MacShim: Compiling MATLAB to a Scheduling-Independent Concurrent Language

Neesha Subramaniam, Ohan Oda, and Stephen A. Edwards*
Department of Computer Science, Columbia University

Abstract

Nondeterminism is a central challenge in most concurrent models of computation. That programmers must worry about races and other timing-dependent behavior is a key reason that parallel programming has not been widely adopted. The SHIM concurrent language, intended for hardware/software codesign applications, avoids this problem by providing deterministic (race-free) concurrency, but does not support automatic parallelization of sequential algorithms.

In this paper, we present a compiler able to parallelize a simple MATLAB-like language into concurrent SHIM processes. From a user-provided partitioning of arrays to processes, our compiler divides the program into coarse-grained processes and schedules and synthesizes inter-process communication. We demonstrate the effectiveness of our approach on some image-processing algorithms.

1 Introduction

In this paper, we present a compilation technique that translates embarrassingly parallel MATLAB-style code (we support only a small subset of the language) into SHIM, a model and programming language that provides scheduling-independent concurrency, i.e., the input/output behavior of a SHIM process is independent of any nondeterministic choices made by the scheduler. In particular, race conditions simply cannot occur in the model.

While the general problem of translating nested loops with affine array indices into fairly efficient parallel code is hardly new, our work is novel for two reasons: the deterministic concurrent (SHIM) model we target and the coarse-grained parallelism approach we take. Specifically, rather than attempting to pipeline or parallelize instruction sequences in loops, we split the code of a loop nest operating on arrays into a user-specified number of concurrent processes, each of which contains a customized version of the loop nest operating on a different part of the array. We do not allow loop-carried dependencies, so our work is only applicable to embarrassingly parallel algorithms. Nevertheless, preliminary experimental results suggests that it is practical.

Our compiler, dubbed MacShim ("MATLAB converter for SHIM"), takes a program written in a MATLAB-like syntax that consists of a series of nested loops performing regular array accesses (with no loop-carried dependencies) and a user-specified partition of the arrays used in the program to parallel processes and compiles it into a concurrent SHIM program. Our main contributions are techniques for analyzing the data dependencies and synthesizing the needed inter-process communication.

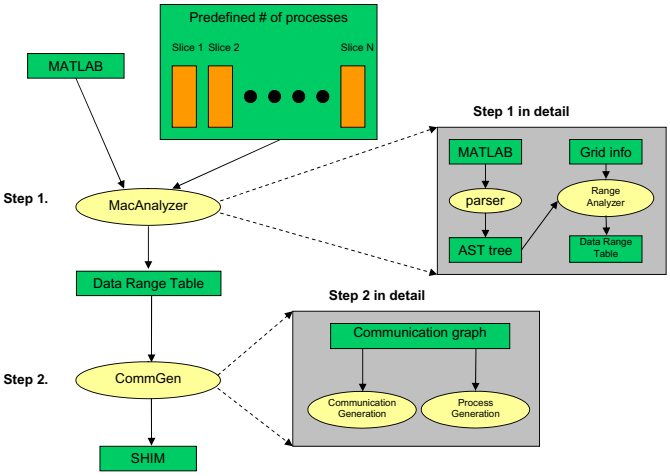

Figure 1: A graphical representation of the structure of our MacShim compiler.

The ultimate targets of our approach are multicore CPUs and systems-on-a-chip. The SHIM model assumes these processors have local memories (i.e., not an arbitrary large shared central memory) and some form of message-passing communication. Programming such systems is known to be difficult despite such standards as MPI and OpenMP. The goal of SHIM is to provide a deterministic concurrent programming environment that prevents many of the common pitfalls of concurrent programming (e.g., nondeterministic data races and deadlocks); the goal of this work is to provide an even more convenient specification for certain types of parallelizable (loops over arrays) algorithms.

1.1 Overview

Figure 1 depicts the structure of our compiler. Starting from a MATLAB-like program and a specification of how the arrays in the program should be distributed among concurrently-running processes, the compiler first deter-

*Edwards and his group are supported by an NSF CAREER award, gifts from Intel and Altera, and grants from the SRC and New York State's NYSTAR program.

```
for y = 2:imageY-1
  for x = 2:imageX-1
    sumX = 0;
    sumY = 0;
    for i = -1:1
      for j = -1:1
        sumX = sumX + inputImage(x+j, y+i) * sobelX(j+2, i+2) / 8;
        sumY = sumY + inputImage(x+j, y+i) * sobelY(j+2, i+2) / 8;
      end;
    end;
    sobelImage(x, y) = sumX * sumX + sumY * sumY;
  end;
end;

for y = 2:2:imageY
  for x = 2:2:imageX
    halfOut(x/2, y/2) = sobelImage(x, y);
  end;
end;
```

Figure 2: MATLAB code for a Sobel-like operation followed by a half-size operation

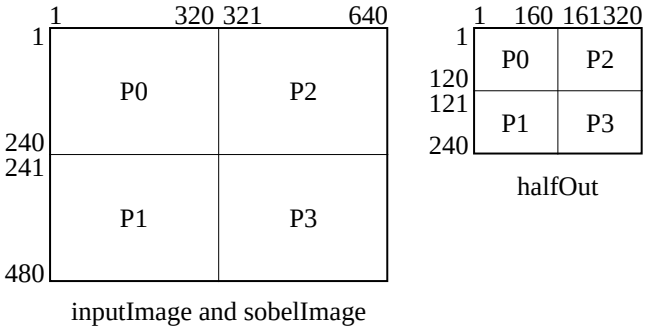

Figure 3: A particular user-specified array partitions for the example in Figure 2

mines, for each process, the regions of each array needed to compute the data for which the process is responsible. This information is summarized in a data range table; Section 4 describes this.

In the second step, the information from the first step is used to synthesize the code for each process. This amounts to copying the structure and modifying details such as array indices and loop bounds. The other important component of the generated code performs inter-process communication. Before the execution of each loop nest, the processes exchange the parts of each array that are owned by other processes. Section 5 describes the communication synthesis procedure; Section 6 describes the code generation procedure.

Figure 2 shows an example we will use to illustrate the operation of our compiler. It consists of two loop nests with simple dependencies. In the first, the *inputImage* array is convolved with the 3×3 *sobelX* and *sobelY* arrays to produce the *sobelImage* array. In the second, the *sobelImage* array is decimated to produce the *halfOut* array.

This basic example uses five arrays: *inputImage*, *sobelImage*, *halfOut*, *sobelX* and *sobelY*. We assume the first three are fairly big arrays and should therefore be distributed among two or more processes. Figure 3 shows a particular distribution for four processes. The two *sobel* arrays are small; each process keeps its own local copy of them.

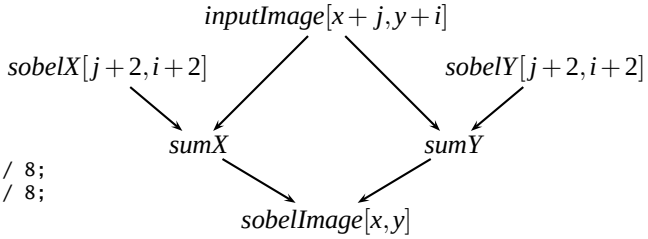

Figure 4: The data dependence graph for the first loop nest in Figure 2

Each iteration of both loop nests are independent (a not-uncommon feature in image processing algorithms) and are therefore easy to partition among multiple processes. The one challenge, and the one our compiler is most concerned with, is the fact that small portions of the *inputImage* array must be shared among multiple processes (i.e., the overlap along boundaries since the loops are performing a pair of convolutions). To address this, our compiler identifies such data dependencies and synthesizes communication actions that make the processes first exchange just enough of each array to be able to perform each part of their computation independently. Identifying such dependencies and synthesizing the communication code they demand is the main focus of our compiler.

In the remainder of the paper, after a review of related work and the SHIM model that is our code generation target, we discuss first the problem of inferring the ranges of each array that are needed by each process (Section 4), then the algorithm we use to synthesize the communication among processes based on the results of the range inference algorithm (Section 5), and finally the challenge of generating code for each process (Section 6).

In Section 7, we present experimental results that shows our technique can provide a modest speedup on a dual-processor Pentium-III system with SHIM running on top of POSIX thread, and finally discuss future work in Section 8.

2 Related Work

Although MATLAB has traditionally been executed by a single-threaded interpreter, it has also been compiled. For example, The MathWorks sells a compiler that translates MATLAB into C code suitable for a single processor. Our MacShim compiler tackles the harder problem of generating code for multiple processors.

Others have attempted to either translate or interpret MATLAB programs for parallel processors. DeRose and Padua's FALCON compiler [2, 3] translates MATLAB scripts into Fortran 90 programs. Quinn et al.'s Otter [8] compiler translates MATLAB programs into SPMD C programs with MPI calls.

```
process source(int32 &A) {
  A = 17;   // Send a sequence
  A = 42;   // to output A
  A = 157;
  A = 8;
}

process buffer(int32 &B, int32 A) {
  for (;;) B = A;  // Copy A to B
}

process sink(int32 B) {
  for (;;) B;   // Read and discard B
}

network main() {
  sink();   // Run processes in parallel
  buffer(); // port connections implicit
  source(); // (done by name)
}
```

Figure 5: A simple SHIM program for a one-place buffer. The source writes four values; the buffer copies its input to its output; and the sink always reads its input.

The RTExpress Parallel Libraries from Integrated Sensors Inc. consist of parallel performance-tuned implementations of over 200 MATLAB functions in C with MPI calls. MultiMATLAB [7] is a parallel MATLAB interpreter that, like our work, targets multiprocessors and networks of machines using MPI, but uses a parallel algorithm libraries instead of trying to analyze the parallelism of a MATLAB program.

The AccelFPGA compiler due to Banerjee et al. [1] translates MATLAB programs (signal-processing algorithms are their main target) into register-transfer-level code that can be synthesized onto field-programmable gate arrays. This approach seeks "instruction-level" parallelism, e.g., when two arithmetic operations in a loop may be pipelined or executed in parallel; by contrast, we partition the iterations of a loop in to multiple tasks that are executed separately.

The Compaan compiler due to Kienhius et al. [6] is probably closest in spirit to our work. Like us, they target a deterministic concurrent model of computation (Kahn process networks, a superset of the SHIM model that adds unbounded buffers), but again, their focus is more on "instruction-level" parallelism and their objective is mainly to pipeline signal processing algorithms. Compaan accepts a subset of MATLAB similar to ours, but richer because they allow certain loop-carried dependencies; we only accept simpler computations.

3 The SHIM language

In the SHIM language [5, 4], a system consists of concurrently-running sequential processes that communicate exclusively through fixed, point-to-point communication channels with rendezvous. SHIM systems are described with an imperative language with C-like syntax. Figure 5 is an example. Each process has local variables; there are no global variables. All processes execute concurrently.

Inter-process communication is synchronous: both sender and receiver must agree on when data is transferred; one always waits for the other. A process's arguments are input and output channels. Each appearance of a channel name becomes a write if it appears on the left of an assignment and a read otherwise.

The topology of communication channels and the number of processes is fixed and each communication channel connects one writing process to one reader. The communication structure of a system is therefore a directed graph whose nodes are processes and whose arcs are channels. The graph may contain cycles.

3.1 Syntax

A SHIM program (e.g., Figure 5) consists of three kinds of declarations: *struct*s, *process*s, and *network*s.

Structs. Struct declarations are C-like type declarations. Current variable types in SHIM are Booleans, fixed-size signed and unsigned integers, structs, and arrays. Booleans are 1-bit unsigned integers. There are no pointer types.

```
struct s {
  bool b;          Boolean
  int32 i;         signed 32-bit integer (including sign bit)
  uint16 t[24];    array of 24 unsigned 16-bit integers
};
```

Processes. A process declaration looks like a C function declaration and contains imperative code that runs sequentially. It is introduced by the *process* keyword followed by the name of the process, the formal arguments of the process between parentheses, and the body of the process delimited with curly braces. E.g.,

```
process xor(int8 I, int8 J, int8 &O) { O = I^J; }
```

The formal arguments of a process are its ports. Mimicking the syntax of C++ pass-by-reference parameters, SHIM uses & to indicate an output port; all others are inputs. Here, I and J are input ports; O is an output.

The body of a process consists of C-like code that may include *if-else* and *switch-case-default* conditionals, *while* and *for* loops (including *break* and *continue*), *label* and *goto* statements, expressions (including assignments), block statements and local variable declarations.

Expressions may mix local variable names and port names freely. Atomic assignments between structs or arrays are supported.

Networks. A network declaration, which instantiates a set of processes or subnetworks, is introduced by the *network* keyword followed by the name, the formal arguments, and the body of the network. The formal arguments are the ports of the network. The body of a network consists in a list of local channel declarations followed by a list of process and network instances.

```
1: procedure analyzeRangeInference(output array o, distributions D_p[o], dependent inputs I[1],...,I[n])
2:   Notation: R[v] is the range of v
3:   Determine the access region R[o] from the indices of o
4:   for each of this output's m processes p = P_1,...,P_m do
5:     if R_p[o] = R[o] ∩ D_p[o] is not empty then
6:       from R_p[o], find the ranges of variables v_1,...,v_k one which o depends
7:       for each dependent inputs i = I[1],...,I[n] do
8:         figure out i's access range R[i]
9:         if R[i] already exists then
10:          let R[i] = R[i] ∪ the newly-computed R[i]
```

Figure 6: The range inference algorithm

```
network xor2(int8 I, int8 J, int8 K, int8 &O) {
  int8 X;
  xor(X/O);
  xor(X/I, K/J);
}
```

Local channels connect one process (or subnetwork) in a network to another process (or subnetwork) in the same network. The types of ports connected through a channel must match, i.e., an output port of type t may only be connected to an input port of type t. Local channel declarations resemble local variable declarations.

The ports of the network come from ports of processes (or subnetworks) of the network that have no matching reading or writing process within the network. Port declarations for networks are no different from port declarations for processes. Port and local channel declarations in networks may be omitted as they are inferred by the compiler.

An instance resembles a function call. It consists of the name of a process or network followed by a list of actual arguments and a semicolon. The syntax for arguments associates formal and actual ports by name instead of position. For example, "`xor(X/I, K/J);`" instantiates process `xor` with actual ports "`(int8 X, int8 K, int8 &O)`", i.e., port name `X` is substituted for name `I`, name `K` for name `J`, whereas name `O` is left unchanged.

4 Range Inference

The first interesting phase in our compiler, after the usual parsing and static semantic analysis, is range inference (Figure 6), which determines what data each process needs to execute each loop nest (e.g., the Sobel and half-size operations in Figure 2).

Some definitions: a *range* is a sequence of integers, which we write with a colon, e.g., $-2:1$ represents the set $\{-2,-1,0,1\}$. We use ranges to model the values of loop indices, although note that our ranges always include every integer while loop indices often have a larger stride. Ranges are actually shorthands for sets, and we will take intersections and unions of them. Note that ranges are closed under these two operations.

A *region* of an n-dimensional array is a vector of n ranges. We write the ranges in a region as a comma-separated list of ranges enclosed in brackets, e.g., if a is a two-dimensional array then $[-1:2,0:3]$ is a range for a consisting of $a_{-1,0}$, $a_{0,0}$, ..., $a_{2,3}$. Note our regions are always rectangular and solid. Like ranges, regions are sets that are closed under intersection, but not under union. Instead, we take the least upper bound when we need a union-like operation.

For each loop nest for each process, our procedure produces a range for each array with elements the process must compute for the loop nest and the range of each array whose elements are needed to compute this result.

4.1 Range Inference on the Example

Consider determining the range information for the first loop nest in Figure 2 using our algorithm (Figure 6). Here, the output array *sobelImage* depends directly on two variables: *sumX* and *sumY*, which in turn depend on *inputImage*, *soelX*, *sobelY*, and themselves. Using a simple data flow analysis, we produce the dependency graph of Figure 4, which we use to determine which parts of which arrays are needed by each process to execute the first loop nest.

Next, we compute the region of each output array (here, just *sobelImage*) written by the loop nest, i.e., which calculations the processes are ultimately responsible for. From the dependency graph, the output array expression is *sobelImage*$[x,y]$ (we assume the index expressions are always linear functions of loop indices and constants, here, just x and y). We evaluate such expressions in a range domain in the obvious way, i.e., using the minimum and maximum values of each variable to obtain the minimum and maximum values of a linear function of these variables.

Here, it is trivial. From the program text, we know x ranges over $2:639$ and y ranges over $2:479$ (*imageX* and *imageY* are the constant values 640 and 480), so this loop nest will compute the region $[2:639,2:479]$ of the *sobelImage* array.

Next, for each process, we compute the regions of input arrays that it will need to compute the part of the output array for which it is responsible. The first step is to intersect the overall region of the output array computed by this loop nest with the part of the output array for which the particular processes is responsible.

Consider doing this for process P1 in Figure 3. The user has said P1 is responsible for *sobelImage*$[1:320,241:480]$. Intersecting this with the *sobelImage* region this loop nest will compute (i.e., $[2:639,2:479]$, computed earlier) gives $[2:320,241:479]$. This is how the first *sobelImage* row of Table 1 is computed. Since this region is non-empty for this process, our algorithm proceeds to determine the regions of the other arrays this process will need to compute the output array.

This final step walks backward through the data dependency graph. From the range required at the output array, our algorithm determines the ranges required of the input ar-

Table 1: Range information determined for the two loop nests in Figure 2 by the algorithm in Figure 6 from the partitions in Figure 3.

Array	**Regions required/produced for the first loop nest**			
	P0	**P1**	**P2**	**P3**
sobelX	$[1:3,1:3]$	$[1:3,1:3]$	$[1:3,1:3]$	$[1:3,1:3]$
sobelY	$[1:3,1:3]$	$[1:3,1:3]$	$[1:3,1:3]$	$[1:3,1:3]$
inputImage	$[1:321,1:241]$	$[1:321,240:480]$	$[320:640,1:241]$	$[320:640,240:480]$
sobelImage	$[2:320],[2:240]$	$[2:320,241:479]$	$[321:639],[2:240]$	$[321:639,241:479]$
	Regions for the second loop nest			
sobelImage	$[2:320,2:240]$	$[2:320,242:480]$	$[322:640,2:240]$	$[322:640,242:480]$
halfOut	$[1:160,1:120]$	$[1:160,121:240]$	$[161:320,1:120]$	$[161:320,121:240]$

```
procedure generateCommunicationGraph()
  for each loop nest do
    for each process p do
      for each output array a do
        Start a new communication graph
        for each process p' do
          if p' ≠ p and p' has part of array a needed by
          process p then
            Add an arc p' → p labeled with the region
            needed from p'
```

Figure 8: Deriving communication graphs from range tables

rays. Unlike the output range computation, this procedure involves solving equations (because we are looking for the inputs that produce a given output), but the equations are easy to solve because we assume the index expressions are linear.

For example, consider determining the region of *inputImage* required to evaluate the first loop nest in P1. Earlier, we determined that P1 would compute the region $[2:320,241:479]$ of *sobelImage*. In the dependence graph Figure 4, there are two equivalent paths from *inputImage* to *sobelImage*. Along either, we find index x in *sobelImage* is determined by index $x+j$ in *inputImage*. Since the target index x is so simple, the range calculation for the first dimension of *inputImage* is simple: $2:320+-1:1=1:321$. Similarly for the second dimension, we have $241:479+-1:1=240:480$. Together, these produce the entry for the *inputImage* row in the P1 column of Table 1.

5 Communication Synthesis and Scheduling

From the region tables, we look at what array regions a process requires and compare them to what processes own the data in these region to derive a communication graph such as Figure 7 using the algorithm in Figure 8. In such a graph, each node represents a process and each directed arc represents a region of an array that needs to be transferred along the arc. We produce a separate communication graph for each loop nest.

```
procedure scheduleCommunication()
  for each loop nest do
    Set remaining pairs to all communication pairs for
    this loop nest
    phase = 1
    while there are remaining pairs do
      Clear the busy flag for each process
      for each remaining pair do
        if both source and destination of the pair are not
        busy then
          Add it to the list of scheduled pairs for this
          phase
          Remove it from the list of remaining pairs
          Mark the source and destination process as
          busy
      phase = phase + 1
```

Figure 9: Scheduling communication from the communication graphs

The main check required to determine whether data must be transferred, i.e., whether one process needs information held by another, amounts to an intersection of regions.

To execute each loop nest, we need to perform a communication for each arc in the communication graph. Such graphs are often dense, meaning each process must perform many communications, but SHIM semantics say that each process may communicate with at most one other process at a time, so the data transfers must be scheduled.

We schedule the required data transfers for each loop nest into a sequence of phases using the heuristic algorithm in Figure 9. In each phase, we attempt to perform as many process-to-process communications as possible, subject to the constraint that each process may communicate with at most one other process in each phase. We use a greedy ap-

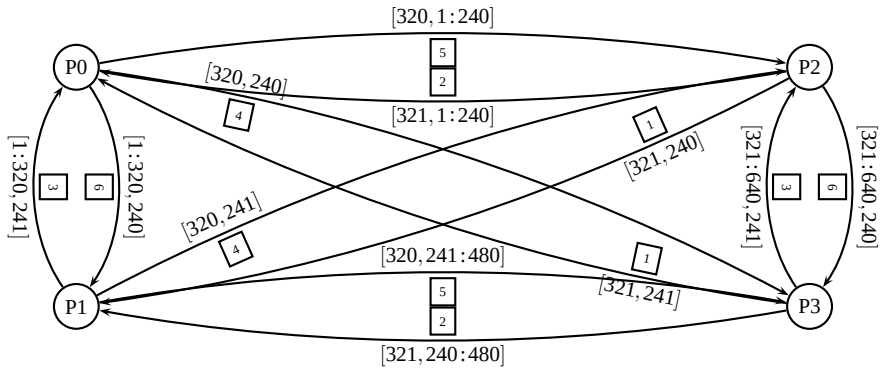

Figure 7: Communication Graph with scheduling information for the first loop nest for the inputImage array. Phase numbers are in boxes.

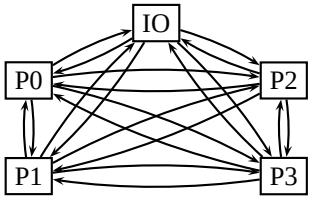

Figure 10: Structure of the synthesized processes for a four-process partition. The IO process distributes inputs and receives results from the four numbered slave processes. Each slave process contains a copy of the loop nests to be executed; each is responsible for computing part of each array.

proach: we simply pick pairs of processes that need to communicate until no more can be selected in the current phase (i.e., the source or destination of every remaining pair is already communicating in the phase). We repeat this selection process for as many phases as necessary to perform the communication requested by every arc in the communication graph. Note that this algorithm ensures the number of remaining communications decreases monotonically in each phase, guaranteeing it will complete.

The labels on the arcs in Figure 7 indicate the phases in which each communication takes place for the loop nest in Figure 2. As can be seen, it is a fully connected graph that requires six phases to complete.

6 Code Generation

The code generation stage generates code for computation corresponding to the input MATLAB program and for inter-process communication. Figure 10 shows a block diagram of the structure of the generated system when there are four computational processes. At the top is an IO process responsible for reading the input image(s) and distributing them to all the processes. At the end of computation, Process IO receives array regions from the other processes, combines them, and writes them out. Figure 12 shows our code generation algorithm that produces SHIM systems with this structure.

```
for (int y = 241 ; y <= 479 ; y = y + 1) {
  for (int x = 2 ; x <= 320 ; x = x + 1) {
    sumX = 0;
    sumY = 0;
    for (int i = -1 ; i <= 1 ; i = i + 1) {
      for (int j = -1 ; j <= 1 ; j = j + 1) {
        sumX = sumX + inputImage[x+j-1][(y-239)+i-1] *
               sobelX[j+2-1][i+2-1] / 8;
        sumY = sumY + inputImage[x+j-1][(y-239)+i-1] *
               sobelY[j+2-1][i+2-1] / 8;
      }
    }
    sobelImage[x-1][(y-240)-1] = sumX * sumX + sumY * sumY;
  }
}
```

Figure 11: SHIM code generated for the first loop nest in Figure 2 for process P1 (Figure 3)

The communication channels connecting various processes are of fixed size since we determined the size of each array received and sent by each process. While computing the range information in the input program, we identify the type of each variable, whether it is an array, loop index, or local variable. For each process, we declare all the local variables. For the arrays, as we already have information about their size in each process, we declare them and initialize them using the data received from Process IO. Following initialization, for each loop nest, communication code is followed by computation code. Generation of computation code is a little tricky since array indices must be adjusted. In particular, for each array index in a process, we subtract the minimum index of each dimension of its region in the process. The range of each *for* loop index is derived from its range, computed in the range inference step (line 6 in Figure 6). If no constrained variable range was computed, the original range is used. Finally, the computed data is sent back to Process IO to be reassembled into the final result arrays.

6.1 An Example

Figure 11 shows part of the SHIM code our compiler generates for one process from the first loop nest in Figure 2.

```
procedure generate()
  call process(p) for each process p
  ioprocess()
  "instantiate each process"
  "instantiate the IO process"

procedure process(p)
  "write io variables, channel widths"
  "initialize arrays, local variables"
  for each input array do
    "receive data from Process IO"
  for each loop nest l do
    communication(p, l)
    body(p, l)
  for each output array do
    "send data to process IO"
procedure communication(p, l)
  for each phase do
    if p sends data this phase then
      "copy the data to a small array"
      "send the small array"
    else if p receives this phase then
      "receive the data into local array"
      "write it to the main array"

procedure body(process_id, loop_nest)
  walk the AST
  for each root node do
    print-SHIM-code(root-node)

procedure ioprocess()
  for each input array do
    read input array
    split array into pieces according to user-provided partition
    send each piece to its owning process
  for each output array do
    receive data from each process
    combine data into one array
    write output array
```

Figure 12: The SHIM code generation algorithm. Code in quotes is output (generated).

This code is for the lower-left process when the arrays are divided into 2×2 grids—the compiler generates three other processes very much like this. A few basic observations: the control structure of the generated code is the same as the source, the ranges of the *for* loops have been modified, and the array index expressions have offsets not present in the original code.

Such transformations are typical. Our range analysis phase determines which parts of each array are owned and therefore must be calculated by each process as well as all the input data that is needed to calculate each part. This information is used to create "bloated" versions of each array that is owned by a process that includes room for the information from other processes. Each loop nest is then rewritten for each process such that the array indices are correct for these bloated arrays.

A convenient side-effect of all this bookkeeping is that MATLAB's indexed-from-one arrays are automatically dealt with and converted to SHIM's indexed-from-zero arrays.

Table 2: Experimental results for the example program

Example	Number of Processes				
	1	2		4	
	time	time	speedup	time	speedup
One Processor (1.6 GHz Pentium M)					
Sobel+Half	0.26s	0.27s	0.96×	0.28s	0.93×
Rotate	0.90s	1.1s	0.82×	1.2s	0.75×
Blend	0.70s	0.67s	1.04×	0.66s	1.04×
Two Processors (750 MHz Pentium III)					
Sobel+Half	0.61s	0.37s	1.6×	0.37s	1.6×
Rotate	2.8s	2.5s	1.1×	2.6s	1.1×
Blend	1.7s	1.3s	1.3×	1.3s	1.3×

7 Experimental Results

We implemented the MacShim compiler in Java and used it to generate SHIM code for a few little MATLAB programs, such as the example program in Figure 2, for one, two, and four processes. We modified the SHIM compiler to produce C code that uses the POSIX threads (pthreads) library for concurrency and inter-thread communication. This is fairly inefficient: in particular, the generated code uses memory-to-memory copies when transmitting arrays; a more shared-memory-aware implementation would certainly improve upon this.

We chose pthreads because they are fairly portable and provide a simple way to harness the power of multiprocessor systems. We used the stock pthreads implementation under Linux 2.6. No attempt was made to optimize the quality of the C code generated from the SHIM code generated by our compiler; this important issue is outside the scope of this paper.

To evaluate the correctness of our approach, we simply compared the output of the SHIM code generated with a single process running some benchmarks on some images with the output from multi-process SHIM code, which was bitwise-identical.

To evaluate the effective speedup of the approach, we compared the execution speed of code generated with one process versus two and four on two platforms: a single-processor Pentium M-based laptop (a baseline) and a dual-processor Pentium III-based server. Each were running Linux; the dual-processor system was using an SMP kernel that automatically migrated processes across processors. Neither of these are particularly high-performance systems; they are intended to demonstrate the correctness and potential efficiency gains of our approach.

Table 2 reports the time it took to execute the kernels of the three examples (*Sobel+Half* is the example in Figure 2

run on a 640×480 image, *rotate* turns a 2000×2000 image ninety degrees; blend merges two 1500×1000 images using a third image as an alpha channel). The times include some communication overhead: the time to distribute the images among the computation processes and to receive all the results plus all inter-process communication. They do not include any system I/O times (i.e., to read and write input and input data). These are wall clock times gathered with the Unix *times* command, which has 10 ms precision at best and is affected by other processes running on the machine; the experiments were conducted when the systems were lightly loaded and the results presented here are averages—the raw times differed in a few 10s of ms.

Not surprisingly, there is a penalty in splitting the code into multiple processes on a single-processor system. This is certainly due to the additional context-switching and communication overhead on a single processor.

The dual-processor results illustrate the advantage of our approach. Splitting the system into two processes produced a $1.6\times$ speedup on the Sobel example, which of course is less than the ideal $2\times$, but is at least noticeably better. The other two examples show a more modest speedup, but this is because their execution time is dominated by communication. Rotate, in particular, does little more than move data around, i.e., is much less computationally intensive than the Sobel example. Blend is more computationally intensive, but also uses three times as much data as the Sobel example, so it too suffers from communication overhead.

The absolute execution times for these examples are poor. In particular, the two-processor server is slower than the one-processor laptop (even with a single process), but this is because the two processor system is two generations behind. However, our goal in this work was a compilation technique that would generate parallel code for the SHIM language; a more efficient implementation of the SHIM semantics on parallel hardware is future work.

8 Conclusions and Future Work

We designed and implemented a compiler that translates simple array operations coded in a MATLAB-like language into the SHIM concurrent language. The compiler performs three main operations: data range analysis from a simple static analysis of the source code, communication analysis from this analysis, and SHIM code generation that makes a modified copy of the original code for each computation process and adds code for inter-process communication code.

We make many simplifying assumptions about the input code to make our task easier. In particular, we do not support loop-carried dependencies, dynamically sized arrays, non-affine array indices, and complex loop ranges. This is restrictive, but remains useful for a variety of image-processing algorithms.

Much remains to be done. The quality of the C code generated from SHIM will be improved. Work on improving the speed of this code and in particular inter-process communication is ongoing. The obvious next step in our compiler is to support more complicated input code, such as allowing loop-carried dependencies. Despite these shortcomings, we have shown that it is possible and realistic to compile loop/array code into a coarse-grain parallel SHIM program.

References

[1] Prithviraj Banerjee, Malay Haldar, Anshuman Nayak, Victor Kim, Vikram Saxena, Steven Parkes, Debabrata Bagchi, Satrajit Pal, Nikhil Tripathi, David Zaretsky, Robert Anderson, and Juan Ramon Uribe. Overview of a compiler for synthesizing MATLAB programs onto FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):312–324, March 2004.

[2] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, Bret A. Marsolf, and David A. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 1033 of *Lecture Notes in Computer Science*, pages 269–288, Columbus, Ohio, August 1995.

[3] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.

[4] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[5] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 14(8):854–867, August 2006.

[6] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: deriving process networks from Matlab for embedded signal processing architectures. In *Proceedings of the International Conference on Hardware Software Codesign (CODES)*, pages 13–17, San Diego, California, May 2000.

[7] Vijay Menon and Anne E. Trefethen. MultiMATLAB: integrating MATLAB with high-performance parallel computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–18, San Jose, CA, November 1997.

[8] Michael J. Quinn, Alexey G. Malishevsky, and Nagajagadeswar Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proceedings of High Performance Distributed Computing (HPDC)*, pages 114–121, Chicago, Illinois, July 1998.