# Adding Parallelism to Visual Data Flow Programs

Philip Cox
pcox@cs.dal.ca

Simon Gauvin
gauvins@cs.dal.ca

Andrew Rau-Chaplin
arc@cs.dal.ca

Faculty of Computer Science,
Dalhousie University
Halifax, Nova Scotia, Canada

## ABSTRACT

*Programming in parallel is an error-prone and complex task compounded by the lack of tool support for both programming and debugging. Recent advances in compiler-directed shared memory APIs, such as OpenMP, have made shared-memory parallelism more widely accessible for users of traditional procedural languages: however, the mechanisms provided are difficult to use and error-prone. This paper examines the use of visual notations for data flow programming to enable the creation of shared memory parallel programs. We present a model, arising from research on the ReactoGraph visual programming language, that allows code in a general class of visual data flow languages to be parallelized using visual annotations, and discuss the advantages this has over current textual methods.*

## Keywords

Data Flow, Parallel, Visual Language

## 1. INTRODUCTION

During the 70s and 80s, several groups of researchers pursued the goal of building highly parallel computers based on data flow [19]. Although differing in detail, these machines shared a common architectural concept, a directed graph with edges, implemented by a configurable communication network, routing data between nodes consisting of simple independent processors. They also shared data-driven data flow semantics, whereby a node would execute after receiving input data. Some of these projects are still continuing: none, however, has produced hardware that is used commercially, or by researchers who use high-performance computing platforms.

Despite its lack of success as a hardware architecture, data flow has survived as a software model, providing the basis for various visual programming languages. Research into visual data flow languages has a long history, and has resulted in various research systems such as Fabrik [3], BDL [18] and Vista [8]. In particular, the project from which the results reported here have arisen involves the design of a message-passing visual data flow language, ReactoGraph [9, 10]. Commercial visual, data flow programming products include LabVIEW [13], VEE [12], Simulink [5], Softwire [21], and Prograph [17]. Data gathered from LabVIEW users, indicates that they prefer visual, data flow languages to textual, procedural ones, and that the usability of the LabVIEW package is to a large extent attributable to the visual data flow nature of the language [1, 22].

After much experimentation, two dominant parallel architectures have emerged, distributed-memory multicomputers as typified by clusters, and shared-memory multiprocessors as typified by the bus-based SMP machines [7]. In fact, today most distributed-memory machines are actually made up of nodes which are themselves SMP machines. Typical corporate servers are four- or eight-way SMP's, and even today's personal computers are often SMP machines, in which two or more processors share access to a global shared memory via a bus.

As parallel machines have developed, the problem of how to program them has been intensively studied [20]. For distributed memory machines, message passing languages or libraries (e.g. PVM or MPI) have been developed. However, this approach often requires existing programs to be mostly re-written with parallelism in mind, if they are to be efficient and scalable. Message-passing systems like MPI are the "assembly language programming" of parallel computing in that the programmer has complete control but must address all technical aspects of the parallelism such as thread management, data dependencies, and synchronization.

In the context of shared-memory or SMP machines, much of the research effort has focused on identifying the potential parallelism inherent in program semantics written in standard procedural languages such as C or Fortran, and rewriting or annotating programs so that they can be deployed on parallel hardware. As a result, various libraries have been developed, providing the programmer with tools to manage the parallel execution of threads [7, 15]. However, these tools are still primitive and complex making their use time consuming and programs constructed with them difficult to understand.

At a slightly higher level of abstraction lies OpenMP, which presents a parallel computation model, realized as a collection of directives, variables, and functions, with which a programmer can annotate a program to run on shared-memory machines [2]. OpenMP is conceptually language independent, and has implementations specific to various standard procedural languages [16], but none to support

visual programming languages. To parallelise an existing sequential program using OpenMP, a programmer inserts compiler directives that identify program blocks as parallel regions. The code in a parallel region is to be executed on some set of threads, as determined by parameters to the parallel directive, and by other related directives. Thus, OpenMP provides a higher level of abstraction than message passing libraries such as MPI; however, it still requires the programmer to pay close attention to important details such as data dependencies and race conditions.

One of the important philosophies behind the design of OpenMP is that one should be able to parallelise existing sequential programs simply by inserting OpenMP directives, thereby producing code that can be compiled both by a normal sequential compiler ignoring the directives, and an OpenMP-compliant parallel compiler. In practice, however, the code usually requires some changes and, to achieve maximum performance, may require major rewriting. It is important to note that at least some of these changes are necessitated by the fact that OpenMP is designed for use with procedural languages. Some of the more subtle and difficult features of OpenMP are present for the same reason.

Although data flow has been used as the basis for parallel architectures, as discussed above, little research has been aimed at realizing the parallelism depicted in visual data flow programs. A simple visual data flow language was implemented for the DDM-1 data flow machine [6]. A simple, conservative system of annotations and a related parallel computation model for Prograph was proposed in [4].

In the following, we present the Visual Parallel Regions (VPR) model, a system of annotations for denoting parallelism applicable to the class of languages of which Prograph, LabVIEW, and ReactoGraph are representatives. These are languages where each wire in an execution instance of a diagram can carry at most one token, and data flow diagrams are contained in control structures which manage iteration and conditional execution. Our annotations are inspired by OpenMP, but do not follow it in precise detail since many OpenMP features are a consequence of the procedural languages to which it applies, and are irrelevant or unnecessary in a data flow context. Nevertheless, we adopt the OpenMP philosophy that the annotations for parallelism should be an overlay that can be applied to an existing data flow program.

Parallel programming is practised primarily by a community of users who are interested in achieving maximum possible performance for well structured code for numerical computations written in Fortran, C, or C++. Although there has been some research into the parallelisation of visual data flow programs, for example [4, 14], there are no such tools available in existing language implementations. Data flow languages such as LabVIEW and Prograph, were designed for users who are attracted to the elegance, ease of use, and speed of application development that visual programming provides. Up to now, however, they have been unable to scale their applications to the new SMP commodity hardware now readily available on many desktops. LabVIEW is widely used by engineers and researchers for simulation, control and large numerical computations, and provides arrays and for loops, ideal candidates for parallelisation. Clearly parallelisation of LabVIEW programs would enable computation on larger data sets on SMP PCs, for example. Similarly, ReactoGraph is being designed for development of web-based enterprise applications on complex platforms such as Web services [23] connecting legacy systems; so parallel annotations for ReactoGraph provide a means for enterprise developers to better access the parallelism in SMP systems now common in most enterprise web system installations.

This work is novel in that it extends a family of visual data flow languages by providing parallel semantics in a visually integrated and non-obstructive way requiring little or no modification to the language syntax yet providing control of parallel execution beyond that inherent in data flow graphs.

# 2. COMPARING PARALLELIZATION OF COMMON TASKS

In the following sections we compare some simple examples of parallel algorithms written in textual programming languages with their equivalent visual data flow code annotated with VPR. The sample programs are small and meant to illustrate common real-world tasks in parallelizing sequential code and to show how VPR can simplify the creation of parallel code, and enhance its comprehensibility.

We present an informal explanation of parallel data flow execution followed by three examples that illustrate various capabilities and facets of VPR increasing in complexity from simple to more involved. The examples cover parallelizing loops, managing data dependencies in arrays, and parallelizing recursion and the accumulation of values across multiple threads respectively.

## 2.1 Parallel Data Flow Execution

In the class of languages that we are concerned with here a *data flow graph* is an acyclic directed graph, the edges and nodes of which are called *data links* and *operations* respectively. Each operation has zero or more *terminals* (inputs) and *roots* (outputs). A data link out of an operation is incident on a root of the operation. A data link into an operation is incident on a terminal of the operation. Each terminal has at most one associated data link while roots may have one or more incident data links. Data links transmit data between operations. An operation uses the values arriving at its terminals to compute values which it places on its roots. Each link has at most one value at any time. An operation may be a primitive, such as an arithmetic or I/O function, or a *call* to another data flow graph.

The execution of a data flow graph is determined by the flow of data through its links. An operation is *ready* to execute when data is present at each of its terminals. On execution, an operation *consumes* the data on its terminals, preventing its reuse. Once execution of the operation is complete, its output values are placed on its roots. If a root is at the tail of more than one link, each link transmits a copy of the value on the root. Each link corresponds to a local variable in a procedural language, but unlike variables in procedural languages, can be assigned a value only once during the execution of a data flow graph call. This property of data flow graphs is called *single assignment*.

Sequential execution of a data flow graph is achieved by executing operators that are ready, one at a time. Note that more than one operator may be in the ready state at any point, so there may be many different execution orderings of a data flow graph on any given input; however, the single assignment property ensures that all executions will produce

the same output. Some data flow languages, such as ReactoGraph, Prograph and LabVIEW, provide other kinds of variables that can be assigned values more than once, allowing side effects not admitted by the "pure" data flow discussed so far. To control these effects, such languages provide synchronization constructs for imposing an order on the execution of operations other than that implied by data flow.

In the case where several operations are ready for execution at the same time, we have an opportunity to execute them simultaneously, leading to what we call *implicit parallelism* inherent in the structure of the data flow graph. Most implementations of data flow that have included parallel execution have concentrated on this implicit parallelism; for example, data flow hardware [19], and software [20].

In the work reported here, we have taken a somewhat different approach to parallelizing data flow programs by allowing arbitrary subgraphs of a graph to be executed in parallel. Note that these subgraphs do not necessarily depend on one another for data. In this approach, which we call *explicit parallelism*, a programmer specifies that copies of some subgraph are to be executed in parallel by some set of threads. This approach requires no modification to the original subgraph but provides a set of explicit annotations, using visual rectangular regions, that are overlaid on the subgraph to define its parallel execution. Although "readiness" is not the criterion for scheduling a subgraph for parallel execution in a region, any operation within a region must be in the ready state before being executed.

## 2.2 Parallel Loops: Computing $\pi$

A common real-world source of parallelism in sequential programs is the parallelization of loops, in which the iterations of a loop are divided between a number of threads [11]. Figure 1 presents a program that calculates $\pi$ using C and the OpenMP API and illustrates the common use of `for()` loops for counted iteration over a set of values or data structures. Such loops are a common source of significant computational work and are prime candidates for parallelism. In principle, a counted loop of fixed length can be easily subdivided among a set of threads; however, care must be taken to identify cases where data within the loop are shared so as not to cause race conditions or corruption of data due to data dependencies. In this example we are interested in showing how OpenMP is used to parallelize such loops and how our model is able to perform the same function with less cognitive work on the part of the programmer.

The example in Figure 1 shows the sequential program [1] computing $\pi$ augmented with four additional Lines, 1, 4, 8, and 9, which transform it into a parallel program through the use of OpenMP directives. In Line 1 the programmer indicates that the OpenMP runtime library will be used. Lines 4 and 8 define a constant, `NUM_THREADS`, used to set the number of threads, in this case 2, to be used in parallelizing the loop. Finally on Line 9 the *parallel for* directive defines the following `for()` loop as a parallel region. The next line after this directive must be a `for()` loop construct otherwise the OpenMP compiler will generate an error. In this case the *parallel for* directive is modified by the addition of two clauses. The first is a data-sharing *private* clause that instructs the compiler to treat the variable `x` as a private value

[1]In the code examples we denote changes to the sequential code with an asterisk next to the line number.

```
1* #include <omp.h>
2  static long num_steps = 10000;
3  double step;
4* #define NUM_TREADS 2
5  void main() {
6    int i; double x, pi, sum = 0.0;
7    step = 1.0/(double)num_steps;
8*   omp_set_num_threads(NUM_THREADS);
9* #pragma omp parallel for private(x)
       reduction(+:sum)
10    for (i=1; i <= num_steps; i++) {
11       x = (i-0.5)*step;
12       sum = sum + 4.0/(1.0+x*x);
      }
13    pi = step * sum;
   }
```

**Figure 1: Parallel $\pi$ program in C using the OpenMP API on the starred lines.**

that is independently allocated in each thread. Note that `x` is not given an initial value from the *master thread* (used to run the program sequentially), meaning that `x` must be initialized within the loop before it can be used. In this case this is done in Line 11. The second clause is a *reduction* clause that instructs the compiler to treat the variable `sum` as if it were a private value, to initialize each thread's copy based on the master thread's value (0.0), and to return to the master thread a value computed by applying the binary associative operator `+` to all of the threads' private values for `sum`.

The presence of the `parallel for` directive results in a program that runs sequentially until Line 9, then forks into two threads with the `for()` loop spread between them, iterations 1 to 5000 running in one thread and iterations 5001 to 10000 in the other. By default, variables in OpenMP are shared between threads. Without the `private(x)` clause, `x` would be shared by default within the parallel region, and would be the source of a race condition, where each thread would compete to write a value to it in Line 11. As well, since its value is dependent on the iteration variable `i` , the value of `x` set by one thread would be out of the loop range in the other thread. Finally, since `x` is shared across Lines 11 and 12, there is a data dependency between these lines, where `x` could change between the write in Line 11 and the read in Line 12. The solution is to make variable `x` private for each thread, using the `private(x)` clause in Line 9. Because the value of `sum` is dependent on all iterations of the loop, it must be reduced at the end of the loop. Since the variable `step` is only ever read in the loop it can be shared. Notice that the iteration variable `i` is not declared private, although it behaves as private, since the `parallel for` directive automatically makes any such counting variables private by default.

From a programming and cognitive work point of view all these data dependency issues need to be considered by the programmer when evaluating any variables used in a loop to be parallelized. Although in OpenMP it is easy to identify a `for` loop and wrap it in a `parallel for` directive, the challenge is in discerning and addressing all of the data dependency issues caused by the directive and its resulting parallel execution of the loop. As we will see, VPR takes

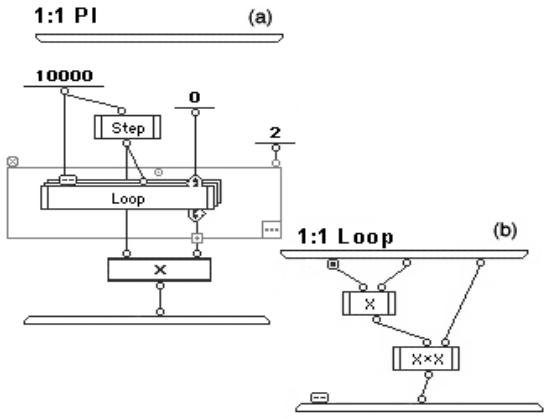



**Figure 2: Parallel data flow $\pi$ program.**

advantage of the fact that data dependencies are explicitly represented in visual data flow programs to significantly alleviate this cognitive overhead.

Figure 2 shows the data flow code corresponding to the C program in Figure 1. Figure 2(a) is approximately equivalent to the procedure `main()` in lines 5 to 13 of Figure 1, while Figure 2(b) depicts the body of the Loop operation and corresponds to lines 10 to 13. Each is an example of a case, consisting of a data flow graph between an input bar at the top and an output bar at the bottom, which respectively transmit data into and out of the diagram.

The Loop operation is a local operation which has been transformed into a counted loop by applying the *n-loop* annotation to one of its terminals, indicating that the terminal must receive a nonnegative integer at execution. On the inside of the local, a root (▣) and terminal (▭), on the input and output bar respectively, correspond to the n-loop terminal on the outside of the local. The root on the input bar provides the current iteration number, starting at 1 and counting to `N` by 1. The terminal on the output bar allows the bounding value of the loop, 10000 in this example, to be changed within the local. The second input to Loop is the value calculated by the local Step corresponding to Line 7 in Figure 1. The locals inside Loop correspond to Lines 11 and 12 in Figure 1. The internals of these locals and Step are trivial and are omitted for clarity.

Ignoring the rectangle enclosing the local Loop, the program in Figure 2 will execute in the usual sequential fashion where the execution order is a total order of the data flow graph. The rectangle is an example of a *parallel region annotation* which instructs the execution mechanism to execute the enclosed code in parallel. In our example, the annotation is in fact a *loop annotation* (see Section 3.1), indicated by the icon in its lower right corner, that parallelizes local operations with either n-loop or list terminals; in other words, counted loops. Parallel annotations have terminals, denoted in the same way as operations, by small icons, circles and squares. The loop annotation has two terminals. The terminal on the right, the *thread terminal*, requires an integer value, which specifies the maximum number of threads to be used in executing the enclosed subgraph of code. The terminal on the left is the *block terminal*, requiring a boolean input used for switching between parallel and sequential execution. If no values are provided on these two terminals, execution defaults to sequential. The root on the inside of the annotation is the *thread id* root which outputs an integer value that uniquely identifies the thread currently executing the enclosed code. All annotation types are identified by unique icons (e.g. loop annotation is ⊡) in the lower right corner. In the example shown in Figure 2, the local Loop will be executed in parallel using 2 threads, each of which will execute half the iterations, as described in our commentary on the code in Figure 1.

This example highlights some of the data sharing properties of VPR, which derive from considering the ways that data links interact with parallel annotations. A link emanates from a root and ends at a terminal, and depending on whether one or both of the root and terminal are inside a parallel region, the link may lie within the region, enter the region, or exit the region. In the C version of the $\pi$ program, the use of an OpenMP private clause, discussed above, is necessary because some procedural languages, C for example, require declarations to precede executable code; hence every variable occurring in a parallel region must be declared outside that region, regardless of whether or not it is intended only for local use. In a data flow graph, however, each root is analogous to a distinct variable that is simultaneously defined and assigned in a procedural program. In VPR, therefore, all links contained within a parallel region are of *private* scope by default (see Section 3.2).

In procedural languages an uninitialized variable can be read. In data flow, however, the order of execution ensures that a link has a value before it is read. Hence, in VPR a link that enters a parallel region is by default *firstprivate*, as defined by OpenMP, meaning that it is private but initialized with the value computed in the master thread for the root, which lies outside the parallel region.

A link that leaves a parallel region is by default *lastprivate* in scope, meaning that value provided by the last returning thread is passed back to the master thread. In our example, however, we require more than just lastprivate, since each thread produces a value from just part of the whole iteration. These individual thread outputs must be added together to produce the required value. This is accomplished by the *reduction annotation* (Section 3.2) located at the intersection of the border of the loop annotation and the link transmitting the output the local Loop. The + sign located in the reduction annotation icon denotes that the binary associative operator + is applied to compute the sum of the values produced by the threads. The link leaving the reduction annotation carries the result of this reduction.

## 2.3 Data Dependencies and Array Processing

In our second example we illustrate the common problem in procedural code with multiple assignment semantics and the data dependencies between array locations. These dependencies occur when a single memory location is read and written by more than one process. In parallel code we typically find three types of dependencies that need to be identified and corrected: 1) *flow*, 2) *anti*, and 3) *output* dependencies [2].

In this example we focus on anti dependencies. As we will see, solving array-related dependency problems in textual code, by using parallel programming models such as OpenMP directives, often requires significant re-structuring of the sequential program, while our data flow approach sup-

```
1 do i = 1, N - 1
2   x = (B(i) + C(i))/2
3   A(i) = A(i + 1) + x
4 enddo
```

(a) Sequential data dependency

```
1* #omp parallel do shared(A, A2)
2  do i = 1, N - 1
3    A2(i) = A(i + 1)
4  enddo
5* #omp parallel do shared(A, A2) private(x)
6  do i = 1, N - 1
7     x = (B(i) + C(i))/2
8     A(i) = A2(i) + x
9  enddo
```

(b) Parallel OpenMP array processing code

**Figure 3: Data dependencies.**

ports parallelization via only minor program annotation. In OpenMP the two most difficult issues that a programmer must grapple with are data dependencies and understanding the behavior of sequential code that has been annotated for parallelism. A race condition is where the output of the program becomes dependent on the relative timing of events, particularly thread scheduling. As will be illustrated in the remainder of this section, because data flow is at a higher level of abstraction than say C, and explicitly captures many data dependencies, these issues are more easily addressed in the visual data flow setting.

The code in Figure 3(a) shows a small sequential Fortran program fragment that uses some arrays in a `do` loop to perform a simple calculation. Using OpenMP we can easily parallelize this loop. However, avoiding data dependencies and race conditions presents a significantly more challenging problem.

Figure 3(a) depicts an example of a *non-loop carried anti dependency* on variable `x`. When `x` is written in Line 2 and read in Line 3 a race condition may occur if a thread sets `x` at Line 2, but `x` is reset by another thread executing Line 2 before the initial thread executes Line 3. The solution to this problem is straight-forward: one simply makes `x` a private variable for each thread.

However, Figure 3(a) also contains an example of a *loop-carried anti dependency* in Line 3 on the array `A`. Note that the code writes to `A(i)` after reading from `A(i + 1)`. This can cause a race condition if index `i+1` is written to by another thread causing the two threads to read different values for the same index. The common solution to this problem is to rewrite the code into two independent parallel loops that are executed one after the other, the first of which copies array `A`, and the second of which uses the copy.

Dealing with data dependencies requires changing the organization of this simple example by modifying the original code. First the data dependent part of the computation in Line 3 is separated into two parts using copies of array `A`, as shown in Figure 3(b). The values of the array `A` are first copied in parallel with no dependencies to a new array `A2`. The `#parallel` directive also defines both arrays to be shared so that each thread has access to all elements of the array. This does not cause race conditions since the iterations are divided across threads to prevent index collision of any writes. Next, the first part of the original code in Line 2 is placed in a second parallel loop, starting at Line 5 in Figure 3(b). Here arrays `A` and `A2` are shared, and the loop-carried anti dependency is removed since we only write to `A` and read from `A2`. No reads on `A2(i+1)` occur while `A2` is being written.

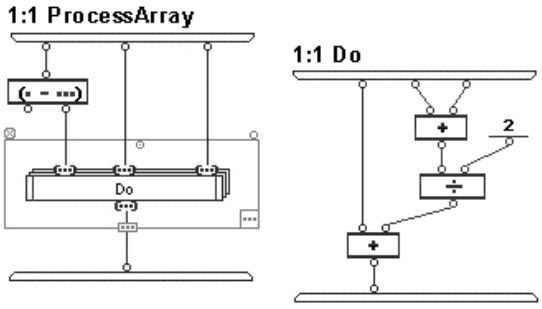



**Figure 4: Parallel data flow dependency solution.**

This example illustrates a common technique for correcting data dependency problems when parallelizing sequential code. This technique yields the program in Figure 3(b) which is semantically identical to Figure 3(a), but with the added benefit of being data dependency free and able to run partially in parallel to improve execution speed. However, the result is an increased use of resources with the addition of the second array `A2`, the need to modify the code to remove dependencies, and more complex code to maintain. With large programs this technique may account for a significant portion of the parallelization effort often requiring entire sections of sequential programs to be re-written in order to run in parallel. Also note that some of the potential parallelism of the OpenMP `do` directive is lost from the initial loop since we now have two parallel loops executing sequentially.

In visual languages of the kind we are considering list processing is indicated by a list interator terminal or root (▪▪▪) on the input or output of an operation repectively. Figure 4 presents a visual data flow equivalent to the textual program in Figure 3(a). In this program the local Do has three list terminals that iterate through corresponding elements of the three input arrays `A`,`B`, and `C`. The leftmost input is processed by the *detach list head* operation (▪ – ▪▪▪) which receives an array as input, and outputs its first element and a copy of the remaining array on its left and right roots respectively. Hence the body of Do is equivalent to Lines 2 and 3 in Figure 3(a). Since the root of Do is also list annotated, it produces a list of the values computed by the individual executions of Do.

As in the example of Section 2.2, the program in Figure 4 is parallelized by enclosing the Do local in a parallel loop annotation. In this case the number of threads has not been specified so the region will be automatically assigned the default maximum number of threads. The total number of iterations, equal to the length of the shortest input list, will be divided among the threads, as before.

The data dependencies in the OpenMP code do not arise in the data flow solution since in the local Do, all links are private by default as discussed above, and the link attached

to the root of Do is lastprivate by default, so each thread creates its own list of results produced by Do. In effect, the creation of the second array required in the OpenMP code is handled automatically by the normal single-assignment semantics of data flow.

It is also important to note that since data links that enter a parallel region are by default firstprivate, lists arriving on these links would normally be copied for each process. However, when a list is routed to a list terminal of an operation in a parallel loop region, it is partitioned into separate lists, one for each of the threads across which the iteration is distributed. This optimization prevents needless copying of large quantities of data. Of course, if a list arrives at such an iterative operation via a simple terminal, then it will be copied like any other firstprivate data.

The example in Figure 4 also features a reduction applied to the output of the Do local. Each thread returns the list resulting from its portion of the computation, and the complete output list is assembled by the list reduction ▭ that concatenates the individual lists.

## 2.4 Parallel Recursion with Quicksort

Our final example illustrates a more advanced work-sharing construct, the *section annotation* (Section 3.1). A section annotation allows the creation of sub-regions of a parallel region, each to be executed by an independent thread. Parallel sections support both the Multiple Program Multiple Data (MPMD) programming style, in which different threads run different code, and Single Program Multiple Data (SPMD) programming, in which different threads run the same code but on different data. They are also very useful in expressing the parallelism inherent in recursive programs, as we will see in this example.

Figure 5 presents a concise version of parallel quick sort written in C using OpenMP. Note that the program is only slightly modified from the original sequential version. The function `quicksort()` starts on Line 3 and receives array `T`, and range values `q` and `r`, and an additional parameter, `deep`, used to control the use of parallelism based on the depth of recursion, to avoid potentially high parallel overheads when dealing with a small input array.

Note that there are two potential sources of parallelism in this example, loop parallelism in the partition step (Lines 9-17), and recursive parallelism associated with the two independent calls to `quicksort()` (Lines 26 and 28). Since we have discussed loop parallelism at some length, we will focus in this example on recursive parallelism. Note, however, that to parallelize the `for()` loop in this case using OpenMP would require significant changes to the code to address data dependency issues.

Lines 21 to 30 thread the two recursive calls in parallel using the OpenMP `sections` work-sharing construct. The use of the *sections* directive in this example shows how two independent recursive calls, normally executed sequentially, can be parallelized by being run in independent threads. The parallel directive on Line 21 begins a parallel region and declares T to be shared. No other variables need to be private since all modifications to the variables occur before the parallel region in the master thread, where those modifications are performed on copies of variables passed as parameters in the function call. Within the `sections` block (Line 23) there are two `section` blocks (Lines 25 and 27). As a result there are only ever two threads spawned at each level of recursion

```
1* #include <omp.h>
2* #define DEEPPAR 3
3  void quicksort(double *T, int q, int r, int deep) {
4     int s; i;
5     double buff, pivot;
6     if (q < r) {
7        pivot = T[q];
8        s = q;
9        for (i = q+1; i <= r; i++) {
10          if (T[i] <= pivot) {
11             s = s+1;
12             buff = T[s];
13             T[s] = T[i];
14             T[i] = buff;
15 }
16       }
17    }
18    buff = T[q];
19    T[q] = T[s];
20    T[s] = buff;
21* #pragma omp parallel if(deep < DEEPPAR) shared(T)
22* {
23* #pragma omp sections
24* {
25* #pragma omp section
26*    quicksort(T, q, s-1, deep+1);
27* #pragma omp section
28*    quicksort(T, q, s+1, deep+1);
29* }
30* }
31  }
```

**Figure 5: Parallel Quicksort in C using OpenMP.**

no matter how many threads may be available for use at Line 21 of the parallel region. As the depth of the recursion reaches a certain threshold, defined by `DEEPAR` in Line 2, the test in the `if()` clause in the pragma at Line 21 switches off parallelism in the region from Line 22 to Line 30, causing the recursive quicksort calls to be executed sequentially.

The equivalent visual data flow version of parallel quicksort, again using lists rather than arrays, is shown in Figure 6. Note that the quicksort algorithm itself does not depend on arrays specifically so our choice of lists allows the data flow code to use the same algorithmic approach but in a way that is more natural in the data flow programming style. Here the identical logic for run-time recursion depth detection is used as well as a similar section annotation. The inputs to Quicksort are a list and an integer used to limit the depth of recursion. This code is a minor variation of the quicksort one would normally construct in a data flow language of this kind. In particular, an extra input has been added to track the depth of recursion, together with a +1 operation to increment it, and an operation to compare it with a limit on depth in the persistent (global variable) DEEPAR.

This example illustrates the use of the *section* annotation in VPR, indicated by the rectangle surrounding the two Quicksort operations in the first case 1:2 Quicksort. This annotation causes each operation in the parallel region to be run independently in its own thread (see Section 3.1). Annotations we have seen in previous examples have been drawn

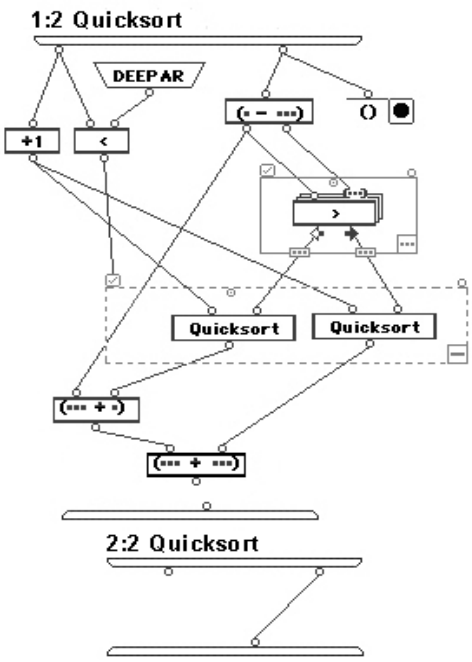



**Figure 6: Parallel visual data flow Quicksort.**

with a solid border indicating a *thread barrier*, which blocks execution of code outside the region until all operations in the region have been executed. In this example, however, the outputs of the recursive Quicksort call operations do not have to be combined in any way, except as indicated by the data flow output links, unlike previous examples where outputs of operations in parallel regions were combined by reduction. Therefore, we have applied *no-wait* (Section 3.3) synchronization to the annotation indicated by its dashed border. This allows operation , which creates a new list by appending an element to the end of the list it receives as input, to be executed as soon as the leftmost Quicksort produces its value, even if the other Quicksort call operation has not finished. This example also illustrates the function of the *block* terminal on a parallel annotation, used to enable or disable parallel execution (Section 3.1).

We also exploit loop parallelism in this example by applying the parallel loop directive to the partition operation $\geq$ which, as discussed in previous examples, divides the iterations of the loop between the available threads. Applying loop parallelism to the C `quicksort for()` loop to achieve the same effect requires far more cognitive work dealing with the resulting data dependencies. The loop concludes with two list reductions that assemble the results of the partitioned list as two lists.

# 3. VISUAL PARALLEL REGIONS

In the previous section, we introduced some of the constructs of VPR via a sequence of examples, assuming a knowledge of the semantics of the kind of visual data flow languages for which VPR is designed. In this section we provide a catalogue of the constructs that VPR provides, and informally describe their functionality. We will explain *parallel regions*, *data flow scoping rules*, *synchronization* and *environment control*.

## 3.1 Annotations for Parallel Regions

To parallelise a data flow graph, the programmer defines subgraphs called *regions*, by drawing rectangular outlines (*annotations*) to enclose operations. An operation can occur in two regions if and only if one of the regions is nested inside the other. A region or operation is said to be *top-level* in a graph $G$ if it is not nested inside another region of $G$. We refer to top-level operations and regions as *units* of $G$.

Each annotation has exactly two terminals, one on the top at the far left, the other at the far right, and exactly one root on the inside of the annotation boundary at the top as shown in Table 3.1. The terminal on the right, the *thread terminal*, requires an integer specifying the maximum number of threads to be used in executing the enclosed code. The terminal on the left is the *block terminal*, and requires a boolean that dictates whether execution of the subgraph in the region should be parallel or not. The root, called the *thread id root*, produces an integer value identifying the thread currently executing the enclosed subgraph. In addition, for each data link connecting a root $r$ of an operation inside the region to a terminal $t$ outside the region, there is a terminal $t'$ on the inside of the annotation border and a matching root $r'$ on the outside, and the link is replaced by two links, from $r$ to $t'$ and from $r'$ to $t$. The terminal $t'$ and root $r'$ are said to be *induced* by the link.

Each annotation has an icon in its bottom right corner identifying its type (ex. loop annotation is ). Table 3.1 depicts the three types of annotations, *parallel*, *section* and *loop*, which are analogous, although not identical, to similarly named constructs in OpenMP.

The notion of data dependency, on which the ordering of execution of operations in a data flow graph is based, is extended to include top-level regions, as follows. If $X$ is a unit and $Y$ is a top-level operation, then $Y$ depends on $X$ iff there is a link from a root of $X$ to a terminal of $Y$. If $X$ is a unit and $Y$ is a top-level region, then $Y$ depends on $X$ iff there is a data link from a root of $X$ to a terminal of $Y$, or to a terminal of some operation in $Y$. Let $G_P$ be the directed graph consisting of the units of G connected by edges implied by this dependency relation. We say that an annotated graph $G$ is *valid* if and only if $G_P$ is acyclic. We can similarly extend the "ready-to-execute" notion for operations, described in Section 2.1, to units. Note that if there is no link into the *thread terminal* of a region, the terminal receives a system-supplied default value. Similarly, the value of an unconnected *block terminal* defaults to *true*.

As the examples in Section 2 illustrate, execution of an annotated graph proceeds sequentially according to the normal execution order dictated by the data flow dependencies in the graph, except where an annotation indicates some form of parallel execution. In general, a graph $G$ is assigned to a thread $P$, called the *master thread* for $G$, which executes the units of $G$ in sequence according to a *schedule* for $G$, consisting of a total ordering of $G_P$. If the first item of the schedule is not ready, $P$ waits. Otherwise, if it is an operation, $P$ executes it in the normal way. If it is a region, then if it has the value *false* on its block terminal, its units are added to the schedule of $P$. Otherwise, suppose its thread terminal has the value $n$, then the following occurs.

If the region is defined by a *parallel* annotation, $n$ copies of the enclosed subgraph are assigned to $n$ threads including $P$, which proceed as described above, in parallel.

If the region is defined by a *section* annotation, the first

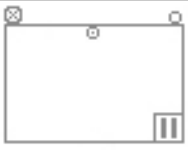
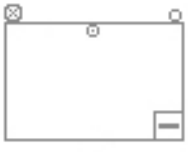
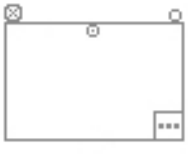
$n$ units of a schedule for the enclosed subgraph are assigned to $n$ threads including $P$, which proceed as described above, in parallel. As soon as one of these threads completes, it is assigned the next unit in the schedule. This continues until all units in the schedule have been assigned to threads.

If the region is defined by a *loop* annotation, each iteration operation in the enclosed subgraph is tagged with a set of $n$ threads including $P$, then a schedule for the subgraph is added to the the beginning of the schedule of $P$. Now during the processing of its schedule, whenever $P$ encounters a tagged iteration, $n$ copies of the loop body are assigned to the $n$ threads in the tag, and each input value is partitioned into $n$ sets, distributed across the $n$ threads, which proceed to execute their portions of the iteration in parallel.

In each of the above cases, as each of the $n$ threads assigned to a region completes its execution, it must wait until all $n$ threads have finished before proceeding to further computation.

This general execution can be modified by various VPR constructs explained below.

**Table 1: Visual Parallel Regions**

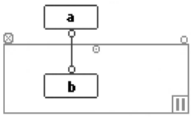
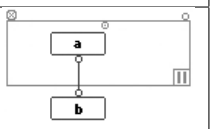
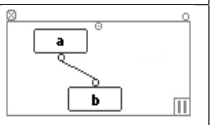
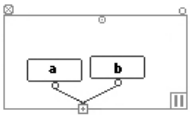
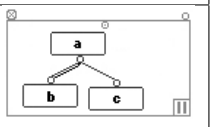
| Region | Representation |
|---|---|
| parallel |  |
| section |  |
| loop |  |

## 3.2 Data Scoping of Parallel Regions

In this section we define scoping rules for data flowing into and out of a region. In procedural languages such as C, a variable can be declared and used in different portions of the code, and can be assigned a value more than once. These properties can lead to data dependency errors when parallelizing sequential textual code as illustrated in Example 3. Scoping rules provided by OpenMP provide some tools for the programmer to address these problems. In contrast, a variable in the data flow languages we are considering is declared, assigned once and used all in the same context, defined by the corresponding root and connected data links. There is less need, therefore, for scoping rules in VPR. However, where links are duplicated as a result of subgraphs being copied across threads, some scoping rules are required to control data sharing and aggregation.

A link emanates from a root and ends at a terminal, and depending on whether one or both of the root and terminal are inside a parallel region, the link may lie within the region, enter the region, or exit the region. Table 3.2 illustrates the possible combinations of data flow code and parallel region with its associated default variable scope. Unlike OpenMP, where various scopes can be declared for a variable, VPR defines a default scope for links and provides a single option for sharing a link between threads, as follows.

**Table 2: Scope Rules for Parallel Regions**

| Scope | Representation |
|---|---|
| first private |   |
| last private |   |
| private |   |
| reduction |   |
| shared |   |

In VPR, a link that enters a region is by default *firstprivate*, that is, private but initialized to the value computed in the master thread for the root, which lies outside the parallel region, for all threads executing in the region. Any link contained entirely within a region is *private* by default, so that its value is unique within each thread executing a copy of the subgraph. A link that leaves a region is by default *lastprivate*, meaning that it is private, but the root induced by the link receives the value provided to the link by the thread which is last to complete execution of a copy of the subgraph.

The default scope for a link contained entirely within a region can be overridden by marking the link *shared*. The value of a shared link is the same for all threads executing a copy of the subgraph, so the value it has at a particular time is the value it most recently received in any of the threads.

In some cases, the various parallel executions resulting from a region may produce different values at the induced terminals; for example, the threads executing portions of an iteration in a loop region. In such situations, the programmer may wish to aggregate these values in some way. To accomplish this in VPR, an induced root can be marked as a *reduction*, indicating that the value of this root is to be computed from the individual values of the corresponding induced terminal. The computation applied can be a primitive ($sum, +, -, /, x, concat$) or a named function. Reduction is illustrated in the preceding examples.

## 3.3 Synchronization

The three types of parallel region defined above do not

provide all the necessary parallelization tools. Finer control of the execution of threads is often required. To address this requirement VPR provides *synchronization regions*. A synchronization region can be placed inside a parallel region to control the parallel execution of the subgraph it encloses. Table 3.2 illustrates the synchronization annotations.

**Table 3: Synchronization of Parallel Regions**

| Region | Representation | Region | Representation |
|---|---|---|---|
| single |  | master |  |
| atomic |  | critical |  |
| no wait |  | barrier |  |

The *single* annotation indicates that the enclosed subgraph is to be executed only once using any of the threads assigned to the enclosing parallel region.

The *master* annotation indicates that the enclosed subgraph is to be executed only once using the master thread of the enclosing region.

The *critical* annotation indicates that although all the threads assigned to the enclosing parallel region must execute the enclosed subgraph, they must do so one at a time.

The *atomic* annotation indicates that once execution of the enclosed subgraph, restricted to contain only primitive operations, has begun in some thread, all other computation on the processor running this thread should be suspended. Correct execution of an atomic region is not guaranteed unless the region contains only primitives supported by the underlying operating system.

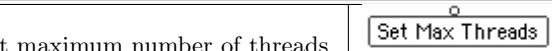
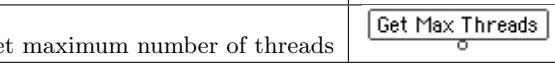
The *no wait* annotation can be added to any of the annotations defined in Section 3.1 to disable the default barrier implicit in these regions. That is, as soon as any thread assigned to the region has completed, it can continue executing rather than wait for all other threads assigned to the region to finish. Care must be taken when using this annotation since the master thread may complete execution of the region and begin to execute operators outside the region before the remaining threads inside the region are complete.

The *barrier* annotation can be added to any of the annotations defined in Section 3.1 to partition the enclosed subgraph into two or more sub regions. It indicates that all threads must finish executing the first region before any thread can begin executing the second, and so forth.

## 3.4 Environment Control

Occasionally, it is necessary for a programmer to control the environment in which a parallel program is executed. The ability to dynamically control the maximum number of threads, or how scheduling is performed, allows a programmer to maximize the performance of an application by tuning the code to run more efficiently on a specific platform. Environment control is provided in VPR via a set of primitive operations that set or get the value of some environmental parameter via a data flow operation. Table 3.4 illustrates one example of a get/set pair of environmental primitive operations. Other primitives are available for manipulating other system-dependent parameters.

**Table 4: Environment Control for Parallel Regions**

| Control | Representation |
|---|---|
| Set maximum number of threads |   |
| Get maximum number of threads |   |

# 4. CONCLUDING REMARKS

In this paper we have presented a model that allows sequential visual data flow code to be parallelized using a visual notation inspired by OpenMP semantics. We have shown how this model can effectively realize the goal of incrementally transforming sequential programs into parallel ones. Through examples, we have explored ways in which the parallelism inherent in existing sequential visual data flow programs can be identified and annotated so that they can be effectively deployed in parallel shared memory settings.

In applying OpenMP-like semantics in the visual data flow context, we have been struck by the significant number of ways in which inherent properties of visual data flow languages aid in making the transformation from sequential to parallel codes easier, more direct, and likely less error prone. These results produce a system of parallel programming that reduces the cognitive load normally associated with text-based languages. In text-based OpenMP it is often easy to specify parallelism, but much harder to create a parallel version of a sequential program that produces correct results. Issues relating to data dependencies, race conditions, and understanding the wider implications of parallel directives, must be expertly finessed by the programmer. When using our visual notation, in many instances, these issues are either less challenging to address or are absent altogether. The explicit visual representation of data flow and the single assignment property of data flow semantics greatly aid in identifying and removing data dependencies and race conditions. In addition, the use of graphics to represent directives provides a concrete representation of the notion of regions enclosing the code to be parallelized, potentially resulting in improved program comprehension.

Our Visual Parallel Regions model provides a level of abstraction equivalent to that provided by OpenMP. We are presently exploring the implementation of VPR in the context of ReactoGraph using OpenMP libaries.

# 5. ACKNOWLEDGMENTS

This work was partially supported by Natural Sciences and Engineering Research Council of Canada Discovery Grants OGP0000124 and OGP170169-04

## 6. REFERENCES


[1] E. Baroth and C. Hartsough. Visual programming in the real world. *Visual Object-Oriented Programming: Concepts and Environments*, pages 21–42, 1995.

[2] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan-Kaufmann, New York, 2000.

[3] Y. Chow, K. Doyle, F. Ludolph, D. Ingalls, and S. Wallace. The Fabrik Programming Environment. *IEEE Workshop on Visual Languages*, pages 222–230, September 1987.

[4] P. T. Cox, H. Glaser, and S. Maclean. A visual development environment for parallel applications. In *Symposium on Visual Languages*, pages 144–151. IEEE, October 1998.

[5] J. B. Dabney and T. L. Harman. *Mastering Simulink 4*. Prentice Hall, New York, 2001.

[6] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, February 1982.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 1998.

[8] J. Frohlich and S. Schiffer. Visual Programming and Software Engineering with Vista. In *Visual Object-Oriented Programming: Concepts and Environments*, pages 21–42. Manning Publications Co., 1995.

[9] S. Gauvin. *ReactoGraph: A visual language for the development of user interfaces, Master's Thesis*. Faculty of Computer Science, Dalhousie University, http://gauvins.cs.dal.ca/reactograph, 2003.

[10] S. Gauvin and T. Smedley. Concrete Programming with Reactive Objects. In *IEEE Symposium on Human-Centric Computing: End-User Programming*. IEEE, September 2002.

[11] A. Grama, V. Kumar, A. Gupta, and G. Karypis. *An Introduction to Parallel Computing: Design and Analysis of Algorithms, 2nd. ed.* Pearson Addison Wesley, New York, 2003.

[12] R. Helsel. *Visual Programming For HP-VEE*. Prentice Hall, New York, 1997.

[13] G. W. Johnson and R. Jennings. *LabVIEW Graphical Programming*. McGraw-Hill, New York, 2001.

[14] B. Lanaspre. *Static Analysis for Distributed Prograph, PhD Thesis*. Dept. of Electronics and Computer Science, University of Southhampton, 1998.

[15] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *USENIX Conference*, pages 29–41, January 1993.

[16] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, Version 2.0*. www.openmp.org, March 2002.

[17] Pictorius Inc. *Prograph CPX Reference Manual*. Pictorius Inc, Halifax, 1996.

[18] A. Schurr. BDL-a nondeterministic data flow programming language with backtracking. *IEEE Symposium on Visual Languages*, pages 391–401, September 1997.

[19] J. A. Sharp. *Data Flow Computing*. Ellis Horwood, 1985.

[20] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[21] Softwire Inc. *http://www.softwire.com*. Softwire Inc., 2004.

[22] K. N. Whitley and A. F. Blackwell. Visual Programming in the Wild: A Survey of LabVIEW Programmers. *Journal of Visual Languages and Computing*, 12(4):435–472, 2001.

[23] World Wide Web Consortium on Web Services Activies. *http://www.w3.org/2002/ws/*. W3C.org, 2002.