers will also need to be modified to use this information in communicating with the `PPGKERNEL`.

Initially the decision to bless an operation will be left to the programmer, who, we believe, is in the best position to know whether an operation is a good candidate for remote execution. However, we expect that by making suitable approximations erring on the conservative side, we will be able to find less expensive forms of the current analysis algorithms [13], thereby providing support for annotation decisions. Another possible approach which would work in concert with the analysis algorithms, would be to collect information about types and side-effects during prototyping on a sequential processor. Tools based on such analysis and

empirical information will eventually be incorporated into the annotation editor.

6 Discussion and related work

The current practice when building applications to execute on parallel hardware is to combine a sequential language such as C or Fortran, with a set of distribution primitives, such as those provided by PVM [9] and more recently MPI [16]. The concurrent programming model offered by such a combination relies on message passing, and leads to code in which details of the algorithms are intimately entwined with the low-level detail of process coordination and communication. Higher level models, offering a clearer separation of program components, include BSP [15], Linda [2] and Nexus [7].

Visualisation has played a role in parallel and distributed computing for many years, from paper-based diagrams for understanding parallel computation, to program animation and visual debugging aids, for example [3, 12]. Because of the natural way in which pictures expose the parallelism in algorithms, visual languages have from time to time been proposed as a basis for programming various kinds of parallel execution models. An early example is GPL, a simple visual dataflow language developed to program the DDM-1 dataflow machine built at the University of Utah in the early 1980s [8]. A more recent example is Pictorial Janus, a visual concurrent logic programming language [11].

We consider that the major obstacle to programming applications for distributed and parallel systems at present is the programmer's ability to deal simultaneously with the complexity of the algorithms under consideration, the parallelisation of those algorithms, and the details of communications between processors. We believe that to alleviate this difficulty, it is necessary to devise appropriate programmer models and languages for parallelism, and that visual programming languages and environments have an important role to play. To our knowledge, there are no practical visual programming systems for developing parallel applications on a par with the Parallel Prograph system we propose.

7 Concluding remarks

In this paper we have reported work in progress on implementing a Prograph-based visual programming environment for parallel applications. Our goal is a practical system in which the sequential syntax of Prograph is preserved so that a programmer can build an application on a sequential machine in the normal way, making use of the comprehensive tools provided by Prograph `CPX`, and deal with parallel deployment once the application is debugged.

To this end we have proposed and implemented *pooling*, a conservative model of parallelism based on "blessing" operations to take advantage of the dataflow and multiplex parallelism exposed by the Prograph language.

The parallel execution engine `PPGKernel` is implemented as a hierarchical API which deals with processors, resources, explicit task placement, and at the lowest level, control messages through sockets. Exposing this layer to the user provides

some of the facilities which are traditionally associated with distributed, as opposed to parallel, computing.

When an early version of this kernel running on networked Pentiums was tested with some genetic programming algorithms, a seven-fold performance improvement over the same algorithms on a single processor was observed [14].

To exploit this kernel, we are currently designing PPGCLIENT, a version of Prograph CPX modified to support blessing at the editor and compiler level, and extended via additions to the application framework to provide high level facilities for specifying platform characteristics and customising blessing annotations.

8 Acknowledgments

The authors acknowledge support from Apple Computer and Pictorius Inc. during this project

9 References

- [1] H. Baker, C. Hewitt, The incremental garbage collection of processes, *Proc. of the Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices* 12, (1977), 55-59.
- [2] N. Carriero, D. Gelernter, Linda in Context, *Communication of the ACM* 32(4), (1989), 444-458.
- [3] K.C. Cox, G-C. Roman, Visualising concurrent computations, *Proc. IEEE Workshop on Visual Languages* (1991), 18-24.
- [4] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, *Proc. IEEE Workshop on Visual Programming*, Rome (Oct 1989), 150-156. Reprinted in *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg and T.G. Lewis (Eds), Manning Publications (1995).
- [5] P.T. Cox, T.J. Smedley, Visual Languages for the Design and Development of Structured Objects, *Journal of Visual Languages and Computing*, v8, Academic Press (1997), 57-84.
- [6] P.T. Cox, H. Glaser, B. Lanaspri, Distributed Prograph, *Parallel Symbolic Languages and Systems*, T. Ito, R.H. Halstead and C. Queinnec (eds.) Springer Verlag LNCS 1068 (1996), 128-133.
- [7] I. Foster, C. Kesselman, S. Tuecke, The Nexus Task-parallel Runtime System, *Proc. of 1st International Workshop on Parallel Processing* McGraw-Hill (1994), 457-462.
- [8] A. L. Davis, S. Lowder, A Sample Management Application Program in a Graphical Data-Driven Programming Language, *Proc. of IEEE COMPCON*(1981), 162-167.
- [9] A. Geist, A. Berguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing* MIT Press (1994)
- [10] R.H. Halstead Jr., Multilisp: A language for concurrent symbolic computation, *ACM Trans. on Programming Languages and Systems*, 7(4) (1985), 501-538.
- [11] K. M. Kahn, V. A. Saraswat, Complete Visualizations of Concurrent Programs and Their Executions, *Proc. IEEE Workshop on Visual Languages*, Skokie, IL, (1990), 7-15.
- [12] H. Koike, T. Takada, T. Masui, VisuaLinda: A Framework for Visualizing Parallel Linda Programs, *Proc. IEEE Visual Languages Symposium*, (1997), 174-180.
- [13] B. Lanaspri, *Static Analysis for Distributed Prograph*, PHD Thesis, Dept. of Electronics and Computer Science, University of Southampton (1998).
- [14] S. Maclean, H. Glaser, *Execution Model Based on Graphical dataflow*, Working Paper, Dept. of Electronics and Computer Science, University of Southampton (1998).
- [15] W.F. McColl: BSP Programming, *Specification of Parallel Algorithms Proc DIMACS Workshop*, Princeton(1994), 21-35.
- [16] Message Passing Interface Forum: MPI - A message-passing interface standard, *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), (1994)
- [17] Pictorius Inc., *Prograph Reference Manual*, (1996).