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, *describes its use, and gives some example programs.*

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, 6]. 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. The appendix gives numerous examples.

2 Overview

In this section, we summarize the flowgraph programming model, the mapping algorithm, how ASSIGN fits into existing systems, and how we build applications. Later sections provide more details.

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[6], and signal flow graphs[10], flowgraphs have been used for years

*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.

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 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 have the important property of being 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 [8] 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 [8].

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 [7].

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. This feature allows us to test flowgraphs on the workstation, generating code for the iWarp only when we are confident that the graph is producing correct results.

Just to make things concrete, here are the commands we would use to generate, compile, and run a flowgraph called `example.graph`. First, we write a graph generator called `example-gen.c`, then compile and run it to generate the flowgraph in file `example.graph`.

```
% make example-gen
% example-gen ¿ example.graph
```

Next we compile the flowgraph using Assign.

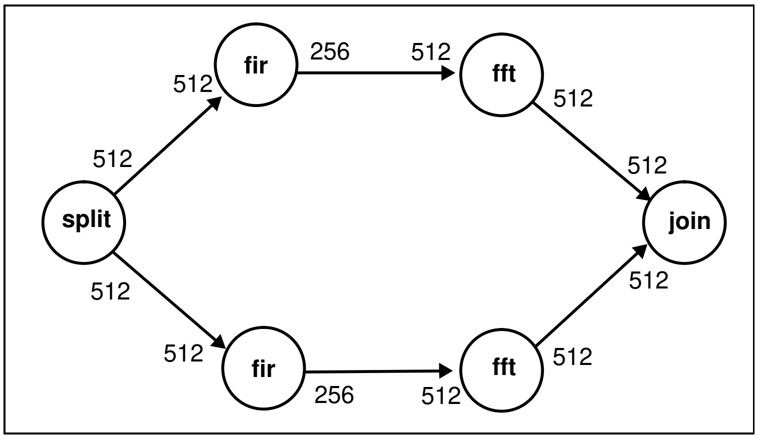

Figure 1: Example flowgraph. Edge labels denote the number of data items produced and consumed each time a node is invoked.

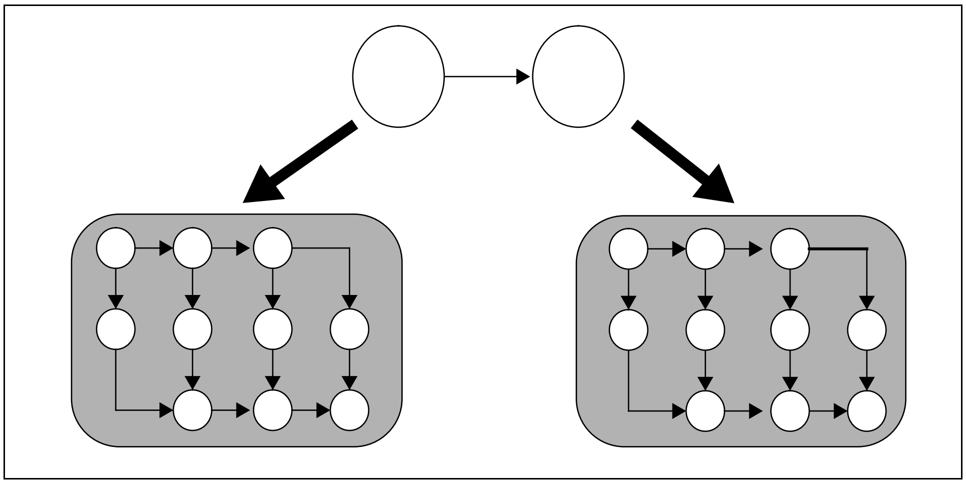

Figure 2: The hierarchical nature of flowgraphs. Here we have a flowgraph model of a real sonar application. The graph consists of two nodes, where each node is itself a subgraph.

% assign example

Next we compile and link the output of ASSIGN using the C compiler and linker supplied by Intel. The makefile for doing this, `example-make`, is generated by ASSIGN.

% make -f example-make

Finally, we run the flowgraph on the iWarp array by executing a shell script, called `go`, produced by the makefile in the previous step.

% go

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

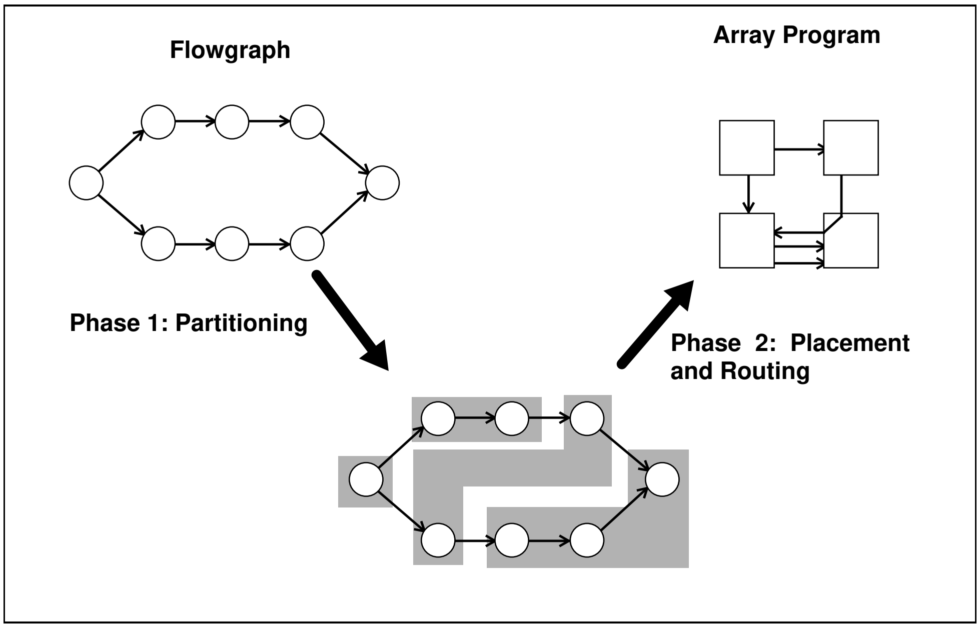

Figure 3: The Assign mapping algorithm. The partitioning phase attempts to minimize the maximum load on any particular cell. The placement and routing phase attempts to minimize the number of flowgraph edges shared by any particular physical communication link.

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).

3.1 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.

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

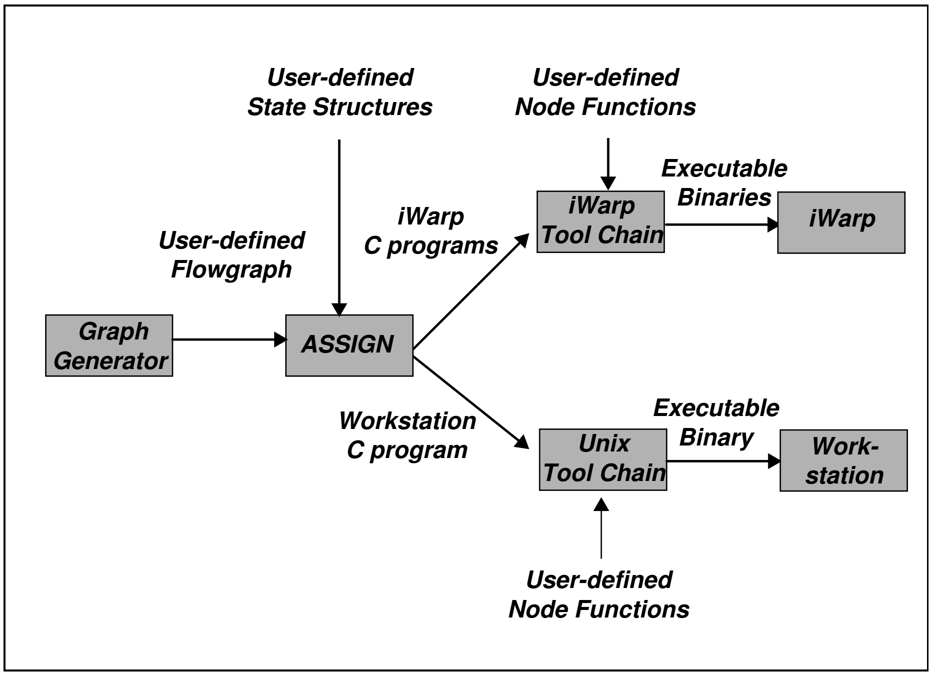

Figure 4: Assign system overview. ASSIGN is a front-end to existing tools. Programs can be compiled onto a parallel computer or a Unix host.

3.2 Ports

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

3.3 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

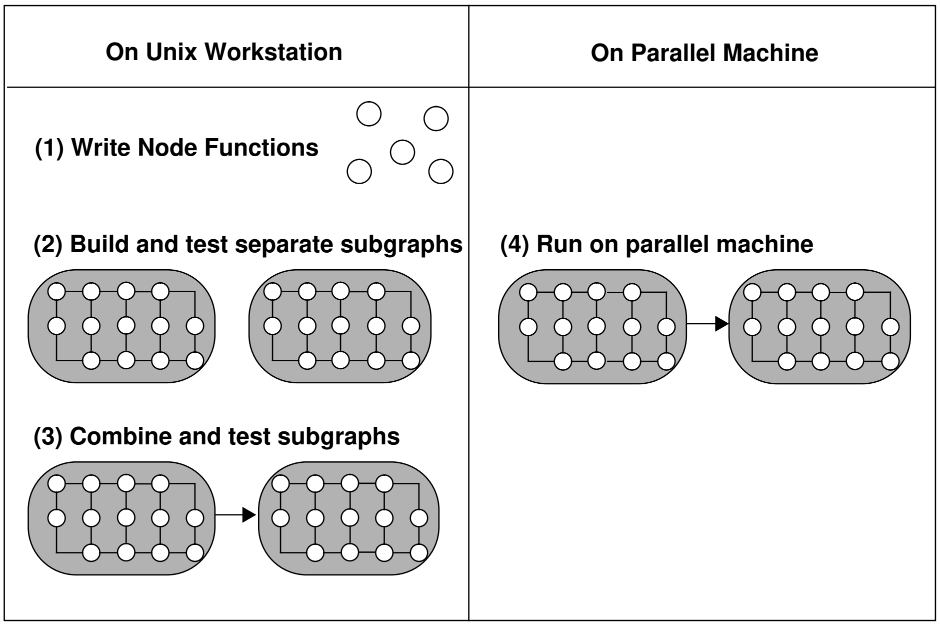

Figure 5: Typical process of programming Assign applications. Notice that the system is constructed incrementally and that most of the work is done on the Unix workstation.

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).

3.4 Execution Weight

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

3.5 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. Each port in a node is bound to at most one edge. We refer to a queue bound to input port k as *input queue* k. Similarly, we refer to a queue bound to output port k as *output queue* k. The two ports bound to an edge must have the same type. The queue corresponding to an edge may be initialized with an arbitrary number of zeros.

3.6 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 [6]. 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 and in the process accumulate a small set of simple primitives that can be used in a wide variety of applications.

4.1 Primitives

The simplest ASSIGN primitive has no input or output arguments.

```
hello()
–
  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. An example of a realistic primitive that uses the state structure is in Section A.6. The example is an n-point complex FFT that uses 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. The 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]));
˝
```

The output function is invoked each time an `incr` sink node writes to its output ports.

```
incr˙out(outp, sp)
float *outp;
struct incr˙state *sp;
–
    int i;

    for (i=0; i¡sp-¿size; i++)
        printf("%f“n", outp[i]);
˝
```

4.4 Dynamic Initializers

Another simple but useful primitive splits an input vector of length `in` into two output vectors of length `out1` and `out2`. Given the following state structure definition

```
struct split˙state –
    int in;
    int out1;
    int out2;
˝;
```

the primitive can be written as

```
split(inp, outp1, outp2, sp)
float *inp, *outp1, *outp2;
struct split˙state *sp;
–
    int i;

    for (i=0; i¡sp-¿out1; i++)
        *outp1++ = *inp++;
    for (i=0; i¡sp-¿out2; i++)
        *outp2++ = *inp++;
˝
```

The dual to the `split` primitive is the join primitive, which appends one input vector to another input vector and places the result in an output vector.

```
struct join˙state –
    int in1;
    int in2;
    int out;
˝;

join(inp1, inp2, outp, sp)
float *inp1, *inp2, *outp;
struct join˙state *sp;
–
    int i;

    for (i=0; i¡sp-¿in1; i++)
        *outp++ = *inp1++;
    for (i=0; i¡sp-¿in2; i++)
        *outp++ = *inp2++;
˝
```

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 a `split` node, for example, we might statically initialize the size of the input vector, and then dynamically determine where the vector will be split.

```
split˙init(sp)
struct split˙state *sp;
–
    sp-¿out1 = sp-¿out2 = sp-¿in/2;
˝
```

A more substantive example can be found in Appendix A.6, where the complex coefficients for an n-point complex FFT are precomputed and stored in the state structure for use in all subsequent invocations of the FFT.

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. For example, functions can be used as an abstraction mechanism to create subgraphs, iteration can be used to generate iterative constructs like pipelines, and recursive functions can be used to create recursive constructs such as fan–out fan–in trees.

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. For example, sonar adaptive beam interpolation (ABI) is characterized entirely by the number of beams and the number of frequencies. It was not difficult for us to write a graph generator that generates arbitrary ABI flowgraphs. Whenever we want to create a different sized ABI program on iWarp, we simply change a couple of constants and recompile the new graph.

Yet another advantage is that a procedural interface based on C is portable across a wide variety of personal computers and workstations and familiar to many programmers.

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.

5.1 Creating Empty Graphs

```
GRAPH *alloc˙graph(name);
char *name;
```

`Alloc_graph` returns a pointer to empty flowgraph `name`.

5.2 Associating Files With a Graph

```
void add˙src˙file(gp, name)
GRAPH *gp;
char *name;
```

```
void add˙inc˙file(gp, name)
GRAPH *gp;
char *name;
```

`Add_src_file` adds source file `name` to a list of source files associated with graph `gp`. Each node function must be contained in some source file. Similarly, `add_inc_file` adds include file `name` to a list of include files associated with graph `gp`. Each state structure definition required by the graph must be contained in some include file.

5.3 Adding Nodes to a Graph

```
NODE *add˙node(gp, name, prim, weight)
GRAPH *gp;
char *name, *prim;
int weight;
```

`Add_node` adds atom `name`, with primitive function `prim` and execution weight `weight` to graph `gp` and returns a pointer to the node that can used in subsequent calls to `add_in_port`, `add_out_port`, and `add_edge`.

```
NODE *add˙graph˙node(gp, gnp)
GRAPH *gp, *gnp;
```

`Add_graph_node` adds the subgraph node `gnp` to graph `gp`. The resulting subgraph node gets the name of subgraph `gnp`, the input ports of subgraph `gnp`'s source node, and the output ports of subgraph `gnp`'s sink node.

```
add˙in˙port(np, name, type, consume˙amt)
NODE *np;
char *name;
int type, consume˙amt;
```

```
add˙out˙port(np, name, type, produce˙amt)
NODE *np;
char *name;
int type, produce˙amt;
```

`Add_in_port` adds input port `name` with type `type`, and consume amount `consume_amt` to node `np`. `Add_out_port` adds output port `name` with type `type`, and produce amount `produce_amt` to node `np`. Valid `type`s are `INT`, `FLOAT`, `DOUBLE`, `COMPLEX`, `COMPLEX_INT`, and `COMPLEX_DOUBLE`.

```
add˙state(np, state, ilist, init)
NODE *np;
char *state, *ilist, *init;
```

`Add_state` associates a state structure with node `np`. The structure is defined by the tag name `state`, initialized at compile time by the static initializer list `ilist`, and initialized at runtime by the dynamic initializer function `init`. If `ilist` is `NULL`, then Assign assumes there is no static initializer list, and similarly for `init`.

```
add˙io(np, input, output)
NODE *np;
char *input, *output;
```

`Add_io` associates input function `input` and output function `output` with node `np`. If `input` is `NULL`, then Assign assumes there is no input function, and similarly for `output`.

5.4 Adding Edges to a Graph

```
add˙edge(gp, out˙np, out˙port, in˙np, in˙port, initq)
GRAPH *gp;
```

```
NODE *out˙np;
char *out˙port;
NODE *in˙np;
char *in˙port;
int initq;
```

`Add_edge` adds an edge to graph `gp` that connects output port `out_port` of node `out_np` to input port `in_port` of node `in_np`. The queue corresponding to the edge is initialized with `initq` zeros (delay elements).

5.5 Outputting Graphs

```
emit˙graph(gp)
GRAPH *gp;

print˙graph(gp)
GRAPH *gp;
```

`Emit_graph` sends an ASCII representation of the flowgraph to the standard output. The output of this function is the input to Assign. `Print_graph` sends a more readable ASCII representation of the flowgraph to the standard output, listing each node, its input and output ports, and its attributes. For example,

```
Node split1
  prim = split, weight = 1000
  state = split˙state, init = split˙init, ilist = 48,16,32
  Input ports:
    left ¡- split0.right : FLOAT, cons = 48
  Output ports:
    down -¿ incr1.left : FLOAT, prod = 16
    right -¿ split2.left : FLOAT, prod = 32
```

Here we have a node called `split1`. `split` is the associated primitive with an execution weight of 1000 clocks. There is a state structure associated with this node that is defined by the tag `split_state`. There is a dynamic initializer functions called `split_init()` that is called before the graph executes. The first three elements of the state structure are initialized at compile time by the C structure initializer list `48,16,32`.

The node has 1 input port (`left`) and two output ports (`down` and `right`). Input port `left` is connected by an edge from node `split0`, port `right`, and has a consume amount of 48 `FLOAT`s. Ouput port `down` is connected by an edge to node `incr1`, port `left`, and has a produce amount of 16 `FLOAT`s. Similarly for output port `right`.

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 [6].

Several systems for compiling flowgraph descriptions of signal processing applications have been developed in recent years, including Gabriel[5] 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. Peter Lieu wrote the iWarp code generator. Jim Wheeler was the first user and made many helpful suggestions. Harry Printz developed the algorithm for normalizing execution rates. Onat Menzilcioglu and Siang Song wrote the original placement and routing code. Robert Cohn wrote the FT Warp simulator used to test earlier versions. Susan Hinrichs patiently answered questions about iWarp software. Thomas Gross, Jon Webb, and Paul Wiley made helpful suggestions.

A Examples

The appendix gives a number of examples of Assign programs using the node functions and state structures developed in Section 4. The aim is to develop some useful graph structures and to illustrate the general way we develop Assign programs.

A.1 incr16

The simplest Assign flowgraph has a single node and no edges, as in Figure 6.

To generate this graph, we start with a simple function that adds an `incr16` node to a graph and returns a pointer to that node.

```
NODE *incr16˙node(gp, name)
GRAPH *gp;
char *name;
–
  NODE *np;

  np = add˙node(gp, name, ”incr”, 16);
  add˙in˙port(np, ”in”, FLOAT, 16);
  add˙out˙port(np, ”out”, FLOAT, sample˙size);
  add˙io(np, ”incr˙in”, ”incr˙out”);
  return(np);
˝
```

The complete graph generator is of the form.

```
#include ¡stdio.h¿
#include ¡assign.h¿

NODE *incr16˙node();

main()
–
  GRAPH *gp = alloc˙graph(”incr˙graph”);

  add˙src˙file(gp, ”incr.c”);
  incr16˙node(gp, ”incr”);
  emit˙graph(gp);
˝
```

A.2 incr

We can generalize the `incr16` graph by using the `incr` primitive from Section 4. The following function adds an `incr` node to a graph and returns a pointer to that node.

```
NODE *incr˙node(gp, name, weight, sample˙size)
GRAPH *gp;
char *name;
int weight, sample˙size;
–
  NODE *np;
  char ilist[32];

  np = add˙node(gp, name, ”incr”, sample˙size);
  add˙in˙port(np, ”in”, FLOAT, sample˙size);
  add˙out˙port(np, ”out”, FLOAT, sample˙size);
  sprintf(ilist, ”%d”, sample˙size);
  add˙state(np, ”incr˙state”, ilist, ”incr˙init”);
  add˙io(np, ”incr˙in”, ”incr˙out”);
  return(np);
˝
```

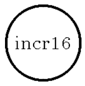

Figure 6: The incr16 graph.

A.3 pipe

This section we describe a general technique for creating pipelines of the type shown in Figure 7. This graph models the common computation where a sequence of operations are performed on a single data stream.

We start with a function that calls `incr_node` to create a pipeline of `incr` nodes and returns a pointer to the resulting graph.

```
GRAPH *incr˙pipe(len, name, weight, size)
int len;
char *name;
int weight, size;
–
  GRAPH *gp = alloc˙graph(name);
  NODE *prev, *curr;
  char s[32];
  int i;

  add˙src˙file(gp, ”incr.c”);
  add˙inc˙file(gp, ”incr.h”);

  prev = incr˙node(gp, ”incr0”, weight, size);
  for (i=1; i¡len; i++) –
    sprintf(s, ”incr%d”, i);
    curr = incr˙node(gp, s, weight, size);
    add˙edge(gp, prev, ”out”, curr, ”in”, 0);
    prev = curr;
  ˝
  return(gp);
˝
```

To emit the pipe in Figure 7, we would write

```
emit˙graph(incr˙pipe(4, “pipe”, 100, 16));
```

A.4 trellis

One of the properties of the current implementation of Assign for iWarp is that communication resources are allocated statically and persist for the duration of the program. In particular, when we map a graph to an iWarp array, each edge that spans two cells is implemented as a statically allocated iWarp pathway. The advantage of this approach is efficient communication between cells, typically one clock between cells in the absence of link contention. The disadvantage is that an iWarp cell can only support a small (currently 20) number of pathways at any point in time. Thus, a node with many input or output edges forces Assign to place that node and all of its adjacent nodes on the same cell. The bottom line is that graphs with high degree, while valid flowgraphs, may end up using only a small number of the available iWarp cells.

This situation typically arises when there are a large number of operations to be performed in parallel on separate data streams. For example, suppose we want to distribute each row of an $n \times m$ matrix to one of n FFT nodes and then collect the resulting vectors to form a new $n \times m$ matrix. Rather than using single nodes — each with n input or output edges — to distribute and collect the rows, we recommend using either a *trellis*, which is described in this section, or a *fan*, which is described in the next section.

Figure 8 gives an example of a trellis with $n = 4$ `incr` nodes, each of which operates on a vector of m data items. Each `split` node peels off the first m data items, passes these m data items to an `incr` node, and passes the rest of its input onto the next `split` node. Each `join` node appends the output of an `incr` node to its input and passes the result to the next `join` node. Note that a trellis is a coarse–grained version of the Apply `GETROW/PUTROW` model[4].

These functions create instances of `split` and `join` nodes.

```
NODE *split˙node(gp, name, weight, in, out1, out2)
GRAPH *gp;
char *name;
int weight;
int in, out1, out2;
–
```

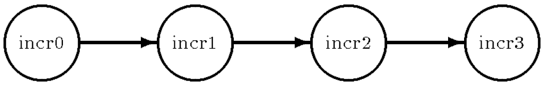

Figure 7: The `pipe` graph.

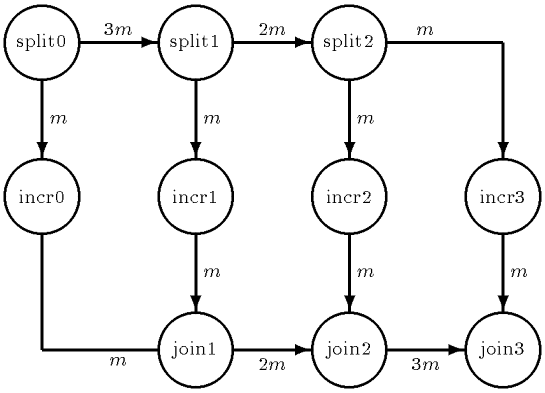

Figure 8: The `trellis` graph.

```
   NODE *np;

   np = add˙node(gp, name, ”split”, weight);

   add˙in˙port(np, ”left”, FLOAT, in);
   add˙out˙port(np, ”down”, FLOAT, out1);
   add˙out˙port(np, ”right”, FLOAT, out2);

   sprintf(s, ”%d,%d,%d”, in, out1, out2);
   add˙state(np, ”split˙state”, s, NULL);

   add˙io(np, ”split˙in”, ”split˙out”);
   return(np);
˝

NODE *join˙node(gp, name, weight, in1, in2, out)
GRAPH *gp;
char *name;
int weight;
int in1, in2, out;
–
   NODE *np;

   np = add˙node(gp, name, ”join”, weight);
   add˙in˙port(np, ”left”, FLOAT, in1);
   add˙in˙port(np, ”up”, FLOAT, in2);
   add˙out˙port(np, ”right”, FLOAT, out);
   sprintf(s, ”%d,%d,%d”, in1, in2, out);
   add˙state(np, ”join˙state”, s, NULL);
   add˙io(np, ”join˙in”, ”join˙out”);
   return(np);
˝
```

This function creates a trellis of n `incr` nodes. We assume there are constants WEIGHT and SIZE that give the execution weight and sample size of an `incr` node.

```
GRAPH *trellis(n, name)
int n;
char *name;
–
   GRAPH *gp;
   NODE *prev˙split˙np, *prev˙join˙np;
   NODE *split˙np, *join˙np, *incr˙np;
   int i, len;

   gp = alloc˙graph(name);

   add˙src˙file(gp, ”incr.c”);
   add˙inc˙file(gp, ”incr.h”);
   add˙src˙file(gp, ”split.c”);
   add˙inc˙file(gp, ”split.h”);
   add˙src˙file(gp, ”join.c”);
```

```
    add˙inc˙file(gp, ”join.h”);

    if (n == 1) –
      incr˙node(gp, ”incr0”, WEIGHT, SIZE);
      return(gp);
    ˝

    len = SIZE*n;
    prev˙split˙np = split˙node(gp, ”split0”, WEIGHT, len, SIZE, len-SIZE);
    prev˙join˙np = incr˙node(gp, ”incr0”, WEIGHT, SIZE);
    add˙edge(gp, prev˙split˙np, ”down”, prev˙join˙np, ”left”, 0);

    for (i=1; i¡n-1; i++) –
      len = (n-i)*SIZE;
      sprintf(s, ”split%d”, i);
      split˙np = split˙node(gp, s, WEIGHT, len, SIZE, len-SIZE);
      sprintf(s, ”incr%d”, i);
      incr˙np = incr˙node(gp, s, WEIGHT, SIZE);
      sprintf(s, ”join%d”, i);
      join˙np = join˙node(gp, s, SIZE, i*SIZE, SIZE, i*SIZE+SIZE);

      add˙edge(gp, split˙np, ”down”, incr˙np, ”left”, 0);
      add˙edge(gp, incr˙np, ”right”, join˙np, ”up”, 0);
      add˙edge(gp, prev˙split˙np, ”right”, split˙np, ”left”, 0);
      add˙edge(gp, prev˙join˙np, ”right”, join˙np, ”left”, 0);

      prev˙split˙np = split˙np;
      prev˙join˙np = join˙np;
    ˝
    sprintf(s, ”incr%d”, n-1);
    incr˙np = incr˙node(gp, s, WEIGHT, SIZE);
    sprintf(s, ”join%d”, n-1);
    join˙np = join˙node(gp, s, SIZE, (n-1)*SIZE, SIZE, n*SIZE);

    add˙edge(gp, incr˙np, ”right”, join˙np, ”up”, 0);
    add˙edge(gp, prev˙split˙np, ”right”, incr˙np, ”left”, 0);
    add˙edge(gp, prev˙join˙np, ”right”, join˙np, ”left”, 0);

    return(gp);
  ˝
```

To generate the trellis in Figure 8, we would write

```
  emit˙graph(trellis(4, “trellis”));
```

A.5 fan

An alternative to the trellis is the *fan* shown in Figure 9. We start with a recursive function that returns fans of size $n = 2^k$. We assume there is a function `pwr2(k)` that returns 2^k, and we assume there are constants WEIGHT and SIZE that give the execution weight and sample size of an `incr` node.

```
  GRAPH *fan2(k, name)
  int k;
  char *name;
  –
    GRAPH *gp, *fan1˙gp, *fan2˙gp;
    NODE *split˙np, *join˙np, *fan1˙np, *fan2˙np;
    int size, i;

    sprintf(s, ”fan%d˙%s”, pwr2(k), name);
    gp = alloc˙graph(s);

    add˙src˙file(gp, ”incr.c”);
    add˙inc˙file(gp, ”incr.h”);
    add˙src˙file(gp, ”join.c”);
    add˙inc˙file(gp, ”join.h”);
    add˙src˙file(gp, ”split.c”);
    add˙inc˙file(gp, ”split.h”);

    if (k == 0) –
      incr˙node(gp, ”incr”, WEIGHT, SIZE);
      return(gp);
    ˝

    size = SIZE*pwr2(k-1);
    split˙np = split˙node(gp, ”split”, 100, size*2, size, size);

    fan1˙gp = fan2(k-1, ”1”);
    fan2˙gp = fan2(k-1, ”2”);
    fan1˙np = add˙graph˙node(gp, fan1˙gp);
    fan2˙np = add˙graph˙node(gp, fan2˙gp);

    join˙np = join˙node(gp, ”join”, size, size, size, size*2);

    add˙edge(gp, split˙np, ”down”, fan1˙np, ”left”, 0);
    add˙edge(gp, split˙np, ”right”, fan2˙np, ”left”, 0);
    add˙edge(gp, fan1˙np, ”right”, join˙np, ”left”, 0);
    add˙edge(gp, fan2˙np, ”right”, join˙np, ”up”, 0);

    return(gp);
  ˝
```

We can use the `fan2` function to create arbitrary sized fans. The idea is based on the fact that an integer n can always be expressed as the sum of integers which are powers of 2. For example, $7 = 4 + 2 + 1$.

```
  GRAPH *fan(n, name)
  int n;
```

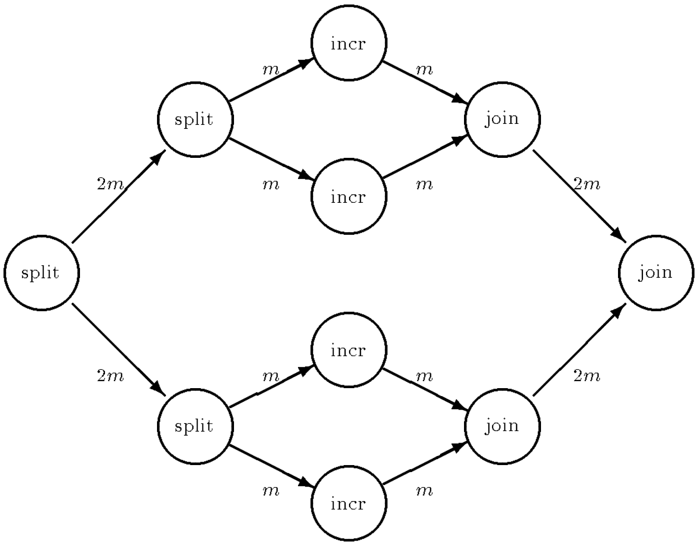

Figure 9: The `fan` graph.

```
char *name;
–
   int m, k; /* n=(m=2ˆk)+r */
   int in1, in2;

   GRAPH *gp, *fan˙gp, *fan2˙gp;
   NODE *split˙np, *join˙np, *fan˙np, *fan2˙np;

   if (n == 0)
      return(NULL);

   /* find the largest k such that 2ˆk = m ¡= n */
   for (k=0,m=1; m¡¡1 ¡= n; m¡¡=1, k++)
      ;

   /* build fans of size 2ˆk and n - 2ˆk */
   fan2˙gp = fan2(k, ”1”);
   if (!(fan˙gp = fan(n-m, ”2”))) –
      return(fan2˙gp);
   ”
   else –
      gp = alloc˙graph(sprintf(s, ”fan%d˙%s”, n, name));
      in1 = SIZE * m;
      in2 = SIZE * (n-m);
      split˙np = split˙node(gp, ”split”, 100, in1 + in2, in1, in2);
      fan2˙np = add˙graph˙node(gp, fan2˙gp);
      fan˙np = add˙graph˙node(gp, fan˙gp);
      join˙np =  join˙node(gp, ”join”, in2, in1, in2, in1 + in2);
      add˙edge(gp, split˙np, ”down”, fan2˙np, ”left”, 0);
      add˙edge(gp, split˙np, ”right”, fan˙np, ”left”, 0);
      add˙edge(gp, fan2˙np, ”right”, join˙np, ”left”, 0);
      add˙edge(gp, fan˙np, ”right”, join˙np, ”up”, 0);
      add˙src˙file(gp, ”incr.c”);
      add˙inc˙file(gp, ”incr.h”);
      add˙src˙file(gp, ”join.c”);
      add˙inc˙file(gp, ”join.h”);
      add˙src˙file(gp, ”split.c”);
      add˙inc˙file(gp, ”split.h”);

      return(gp);
   ”
”
```

To generate the fan in Figure 9, we would write

```
emit˙graph(fan(4, ‘‘fan”));
```

A.6 fft

We conclude with an example of a realistic ASSIGN primitive, an n point complex FFT. We start by defining the 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 initialize the size of the FFT when the node is created, and then dynamically precompute the complex coefficients using the following dynamic initializer.

```
void fft˙init(sp)
struct fft˙state *sp;
–
  int n, mmax, m, j, istep, i, cnt;
  double wtemp, wr, wpr, wpi, wi, theta;

  cnt = 0;
  n = (sp-¿n)¡¡1;
  mmax = 2;
  while (n ¿ mmax) –
    istep = 2*mmax;
    theta = M˙2˙PI / (sp-¿isign*mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;
    sp-¿wp[cnt].r = wr;
    sp-¿wp[cnt].i = wi;
    cnt++;
    for (m=0; m¡mmax-1; m+=2) –
      wr = (wtemp=wr)*wpr - wi*wpi + wr;
      wi = wi*wpr + wtemp*wpi + wi;
      sp-¿wp[cnt].r = wr;
      sp-¿wp[cnt].i = wi;
      cnt++;
    ”
    mmax = istep;
  ”
”
```

The following FFT primitive uses the precomputed coefficients to compute an n point complex DFT.

```
  void fft(inp, outp, sp)
  complex *inp, *outp;
  struct fft˙state *sp;
  –
    int i, j=0, m, istep;
    int n = sp-¿n, isign = sp-¿isign, mmax = 1;
    complex *wp = sp-¿wp-1, temp;

    /* sort input into bit reversed order */
    for (i=0; i¡n; i++) –
      if (j ¿ i)
        SWAP(inp[i], inp[j]);
      m = n ¿¿ 1;
      while (m ¿= 1 && j ¿= m) –
        j -= m;
        m ¿¿= 1;
      ”
      j += m;
    ”

    /* perform the FFT on the bit reversed input */
    while (n ¿ mmax) –
      istep = 2*mmax;
      wp++;
      for (m=0; m¡mmax; m++) –
        for (i=m; i¡n; i += istep) –
          j = i + mmax;
          temp.r = (*wp).r * inp[j].r - (*wp).i * inp[j].i;
          temp.i = (*wp).r * inp[j].i + (*wp).i * inp[j].r;
          inp[j].r = inp[i].r - temp.r;
          inp[j].i = inp[i].i - temp.i;
          inp[i].r += temp.r;
          inp[i].i += temp.i;
        ”
        wp++;
      ”
      mmax = istep;
    ”

    for (i=0; i¡n; i++) –
      (outp)-¿r = (inp)-¿r;
      (outp++)-¿i = (inp++)-¿i;
    ”
  ”
```

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] L. G. Hamey, J. Webb, and I. C. Wu. An architecture independent programming language for low-level vision. *Computer Vision, Graphics, and Image Processing*, 48:246–264, 1989.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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.