

Visual, Object-Oriented Development of Parallel Applications

James Webber (James.Webber@ncl.ac.uk)
P. A. Lee (P.A.Lee@ncl.ac.uk)
Department of Computing Science,
The University of Newcastle upon Tyne,
Newcastle upon Tyne.
NE1 7RU
United Kingdom

Abstract

Parallelism is always going to be required to support the computational demands of some problem domains, and will continue to be exploited via "traditional" parallel processing methods and languages. However, this paper argues that parallelism will also become a requirement for the non-specialist user, but that the traditional parallelism languages and techniques do not have the right support for the engineering of large-scale, parallel applications. The paper discusses this issue, and presents a visual, object-oriented parallel programming language, Vorlon, which addresses the management of both problem domain complexity and implementation complexity, to support the development of general-purpose parallel applications by programmers who are non-specialists in parallelism.

1. Introduction

Parallel processing has always been a necessity in specialised problem domains, such as weather forecasting and simulation, where the performance of the (current) single processor was insufficient. Parallel programmers in these domains are usually specialists willing to take on the added complexity of exploiting parallelism on a particular system. In other, more general-purpose problem domains, the need for parallelism is continually being espoused, but time and time again technology pundits have failed to anticipate the continuing success of semiconductor manufacturers in following the predictions of Moore's First Law [2] and hence removing the need for exploiting parallelism.

2. Moore's Laws and the Software Stretch

Proponents have always backed parallel computing as the most promising means of circumventing physical limitations on computational speed of semiconductor technology [1]. Time and again technology pundits have

been proven wrong as Moore's First Law has held true to form. The technological barriers once envisaged have been, and shall continue to be, surmountable. Indeed, Moore has stated that there are no likely technological barriers impeding the continuity of the first law. However, that is not to say that the first law *will* continue to hold indefinitely. More recently, Moore's Second Law, while maintaining consistency with the first law on technological issues, observes that issues of economics may well be the limiting factor in the performance of uniprocessor computing systems [3].

The Software Stretch

The essence of the crisis induced by the effects of Moore's Second Law is not dissimilar to the original software crisis of some two decades ago. The crux of the problem is that the demands which users place upon (parallel) software is beginning to outstrip the ability to produce such software. This phenomenon, the "Software Stretch," is visualised in Figure 1.

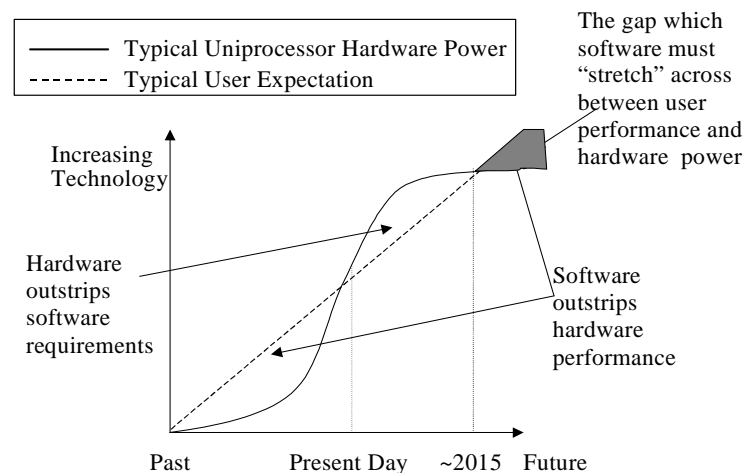


Figure 1. The Software Stretch

In Figure 1, the increasing computational ability of uniprocessor computing systems is seen as a solid, ogive-like curve. As is evident, the curve is currently at a point consistent with Moore's First Law whereby the power of such systems growing at an exponential rate. However, the curve is worryingly punctuated by two major events. Firstly, the rate of increase of computational power begins to level-off in accordance with Moore's Second Law, increasing the length of time between successive generations of computer hardware. Secondly, and more worryingly is the fact that there comes a point where the required performance outstrips the performance offered

by hardware. At the point where hardware cannot evolve sufficiently quickly to support increasingly complex software, the software itself is “stretched” between linearly increasing user expectation, and less than linearly increasing hardware performance. This is a fundamental and rather alarming point for software engineers, in that it is software which must bridge the ensuing performance gap.

While there is little room for complacency about the likelihood of the software stretch occurring, there is room for a little optimism about possible solutions to the problem. In particular, the application of parallelism has provided, within certain problem domains, the required increase in software performance, without the otherwise necessary requirement for increases in hardware performance.

Though parallelism may well be a valid method of avoiding or alleviating the software stretch, it is clear that the software engineering community is not yet prepared for the transition from a largely serial processing form to a predominantly parallel form. Although the concepts underpinning parallel programming are sound and well known, the practise of parallel software engineering remains immature, though effort is beginning to be invested in maturing the techniques [4]. This in itself should not come as much of a surprise. The parallelism community, by-and-large, is not concerned with software engineering issues, but solely with increased computational performance, or speedup over the equivalent serial program. The almost exclusive focus on speedup of parallel applications comes at a cost. Parallel applications are notorious for being architecture-specific, difficult to maintain or re-use, and often address specific and well known problem domains (e.g. numerical analysis). Rigorous software engineering is rarely applied to the construction of parallel software to anywhere near the degree employed for serial software.

Sequential software engineers must also shoulder some of the responsibility, in that for too long now they have produced ever more fanciful and ambitious software, well engineered as it may be, which has been completely reliant upon advances in uniprocessor computational power (e.g. GUIs). This dichotomy of skilling is unfortunate.

The Orthogonal Abstraction Problem

Parallel programming is not necessarily an easy way out of the problem. For the majority of problems undertaken by traditional parallel programmers, the tools already available, such as HPF [5], PVM [6] and MPI [7], are sufficient. There is often no need to future-proof, maintain, or otherwise engineer parallel software to the same level required by other application areas. So while the implementation of parallel software remains intricate, and indeed no-one is suggesting that parallel programming is by any means a trivial discipline, the scale of the

software implemented by today's parallel-programming community is small enough such that it is tenable, while the problem domains under consideration themselves are well understood.

Although parallelism is known about and practised "in the small," the application of parallel programming techniques to large-scale software systems is a daunting task, and one to which current parallel-programming techniques, with their implementation-centric methodologies, are simply not suited.

This is a problem which requires a bold solution. It is almost certainly not the case that current parallel programming techniques can somehow be tailored to be usable by the present-day serial programming community. What mainstream software engineers need is not solely computational speed, but also powerful paradigms through which complex problem domains can be conceptualised and managed.

Fortunately, there have been several notable attempts to reduce the implementation complexity of parallel programming through the use of visual programming techniques [8-15]. The fundamental axiom of these techniques is that the complex control-flow behaviour of parallel programs can be more intuitively represented of graphically rather than by the textual annotations which traditionally comprise programming languages. However, despite their promise, current visual parallel programming languages have supported traditional parallel programmers rather than trying to support fledging developers in the parallelism arena. That is to say, the programming paradigms which visual parallel programming language designers have chosen to adopt have reflected those in common use by the parallel programming mainstream, which in turn are heavily influenced by parallel hardware itself.

Ironically, the same visual parallel programming technology which has not taken the parallel programming world by storm is unsuitable for non-specialist developers. Though implementation complexity of parallel applications can be managed through the use of graphical notations, the fact that visual parallel programming languages tend to support architecturally-influenced modes of computation (such as passing values between processes) renders them underpowered for abstracting the complexity of any reasonably sized problem domain. This mismatch in required abstractions can be summed up by Figure 2.

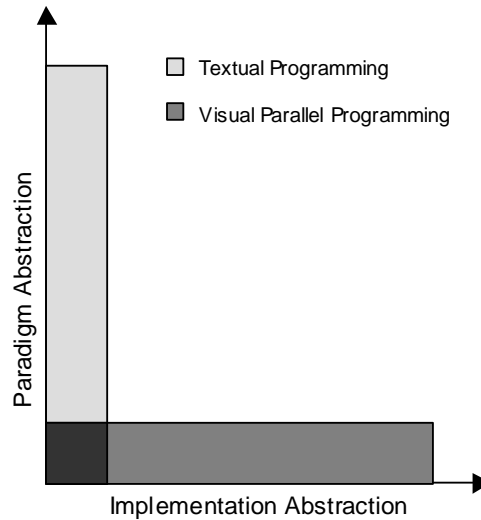


Figure 2. The Orthogonal Abstraction Problem

Figure 2 demonstrates the primary difficulty that lies in the transfer of technology between the parallel programming community and the mainstream of software engineering. If the *overall* level of abstraction provided by a particular implementation style is given by the area covered on the plane, and the horizontal and vertical dimensions give the expressive power in terms of implementation abstraction (the expressive power of the source code syntax) and paradigm abstraction (the power of the software methodology as a whole) respectively, then a pattern of available abstractions emerges. The textual programming languages in common use provide highly abstract models of computation, through which problems can be conceptualised and managed. Object-orientation is a prime example of such a present-day model. Although implementation may be based on intricate textual syntax, the actual implementation complexity, being free of parallel activity, is manageable. Conversely the visual parallel programming languages touted as the way forward for parallel programming suffer in the opposite fashion. That is, visual parallel programming languages offer a syntax which is often abstracted from the complexities of the underlying (parallel) system, and may even implicitly shoulder the burden of low-level locking, synchronisation, and suchlike. However, they do not offer a sufficiently abstract model of computation to warrant their uptake in more general software engineering, since their computational models reflect those of Fortran-like environments, and not of typical problem domains.

Seeing the disjoint promise of both techniques naturally leads to the notion of combining abstract programming models and programming language syntax to produce a software development methodology which has both an abstract programming model for problem domain management, and a syntax which abstracts the complexity of

parallel computing hardware. Visual, object-oriented parallel application development is one such combination to be explored further in this paper.

3. The Parallel Object-Flow Execution Model

The parallel object-flow execution model draws upon both object-oriented and dataflow models. As such it is able to manage both parallelism-oriented aspects, like synchronisation with dataflow-like constructs, and problem domain complexity through types and type interrelations. It is not, however, to be confused with the Object-Flow model used in Prograph [10, 16], which although supporting a concurrent programming model, is not a model aimed at high-performance computing.

The parallel object-flow model borrows heavily from the look-and-feel of other visual parallel programming languages. Parallel object-flow graphs consist of nodes and arcs: nodes represent some form of computational element; and arcs describe control-flow interdependencies and possible transportation routes for parameters to pass between nodes. The parameters passed between nodes in parallel object-flow graphs are unlike parameters in other visual parallel programming methods, in that they are in fact handles, or references, to objects stored somewhere within the computing system, and not actual data or control-flow signals. That is to say that the developer does not have to be concerned with the actual location of a particular object within a computing system, since all access to objects is performed through the use of handles. Similarly the nodes representing computational elements in fact usually represent method calls on objects as opposed to separate functions operating on input data. A typical parallel object-flow method graph can be seen in Figure 3 below.

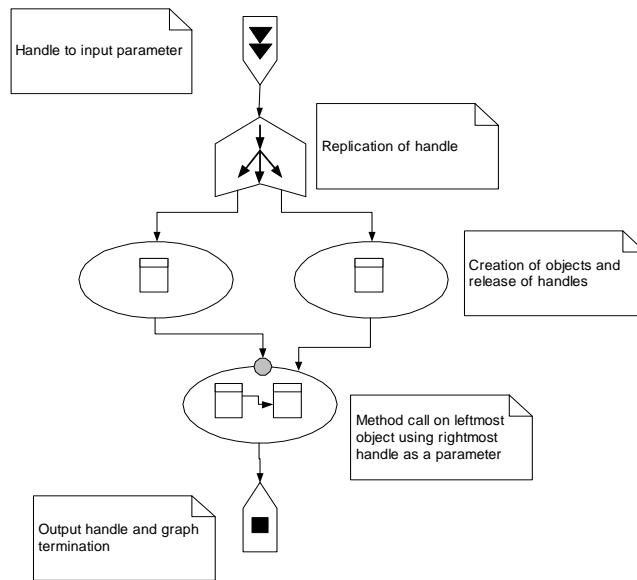


Figure 3. The Parallel Object-Flow Execution Model

Although the diagram in Figure 3 is relatively small, it highlights the most of the salient features of the parallel object-flow execution model. Of particular importance are:

- All concurrent activity is presented in its most intuitive form: visually. Only sequential code is allowed to be in textual form within nodes.
- It is handles to objects and not objects themselves which flow along arcs. Transmission of large objects across slow interconnects can be avoided, particularly in situations where the object is to carry out a service rather than provide volumes of data to the requester.
- Since handles to objects may need to be replicated, concurrency issues pertaining to the possible existence of multiple handles to a single object must be handled, either by the language itself, or by a supporting run-time system.
- New objects may be created somewhere in the memory of the parallel system, though only handles to those objects are released onto graphs. Users need not be concerned with physical object locations, which may not even be constrained within the memory space of a single computer.
- Method calls are made in places where computational nodes are normally expected. Methods themselves are specified graphically in a similar fashion to that of Figure 3. Thus a method call can be seen simply as providing sufficient input parameters to another graph, such that its execution can proceed. A typical application may consist of many method graphs and similarly many objects and handles.

- A termination node indicates the point at which a result may be returned from the execution of a graph.
- Where desirable, purely sequential textual object-oriented source code may be entered into special purpose graph elements. Thus the developer may choose to view the graph-level constructs as a co-ordination language much as ParADE [8] and VPE [14] developers view their languages. This convenient feature circumvents a common argument levelled against visual programming techniques, in that they are often considered to be extremely verbose for simple programming tasks.

The features offered by the parallel object-flow paradigm mean that the user benefits both in terms of ability to manage problem domain complexity, through object-orientation, and implementation complexity, through an abstract visual syntax. The developer is never forced into a parallelism-centric mode of operation. The developer's paradigm is simply object-orientation, yet because of the nature of the programming language syntax, potential parallelism is trivial to extract automatically.

Furthermore, since the parallel object-flow paradigm gains benefits from existing object-oriented technologies, it comes equipped with full lifecycle support as does any other object-oriented application. That is, to engineer software with parallel object-flow languages, it is only the implementation phase of the software lifecycle which is affected, while other activities such as analysis and design are not.

4. The Vorlon Programming Language

Vorlon is a visual parallel programming language under development in the Department of Computing Science, University of Newcastle upon Tyne. Vorlon adheres to the parallel object-flow model described above and is intended to be a first research vehicle exploring the utility of this model. Although the parallel object-flow paradigm is largely concerned with specifying parallel activity through user-level method graphs, graphs themselves are not enough to successfully build and execute parallel applications. For an actual language implementation two further layers of abstraction which insulate the developer from the underlying parallel system are required. These layers are necessary to provide both concurrency control features, and run-time resource management to the application. As such, the graphs which the user constructs are considered to reside at the top-level of the application architecture, supported by a concurrency control layer, which in turn is managed at run-time by a parallel run-time system.

Vorlon Application Architecture

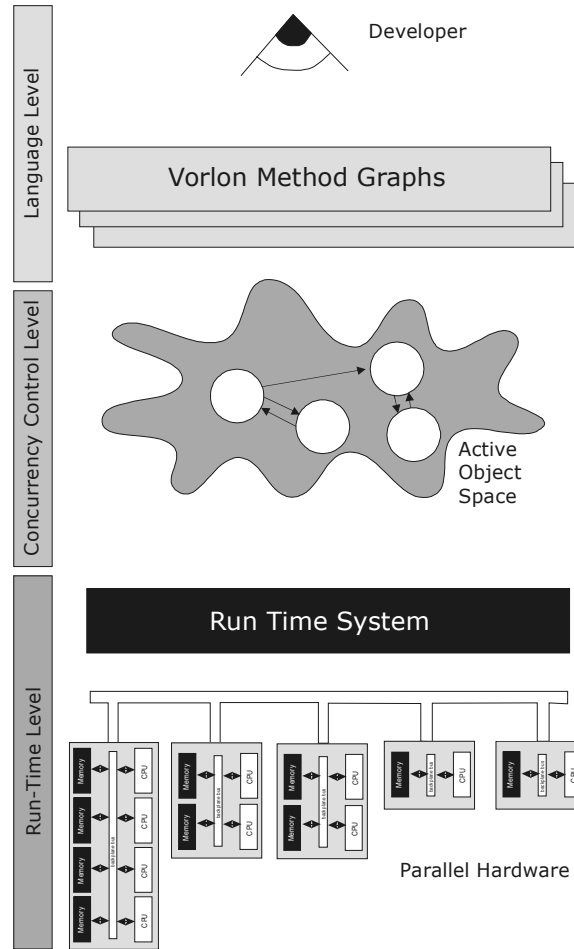


Figure 4. Architecture of a Vorlon Application

Figure 4 shows the conceptual architecture of a Vorlon application. In the uppermost level, the developer and the method graphs as described above reside. It is here that the developer constructs an application by constructing a data dictionary which declares types and type interrelations from the problem domain, and by implementing methods belonging to those types using the parallelism-friendly visual syntax. This construction is completely underpinned by the object-oriented paradigm, and is insulated from the complexity of the underlying hardware. This in itself is an important point: the programmer is able to deal with problem domain complexity through object-orientation, while hardware complexity is hidden by the visual syntax of the Vorlon programming language.

Concurrency Control in Vorlon

The next layer down the architecture is the concurrency control layer. In the Vorlon programming language the concurrency control mechanism chosen was the Active Object model [17, 18] with some relaxation to support increased parallel activity. Each object instantiated by method graphs at the language level logically, though for

reasons of efficiency not actually, becomes an active object at run-time. Under the canonical active object pattern, each object is considered as an autonomous computational unit, which manages a queue of incoming requests for service (method calls) which it will execute in order. Active objects in Vorlon gain the same benefits in terms of elegant concurrency control that the standard active object patterns epitomise, but have two significant performance benefits over the original model. Firstly Vorlon active objects are more relaxed about the ordering of execution of incoming requests for service. In particular, if several consecutive requests for service requiring read-only access arrive then all of these can be executed in parallel providing sufficient computational resources exist. Any requests for methods that may update the state of the serving object are executed one at a time, giving complete mutual exclusivity and thus safety. If this effect is multiplied across the number of objects in an application, all of which may potentially be executing several methods, the level of potential parallelism in even a modest application could be significant, yet it comes at no cost to the developer.

The second important factor in the performance of Vorlon programs over traditional active object based applications, is the fact that not only may parallelism occur at the object level, where each object may be executing several methods concurrently, but also the methods themselves may be highly parallel. Again the parallel nature of Vorlon object methods comes at a low cost to the developer, since the visual syntax allows parallelism to be implicitly extracted.

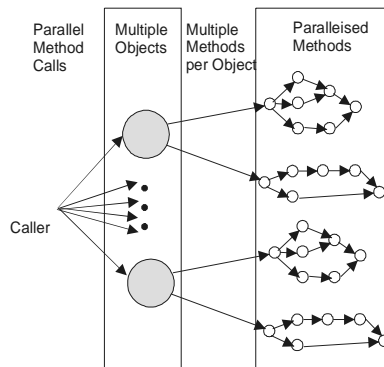


Figure 5. Exploiting Maximal Parallelism in Vorlon

Managing Runtime Parallelism

With the realisation that the level of available parallelism in a Vorlon application is likely to be high, a whole array of performance related problems appears. With such a potentially large number of parallel tasks on offer from a Vorlon application, any hardware supporting the activity may well be swamped with parallel tasks, particularly on

commodity parallel hardware where the number of available processors may be comparatively small and communication between processors low in bandwidth and highly latent. Furthermore, not only may the underlying hardware architecture be swamped with a large number of parallel tasks, but the computational size of those tasks may be small compared to their initialisation or intercommunication costs. That is to say, it is likely that many *fine-grained* tasks could be produced, and as a consequence application performance would suffer.

It is important to note that the problem of exploiting fine-grained parallel activity in the Vorlon application architecture does not fall to the concurrency mechanism. Instead, a second, low-level layer has been introduced into the Vorlon application architecture, whose purpose is to manage the mapping between logically parallel tasks and the hardware resources available to execute those tasks. This decoupling between concurrency and execution layers allows the adoption of more ambitious and powerful run-time systems than would be possible than if run-time and concurrency management were viewed as a single entity. Since the details of the run-time system are kept hidden from layers higher up the architecture, the run-time system can operate in an efficient and low-level way as required without polluting the simple developer-level view of the application as being objects plus elegant concurrency model.

The NIP parallel run-time system [20, 21], has characteristics which are suited to Vorlon's own behavioural model. NIP's philosophy is to take a computer, or collection of computers, and represent that architecture as a single computing system, albeit one with a large memory and number of processors. However, the NIP system differs further from the traditional parallel run-time system approaches (such as PVM [6], MPI [7] and Linda [19]) in that it does not leave the user to manage the virtual machine that it provides.

Under the NIP regime potentially parallel tasks are not immediately executed as they become available, but are simply registered as existing (they are said to be *activated*) and placed in a queue where they wait until a processing resource becomes available (lazy task creation and execution). The interesting point about this scheme is that if no more than a single processing resources become available, the tasks are executed sequentially. From the point of view of a Vorlon program this behaviour is quite suitable since it has the side-affect of increasing the grain size of the potentially fine-grained parallel computations that Vorlon code may produce.

If processing resources become available during the execution of an application, tasks originally destined for sequential execution can be *stolen* from their original queue and offered for execution to another processor. While normally this would pose a significant opportunity to re-introduce problems of transferral of single fine-grained

tasks between processors, the NIP system takes account of its hardware environment and moves tasks not individually, but in groups which maintain an efficiently exploitable grain-size.

Thus instead of adopting the approach of previous visual parallel programming languages, where graph-level objects map onto physical operating system level processes, graph level objects in Vorlon map onto the more abstract NIP work unit, known as the *tasklet*. That is to say graph level parallelism in Vorlon methods is distinct from actual run-time exploitation of that parallelism. Since it is often the case that logical parallelism and physical parallel hardware are not complementary, abstracting the details of one from another allows performance tuning to occur automatically. Moreover, if run-time conditions change during the execution of an application, the NIP run-time system is able to reconfigure the work pattern of the parallel tasks identified in the Vorlon method graphs to optimise task execution.

Vorlon Syntax and Semantics

Though the Vorlon language is underpinned by a novel parallel programming paradigm, it maintains a certain degree of similarity with previous work in visual parallel programming. Object methods are acyclic graphs composed from nodes representing various forms of computational activity (creation of objects, method calls on objects, units of computation) and arcs which represent control-flow dependencies and communication paths between nodes. The nodes themselves become active when they have a full compliment of incoming handles to objects, one per incident arc. Once activated, the implication is that all predecessor nodes have completed their execution, and the current node may then proceed.

The different kinds of Vorlon graph elements can be categorised as follows:

- sources and sinks
- arcs and related control-flow manipulators
- loop and stream nodes
- computational and parallel computational nodes
- object interaction nodes

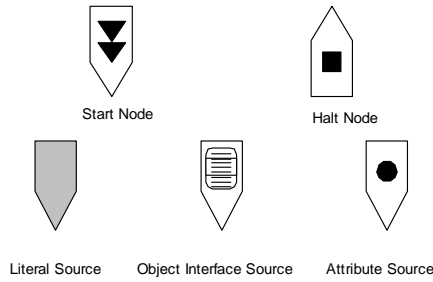


Figure 6. Vorlon Source and Sink Nodes

Start and halt nodes from Figure 6 provide a means of de-coupling the calling graph from the called graph in an application. There are also nodes which allow access to the current object.. Since Vorlon has a strict firing rule for its nodes in that all handles required activate a node must be delivered to that node via arcs, a mechanism is required to provide access to elements of the object itself. The attribute source delivers sub-objects of the current object (its *data members* or *attributes*), while the object interface source provides access to the current object's interface so that other methods may be called from the same object. The final types of source/sink nodes in Vorlon is the literal source which is used to introduce instances of value types into an application.

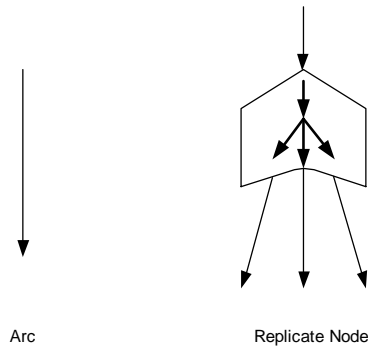


Figure 7. Arcs and Control Flow Manipulators

The standard arc simply provides a route via which a handle can pass between its source and destination nodes. The replicate node provide a means for making copies of object handles across a set of arcs, supporting the construction of arbitrary control-flow patterns within a graph.

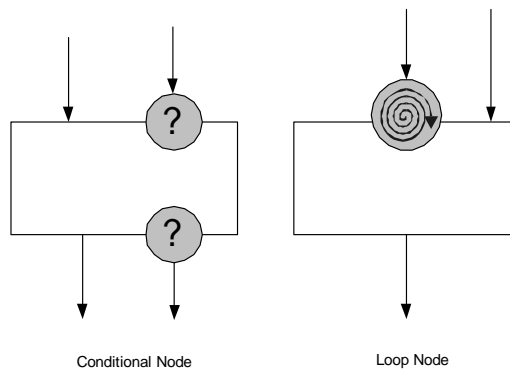


Figure 8. Loop and Conditional Nodes

Since Vorlon graphs are acyclic, a mechanism for repeating sections of code without breaking that acyclic syntax is required. The loop node, based on the Loop Actor in the ParADE language [8], provides such a mechanism by relaxing the strict firing rule found in other Vorlon nodes. The loop node encapsulates an entire sub-graph whose contents are repeated so long as the loop node's repeat condition remains valid. While the condition is valid, the loop node will create and dispatch object handles into a new copy of its encapsulated sub-graph taking a set of handles from input arcs as parameters for each iteration. The conditional node works in a similar fashion to the loop node, with the exception that since it is designed to support conditional execution. Activity within the nodes' sub-graphs is completed (or potentially not executed in the case of the conditional node) before the node signals its own completion in accordance with the normal firing rules.

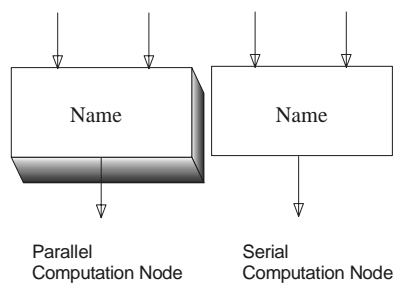


Figure 9. Computation Nodes

The computation nodes are the basic building blocks of the Vorlon programming language. Each computation node provides an environment within which sequential code written in a standard sequential object-oriented language (C++ in the prototype) resides. This code provides the means to express computations in a compact way where the

visual component of the language (used for control-flow) would be less suitable. The mapping between textual and graph level entities is completely automated and as such nodes may be programmed in isolation from one-another and the graphical parts of the application. The serial computation node is activated by a full compliment of object handles reaching its uppermost edge, which are then translated into a form suitable for the textual language to work with. The computation proceeds and at some point later in time a single output handle is produced from one of the outgoing arcs on the underside of the node. Although there may be many such outgoing arcs, only one ever contains an output per activation of the node, and as such conditional execution may be achieved. The parallel computation node (Figure 9) works in much the same way, apart from the fact that multiple instances of the node may execute concurrently given sufficient input object handles. Such handles may either be streamed into the node individually, or be supplied as part of a Vorlon list data-structure which the node will automatically decompose into individual elements as it is received, in a similar fashion to the Prograph system [10]. The important point to note about the parallel computation node is that execution order is preserved. Whatever the relative computation time taken of each of the instances of the node the computation invoked by an earlier arrival of input object handles will always logically complete before a computation invoked by a later set of input object handles. Furthermore, if input is provided in the form of an automatically decomposed list then output will also be contained within a list structure, or structures depending on the outcome of the conditional execution logic embedded within each of the invocations of the node – it is possible for one large list of input parameters to produce several smaller lists of output.

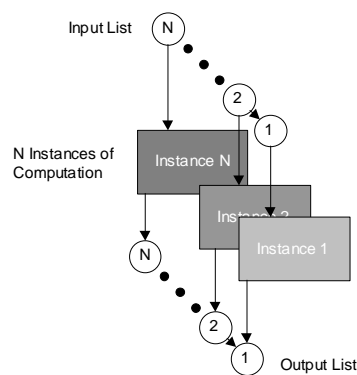


Figure 10. Automatic Decomposition of Parameters for Parallel Computation Nodes

Although the features for automatic decomposition of parameters to a computation node are by no means as powerful as those proposed in the ParADE system [8] where automatic templates for the decomposition,

transmission and re-composition of data were available, the lessons learned from that work have not gone unheeded. The need for automatic decomposition of inputs and outputs to data-parallel tasks was lacking in earlier visual parallel programming systems [9, 14, 15, 22], much to their detriment and although Vorlon is not data-oriented as those systems were, the ability to perform the same task in parallel to some run-time specified level was though important.

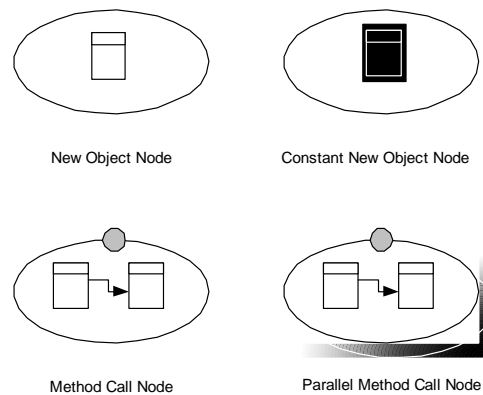


Figure 11. Object Interaction Nodes

Since Vorlon is an object-oriented language, one of the largest sets of primitives is devoted to the management of interactions between objects. The top row of Figure 11 depicts the two mechanisms via which objects are created and handles introduced into a Vorlon method graph. Both versions of the new object node take parameters for the construction of new object via arcs incident upon their uppermost edge, and release a single handle to the newly created object through an arc on their lowermost edge. The constant new object node has the distinction that it produces a handle to an object which is immutable.

The two method call nodes have similar execution semantics to the computation nodes described earlier. Each invokes the method named on the node icon, on an object whose handle appears on the greyed circle on the uppermost edge of the node, and passes as parameters to that method call handles from other arcs incident on the uppermost edge. When a method is called the name of the method to be called along with any parameters required are passed to the serving object. The serving object then queues that request for method invocation until it is ready to be served in accordance with the relaxed active-object execution model.

. The parallel method call node is used where the same method is to be called on multiple objects. It has two possible, though equivalent execution modes, as seen earlier in Figure 10. Either:

- handles to serving objects and required parameters can be streamed into the node, invoking one instance of the method call on an object per complete set of input handles; or
- list structures (one for the serving objects, and one for each subsequent required parameter) which the node automatically decomposes and feeds to individual invocations of the node can be used.

Although the same overall semantics can be achieved by using several of the standard synchronous method call nodes, as exemplified in Figure 12 below, using the parallel version has the significant advantage of not fixing the level of data-parallelism at compile-time. Instead, the parallel synchronous method call node can produce as much potential parallelism as the particular execution of the application demands, and as such, in common with the parallel computation node, is the preferred way of instigating data-parallel activity in Vorlon.

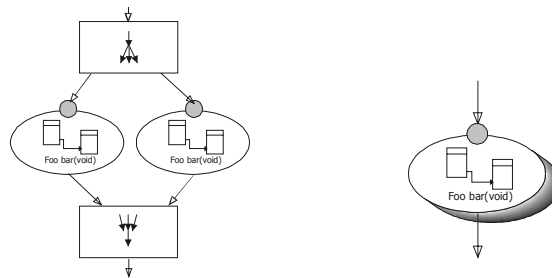


Figure 12. Compile-Time and Run-Time Parallel Method Calls

Expected Results

Since the primary focus of this research has been on visual language development rather than performance optimisation, there is as yet little evidence as to the true performance of a Vorlon application used in anger. Experiments using Vorlon have been conducted and obtained speedups for certain classes of applications. Vorlon has performed best when used to develop its own (parallel) compilation system. Future tests for Vorlon will assess higher-level concerns such as how well the object-oriented paradigm fits into the development of parallel applications. While there is a feeling of optimism about the outcomes of these experiments, the true worth of Vorlon will not be known for some months.

5. Conclusions

Visual parallel programming seeks to provide new programming tools and languages for the community of scientific and engineering users who already possess the required level of tool support for creating specialised parallel applications. It is becoming clear that parallel computation may be one way of providing increased computational power, not just to high-end users, but to traditionally low-end users of productivity applications and computer entertainment. The tools for developing such applications have not been widely considered, and if left in such a state may ultimately cause difficulties as Moore's Second Law impacts computing evolution.

The Vorlon programming language is a prototype visual, object-oriented language for developing parallel applications. It benefits both from a powerful paradigm for managing complexity in the form of object-orientation, and from a visual syntax which can be used to abstract the complexity of parallel control-flows within an application.

Although Vorlon and its associated run-time system are prototypes, the results gained so far have been encouraging, and the intention now is to perform future research into its performance and utility.

The long-term vision for Vorlon is an object-oriented software development environment in which a software engineer can use (visual) notations to design, implement, and debug the performance of a parallel, object-oriented software system. The parallelism will be automatically extracted from the notations used, and the engineer will not be concerned with the detail of specifying and controlling parallelism, nor with tailoring software to run on a particular parallel system.

Furthermore, the developer will be able to link the software with large, object-oriented databases such as the Polar database [23], which can "feed" objects into the software system for processing. Though the current work lags behind that vision further research is continuing to address these issues.

6. References

1. Amdahl, G., *The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. AFIPS, 1967.
2. Moore, G.E., *Cramming More Components Onto Integrated Circuits*, in *Electronics Magazine*. 1965. p. 114-117.

3. Moore, G.E., *An Update on Moore's Law*, . 1997, Intel Developer Forum Keynote: San Francisco.
4. Jelly, I., I. Gorton, and P. Croll, *Software Engineering for Parallel and Distributed Systems*. 1996: Chapman and Hall.
5. Koelbel, C.H., *et al.*, *The High Performance Fortran Handbook*. 1994: MIT Press.
6. Geist, A., *et al.*, *PVM 3 User's Guide and Reference Manual*. 1994: Oak Ridge National Laboratory.
7. Snir, M., *et al.*, *MPI: The Complete Reference*. 1996: MIT Press.
8. Allen, R.S., *A Graphical System for Parallel Software Development*, Ph.D. Thesis, *Department of Computing Science*. 1998, The University of Newcastle upon Tyne.
9. Browne, J.C., *et al.*, *Visual Programming and Debugging for Parallel Computing*. IEEE Parallel and Distributed Technology, 1995(Spring 1995)
10. Cox, P.T., F.R. Giles, and T. Pietrzykowski, *Prograph*, in *Visual Object-Oriented Programming: Concepts and Environments*, M.M. Burnett, A. Goldberg, and T.G. Lewis, Editors. 1995, Prentice Hall. p. 45-66.
11. Dogru, S., *et al.*, *A Graphical Data Flow Language for Retrieval, Analysis, and Visualisation of a Scientific Database*. *Journal of Visual Languages and Computing*, 1996(7): p. 247-265.
12. Kacsuk, P., *et al.*, *A Graphical Development and Debugging Environment for Parallel Programs*. *Parallel Computing Journal*, 1997. **22**(13): p. 1747-1770.
13. Kacsuk, P., T. Dozsa, and G. Fadgyas. *A Graphical Environment for Message-Passing Programs*. *2nd International Workshop for Parallel and Distributed Systems*. 1997. Boston.
14. Newton, P. and J. Dongarra, *Overview of VPE: A Visual Environment for Message-Passing Parallel Programming*, 1994, The University of Tennessee, Knoxville.
15. Newton, P.W., *A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation*, Department of Computer Sciences, 1993, The University of Texas at Austin: Austin. p. 192.
16. Burnett, M.M., *et al.*, *Visual Object-Oriented Programming*. 1995: Manning.
17. Agha, G. and C. Hewitt, *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Editors. 1987, MIT Press. p. 49-74.

18. Lavender, R.G. and D.C. Schmidt, *Active Object: An Object Behavioural Pattern for Concurrent Programming*, in *Pattern Languages of Program Design 2*, J. Vlissides, J. Coplien, and N. Kerth, Editors. 1996, Addison-Wesley.
19. Carriero, N. and D. Gelernter, *How to Write Parallel Programs: A Guide for the Perplexed*. ACM Computing Surveys, 1989. **21**(3): p. 322-357.
20. Watson, P. and S. Parastatidis. *An Optimised Lazy Task Creation Technique for Iterative and Recursive Computations. International Conference on Parallel and Distributed Processing Techniques and Applications*. 1999. Las Vegas, Nevada, USA.
21. Watson, P. and S. Parastatidis. *The NIP Parallel Object-Oriented Computational Model. Network-based Parallel Computing: Communication, Architecture, and Applications*. 1999. Orlando, Florida, USA: Springer-Verlag.
22. Beguelin, A., et al., *HeNCE: A User's Guide*. 1994. <http://www.netlib.org/hence/>
23. Watson, P. *The Design of an ODMG Compatible Parallel Object Database Server. VecPar'98- Third International Meeting on Vector and Parallel Processing*. 1998. Porto, Portugal: Springer-Verlag.

7. Acknowledgements

With grateful thanks to the anonymous referees of this paper, and colleagues of the parallelism research group of the University of Newcastle upon Tyne.