PIL API
v0.4

Adam R. Smith, Chih-Chieh Yang,
David Padua

Department of Computer Science
University of Illinois
Urbana, IL 61801

May 23, 2013

Contents

1 PIL API **4**
1.1 Introduction . . . 4
1.1.1 Supported Runtimes . . . 4
1.1.2 Using the PIL compiler . . . 4
1.2 Control Flow . . . 5
1.2.1 Fundamental Parallel Operations . . . 5
1.2.2 Task Graphs . . . 6
1.3 Data Management in PIL . . . 7
1.3.1 Data Availability . . . 7
1.3.2 Primitive Types . . . 7
1.3.3 Data Acquisition . . . 7
1.4 Writing Libraries in PIL . . . 11
1.5 Arrays in PIL . . . 12
1.5.1 Array Declaration and Allocation . . . 12
1.5.2 Array Deallocation . . . 13
1.5.3 Array Accesses . . . 14
1.5.4 Array Dimension Indices . . . 15
1.5.5 High Level Array Operations . . . 16
1.6 Code Examples . . . 18
1.6.1 Hello World . . . 18
1.6.2 Parallel Hello World . . . 18
1.6.3 Matrix-Matrix Multiply . . . 19
1.6.4 PIL Library Example . . . 20

2 SPIL: Syntatic Sugar for PIL **22**
2.1 Overview . . . 22
2.2 SPIL Control Constructs . . . 22
2.2.1 Sequential Functions . . . 22
2.2.2 Control Constructs . . . 22
2.2.3 Compound Parallel Operations . . . 24
2.3 SPIL Arrays and Tiling . . . 24
2.3.1 Data Layout . . . 24
2.4 SPIL Task Graphs . . . 24

2.5 Code Examples 25
2.5.1 Hello World 25
2.5.2 Parallel Hello World 25
2.5.3 Array Example 25
2.5.4 Parallel Tiled Matrix-Matrix Multiplication 26

3 SPMD PIL **27**
3.1 Introduction 27
3.2 Program Execution 27
3.3 SPMD PIL Code Examples 30
3.3.1 Barrier Synchronization 30
3.3.2 Send/Receive 31

Index **32**

Chapter 1

PIL API

1.1 Introduction

The Parallel Intermediate Language (PIL) is intended to be used as the target of parallel compilers for any parallel programming language or the implementation language for any high level API, and it is expected that efficient code could be generated from it for any parallel runtime. We call this the any-to-any implementation goal. Once a program is represented in PIL, the user can choose which parallel runtime to generate code for. This document describes the PIL API that can be the target of a compiler of a parallel programming language, or can be handwritten by a user.

1.1.1 Supported Runtimes

PIL can generate code for a variety of runtimes. Currently, PIL can target sequential C (C), C with parallel OpenMP annotations (OMP), the SWARM Codelet Association Language (SCALE), and the Open Community Runtime (OCR).

1.1.2 Using the PIL compiler

The PIL compiler takes as input a source file written in the PIL programming language and generates code for the specified runtime which is output to `stdout`. PIL can generate code for any of the backends described in Section 1.1.1 by specifying which backend to generate code for with the -o option. An example to generate SCALE code can be seen here.

```
> pilc -o scale src.pil >out.swc
```

By default, if no runtime is specified with the -o option, sequential C will be assumed.

1.2 Control Flow

A PIL node can be created with the following line.

```
node(label, index, [lower:step:upper], target,
     [label1, label2, ..., labelN], func(arg1, arg2, ..., argN))
```

Each node must have a unique label. In PIL each node is labeled with a unique integer chosen by the user. The label 0 is a special case that means `exit` and can only be used as the target of a node.

1.2.1 Fundamental Parallel Operations

Forall

A forall loop is the easiest parallel construct to use in PIL. Each node in a PIL graph is a forall loop. If the user wants to create a loop with only one iteration for sequential execution, then there will only be one PIL node created for that loop and it will execute sequentially.

Flattened Nested Forall This will allow the use of more than one index variable in the parallel loop. The user will needs to tell us how many loops there will be with the num_loops variable. Then they will provide one index variable, range pair for each of the loops. The general form is as follows.

```
node(label, num_loops, index1, index2, index3, [lower1:step1:upper1],
     [lower2:step2:upper2], [lower3:step3:upper3], target,
     [label1, label2, ..., label3], func(arg1, arg2, ..., arg3))
```

The Following example will create 2 unique values for i, 3 unique values for j, and 4 unique values for k for a total of $2 \times 3 \times 4 = 24$ node instances each with a different combination of i, j, and k.

```
// pseudo code
forall i in [0..1] do
    forall j in [0..2] do
        forall k in [0..3] do
            func(&target, i, j, k)

// pil code
node(1, 3, i, j, k, [0:1:1], [0:1:2], [0:1:3], target, [0],
     func(&target, i, j, k))
```

Reductions

See Section 1.5

```
1  // global data
2  int x;
3  float y;
4  index_t i;
5  index_t j;
6  pil_target_t target;

8  // functions
9  void a(int *target) {...}
10 void b(int *target) {...}
11 void c(int *target, float y) {...}
12 void d(int *target) {...}
13 void e(int *target) {...}

15 // PIL nodes
16 node(1, i, [l:s:u], target, [1, 2, 3], a(&target))
17 node(2, j, [l:s:u], target, [2, 3], b(&target, x))
18 node(3, i, [l:s:u], target, [4], c(&target, y))
19 node(4, i, [l:s:u], target, [5, 2], d(&target))
20 node(5, i, [l:s:u], target, [0], e(&target))
```

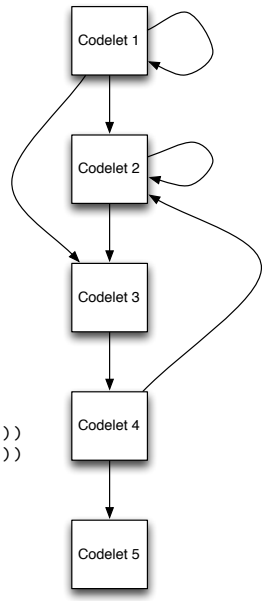

Figure 1.1: A sample PIL program and its task graph.

1.2.2 Task Graphs

A task graph can easily be constructed by traversing at the targets of each node. For example the task graph in Figure 1.1 corresponds to the code in Figure 1.1. In that example, it is easy to see that there are three outgoing edges from node 1 that go to nodes 1, 2, and 3. Outgoing edges can be determined for each of the nodes in the graph. Currently, the first node specified in the file is the entry node.

1.3 Data Management in PIL

All variables that are used as parameters in PIL nodes must be declared in the "global" data section. All variables declared here will be available to the user when their body function that uses the variable is executed.

1.3.1 Data Availability

Since PIL code is agnostic to the underlying runtime, the compiler must determine which variables should be passed between nodes. This is achieved by looking at the parameters of the functions associated with each node. PIL will ensure that the data is available when the node executes. For example, the user may want to generate code for a distributed memory machine. If the variable they wish to access is stored in a different address space from the one their node is currently running in they may not be able to access it. PIL will ensure that the data is reachable in the node either automatically by the compiler or manually by the user as described in the remainder of this section.

1.3.2 Primitive Types

PIL supports all of the primitive data types in C including, but not limited to float, double, long double, signed/unsigned char, and signed/unsigned short/-long/long long int. PIL also supports pointers to any of the primitive types. However, PIL only manages access to the pointer itself and not the data it points to. For example, if a PIL node uses a pointer `void *p` in a function, PIL will only ensure that the pointer `p` is available for use, and not the data that it points to, since it has no knowledge of the number of bytes pointed to by the pointer. Any non-primitive data that is allocated in PIL should be managed with the memory operations in Section 1.3.3. Since manual data movement can be cumbersome for users, we provide one of the most commonly used structures, the Array, and provide ways to have PIL perform automatic data movement of this structure. The built-in structure `Array` and its associated routines are described in Section 1.5 and automatic data movement is discussed in Section 1.3.3.

1.3.3 Data Acquisition

Since data needs to be made available before it can be used in a node, it can be acquired in one of two ways: it can be acquired automatically by PIL or manually by the user. We expect that most algorithms can be formed in a way that allows the use of automatic movment of arrays. However, if the user should desire the lower level and more involved manual data movement operations, we provide those as well.

Automatic Data Movement

Data movement can be handled automatically for the primitive data types in C as described in Section 1.3.2 or for the built-in array structure described

in Section 1.5. The automatic data movment is implemented in terms of the manual data movement operations described later. To provide automatic data movement, the user can use the keywords in Table 1.1.

Table 1.1: Data movement keywords.

Keyword	Description
`in`	Consumed by the node.
`out`	Produced by the node.
`inout`	Consumed, changed, and reproduced by the node.

A simple example node using these keywords can be seen here.

```
node(1, i, [1:1:1], target, [0], f(out &target, in i))
```

The `in` directive tells the PIL compiler that the variable will be needed by the current node, but will not be forwarded to the next node that is fired after this one. The `out` directive tells the PIL compiler that the value *will* be needed by the next node and the compiler should ensure that the data is available to the next node. The `inout` is a combination of the `in` and `out` where the value is used as input in the current node, modified in some way, and then needs to be forwarded to the next node.

In this example, the variable `target` is an `out` variable since it will be produced by the node. The target variable is a special case in PIL where it is an output of a node, but it is not necessarily and input to the next node. However, it is needed by the PIL runtime to determine which node to fire next, and so is output from this node to the runtime. In general, the target of a node should usually be an `out` variable. The variable `i` is an `in` variable since it is needed by the node to perform some computation. Since it is only used as input, PIL does not need to forward to the next node.

Further examples of how to use these keywords can be found in Section 1.6.

Manual Data Movement

The automatic management of data movement described in Section 1.3.3 is provided to the user as a convenience and for syntactic sugar. If the user wishes to do data movement manually because they wish to have their own complex data structures or for any other reason they may do so. The routines in Table 1.2 are provided to the user for manual data movement.

pil_alloc. Memory allocation is performed with the `pil_alloc` routine. This is very similar to the `malloc` routine in the C language in that it takes as input the number of bytes required by the user and allocates a chunk of contiguous memory of that size. However, there is one very important distinction. The return value of the routine is an ID. IDs are unique identifiers for a piece of memory. Each allocation of a memory block will be assigned a new and unique ID.

Table 1.2: Reduction Dimensions

Routine	Description
`int pil_alloc(size_t NumBytes)`	Allocate memory
`void *pil_mem(int id)`	Acquire a pointer to memory
`void pil_release(int id)`	Notify runtime you are done accessing memory
`void pil_free(int id)`	Deallocate memory

pil_mem. IDs are needed in case the data the user wants to access is migrated in between node executions by the runtime, since pointers by themselves contain no information about the size of the data being pointed to. Once a user has acquired a pointer to a memory block, they can be sure that the pointer will be valid throughout the lifetime of the node the pointer was acquired in. However, once the node ends, the pointer may become invalid if the data the pointer points to is moved by the runtime. This means that all pointers must be assumed to be invalid when a node begins to execute and that the user will need to use the ID associated with the data block to acquire a valid pointer to the data. This can be achieved with the `pil_mem` routine. The user provides as input a valid ID and the runtime will return to the user a pointer to the block associated with that ID. The runtime also registers the block of memory as in use by the current node to ensure that the data is not moved while the node is executing.

pil_release. Before a node finishes, it must release its associations will all acquired memory blocks. This can be done with the `pil_release` routine.

pil_free. When a user wishes to deallocate data and free it from the heap, they may do so with the `pil_free` routine. This routine takes as input the ID of the memory block the user wants to deallocate.

Data serialization. Since pointers must be assumed to be invalidated when crossing node boundaries, complex pointer based structures like the arrays describe in Section 1.5 must have each their pointers revalidated every time the data is passed to a new node. This means that every field of a struct and every pointer used must be passed as a separate input to nodes where they are used. It is convenient for the user to create a serialization and deserialization routine for any complex data structure they wish to create. Serialization routines turn a complex data structure into a single contiguous memory block to be accessed with a single pointer. This means that the data can all be passed as a single memory block between PIL nodes, with a single `pil_mem` and `pil_release` call for that block. As soon as the user acquires a pointer to the serialized structure, they may call the deserialize routine to rebuild the structure to a more useful state. Then, before the end of the node the user can serialize the data and pass it to the next node. These serialization and deserialization routines can greatly simplify a user's code.

All of the operations of acquiring and releasing memory blocks and serializing and deserializing data structures can be performed automatically for supported data types as described in Section 1.3.3.

1.4 Writing Libraries in PIL

We provide a way for a user to write libraries in PIL. Each library operation in PIL is relatively small collection of PIL nodes. These nodes work together to perform some operation. We refer to a collection of PIL nodes that performs a library operation as a PIL library routine.

In order to use PIL library routines, the user will use the `pil_main` routine to write their program. An example of how the control flow through a PIL library routine can be seen in Figure 1.2. The `pil_main` routine is the body of a special PIL node. Each statement in this routine is executed sequentially until the `pil_enter` routine is encountered. At this point the PIL node associated with the `pil_main` routine suspends execution and the node that is specified with the `pil_enter` routine begins execution. The nodes associated with the PIL library routine are executed. At the completion of the last node in the library routine, control is passed back to the `pil_main` function and executions continues as normal.

If the user wishes to use the library functionality in PIL, their program will need to be structured slightly differently from a regular PIL program. First, by creating a `pil_main` function in a user's code, the user is letting the compiler know that they will be using PIL library routines. This means that the execution of the users program will begin with the execution of the `pil_main` routine and the only way to execute other PIL nodes is with calls to the bulk synchronous PIL library routines. Secondly, the node 0 no longer exits the program. Instead, it is used in the PIL library routine when the library routine wants to finish. It can be thought of as a `return` statement in a C function.

Once the user has their library operation written as a PIL library routine, they may write programs that use that routine many times. This can be best used when providing a user with a prewritten package of library routines.

An example of a PIL program that uses PIL library routines can be seen in Section 1.6.4.

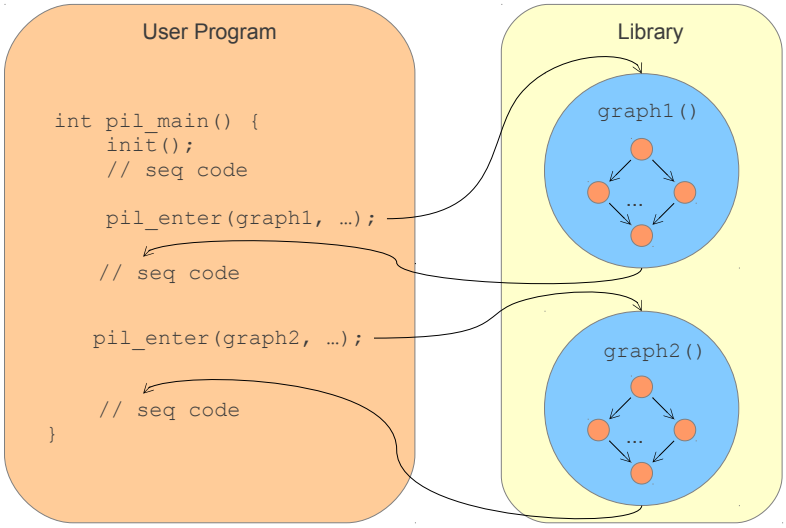

Figure 1.2: Control flow when using `pil_enter`.

1.5 Arrays in PIL

In order to handle the complex data structure of arrays, we have chosen to provide the user with a built-in array structure. Arrays in PIL are N-dimensional arrays that are made up of tiles.

1.5.1 Array Declaration and Allocation

To declare an array, the user will have to specify the number of dimensions of the array, the size of each dimension, and the size of each tile. The `new_array` function will allocate the required space for the array and return an initialized Array structure. The user must also specify how to store the data as described below.

```
Array A1 = new_array(ROW, 1, X, N);
Array A2 = new_array(TILE, 2, X, Y, N, M);
Array A3 = new_array(SPARSE, 3, X, Y, Z, N, M, L);
```

Array Layout. Data in dense arrays are stored in either row-major or tile-major data layout. Examples of these layout schemes can be seen in Figure 1.3.

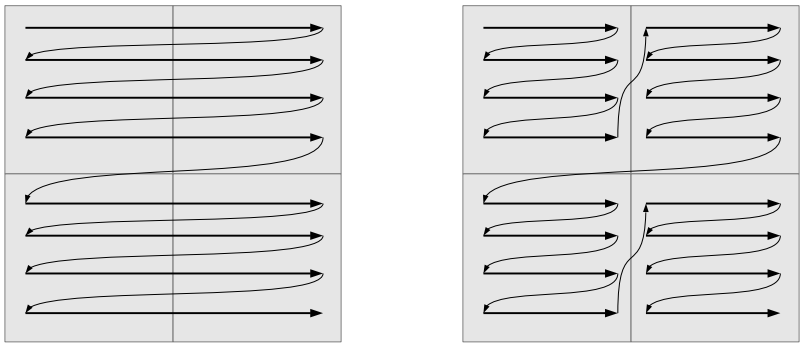

Figure 1.3: Row-major and Tile-major data layout.

If the array is in row-major layout, each element across every row of every tile in the array will be contiguous in memory. In other words, the entire array is laid out such that it is in row-major order. The tiles, however, will have consecutive rows stored in noncontiguous locations in memory. If the array is in tile-major layout then every tile will be stored in row-major layout such that the entire tile is in one contiguous memory block. The tiles themselves will also be laid out in row-major order such that neighboring tiles will be in consecutive memory blocks. However, in this layout scheme, if you access every element in a row, the elements in each tile will be contiguous, but at the tile boundaries, the first element of the next tile will be the start of new memory block. Alternatively, arrays can be stored as sparse arrays. The keywords for specifying how to store an array can be seen in Table 1.3

Table 1.3: Array Storage Schemes

Keyword	**Storage Scheme**
ROW	Row-major
TILE	Tile-major
SPARSE	Sparse

1.5.2 Array Deallocation

To release the memory used by an array the user will call the `free_array` routine.

```
free_array(A1);
```

1.5.3 Array Accesses

The user can use the `get_tile` routine to get access to a specific tile in an array. For example, to get the tile in Figure 1.4 that contains the red element, the user would want to get tile [2,2]. The `get_tile` routine expects the number of dimensions of the tile as well as the tile indices.

```
Tile get_tile(Array A, 1, int x);
Tile get_tile(Array A, 2, int x, int y);
Tile get_tile(Array A, 3, int x, int y, int z);
Tile T2 = get_tile(A2, 2, 2, 2);
```

In order to find a specific element in an array we will provide macros to access elements. Macros are needed to perform this action since the arrays may be stored in one of the dense forms or sparse forms from Section 1.5.1. Macros are used to alleviate the overhead of requiring functions to access elements. Elements in an array can be accessed with a tile coordinate and indices in the tile pair or as an absolute index into an array. For example the red element of array C in Figure 1.4 can be accessed in either of the two ways below.

C{1,1}(1,1) C(1,1)	C{1,1}(1,2) C(1,2)	C{1,1}(1,3) C(1,3)	C{1,1}(1,4) C(1,4)	C{1,2}(1,1) C(1,5)	C{1,2}(1,2) C(1,6)	C{1,2}(1,3) C(1,7)	C{1,2}(1,4) C(1,8)
C{1,1}(2,1) C(2,1)	C{1,1}(2,2) C(2,2)	C{1,1}(2,3) C(2,3)	C{1,1}(2,4) C(2,4)	C{1,2}(2,1) C(2,5)	C{1,2}(2,2) C(2,6)	C{1,2}(2,3) C(2,7)	C{1,2}(2,4) C(2,8)
C{1,1}(3,1) C(3,1)	C{1,1}(3,2) C(3,2)	C{1,1}(3,3) C(3,3)	C{1,1}(3,4) C(3,4)	C{1,2}(3,1) C(3,5)	C{1,2}(3,2) C(3,6)	C{1,2}(3,3) C(3,7)	C{1,2}(3,4) C(3,8)
C{1,1}(4,1) C(4,1)	C{1,1}(4,2) C(4,2)	C{1,1}(4,3) C(4,3)	C{1,1}(4,4) C(4,4)	C{1,2}(4,1) C(4,5)	C{1,2}(4,2) C(4,6)	C{1,2}(4,3) C(4,7)	C{1,2}(4,4) C(4,8)
C{2,1}(1,1) C(5,1)	C{2,1}(1,2) C(5,2)	C{2,1}(1,3) C(5,3)	C{2,1}(1,4) C(5,4)	C{2,2}(1,1) C(5,5)	C{2,2}(1,2) C(5,6)	C{2,2}(1,3) C(5,7)	C{2,2}(1,4) C(5,8)
C{2,1}(2,1) C(6,1)	C{2,1}(2,2) C(6,2)	C{2,1}(2,3) C(6,3)	C{2,1}(2,4) C(6,4)	C{2,2}(2,1) C(6,5)	C{2,2}(2,2) C(6,6)	C{2,2}(2,3) C(6,7)	C{2,2}(2,4) C(6,8)
C{2,1}(3,1) C(7,1)	C{2,1}(3,2) C(7,2)	C{2,1}(3,3) C(7,3)	C{2,1}(3,4) C(7,4)	C{2,2}(3,1) C(7,5)	C{2,2}(3,2) C(7,6)	C{2,2}(3,3) C(7,7)	C{2,2}(3,4) C(7,8)
C{2,1}(4,1) C(8,1)	C{2,1}(4,2) C(8,2)	C{2,1}(4,3) C(8,3)	C{2,1}(4,4) C(8,4)	C{2,2}(4,1) C(8,5)	C{2,2}(4,2) C(8,6)	C{2,2}(4,3) C(8,7)	C{2,2}(4,4) C(8,8)

Figure 1.4: The two different ways to access elements in an 8x8 array with 2x2 tiles.

```
int get_element(Array A, int numDim, Idx i, Idx j, ..., Idx N);
int c = get_element(C, 2, 5, 5);
```

```
// in Array C with 2 dimensions, access in tile [2,2] element [1,1]
int c = get_element_tiled(C, 2, 2, 2, 1, 1);
```

Similarly, we provide macros to write to an array.

```
int write_element(Array A, float val, int numDim, Idx i, Idx j, ..., Idx N);
write_element(C, c, 2, 5, 5);

// in Array C with 2 dimensions, write the value c
// in tile [2,2] element [1,1]
write_element_tiled(C, c, 2, 2, 2, 1, 1);
```

Dense array optimization.

The ways to access elements in an array described above will work for an array with any layout scheme. However, due to the overhead and verbosity of these functions, we expect their usage to be limited mostly to sparse arrays. We also provide macros to turn array indices in dense arrays into the explicit index in the array storage. We provide these macros for two and three dimensions.

```
int Index2D(Array A, Idx i, Idx j);
int Index3D(Array A, Idx i, Idx j, Idx k);
int IndexTiled2D(Array A, TIdx x, TIdx y, Idx i, Idx j);
int IndexTiled3D(Array A, TIdx x, TIdx y, TIdx z, Idx i, Idx j, Idx k);
```

To access the element at location `A[i,j]` the user can write the following code.

```
A.data[Index2D(A, i, j)]
```

The array A must be passed to the macro `Index2D` so the macro can make the appropriate calculation based off of the data layout of array A, which is a filed in the array structure. The return value of the macro is the index into the one dimensional array storage field `data`.

Examples on how to use these macros can be seen in Section 1.6.

1.5.4 Array Dimension Indices

Table 1.4 shows the translation from a dimension index to the corresponding dimension. In this document whenever accessing an array with an index variable, we always use the same index variable with the same dimension. For example, we will always use `i` when accessing the X dimension. In some instances it is appropriate to specify either a negative or positive array index value. For example, in the `circshift` routine described in Section 1.5.5, array values can be shifted to the right by one by specifying a circular shift with the array dimension 1. Similarly, the values could be shifted to the left by one by specifying a circular shift with the array dimension -1.

Table 1.4: Array Dimension Indices

Number	Dimension	Index
0	X	i
1	Y	j
2	Z	k

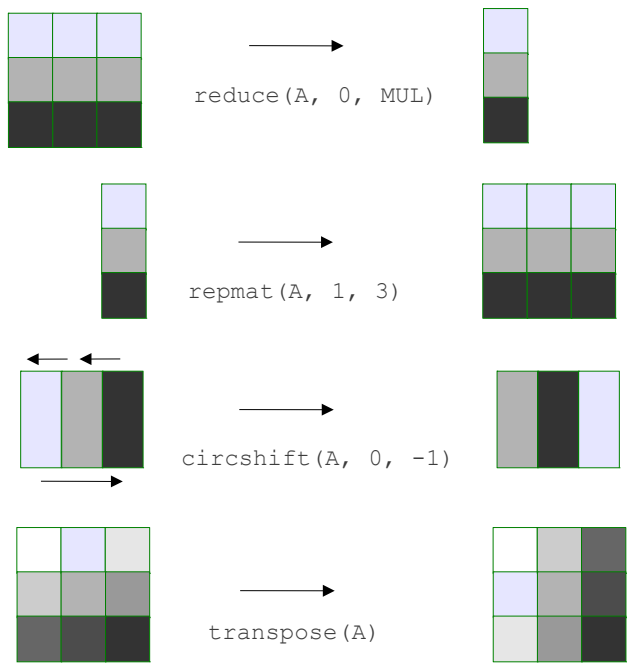

Figure 1.5: Array operations

1.5.5 High Level Array Operations

The operations on an array in Figure 1.5 will be provided along with PIL as a collection of PIL library routines. If the user writes code with the library style of PIL, they will have access to them by default. They are expressed to the user as a regular C function, but underneath will be implemented as PIL library operations.

Array Serialization

In order to facilitate the transfer of data across nodes, all data must be serialized. We provide built-in routines for the serialization and deserialization of arrays in PIL. The `serialize_array` routine takes as input an `Array` and produces a pointer to a serialized buffer that contains the contents of the array. The

`deserialize_array` routine takes as input a pointer to a buffer that contains a serialized array and returns an `Array` constructed from the buffer.

```
void *serialize_array(Array);
Array deserialize_array(void *);
```

The user should never have to explicitly call these routines, but they should know that they exist, and, more importantly, that all data in PIL must be serializable to be transferred between nodes. If the user should wish to create their own data structures, they will have to create their own serialization and deserialization routines for that data structure. PIL automatically takes care of the serialization, deserialization and data transfer of arrays as described in Section 1.3.3. The manual movement of data is discussed in Section 1.3.3.

1.6 Code Examples

1.6.1 Hello World

```
# global data
int i;
int target;

# body functions
void f(int *target)
{
    printf("Hello World!\n");
    *target = 0;
}

# pil nodes
node(1, i, [1:1:1], target, [0], f(&target))
```

1.6.2 Parallel Hello World

```
# global data
int i;
int target;

# body functions
void f(int *target, int i)
{
    printf("Hello from node %d!\n", i);
    *target = 0;
}

# pil nodes
node(1, i, [0:1:9], target, [0], f(&target, i))
```

1.6.3 Matrix-Matrix Multiply

```
int i;
int X;
int Y;
int target;
int argc;
char **argv;
Array A;
Array B;
Array C;

void setup(int *target, int argc, char **argv, int *X, int *Y,
           Array *A, Array *B, Array *C) {
    int Z;

    // read command line paramaters for X, Y, Z
    read_command_line(argv, argc, X, Y, &Z);

    // generate data for A, B. Set C to 0s.
    initialize_arrays(A, B, C, *X, *Y, Z);

    *target = 2;
}

void multiply(int *target, int i, int j, Array *A, Array *B, Array *C) {
    int k;

    for (k = 0; k < A->numcols; k++) {
        C->data[Index2D(C,i,j)] += A->data[Index2D(A,i,j)] * B->data[Index2D(B,i,j)];
    }

    *target = 3;
}

void cleanup(int *target, Array *C) {
    /* use result in C */
    *target = 0;
}

node(1, i, [1:1:1], target, [2], setup(out &target, in argc,
     in argv, out &X, out &Y,, out &A, out &B, out &C));
node(2, 2, i, j, [0:1:X], [0:1:Y], target, [3],
     multiply(out &target, in i, in j, in &A, in &B, inout &C));
node(3, i, [1:1:1], target, [0], cleanup(out &target, in &C));
```

1.6.4 PIL Library Example

In this example we create three library functions in PIL. Each is a simple collection of two PIL nodes. While this example was kept simple to be contained in this document, it is easy to see how having a collection of several PIL nodes into a library function could be implemented.

```
#define ADD 10
#define MMM 20
#define INIT 30

int i;
int X;
int Y;
int target;
Array A;
Array B;
Array C;
int val;

void add_enter(int *target, int *X, int *Y, Array *A) {
    *X = A->numrows;
    *Y = A->numcols;
    *target = 11;
}

void add(int *target, int i, int j, Array *A, Array *B, Array *C) {
    C->data[Index2D(C,i,j)] = A->data[Index2D(A,i,j)] + B->data[Index2D(B,i,j)];
    *target = 0;
}

void mmm_enter(int *target, int *X, int *Y, Array *A) {
    *X = A->numrows;
    *Y = A->numcols;
    *target = 21;
}

void multiply(int *target, int i, int j, Array *A, Array *B, Array *C) {
    int k;

    for (k = 0; k < A->numcols; k++) {
        C->data[Index2D(C,i,j)] += A->data[Index2D(A,i,j)] * B->data[Index2D(B,i,j)];
    }

    *target = 0;
}

void init_enter(int *target, int *X, int *Y, Array *A) {
    *X = A->numrows;
    *Y = A->numcols;
    *target = 31;
}

void init(int *target, int i, int j, Array *A, int val) {
    A->data[Index2d(A,i,j)] = val;
    *target = 0;
}
```

```
node(10, i, [1:1:1], target, [11],
     add_enter(out &target, out &X, out &Y, in &A));
node(11, 2, i, j, [0:1:X], [0:1:Y],
     add(out &target, in i, in j, in &A, in &B, out &C);
node(20, i, [1:1:1], target, [21],
     mmm_enter(out &target, out &X, out &Y, in &A));
node(21, 2, i, j, [0:1:X], [0:1:Y], target, [0],
     multiply(out &target, in i, in j, in &A, in &B, out &C));
node(30, i, [1:1:1], target, [31],
     init_enter(out &target, out &X, out &Y, in &A));
node(31, 2, i, j, [0:1:X], [0:1:Y], target, [0],
     init(out &target, in i, in j, in &A, in val));

void pil_main() {
    A = new_array(2, 10, 10);
    B = new_array(2, 10, 10);
    C = new_array(2, 10, 10);

    // note: sequential code is allowed between each
    // pil_enter call if desired
    pil_enter(INIT, 2, A, 1); // set A to all 1s
    pil_enter(INIT, 2, B, 2); // set B to all 2s
    pil_enter(INIT, 2, C, 0); // set C to all 0s

    pil_enter(MMM, 3, C, A, B); // C = A * B
    pil_enter(ADD, 3, B, C, A); // B = C + A
}
```

Chapter 2

SPIL: Syntatic Sugar for PIL

2.1 Overview

In this chapter we describe a high level language for parallel computation called Structured PIL (SPIL) . This high level language has a source-to-source compiler that generates PIL code. Once the PIL code has been generated, it can then be targeted to any of the runtimes that PIL supports as described in Chapter 1.

2.2 SPIL Control Constructs

In general, programs in SPIL are written as two sections. The first is a collection of functions that perform the work of the program. The second is a recipe for how the functions should be executed with various control constructs.

2.2.1 Sequential Functions

All functions in SPIL are to be written as valid sequential C functions. There will be one special function called `spil_main` that will contain all parallel constructs.

2.2.2 Control Constructs

Sequential Execution

The user can write sequential code in SPIL as either a single function or a sequence of functions. By default, if no information is given at the end of execution of a function, the next function in the program is executed.

```
func1();
func2();
func3();
```

Conditional Statements

If the user wants to conditionally execute code to specify which function should be executed next instead of the next function in the program, the user can use the `case` statement.

```
case f()
    0: g()
    1: h()
```

In this example, the function `f` will return a value that specifies which of the cases to return next. If `g` should be executed, `f` should return 0. If `h` should be executed, `f` should return 1. With this construct the user can have an arbitrary number of options for functions to execute next. The number for cases should begin at 0 and proceed incrementally.

Loops

Loop ranges. A user can specify a range of values with a triplet `l:s:u` . In the example below, `l` is the lower bound, `u` is the upper bound, and `s` is the step for the loop bounds. All three values are required for a loop range. The step can be negative if desired.

While loops. The user can write loops in SPIL with a `while` loop. The condition function `c` must return an integer. A return value of 0 will fail the condition check and the loop body will not be executed. A nonzero value will succeed and the loop body will execute. The body of the loop is a collection of statements that will execute in order. Upon completion of the body, the condition function is executed again and the return value checked.

```
while c() {
    func1();
    forall i in 0:1:N-1 do func2();
    func3();
}
```

For loops. The user can write loops in SPIL as with a `for` loop. For loops are similar to while loops, but an explicit loop range is specified for a loop index variable. The index variable can be used in the loop body.

```
for j in 0:1:N-1 {
    func1();
    forall i in 0:1:j do func2();
    func3();
}
```

Parallel Forall Loops

Parallel execution of functions can be achieved with the parallel `forall` loop. Forall loops can only have a single function as the body of the loop. Any necessary arguments must be passed to the function as parameters. There will be a single function implemented for each `forall` loop body.

```
forall i in l:s:u do func();
```

2.2.3 Compound Parallel Operations

Compound parallel constructs such as reduce, repmat, circshift, and transpose will not be primitive operations in SPIL, but can easily be constructed with a combination of constructs in Section 2.2.2. These operations are provided as library routines that can be called in a users program.

2.3 SPIL Arrays and Tiling

The only data structure in SPIL are the `Array` and the `Tile` . Arrays in SPIL will be constructed as a collection of tiles.

```
// one dimensional tile with 10 elements
Tile T1 = new_tile([10]);

// two dimensional tile with 4x4 elements
Tile T2 = new_tile([4, 4]);

// 2x2 tiles of 4x4 elements each stored in row-major layout
Array A2 = new_array([2, 2], [4, 4], ROW);
```

2.3.1 Data Layout

Data layout of Arrays in Tiles in SPIL is achieved through the formats supported in PIL. Please refer to Section 1.5.1 for more information.

2.4 SPIL Task Graphs

Should the user find that these new syntactic constructs do not fit their needs for any reason, they may insert a section of PIL code into their SPIL code. This is achieved by allowing the use of the `pil_enter` routine to call PIL library routines. This allows existing PIL library routines to be called from SPIL and arbitrarily complex task graphs to be created in SPIL. Please refer to Section 1.2.2 for more information about task graphs and Section 1.4 for more information on writing libraries in PIL.

2.5 Code Examples

2.5.1 Hello World

```
void hello() {
    printf("Hello World!\n");
}

void spil_main() {
    hello();
}
```

2.5.2 Parallel Hello World

```
void hello(int i) {
    printf("Hello from %d!\n", i);
}

void spil_main() {
    forall i in 0:1:9 do hello(i);
}
```

2.5.3 Array Example

```
void init(int *N, Array *A2) {
    // initialize N and A2
}

// There are N instance of work() called. One for each value of i.
void work(Array A2, int i) {
    // perform computation on A2
}

void func3(Array A2) {
    // use result in A2
}

void spil_main() {
    int N;
    Array A2 = new_array([2, 2], [4, 4], ROW);
    init(&N, &A2);
    forall i in 0:1:N-1 do work(A2, i);
    finalize(A2);
}
```

2.5.4 Parallel Tiled Matrix-Matrix Multiplication

```
// This is here for completion. It could be replaced with a call to MKL
// or some other efficient GEMM.
void gemm(Tile c, Tile a, Tile b, int TS) {
    int i, j, k;
    for (i = 0; i < TS; i++)
        for (j = 0; j < TS; i++)
            for (k = 0; k < TS; k++)
                c[i,j] += a[i,k] * b[k,j];
}

// There is one instance of matmul for each thread/codelet that iterates
// through the tiles to calculate the tile C[i,j] they are assigned.
void matmul(Array C, Array A, Array B, int i, int j, int NT, int TS) {
    int k;
    c = get_tile(C, i, j);
    for (k = 0; k < NT; k++) {
        a = get_tile(A, i, k);
        b = get_tile(B, k, j);
        gemm(c, a, b, TS);
    }
}

void spil_main() {
    int TS = TILE_SIZE;
    int NT = NUM_TILES;
    Array A = new_array([NT, NT], [TS, TS], TILE);
    Array B = new_array([NT, NT], [TS, TS], TILE);
    Array C = new_array([NT, NT], [TS, TS], TILE);

    // initialize A, B. Set C to 0.

    forall i in 0:1:NT-1, j in 0:1:NT-1 do matmul(C, A, B, i, j, NT, TS);
}
```

Chapter 3

SPMD PIL

3.1 Introduction

SPMD PIL is an extension of PIL to support programming in distributed memory environment. In this programming model, all Processing Elements (PE) execute the same code independently, and collaboration among PEs is achieved by programmers explicitly specifying where to communicate and synchronize. PEs manage computational data locally, and programmers are responsible for writing the algorithm in a way that each execution instance is able to derive information about the source and destination.

Section 3.2 describes the execution flow of SPMD PIL and the new keywords and API functions. Section 3.3 are short examples of how SPMD PIL applications can be written.

3.2 Program Execution

The typical execution flow of SPMD PIL programs starts from `_pil_main()` function. All PEs in the system execute independently before a call to enter some PIL node is made, which transfers the control flow to the PIL node selected. All PEs keep running independently while executing a PIL node, until communication APIs are called or encountering the function exit.

In the case of communication, PIL node execution continues without blocking until it exits the function, where the execution is suspended. The completion of communication will trigger the PE to execute the continuation function. At this point, data is received in buffer and computation that depends on it can be performed. For barrier synchronization, the execution continues when all PEs' executions reach the barrier.

```
int i;
int target_id;
_pil_node foo void f(int *target_id) {
    _pil_context int data;
    /* Perform computation */
    pil_recv(src, &data, sizeof(int), recv);
}

_pil_nodelet foo void recv(int *target_id) {
    /* Perform computation after receiving data */
}

node(1, i, [0:1:2], target_id, [0], foo(&target_id))

void pil_main(int argc, char ** argv) {
    pil_enter(1, 0);
}
```

Figure 3.1: Short example of SPMD PIL.

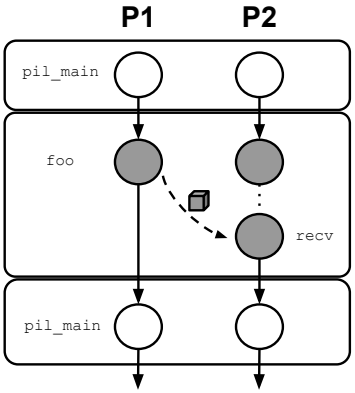

Figure 3.2: Execution flow of SPMD PIL with communication among PEs.

Figure 3.1 is a pseudo code example, and Figure 3.2 illustrates the execution flow. In this example, both PEs execute `_pil_main()` in the beginning, and then enters PIL node 1, which invokes function `foo` marked with keyword `_pil_node`. This keyword is newly added for indicating the functions used in SPMD mode. In a call to a communication or synchronization API function, programmer can optionally mark a function with keyword `_pil_nodelet` for invocation when the corresponding communication function is completed (in this case, the `recv` function) to continue unfinished computation tasks. Control is transferred back to `pil_main()` when `foo` and its pairing nodelet function `recv` both finish.

In a `_pil_node` function, programmers can add a modifier `_pil_context` at the beginning of a variable definition to tell PIL compiler that the variable belongs to program context that has to be passed to nodelet functions. Variables without this modifier are treated as ordinary stack variables, and their lifetime

ends when `_pil_node` function exits, while variables stored in program context are accessible in the corresponding `_pil_nodelet` function.

The following is the description of new communication and synchronization APIs added for SPMD PIL.

pil_get_nwID. By calling this function, an execution instance can get an integer ID to identify which PE it runs on. Different decisions can be made according to the ID. For example, consider performing a circular shift operation on a distributed array in the system. Each execution instance can set the destination ID to be an incremented value of its own ID.

pil_barrier_all. This is an implementation of a global barrier among all PEs. All the execution instances have to reach the barrier before any of them continues. Programmers may specify a continuation nodelet function to deal with the work that needs to be done after the barrier. Currently, this is the only way to explicit synchronize. We are still evaluating whether there is a need to add a synchronization function that is able to block only those PEs involved.

pil_send. The function, when invoked, copies a memory object and schedules a receiving codelet at the destination PE. If the code written in the same `_pil_node` function after the line of `pil_send()` modifies the memory object sent, the changes will not be propagated to the receiver.

pil_recv. This function is used to indicate the PE expects to receive a memory object from some source PE in the system. A nodelet function has to be specified as the continuation of execution, and `_pil_context` variables can be used in its scope. No changes should be made to `_pil_context` variables or the memory buffer allocated for receiving the data from remote PE, since the changes are not thread-safe.

3.3 SPMD PIL Code Examples

3.3.1 Barrier Synchronization

```
// global data
int i;
int target_id;

_pil_node f void f(int *target_id)
{
    fprintf(stdout, "%d: before barrier\n", pil_get_nwID());
    fflush(stdout);
    pil_barrier_all(bar);
}

// continuation
_pil_nodelet f void bar(int *target_id)
{
    fprintf(stdout, "%d: after barrier barrier\n", pil_get_nwID());
    fflush(stdout);
    *target_id = 0;
}

// node declaration
node(1, i, [0:1:9], target_id, [0], f(&target_id))

void pil_main(int argc, char **argv)
{
    pil_enter(1, 0);
    if (pil_get_nwID() == 0)
        printf("SUCCESS");
}
```

3.3.2 Send/Receive

```
// global data
int i;
int target_id;

_pil_node f void f(int *target_id)
{
    _pil_context int num;
    _pil_context size_t size;
    int target = 1;
    int src = 0;
    size = sizeof(int);
    if (pil_get_nwID() == 0)
    {
        num = 42;
        printf("%d: sending %d to node %d\n", pil_get_nwID(), num, target); fflush(stdout);
        pil_send(target, &num, size, recv);
    }
    else if (pil_get_nwID() == 1)
    {
        pil_recv(src, &num, size, recv);
    }
    *target_id = 0;
}

// continuation
_pil_nodelet f void recv(int *target_id)
{
    // _pil_context int num inherited from above
    // _pil_context size_t size inherited from above
    fprintf(stdout, "%d: num = %d\n", pil_get_nwID(), num); fflush(stdout);
}

// node declaration
node(1, i, [0:1:9], target_id, [0], f(&target_id))

void pil_main(int argc, char **argv)
{
    pil_enter(1, 0);
    if (pil_get_nwID() == 0)
        printf("SUCCESS");
}
```

Index

Array, 7, 24

case, 23
Compiler, 4

for, 23
forall, 5, 24
    flattened nested, 5

in, 8
inout, 8

loop range, 23

node, 5

out, 8

PIL, 4
    Compiler, 4
    Examples, 18
pil_alloc, 8
pil_enter, 11, 24
pil_free, 9
pil_main, 11
pil_mem, 9
pil_release, 9

SPIL, 22
    Examples, 25
spil_main, 22
SPMD PIL, 27
    _pil_context, 28
    _pil_node, 28
    _pil_nodelet, 28
    barrier, 27
    communication, 27
    Examples, 30
    pil_barrier_all, 29
    pil_get_nwID, 29
    pil_recv, 29
    pil_send, 29

task graph, 6, 24
Tile, 24

while, 23