The Assign Parallel Program Generator*

David R. O'Hallaron
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213

Abstract

ASSIGN *is a tool for building large–scale applications, in particular signal processing applications, on distributed-memory multicomputers. The first target machine is iWarp, a multicomputer system developed jointly by Intel Corporation and Carnegie Mellon University. This paper gives a high-level introduction to* ASSIGN .

1 Introduction

ASSIGN is a tool for building large–scale applications on distributed–memory multicomputers. The programming model is the coarse-grained *flowgraph*, where nodes correspond to coarse–grained sequential computations and directed edges correspond to FIFO queues[2, 5]. Given a flowgraph, ASSIGN automatically generates a parallel program consisting of a C program for each cell (processor) in the target machine. Details such as partitioning the flowgraph, placing and routing the partitioned flowgraph, creating communication pathways, transferring data from cell to cell, flow control, and balancing workload across processors and communication pathways are all handled automatically.

ASSIGN is intended for applications that can be described with static coarse-grained flowgraphs. In particular, it is designed for large–scale digital signal processing (DSP) applications, which are described very naturally using coarse-grained flowgraphs.

Aside from the obvious merits of automating machine–dependent details, the primary virtues of ASSIGN include the hierarchical nature of its flowgraphs, the ability to parameterize both flowgraphs and individual signal processing operations, the ability to model and compile large graphs with thousands of nodes, a flexible and general–purpose mechanism for transferring data to and from the host, and a minimum of run-time scheduling overhead. The primary limitation is that it is suitable only for those applications that can be modeled as static, coarsed-grained flowgraphs.

The first target machine is the iWarp[12, 13], a multicomputer system developed jointly by the Intel Corporation and Carnegie Mellon University. We are currently using ASSIGN to generate working programs for the first 64-cell iWarp systems delivered by Intel to Carnegie Mellon.

In this paper, we introduce ASSIGN. The emphasis is on techniques for writing ASSIGN programs, rather than on the details of the automatic mapping and code generation algorithms. Section 2 give an overview. Section 3 describes the underlying flowgraph model. Sections 4 and 5 describe the techniques for building Assign programs. Section 6 discusses related work. A more detailed discussion is given in [8].

2 Overview

2.1 Flowgraphs

The flowgraph is the underlying programming model for ASSIGN. Under the guise of various names such as block diagrams[9], large-grained dataflow graphs[2], synchronous dataflow graphs[5], and signal flow graphs[10], flowgraphs have been used for years in the literature and in practice to model DSP applications.

A flowgraph is a collection of *nodes* and directed *edges*. Nodes represent computations and edges represent FIFO queues. Each node iterates an infinite number of times. Each iteration, the node consumes some data items from each of its input edges, performs a computation on the inputs, and produces some data items on each of its output edges. The sizes of the inputs and outputs are indicated by integer edge labels.

Figure 1 shows an example flowgraph for an application that performs the same operations on two independent data streams. The edge labels correspond to the number of data items produced and consumed each time a node is

*The complete version of this paper appears as Carnegie Mellon Technical Report CMU-CS-91-141. The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035, and in part by the Office of Naval Research, Air Force Office of Scientific Research, and Naval Ocean Systems Center under Contract N00014-90-J-1939.

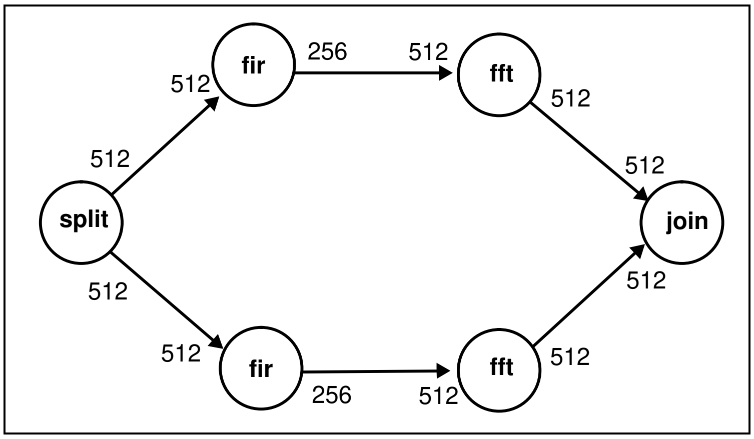

Figure 1: Example flowgraph. Edge labels denote the number of data items produced and consumed each time a node is invoked.

invoked. For example, the node labeled *fir* corresponds to an FIR filter that inputs 512 data items, performs a 2:1 downsampling operation, and outputs the result. The FFT is invoked whenever 512 data points have arrived. Thus, for each invocation of the FFT, there are two invocations of the FIR filter.

ASSIGN flowgraphs are hierarchical. A node in a graph can correspond to a subgraph, so that graphs can be constructed hierarchically out of smaller graphs. This is illustrated in Figure 2. The example is a real graph of a sonar adaptive beam interpolation application implemented on iWarp using Assign. At the highest level, the graph consists of two nodes, where each node corresponds to a subgraph.

2.2 Mapping Algorithm

Assign uses the two-phase mapping algorithm shown in Figure 3. The first phase, partitioning, uses an extension of the chain mapping methods developed in [7] to partition the M nodes of the flowgraph into at most P blocks, where P is the number of the cells in the target machine. Since the throughput of the system is bounded from above by the cell with the maximum load, the optimization goal of this phase is to find a partition that minimizes the maximum load on any particular cell.

There are several interesting aspects of the partitioning algorithm. First, the load model for each cell incorporates communications cost, and thus the optimal partition does not always use all of the cells. Second, although the partitioning problem is intractable in general, the extended chain method we are using is guaranteed to find an optimal partition in polynomial time if the input flowgraph is a chain [7].

The second phase, placement and routing, places the partition blocks from the first phase on the cells of the target array, and routes any exposed edges through the cells of the array. The optimization goal of this phase is to minimize the number of flowgraph edges shared by any particular physical communications link. The placement and routing algorithm is borrowed directly from [6].

The interesting aspect of the placement and routing phase is the simple and beautiful correspondence between exposed graph edges and iWarp pathways. Each pathway is set up once when the program is loaded, and data flow through intermediate cells transparently, with no effect on any computations on the intermediate cells.

2.3 System Overview

ASSIGN is a front-end to the existing tools supplied by the vendor of a parallel machine. For example, an overview of ASSIGN's place in the iWarp tool chain is shown in Figure 4.

Flowgraphs are created by running a C program called a *graph generator*, which uses a library of ASSIGN functions to build and emit a flowgraph on the standard output.

ASSIGN inputs the flowgraph produced by the graph generator along with a collection of *state structures*, which, as we shall see, are arbitrary user–defined C structures that can be used to parameterize an application and hold internal state. On the command line, the user can specify such options as the number of cells to be compiled onto, an optimization level that controls the quality and running time of the mapping algorithm, and the *arbitrary* 2D topology of the target iWarp array.

The output of Assign is a C program that will, at the discretion of the user, be compiled to run on either the Unix workstation or the iWarp array.

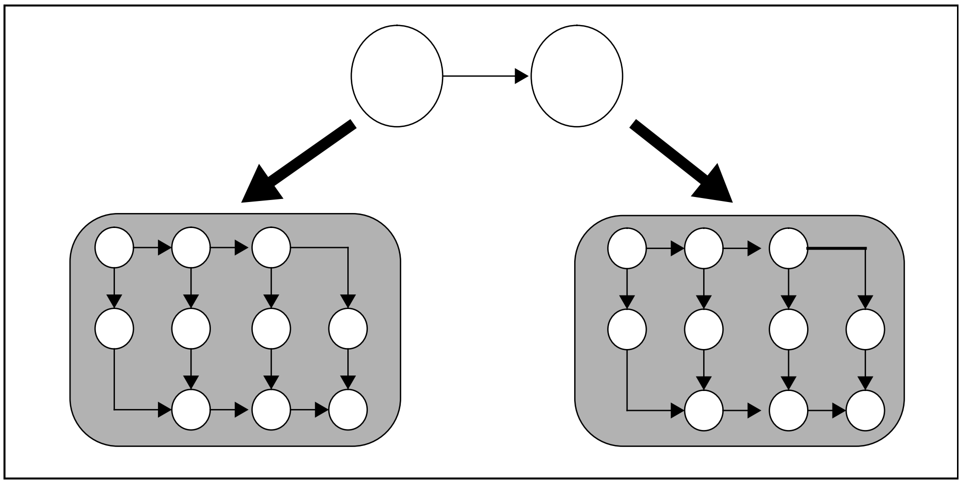

Figure 2: The hierarchical nature of flowgraphs. Here we have a flowgraph model of a real sonar application. The graph consists of two nodes, where each node is itself a subgraph.

2.4 Programming Overview

The process of programming applications using ASSIGN is determined largely by (1) the ability to compile flowgraphs onto either the host Unix workstation or the attached parallel computer, and (2) the hierarchical nature of the flowgraphs. The ability to compile onto the workstation allows us to do most of our testing and debugging on the workstation rather on the parallel machine. The hierarchical nature of flowgraphs allows us to build applications in a sequence of incremental steps (See Figure 5):

Step 1: Write node functions. These are sequential C functions, instances of which will correspond to nodes in the flowgraph. Each of these functions can be tested independently on the Unix workstation.

Step 2: Generate and test subgraphs. ASSIGN flowgraphs are hierarchical, meaning that graphs can be formed by composing smaller subgraphs. Typically, we independently generate and test a collection of smaller subgraphs on the Unix workstation, using the node functions developed in the previous step. As we shall see, flowgraphs are generated by writing and running a C program called a *graph generator*.

Step 3: Combine and test subgraphs. Once we are satisfied that each of the subgraphs is producing correct results, we combine them, *without change*, to form the final graph. The testing of the final graph can be performed entirely on the Unix workstation.

Step 4: Run final graph on parallel machine. When we are satisfied that the final graph is producing correct results, we can recompile the graph, *without change*, and then run it on the parallel machine. It is important to note that most of the program development is done on the Unix workstation and not on the parallel machine.

3 Flowgraphs

A *flowgraph* is a (possibly cyclic) multidigraph consisting of a collection of *nodes* and directed *edges*.[1] Nodes represent computations and edges represent FIFO queues. Each node iterates an infinite number of times. Each iteration, the node consumes some data items from each of its input edges, performs a computation on the inputs, and produces a vector of data items on each of its output edges. The sizes of the inputs and outputs are indicated by integer edge labels. A flowgraph has exactly one *source node* (i.e., a node with no input edges) and exactly one *sink node* (i.e., a node with no output edges).

Nodes. A node is either an *atom* or a *subgraph node*. An atom corresponds to a coarse–grained sequential task such as an FFT or FIR filter. A subgraph node corresponds to a flowgraph. Thus, flowgraphs are hierarchical entities that can be constructed out of instances of smaller flowgraphs.

Ports. Each node has a (possibly empty) list of *input ports* and a (possibly empty) list of *output ports*. Associated

[1] A multidigraph is a directed graph where multiple edges are allowed between the same pair of nodes.

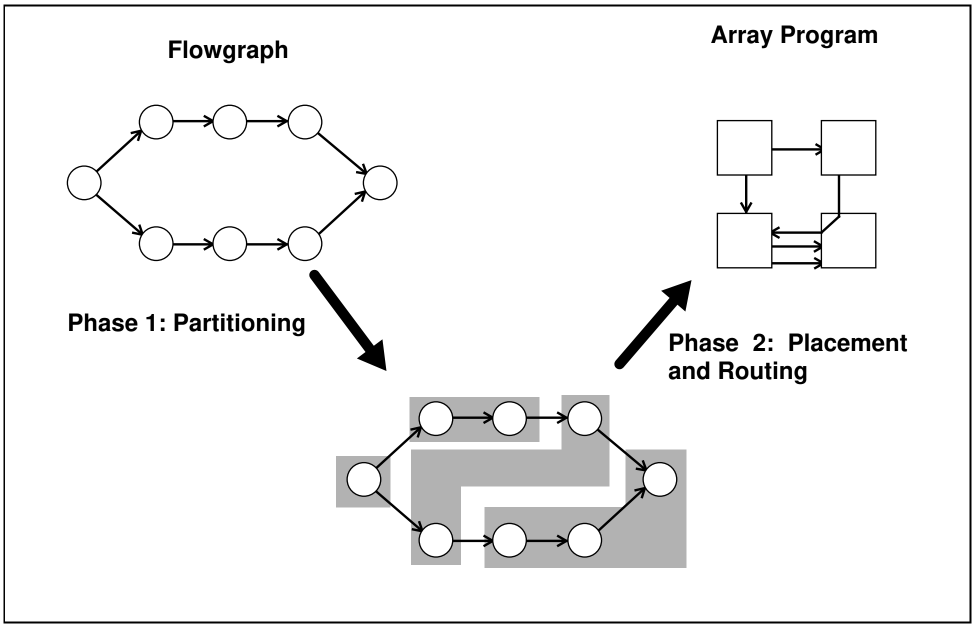

Figure 3: The Assign mapping algorithm. The partitioning phase attempts to minimize the maximum load on any particular cell. The placement and routing phase attempts to minimize the number of flowgraph edges shared by any particular physical communication link.

with each input port k is a name, a scalar type (int, float, double, or complex), and an integer *consume amount*, c_k. Associated with each output port k is a name, a type and an integer *produce amount*, p_k. We will explain the meaning of the consume and produce amounts later in this section when we discuss node execution rules.

Node Functions. Each node in a graph is associated with a collection of *node functions*, which are C functions that control the computation performed by a node. There are three types of node functions: *primitives*, *dynamic initializer functions*, and *input/output functions*.

Primitives. Each atom must be associated with a *primitive*. A primitive is a user-written C function that follows ASSIGN's calling convention. The argument list for a primitive consists of a (possibly empty) list of pointers to input vectors, followed by a (possibly empty) list of pointers to output vectors, followed by an optional pointer to a state structure. Typical primitives are FFT's and FIR filters.

Input and Output Functions. An atomic source node must have an associated *input function*, which is a user–written C function that returns an input vector of length c_k for each inport port k. Similarly, an atomic sink node must have an associated *output function* that outputs a vector of length p_k for each output port k. The input and output functions provide a general-purpose mechanism for handling data transfers between a host computer and a graph. ASSIGN makes no assumptions about the format of the input and output data on the host.

State Structures and Their Initializers. An atom may have internal state in the form of an optional *state structure*, which is an arbitrary C structure. A pointer to the state structure is passed to a primitive each time it is invoked. The primitive can freely inspect and modify the state structure. State structures can be initialized at compile time with a *static initializer* and/or at run time with a *dynamic initializer*. A static initializer is an arbitrary C structure initializer list. A dynamic initializer is an arbitrary C function that is called once at run time before the primitive is invoked.

State structures and their initializers provide a general–purpose mechanism for passing arbitrary parameters to primitives. Examples of state structure elements include the sizes of the input and output vectors, precomputed coefficients that are a function of the input size (e.g, FFT twiddle factors), and dynamic internal state that must be retained from invocation to invocation, (e.g., previous outputs of an IIR filter).

Execution Weight. Each atom has an *execution weight*, which is the estimated number of cycles consumed by one

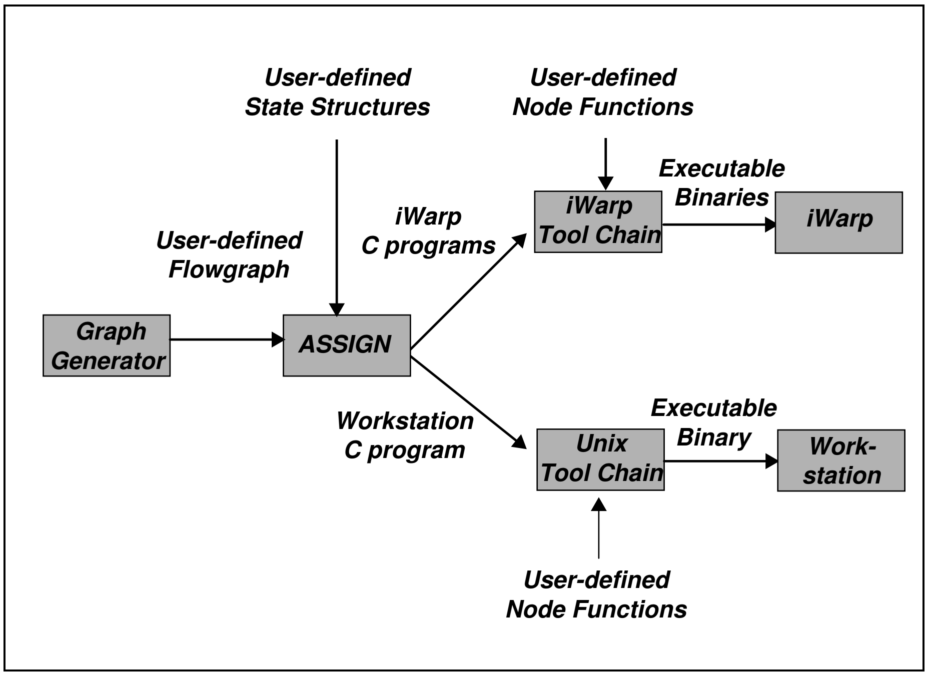

Figure 4: Assign system overview. ASSIGN is a front-end to existing tools. Programs can be compiled onto a parallel computer or a Unix host.

invocation of its primitive. The execution weight does not affect the semantics of the flowgraph. Rather, it is a hint supplied by the user to help ASSIGN balance the workload across the cells of the iWarp array.

Edges. An edge binds an output port on one node to the input port of another node. Each edge corresponds to a FIFO queue with exactly one writer and one reader.

Graph Execution Rules. Each atom executes according to the following rules: Initially, the atom calls an optional dynamic initializer. This is followed by a (possibly infinite) sequence of the following steps:

1. Remove c_k data items from each input port k. If this is a source node, the data for each input port is received via an invocation of the input function. Otherwise, the data for each port k comes from input queue k.

2. Invoke the associated primitive using as inputs the c_k data items consumed from each input port k.

3. Write p_k data items to each output port k. If this is a sink node, the data written to the port is passed to the output function, which sends the data to the host computer. Otherwise, data written to the output port k is deposited in output queue k.

Each atom executes independently according to the atom execution rules; each graph node executes the corresponding subgraph, with the input edges of the graph node bound to the input ports of the subgraph's source node and the output edges of the node bound to the output ports of the subgraph's sink node.

The flowgraph model we have described in this section is equivalent to the synchronous dataflow model of Lee and Messerschmitt [5]. It is sufficiently powerful to model real applications with multiple sampling rates and redundant computations[10].

4 Writing Node Functions

Typically, the first step in building an ASSIGN application is writing the appropriate node functions. These functions (primitives, dynamic initializers, input/output functions) can be placed in a single file or separate files according to taste. In this section, we describe techniques for writing node functions.

4.1 Primitives

The simplest ASSIGN primitive has no input or output arguments.

```
hello()
```

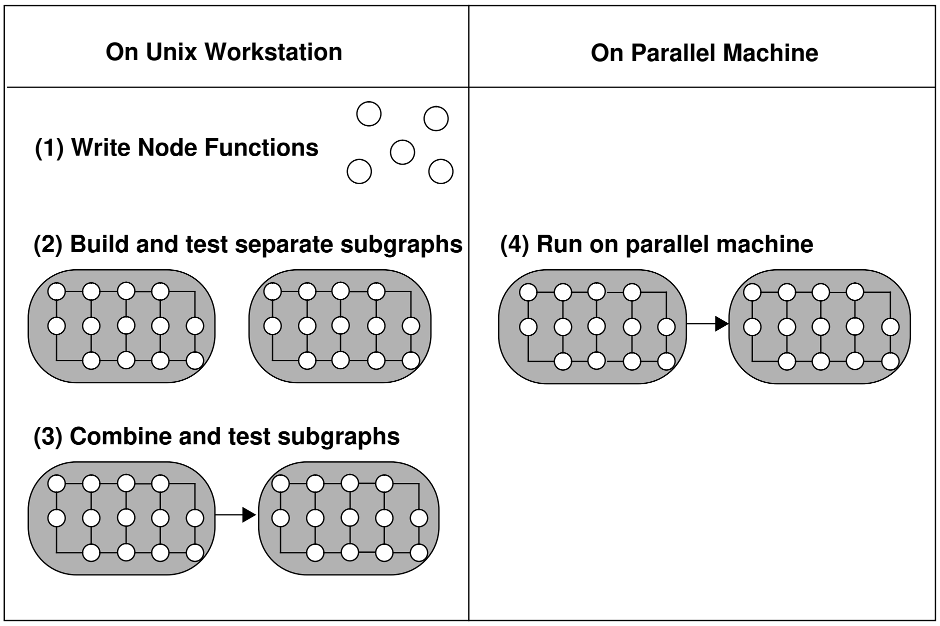

Figure 5: Typical process of programming Assign applications. Notice that the system is constructed incrementally and that most of the work is done on the Unix workstation.

```
{
    printf(‘‘hello, world\n”);
}
```

A somewhat more useful primitive inputs a vector of 16 `floats`, increments each element of the vector, and places the result in an output vector.

```
incr16(inp, outp)
float *inp, *outp;
{
    int i;

    for (i=0; i<16; i++)
        *outp++ = (*inp++) + 1;
}
```

Notice that the argument list is a list of *input arguments* followed by a list of *output arguments*. Each input and output argument is a pointer to a vector of either `int`, `float`, `double`, `complex`, `complex_int`, or `complex_double`, where the complex types are defined by

```
typedef struct {float r, i} complex;
typedef struct {int r, i} complex_int;
typedef struct {double r, i} complex_double;
```

4.2 State Structures

Of course, a primitive like `incr16` would be a lot more useful if it worked on vectors of any size. Fortunately, ASSIGN provides a mechanism called the *state structure* for writing such general–purpose primitives. To use a state structure in a primitive, we define the structure in an include file

```
struct incr_state {
    int size;
};
```

Note that each state structure must be defined in some `.h` include file. Then we add a state structure argument to the end of the argument list.

```
incr(inp, outp, sp)
float *inp, *outp;
struct incr_state *sp;
{
    int i;

    for (i=0; i < sp->size; i++)
        *outp++ = (*inp++) + 1;
}
```

When the graph is created, each `incr` node will be allocated a *distinct* state structure defined by the tag `incr_state`. Thus, a graph might contain many `incr` nodes, each using the identical `incr` primitive to operate on different sized input vectors.

Much of the power of the state structure derives from the fact that the fields are arbitrarily defined by the user. For example, an n-point complex FFT can use the state structure to hold the size of the input vector and a collection of dynamically precomputed complex coefficients.

4.3 Input and Output Functions

If we ever plan to associate `incr` with a source or sink node, then we must write input and output functions for that node so that it can transfer data to or from the host computer. We typically place these functions in the same file as the primitive. For example, this input function is invoked each time an `incr` source node reads its input ports.

```
incr_in(inp, sp)
float *inp;
struct incr_state *sp;
{
    int i;

    for (i=0; i<sp->size; i++)
        scanf("%f", &(inp[i]));
}
```

4.4 Dynamic Initializers

As we learned in the previous section, some or all of the elements of the state structure can be initialized *statically* when a a node is created or *dynamically* at run time before the graph is executed. We will learn how to statically initialize state structures in the next section. To dynamically initialize a state structure, we provide a C function. For example, given the following state structure,

```
struct fft_state {
    int n;          /* number of complex points */
    int isign;      /* 1 is fft, -1 is ifft */
    complex wp[MAXN]; /* twiddle factors */
};
```

we might use a dynamic initializer of the following form to precompute the twiddle factors of an n-point complex FFT.

```
void fft_init(sp)
struct fft_state *sp;
{
    ...
}
```

5 Writing Graph Generators

Once a collection of node functions is assembled, the next step is to create a flowgraph using a graph generator. A graph generator is a C program that uses a library of ASSIGN functions to emit a flowgraph on the standard output.

For the initial implementation of ASSIGN we have chosen to provide a procedural interface rather than a visual interface. Each approach has its advantages and disadvantages. The chief advantage of the procedural approach is that we can exploit the tremendous expressive power of C, including iteration, conditionals, functions, and recursion. This expressive power is extremely helpful for generating large graphs with hundreds or thousands of nodes. Another key advantage is that command line arguments and manifest constants can be used to parameterize the graph generator, so that one graph generator can produce an entire family of graphs.

The main disadvantage of the procedural approach is that people tend to think of flowgraphs as graphical objects. We are investigating existing visual interfaces, and if we are able to find a system with enough expressive power to handle large parameterized graphs, it will not be difficult to replace the existing interface.

The library of graph generator functions provided by ASSIGN includes functions that create an empty graph, associate source and include files with a graph, add an atomic node to a graph, add a graph node to a graph, add input and output ports to a node, add a state structure to a node, add input and output functions to a node, add an edge to a graph, and emit a graph on the standard output. Complete descriptions of these functions, as well as numerous examples, can be found in [8].

6 Related Work

ASSIGN continues the tradition of building parallel program generators for restricted classes of applications. Examples of program generators for image processing, matrix computations, and nested loop computations can be found in [3, 15, 11], respectively.

The flowgraph model has been studied extensively. Some seminal theory on scheduling flowgraphs and an extensive survey of previous results can be found in [5].

Several systems for compiling flowgraph descriptions of signal processing applications have been developed in recent years, including Gabriel[4] at Berkeley and ZC[10] at Carnegie Mellon. Gabriel is a task–level parallel program generator that was originally designed to generate code on single processor systems or small shared memory parallel systems. However, there has been recent work to extend it to distributed–memory systems[14] as well. The ZC system is designed to compile flowgraphs onto linear systolic arrays like the Warp computer [1]. Unlike ASSIGN or Gabriel, where nodes correspond to sequential tasks, ZC allows for nodes to correspond to parallel tasks.

Acknowledgements

A number of people contributed to the development of ASSIGN. Many thanks to Peter Lieu, Jim Wheeler, Harry Printz, Onat Menzilcioglu, Siang Song, Robert Cohn, Susan Hinrichs, Thomas Gross, Jon Webb, and Paul Wiley.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[2] R. G. Babb. Parallel processing with large grain data flow techniques. *Computer*, 17, July 1984.

[3] L. G. Hamey, J. Webb, and I. C. Wu. Low-level vision on Warp and the Apply programming model. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*. Kluwer Academic Publishers, 1987.

[4] E. A. Lee, W. Ho, E. Goei, J. Bier, and S. Bhattacharyya. Gabriel: A design environment for DSP. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(11):1751–1762, November 1989.

[5] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[6] O. Menzilcioglu, H. T. Kung, and S. W. Song. Comprehensive evaluation of a two-dimensional configurable array. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, pages 93–100. IEEE, 1989.

[7] D. M. Nicol and D. R. O'Hallaron. Efficient algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, March 1991.

[8] D. R. O'Hallaron. The assign parallel programming generator. Technical Report CMU-CS-91-141, Carnegie Mellon University, 1991.

[9] A. Oppenheim, A. Willsky, and I. Young. *Signals and Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[10] H. Printz, H. T. Kung, T. Mummert, and P. Scherer. Automatic mapping of large signal processing systems to a parallel machine. In *Proceedings of SPIE Symposium, Real-Time Signal Processing XI*, San Diego, CA, August 1989. Society of Photo-Optical Instrumentation Engineers.

[11] H. B. Ribas. Automatic generation of systolic programs from nested loops. Technical Report CMU-CS-90-143, Department of Computer Science, Carnegie-Mellon University, June 1990.

[12] S. Borkar et. al. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, Kissimmee, FL, November 1988.

[13] S. Borkar et. al. Supporting systolic and memory communication in iWarp. In *17th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM, May 1990.

[14] G. C. Sih and E. A. Lee. Scheduling to account for interprocessor communication within interconnection-constrained processor networks. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 1990.

[15] P. S. Tseng. A parallelizing compiler for distributed memory parallel computers. Technical Report CMU-CS-89-148, Department of Computer Science, Carnegie-Mellon University, May 1989.