A Comparative Study Of The Programmability Of A Signal Processing Application In An MIMD And An SIMD Multiprocessor

Dae-Kyun Yoon and Jean-Luc Gaudiot

CENG Technical Report 94-22

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4484

September 1994

A Comparative Study of the Programmability of a Signal Processing Application in an MIMD and an SIMD Multiprocessor

Dae-Kyun Yoon and Jean-Luc Gaudiot

CENG Technical Report 94-22

Computer Engineering Division
Electrical Engineering - System Department
University of Southern California
Los Angeles, CA 90089-2562

September 1994

Contents

1 Introduction 3

2 SISAL Programming Environment 3

2.1 SISAL . . . 3

2.2 IF1 . . . 4

2.3 OSC . . . 6

3 Data-parallel Programming on MP-1 8

3.1 Overview of the MP-1 System . . . 8

3.2 Programming the MP-1 System . . . 8

3.2.1 Data-parallel programming . . . 9

3.2.2 Overview of MPL . . . 10

3.2.3 Communication . . . 12

4 SISAL Programming on MP-1 13

4.1 Data-Parallel Transformation of IF1 Graph . . . 14

4.2 Generating MPL code . . . 18

5 Implementation of the Target Application 20

5.1 Description of the Application – Split-Step Algorithm . . . 20

5.2 Analysis of the Application using SISAL Tools . . . 21

5.3 Implementation of the Application on MP-1 . . . 25

6 Discussion: SIMD *vs* MIMD 30

7 Conclusion 32

A SISAL code of Split-Step 35

B MPL code of Split-Step 36

C SISAL code of FFT 39

D IF1 Graph of FFT 41

E MPL code of FFT 44

1 Introduction

The major goal of this project is to assess the programmability and the performance of signal processing applications on a SIMD machine, specifically, in the functional programming environment. In a pure functional programming language [2], parallelism need not be expressed by the programmer since it is implicit. Because a program written in a pure functional language is side-effect-free, the relative order of the program statements is not as significant as in an imperative language. Each statement can be executed in any order as long as the data dependencies are respected. Therefore, it is easier for the compiler to extract parallelism out of a program written in a functional programming language. Moreover, programs can be easily ported to other platforms since details about specific machines or operating systems can be hidden from the programmer.

We chose a functional language, SISAL (Streams and Iterations in a Single Assignment Language) [14], for the parallel implementation of our application. Besides the SISAL is a functional language it has already been available on many shared memory multiprocessor systems as well as many sequential machines. However, there is no SISAL compiler available for any SIMD machine. Our research is therefore mainly focused on how a SISAL program can be effectively implemented on a data-parallel machine such as MP-1/MP-2.

In this report, we present many issues of programming a signal processing application in the SISAL programming environment. In the rest of this report, we will present the features of SISAL programming environment and the parallel programming on MP-1 system. We will also describe our target application and will show how it can be implemented on MP-1 coupled with SISAL programming environment. In the last part of this report, the comparison of programming on two different types of parallel machines, MIMD *vs.* SIMD, will be presented.

2 SISAL Programming Environment

In this section, we will describe the characteristics of SISAL and its intermediate code, IF1. We will also describe the Optimizing SISAL Compiler (OSC), which has already been implemented on many other shared-memory multiprocessor systems.

2.1 SISAL

SISAL has been proven highly effective in developing applications for multiprocessor systems [4]. The optimizing SISAL compiler (OSC) has been developed for various shared memory multiprocessor systems [3, 5, 16]. It has also been shown that the performance of several representative SISAL programs is comparable to

(or better than) that of Fortran programs on multiprocessor systems and even on single processor systems. [4].

SISAL has the following characteristics:

- *It is applicative*: Each function call returns one or more values. There is no global space for the sharing of variables. Communication between the caller and the callee function is done only through the input arguments and the returned values.
- *It follows the single assignment rule*: Each variable can be assigned a value only once within the same scope (although it can of course be used many times in the life of the program). Due to the single assignment rule, there is no aliasing problem, which thus facilitates the detection of parallelism in a program. [1]
- *It is strongly typed*: Each value has a type. Thus, in the definition part of each function, every argument is explicitly given a type.

There are six basic scalar types in SISAL: boolean, integer, real, double real, null and character. A data structure in SISAL consists of arrays, streams, records and unions. Each basic data type has its associated set of operations, while record, union, array and stream types are treated as mathematical sets of values just as the basic scalar type. Interestingly, there is a special value called *error*. This value is associated with a special operator *iserror* which is used to handle exceptions at run-time such as division by zero, *etc.*

Three types of control mechanisms are implemented in SISAL: the *forall*, the *select*, and the *iteration* constructs. These constructs can be nested and are implemented by multilevel (hierarchical) graph structures in the lower-level graph representation (IF1) . The *forall* construct is used to specify parallel loops enabling the parallel execution of the loop body while the *iteration* only allows the sequential execution of the loop body. *Select* corresponds to an *if–then–else* structure. Examples of SISAL programs are shown in figure 1.

2.2 IF1

The IF1 (Intermediate Form 1) graph is produced by the front-end of the SISAL compiler. The basic components of an IF1 graph are *graph*, *node*, and *edge (arc)*.

There are two types of IF1 nodes: the simple node and the compound node. While simple nodes implement actual operations, compound nodes correspond to the control structure of the SISAL program. Each compound node has one or more subgraphs according to the node type. For example, the *forall* compound node has three subgraphs, *generator*, *body*, and *returns*. Each subgraph of a compound node is controlled by predefined implicit data dependencies. In other words, no explicit arcs exist between subgraphs. There are four types of compound nodes: *forall*, *select*, *iteration*[2], and *tagcase*.

Each edge is associated with a type. This implies that each edge can carry either a single value (scalar) or a structured data (array or record) together with the source node (and port) and the destination node (and port). When an edge also has immediate data, it is called a *literal edge* and has no source node.

[1] Consider conversely the *common* or the *equivalent* statements in Fortran. They allow reference to the same storage cell under many different names.

[2] *Iteration* is actually a combined construct of two types of sequential loops (*loopA* and *loopB*)

```
% Example of forall
function DotProd(n:integer;a,b:array[integer] returns integer)
        for i in 1,n
                s := a[i]*b[i]
        returns
                value of sum s
        end for
end function

% Example of iteration
function factorial(n:integer returns integer)
        for initial
                f := 1;
                i := 0;
        while i < n
        repeat
                i := old i + 1;
                f := old f * i;
        returns
                value of f
        end for
end function

% Example of select
function Max(a,b:real returns real)
        if a > b then a else b end if
end function
```

Figure 1: Examples of SISAL program

As an example, the IF1 graph of the `DotProd()` function (the corresponding SISAL code is shown in figure 1) which calculates the *inner-product* of two vectors is shown in figure 2.

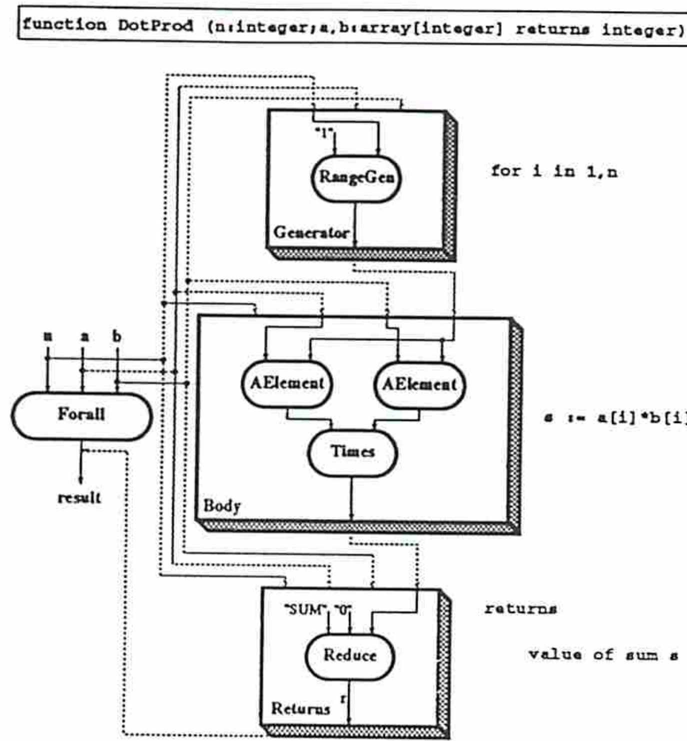


Figure 2: IF1 graph of dot-product function

In the figure, the dotted lines are not real edges in the IF1 graph. They are only defined by implicit dependencies between the subgraphs of the *forall* compound node.

2.3 OSC

OSC (Optimizing SISAL Compiler) has been developed as a part of the SISAL project [10]. The objectives of the project are to:

1. define a general-purpose applicative language,
2. define a language-independent intermediate form for data-flow graphs,
3. develop optimization techniques for high-performance parallel applicative computing,
4. develop a micro-tasking environment that supports data-flow on conventional computer systems,
5. achieve execution performance comparable to that of Fortran, and
6. validate the applicative style of programming for large-scale scientific applications.

OSC targets a wide variety of platforms including conventional single processor machines and shared memory multiprocessor systems. The portability of OSC is due to the choice of C as a target object language.

Most of the compilation steps are therefore the same on all the platforms. The machine dependent features of each platform are implemented in the run-time library of OSC. The structure of OSC is shown in figure 3.

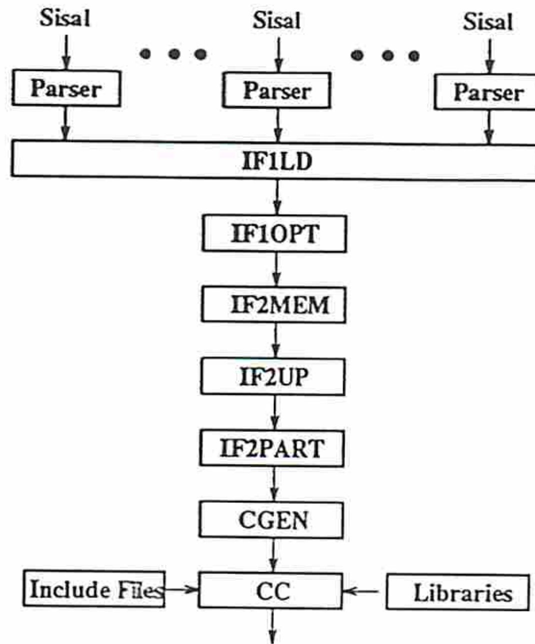


Figure 3: Structure of OSC

IF1OPT performs the various machine-independent scalar optimizations such as loop invariant removal, common subexpression elimination, dead code elimination,*etc*.

The major overhead incurred during the execution of a SISAL program (and any pure functional program) is caused by copy operations of structured data because of its *side-effect-free* principle (embedded in the single assignment rule). OSC reduces this overhead by incorporating *in-place* operations (also called *AT* operations) which are used to access structured data by reference rather than by their values while preserving the overall data dependence. In figure 3, IF2MEM and IF2UP extend the IF1 graph with these *in-place* operations. The extended graph is called IF2 [17]. The main job of IF2MEM is thus to preallocate storage for new structures (*build-in-place* analysis) while IF2UP facilitates the modification of a single data element inside the structure (*update-in-place* analysis). Note that it may be necessary to serialize some structure handling operations in order for the IF2 graph to work correctly after the *update-in-place* analysis. However, it should also be noted that this loss of parallelism is compensated by the reduced overhead that would be increased if whole new structure were created/copied whenever any single components were modified.

IF2PART is a parallelizing module designed to define the desired granularity of parallelism. The analysis is based on the estimated execution time which is determined by various parameters such as computation time, and spawning overhead. The current implementation selects only *forall* compound nodes to slice the loop into many pieces for parallel execution.

CGEN translates the optimized IF2 graphs into equivalent C code. The generated C code together with the machine-dependent run-time libraries are then compiled to produce an executable.

3 Data-parallel Programming on MP-1

In this section, we describe the overall programming environment of the MP-1 system from both the hardware and the software perspectives [12, 13].

3.1 Overview of the MP-1 System

The MasPar MP-1 massively parallel computer consists of a front-end host workstation and a back-end *data-parallel unit (DPU)*. The DPU has two components: an Array Control Unit (ACU) and an array of PEs (Figure 4). Each component can be described as follows:

- *Front End* : The front end mainly handles the user interface, inputs/outputs, and the normal functions which a conventional workstation would provide, such as compiling and network functions. The initial data setup and the final collection of results can also be performed by the front end. Incidentally, it should be noted that the front end of the MP-1 is a DECstation running the Ultrix operating system.
- *DPU* : The DPU handles most of the computations. It can be viewed as a massively parallel machine (PE Array) plus an Added Scalar Processor (ACU).
  - *ACU* : The main job of the ACU is to decode and broadcast instructions to all the PEs in the PE array. The ACU also has a RISC processor that can operate on scalar variables.
  - *PE Array* : The Processor Element (PE) Array forms the computational core of the MP-1 system and includes up to 16,384 PEs operating in parallel. Each PE is a custom-designed RISC processor with 64KB dedicated data-memory and 40 32-bit registers. The data in the PE memory space is distributed, while each instruction from the ACU is executed on all the PEs simultaneously.

The highly optimized communication between neighboring PEs is achieved by the *Xnet* while a flexible global communication is performed by the *router*. The communication primitives will be later described in section 3.2.3.

3.2 Programming the MP-1 System

The MP-1 massively parallel computer provides the programming tools to support data-parallel programming model. These include the MasPar Programming Language (MPL) as well as its own runtime libraries for communication. In this section, we describe the concept of data-parallel programming and the programming tools of the MP-1 system.

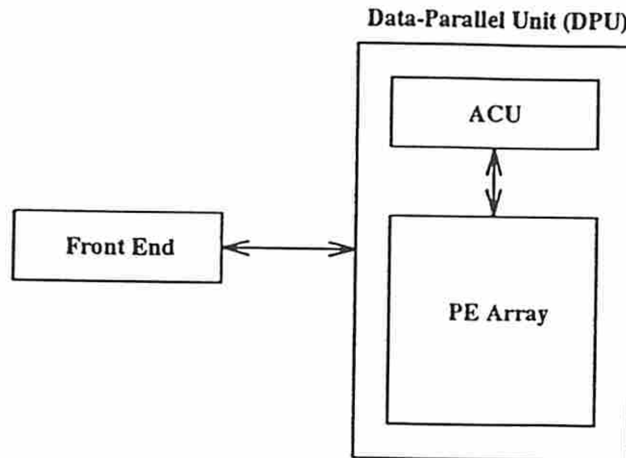


Figure 4: Overall structure of the MP-1 system

3.2.1 Data-parallel programming

Generally speaking, there are two basic programming models which programmers can employ to take advantage of parallel systems:

1. Data-parallel
2. Control-parallel

In the *data-parallel* model, there is a large data set that needs to be processed, and each processor executes the same set of instructions on the different data in the set (SIMD: Single Instruction Stream Multiple Data Streams). The main feature of the architecture for data-parallel model is that it includes a large number of rather simple processors. In order for a program to be executed efficiently in this model, the synchronization between processors should occur in a very regular pattern. Thus, by highly optimizing a certain type of communication pattern (*e.g* between neighboring processors), the overall communication overhead can be well-matched to the speed of the processor.

In the *control-parallel* model, each processor executes separate processes/functions to solve either independent problems or cooperate on the same problem (MIMD: Multiple Instruction Stream Multiple Data Stream). This model is more flexible than the data-parallel model, and its applicability is wider than the data-parallel model. However, the underlying architecture for the model usually results in more complex control and communication structures. Moreover, due to the complexity, it is not yet feasible to employ very large numbers of processors in a single system as easily as in data-parallel architectures.

When using either the data-parallel or the control-parallel model, the algorithms need to be redesigned to take advantage of the corresponding model. In the case of data-parallel programming, the algorithms should be designed for large amounts of data, and it assumes that each data element is assigned to one processor. Therefore, as we increase the data size, the usable parallelism increases accordingly and the program can be scaled up easily to provide increased performance. For example, consider the following

expression:

For $i = 0 \ldots N-1$

$$C[i] = f(A[i], B[i])$$

where f is an arbitrary function of two inputs which we need to evaluate. In the data-parallel programming model, the above expression can be simply implemented as follows:

$$c = f(a, b)$$

assuming the elements of A and B are distributed among the PEs such that the value of a on PE_i is indeed the value of $A[i]$ (Figure 5).

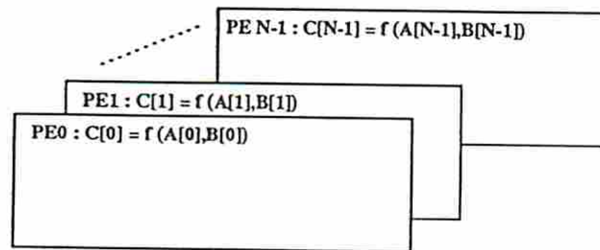


Figure 5: The function f is evaluated in parallel for each element of A and B across the PEs

3.2.2 Overview of MPL

The MasPar Programming Language (MPL) is used to program the DPU. MPL is based on ANSI C with the added statements, keywords, and library functions to support data parallel programming.

The key features of MPL are its support of the following data types:

- The *Plural* data type is used to specify data storages in DPU, *i.e.* parallel data, while without *plural* the storage is defined in the ACU.
- The *Plural* expressions can be defined by any arithmetic and addressing operations on the *plural* data type. For example, if k, i, j are plural types, the expression, `k = i + j`, is executed on all the active PEs in the DPU.
- The semantics of the SIMD control statement are implemented by the concept of the active set. The active set is the set of PEs that is enabled at any given time during the execution. This set is defined by conditional tests in users program. For example, consider the code segment shown in Figure 6.

  When the first statement is executed, the active PEs are those in which the condition, `i > j`, is true, and for the second statement the active set becomes complemented. At program startup, all PEs are active and the active set varies depending on the program's control structure.

```
plural int i,j,k;
...
if ( i > j)
    k = i - j;
else
    k = j - i;
...
```

Figure 6: An example of activeness control in MPL.

Consider the following example once again:

For $i = 0 \ldots N-1$

$$C[i] = f(A[i], B[i])$$

Now, assuming the number of elements is less than the number of PEs, we can write an MPL code as shown in Figure 7:

```
1. float A[N],B[N],C[N];
2. plural float a,b,c;
   ...
3. if (iproc < N) {
4.      a = A[iproc], b = B[iproc];
5.      c = f (a,b);
6. }
7. for (i=0;i<N;i++)
8.      C[i] = proc[i].c;
   ...
9. plural float f (plural float a,plural float b)
   ...
```

Figure 7: An example of an MPL code for parallel evaluation of function f.

In step 2, we define a, b, c as a parallel data whose storage is defined in all the PEs. Steps 4 and 5 are executed on those PEs whose number is less than N. Note that the function f is declared also as a parallel function in order for each PE to be able to call the same function simultaneously (step 9). After the parallel evaluation of f is done, the data is collected by C in steps 7 and 8. If there are any further computations on the result of f, the data collection will be deferred until all the parallel computations have completed on all PEs.

3.2.3 Communication

MPL provides a direct control over the PE-to-PE communication. There are two kinds of communication mechanisms available: near-neighbor regular communication via the *Xnet* communication network, and random communication via the global *router*.

<u>Xnet</u>

Xnet is faster than the global router. It is therefore advantageous to use Xnet for inter-PE communication whenever possible. Xnet connects each PE to its neighboring PEs in a 2-D mesh. The overall topology of Xnet is the 8-way toroidal wrap-round, *i.e.*, each PE is connected to 8 neighboring PEs (Figure 8). The *Xnet* library function is used to implement Xnet communication in MPL in the following form:

xnet*DD*[*distance*].*variable*

The above function designate the *variable* in the PE which is *distance* away in the *DD* direction. *DD* can be one of *N, NE, E, SE, S, SW, W* and *NW*, each corresponding to one of the eight directions.

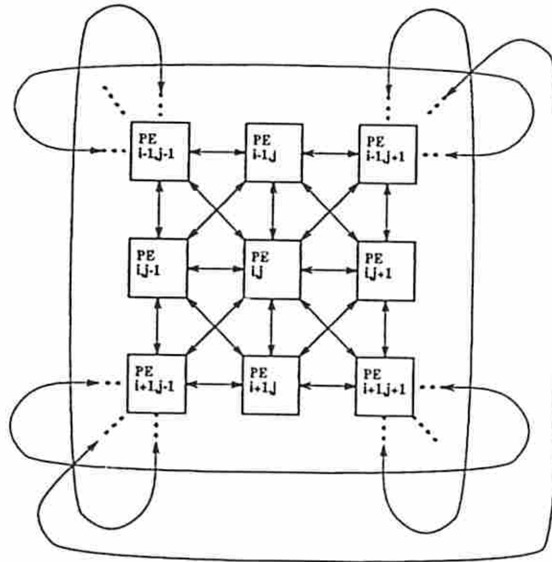


Figure 8: 8-way toroidal wrap around topology of Xnet.

As an example of using Xnet, we chose an image processing algorithm in which one is to average pixel values with 8 neighbors for each pixel[3]. This can be achieved with the MPL code shown in figure 9.

<u>Global Router</u>

The *global router* allows PEs to directly access any other PE in the PE array. It is mainly used for the communication patterns that are not regular, especially when the communication pattern is computed at run time. The format of the *router statement* is as follows:

[3]The pixel averaging is one of the popular and the simplest ways for anti-aliasing in computer image generation [6].

```
   ...
1. pixel = pixel + xnetW[1].pixel + xnetE[1].pixel;
2. pixel = pixel + xnetN[1].pixel + xnetS[1].pixel;
3. pixel = pixel/9;
   ...
```

Figure 9: An example of Xnet: Averaging pixels with 8 neighbors.

router[*PEnum*].*variable*

The above function designates the *variable* in the PE *PEnum*.

There is one router channel for each set of 16 PEs. Hence there is a possibility of contention as the router channel works on a *first-come-first-served* basis. Unlike Xnet, the communication time of the router is not dependent on the distance of the two communicating PEs. In general, Xnet performs better for close neighbors (whose distance is less than 32) than the router. However, the algorithms requiring irregular or complex communication patterns will be solved better by using the router.

4 SISAL Programming on MP-1

As mentioned earlier in this report, there is no SISAL compiler available for any data-parallel machine. We therefore designed schemes to translate SISAL programs into MPL (a data-parallel version of C). The steps of translating a SISAL program into the corresponding MPL code can be described as follows:

1. Compile the SISAL program into an IF1 graph — We use the front-end of OSC (Optimizing SISAL Compiler) to generate the IF1 graph.
2. Perform traditional scalar optimization and structure optimizations. — The optimizer of OSC can be used for this purpose. It performs data-dependent analysis and also optimizes the structure (array) handling operations.
3. Transform the IF1 graph for the exploitation of data-parallelism. — In this step, parallel *forall* loops are transformed for data-parallel execution. This part is indeed the major contribution of our work. We propose a transformation scheme which will be discussed later in this report.
4. Generate MPL code. — Finally, we generate MPL code from the transformed data-flow graph.

The overview of the translation is also show in figure 10. In the figure, the shaded steps are already implemented in OSC.

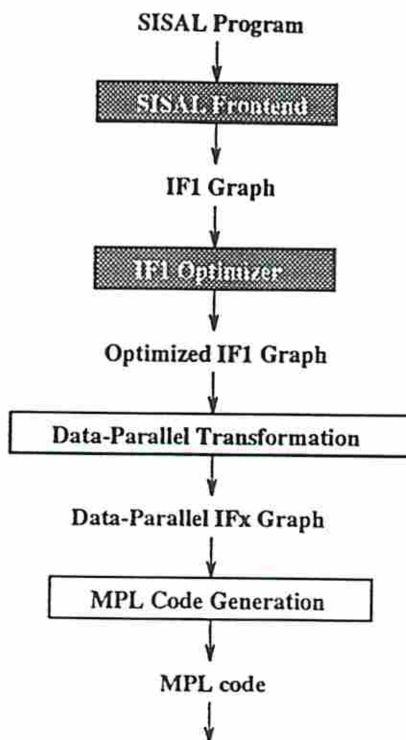


Figure 10: The overview of the translation steps.

4.1 Data-Parallel Transformation of IF1 Graph

In an IF1 graph, the main construct in which we can exploit data-parallelism is the *forall* loop. Moreover, on data-parallel machines like MP-1/MP-2, a large data array must be allocated across the PE array due to the limitation of memory space on the Array Control Unit (ACU). Therefore, even the array is to be implemented for the sequential execution only, it is reasonable to assume that all the arrays are simply allocated on all PEs across the PE array. In describing our transformation scheme, we also assume that all the input arrays are already allocated over the PEs based on *cut-and-stack* mapping [12].

A Forall loop consists of three subgraphs, the *generate* graph, the *body* graph and the *return* graph. For example, the SISAL program shown in figure 11 is translated into the IF1 graph shown in figure 12.

Transforming *MemAlloc* and *AGatherAT*

MemAlloc is a special node of IF1 (IF2) used to specify the allocation of a memory block. In a shared memory multiprocessor system, MemAlloc simply allocates a contiguous block of memory for an array. However, in the data-parallel programming paradigm, when an array must be allocated across the PE array, only a small portion of the array can be assigned to each PE. For this purpose, we thus introduce a new construct called *pMemAlloc* (figure 13). The input to *pMemAlloc* is the total size of the array and the output is the pointer to the newly allocated slot on each PE. The size of the space allocated on each PE is also determined by the total number of processors ("nproc").

The *AGatherAT* node is used to collect the result of the loop-body. We simply replace *AGatherAT* with

```
function foo (v:OneDim;n:integer returns OneDim)
     for i in 0,n-1
          r := f (v[g(i)],v[h(i)]);
     returns
          array of r
     end for
end function
```

Figure 11: A simple SISAL program with forall loop.

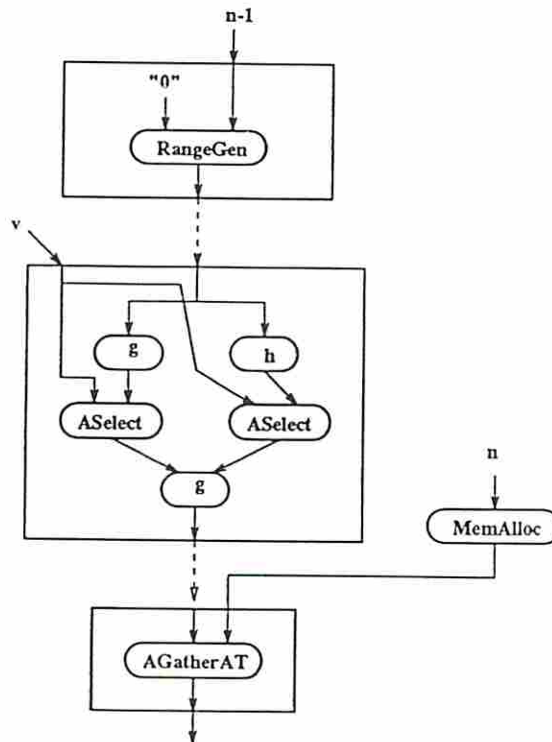


Figure 12: A simple IF1 graph with forall loop.

pAGatherAT in accordance with the output of *pMemAlloc*.

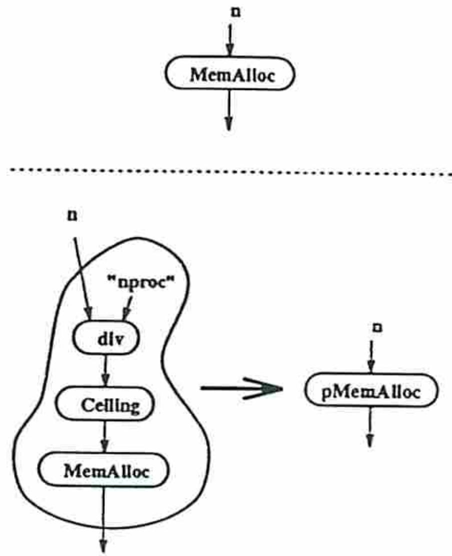


Figure 13: Transforming *MemAlloc* for array distribution across the PE array.

Transforming *RangeGenerate*

RangeGenerate initiates the parallel invocation of a loop body by generating iteration indices. Just as in the case of *MemAlloc*, on each PE, only a small number of iterations are performed. This number is based on the total number of iterations and the number of available PEs (nproc). The *RangeGenerate* node is replaced with two new constructs called *SlashBounds* and *pRangeGenerate* as shown in figure 14.

In most cases, the array index is used in the body of a forall loop. It is therefore necessary to retrieve the original index out of the local index (of the sliced array on each PE). Retrieval of the index can be efficiently performed using a bit-wise shift operation only if the number of processors is a power of two. For example, if the number of processors is 2^k, and the local index is i_p, the original index i can be obtained by shifting i_p k bits to the left and adding the current PE number. This transformation is shown in figure 15, where "nshift" and "iproc" corresponds to k and *PE number* respectively.

Transforming *ASelect*

As we described earlier, an array must be sliced into many small pieces and each sliced block is allocated to one of the processors. Therefore, we need a mechanism to map an array index to the corresponding PE number and the index of the local array on the designated PE. Accessing an element of the array involves communication between processors (in most cases). The new construct, *pASelect* is introduced for this purpose. Indeed, *pASelect* performs the following two functions:

- To map an array index to the PE number and the index to the local array.
- Retrieve the data and copy to the local memory area.

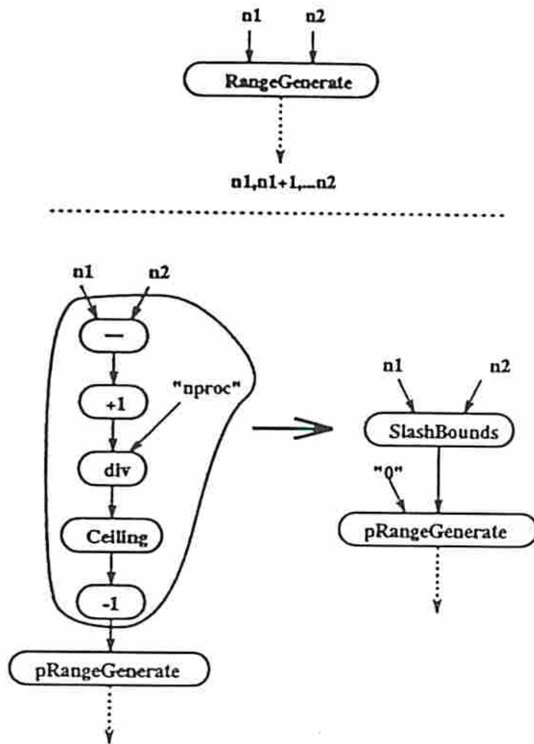


Figure 14: Transforming *RangeGenerate* for a data-parallel forall loop.

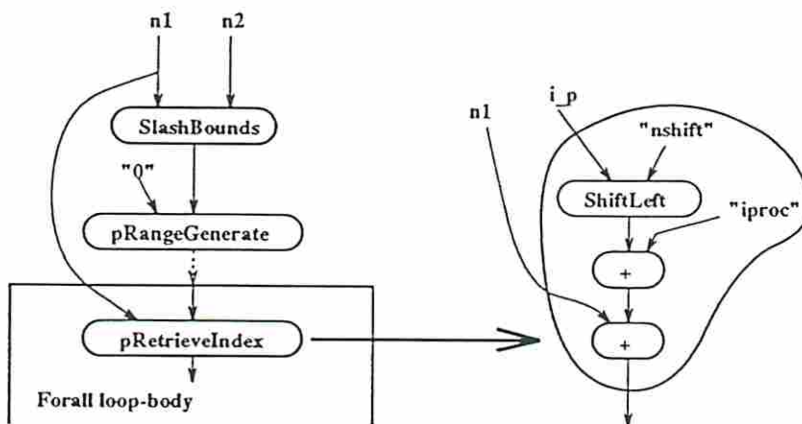


Figure 15: Retrieving the original index from the sliced index at the beginning of the loop-body.

All the *ASelect* nodes in the *forall* loop-body are replaced with *pASelect* constructs. The implementation of *pASelect* is shown in figure 16.

In a way similar to how we implemented the *pRetoreIndex* node, *pASelect* can also be efficiently implemented using bit-wise *shift* and *and* operations. Assuming again that the number of PEs is a power of 2, "nmask" simply becomes "nproc" – 1. As can be implied by the name, the *RouterGet* construct retrieves a data element from the designated PE using *router* communication of MP-1 [12, 13]. The cost of executing the *RouterGet* thus depends on the communication pattern. If more PEs are involved in the random array access, the overall cost of communication becomes significantly larger.

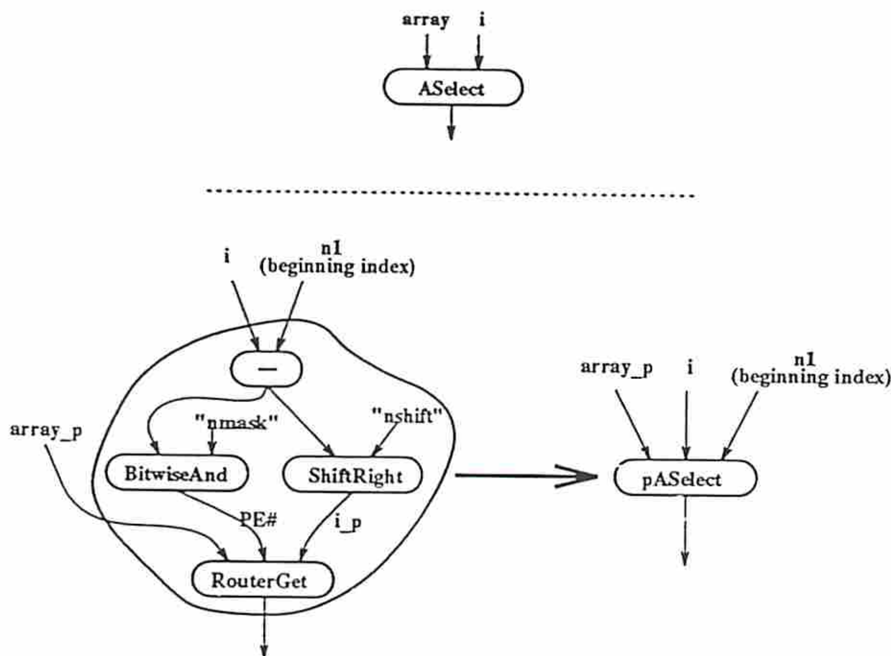


Figure 16: Accessing an array element across the PEs

An Example

By applying the scheme described above, the IF1 graph in figure 12 is transformed as shown in figure 17. In the figure, "v_p" is the sliced array of "v", which is assumed to be already stored in the local memory on each PE.

4.2 Generating MPL code

Generating MPL code from the transformed IF1 graph consists of the following steps:

- Generating the type declaration for each type of IF1 graph.
- Identifying edges which are to be implemented as plural data – In a simplistic way, all the edges in the forall body can be marked as plural data.

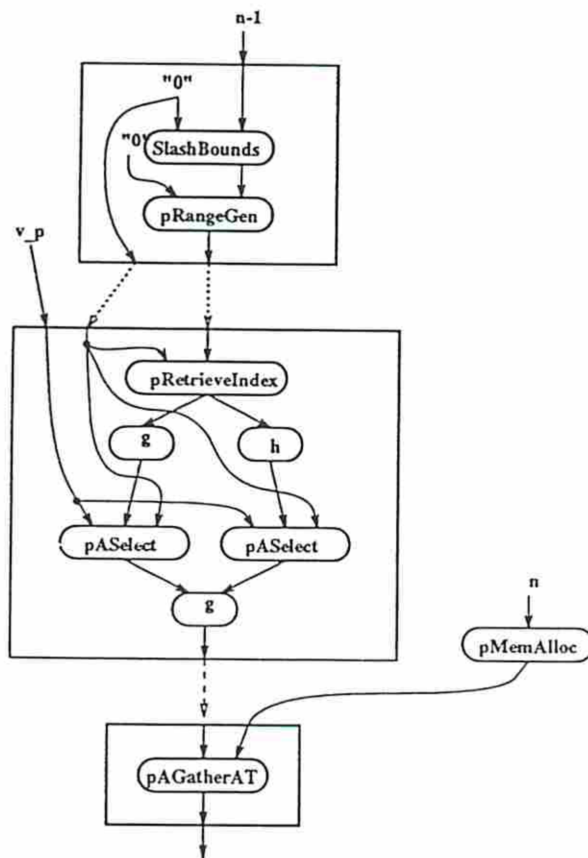


Figure 17: An example of transformed IF1 graph for data-parallel execution.

- Traverse the transformed IF1 graph and generate the appropriate sequence of MPL statements

Besides implementing the above steps, we also need the following for the compilation of the generated MPL code:

- Definitions of the IF1 constructs including the extended ones for data-parallelism in MPL. – This can be accomplished by macros and/or functions.
- Run-time library for the execution of the compiled program. – This implements the memory allocation, input/output, and the machine specific features.

Since the actual implementation of the MPL code generation is beyond the scope of this project, we hand-coded an FFT program based on the proposed translation scheme for the validation of our work. The results are shown in the following section.

5 Implementation of the Target Application

5.1 Description of the Application – Split-Step Algorithm

Most signal processing applications are highly computation-intensive, and in many cases, real-time performance is needed. While the algorithms for solving such applications are well known and understood, their execution is still one or two orders of magnitude slower than what is needed for real-time processing. Our target application is the kernel of electro-magnetic wave propagation modeling [8]. The numerical solution of this application is centered around the *split-step* algorithm in which both forward and backward Fourier transforms are invoked repeatedly.

The *split-step* algorithm which is based on the following two equations is used to compute $u(x, z)$, the power gains (or losses) at (x,z), where x is the horizontal range and z is the vertical range from the source (*e.g.*, an antenna). Forward and backward Fourier transforms are denoted F and F^{-1} respectively.

$$U(x,p) = F[u(x,z)]$$

$$u(x+\delta x, z) = e^{i(k/2)(n'^2-1)\delta x}\, F^{-1}[U(x,p)e^{-i(p^2\delta x/2k)}]$$

where:

$$\begin{aligned}
p &= k\sin\theta \\
\theta &= \text{The angle from the horizontal} \\
k &= \omega\sqrt{\mu\epsilon(a,0)} \\
\epsilon(a,0) &= \text{The permittivity just above the earth's surface} \\
a &= \text{Radius of the earth}
\end{aligned}$$

The split-step algorithm is applied along the horizontal range. Therefore, the dependence between each step of computation is as follows: the label (F or F^{-1}) on the arrow shows the Fourier transform involved at each step.

$$u(0,z) \xleftarrow{F^{-1}} U(0,p) \xrightarrow{F^{-1}} u(\delta x, z) \xrightarrow{F} U(\delta x, p) \xrightarrow{F^{-1}} u(2\delta x, z) \xrightarrow{F} \cdots$$

To begin calculations, we have to find the initial value(s), $U(0,p)$. $U(0,p)$ is obtained by resolving the following equations:

$$\begin{aligned}
U(0,p) &= U_e(0,p) + U_o(0,p) \\
U_e(0,p) &= 2F_c[f(z)\cos p_e z]\cos pz_A - 2iF_s[f(z)\sin p_e z]\sin pz_A \\
U_o(0,p) &= 2F_s[f(z)\sin p_e z]\cos pz_A - 2iF_c[f(z)\cos p_e z]\sin pz_A \\
f(z) &= F_c^{-1}[F_d(p)]
\end{aligned}$$

Where $F_d(p)$ is the antenna pattern and F_c and F_s are cosine and sine transforms. The initial function $U(0,p)$ is obtained by proper modeling of the source which is given by $F_d(p)$ in the above equations. A detailed description of source modeling can be found in [8]. The application kernel and the recursive FFT [9, 15] are described in algorithmic form in figure 18 and figure 19, respectively.

The *split-step* algorithm performs $O(n)$ Fourier transforms. The *FFT* itself has a time complexity $O(n \log n)$ when it is executed sequentially. Therefore the *split-step* algorithm takes $O(mn \log n)$ to complete its computation on a sequential machine, where m is the number of steps and n is the number of sampled points. The *Split-step* algorithm is based on iterating over each horizontal range step. In other words, each step is dependent on the result of the previous step. Therefore, the parallelism of the *split-step* algorithm is limited only by the parallelism which is achieved by the Fourier transform at each step. Thus, even if we increase the problem size with the number of range steps, parallelism does not increase. In other words, speed-up with a large number of processors can be obtained only if we have enough sampled points for the Fourier transform. Therefore, as predicted by *Amdahl's law* [1], the execution time of the algorithm is dominated by the iteration over each range step which is the sequential part of the algorithm. Assuming an infinite number of processors, the *split-step* algorithm takes $O(m \log n)$ time.

5.2 Analysis of the Application using SISAL Tools

Several steps are involved in developing an application in a parallel programming environment. First of all, we have to select (or design) an algorithm which is suitable for parallel implementation. The selected algorithm is then coded in a target language. The next step is to debug and optimize the program using parallel programming tools. These steps are indeed the same which are needed during the development of an application on a sequential machine. However, the appropriateness of the algorithm for parallel execution is mainly determined by the potential parallelism of the algorithm. If the algorithm itself does not have any parallelism, we cannot expect any speedup no matter the number of available processing elements. In this section, we present the simulated performance measures of the algorithm which are obtained using the SISAL programming tools.

Algorithm ***split-step***

Input:

$F_d(p)$: Sampled values which represent the antenna pattern.

n : Number of steps over the horizontal range.

δx : Incremental value for each range step.

k : $\omega\sqrt{\mu\epsilon(a,0)}$ as described above.

n' : Refractive index.

θ_{max} : Maximum angle for which the antenna pattern is sampled.

n_p : Number of sampled points.

Output:

$u(x,z)$: The power gains (or losses) at (x,z)

Begin

1. Find the initial condition $U(0,p)$ from $F_d(p)$.
2. $u(0,z) \leftarrow$ *inverse_fft* $(U(0,p))$.
3. $c_1 \leftarrow e^{i(k/2)(n'^2-1)\delta x}$
4. $x \leftarrow 0, j \leftarrow 1$
5. While $j \leq n$ Repeat

   5–1. $c_2(p) \leftarrow e^{-i(p^2\delta x/2k)}$

   5–2. $u(x+\delta x, z) \leftarrow c_1$ *inverse_fft* $(U(x,p)c_2(p))$

   5–3. $U(x+\delta x, p) \leftarrow$ *fft* $(u(x+\delta x, z))$

   5–4. $x \leftarrow x + \delta x, j \leftarrow j+1$

End

Figure 18: The *split-step* algorithm.

Algorithm *fft*
Input:

f : array of N complex numbers, where N is a power of 2.

Output:

F : array of N complex numbers which is the Fourier transform of f.

Begin

1. Let f^e be the array of even components of f.
   Let f^o be the array of odd components of f.
2. Find F^e = *fft* (f^e) and F^o = *fft* (f^o).
3. Let W be $e^{2\pi i/N}$.
   For $k = 0, \ldots, N/2$, let $L_k = F_k^e + W^k F_k^o$.
   For $k = 0, \ldots, N/2$, let $U_k = F_k^e - W^k F_k^o$.
4. For $k = 0, \ldots, 2/N$, let $F_k = L_k$ and $F_{k+N/2} = U_k$. In other words, F is the concatenation of L and U.

End

Figure 19: The recursive FFT algorithm.

The potential parallelism obtained by the simulation of a *split-step* with 16 sampled points over 16 range steps is shown in Figure 20. As we discussed in the earlier section, the *split-step* executes the loop body sequentially over each range step. Thus, as can be seen in Figure 20, there is little parallelism during the execution of the whole program. However, if we look closely at the graph, we find $2n$ repeated patterns, where n is the number of range steps over which we apply the *split-step* algorithm. Parallel execution of the program, indeed, can reduce the interval between steps, since there is potential parallelism at each step, depending on the number of sampled points for which the Fourier transform is performed. The potential parallelism of a 512 point FFT is shown in Figure 21. The first graph shows the maximum potential parallelism with an infinite number of processors while the second part shows the clipped parallelism with 32 processors.

Running the *split-step* program for various data sizes, 64×32, 64×64, 128×32 and 128×64, where $m \times n$ means m sample points over n split-steps with 1 to 256 processors results in the theoretical speedups shown in Table 1 and Figure 22.

This simulation results confirm that a significant speedup can be obtained by employing a large number of processors only if we have enough sampled points. The fact that the number of steps over the horizontal range cannot increase the potential parallelism is rather disappointing. However, we can still benefit by employing multiple processors, if there are a large number of sampled points compared to the number of available processors. For example, as can be seen in Figure 22, we can achieve a nearly linear speedup up to 32 processors when we have 128 sampled points in an ideal case.

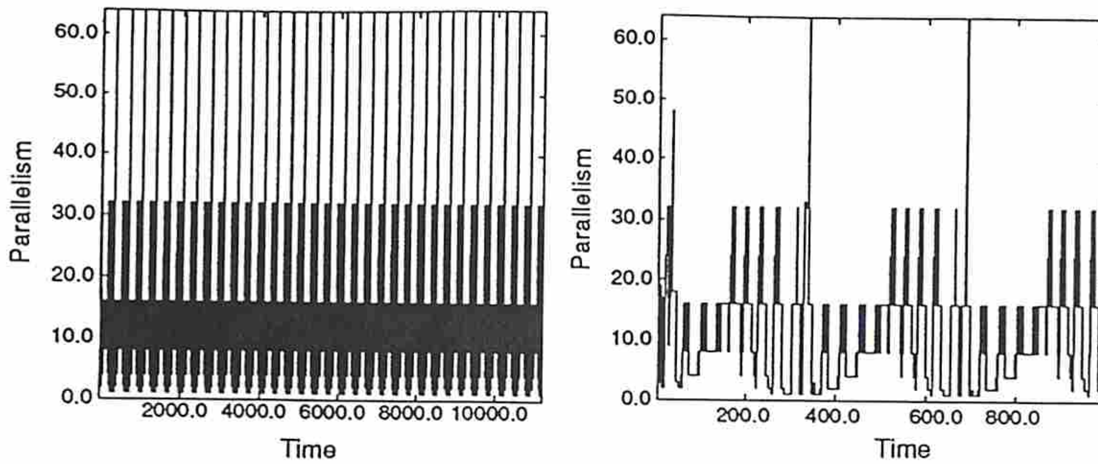
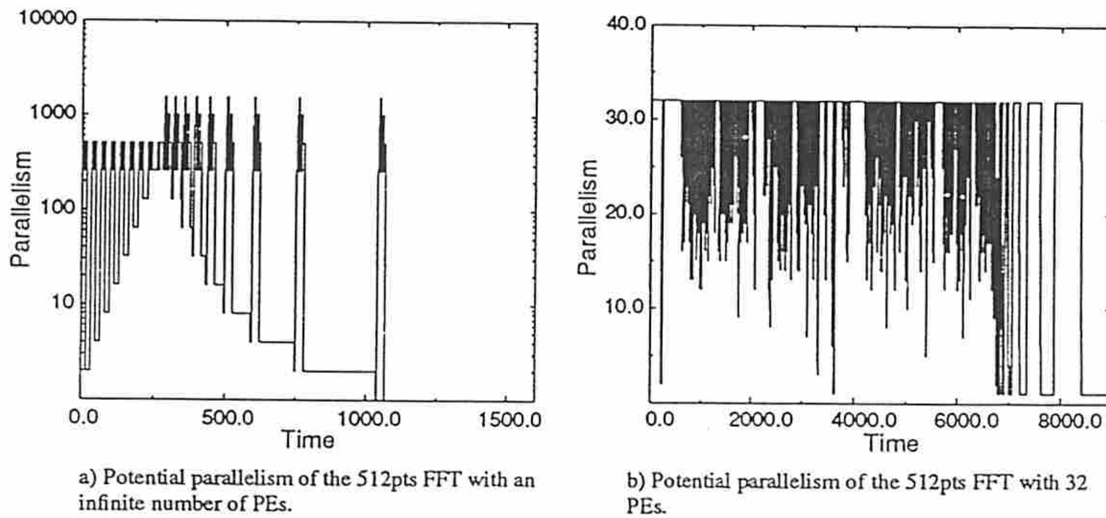


Figure 20: Potential parallelism of the *split-step* algorithm. The histogram shown on the right is the magnified view of the first three steps.



a) Potential parallelism of the 512pts FFT with an infinite number of PEs.

b) Potential parallelism of the 512pts FFT with 32 PEs.

Figure 21: Potential parallelism of FFT

	64×32	64×64	128×32	128×64
4	3.7	3.7	3.8	3.8
8	7.0	7.0	7.2	7.2
16	12.3	12.3	13.3	13.3
32	20.2	20.2	22.8	22.8
64	31.6	31.6	36.6	36.6
128	36.6	36.6	53.3	53.2
256	36.7	36.7	61.3	61.2

Table 1: Speedup of *split-step*.

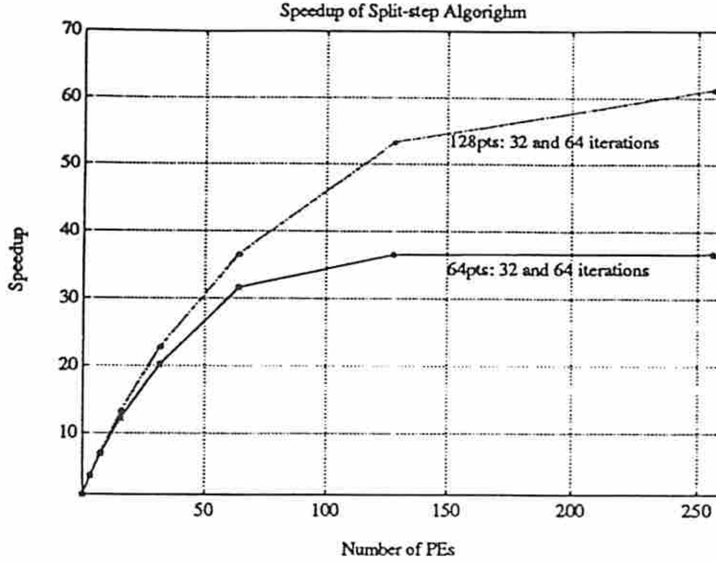


Figure 22: Speedup of *split-step*

5.3 Implementation of of the Application on MP-1

As has been noted in the previous section, most of the parallelism is proportional the number of input points to the FFT function. However, the FFT algorithm in its simplest form which is recursively described (section 5.1) cannot be effectively implemented in parallel in a data-parallel programming paradigm. Hence, we used a different parallel FFT algorithm without any recursion.

<u>Algorithm Description</u>

The algorithm is based on the work shown in [11]. In the algorithm, the array index is calculated to find the *correct* data-dependency between the iterations of FFT. Let $g(l,i)$ be the i_{th} element of g which is an intermediate result after l_{th} iteration, and let $f(i)$ be the i_{th} element of input array of size $N = 2^D$, then $g(l,i)$ can be computed as follows:

$$\begin{aligned} g(0,i) &= f(i) \\ g(l+1,i) &= \begin{cases} g(l,p) + \omega_{2^l}^k g(l,q) & \text{if } 0 \le i < 2/N \\ g(l,p) - \omega_{2^l}^k g(l,q) & \text{if } 2/N \le i < N \end{cases} \end{aligned}$$

where

$$\begin{aligned} k &= \left\lfloor \frac{i}{N/2^{l+1}} \right\rfloor \\ \omega_n^k &= e^{-2k\pi i/n} \\ p &= k\frac{N}{2^l} + mod(i, \frac{N}{2^{l+1}}) \\ q &= p + \frac{N}{2^{l+1}} \end{aligned}$$

The final results are $g(D,i), 0 \le i < N$.

Note that, in the above equation, the computations of $g(l,i)$ for each $i = 0 \ldots N-1$ are independent of each other, and can therefore be computed in parallel. Hence, if we have enough PEs, the time complexity of the algorithm based on the above equation becomes $O(D)$, *i.e.*, $O(\log N)$.

Data Mapping

The data mapping (or data partitioning) is one of the key issues concerning the performance of the program, especially in a data-parallel programming model. When mapping data into the PE array, the goals are to:

1. minimize communications
2. balance the load for greater utilization of PEs
3. keep the algorithm simple.

The first and the second issues are more performance oriented while the last one is oriented toward higher programmability. Obviously, all of the above goals cannot be simultaneously achieved (they are somewhat conflicting). Hence, we may have to give different weight to each goals when designing a data mapping strategy.

For our implementation, we chose to have the simplest data mapping scheme called *1D cut-and-stack data mapping* [12] which makes the algorithm simple. In this mapping scheme, we distribute the element of a data array over the PE array. If the number of data elements are greater than the actual size of the PE array (number of PEs), multiple data elements can be found on the same PE. The main advantage of this mapping scheme is that we can maximize the utilization of the PE array since all the data elements are evenly distributed over the PEs for greater load balancing. The actual mapping of a data element (an index in the data array) to the PEnum (*iproc* in MPL) can vary from application to application. In our implementation, we simply map i_{th} element of a data array to PE_i. Let N be the size of data array A, and z be the local array on each PE which holds a segment of the input array. Also let *nproc* be the number of PEs and *iproc* is the PE number. We first have to find out the size of local array z.

$$zs = \lceil N/nproc \rceil$$

Then for each PE,

$$z[i] = A[i \times nproc + iproc]$$

Conversely,

$$A[i] = proc[mod(i, nproc)].z[\lfloor i/nproc \rfloor]$$

Assuming that the data array is partitioned as described above and the input array is already allocated accordingly, the final data-parallel algorithm of FFT is as shown in Figure 23. On each PE, after the algorithm has been successfully executed, the results will still be stored in the same position as the input data element was stored.

The transformation of the above algorithm for data-parallel execution (step 3 in figure 10) and the MPL code generation (step 4 in figure 10) have been manually done for our experimentation. The SISAL code of FFT and the corresponding IF1 graph are shown in the appendix. The manually coded corresponding MPL program is also shown in the appendix.

At the current state of this work, array access to other PE is implemented by using *router*[4]. The *router* implementation is the simplest way of fetching an array element from other PE, and the performance is also

[4]In our implementation, **rfetch** routine is indeed used instead of **router** statement. – Since the array index is computed in all

for $l = 0 \ldots D-1$ **do**

  for $zi = 0 \ldots zs-1$ **do**

    $i \leftarrow zi \times nproc + iproc$

    $k \leftarrow \left\lfloor \frac{i}{N/2^{l+1}} \right\rfloor$

    $p \leftarrow k\frac{N}{2^l} + mod(i, \frac{N}{2^{l+1}})$

    $q \leftarrow p + \frac{N}{2^{l+1}}$

    $first \leftarrow$ **router**$[mod(p, nproc)].z[\lfloor p/nproc \rfloor]$

    $second \leftarrow$ **router**$[mod(q, nproc)].z[\lfloor q/nproc \rfloor]$

    if $i < N/2$ **then**

      $z[zi] \leftarrow first + \omega_{2^l}^{k} \times second$

    else

      $z[zi] \leftarrow first - \omega_{2^l}^{k} \times second$

    endif

  end for

end for

Figure 23: A data-parallel algorithm for computing FFT.

good when we have rather smaller size of input data. However, as the data-size gets larger, the contention on the router becomes significant and the performance is thus degraded. Further optimization may be possible in both algorithm and in translation phase. However, as shown in table 2 and in figure 24, the performance is scalable in terms of both data-size and the number of PEs.

#PE	128pts	512pts	2048pts	8192pts	32768pts	131072pts
16	0.049	0.219	1.031	4.811	—	—
32	0.028	0.116	0.526	2.438	11.146	—
64	0.021	0.062	0.270	1.236	5.634	—
128	0.017	0.042	0.174	0.786	3.560	15.970
256	0.017	0.032	0.126	0.560	2.522	11.277
1024	0.017	0.030	0.044	0.174	0.748	3.275
4096	0.017	0.030	0.037	0.071	0.280	1.173

Table 2: Execution time of FFT on MP-1

The execution time of the split-step algorithm based on the above FFT implementation is shown in table 3. In figure 25 and figure 26, we compared the execution time of split-step algorithm for various data size. The SISAL code and its hand-translated MPL code are shown in the appendix.

the PEs at the same time, the **router** statement fetches the array element whose index has actually been computed *remotely*. For example, in the statement, **router**$[p].a[i]$, i is a remote value in processor p. Therefore, using the **router** statement, we cannot access the correct remote array element whose index must be computed locally. On the contrary, **rfetch**, which is a variation of the **router** statement, uses a local array index to fetch a remote array element. An example of using **rfetch** is shown in the appendix (MPL code of FFT).

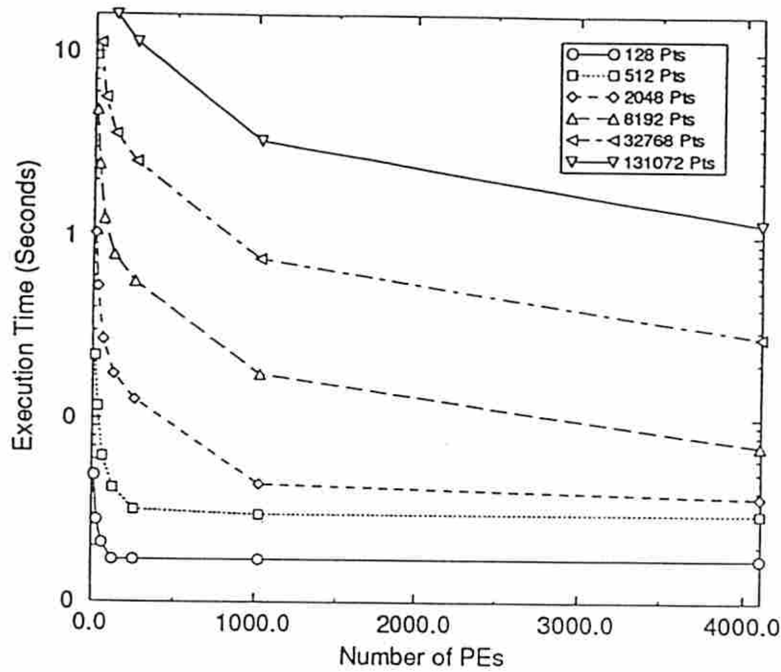


Figure 24: Execution time of FFT on MP-1

#PE	256×64	256×128	1024×64	1024×128	4096×64	4096×128
128	2.176	4.314	10.147	20.165	46.719	92.864
256	1.783	3.535	8.050	16.004	36.874	73.350
512	2.627	5.216	5.525	10.984	24.366	48.480
1024	2.627	5.216	2.944	5.846	12.634	25.132
2048	3.023	6.004	3.841	7.634	7.398	14.712
4096	4.183	8.315	4.842	9.628	5.871	11.677

Table 3: Execution time of Split-Step on MP-1. (On each column $X \times Y$ means Y iterations on X input points.)

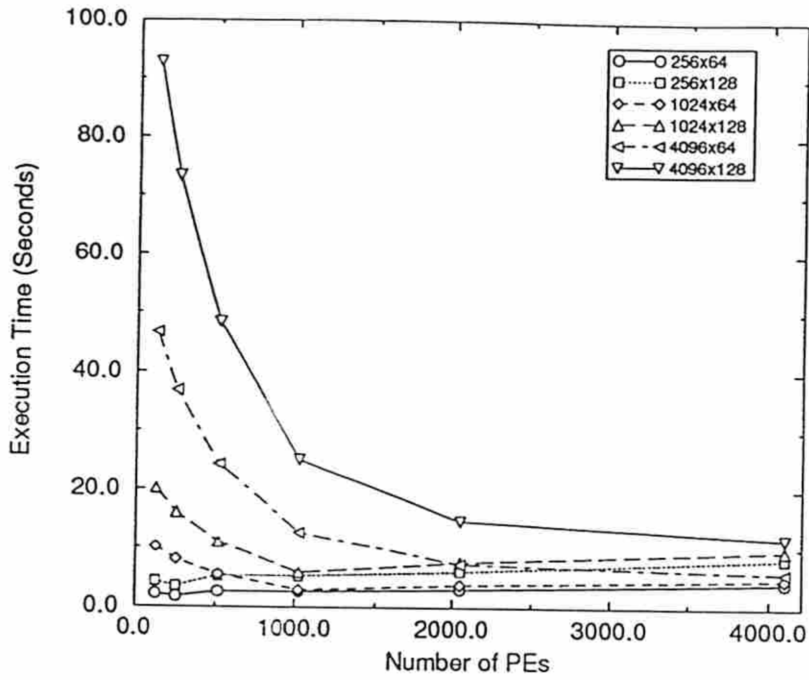


Figure 25: Execution time of FFT on MP-1

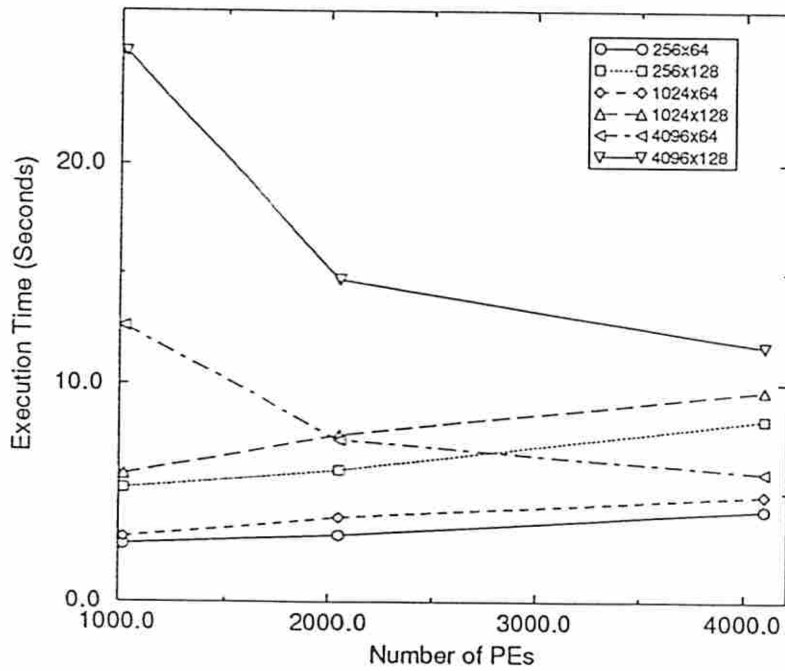


Figure 26: Execution time of FFT on MP-1: A closer look-at when the number of PEs are greater or equal to 1024.

Since all the function calls are *side-effect-free* in SISAL, we can simply replace the function without modifying the rest of the program. There is a set of FFT functions provided by the MasPar's Math Library (MPML). These routines are highly optimized specifically for MP-1 systems. We therefore replaced the FFT (and inverse FFT) function calls with these MPML routines, and showed the execution time in table 4. This demonstrates that any big application can be highly modularized in the SISAL programming environment, and we can indeed reduce the overall development cost.

#PE	256×64	256×128	1024×64	1024×128	4096×64	4096×128
4096	0.292	0.514	0.337	0.604	0.417	0.765

Table 4: Execution time of Split-Step using MPML FFT library. (On each column $X \times Y$ means Y iterations on X input points.)

6 Discussion: SIMD *vs* MIMD

In the previous section, we have shown how a SISAL program can be implemented on the MP-1 (SIMD). Also in the our previous work [18], we presented the result of the parallel implementation of the recursive FFT algorithm on the Sequent Balance (MIMD). In this section, we thus discuss various issues of programming these two types of multiprocessor systems based on our previous work.

The most important advantage of programming in an SIMD machine is that we can exploit *massive parallelism* without paying too much control overhead. This is achieved by the synchronous execution of each instruction in SIMD model. Due to the synchronous execution, debugging can be similarly done as in a sequential machine. However, in the SIMD model, all kinds of sources of parallelism cannot be properly implemented. For example, implicit parallelism, such as double-recursion which is the core of *divide-and-conquer* style algorithm [7], cannot be effectively exploited.

On the other hand, in the MIMD model, exploitation of parallelism is not limited to the synchronous parallel execution of instructions. It is therefore possible to exploit any type of parallelism including the asynchronous parallel constructs which can enable non-strictness of a functional program. However, implementation of general parallelism is achieved at the cost of *synchronization overhead*. In MIMD model, programmers have to take care of synchronization between parallel tasks to obtain the correct results. Implementation of this synchronization may become exponentially complex even when the parallelism increases only linearly. Hence, massive parallelism cannot be effectively handled in an MIMD machine compared to an SIMD machine.

When an application (algorithm) has highly parallel *forall* loops in its kernel and the array access pattern is regular, an SIMD machine would outperform an MIMD machine. Most signal processing applications belong to this type. On the contrary, if the algorithm mainly relies on implicit and/or dynamic parallelism, an MIMD machine could be the only choice for the parallel implementation.

There are also different issues of programming in an SIMD machine and an MIMD machine. For an SIMD machine, the main concern about parallelization is how to distribute array elements over PEs to take

advantage of the regularity of an algorithm. However for an MIMD machine, programmers must pay a great deal of attention to the implementation of synchronization to obtain the correct results and also to reduce the latencies incurred by the synchronization.

A machine-independent higher level language can hide these issues by embedding them in its lower level implementation of the compiler. And the program written such a higher level language can be more portable and the development cycle can thus be reduced. For example, our application language, SISAL, proved to be the right candidate for the parallel development of various types of applications, because:

- There is no machine-dependent notation for parallel execution in the language definition.
- SISAL has already been implemented on many shared memory MIMD machines.
- In this work, we have demonstrated that a SISAL program can be also implemented on the MP-1 (SIMD machine).

The above conditions together with the general side-effect-free nature of functional languages support the SISAL as a strong candidate for the parallel implementation of computation intensive applications.

In table 5 and table 6, we presented the execution time of two different implementations of FFT. The results shown in table 5 are obtained by running *recursive FFT* algorithm on the Sequent Balance shared multiprocessor system. In table 2, we showed the execution time of data-parallel version of FFT algorithm described in section 5.3 on the MP-1. In both implementations, we coded the algorithm in SISAL and the the SISAL code has been translated (manually) for the C-compiler of the target machine. As can be seen on the table, we can exploit reasonable speed up on the MP-1 even when the large number of PEs are employed. However, the recursive FFT algorithm could have not been implemented effectively on the MP-1.

#PE	4096pts	16384pts	32768pts
1	40.0	214.0	501.3
2	20.4	97.0	215.6
3	14.6	66.5	142.6
4	12.6	54.7	115.8
5	10.4	51.5	102.7
6	9.9	42.9	86.7
7	8.7	42.7	82.8
8	8.5	37.1	76.6
9	8.6	33.9	73.9
10	8.0	33.5	70.4
11	8.1	32.7	70.1
12	7.6	30.3	65.7
13	7.6	30.1	65.5
14	7.5	29.4	62.2
15	7.3	29.0	62.5
16	8.4	32.3	58.7

Table 5: Execution time of the recursive FFT on the Sequent Balance.

#PE	128pts	512pts	2048pts	8192pts	3276pts	131072pts
16	0.049	0.219	1.031	4.811	—	—
32	0.028	0.116	0.526	2.438	11.146	—
64	0.021	0.062	0.270	1.236	5.634	—
128	0.017	0.042	0.174	0.786	3.560	15.970
256	0.017	0.032	0.126	0.560	2.522	11.277
1024	0.017	0.030	0.044	0.174	0.748	3.275
4096	0.017	0.030	0.037	0.071	0.280	1.173

Table 6: Execution time of the data-parallel FFT on the MP-1.

7 Conclusion

The major part of this work has been centered around the demonstration of the feasibility of employing data-parallel programming paradigm for the implementation of a signal processing application in the functional programming environment. We specifically selected SISAL as our application language mainly due to its flexibility and availability on many other shared memory multiprocessor systems. The experimental results we have obtained on the MP-1 SIMD machine have been presented together with the results on the Sequent Balance MIMD machine for the comparison between an SIMD and an MIMD machine in terms of both programmability and the performance. The summary of the work we have done in this project is as follows:

- Development of a scheme to translate a SISAL program into MPL.
- Implementation of the split-step algorithm and the FFT algorithm on the MP-1 based on the proposed translation scheme.
- Comparative analysis of SIMD and MIMD machine in terms of both programmability and performance.

As has been noted earlier, the implementation of the translator from SISAL to MPL is beyond the scope of this project. Therefore, we manually applied the proposed translation scheme to implement our application on the MP-1. The implementation of the translator is thus planned for the future work as summarized in the following:

- Extend the translation scheme:
  - Support more SISAL constructs.
  - Improve remote array element access by analysis of the access pattern.
- Implement the translation scheme into the SISAL compiler.
- Perform more benchmarks written in SISAL on both SIMD and MIMD machines.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 483–485, Atlantic City, N.J., April 1967.

[2] J. Backus. Can programming be liberated from the von Neumann style ? *Communications of the ACM*. 21(8):613–641, 1978.

[3] D. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.

[4] D. Cann and J. Feo. Sisal versus FORTRAN: a comparison using the Livermore loops. Technical Report (Unpublished), Lawrence Livermore National Laboratory, 1990.

[5] D. Cann and R. Oldehoeft. A guide to the optimizing Sisal compiler. Technical Report UCRL-MA-108369, Lawrence Livermore National Laboratory, Sep. 1991.

[6] U. Claussen. Parallel subpixel scanconversion. In *Proceedings of the Second Eurographics Workshop on Graphics Hardware, Amsterdam*, Spring 1988.

[7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge. Massachusetts, 1990.

[8] G. Dockery. Modeling electromagnetic wave propagation in the troposphere using the parabolic equation. *IEEE Transactions on Antennas and Propagation*, 36(10):1464–1470, October 1988.

[9] D. Elliott and K. Ramamohan Rao. *Fast transforms: Algorithms, Analyses, Applications*. Academic Press, 1982.

[10] J. Feo and D. Cann. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, pages 349–366, 1990.

[11] T. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

[12] MasPar Computer Corporation. *Data-parallel Programming Guide*, 1991.

[13] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) User Guide*, 1992.

[14] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL : Streams and Iteration in a Single Assignment Language – language reference manual version 1.2. Technical Report TR M-146, Lawrence Livermore Laborartory, March 1985.

[15] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

[16] V. Sarkar and D. Cann. POSC — a partitioning and optimizing Sisal compiler. Technical report, Lawrence Livermore National Lab., 1990.

[17] M. Welcome, S. Skedzielewski, R. Yates, and J. Ranelleti. IF2: An applicative language intermediate form with explicit memory management. University of California Lawrence Livermore National Laboratory, Manual M-195, November 1986.

[18] D-K. Yoon and J-L. Gaudiot. Programming and evaluating the performance of signal processing applications in the sisal programming environment. In *Proceedings of the Second Sisal Users' Conference*, pages 67–82. Lawrence Livermore National Laboratory, 1992.

APPENDIX

A SISAL code of Split-Step

```
%$entry=SplitStep
define SplitStep

%
%              FAST FOURIER TRANSFORM (Recursive)
%

type complex =          record [r, i:double_real];
type complexOneDim =    array [complex];
        % Index ranges from 0 to N

%
%
% Implementation of Split-Step algorithm in SISAL.
%
% In this implementation, U(0,p) is given as an initial value.  U (0,p)
% is actually obtained by the apperture function.  Apperture function is
% again obtained by the antenna pattern.  In the following function, we
% start from this U (0,p) and calculate u(x,z) by iteration.
%
% The solution of this function covers the horizontal range from 0 to
% n x dx.  Other parameters, k and nprime, are just assummed to be a constant
% values in this implementation.
%

function SplitStep (
     Uzero:complexOneDim; % Initial solution
     n:integer;           % Number of iterations.
     dx:double_real;              % incremental range step
     k:double_real;               %
     nprime:double_real;          % refractive index
     thetamax:double_real;        % Maximum angle for which antenna pattern is sampled
     nsample:integer      % number of sampled points
returns
     array [complexOneDim])

     let
          dtheta := thetamax / double_real (nsample - 1);

          % prange is the array of p over which the antenna pattern is sampled
          prange := for i in 0,nsample-1
                    theta := dtheta * double_real (i);
                    p := k * sin (theta);
               returns
                    array of p
               end for;
```

```
          t1 := (k/2.0d)*(nprime*nprime -1.0d)*dx;
          coef1 := record complex [r:cos(t1); i:sin(t1)];
     in
          for initial
               i := 1;
               bigU := Uzero;
               u := ifft (bigU);
          while i <= n
          repeat
               t2 := for j in 0,nsample-1
                    returns
                         array of prange[j]*prange[j]*dx / (k*2.0d)
                    end for;
               coef2 := for j in 0, nsample-1
                    returns
                         array of record complex [r:cos(t2[j]); i: -sin(t2[j])]
                    end for;
               newU := for j in 0, nsample-1
                    returns
                         array of mulc(old bigU [j],coef2[j])
                    end for;

               tu := ifft (newU);
               u := for j in 0,nsample-1
                    returns
                         array of mulc (coef1,tu[j])
                    end for;
               bigU := fft (u);
               i := old i + 1;
          returns
               array of u
          end for
     end let

end function
```

B MPL code of Split-Step

```
#include "common.h"


void SplitStep (bigU, u, zs, n_iter, dx, k, nprime, thetamax, n_points)
  plural complex_t *bigU; /* Input complex numbers: initial solution */
  plural complex_t *u;  /* The solution */
  int   zs;             /* Size of bigU[] and u[] */
  int   n_iter;         /* Number of iterations */
  int   n_points;       /* Number of sampled points */
  real  dx;             /* Incremental range step */
  real  k;              /* */
  real  nprime;         /* Refractive index */
```

```
  real  thetamax;        /* Maximum angle for which antenna patter is sampled */
{

  int depth;                   /* log(n_points) */
  plural int **j1;             /* List of j1 values. size is zs*depth  */
  plural int **j2;             /* List of j2 values */
  plural complex_t **zomega;   /* List of omega values */

  /* Misc. scratch arrays and variables */
  int i;
  real t1;
  plural real p_t2;
  plural complex_t *work1;
  plural complex_t *work2;
  plural int p_i,p_idx;

  /* constants which computed once in the beginning of the function */
  plural real *prange;
  real dtheta;
  complex_t coef1;

  /***************************/
  /* Allocate scratch arrays */
  /***************************/

  prange = (plural real *) p_malloc (sizeof(real)*zs);
  work1 = (plural complex_t *) p_malloc (sizeof(complex_t)*zs);
  work2 = (plural complex_t *) p_malloc (sizeof(complex_t)*zs);


  /*************************************/
  /* Allocate various tables for FFT  */
  /*************************************/

  depth = ilog2(n_points);

  j1 = (plural int **) p_malloc (sizeof(int *)*zs); /* Index lookup tbl 1 */
  j2 = (plural int **) p_malloc (sizeof(int *)*zs); /* Index lookup tbl 2 */
  zomega = (plural complex_t **)
     p_malloc (sizeof(complex_t *)*zs); /* Table of roots on unity */

  for (p_i=0;p_i<zs;p_i++) {
     j1[p_i] = (plural int * plural) p_malloc (sizeof(int)*depth);
     j2[p_i] = (plural int * plural) p_malloc (sizeof(int)*depth);
     zomega[p_i] = (plural complex_t * plural)
        p_malloc (sizeof(complex_t)*depth);
  }


  /* Calculate the intervals over which data are sampled */

  dtheta = thetamax / ((real) (n_points - 1));
  for (p_i=0; p_i<zs;p_i++) {
     plural real p_theta;
```

```
      p_idx = (p_i<<MemShift) + iproc;
      p_theta = dtheta * (plural real) p_idx;
      prange[p_i] = k  * fp_sin (p_theta);
   }
   t1 = (k/2.0)*(nprime*nprime - 1.0)*dx;
   makec (f_cos(t1), f_sin(t1), coef1);


   /* Prepare for the FFTs */
   fft_init (j1,j2,zomega,zs,depth, n_points);

   /* perform initial fft (inverse fft ) */

   for (p_i=0;p_i < zs ; p_i++) {
      copyc (bigU[p_i], u[p_i]);
   }

   /* The actual solution of SplitStep() is the *array* of "u", i.e., */
   /* the solution is two dimensional array which contains all the values */
   /* over the iteration steps.  For the sake of simplicity and the memory */
   /* consideration, we only returns the last element of array of "u". */
   ifft (u,work1,j1,j2,zomega,zs,depth,n_points); /* the first solution */


   /*********************************/
   /* Now we begin the main  loop  */
   /*********************************/

   for (i=0;i<n_iter;i++) {

      for (p_i=0;p_i<zs;p_i++) {
         p_t2 = prange[p_i] * prange[p_i] * dx / (k*2.0);
         makec (fp_cos (p_t2), -fp_sin (p_t2), work1[p_i]);
         mulc (bigU[p_i], work1[p_i], work2[p_i]);
      }

      ifft (work2,work1,j1,j2,zomega,zs,depth,n_points);

      for (p_i=0;p_i<zs;p_i++) {
         mulc (coef1, work2[p_i], u[p_i]);
      }

      for (p_i=0;p_i<zs;p_i++) {
         copyc (u[p_i], bigU[p_i]);
      }

      fft (bigU,work1,j1,j2,zomega,zs,depth,n_points);
   }
}
```

C SISAL code of FFT

```
%$entry=fft
define fft

%
%                FAST FOURIER TRANSFORM (Non-Recursive Version)
%

type complex =          record [r, i:double_real];
type complexOneDim =    array [complex];
        % Index ranges from 0 to N

global sin(x : double_real returns double_real)
global cos(x : double_real returns double_real)

function addc(a, b:complex returns complex)
     record complex [ r:a.r+b.r ; i:a.i+b.i ]
end function % addc

function subc(a, b:complex returns complex)
     record complex [ r:a.r-b.r ; i:a.i-b.i ]
end function % subc

function mulc(a, b:complex returns complex)
     record complex [ r:a.r*b.r-a.i*b.i ; i:a.i*b.r+a.r*b.i ]
end function % mulc

function makec(a, b:double_real returns complex)
     record complex [r:a; i:b]
end function

function ilog2(n:integer returns integer)
     for initial
          x := -1;
          k := n;
     while k > 0
     repeat
          k := old k / 2;
          x := old x + 1;
     returns
          value of x
     end for
end function

%
% fft : non-recursive fast fourier transform.
%
% input
%        v : Array of Complex values
%
% output
%        fft of v
%
```

```
function fft(v : complexOneDim returns complexOneDim)
    let
        PI := 3.1415926536D;
        twoPI := 2.0D*PI;
        n := array_size (v);
        half_n := n/2;
        depth := ilog2 (n);
    in
        for initial
            res := v;
            l := 0;
            n1 := 2;
            n2 := n/2;
            n3 := n;
        while l < depth
        repeat
            theta := twoPI / double_real (old n1);

            res := for j3 in 0,n-1
                    t3 := if j3 < half_n then j3 else j3 - half_n end if;
                    k := t3 / old n2;
                    i := mod (t3, old n2);
                    j1 := k * old n3 + i;
                    j2 := j1 + old n2;

                    theta_k := theta * double_real (k);

                    omega := makec (cos (theta_k),-sin (theta_k));
                    omega_f := mulc (omega, old res[j2]);
                    myres := if j3 < half_n then
                            addc (old res[j1], omega_f)
                        else
                            subc (old res[j1], omega_f)
                        end if
                returns
                    array of myres
                end for;
            n1 := old n1 * 2;
            n2 := old n2 / 2;
            n3 := old n3 / 2;
            l := old l + 1;
        returns
            value of res
        end for
    end let
end function
```

D IF1 Graph of FFT

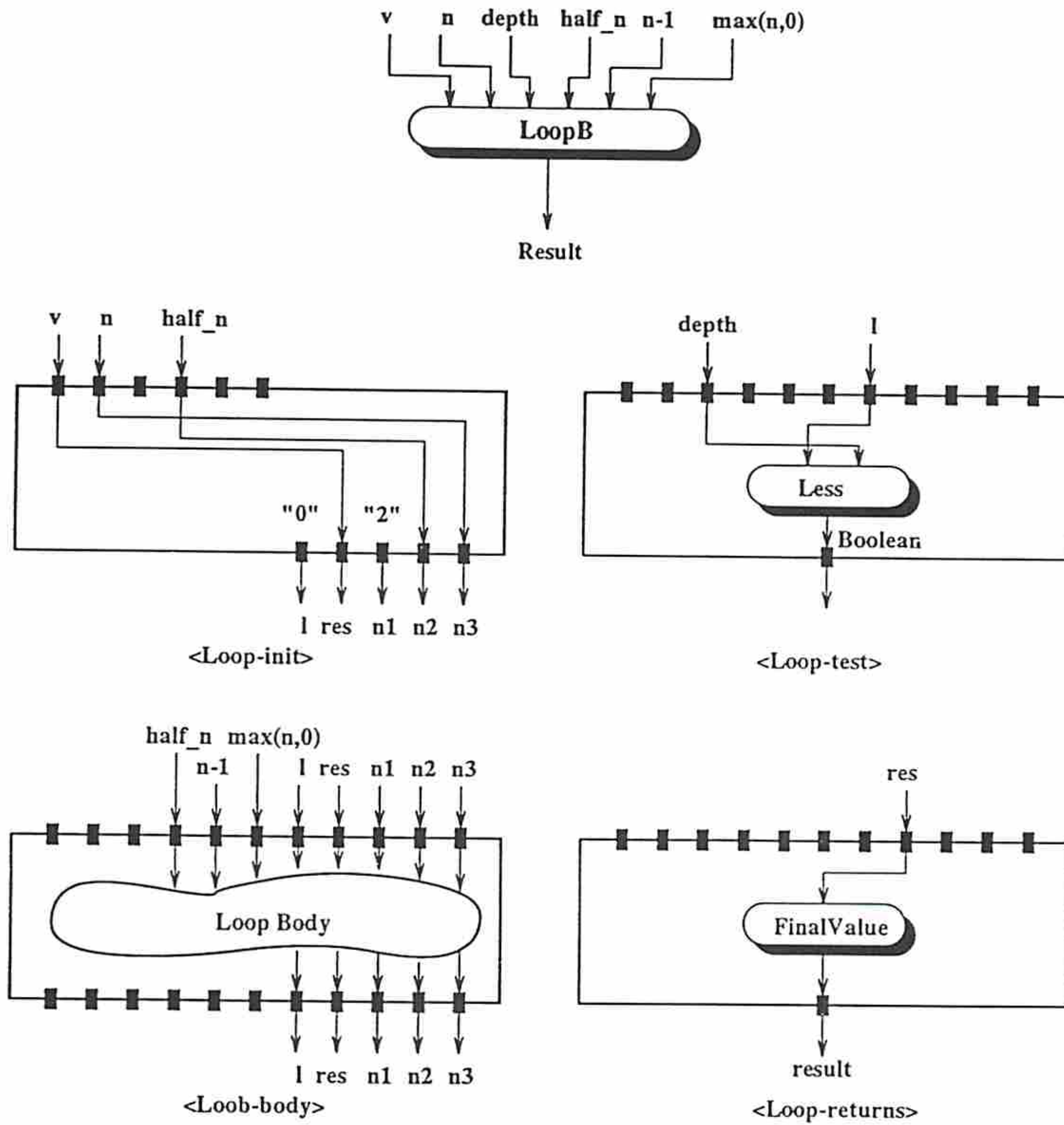


Figure 27: IF1 graph of LoopB.

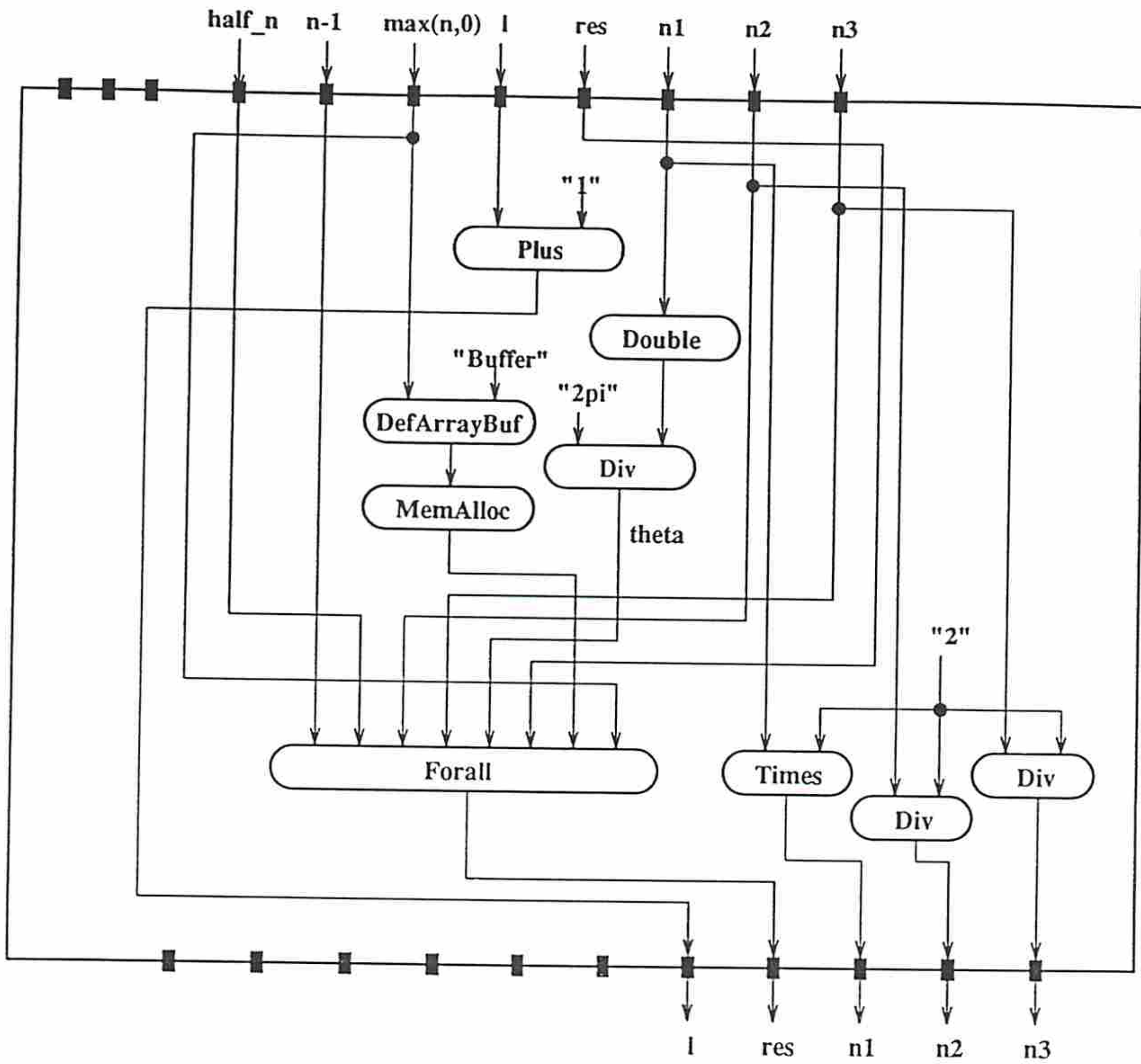


Figure 28: IF1 graph of LoopB-body.

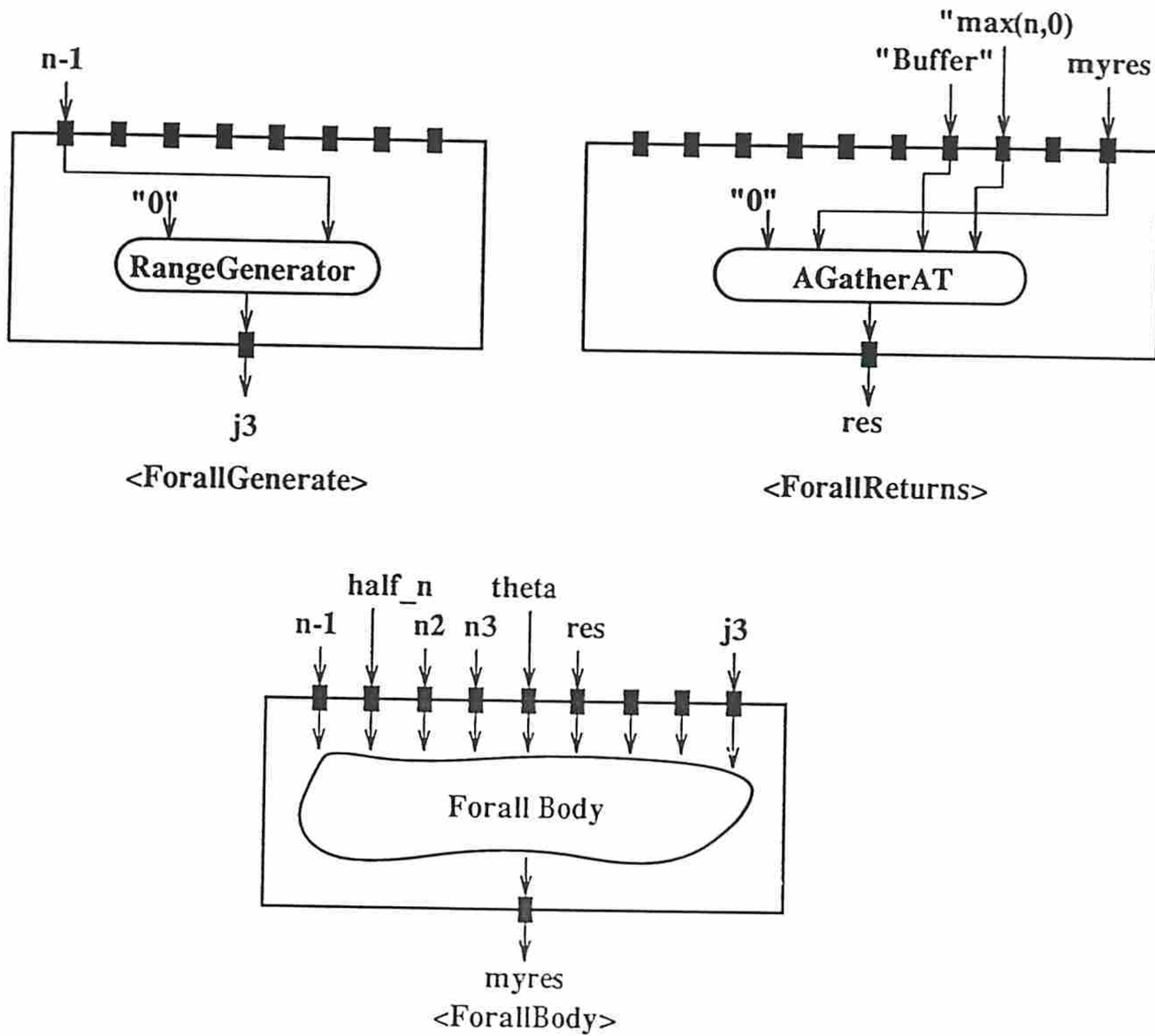


Figure 29: IF1 graph of Forall.

E MPL code of FFT

```
main()
{
    ...
    nmask = NumPEs-1;
    nshift = ilog2(NumPEs);
    ...

    /* Prepare input on all the PEs */
    zs = (int) ceil ((double) Asize / (double) NumPEs);
    z = (plural complex_t *) p_malloc (sizeof(complex_t)*zs); /*Input*/
    newz = (plural complex_t *) p_malloc (sizeof(complex_t)*zs);/*Output*/
    for (p_i=0;p_i<zs;p_i++) {
       plural int p_idx;

       p_idx = (p_i<<nshift) + iproc;
       if (p_idx == 0 || p_idx >= (Asize-1))
           makec (0.0,0.0,z[p_i]);
       else
           makec (1.0,0.0,z[p_i]);
    }

    ...
    fft (z, newz, Asize, zs); /* Call fft */
    ...

}


void fft (a,b,n,zs)
  plural complex_t *a,*b;
  int n,zs;
{
   plural complex_t p_omega,p_omega_f,p_lc,p_rc;
   plural real p_t,p_theta_k;
   plural int p_k,p_i,p_zi;
   plural int p_j1,p_j2,p_j3; /* f(j3,l+1) = f(j1,l) +(-) w*f(j2,l) */

   real twoPI,theta;
   int half_n = n/2;
   int depth,l; /* depth log2(n) */
   int n1,n2,n3; /* 2^x, used in various ways. See below */

   twoPI = PI*2.0;
   depth = ilog2(n);
   n1 = 2; /* 2^(l+1) */
   n2 = n/2; /* N/(2^(l+1)) */
   n3 = n; /* N/(2^l) */

   for (l=0;l<depth;l++){
      theta = twoPI / n1; /* for w(2^(l+1),x) */
```

```
        for (p_zi=0;p_zi<zs;p_zi++){
    plural int p_tj3;

    p_j3 = (p_zi<<MemShift) + iproc;
     /* p_j3 the index this PE is procsessing */
    if (p_j3 < half_n)
       p_tj3 = p_j3;
    else
       p_tj3 = p_j3 - half_n;

    p_k = p_tj3/n2;
    p_i = p_tj3%n2;
    p_j1 = p_k*n3 + p_i;
    p_j2 = p_j1 + n2;

    /* use "rfetch" for remote array access */
    ps_rfetch (p_j1&ProcMask, (plural char * plural)&(a[p_j1>>MemShift]),
       (plural char *)&p_lc, sizeof (complex_t));
    ps_rfetch (p_j2&ProcMask, (plural char * plural)&(a[p_j2>>MemShift]),
       (plural char *)&p_rc, sizeof (complex_t));

    p_theta_k = theta * p_k;
    makec (p_cos(p_theta_k), -p_sin(p_theta_k), p_omega);

    mulc (p_omega, p_rc, p_omega_f);
    if (p_j3 < half_n)
       addc (p_lc,p_omega_f,b[p_zi]);
    else
       subc (p_lc,p_omega_f,b[p_zi]);
         }

         n1 *= 2;
         n2 /= 2;
         n3 /= 2;

         for (p_zi=0;p_zi<zs;p_zi++)/* copy back the results for next iteration */
   copyc (b[p_zi],a[p_zi]);
     }
}
```