

MSL: a Synthesis Enabled Language for Distributed High Performance Computing Implementations

by

Zhilei Xu

B.Eng., Tsinghua University (2008)

S.M., Massachusetts Institute of Technology (2011)

Submitted to

Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

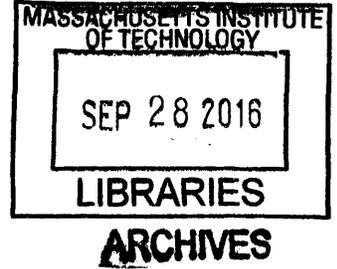
Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Massachusetts Institute of Technology 2016. All rights reserved.



Signature redacted

Author

Department of Electrical Engineering and Computer Science

August 25, 2016

Signature redacted

Certified by

Armando Solar-Lezama

Associate Professor of Computer Science at MIT

Thesis Supervisor

Signature redacted

Accepted by

UU

Leslie A. Kolodziejcki

Chair, Department Committee on Graduate Theses

MSL: a Synthesis Enabled Language for Distributed High Performance Computing Implementations

by

Zhilei Xu

Submitted to Department of Electrical Engineering and Computer Science
on August 25, 2016, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

SPMD-style (single program multiple data) parallel programming, usually done with MPI, are dominant in high-performance computing on distributed memory machines. This thesis outlines a new methodology to aid in the development of SPMD-style high-performance programs. The new methodology is supported by a new language called MSL that combines ideas from generative programming and software synthesis to simplify the development process as well as to allow programmers to package complex implementation strategies behind clean high-level reusable abstractions. We propose in this thesis the key new language features in MSL and new analyses in order to support synthesis and equivalence checking for programs written in SPMD-style, as well as empirical evaluations of our new methodology.

Thesis Supervisor: Armando Solar-Lezama

Title: Associate Professor of Computer Science at MIT

Acknowledgments

First and foremost, I want to thank my advisor, Dr. Armando Solar-Lezama, for his consistent guidance, extraordinary patience, and unreserved support. I could not have possibly gone this far without the endless help I got from him. I feel extremely fortunate to have him as my advisor, my mentor, and my friend. His huge influence on me shapes my thoughts about how I should spend my life and what kind of person I ought to become. I would also like to express my sincere gratitude to Dr. Saman Amarasinghe and Dr. Srinivas Devadas for their invaluable insights. This thesis would not have been possible without their advice and encouragement.

My collaborator, Dr. Shoaib Kamil made substantial contribution to the MSL project, and has constantly helped me in the area of high performance computing. This work has received precious advice and feedback from Dr. Rastislav Bodik and Dr. Adam Chlipala from its very early stage. I owe them many thanks.

The MSL project was partly inspired by a research project on GPU programming done by Nicholas Tung (she is now Zora Tung). I want to thank her and wish all the best to her.

I also got innumerable help from my group mates and office mates, Dr. Alvin Cheung, Dr. Sicun Gao, Dr. Shachar Itzhaky, Deokhwan Kim, Jimmy Koppel, Ivan Kuraj, Dr. Nadia Polikarpova, Evan Pu, Dr. Xiaokang Qiu, Dr. Rishabh Singh, Rohit Singh, Zenna Tavares, Dr. Jean Yang, and Dr. Kuat Yessenov. My life as a graduate student would not have been so colorful without them.

My roommate and all-time friend, Dr. Xi Wang, has always given me gracious help in both research and life. My mentor, Dr. Xuezheng Liu and my supervisor, Dr. Lidong Zhou at Microsoft Research Asia, were always willing to give me support and advice, even after I left the job to pursue my PhD at MIT. In some sense, they are the ones who have brought me into the career of a computer science researcher.

My friend, Dr. Shuo Tang and Dr. Haohui Mai, were the source of help when I was in Champaign / Urbana. They were always there whenever I needed a hand.

Dr. Yuan Zhou, my old friend since elementary school, has given me lots of advice

and inspiration all the time. I was lucky to be his roommate during the last two years of my PhD. Dr. Zhou's father, Mr. Qihua Zhou, introduced programming to me, and influenced my entire life. I also owe my deepest gratitude to my middle school teacher, Mr. Tao Jiang, who brought me to the world of algorithmic programming contest. He has always been my role model.

I want to thank my parents, Baoan Xu and Wenfang Lu, for their love and support.

Finally I would like to thank my wife, Dr. Bin Lian, for being in my life for the last twenty-four years. I could never imagine a life without her.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 13 |
| 1.1 | The MSL Programming Model | 16 |
| 1.2 | Contributions | 24 |
| 1.3 | Structure of the thesis | 25 |
| 2 | Related work | 27 |
| 2.1 | Distributed memory SPMD programming for high performance computing (HPC) | 27 |
| 2.2 | General Purpose Languages for HPC | 28 |
| 2.3 | Domain-Specific Languages for HPC | 31 |
| 2.4 | Software Synthesis | 33 |
| 2.5 | Generative Programming | 34 |
| 2.6 | Verification of Parallel Programs | 36 |
| 2.7 | Compiler-based Optimization | 36 |
| 2.8 | Autotuning | 37 |
| 3 | The MSL programming language | 39 |
| 3.1 | Core imperative language | 39 |
| 3.1.1 | Native array support | 40 |
| 3.1.2 | Implicit parameters | 46 |
| 3.1.3 | Pass-by-reference parameters | 46 |
| 3.2 | Basic Synthesis Features | 47 |
| 3.2.1 | Unknown integer holes | 47 |

| | | |
|----------|---|------------|
| 3.2.2 | Choice expression | 47 |
| 3.2.3 | Assertions | 48 |
| 3.2.4 | Refinement relationship among functions | 48 |
| 3.2.5 | Generator functions | 49 |
| 3.3 | SPMD programming in MSL | 50 |
| 3.3.1 | Barrier-like behavior | 50 |
| 3.3.2 | Expressing parallelism by <code>spmd</code> | 51 |
| 3.3.3 | Communication/synchronization mechanisms | 52 |
| 3.3.4 | Enforcing bulk-synchronous constraints | 55 |
| 3.3.5 | Relating SPMD and sequential programs | 58 |
| 4 | SPMD-to-sequential transformation | 65 |
| 4.1 | The L_{small} language | 66 |
| 4.2 | Semantics of L_{small} | 70 |
| 4.3 | Reducing SPMD programs to sequential ones | 78 |
| 5 | Synthesis and equivalence checking | 87 |
| 5.1 | Constraint generation | 87 |
| 5.2 | Solving for control parameters | 89 |
| 5.3 | Correctness checking procedure | 89 |
| 5.4 | Completing the sketch | 90 |
| 6 | Code generation | 93 |
| 6.1 | From bulk-synchrony to point-to-point communications | 94 |
| 6.2 | Formally proving the correctness of the optimized point-to-point transfer | 97 |
| 6.3 | Other implementation details | 101 |
| 7 | Combining distributed kernels | 103 |
| 7.1 | Package System in MSL | 106 |
| 7.2 | Defining the refinement relation | 109 |
| 7.3 | Establishing equivalence between packages | 112 |
| 7.4 | SPMD programming with packages | 117 |

| | | |
|----------|--|------------|
| 8 | Case studies | 121 |
| 8.1 | 3D Transpose from FT | 122 |
| 8.2 | The MG Benchmark | 122 |
| 8.3 | Specialized Reduction for SpMV from CG | 124 |
| 8.4 | Time cost of synthesis step | 126 |
| 8.5 | Performance of Case Studies | 126 |
| 9 | Conclusion | 129 |

List of Figures

| | | |
|------|--|----|
| 1-1 | A simple kernel: transposing a 3D array | 16 |
| 1-2 | A sample partition scheme for the 3D arrays | 17 |
| 1-3 | A simple example illustrating All_to_all, for the two-processes case . . . | 19 |
| 1-4 | How MSL programs are processed | 25 |
| 3-1 | Example of bulk array access | 41 |
| 3-2 | Example of an nested array, and the same array treated as a two-dimensional array | 44 |
| 4-1 | The simplified language L_{small} | 67 |
| 4-2 | Programs in L_{small} | 68 |
| 4-3 | The execution model of L_{small} | 68 |
| 4-4 | Variables in L_{small} | 69 |
| 4-5 | The instruction language L_{inst} | 71 |
| 4-6 | Translating L_{small} statements to L_{inst} instructions | 71 |
| 4-7 | The state that the L_{inst} programs operate on | 72 |
| 4-8 | Program counters and Execution mode in L_{inst} | 72 |
| 4-9 | The execution semantics of L_{inst} | 73 |
| 4-10 | State transition graph of subprocesses executing a group of tin , trn , and $wait$ instructions. | 77 |
| 4-11 | Reduction rules from L_{small} to L_{small}^{seq} | 80 |
| 6-1 | (a) the straightforward MPI implementation for <code>transfer</code> and (b) an illustration of its runtime behavior | 94 |

| | | |
|-----|---|-----|
| 6-2 | Buggy behavior after removing barriers naïvely | 95 |
| 6-3 | an optimized implementation for <code>transfer</code> (slightly simplified) | 96 |
| 6-4 | Using tagging to prevent mismatched messages | 96 |
| 6-5 | The new semantics of the optimized <code>transfer</code> | 97 |
| 6-6 | Illustration of transfer with (a) the old channel and (b) the new channel | 98 |
| 7-1 | Simple example of composing distributed kernels | 103 |
| 7-2 | Re-organizing the example with abstract data type | 104 |
| 8-1 | Performance of our three benchmarks from NAS | 127 |

Chapter 1

Introduction

High-performance computing on distributed memory machines is usually implemented in an SPMD-style (single program multiple data) parallel programming paradigm. In this paradigm, there are multiple processes running the same program in parallel, but the state is partitioned into individual local states, owned by each individual process. Processes cannot access each others' memory, so they coordinate by sending and receiving messages.

In practice, the dominant programming model to write SPMD distributed implementations is still to use a rather low-level programming language such as Fortran or C/C++ with MPI (message passing interface) [35], a library and runtime to support coordinations among distributed processes. The inherent non-determinism of the MPI semantics, combined with the usual shortcomings of programming in a low-level language, makes distributed implementations difficult to write, hard to test, and error-prone [26, 31, 57, 96].

This thesis shows how new techniques from the areas of software synthesis [84], automated bug-finding [20] and generative programming [73, 95] can be combined to aid the development of high-performance distributed implementations. It expands upon our work published in [104].

One of the main challenges for this work is scaling the symbolic reasoning techniques required for synthesis and automated bug-finding to complex distributed memory implementations based on message passing. The main contribution of this thesis is

to show that it is possible to scale symbolic reasoning by restricting the language to one that uses high-level notations for array manipulations and bulk-synchronous parallelism, and by leveraging these notations in new symbolic encodings.

We have developed a language called MSL as a vehicle to explore these ideas. The goal of MSL is not to replace commonly used languages like C++ and Fortran, but to demonstrate how synthesis can be incorporated into an imperative language given suitable abstractions for communication and array manipulations. MSL is able to assist programmers by synthesizing messy low level details involved in writing complex distributed memory implementations. By using a set of benchmarks (including NAS parallel benchmarks [3] as a target, we show how the synthesis features of MSL can help in deriving the details of complex distributed implementations and check for bugs against a sequential reference implementation. MSL generates standard C++ code that can be integrated with larger programs written in traditional languages, and we show that this code generation does not introduce performance or scalability problems by comparing the performance of our generated code with that of standard MPI and Fortran/C++ implementations.

Our methodology of enforcing bulk-synchrony is innovative in that it explores a new territory not touched by existing approaches. On the one extreme, there are languages that enforce bulk-synchrony strictly, by type systems (like Titanium¹ [43] and Split-C [1]) or fully restricted programming model (like Pregel [71] and Giraph [18, 75]) — these methodologies are provably sound, but require a lot of efforts on the programmers' side. On the other extreme, there are languages that only check for bulk-synchrony dynamically (like the modified version of Titanium [53] and UPC² [92]), thus only detect deviations at runtime; and languages like C and Fortran give the programmers full flexibilities, so they can organize their SPMD code to conform to the bulk-synchronous style if they are willing to, but the compilers generally cannot prove or enforce this conformity, thus cannot take advantage of it to facilitate analyses or optimizations.

¹Titanium has both message-passing and PGAS mechanisms for communications. When restricted in the message-passing subset, Titanium is guaranteed to be bulk-synchronous because of its enforcement of *single-valued expressions* in the type system.

²UPC provides *named barriers*, which can be used to check for bulk-synchrony at runtime.

MSL is different from all of them: it adds checks similar to the dynamic approaches like [53], but these checks are only added at synthesis (compile) time (see Chapter 4 for how these checks are added), thus not paying the runtime cost. At synthesis time, MSL relies on the correctness checking feature of SKETCH to do all the checks, which also include the checking for array bound violations. The checking procedure in SKETCH, similar to the bounded model-checking, is unsound, but in practice, it detects bugs effectively, and enforces a certain discipline for the programmers to get the code right. We name this new variant a *static-dynamic checking* approach, because the kind of program checking is similar to what would be done dynamically, but we do it statically at synthesis time, by means that is unsound yet practically effective.

In the rest of this chapter, we provide a high-level overview of MSL and the programming model that it supports, and a summary of the techniques underneath MSL.

1.1 The MSL Programming Model

We illustrate the MSL programming model through a simple example: transposing a three dimensional array and getting a new array, as illustrated in Figure 1-1.

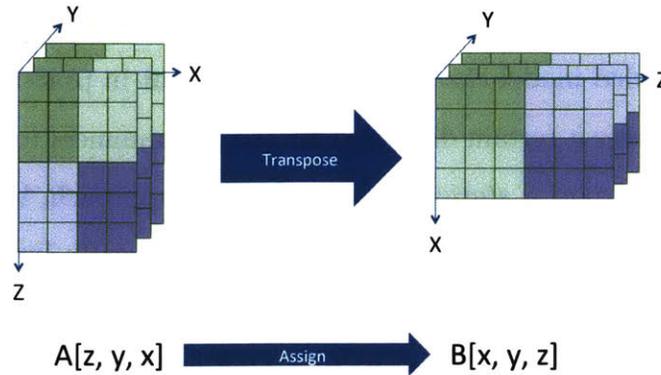


Figure 1-1: A simple kernel: transposing a 3D array

Below is a simple reference implementation of the transpose in MSL.

```
void trans(int nx, int ny, int nz, double[nz,ny,nx] A, ref double[nx,ny,nz] B)
{
  for (int x = 0; x < nx; x++)
    for (int y = 0; y < ny; y++)
      for (int z = 0; z < nz; z++)
        B[x,y,z] = A[z,y,x];
}
```

The core language is similar to C, but with additional features to facilitate analysis of programs. One such feature illustrated in this example is native support for multi-dimensional arrays, including the ability to describe the dimensions of an array as part of its type. For example, in the code above, the expression $A[z,y,x]$ is legal for all $0 \leq z < nz$, $0 \leq y < ny$, $0 \leq x < nx$. Arrays are row-major so nx represents the unit-stride dimension of A and nz represents the slowest-growing dimension. The language also supports reference parameters, but places strong restrictions on aliasing to simplify the analysis: the parameters of a function are enforced to not alias with each other, similar to what Fortran requires for array types.

Transitioning from shared to distributed memory is usually a challenge because data must be *partitioned* across different processes, with each owning a portion of the global data. For our running example, we illustrate how this process works in MSL

for the case where the 3-dimensional grid is partitioned across its slowest growing dimension; specifically, given N processes, we partition the z dimension of array A , and the x dimension of the transposed array B . This partitioning scheme is shown in Figure 1-2.

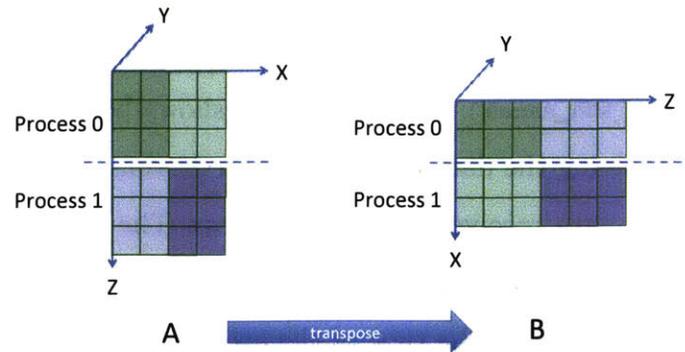


Figure 1-2: A sample partition scheme for the 3D arrays
Note that the Y dimension is elided in the picture.

Because the dimension across which we partition is different for the two arrays, the transpose requires a re-distribution of data across the machine.

```
smpd void dtrans(int nx, int ny, int nz, double[nz/nprocs, ny, nx] LA, ref double[nx/nprocs, ny, nz] LB)
{
    int bufsz = (nx/nprocs)*ny*(nz/nprocs);
    view LA as double[nprocs, bufsz] abuf;
    view LB as double[nprocs, bufsz] bbuf;

    pack(LA, bbuf);

    All_to_all(bbuf, abuf); // re-distribute

    unpack(nx, ny, nz, abuf, LB);
}
```

The implementation in MSL shown above uses the efficient transposition strategy used by the NAS FT benchmark [3]. The `smpd` keyword tells the compiler that `dtrans` is a function that runs on multiple processes in SPMD mode, so code inside `dtrans` gets access to a constant integer `nprocs`, which equals the total number of processes.

The first observation is that in typical SPMD style, the code now operates on a local partition of the original arrays. Specifically, note that the size of the innermost dimension for each array has now been divided by the number of processes. It is

possible to generalize this to non-uniform partitions, but we use this form in the introduction for clarity of presentation.

The code involves three steps: a local packing step that groups together data that must be sent to the same destination process, a global exchange through `All_to_all()`, and a local rearrangement to unpack the received data into the output array.

The function `All_to_all()` is part of the MSL standard library and is implemented efficiently in terms of `MPI_Alltoall()`. As its interface shows below, it takes as input a source 2D array containing the data to send to each destination process, and returns a new array with the data received by every process. The bracket around the first parameter indicates that it is *implicit*; if it is not passed, as is the case in the example above, the system infers it from the type of the other arguments.

```
void All_to_all([int bufsz], double[nprocs, bufsz] src, ref double[nprocs, bufsz] dst);
```

`All_to_all` performs a global exchange among all processes: each process behaves as both a sending process and a receiving process. The source data `src` can be seen as a group of `nprocs` one-dimensional arrays, each of length `bufsz`. During the exchange, the i -th array in `src` from every sending process is sent to the i -th receiving process. Because there are `nprocs` sending processes in total, after the exchange, each receiving process receives `nprocs` many 1D arrays, each of size `bufsz`, and these 1D arrays are grouped together to form a 2D array of type `double[nprocs, bufsz]`, stored into the `dst` variable on the receiving process; when grouping the received arrays, the order is determined by each array's originating process: the 1D array sent from the i -th sending process is put into the i -th location in `dst`. The above procedure is illustrated in Figure 1-3.

From its definition and the illustration in Figure 1-3, we can see that `All_to_all` performs part of the transpose we want to achieve, but not the complete transpose: in the running example, the 3D transpose not only exchanges arrays among processes, it also changes the shape of those arrays, and re-arranges elements in them, as is shown in Figure 1-2.

This is where the programmer's insight shines: she knows that `All_to_all` does something similar to the 3D transpose, and calling `All_to_all` instead of performing

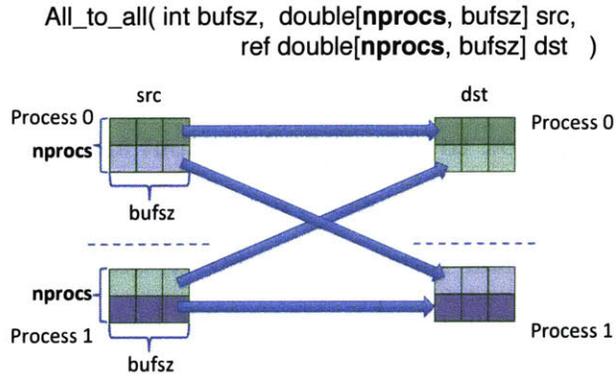


Figure 1-3: A simple example illustrating All_to_all, for the two-processes case

such exchange manually offers better performance; and she also knows that `pack()` and `unpack()` functions are needed to re-arrange elements in the arrays.

An important optimization in the `dtrans()` code above is that the temporary buffers `abuf` and `bbuf` used to send and receive information through `All_to_all()` are not allocated as separate buffers; instead, they are different *views* of the original arrays `LA` and `LB`. These views introduce an aliasing relationship between `LA/LB` and `abuf/bbuf` respectively, but the language does not suffer from the typical aliasing problems because the aliasing relationship is static and the underlying analysis in MSL ensures that the two views are compatible, *e.g.* a legal access to `abuf` will always translate to a legal access to `LA`.

The function `pack()` which packs the data for `All_to_all()` illustrates some of the benefits of relying on synthesis to support the implementation. Instead of having to derive by hand the necessary code to collect the data, the user provides a sketch of the necessary code.

```

void pack([int an1, int an2, int an3, int bn1], double[an1, an2, an3] in, ref double[nprocs, bn1] out)
{
  view out as double[nprocs*bn1] fout;
  generator int num() {
    return { | an1 | an2 | an3 | bn1 | nprocs | };
  }
  generator int dim() {
    return { | ?? | num() | num()/nprocs | dim()*dim() | };
  }
  for (int i = 0; i < an1; i++)
    for (int j = 0; j < an2; j++)
      for (int k = 0; k < an3; k++)
        fout[i*dim()+j*dim()+k*dim()] // i+j*an1+k*an1*an2
        = in[i][j][k];
}

```

```
}
```

In the sketch above, the user first creates flat views of `out` to give the synthesizer maximum flexibility as to how to index into the output array. The programmer then defines two *generators*, `num()` and `dim()`, which together define a grammar of expressions involving all the available scalar variables. Specifically, the choice syntax `{|·|·|}` lets the synthesizer choose among many possible expressions, including recursively generated ones. The generator contains high-level insight into how dimensions combine when indexing into a flat array: for example, dividing one of the dimension sizes (such as `an3`) by `nprocs` might be useful because we might want to break one of the dimension into `nprocs` pieces; multiplying two dimensions together also makes sense and may be useful, so is also included as a choice. The programmer can then use these generators within her code in place of actual expressions and the synthesizer will discover the correct expressions (shown in the comments). Here the input array `in` is visited normally, and `fout` is indexed by a combination of the loop variables and `dim()` generators.

The function `unpack()` is written in a similar way, as shown below. The exact indexing logic for this function is trickier than `pack()`, but the programmer can rely on the synthesizer to discover it automatically.

```
void unpack([int an1, int bn1, int bn2, int bn3], double[nprocs, an1] in, ref double[bn1, bn2, bn3] out)
{
    view in as double[nprocs*an1] fin;
    view out as double[bn1*bn2*bn3] fout;

    generator int num() {
        return {| an1 | bn1 | bn2 | bn3 | nprocs |};
    }
    generator int dim() {
        return {| ?? | num() | num()/nprocs | dim()*dim() |};
    }

    for (int p = 0; p < nprocs; p++)
        for (int i = 0; i < dim(); i++) // bn1
            for (int j = 0; j < dim(); j++) // bn2
                for (int k = 0; k < dim(); k++) // bn3/nprocs
                    fout[p*dim() + i*dim() + j*dim() + k*dim()] // p*(bn3/nprocs) + i*(bn2*bn3) + j*bn3 + k*1
                        = fin[p*dim() + i*dim() + j*dim() + k*dim()]; // p*an1 + i*(bn2*bn3/nprocs) + j*(bn3/nprocs) + k*1
}
```

In order for the synthesizer to discover the details of these methods, the developer has to specify the relationship between the behaviors of `dtrans()` and `trans()`. To do this,

the programmer must specify two things: the relationship between the computation performed by the two functions, and the relationship between the distributed and non-distributed data used by each. These are done by writing a *test harness* function, as shown below:

```
void tester(int nx, int ny, int nz, double[nz,ny,nx] A, ref double[nx,ny,nz] B) implements trans
{
    assume nz % nprocs == 0 && nx % nprocs == 0;
    spmdfork {
        double[nz/nprocs,ny,nx] LA = distribute(A);
        double[nx/nprocs,ny,nz] LB = distribute(B);

        dtrans(nx, ny, nz, nprocs, LA, LB);

        collect(A, LA);
        collect(B, LB);
    }
}
```

The `tester` function is a test harness, as is designated by the keyword `implements`, which states that `tester`, at the function level, should perform equivalent computation to what `trans` does. The first line of `tester` uses the `assume` statement to narrow the scope of the input space being tested: in this case, we are only interested in the cases where the X and Z dimensions are both divisible by the number of processes. The functional equivalence is checked exhaustively for a bounded set of inputs satisfying the assumptions.

We can see that the above concepts are much like what one would specify in a typical unit test: the function being tested, the expected behavior, and some preconditions under which the test should be run. An important difference is that while a unit test runs dynamically, MSL checks the test harness statically at synthesis time, as we have promulgated in Chapter 1.

The `tester` function works by distributing work to a set of parallel processes, each executing the same code, as is specified by the `spmdfork` construct, which indicates that the code in its body is to be executed by parallel processes. On each of these processes, the code works like this: first get `LA`, each process's local portion of the input array from the global array `A` (and likewise, get `LB` from `B`); then call the `dtrans` function on the distributed local arrays `LA` and `LB`; and finally collect the possible changes in `LA` and `LB` back into the global arrays `A` and `B`.

From the above description we already see that in order to relate the computation, the programmer needs to provide facilities to relate the distributed and non-distributed data layouts, used by SPMD and sequential programs, respectively. This is done by writing the `distribute()/collect()` functions, which partition the global array along its slowest growing dimension. It is important to note that these functions will not be part of the kernel we are trying to generate; the programmer is providing them simply as a means to describe how the different representations of the data are related so the system can reason about the equivalence of the two implementations, thus these functions are only used at synthesis time.

We have used the strategy described above to synthesize more sophisticated kernels from NAS parallel benchmarks. As in the example, the synthesizer is leveraged to avoid having to specify low level details regarding loop iteration bounds and array indexing patterns. Just as important, synthesis allows us to reuse the same patterns to implement many related computations without having to write additional code. For example, in the case of transpose in this section, the same generators can be reused to transpose any pair of dimensions. Without synthesis, the usual approaches to coping with such repeating code patterns are to (a) copy and paste the source code and change low level details manually, which is prone to introducing bugs [67]; (b) wrap the implementation strategy in generic subroutines, specifying details as extra parameters, and manually defining the extra parameters in a “magic number table” (as is done in the official Fortran implementation of FT), which is also error-prone, and introduces runtime costs; or (c) write a generic implementation like the previous case, but in C++ as templated functions, and rely on the C++ meta programming feature to specialize the generic implementation at compile time, thus to reduce the runtime costs. This approach still suffers from the problem that the generic implementation would be hard to write and check, and harder to debug if things go wrong, which is compounded by the difficulty of C++ meta programming and poor debugging support for templates. To make things worse, the mainstream C++ compilers are still not completely compliant to the standard, particularly on the more “sophisticated” subset of C++ such as templated programming, so programs using those templating

techniques that behave correctly on one compiler might behave differently or even fail to compile on another compiler, which hinders the pervasive adoption of the C++ meta programming approach.

The synthesis features in MSL help make the implementation strategy generic yet clean and performant, and equivalence checking helps prevent the kinds of bugs introduced by specifying low level details manually.

1.2 Contributions

Overall, this thesis makes the following contributions.

- We describe a new language, MSL, which builds on previous work on constraint based synthesis and generative programming to support programming of high-performance kernels on distributed memory machines.
- We describe a mechanism for expressing bulk-synchronous SPMD computation in MSL and demonstrate an encoding that reduces synthesis of such computations to a tractable sequential synthesis problem.
- We implement a strategy for generating efficient distributed MPI code from synthesized MSL kernels. Our generated code obtains comparable efficiency to hand-written code in Fortran/C++ with MPI.
- We evaluate the expressiveness, performance and reusability of code in MSL by implementing core parts of three distributed applications (3D matrix transpose, Sparse Matrix Vector Multiplication (SPMV), and MG from the NAS parallel benchmarks). We show that MSL does not sacrifice performance compared with hand-written Fortran with MPI code, even when running across thousands of cores.

1.3 Structure of the thesis

Chapter 2 reviews the previous work that are related to this thesis. Then Chapter 3 lists the key language features of MSL, focusing on the unique features for synthesis and bulk-synchronous distributed programming, which allow the programmer to implement a SPMD kernel with MSL. Chapter 7 will expand MSL to include additional features that allow the programmer to further combine multiple kernels in MSL with correctness guarantees. Chapters 4 through 6 describes how MSL programs are synthesized and generated, as depicted in Figure 1-4.

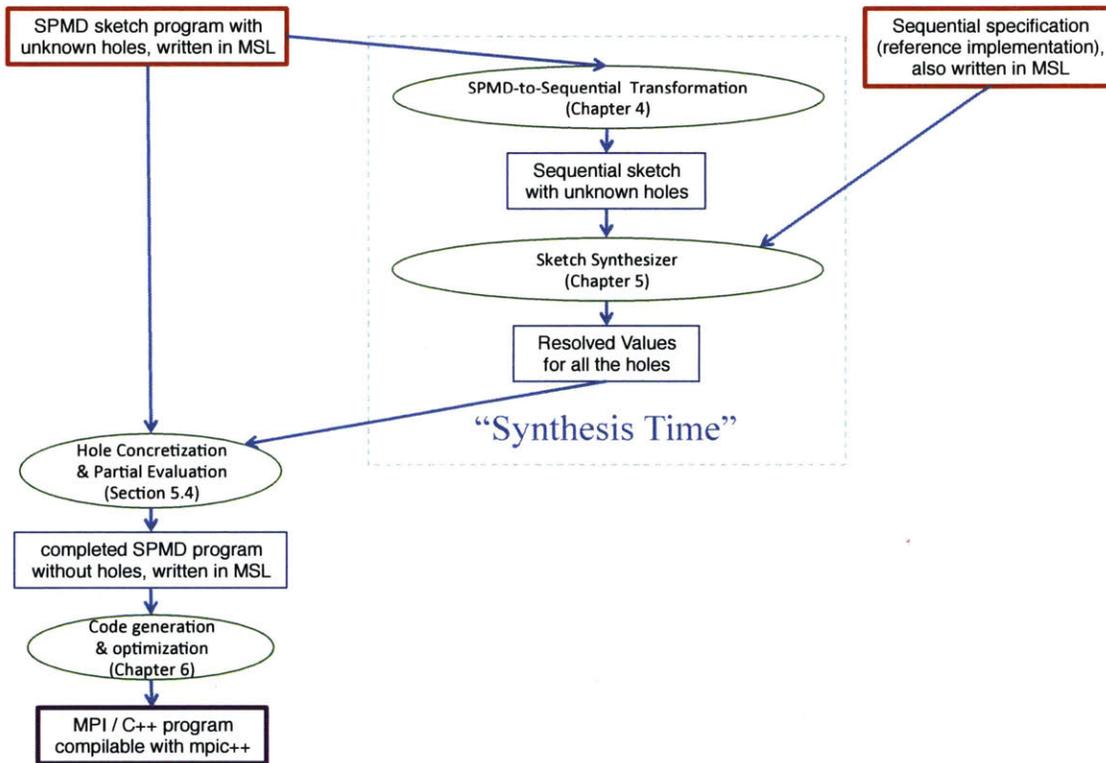


Figure 1-4: How MSL programs are processed

Rectangular boxes represent entities, and oval boxes represent steps to process the entities, annotated with the chapter or section numbers to present the steps in detail.

On the top are user-provided input. At the bottom is the final output.

The high-level strategy employed by MSL is to reduce synthesis of SPMD programs to the problem of synthesizing purely sequential programs, and then rely on the SKETCH solver to synthesize the program. The first step is to transform an SPMD program written in MSL into a deterministic sequential program. Chapter 4 details

the transformation process, and proves that the transformation is sound. The next step is to feed the transformed sequential program into a *synthesizer*, in order to find out the unknown parts in the program, so that the completed program is functionally equivalent to a *specification* (another sequential program written by the programmer, also in MSL). Chapter 5 talks about the synthesis and correctness checking process, which is based on previous work [84, 87] on constraint-based software synthesis for sequential programs. The above two steps are called to be at “synthesis time”, and they are solely for the purpose of analyzing and completing the SPMD program. By no means do they limit the parallelism of the generated code.

Then the information found by the synthesizer is used to complete the original SPMD program written in MSL, by means of *hole concretization* and *partial evaluation*, which are standard compiler passes and will be briefly explained in Section 5.4. Finally, this completed SPMD program is translated into efficient, distributed C++ source code. The generated C++ source code can then be used within a larger application written in languages other than MSL. Chapter 6 presents this code generation process, focusing on how to implement the synchronization features in MSL efficiently based on the MPI library. This is nontrivial because a naïve syntactic translation from MSL to C++ would result in unacceptable synchronization overhead and poor performance.

Chapter 8 shows several examples of using MSL for distributed implementations, taken from NAS parallel benchmarks [3]. It also compares the performance of MSL programs with hand-written official Fortran / C++ implementations.

Chapter 2

Related work

2.1 Distributed memory SPMD programming for high performance computing (HPC)

During the period of 1970s to 1980s, the area of high performance computing was concerned with vector processors and symmetric multiprocessors (SMP). Starting from late 1980s, the focus of interest transitioned to distributed memory systems, in which many processors are connected by network, each with their private memory space, and data must be explicitly exchanged by passing messages. From early 1990s, the Message Passing Interface (MPI) [35] has become the *de facto* standard for programming such massively parallel distributed memory systems. For a brief history of high performance computing, see [6, 29, 102].

MPI provides a general scheme for launching parallel processes and doing communications between processes, and there are multiple programming models that build upon the MPI standard. For example, the master/slave model [44] is widely used to achieve *task parallelism*. However, by far the dominating programming model for high performance computing is the SPMD model: in this model, the problem domain (and the data associated with the problem) is *partitioned* into multiple parts, and the MPI runtime launches multiple processes to process the data; each process has its unique process ID, using which it can get its own position in the partitioned problem space,

and its own data partition; all processes run the same program from start to end, and they communicate with each other by the message passing mechanisms in the MPI library.

A very useful subset of the SPMD model is the *bulk-synchronous* model. In this model, the execution of a program is split into a sequence of *supersteps*. Each superstep consists of a *computation phase* and a *communication phase*. During the computation phase, all processes perform purely local computations with only data in their private memory, concurrently and independently with each other. During the communication phase, they exchange data with one another. Usually it is needed to insert a barrier at the end of each communication phase, in order to synchronize the progress of all processes, so that the incompleting communications of the previous phase would not affect the next phase.

The execution traces of many MPI programs fall into the bulk-synchronous model, although the source code of those programs might not be organized explicitly in accordance with the bulk-synchronous style.

One of the key design choices of MSL is to enforce the bulk-synchronous requirement at the language level, and use that to facilitate the analysis of the SPMD programs written in MSL. Another important feature is to remove the barriers for point-to-point communications while preserving the semantic correctness.

2.2 General Purpose Languages for HPC

Distributed-memory languages for high performance computing include Titanium [43], a high-level SPMD language with Java syntax; Universal Parallel C (UPC) [92], an extension to C for SPMD programming on distributed memory machines; and X10 [17], a distributed language with a notion of *places* that correspond to local state in our model. These have generally similar goals to MSL — facilitating high performance computing with a high-level language that captures the commonly used programming paradigms, and some of them have similarly exploited the advantages of the SPMD execution model, but all of them lack the extensive generative programming and

synthesis capabilities of our language. They also depend heavily on sophisticated runtime systems, like X10RT [40] for X10, and GASNet [8] for Titanium and UPC; while MSL has a very thin runtime library and relies mostly on the well-adopted MPI system.

Titanium [43] is a high-level HPC language with Java-like syntax, SPMD as its execution model — the same as MSL, and PGAS (Partitioned Global Address Space) as its memory model, which has some close similarity to the conceptual memory model we use to analyze and sequentialize MSL programs (detailed in Chapter 4). In Titanium, a unified global address space is *partitioned* into distributed, local memories, which is named *demesnes*, and each *process* is associated with one demesne. Clearly the concept of demesnes in Titanium is the same as that of the local states in MSL. The difference is that while Titanium programs run under PGAS model, in which a process is allowed to directly access another process’s local memory (or *demesne* as they call it), MSL only utilizes this PGAS-like concept for the analysis of programs at synthesis time, and at runtime MSL programs (compiled down to MPI/C++) run under totally isolated memory spaces, and processes can only communicate with one another through message passing.

Like MSL, Titanium also relies heavily on the bulk-synchronous property for analyses and optimizations. In Titanium, the main synchronization mechanism is *barrier* and other collective operations like *exchange* and *broadcast*. All these synchronization procedures are treated as having *global effects*, as well as all the functions that might call (directly or indirectly) into them. Anything with global effects must be called in the bulk-synchronous manner, just like what is required in MSL. Titanium is a major inspiration to MSL’s exploitation of bulk-synchrony. The main difference is that Titanium uses a delicate type system (with a focus on *single-valuedness*) to enforce bulk-synchrony, which is found [53] to be overly restrictive and hard to use; while MSL takes the *static-dynamic checking* approach and removes most of the annotation burden on programmers.

UPC [92] is quite similar to Titanium in its memory model and parallel execution model, but its syntax is based on C, and it is much more flexible and closer to bare-metal.

In addition to the well-structured bulk-synchronous mechanisms as in Titanium, UPC adds other synchronizations like locks, and actually allows for non-bulk synchrony.

Similar to Titanium and UPC, X10 [17] is designed specifically for parallel computing using the PGAS model, and it focuses especially on executing *asynchronous tasks*, thus its model is dubbed as APGAS (asynchronous partitioned global address space). An important concept in X10 is the notion of a *place*, which is similar to the local state space in MSL, staying on one node in the distributed computing system. An asynchronous task (also called an *activity*) can take place either locally, or at some remote *place*. In X10, a task can *spawn* children tasks, possibly executed asynchronously at remote places, and can wait for the completion of the children tasks. A common idiomatic usage is to spawn a remote task for every place (process) in the system, thus mimicing the fork-join style parallelism (also similar to the semantic of `spmdfork` in MSL).

The flexibility of X10 asynchronous tasks makes it powerful to express many useful distributed programming schemes, in particular those taking advantage of task-level parallelisms, yet leaves the room for concurrency bugs to sneak in. Injudicious use of the X10 parallel primitives might introduce deadlocks and race conditions, or impose high synchronization overhead and impede the performance, and that is why it is advocated [89] to stick to the best practised idioms whenever possible. Those common idioms include models not covered by MSL like recursive parallel decomposition and active messaging, as well as the important SPMD paradigm, which is the focus of MSL.

MSL takes a different position from X10 by *enforcing* rather than advocating the programmers to stick to the good discipline, and by taking advantage of the disciplined bulk-synchronous SPMD model, the compiler can analyze the program more easily and effectively.

Other efforts to define future parallel languages include Chapel [15], a language aimed to provide a general solution to support all kinds of parallelisms; High-Performance Fortran (HPF) [34, 78], a language extension to Fortran with focus on constructs to declare and process distributed arrays and exploit data parallelisms.

2.3 Domain-Specific Languages for HPC

To ease development of high performance parallel software, a number of domain-specific languages have been developed, including for example PDE solvers [28] and stencil computations [19, 54]. Such packages require large effort to build entire compiler toolchains and must still retain interoperability with user programs written in general purpose languages. As a result, recent work has concentrated on building DSEL frameworks such as Asp [55] and Delite [10] that leverage advanced metaprogramming infrastructure to reduce that effort.

MSL, in contrast, combines generative programming and synthesis to reduce the effort substantially; however, the tradeoff is that it is more difficult to deploy efficient domain-specific reasoning engines as the DSEL systems do.

ZPL [16] is an array programming language designed for fast execution on both sequential and parallel computers. It provides a convenient high-level programming medium for supercomputers and large-scale clusters with efficiency comparable to hand-coded message passing. The central design of ZPL is an extended notion of arrays and a powerful set of array operations. By organizing programs based on arrays, ZPL gets *data parallelism* easily, and a seemingly sequential program in ZPL can be automatically translated into an SPMD program by partitioning the arrays and distributing the array operations to multiple machines. There are recent efforts in this kind of DSEL approach, like Julia Distributed Arrays [50, 51]. MSL also provides native support for convenient array manipulations, but the arrays in MSL are not distributed automatically: we provide the `distribute/collect` mechanism for the programmers and library writers to design their own array partitioning schemes, and rely on the synthesis feature to ease the development, rather than requiring the users to choose from a set of pre-defined array partitioning schemes.

Halide [77] is a DSL for writing high-performance stencils and combining them to perform computation-intensive tasks like image processing. It supports to represent and explore the complex tradeoff space involving parallelism, locality, and redundant recomputation of shared values in stencil pipelines, which is fundamental to get high

performance in image processing applications. Its language design offers a systematic model of the tradeoff space, a scheduling representation that spans this space of choices, and an autotuner that searches this tradeoff space and infers high performance schedules. The inferred algorithms can be up to 5x faster than hand-optimized kernels written by experts. Distributed Halide [25] extends Halide to further support distributed memory programs. Helium [72] is a system that lifts computation kernels in legacy binary code to high-level Halide DSL code, and get improved performance. The fundamental philosophy behind the Halide approach and MSL is similar: they both allow the user to represent some design space, and rely on the system to find out a suitable point in the space. In Halide, programs in this design space are all correct (functionally equivalent), and the searching procedure (based on stochastic search) aims to find out the one with better performance. Because Halide has separated algorithms and schedules, once the algorithm is written and fixed by domain experts, the problem of finding a performant schedule can be done with an autotuner or manually by performance experts. In MSL, we rely on the programmers' insights to provide a sketch that represents a *skeleton* of a high-performance program, but this skeleton has unknown parts that the programmer wants to omit. Thus the design space in MSL contains a lot of wrong, or even ill-formed programs, and the searching procedure (based on constraint solving) aims to find out the one that is functionally correct.

Simit [58, 59] is a DSL for writing high performance physical simulation code. Its main innovation is to be aware of the local-global duality at the language level: to achieve high performance in this area, programmers often need to switch back and forth between a local, graph view and a global, matrix view of the data, and make optimizations on both views. Simit allows the programmer to define mapping functions between the two views, which is *not* an imperative instruction for Simit to materialize a (global) matrix in (local) memory: rather, it is an abstract definition of the matrix. This concept of local-global duality and mapping functions draw an intriguing parallel to that of the local/global states and distribute/collect functions in MSL, alluding behind them a general methodology, applicable to a wide range of high performance computing problems.

2.4 Software Synthesis

In general, synthesis helps derive correct and efficient programs from specifications or higher level models. High-performance libraries such as Atlas [100] and SPIRAL [76] use high-level representations of computation, combined with derivation rules, to synthesize optimized code for particular classes of computation.

In contrast, constraint-based synthesis, which is used by SKETCH [84, 87] and MSL, relies on generalized solvers and can deal with general programs, but at a much higher cost of solving. Prior work has applied constraint-based synthesis to find function inverses [88], to reverse engineer malware [46], and even to automatically grade programming assignments [82]. There has even been recent work on frameworks to make it easier to create synthesis-enabled domain specific languages [90, 91]. To our knowledge, however, this is the first work that uses constraint-based synthesis in a general language that can be automatically converted to practical MPI code.

In the domain of software synthesis, the previous work on SKETCH language is most relevant to our work, as MSL uses SKETCH as its underlying synthesis solver. In SKETCH, the programmer provides an incomplete implementation with unknown parts in it, and she also provides a complete program as the specification. The SKETCH solver uses a technique called Counter Example Guided Inductive Synthesis (CEGIS) to find the unknown parts and complete the implementation, so that the implementation and the specification are functionally equivalent, and uses bounded model checking to check that equivalence and find bugs in the implementation. Both the implementation and the specification in SKETCH are written in a sequential imperative language.

In MSL, we extend the language to include SPMD style parallel execution model, and we have carefully put restrictions on the communication patterns to make the programs in MSL deterministic, so that we can transform the parallel programs into sequential ones, and re-use the SKETCH synthesizer to find out the unknown parts in the implementation and check for bugs.

There has also been much prior work synthesizing concurrent data structures [41, 64, 70, 86, 94]. They differ significantly from this work because the key problem

they deal with is concurrency as well as non-determinism introduced by concurrency, whereas we assume determinism directly from our model.

2.5 Generative Programming

Generative programming adds a separate stage (*generating stage*) into the process of program development. In the generating stage, a program *generator* generates parts of or the whole program, which is later compiled and executed. The advantage of generative programming is usually higher level of code re-use, avoidance of repetitive code typing, and advanced domain-specific optimizations that are hard to do by hand or by general-purpose compilers. This technique is widely used in the domains of scientific computing and high-performance computing.

Expression templates [95] are a metaprogramming technique in C++ which uses templates to create domain specific embedded languages (DSEs). This form of generative programming is essentially using C++ template concretization (specialization) stage as a program generator. While expression templates enhance the ability for C++ programmers to include metaprogramming in libraries, they require substantial effort and complexity, even when using systems such as Boost Proto [73]. Modern efforts to provide fast scientific libraries have incorporated C++ template metaprogramming to provide reusable abstractions, including the Epetra and Tpetra packages in Trilinos [42] and the Matrix Template Library [81].

Other generative programming examples include macro-based metaprogramming such as that in Lisp, newer versions of Scala [12], the *staged function* feature in Julia [7] (recently renamed *generated function*), and Terra [27]. Those are more powerful than the generative programming mechanisms in C++.

Lightweight Modular Staging (LMS) [79] is a language extension to Scala that provides a framework to facilitate generative programming. One key feature that sets LMS apart from other generative programming techniques is its extensive use of types within the host language (Scala) itself to distinguish stages, so that they can use the type inference mechanisms in Scala to do local binding-time analysis. Using this

technique, they can implement generative programming as a library with very minimal language support. LMS also provides convenient mechanisms to transform and interpret the staged IR, and a set of libraries to do optimization and code generation. People have applied the LMS approach to parallel computing [14, 65], high-performance math kernels [74], database systems [60], and web frontend [62].

The *generator* mechanism provided in MSL is a special form of generative programming. Inherited from SKETCH, it is especially powerful due to its integration with synthesis mechanisms, because the programmer can omit tedious details when developing the generators, and rely on the synthesizer to figure them out. We have found this combination of generative programming and synthesis particularly useful to distributed memory high performance computing, because in this area lots of low-level details are critical to the performance, yet they are derivable from some clean, high-level insights in the form of generators. This technique could also be used to enhance generative programming support in Scala or C++ if it were integrated with those languages.

Roughly speaking, there are three major concerns in generative programming:

1. The design of the *staged IR*. The staged IR is the intermediate representation of the program that the user wants to generate in the generating stage. In most C++ based generative programming tools like expression templates, the staged IR is a set of templated types. In macro-based metaprogramming like Lisp, the staged IR is a complete representation of the host language, like s-expressions. The MSL has a built-in support for generator functions, so the staged IR is just the normal IR for the MSL compiler.
2. Producing the staged IR: how does a user provide staged IR to the program generator. This is the user interface of generative programming languages, so those languages usually strive to facilitate and ease this step, and to make it similar to writing code in the host language. In expression templates, this step is implemented by operator overloading and template specialization. In macro-based approaches, this step is implemented by the *quote* features provided by the host languages. In MSL, this step is tagging a function with the `generator` keyword, and using generator functions

in the program.

3. Interpreting the staged IR: translating the staged IR into the host language or some other language, to feed into a compiler, so that the later stages (compiling and execution stages) can happen. This step gives semantics to the staged IR. Usually there are also some *transformations* of the staged IR to perform domain-specific optimizations. In expressoin templates, this step is implemented by template concretization. Macro-based approaches usually relies on the *eval* mechanisms provided by the host languages. In MSL, this step is done by the synthesizer and the partial evaluator: the synthesizer finds all unknown parts in the generator functions and concretizes them, and the partial evaluator figures out what parts in the program can be evaluated to constants at compile time and uses this information to aggressively optimize the generator functions as well as normal functions. Because the generators are always fully inlined into their calling contexts, thus get concrete actual arguments in place of formal parameters, they are especially amenable to partial evaluation, and often yield much simpler and faster generated code than normal functions would.

2.6 Verification of Parallel Programs

Automatic static verification approaches for parallel programs employ similar simplifications to those we use. PUG [66] and GPUVerify [4] both take advantage of barriers to limit the size of epochs over which to verify and use *two-thread reduction* to turn execution of multiple threads into just two threads. This reduced execution can be sequentialized, enabling verification using traditional methods. For MPI programs, considering only a subset of all interleavings and state simplification are both necessary to make verification tractable [38, 80].

2.7 Compiler-based Optimization

There is a long history of compiler optimizations to improve performance, including the polyhedral model [9, 39, 61] for loops and automatic vectorization techniques targeting

SIMD architectures [47]. Our approach explicitly requires programmer guidance for optimization. This has the advantage that it enables aggressive optimization strategies that do not apply for all programs, but the tradeoff is that our approach does not guarantee full correctness. In essence, MSL provides mechanisms for aiding programming and optimization while compilers strive for fully-automated and fully-general optimization strategies.

2.8 Autotuning

Due to the failure of general compilers to achieve high percentages of peak performance, kernels that require highest performance have begun relying on autotuning to obtain efficient code. This empirical approach generates a large space of implementations for a kernel and runs them to find the best-performing variant on each platform. Atlas applies this approach to dense linear algebra, but others [36, 54, 56, 97] have extended this to other important domains and to compilers [2, 49]. Variants are created either by applying compiler optimizations with different parameters or through domain-specific scripts. Our approach is somewhat orthogonal; programmers must create their own MSL implementations for each variant, but these could then be autotuned.

Chapter 3

The MSL programming language

The MSL language builds on previous work of SKETCH, a synthesis-enabled language [87] for purely sequential imperative programs, with the addition of a clean mechanism to describe SPMD parallel execution model and bulk-synchronous communication patterns.

The design of MSL revolves around the problem of enabling programmability without sacrificing two key goals: a) efficient code generation, and b) deep semantic analysis to support synthesis and help discover bugs. In the remainder of this chapter, we describe the key language features that the readers need to know about MSL. Many of the MSL features can also be found in the SKETCH programming manual [83], because they have now been incorporated into the main distribution of SKETCH, after initially developed for MSL.

3.1 Core imperative language

At its core, MSL (just like SKETCH) is a imperative language that borrows most of its syntax from C. Thus it has the primitive types like `int`, `double`, `void`, the C-like ways of declaring variables, composing expressions, forming statements, and defining functions. We do not try to repeat these basic imperative language features here, and interested readers may refer to the SKETCH programming manual [83] for a complete description. We will highlight a few distinct language features in SKETCH (and MSL) in the rest

of this section.

3.1.1 Native array support

In MSL, you can define arrays of any fixed dimension, provided the lengths of each dimension. We start with the syntax for defining one-dimensional arrays.

```
 $\tau[N]$  a;
```

The above code defines a one-dimensional array `a` of length `N`, and its base type (type of its elements) is `τ` . For the above code to be valid, there must be an integer variable named “`N`” within the scope of the array definition.

Arrays can also be used to declare function parameters and return type:

```
int[m+1] f(int m, float[m*2] x) { ... }
```

In the above code, the type of parameter `x` is an 1D array of `float`, with length `m*2`; the return type of `f` is an 1D array of `int`, with length `m+1`. The parameter “`m`” can be used in the types of the subsequent parameter definitions, the return type declaration, and type definitions in the function body.

Array Indexing

Arrays in MSL are 0-based, and the syntax for array indexing is standard: `a[i]` selects the *i*-th element from array `a`, given that $0 \leq i < N$, where `N` is the length of `a`. At synthesis time, the synthesizer automatically adds assertions like “`assert 0 <= i && i < N;`” to check for array bounds violations. This catches a lot of low-level bugs, which are prevalent in programs written in languages like C. As we have noted in Chapter 1, these assertions are only added and checked at synthesis time, *not* in the generated C++ code, thus the performance of the generated code is not affected.

Bulk array access

Just like such languages as ALGOL, Go, Python, and Matlab, MSL allows the programmer to select a consecutive range of elements from an array and form another array, which is called *bulk array access*: Suppose `A` is of type `$\tau[N]$` , then `A[i:len]` is a valid

expression of type $\tau_{[len]}$, given that $i \geq 0$, $len \geq 0$, and $i+len \leq N$. $A[i:len]$ represents a (possibly smaller) array consisting of the i -th to the $(i + len - 1)$ -th elements of A , and it can be used in any place where an array of type $\tau_{[len]}$ is expected. Note that bulk array access does not create a new array: it just refers to a subset of memory locations occupied by the original array. If some element is changed through bulk array access, the exact change is reflected in the original array.

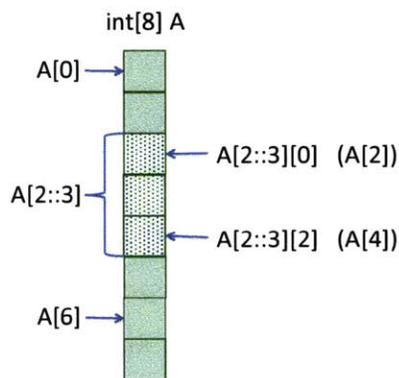


Figure 3-1: Example of bulk array access

Figure 3-1 shows an example of bulk array access: A is of type `int[8]`, representing 8 elements grouped in the memory contiguously. $A[2::3]$ is another array of type `int[3]`, formed by grouping a subset of the elements in A . Because $A[2::3]$ is a valid array expression, it can be used like a normal array: specifically, it can be indexed like a normal array. $A[2::3][0]$ means the 0-th element of the array $A[2::3]$, which is the same as $A[2]$; likewise, $A[2::3][2]$ is the same as $A[4]$.

Reading from and writing to the elements in a bulk array access (like $A[2::3][2]$) is automatically translated into the corresponding operation to the elements in the original array (like $A[4]$).

Array efficiency

The representation of arrays in MSL is very efficient, in terms of both space and time, because the arrays are *plain and flat* data types like in C: an array of length N with base type τ is just N elements grouped in the memory, contiguously, and each element is of type τ . Unlike Java arrays and C++ vectors, no metadata such as the array

length is stored along with the array, only the data of the array elements is stored.

The array length is a very useful piece of information for compiler analysis, optimization, safety checking, and bug detection, so many languages like Java and Python choose to store it along with the array contents, which is convenient but may also introduce redundancy, as oftentimes the length information is already stored in some variables used by the program, and in many cases the programmer creates many arrays of the same length (like an array of same-sized arrays), while multiple copies of this unique length is stored wastefully. Low-level languages like C, on the other hand, only use the length when allocating an array (either statically or dynamically), and lose it afterwards. If the programmer wants to use the length of an array, she must remember to keep and pass it around properly, entirely by good coding practice (readers familiar with the source code of web servers or image processing applications might naturally recall many functions expecting a pair of `char *` and `size_t` as parameters), which is often laborious and error-prone. That is why it is difficult if not impossible to tell the length of an array (represented as a good old C pointer, of course) in a C program.

The representation of arrays in MSL (inherited from SKETCH) is based on the notion of *dependent type* — a type whose definition depends on a value rather than just other types. Specifically, any array type in MSL is required to depend on the length (a value): whenever an array occurs in the program, its length expression must be valid in the scope, i.e. its length must be an expression composed from constants and variables accessible in the scope where the array occurs. This guarantees that whenever the length information is needed, the compiler can always use those values in the length expression to re-construct the needed length. To illustrate this, consider this code snippet below:

```
int[m+1] f(int m, float[m*2] x) {
    x[0] = 1.0;
    int[m+1] a;
    return a;
}

void main(int m) {
    float[m*2] x;
    int[m+1] a = f(m, x);
    a[0] = 2;
}
```

In the code above, `m` is necessary for computing the length expressions of the four array variables (of two array types) in two functions, but the compiler really needs not store an extra copy of `m` somewhere: `m` is readily in the scope wherever it is needed to compute the array lengths.

We can see that this dependent type requirement is not just for the creation of arrays (as in C), but also for any use of arrays. Combined with the fact that MSL is statically and strongly typed, meaning that every variable must have a well-founded type at compile time, this requirement shuts the door to represent a “bare” array without length in MSL. Thus the type system actually enforces the discipline to pass around the length information along with the array, as we have seen in the above example, and better yet, MSL checks type congruity (you cannot pass a smaller array to where a bigger array is expected) and detects bounds violation for arrays. This prevents a wide range of notorious bugs like buffer overrun caused by playing with array lengths recklessly. In the meantime, the static-dynamic checking approach (defined in Chapter 1) used by MSL avoids most of the annotation efforts from the programmers when type checking dependent array types.

Nested arrays (multi-dimensional arrays)

In MSL, the base type of an array can itself be an array type: then we get a *nested array*, like the example code below:

```
int[4][8] A;  
T[n][m][k] B;
```

`A` is an array with 8 elements, and each element `A[i]` ($0 \leq i < 8$) is an array of type `int[4]` (an array with 4 `int` elements). `B` is an array with `k` elements, and each element `B[i]` ($0 \leq i < k$) is an array with `m` elements, while each element `B[i][j]` ($0 \leq i < k, 0 \leq j < m$) is of type `T[n]` (an array with `n` elements).

At first glance, a nested array in MSL is like an array of arrays in Java (or a vector of vectors in C++), but they are vastly different species, because

1) At each nesting level, the outer array stores the *contents* of the inner arrays themselves as elements, *not the pointers* to the inner arrays, so all the elements occupy

a set of contiguous memory locations.

2) Arrays are plain and flat: the content of an array is just the cartesian product of all the contents of its elements; array length is not stored in the array itself. This requires that all the inner arrays are *homogeneous*: they have the same base type and length. This is in contrast to the case in Java: a variable like `int[][] x` can store inner arrays of different lengths: `x[0]` and `x[1]` can have different lengths, because each inner array stores its own length information together with the array elements in Java (it is the similar case with vectors in C++).

Figure 3-2(a) illustrates a nested array `A` of type `int[4][8]`, so each array element `A[i]` ($0 \leq i < 8$) is an array of type `int[4]`. In the picture, we have emphasized one of the element `A[6]` by slashed patterns. `A[6]` is an array with 4 `int` elements in it. We have also emphasized a bulk array access `A[2::3]` with dotted pattern: `A[2::3]` is of type `int[4][3]`, consisting of 3 elements, and each element is of type `int[4]`.

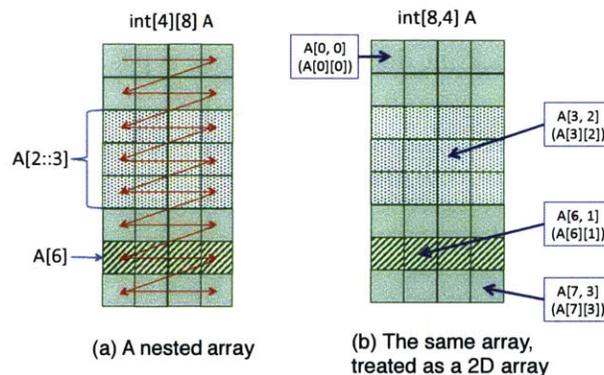


Figure 3-2: Example of an nested array, and the same array treated as a two-dimensional array

The red arrows in Figure 3-2(a) show the growing order of the memory locations, from which we can see how elements of each inner array are arranged contiguously in the memory, and all the inner arrays are in turn arranged contiguously in the memory occupied by the outer array.

Because of this property, `A` is naturally *homomorphic* to a *two-dimensional array* of 8 by 4 `int` elements, as is shown in Figure 3-2(b).

In fact, all nested arrays in MSL can also be viewed as multi-dimensional arrays. MSL provides an alternative way of declaring and indexing them: when declaring an array type, $\tau[l_{en_1}, \dots, l_{en_k}]$ is a syntactic sugar for $\tau[l_{en_k}][l_{en_{k-1}}] \dots [l_{en_1}]$; when indexing an array A , $A[i_1, \dots, i_k]$ is a syntactic sugar for $\tau[i_1][i_2] \dots [i_k]$.

The seemingly asymmetric notations for declaring and indexing an array is intentional: in this way, a number (length for declaring an array type, or index for selecting an array element) closer to the right bracket (“]”) always corresponds to a faster-growing dimension, and a number closer to the left bracket (“[”) always corresponds to a slower-growing dimension. So the multi-dimensional arrays in MSL are *row-major*, different from languages like Fortran.

In Figure 3-2(b), the array A is viewed as a two-dimensional array (`int[8,4] A`), and several elements of A are picked to show the two equivalent indexing methods. We can see that the number on the right of the comma (“,”) always refers to the unit-stride dimension, whether it’s used to declare the type of A or to index elements; the number on the left of the comma always refers to the slowest-growing dimension.

Note that in the current implementation of MSL, any multi-dimensional (or nested) array is transformed into a one-dimensional array of the same element type, and the compiler uses the size information declared in the array type to map the multiple index accesses to single index accesses, and adds assertions to detect array bound violations, which are checked at synthesis time by a bounded model checker.

Array views

Because any array, regardless of its dimension, is flattened into an underlying one-dimensional array, it is possible (and convenient in some cases) to view the same array under different dimensional configurations. MSL supports creating *array views*, as shown below:

```
int[n, m] a;
view a as int[m*n] b;
view a as int[2, n/2, m] c;
```

In the above example, both `b` and `c` are views of `a`, but with different types (dimensions). At synthesis time, MSL examines the array types and adds assertions to enforce that

an array view cannot be bigger than the original array. Array views provide an easy way to access the same underlying one-dimensional array under different dimensional configurations. Similar functionality is present in languages such as Fortran and HPF [78], and in distributed array libraries such as Adlib [13].

3.1.2 Implicit parameters

A parameter to a function can be defined as *implicit* if its value can be inferred from the latter parameters, and the caller of the function can omit passing the implicit parameters, as illustrated below:

```
void f([int n, int m], int[n] a, double[m] b) { ... }

int[5] A; double[t] B;
f(A, B); // caller of f
```

In the above code, the caller of `f` implicitly passes `5` (as `n`) and `t` (as `m`) to `f`, based on the type of `A` and `B`.

3.1.3 Pass-by-reference parameters

In MSL, a parameter of a function can be declared as `ref`, which means the function can pass value out to its caller through the parameter, as below:

```
void f([int n], ref int[n] a, ref int x) {
    a[0] = x;
    x = x+1;
}

int[2] A = {0, 0};
int X = 5;
f(A, X);
```

In the above code, the value of `A` becomes `{5, 0}` and `x` becomes `6`, because both `a` and `x` are declared as reference parameters in `f`. Note that unlike in C, even array parameters are pass-by-value (copying semantics) by default, if not declared as `ref`.

The reference parameters of a function are enforced to be strictly non-aliasing to each other in MSL, similar to what Fortran requires for array parameters.

3.2 Basic Synthesis Features

The SKETCH language [87] first showed how to add synthesis capabilities to an imperative language by combining unknown integer constants with *generators*, and we review the key features here.

3.2.1 Unknown integer holes

The programmer can denote an unknown integer constant by the token `??` (also called a *hole*), which can be put at any place where an integer can occur in the program. The integer hole acts as a placeholder that the synthesizer must replace with a concrete value. The synthesizer's goal is to ensure that the resulting code avoids assertion failures for any input in the input space under consideration.

3.2.2 Choice expression

The programmer can specify a fixed but unknown choice among several branch expressions by a *choice expression*, with the syntax `{|·|·|}`. Note that there can be any fixed number of branches, as long as all the branch expressions have the same type, like this: `{| a | b | 5 | i-1 |}` is a valid choice expression if all `a`, `b`, and `i` are integer-typed.

The choice expression is just a convenient syntactic sugar on top of the integer holes. It is easy to see that by the following example:

```
int x = {| a | b | 5 | i-1 |};
```

can be mechanically transformed to its equivalence:

```
int x;  
switch(??) {  
  case 0: x = a; break;  
  case 1: x = b; break;  
  case 2: x = 5; break;  
  default: x = i-1;  
}
```

So the only essential unknown values that need to be discovered by the synthesizer are the integer holes (`??`).

3.2.3 Assertions

The programmer uses *assertions* to specify what is the correct behavior of the program, so as to guide the synthesizer to find the suitable values of the holes. The synthesizer's goal is to ensure that the resulting code avoids assertion failures for any input in the input space under consideration. For example:

```
void f(int x) {
    int y = x * ??; // synthesizer will find that ?? must be 2
    assert x+x == y;
}
```

The synthesizer will find the correct value (2) of the hole, in order to satisfy the condition of the `assert` statement, for any input `x`.

3.2.4 Refinement relationship among functions

The programmer can also specify the correct behavior of a function `f` by stating that it is a refinement of another function `g` using the keyword `implements`, as long as these two functions have the same function signature (parameter list and return type):

```
int f(int x) implements g {
    return x*??; // synthesizer will find that ?? must be 2
}

int g(int x) {
    return x+x;
}
```

The synthesizer will find the correct values of the holes to satisfy two conditions: a) no assertions in the program are failed; b) the output of the two functions (`f` and `g`) are the same if they are both fed the same input.

The refinement relationship can be mechanically transformed into solely assertions.

The above example can be transformed into the equivalent code below:

```
int f(int x) {
    return x*??;
}

int g(int x) {
    return x+x;
}

void fgTester(int x) {
```

```

int y1 = g(x);
int y2 = f(x);
assert y1 == y2;
}

```

So what the synthesizer really sees are just holes and assertions. Finding values for the holes to satisfy all the assertions is also called *concretization*.

3.2.5 Generator functions

Simple integer unknowns and choices can be used to define spaces of more complex program fragments by packaging them into *generator* functions, or simply generators. A generator can be thought of as a function that will get inlined into its calling context and partially evaluated after all unknowns are resolved. Each invocation of the generator is replaced by potentially different code fragments.

As a simple example, consider the problem of specifying the set of linear functions of two parameters x and y . That space of functions can be described with the following simple generator function:

```

generator int legen(int i, int j) {
    return ??*i + ??*j+??;
}

```

The generator function can be used anywhere in the code in the same way a function would, but the semantics of generators are different from functions. In particular, every call to the generator will be replaced by a concrete piece of code in the space of code fragments defined by the generator. Different calls to the generator function can produce different code fragments. For example, consider the following use of the generator.

```

void main(int x, int y) {
    assert legen(x, y) == 2*x + 3;
    assert legen(x,y) == 3*x + 2*y;
}

```

The synthesizer would generate something like this:

```

void main(int x, int y) {
    assert (2 * x + 0 * y + 3) == 2*x + 3;
    assert (3 * x + 2 * y + 0) == 3*x + 2*y;
}

```

Note how the holes in the generator `legen` get different concretizations for the two different invocations.

3.3 SPMD programming in MSL

MSL supports expressing SPMD style parallelism and bulk-synchronous communication patterns. This section lists the key language features related to SPMD programming. To begin with these features, we first introduce the concept of *barrier-like* behavior, which lies in the center of SPMD programming in MSL.

3.3.1 Barrier-like behavior

The *barrier* is a well-known and widely adopted synchronization primitive. A barrier forces its calling process to wait until all processes have reached the barrier. It imposes two important restrictions on its caller: 1) *all processes* must participate in the synchronization *collectively*; and 2) they must proceed with the barrier *simultaneously*.

A particular kind of barrier that will be of interest to us is *textual barrier*, which can be found in languages like Titanium [52] and an extension to Co-Array Fortran [30]. A textual barrier additionally require that all processes must reach the *same textual instance* of a barrier before any process can proceed with that barrier.

Below are some examples of correct and incorrect usages of barriers, adapted from the seminal work [1] on formalizing a type system for barrier inference.

Example 1:

```
work1();  
barrier();  
work2();  
barrier();  
work3();
```

Assuming that `work i ()` are functions without any synchronization, the above usage is correct, and demonstrates a typical SPMD skeleton: the barriers serve to guarantee that all processes finish `work1()` before proceeding to `work2()`. The only synchronization is at the barriers — processes execute `work i ()` asynchronously and enjoy the full parallelism.

Example 2:

```
if (different()){ barrier(); }  
else { barrier(); }
```

Assuming `different()` can return different boolean values in different processes, the above example is wrong for textual barriers, because although all processes call into some barrier, only a *subset* of processes will call the first barrier, and the others will not, hence violating the *collective* requirement. This will cause the processes that call the first barrier to *hang infinitely* upon it, and the other processes hang infinitely upon the second barrier.

Example 3:

```
if (x) barrier() else work1();
```

This is only correct if all processes agree on the value of the boolean variable “x”. There is actually a more general for the correct behavior of barriers — all processes must agree on the path conditions that control synchronization calls, which will be further discussed in Section 3.3.4.

Barriers are strongly related to the bulk-synchronous parallel execution model: if all the synchronizations in the program are barriers or barrier-like, then the execution of the program is naturally alternating between computation and communication phases: during the computation phase, all processes execute asynchronously and independently; and only upon facing a synchronization procedure like `barrier`, they perform collectively the communication phase, and proceed to the next computation phase simultaneously.

In the remainder of this thesis, we will call a program construct to have *barrier-like* behavior if the call to such construct must be performed by all processes collectively and proceeded simultaneously, just like a textual barrier. Put it another way, if we replace in the program text some call to a barrier-like construct by a call to a textual barrier, the program should still behave well with regard to the concurrency and synchronization aspect (no deadlocks).

3.3.2 Expressing parallelism by `spmd`

In MSL, the programmer can tag a function definition with the keyword `spmd` to require that the function be run in SPMD style parallel mode, like this: `spmd T f(...)`. We

call such functions **spmd** functions.

An **spmd** function must be called by all processes collectively, in the *barrier-like* fashion defined in Section 3.3.1. This is a very strict semantic requirement. Together with the similar requirements for communications, it guarantees determinism and facilitate analysis. At runtime, this requirement is loosened for better performance, but without losing the correctness guarantees. See Section 3.3.4 for more details.

The **spmd** functions can access two pre-defined constant integers: **nprocs** — the total number of processes, which is the same for all processes; and **mypid** — the process ID of the executing process, which ranges from 0 to **nprocs**-1, and is unique for each process.

In the SPMD execution model, each of the **nprocs** parallel process has its own memory, and a process cannot access other processes' memory. This implies that any data declared inside an **spmd** function is considered to be *local* to the executing process, and *distributed* among all executing processes. For example:

```
spmd double[n] f(int n, int[n] x, ref int[n] y) {  
    double[n] z;  
    ...  
}
```

In the above code, The variables *n*, *x*, *y*, *z*, and the return value of *f* are all *local* to their executing process, i.e. conceptually there are **nprocs** independent versions of each variable, one for each process, and process *i* can only access the *i*-th version. If an **spmd** function needs to exchange information with other processes, it has to use the communication mechanism described in the next section.

MSL imposes a restriction upon the usage of **spmd** functions: **spmd** functions can call other **spmd** functions as well as normal functions, but normal functions cannot call **spmd** functions. Section 3.3.4 will explain the logic behind this restriction.

This of course raises a question of who calls the very first **spmd** function, which we will answer later in Section 3.3.5.

3.3.3 Communication/synchronization mechanisms

Communication is achieved through a set of functions that support reductions, all-to-all communication, and point-to-point communication. In the rest of the section we will

list these communication and synchronization functions. Note that we will assume that the elements being transferred (if any) are of **double** type in the discussion, but in the real setting, MSL provides polymorphic communication functions for different element types, such as **int** and **float**. All the communication and synchronization functions must be called by **spmd** functions, and further more, they all have *barrier-like* behavior, so any one of them must be called by either *all processes* or *none of the processes* at runtime, otherwise a *dead lock* will occur. This is enforced by the MSL correctness checking procedure, explained in Section 3.3.4.

Barrier

As we have stated in Section 3.3.1, there is a *textual barrier* in MSL. A barrier does not communicate any content, but forces all parallel processes to sync up to the same place of the code: a barrier blocks its caller (and the executing process) until all processes have entered the same barrier, and then any caller (executing process) can continue. Its syntax is simple:

```
barrier();
```

Note that because `barrier` only affects the interleavings of processes, and any MSL program can be transformed into a functionally equivalent deterministic sequential program (see Chapter 4), introducing barriers will not make any difference on the program state or computation result, but it is useful for things like timing a benchmark: usually people need to wrap the code snippet being timed with a pair of barriers, so as to make sure that all processes enter and leave the code snippet together.

Reductions

MSL supports `reduce`, which is similar to `MPI_Allreduce`:

```
void reduce([int n], double[n] sendbuf, ref double[n] recvbuf, int opcode)
```

For each index i from 0 to `nprocs-1`, all processes reduce their `sendbuf[i]`, and the result is stored to `recvbuf[i]` on every process. `opcode` specifies the actual reduction, which can be chosen from pre-defined constants `SPMD_MAX`, `SPMD_MIN`, `SPMD_SUM` and so on.

All-to-all communication

We have seen in Chapter 1 the `All_to_all` function that sends data from and to all processes. It is an abstraction of the `MPI_Alltoall` function.

```
void All_to_all([int bufsz], double[nprocs, bufsz] src, ref double[nprocs, bufsz] dst);
```

It takes as input a source 2D array containing the data to send to each destination process, and returns a new array with the data received by every process.

Point-to-point communication

Point-to-point communication is supported by `transfer`:

```
void transfer([int n], bool scond, double[n] sendbuf, int rid, bool rcond, ref double[n] recvbuf)
```

Each process passes the information it wants to send through the `sendbuf` parameter and uses `rid` to indicate the id of the intended recipient. When the call returns, `recvbuf` will contain the information received by the process. The arguments `scond` and `rcond` determine whether data will be sent and received by the current process, respectively. A programmer familiar with two-sided messaging models such as MPI can think of `transfer` as encapsulating a Send-Receive communication between two processes, where the call does not complete until the communication is finished.

MSL additionally includes a few variants of `transfer` to handle, for example, the case when the send and receive buffers have to be of different length.

Just like barriers, reductions, and all-to-all communications, `transfer` is also required to be barrier-like. This means that if we only want some subset of the processes to send or receive messages, all processes must still call `transfer`, but those that do not engage in communication can set `scond` and/or `rcond` to false. `transfer` also has two additional constraints: a) each process can only receive messages from one other process for a given `transfer` call; and b) if a process sends a message to recipient r , then `rcond` of process r should be true, and if no process sends to r , then `rcond` of process r should be false. The conditions imposed on `transfer`, along with the bulk-synchronous nature of the program, ensure that all the messages will reach their destinations and that execution will be fully deterministic, as will be discussed in Chapter 4.

3.3.4 Enforcing bulk-synchronous constraints

MSL enforces some constraints on the use of `spmf` functions and the communication / synchronization functions, in order to make all programs conform to the *bulk-synchronous* model. As pointed out in Section 3.3.1, bulk-synchrony is achieved as long as all synchronizations have barrier-like behavior, so the keystone of the bulk-synchronous constraints is to restrict that all synchronizations are used in the barrier-like manner. This requires them to be called by all processes collectively. Following the terminology of Titanium language [43], we say that the control flow leading to a synchronization operation is *coherent* if the portion of the control flow that may affect the synchronization is identical across all processes. Enforcing barrier-like property boils down to checking that the control flow leading to every synchronization is coherent, which requires reasoning about inter-procedural control flow from the beginning of the program (i.e. `main()`), a rather heavy-weight endeavor. In MSL, we strengthen the restrictions and break them into several phases, as shown below:

Phase 1. MSL introduces the notion of `spmf` functions, and requires that all the communication / synchronization functions can only be called from `spmf` functions. Furthermore, `spmf` functions themselves can only be called by `spmf` functions. Following the convention in Titanium, we call both `spmf` functions and communication / synchronization functions to be *collective*. This rule put restrictions on the inter-procedural control flow on collectives. It is enforced by a standard program analysis.

Phase 2. Within an `spmf` function, the conditions of the (intra-procedural) control flow statements that lead to any collective function are identified as *coherent*. In current MSL, the control flow statements are just `if` branches and loops. This phase identifies those conditional statements with collective operations inside, and the corresponding condition expressions are required to be coherent. The boolean value of a coherent condition expression must not vary across different processes. Phase 2 only needs intra-procedural information, and is done by a standard control flow analysis.

Phase 3. The conditions identified in Phase 2 are checked to be actually coherent, by the static-dynamic checking approach, discussed in Chapter 1. This checking procedure

reasons about inter-procedural control flow and data flow all the way back to the test harness, so it is very precise, but unsound due to the inherent limitations of the bounded model checker.

The Phase 1 and Phase 2 are similar to existing work like [1, 43], In particular, Titanium has a notion of *single-valuedness* that involves checks similar to the Phase 1 and Phase 2 of our rules in MSL. Their type system enforce the rules by local checks. The Titanium compiler can prove very fast and soundly that a program follows the bulk-synchronous model. To achieve that goal, the checks in Titanium are quite conservative, and oftentimes the programmers need to provide a great deal of annotations to satisfy the compiler. MSL makes the trade-off differently than Titanium, because our Phase 3 enforces the rules via a more precise but slower inter-procedural checking procedure. This lets MSL be more expressive, and light-weight in terms of annotation burdens, but the static-dynamic checking approach is unsound and slower than the type system approach in Titanium.

Example 3.3.1. The code snippet below illustrates the bulk-synchronous constraints:

```

smpd void g(int x, ...) {
    ...
    barrier();
    ...
}
smpd void f(int x, ...) {
    ...
    if (different()) { y = square(x); }
    else { y = factorial(x); }
    if (x > 10) g(x, ...);
    ...
}

```

In the above code, both `f` and `g` are **smpd** functions, `barrier` is a synchronization primitive, and all three are collectives. Both `square` and `factorial` are normal (non-**smpd**) functions, and they are not collectives. As before, `different()` can return different values in different processes, so the control flow induced from `f` can diverge across different processes: some processes will call `square` while others will call `factorial`. But as long as “`x > 10`” evaluates to the same boolean value across all processes, the control flow that may affect the collectives — `g`, and `barrier` would stay the same for all processes, so the execution still falls into the bulk-synchronous model. This restriction on the condition is different

from Titanium, because Titanium’s type system would require the value of “ x ” to be coherent in order for the whole condition to be coherent across processes. In this case, because x is a formal parameter to f , it must be tagged as `single` in Titanium, and all processes must have exactly the same value for x . Most likely the caller of f also needs to make the corresponding actual argument `single`. This kind of restriction and annotation burden can spread quite pervasively in a Titanium program. Using these conservative, local guarantees, the Titanium compiler can prove that the whole program is bulk-synchronous very fast. Whereas in the case of MSL, different processes can have different values of x , as long as they are all > 10 or all ≤ 10 . This is because MSL uses a more sophisticated checking procedure to actually “evaluate” and check (at synthesis time) whether the whole condition expression has coherent value across processes. This checking reasons about the control and data flow starting from the beginning of the program execution (the test harness function), so it is much slower than Titanium’s compiler, and it is unsound because of its reliance on the bounded model checker. Nevertheless this static-dynamic checking is effective in practice to enforce the required coding discipline.

Now take a close look at the control flow induced from the function f . Because f is required to be barrier-like, all processes should work in tandem when they enter f , then the bulk-synchronous constraints imposed on the control flow force them to enter g in tandem (or *none* of them calls g at all), and finally all processes should call `barrier` together. Thus bulk-synchrony is maintained by our rules in the above example.

Note that these bulk-synchronous constraints are only for the *semantics* of the program: they do not imply that at runtime we put barriers before and after each call to an `spmd` function or a communication primitive like `transfer`. The constraints just require that the result of the program should be *as if* every collective operation has the barrier-like behavior, and all processes sync up with each other upon collectives. The compiler and/or runtime system can loosen the simultaneity requirement for the purpose of better performance and parallelism. In Chapter 6 we will see that an important optimization allows the `transfer` function to behave correctly without introducing any barriers: in the generated code, the point-to-point communication will

be performed by each process independently, and faster processes do not need to wait for other slower processes. Similarly, a call to an `spmd` function never requires a global synchronization.

3.3.5 Relating SPMD and sequential programs

Section 3.3.2 and Section 3.3.4 state that the caller of an `spmd` function must also be an `spmd` function. The question is: who is the “root” caller of the `spmd` functions?

At runtime, this hardly becomes a problem, because ultimately MSL programs are translated into MPI code, which naturally maps into the SPMD execution model: there are multiple processes running in parallel from the beginning of a program’s lifetime, each process has its own memory, and processes communicate with each other only through the provided mechanisms. In this sense, there needs not to be any “root” caller, or to put it another way: the root caller is the MPI runtime itself.

The real problem surfaces at synthesis time: the synthesizer needs to establish equivalence between an SPMD program and a sequential program. The sequential program runs in serial order, and does not have a notion of *distributed*, *local* data: it always operates on *non-distributed* data. This means that the programmer needs some way to relate SPMD programs with sequential programs, and to relate distributed data with non-distributed data.

Inside any test harness function in MSL, the programmer can use `spmdfork` to initiate the SPMD execution for a code region, like this example below:

```
double[N] tester(int N, int[N] A) implements seq_f {  
  // sequential execution mode:  
  double[N] B;  
  // start of SPMD execution mode:  
  spmdfork {  
    // code executed by multiple processes  
    int[N/nprocs] x;  
    double[N/nprocs] y;  
    f(x, y);  
    ...  
  }  
  // back to sequential execution again  
  return B;  
}  
  
spmd void f(int N, int[N/nprocs] a, ref double[N/nprocs] b) { ... }
```

In the above code, the function `tester` has both sequential and parallel execution modes. Initially `tester` executes in sequential mode, because it is not tagged with the `spmd` keyword. We name the executor of the sequential code the *main process*. The semantic of `spmdfork` is similar to the *fork-join parallelism*: upon reaching an `spmdfork`, the main process spawns a set of `nprocs` sub-processes, running in parallel, each executing the code inside the body of `spmdfork` (we call that body the *parallel region*, and the code outside the `spmdfork` are in the *sequential region*), and the main process is blocked on waiting for all sub-processes to finish; after all sub-processes complete the code in the parallel region, they are terminated, and the control goes back to the main process, which executes in sequential mode again.

This multi-process parallelism seems very different from SPMD model, but at least inside the parallel region, the multiple sub-processes can simulate the SPMD processes faithfully, as long as we have this restriction: each sub-process has its own private memory; the code executed by a sub-process can only access its own private memory, and cannot access other processes' memory; any data declared inside the parallel region is *local* to the executing process's private memory. The data declared outside the parallel region are considered *global*.

In the above example, variables `N`, `A`, and `B` reside in the *global* memory, so there is only one `N`, one `A`, and one `B` per each call of `tester`. The variables `x` and `y` are both *local* to their executing process, i.e. conceptually there are `nprocs` independent versions of each variable: $x_0, x_1, \dots, x_{nprocs-1}$ and $y_0, y_1, \dots, y_{nprocs-1}$, corresponding to each process, and process i can only access x_i and y_i .

Because the fork-join parallelism introduced by `spmdfork` can simulate the real SPMD parallelism, it is allowed to call `spmd` functions (like `f` in the example) inside an `spmdfork`. Thus `spmdfork` is the *root caller* of all the `spmd` functions during synthesis time.

The `tester` function has a very nice property: all the parallelism and potential non-determinism are wrapped inside the `spmdfork`. The entry and exit parts of `tester` are *sequential*, and the input and output of `tester` are both *global* data. This is very important, because then at least at the function level, `tester`'s signature is no different from a normal sequential function, thus it can be compared (apples-to-apples) with a

sequential specification. Below is a valid syntax:

```
double[N] tester(int N, int[N] A) implements spec {  
    // the same code as in the previous example code  
    ...  
}  
  
double[N] spec(int N, int[N] A) { ... }
```

So we can establish refinement (equivalence) relationship between a sequential specification and a tester function which has `spmdfork` in it and calls some `spmd` functions. By this means, we indirectly specifies what the correct behaviors of these `spmd` functions should be.

As stated above, the code inside the parallel region can only access the local data, but this is too restrictive: In `tester`, the `spmd` function operates on local data `x` and `y`, but we need some way to relate them with the global data `A` and `B` outside the `spmdfork`, because otherwise, the `spmd` function `f` will not be relevant to the outside world, and in particular, its correct behavior will not be related to the sequential function `spec` by any means. For this reason, we allow the code inside `spmdfork` to access global data in a very restricted form, through the *distribute* and *collect* functions.

A *distribute* function (tagged by the `distr` keyword) takes as input some global data, and outputs some local data, which can later be accessed by the executing process. A *collect* function (tagged by the `coll` keyword) takes as input some local data, and also takes some global data as its pass-by-reference parameter; the *collect* function uses the local data to update the global data. The *distribute* functions must be put at the beginning of a parallel region, before any other statements, and the *collect* functions must be put at the end of a parallel region, after any other statements. As illustrated below:

```
double[N] tester(int N, int[N] A) implements spec {  
    double[N] B;  
    spmdfork {  
        int[N/nprocs] x = dist1(A);  
        double[N/nprocs] y = dist2(B);  
  
        f(x, y);  
  
        collect2(B, y);  
        collect1(A, x);  
    }  
}
```

```

    return B;
}

distr int[N/nprocs] dist1([int N], int[N] A) {
    int[N/nprocs] a;
    for (int i = 0; i < N/nprocs; ++i)
        a[i] = A[i+(N/nprocs)*mypid];
    }
    return a;
}

coll void collect1([int N], ref int[N] A, int[N/nprocs] a) {
    for (int i = 0; i < N/nprocs; ++i)
        A[i+(N/nprocs)*mypid] = a[i];
    }
}

distr double[N/nprocs] dist1([int N], double[N] A) {
    int[N/nprocs] a;
    for (int i = 0; i < N/nprocs; ++i)
        a[i] = A[i+(N/nprocs)*mypid];
    }
    return a;
}

coll void collect2([int N], ref double[N] A, double[N/nprocs] a) {
    for (int i = 0; i < N/nprocs; ++i)
        A[i+(N/nprocs)*mypid] = a[i];
    }
}

```

In the above code, both `A` and `B` are global data, which cannot be accessed directly by the code inside `spmdfork`, so the programmer calls the distribute functions `dist1` and `dist2` to produce local data `x` and `y`, and then the call to the subroutine `f(x,y)` can access `x` and `y`. At the end, the programmer calls the collect functions `collect1` and `collect2` to update the global data `A` and `B` from the information in distributed local data `x` and `y`. The distribute/collect functions themselves are very simple: together they form a *partition* of a global array into `nprocs` pieces.

Introducing distribute and collect functions raises another problem: now the multiple sub-process spawned by `spmdfork` can touch global data, not purely local data, and there might be race conditions because of non-deterministic interleavings. To eliminate such problem, MSL enforce that:

1. Any distribute / collect function is implicitly wrapped inside a pair of `barrier` statements. This ensures that any potential non-determinism is restricted inside the distribute /

collect function itself, and never polutes the other parts in the parallel region.

2. Any distribute function should only *read from* the global data, never write to. This ensures that the calls to the distribute function executed by different sub-processes commute with each other.

3. Any collect function should only *read from* the local data, never write to. But collect functions still need to write to the global data. So we further require that the calls to the collection function executed by different sub-processes commute with each other. Recall that the collect function takes an implicit parameter `mypid`, and if we make this parameter explicit, we get something like this:

```
void collect(int mypid, ref T globalData, S localData) { ... }
```

The MSL synthesizer adds assertions to enforce that

```
collect(pid1, globalData, localData1);  
collect(pid2, globalData, localData2);
```

is functionally equivalent to

```
collect(pid2, globalData, localData2);  
collect(pid1, globalData, localData1);
```

for any pair of `pid1` and `pid2`.

The above rules make sure that the introduction of *distribute/collect* functions does not introduce new non-determinisms into the parallel region: they just allow the code inside `spmdfork` to access the global data. Any possible non-determinism should come directly from the original code inside the parallel region (the call to those `spmd` functions), and we will prove in Chapter 4 that there really is *no such non-determinisms*.

The rules 2 and 3 above seem quite complex at first glance, but natural distribute / collect functions satisfy them without much effort, because usually they are just representing some partitioning of the global data, and different sub-processes touch different portions, which do not overlap with each other.

Cautious reader may have noticed that the example code inside `spmdfork` has used the global variable `N` without an explicit `distribute` call. The reason is that MSL provides a default `distribute / collect` behavior for any global variable accessed inside the parallel region:

The default *distribute* is to *broadcast* the global variable to each process;

The default *collect* is to check that the value of the local variables are unique across all process, and write this unique value back to the global variable.

Below shows the above example after this default distribute/collect behavior is inserted:

```
double[N] tester(int N, int[N] A) {
    double[N] B;
    spmdfork {
        int N' = defaultDist(N);
        int[N'/nprocs] x = dist1(A);
        double[N'/nprocs] y = dist2(B);

        f(x, y);

        collect2(B, y);
        collect1(A, x);
        defaultCollect(N, N');
    }
    return B;
}

distr int defaultDist(int N) {
    return N;
}

coll void defaultCollect(ref int N, int n) {
    if (mypid == 0) {
        N = n;
    } else {
        assert(N == n);
    }
}
```

It is important to note that *distribute/collect* functions exist solely to allow the analysis engine to relate the distributed memory implementation to the sequential implementation; they do not actually perform any communication, and will not be part of the synthesized kernel. These functions enable the reasoning in Chapter 4. The code generation phase (Chapter 6) will bypass all *distribute/collect* functions.

Chapter 4

SPMD-to-sequential transformation

In Chapter 3 we have listed the main language features of MSL. With MSL, the programmer can write sequential and SPMD programs, can assert functional equivalence among those programs, and can synthesize unknown parts in the SPMD implementation based on the equivalence relationship against a sequential specification.

The equivalence checking and synthesis features of MSL are based on the previous work of SKETCH, which is a synthesis-enabled language for purely sequential programs. As we have seen in Figure 1-4, MSL transforms an SPMD program into an equivalent sequential program during synthesis time, and then feeds this sequential program into the SKETCH synthesizer to discover the unknown parts (holes) and check for correctness. The purpose of this transformation is purely for analyzability — to reduce the synthesis problem for SPMD programs into an analyzable sequential synthesis problem. This analyzability comes with some restrictions on MSL’s programming model, but due to the important optimizations (described in Chapter 6), we still get scalability and high performance, as will be seen in Chapter 8.

We will cover the transformation process in this chapter via the following strategy: First, we present a set of formal language semantics for both parallel and sequential programs in MSL — or strictly speaking, we restrict our discussion to a simplified language L_{small} (introduced in Section 4.1) that preserves the key features of MSL from the point of view of SPMD programming. It is not a full specification of semantics for the whole MSL, but nevertheless captures the interesting aspects related to our

discussion of the transformation process. The semantics of L_{small} is given in two steps in Section 4.2: we present a reduction $\llbracket \cdot \rrbracket_{Tr}$ from L_{small} to a language of instructions L_{inst} , and define the semantics of L_{inst} in terms of the effect of each instruction on the program state.

Then we will describe a transformation that converts a parallel program to a sequential program with the following property: for any execution produced by the original parallel program, the execution of the transformed sequential program either produces the same final state, or causes an assertion failure if the original program failed to satisfy the bulk-synchronous constraints listed in Section 3.3.4. During the transformation process, the MSL compiler identifies the places where the bulk-synchronous constraints need to be checked and adds the corresponding assertions, which will be checked by the synthesizer described in Chapter 5. Note that this implies that any SPMD program in MSL is essentially deterministic, because it must produce the same result as a deterministic sequential program regardless of the different interleavings that may happen at runtime. In explaining the above transformation process, we will also limit our discussion to the simplified language L_{small} . In Section 4.3, we define a reduction $\langle \cdot \rangle_{seq}$ from L_{small} to L_{small}^{seq} , a sequential subset of L_{small} and prove that the semantics of the resulting program in L_{small}^{seq} are equivalent to those of the original program in L_{small} . This $\langle \cdot \rangle_{seq}$ is a simplified version of the actual SPMD-to-sequential transformation process in MSL, and it shows the key transformation rules.

The above strategy is illustrated by the commutativity diagram below:

$$\begin{array}{ccc}
 L_{small} & \xrightarrow{\langle \cdot \rangle_{seq}} & L_{small}^{seq} \\
 \downarrow \llbracket \cdot \rrbracket_{Tr} & & \downarrow \llbracket \cdot \rrbracket_{Tr} \\
 L_{inst} & \xlongequal{\quad} & L_{inst}
 \end{array}$$

4.1 The L_{small} language

In this section, we present the L_{small} language, which is a shared-memory, parallel language. On one hand, it is used to model the execution of distributed memory SPMD

programs. On the other hand, it has a well-distinguished subset that can model fully sequential programs.

$$\begin{aligned}
 var &:= L[p].x \mid x \mid G.x \\
 exp &:= n \mid mypid \mid nproc \mid var \mid exp_1 \text{ op } exp_2 \\
 stmt &:= var = exp \mid assert exp \mid stmt_1; stmt_2 \mid \\
 &\quad if(exp) stmt_1 \text{ else } stmt_2 \mid while(exp) \{stmt_1\} \mid \\
 &\quad reduce(exp, var) \mid \\
 &\quad transfer(scond, exp, rid, rcond, var) \\
 Prog &:= stmt \mid fork\{stmt\} \mid Prog; Prog
 \end{aligned}$$

Figure 4-1: The simplified language L_{small}

The L_{small} language, shown in Figure 4-1, is a standard imperative language with a few additional constructs to express bulk-synchronous SPMD computation. We walk through the language features one by one:

Starting from the top-level rule, $Prog$. It simply says that a L_{small} program can be a statement ($stmt$), or a statement distributed to parallel processes ($fork\{stmt\}$), or a sequential composition of any number of those two ($Prog; Prog$).

Without being wrapped inside a $fork$, a statement is executed by a *main process* sequentially, as shown in Figure 4-2(a). We say in this case the *execution model* is sequential ($EM = Seq$). A $fork\{stmt\}$ distributes work to a number of sub processes, which execute the same statement in parallel, as shown in Figure 4-2(b). We say in this situation the execution mode is parallel ($EM = Par$).

When these two execution modes are composed together in one program, the overall execution model is illustrated in Figure 4-3: The main process runs a part of the program sequentially until it faces a $fork\{stmt\}$, and then the control of execution switches to the pool of subprocesses, which start executing together in parallel, while the main process suspends and wait for the sub processes to finish. We call the event that all subprocesses have completed executing $stmt$ a *join* event, which corresponds to an actual instruction we will introduce later in Section 4.2. After the *join* point, all sub processes suspend, and the execution turns back to the main process, which

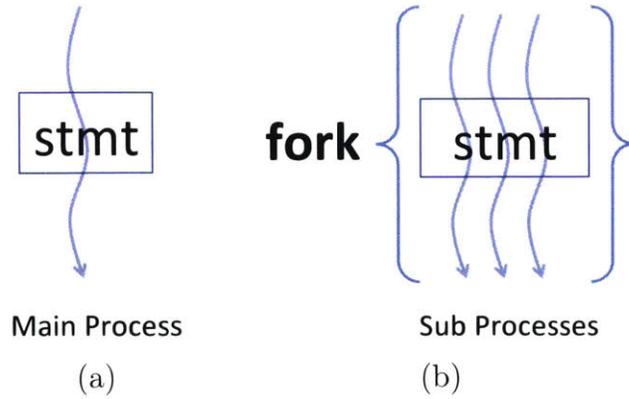


Figure 4-2: Programs in L_{small}

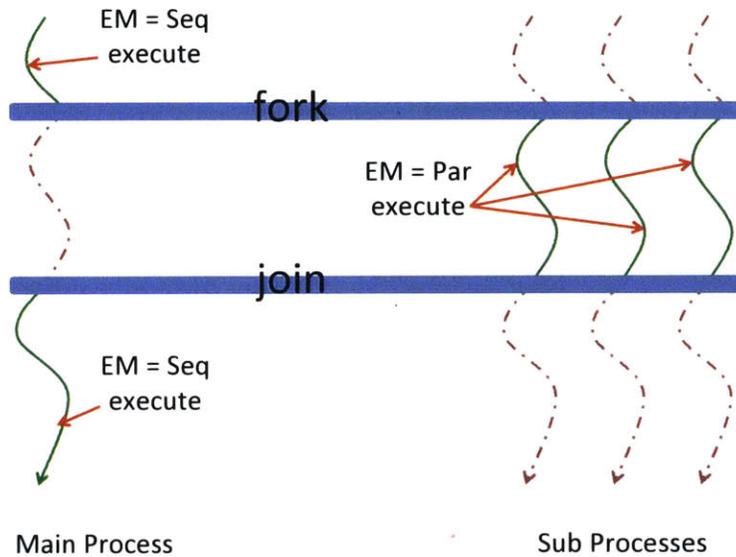


Figure 4-3: The execution model of L_{small}

continues the rest of the program. This kind of execution mode switching may happen multiple number of times in a valid L_{small} program. This execution model is an instance of the classic *fork-join parallelism* [22, 101], and the only special thing is that there is no “creation” or “annihilation” of subprocesses, as all the subprocesses are always there, and they simply suspend when $EM = Seq$. This is also like the *thread pool* model [37, 103].

The statements ($stmt$) in L_{small} are standard, including assignment to a variable, assertions, sequential compositions, *if* conditions, and *while* loops, and additionally communication / synchronization primitives: *reduce* and a *transfer*, which will be

explained in detail later.

The expressions (exp) are also standard, including integer constant literals (n), two special integer constants $mypid$ — the executing process’s ID — and $nproc$ — the total number of subprocesses — variables, and arithmetic expressions.

The variables in L_{small} are special and of particular interest to the discussion, as they are an abstraction of the underlying storage model for the data.

The standard data layout for distributed memory SPMD programs is for each process to have its own local memory, containing its own version of variables. Any process only get access to its local memory. In L_{small} , we simulate this distributed data layout by wrapping all the local memories into a *local store*, named L , which is a mapping from *process ID* (Pid) and *variable name* (Var) to values, shown in Figure 4-4(a). In addition to that, L_{small} has a *global store* G , which is a mapping from variable name to values, shown in Figure 4-4(b), and can only be accessed by the *main process*.

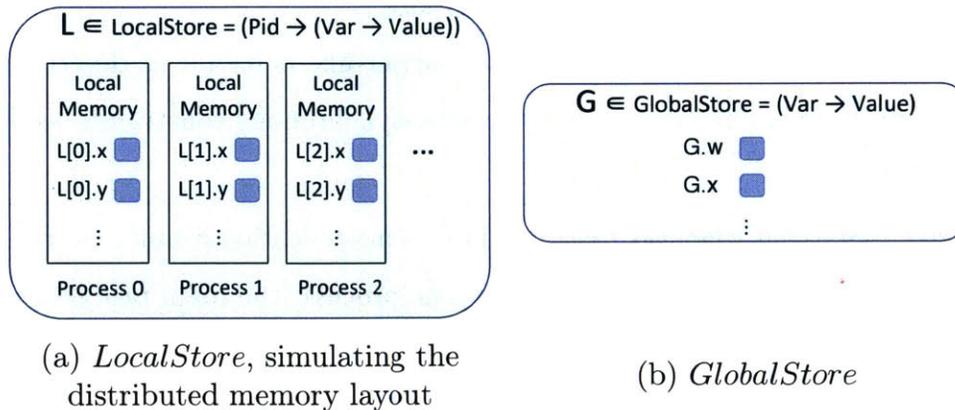


Figure 4-4: Variables in L_{small}

The “*var*” production rule of Figure 4-1 provides syntax for accessing variables in both local and global stores: $L[p].x$ is for referring to variable x on subprocess p ’s local memory, which can be used by both the main process (statements not inside a **fork**) and the subprocesses (statements wrapped inside a **fork**), but in the latter case, that “ p ” must be “ $mypid$ ”, the executing process’s ID, because as we said, each subprocess can only access its own local memory. The “*var*” rule also provides “ x ” as a shorthand for “ $L[mypid].x$ ”. $G.x$ is the syntax for referring to variable x residing in the global store, which can only be used in statements not inside a **fork**.

Inside a `fork`, any statement only access its own local variables, thus the `fork { stmt }` part of L_{small} simulates distributed memory SPMD programs faithfully. When the program does need to cooperate among different subprocesses, it must make use of the two communication primitives: `reduce` and `transfer`.

`reduce(exp,var)` evaluates an expression `exp` at every process, adds all their values, and writes the sum to the output variable `var` on every process. Right after the execution of `reduce`, the local variables $L[p].var$ will have the same value (the calculated sum) for all process IDs p .

`transfer(scond,exp,rid,rcond,var)` sends and receives data from peer processes: `scond` is the send condition, an expression that tells whether the executing process should send anything; `exp` is an expression representing the value to be sent; `rid` (an expression) tells the process id of the receiver; `rcond` is the receive condition, an expression that tells whether the executing process should receive anything; and the output local variable `var` stores the value received from some peer process, if any.

Both these communication primitives have barrier-like behavior as described in Section 3.3.1 and their usage must follow the bulk-synchronous constraints as discussed in Section 3.3.4.

It is easy to see that when `fork { stmt }` and the communication primitives are excluded, L_{small} becomes a subset language with only one process (the main process) executing sequential programs, and we call this subset language L_{small}^{seq} . Later in Section 4.3 we will see that any program in L_{small} can be transformed into a equivalent program in L_{small}^{seq} while preserving its semantic meaning.

4.2 Semantics of L_{small}

In order to define the semantics of programs in L_{small} , we first introduce a lower-level language of instructions called L_{inst} , shown in Figure 4-5. L_{inst} splits the monolithic communication primitives (`reduce` and `transfer`) into finer-grained instructions, allowing us to reason about the *interleavings* of actions in the program, as we will demonstrate in Section 4.3. Figure 4-6 shows the translation $[\cdot]_{Tr}$ from L_{small} to L_{inst} .

$fork \quad join \quad var := exp \quad jmp(exp, loc) \quad wait$
 $rin(exp) \quad red(var) \quad tin(scond, exp, rid) \quad trn(rcond, var)$

Figure 4-5: The instruction language L_{inst}

$\llbracket var = exp \rrbracket_{Tr} = var := exp$
 $\llbracket assert\ exp \rrbracket_{Tr} = jmp(\neg(exp), \perp)$
 $\llbracket stmt_1; stmt_2 \rrbracket_{Tr} = \llbracket stmt_1 \rrbracket_{Tr}; \llbracket stmt_2 \rrbracket_{Tr}$
 $\llbracket if\ (exp)\ stmt_1\ else\ stmt_2 \rrbracket_{Tr} = jmp(\neg(exp), l_1); \llbracket stmt_1 \rrbracket_{Tr}; jmp(true, l_2); [l_1]; \llbracket stmt_2 \rrbracket_{Tr}; [l_2]$
 $\llbracket while(exp)\{stmt\} \rrbracket_{Tr} = [l_1]; jmp(\neg(exp), [l_2]); \llbracket stmt \rrbracket_{Tr}; jmp(true, [l_1]); [l_2]$
 $\llbracket reduce(exp, var) \rrbracket_{Tr} = rin(exp); red(var); wait$
 $\llbracket transfer(scond, exp, rid, rcond, var) \rrbracket_{Tr} = tin(scond, exp, rid); trn(rcond, var); wait$
 $\llbracket fork\ \{stmt\} \rrbracket_{Tr} = fork; \llbracket stmt \rrbracket_{Tr}; join$

Figure 4-6: Translating L_{small} statements to L_{inst} instructions

On the left of “=” are programs in L_{small} ; on the right of “=” are programs in L_{inst} .

The rules use the notation $[l_i]$ as a way to name a particular location in the program (similar to *line numbers*, but they are labeling instructions). Also suppose that there is a reserved error location \perp , so that whenever a program counter enters \perp , the program halts with assertion failure; hence, *assert exp* translates to $jmp(\neg(exp), \perp)$.

Some of the rules in Figure 4-6 are easy to follow, like the rules to translate assignments, assertions, and sequential compositions. The translation from control flow constructs (**if** and **while**) to lower level instructions involving *jmp* is also a standard practice that helps define small-step operational semantics for the programs, as we will show later. The most tricky translation rules are the communication operations (*reduce* and *transfer*) and the **fork** construct, and we will soon see the rationale behind them once we define the operational semantics of L_{inst} .

The program in L_{inst} executes by taking actions and changing the state. Figure 4-7 shows the state that the program operates on.

The state of L_{inst} includes the local store L and the global store G , already shown in Figure 4-4. In addition, the state contains program counters to make the control flow transfers explicit. As shown in Figure 4-8, there is a program counter MC for the main process, and a group of program counters PC for subprocesses ($PC[p]$ indicates the

$G \in GlobalStore = (Var \rightarrow Value)$
 $L \in LocalStore = Pid \rightarrow (Var \rightarrow Value)$
 $MC \in MainProcPC = LocId$
 $PC \in SubProcPC = Pid \rightarrow LocId$
 $EM \in ExecMode = \{Seq, Par\}$
 $C \in Channel = (Pid \rightarrow Value)$
 $\sigma \in State = GlobalStore \times LocalStore \times ExecMode \times MainProcPC \times SubProcPC \times Channel$

Figure 4-7: The state that the L_{inst} programs operate on

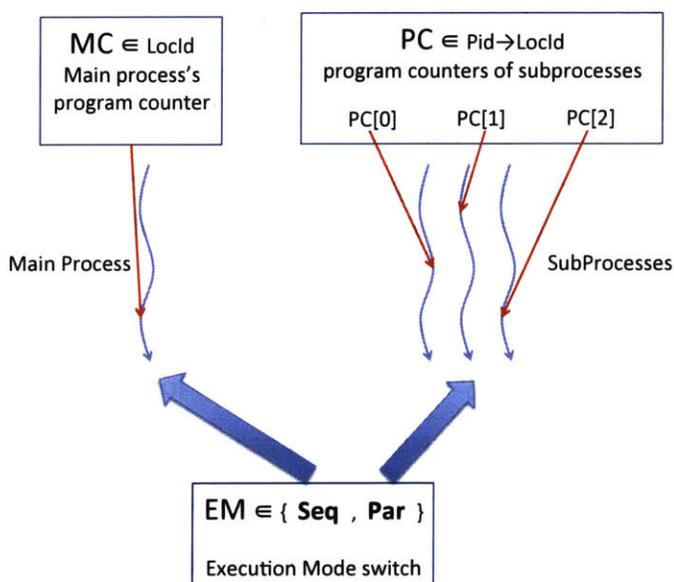


Figure 4-8: Program counters and Execution mode in L_{inst}

program counter for subprocess p). The execution of L_{inst} switches between sequential mode and parallel mode, so there is an indicator EM to tell between the two modes. Before the program starts, $EM = Seq, MC = 0, PC = \lambda u.0$.

And finally, there is a channel C , which can be seen as an array of mailboxes for each subprocess. The execution of `reduce` and `transfer` will use C as a working buffer.

Figure 4-9 shows the small step semantics rules for L_{inst} , and we go through them one by one. Given the current state $\sigma = (G, L, MC, PC, EM, C)$, the rules define how to compute the next state $\sigma' = (G', L', MC', PC', EM', C')$.

Starting with the first ASNGG rule:
$$\frac{EM = Seq, I(MC) : G.x := e}{\sigma \xrightarrow{G.x:=e} \sigma'} \quad G'.x = (e)_{G,L}, MC' = MC + 1$$
. The

$$\begin{array}{c}
\text{[ASGNG]} \quad \frac{EM = Seq, I(MC) : G.x := e \quad G'.x = (e)_{G,L}, MC' = MC + 1}{\sigma \rightarrow^{G.x:=e} \sigma'} \\
\text{[ASGNL]} \quad \frac{EM = Seq, I(MC) : L[p].x := e \quad L'[p].x = (e)_{G,L}, MC' = MC + 1}{\sigma \rightarrow^{L[p].x:=e} \sigma'} \\
\text{[JMP]} \quad \frac{EM = Seq, I(MC) : jmp(e, loc) \quad (e)_{G,L} \Rightarrow MC' = loc \quad \neg(e)_{G,L} \Rightarrow MC' = MC + 1}{\sigma \rightarrow^{jmp(e,loc)} \sigma'} \\
\text{[FORK]} \quad \frac{EM = Seq, I(MC) : fork \quad EM' = Par \quad \forall u. PC'(u) = MC + 1}{\sigma \rightarrow^{fork} \sigma'} \\
\text{[JOIN]} \quad \frac{EM = Par, pc = PC(p) \quad I(pc) : join \quad \forall u. PC(u) = pc \quad EM' = Seq, MC' = pc + 1}{\sigma \rightarrow_p^{join} \sigma'}
\end{array}$$

Below are rules for execution in parallel mode. They implicitly assume $EM = Par$ and $pc = PC(p)$. p denotes the executing subprocess, picked non-deterministically:

$$\begin{array}{c}
\text{[ASGNpar]} \quad \frac{I(pc) : x := e \quad L'[p].x = (e)_{L[p]} \quad PC'(p) = pc + 1}{\sigma \rightarrow_p^{x:=e} \sigma'} \\
\text{[JMPpar]} \quad \frac{I(pc) : jmp(e, loc) \quad (e)_{L[p]} \Rightarrow PC'(p) = loc \quad \neg(e)_{L[p]} \Rightarrow PC'(p) = pc + 1}{\sigma \rightarrow_p^{jmp(e,loc)} \sigma'} \\
\text{[TIN]} \quad \frac{I(pc) : tin(scond, e, rid) \quad \forall u. PC(u) \in \{pc, pc + 1\} \quad (scond)_{L[p]} \Rightarrow C'[(rid)_{L[p]}] = (e)_{L[p]} \quad PC'(p) = pc + 1}{\sigma \rightarrow_p^{tin(scond,e,rid)} \sigma'} \\
\text{[TRN]} \quad \frac{I(pc) : trn(rcond, x) \quad \forall u. PC(u) \in \{pc, pc + 1\} \quad (rcond)_{L[p]} \Rightarrow L'[p].x = C[p] \quad PC'(p) = pc + 1}{\sigma \rightarrow_p^{trn(rcond,x)} \sigma'} \\
\text{[WAIT]} \quad \frac{I(pc) : wait \quad \forall u. PC(u) \neq pc - 1 \quad PC'(p) = pc + 1}{\sigma \rightarrow_p^{wait} \sigma'} \\
\text{[RIN]} \quad \frac{I(pc) : rin(e) \quad \forall u. PC(u) \in \{pc, pc + 1\} \quad C'[p] = (e)_{L[p]}, PC'(p) = pc + 1}{\sigma \rightarrow_p^{rin(e)} \sigma'} \\
\text{[RED]} \quad \frac{I(pc) : red(y) \quad \forall u. PC(u) \in \{pc, pc + 1\} \quad L'[p].y = \sum_u C[u] \quad PC'(p) = pc + 1}{\sigma \rightarrow_p^{red(y)} \sigma'}
\end{array}$$

Figure 4-9: The execution semantics of L_{inst}

rule states that, when all the predicates above the line are satisfied, it is possible for the program to execute an instruction and transition from the old state σ to a new state σ' , denoted as $\sigma \rightarrow^{G.x:=e} \sigma'$ (we write the instruction on the upper-right corner of the arrow to make it clear to the reader which instruction is executed in this transition). We use the map $I : LocId \rightarrow Instr$ to determine that the instruction at a given program location is $Instr$. Specifically here, when the execution model is Seq , and the current instruction for the main process is an assignment to a global variable ($I(MC) : G.x := e$), the program is enabled to move to a new state σ' in which $MC' = MC + 1$, and the x component of the new global store is assigned the value $(e)_{G,L}$, where $(e)_{G,L}$ means the evaluation of the expression e under the global store G and local store L (both from the old state σ), i.e. any referred variable in e is replaced with its value in G or L when evaluating e . Also note that we implicitly assume that any component of the state not mentioned in the predicates remain the same from the old state to the new state. For example, in this rule we assume that $G'.y = G.y$ for all variable y other than x , and $L' = L$, $PC' = PC$, $EM' = EM$, etc.

Then we have the ASGNL rule, which is almost identical to the previous ASGNG rule, but for assignments to local stores, executed by the main process; and the JMP rule, which is for the *jmp* instruction executed by the main process — note that *jmp* changes the program counter (MC) conditionally based on the value of the condition expression, so there are two conditional predicates governing the new MC' .

The FORK rule:
$$\frac{\begin{array}{l} EM = Seq, I(MC) : fork \\ EM' = Par \\ \forall u. PC'(u) = MC + 1 \end{array}}{\sigma \rightarrow^{fork} \sigma'}$$
 deals with the *fork* instruction.

When the execution mode is Seq , and the main process faces a *fork*, the next state will be in parallel execution mode ($EM' = Par$), and the program counters for all subprocesses are set to be the next location after the *fork* ($\forall u. PC'(u) = MC + 1$). This will enable all subprocesses to make moves.

The rest of the rules are all under the parallel execution mode, in which multiple rules might be satisfied simultaneously, and multiple subprocesses may fit the predicates of a rule at the same time, but we do assume that at any moment, only one subprocess

that fits some rule, chosen non-deterministically, will make the “next” move and execute some instruction. This clearly manifests that our language semantics allow for non-deterministic, parallel execution. Despite this non-determinism, however, the overall execution of the program will be deterministic, as we will show later in Section 4.3. Let p denote the ID of the (non-deterministically) chosen subprocess. The following rules all assume $EM = Par, pc = PC(p)$.

The first rule under the parallel mode is the JOIN rule:

$$\frac{\begin{array}{c} EM = Par, pc = PC(p) \\ I(pc) : join \\ \forall u. PC(u) = pc \\ EM' = Seq, MC' = pc + 1 \end{array}}{\sigma \xrightarrow[p]{join} \sigma'}$$

Opposite to *fork*, *join* switches the execution mode from parallel to sequential. When $EM = Par$, and some subprocess p is facing a *join* instruction ($I(PC(p)) : join$), it can be seen from Figure 4-9 that the only rule that process p may fit is the JOIN rule. In order for process p to move, the JOIN rule requires all the other subprocesses to come to the same program location as process p ($\forall u. PC(u) = pc$). In this situation, we say that subprocess p is “waiting” for other subprocesses to “catch up”. Strictly speaking, though, subprocess p is not actively doing anything — it is just that when others do not catch up, no execution rules will be satisfied by subprocess p — in some sense, subprocess p “waits for” other subprocesses passively upon the *join* instruction. Once all the subprocesses come to the same *join*, the execution goes back to the main process ($EM' = Seq$), which starts executing the next instruction after the *join* ($MC' = pc + 1$), and the whole system transition into a new state ($\sigma \xrightarrow[p]{join} \sigma'$). In addition to the executed instruction, we write the executing process ID p on the lower-right corner of the transition arrow. Because of its nature, whenever the JOIN rule is enabled, it must be enabled by all the subprocesses simultaneously, and the choice of the executing process p is arbitrary and non-deterministic, but all choices of p are equivalent.

The FORK and JOIN rules together maintain the execution model as depicted in Figure ??(c): the control of execution switches between the main process (when $EM = Seq$) and the pool of subprocesses (when $EM = Par$), but never mixed.

The ASGN_{par} rule:

$$\frac{\begin{array}{l} I(pc) : x := e \\ L'[p].x = (e)_{L[p]} \\ PC'(p) = pc + 1 \end{array}}{\sigma \xrightarrow[p]{x:=e} \sigma'}$$

is very similar to the ASGNL rule, but

it describes the case where a subprocess assigns an expression e to a local variable x (recall that x is just a shorthand for $L[p].x$). As we stated before, a subprocess can only access its own local memory, so the evaluation of e is taken under the environment $L[p]$, written as $(e)_{L[p]}$. Note that there is no counterpart for the ASGNG rule in parallel mode, because the subprocesses cannot modify the global memory G .

Next we examine the communication rules: TIN, TRN, and WAIT. They together maintain the barrier-like behavior of `transfer`. Recall in Figure 4-6 that a monolithic `transfer` statement in L_{small} is translated into a trio like this:

$$\llbracket \text{transfer}(scond, exp, rid, rcond, var) \rrbracket_{Tr} = tin(scond, exp, rid); trn(rcond, var); wait$$

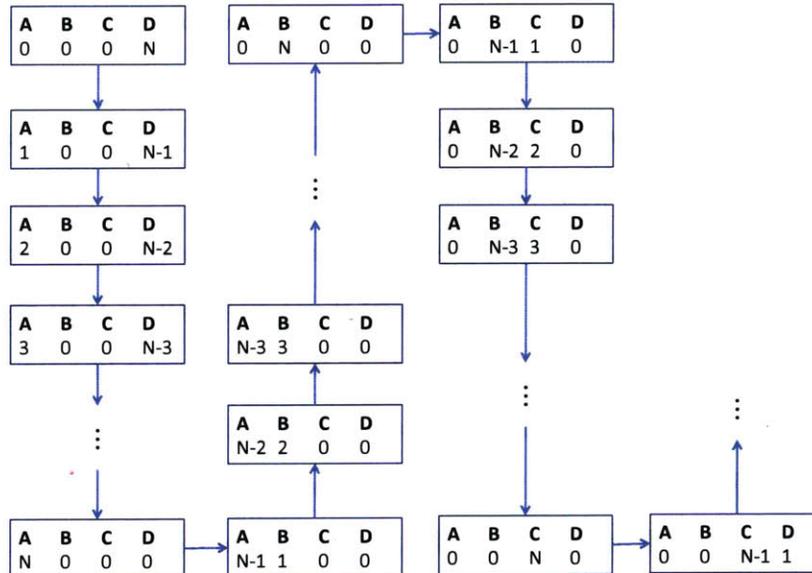
Theorem 4.2.1. When *tin*, *trn*, and *wait* are used together in a group like the above, and there is no *jmp* instruction in the program that jumps into the middle of the group (jumping onto the *trn* or the *wait* instruction), the *tin* actions executed by all subprocesses are always grouped together, and the same is true for *trn*. In detail, the following properties are maintained:

1. When some subprocess encounters the *tin* instruction, it cannot proceed until all subprocesses are facing the same *tin* instruction. Then they can execute the *tin* instruction in tandem.
2. When some subprocess encounters the *trn* instruction, it cannot proceed until all subprocesses have all executed the previous *tin* instruction and come to the same *trn* instruction. Then they can execute the *trn* instruction in tandem.
3. When some subprocess encounters the *wait* instruction, it cannot proceed until all subprocesses have all executed the previous *trn* instruction and come to the same *wait* instruction.

We sketch the proof of Theorem 4.2.1 by demonstrating how the program executes according to the three rules:

| | | |
|---|---|--|
| $ \begin{array}{l} I(pc) : tin(scond, e, rid) \\ \forall u. PC(u) \in \{pc, pc + 1\} \\ (scond)_{L[p]} \Rightarrow C'[(rid)_{L[p]}] = (e)_{L[p]} \\ PC'(p) = pc + 1 \end{array} $ | $ \begin{array}{l} I(pc) : trn(rcond, x) \\ \forall u. PC(u) \in \{pc, pc + 1\} \\ (rcond)_{L[p]} \Rightarrow L'[p].x = C[p] \\ PC'(p) = pc + 1 \end{array} $ | $ \begin{array}{l} I(pc) : wait \\ \forall u. PC(u) \neq pc - 1 \\ PC'(p) = pc + 1 \end{array} $ |
| $\sigma \xrightarrow{tin(scond, e, rid)}_p \sigma'$ | $\sigma \xrightarrow{trn(rcond, x)}_p \sigma'$ | $\sigma \xrightarrow{wait}_p \sigma'$ |

Suppose there are N subprocesses in total, and let A denote the number of subprocesses facing a *tin* instruction, B denote the number of subprocesses facing the corresponding *trn* instruction, C denote the number of subprocesses facing the corresponding *wait* instruction, and D denote the number of all the other subprocesses ($A + B + C = N - D$). From the rules above, we can infer that the state transition of the subprocesses must obey the graph shown in Figure 4-10, as will be explained below.



A, B, C, D — number of subprocesses facing the *tin*, *trn*, *wait*, and other instructions, respectively. N — total number of subprocesses.

Figure 4-10: State transition graph of subprocesses executing a group of *tin*, *trn*, and *wait* instructions.

At first, none of the subprocesses has encountered the *transfer*, so $D = N, A = B = C = 0$. Then if the trio is ever executed, there must be some subprocess (let's call it subprocess p) that first come to the *tin*, so the whole system transition to a $(1, 0, 0, N - 1)$ state. The TIN rule requires that the subprocess p cannot make a move until all subprocesses have come to either *tin* or the following *trn* ($\forall u. PC(u) \in \{pc, pc + 1\}$), but because of the “no jumping into the middle of the trio” property, other subprocesses

cannot come to the *trn* before they come to the *tin*. That is why the $(1, 0, 0, N - 1)$ state must transition to the $(2, 0, 0, N - 2)$ state, and now there are *two* subprocesses hanging around the *tin* instruction and waiting for other subprocesses to come to the same *tin*. Likewise, the state will transition to $(3, 0, 0, N - 3) \dots (N - 1, 0, 0, 1)$ and finally $(N, 0, 0, 0)$, a state where *all* subprocesses come to the same *tin* instruction. We can clearly see that this is a barrier-like synchronization.

After that, some subprocess (call it process p') can move according to the TIN rule: it evaluates the send condition, the ID of the recipient process, and the value to be transferred, under its own local environment, and sends the value to the recipient process's mailbox if the send condition is true $((scond)_{L[p]} \Rightarrow C'[(rid)_{L[p]}] = (e)_{L[p]})$. Regardless of the send condition, it advances the program counter of process p' , and the whole system transitions into a $(N - 1, 1, 0, 0)$ state. Now process p' is governed by the TRN rule, which requires it to wait for all processes to come to either the same *trn* instruction, or the following *wait* instruction. Following the same argument as before, we can see that the system must transition through a series of $(N - 2, 2, 0, 0), (N - 3, 3, 0, 0), \dots$ states until it reaches a $(0, N, 0, 0)$ state, where *all* subprocesses come to the same *trn* instruction, again a barrier-like synchronization.

Now some subprocess p'' might execute according to the TRN rule: it evaluates the receive condition, and if that is true, it copies the value in its mailbox to its local variable $((rcond)_{L[p]} \Rightarrow L'[p].x = C[p])$. Regardless of the receive condition, process p'' will face the *wait* instruction after its program counter is advanced, and the WAIT rule requires it to hang there and wait for other subprocesses to catch up.

The RIN, RED, and WAIT rules play together to enforce a similar behavior for the group of *rin*, *red*, and *wait* instructions, but to implement a correct *reduce* statement.

4.3 Reducing SPMD programs to sequential ones

Having defined the semantics of L_{small} , we now present a reduction $\langle \rangle_{seq}$ that converts an program in L_{small} into a functionally equivalent sequential program in L_{small}^{seq} (a subset of L_{small} without *fork*). For any program s in L_{small} , when we write $\langle s \rangle_{seq} = s'$,

the reduced program $s' \in L_{small}^{seq}$ is guaranteed to have this property: s' triggers an assertion failure if the original program s failed to satisfy the rules of usage for communication primitives (violating the bulk-synchronous constraints); otherwise, for any possible execution trace of s , the final state will be equivalent to running s' . Note that this implies that the behavior of a correct L_{small} program is deterministic, despite the fact that multiple subprocesses can take actions non-deterministically and many valid interleavings exist. To prove the correctness of our reduction, we will use Lipton's theory of left-mover and right-mover actions [33, 68]. An action b is a *left-mover* if for any execution where some action a is immediately followed by an action b performed by a different process, the actions a and b commute — *right-mover* actions are similarly defined.

The rules are listed in Figure 4-11. Each rule deals with certain kind of language constructs. We go through them one by one, and demonstrate their soundness.

The PRI rule $\boxed{\frac{s \text{ is private}}{\langle fork\{s\} \rangle_{seq} = \forall pid: s}}$ deals with *private* statements — any statement that does not invoke `reduce` or `transfer` and therefore involves only control flow and accesses to constants and local variables. As usual, above the line are the conditions for the rule to be applicable, and below the line is the actual transformation rule. $\forall pid : s$ is a shorthand for a loop to simulate a *fork*. It expands to

$$\begin{aligned} G.pid &= 0; \\ \text{while } (G.pid < nproc) \{ \\ &\quad s[mypid \rightarrow G.pid]; \\ &\quad G.pid = G.pid + 1; \\ \} \end{aligned}$$

Here pid is a fresh variable name, and $s[mypid \rightarrow G.pid]$ means replacing any occurrence of $mypid$ in s with $G.pid$. Note that as s is wrapped inside a *fork* in the original program, it can only access local variables of the form $L[mypid].x$, and in the transformed program those local accesses are converted to the form of $L[G.pid].x$. The PRI rule simulates a particular, fixed scheduling of the parallel program. It is

| | |
|--|-----|
| $\frac{s \text{ is private}}{\langle \text{fork}\{s\} \rangle_{seq} = \forall pid: s}$ | PRI |
| $\frac{\begin{array}{l} s \text{ or } t \text{ is collective} \quad v \text{ is a fresh variable name} \\ s' = \langle \text{fork}\{s\} \rangle_{seq} \quad t' = \langle \text{fork}\{t\} \rangle_{seq} \end{array}}{\langle \text{fork}\{\text{if}(e) s \text{ else } t\} \rangle_{seq} = \begin{array}{l} G.v = (e)_{L[0]}; \\ \forall pid: \text{assert}((e)_{L[mypid]} == G.v); \\ \text{if}(G.v) s' \text{ else } t' \end{array}}$ | IF |
| $\frac{v \text{ is a fresh variable name}}{\langle \text{fork}\{\text{reduce}(e, y)\} \rangle_{seq} = \begin{array}{l} G.v = \Sigma_{mypid}(e)_{L[mypid]}; \\ \forall pid: L[mypid].y = G.v \end{array}}$ | RED |
| $\frac{v \text{ and } f \text{ are fresh variable names}}{\langle \text{fork}\{\text{transfer}(scond, e, rid, rcond, y)\} \rangle_{seq} = \begin{array}{l} \forall pid: L[mypid].f = 0; \\ \forall pid: \text{if}((scond)_{L[mypid]}) \{ \\ \quad G.v = (rid)_{L[mypid]}; \\ \quad \text{assert}(!L[G.v].f); \\ \quad C[G.v] = (e)_{L[mypid]}; \\ \quad L[G.v].f = 1 \} \\ \forall pid: \text{if}((rcond)_{L[mypid]}) \{ \text{assert}(L[mypid].f); L[mypid].y = C[mypid] \} \end{array}}$ | TX |
| $\frac{s'_1 = \langle \text{fork}\{s_1\} \rangle_{seq} \quad s'_2 = \langle \text{fork}\{s_2\} \rangle_{seq}}{\langle \text{fork}\{s_1; s_2\} \rangle_{seq} = s'_1; s'_2}$ | SEQ |
| $\frac{s \text{ is collective, } v \text{ is a fresh variable name, } s' = \langle \text{fork}\{s\} \rangle_{seq}}{\langle \text{fork}\{\text{while}(e) s\} \rangle_{seq} = \begin{array}{l} G.v = (e)_{L[0]}; \\ \forall pid: \text{assert}((e)_{L[mypid]} == G.v); \\ \text{while}(G.v) \{ \\ \quad s'; \\ \quad G.v = (e)_{L[0]}; \\ \quad \forall pid: \text{assert}((e)_{L[mypid]} == G.v) \\ \} \end{array}}$ | WHL |

Figure 4-11: Reduction rules from L_{small} to L_{small}^{seq}

$\forall pid: s$ is a shorthand for a *sequential* loop that iterates over all process ids (starting from 0 to $nprocs - 1$), executing s in each iteration.

easy to see that this simulation is faithful using the mover's argument: any valid execution trace of $fork\{s\}$ must take the form $fork, a_1, a_2, \dots, a_n, join$, and each a_i must come from a *private* statement, *i.e.*, a_i only reads and/or writes the local state of a particular subprocess (its local store and program counter), so any a_i is a left-mover to another a_j when they come from different subprocesses. Then it is trivial to see that we can always re-arrange the actions such that all actions coming from subprocess 1 occurs first, all actions coming from subprocess 2 occurs after them, then occur all actions coming from subprocess 3, so on and so forth. This is exactly what $\forall pid : s$ will produce.

| | | |
|-------------|---|----|
| | s or t is collective v is a fresh variable name $s' = \langle fork\{s\} \rangle_{seq}$ $t' = \langle fork\{t\} \rangle_{seq}$ | |
| | $\langle fork\{if(e) s \text{ else } t\} \rangle_{seq} = G.v = (e)_{L[0]};$ $\forall pid: assert((e)_{L[mypid]} == G.v);$ $if(G.v) s' \text{ else } t'$ | |
| The IF rule | | is |

more complex. First, its conditions require that either s or t is *collective*, *i.e.* involving communication operations inside them, because otherwise the whole *if* statement will be private and can be captured by the PRI rule. Then it apply the $\langle \cdot \rangle_{seq}$ transformation recursively on $fork\{s\}$ and $fork\{t\}$, two smaller programs than the original program, and get their sequentializations s' and t' . The overall sequentialized program for $fork\{if(e) s \text{ else } t\}$ is obtained by combining s' and t' in a conditional construct. Because s or t is collective, the bulk-synchronous constraints require the path condition leading to at least one of them be coherent across processes, and in particular, the condition expression e must evaluate to coherent value. That is why the transformed program first assigns $(e)_{L[0]}$ to a temporary variable $G.v$, and then uses a loop to assert that the value of e , evaluated on all subprocesses, is the same to $G.v$. After that, as evaluations of the condition expression agree across all subprocesses, the program conditionally executes either s' or t' depending on the condition value stored in $G.v$.

To argue the soundness of the IF rule, we first observe that the first and decisive action produced by the whole *if* construct is the conditional jump $jmp(\neg(e)), loc$ (see Figure 4-6 for the complete translation of *if*). The evaluation of $\neg(e)$ is purely

private, *i.e.* only accessing the executing process's local state, thus it is a left-mover with respect to any other actions coming from a different process. Hence for any execution trace of $fork\{if\ (e)\ s\ else\ t\}$, we can always move the actions of this conditional jump from all subprocesses to the front of the trace, and because of the bulk-synchronous constraints, all these jumps must make the same decision. After this decision has been made, the rest of the trace must be equivalent to either $fork\{s\}$ or $fork\{t\}$, depending on the coherent condition value, which are in turn equivalent to what s' or t' (respectively) would produce. As long as the recursive transformations of $fork\{s\}$ and $fork\{t\}$ are semantics preserving (an inductive hypothesis), so is the transformation done by the IF rule.

The TX rule

| |
|---|
| v and f are fresh variable names |
| $\langle fork\{ transfer(scond, e, rid, rcond, y) \} \rangle_{seq} =$ $\forall pid: L[mypid].f = 0;$ $\forall pid: if((scond)_{L[mypid]}) \{ G.v = (rid)_{L[mypid]};$ $assert(!L[G.v].f);$ $C[G.v] = (e)_{L[mypid]};$ $L[G.v].f = 1 \}$ $\forall pid: if((rcond)_{L[mypid]}) \{ assert(L[mypid].f); L[mypid].y = C[mypid] \}$ |

simulates the *transfer* operation directly from its definition. The transformed program uses a fresh temporary local variable $L[p].f$ to store whether process p 's channel has already been filled by some sender. Using that, it can check the correctness of usage of *transfer*: no two senders sending to the same receiver; if some process's *rcond* is true, then it must be sent something. Although the traces of actions are vastly different between the original and the transformed programs, it can be easily checked that they perform the same overall state transition. The same argument extends to the RED rule, which simulates *reduce*.

The SEQ rule $\frac{s'_1 = \langle fork\{s_1\} \rangle_{seq} \quad s'_2 = \langle fork\{s_2\} \rangle_{seq}}{\langle fork\{s_1; s_2\} \rangle_{seq} = s'_1; s'_2}$ also apply the $\langle \cdot \rangle_{seq}$ transformation recursively on $fork\{s_1\}$ and $fork\{s_2\}$, and combine them to get the sequentialization of $fork\{s_1; s_2\}$. To demonstrate the soundness of this rule, we only

need to argue that $prog1 = fork\{s_1; s_2\}$ is semantically equivalent to $prog2 = fork\{s_1\}; fork\{s_2\}$. $prog2$ essentially reduces possible interleavings from $prog1$: in some valid scheduling of $prog1$, an action originating from s_1 might come after an action originating from s_2 , executed by different subprocesses, but for $prog2$, all actions originating from s_1 must come before all actions originating from s_2 .

Consider a trace t produced by $fork\{s_1; s_2\}$; we now prove that there is an equivalent trace t' in which any action originating from s_1 occurs before any action originating from s_2 (we call this a “good” trace). The main idea is that if t is not already a good trace, there will be two consecutive actions a_2, a_1 in t where a_1 comes from s_1 and a_2 comes from s_2 . If we can show that we are always able to swap a_2, a_1 to a_1, a_2 , then we will have a strategy to convert t into a good trace t' .

Lemma 4.3.1. If there are two consecutive actions a_2, a_1 in t where a_1 comes from s_1 and a_2 comes from s_2 , we can always swap them to be a_1, a_2 and get an equivalent trace.

Proof. First, a_1 and a_2 must be executed by two different processes, because the execution of a single process is sequential. Without loss of generality suppose that a_1 and a_2 are executed by process 1 and process 2 respectively. The proof is by cases; there are a total of 7 cases per action, corresponding to each of the 7 instructions in Figure 4-9 that can execute within a subprocess. Of those 49 cases, however, all the cases involving local writes and jumps are trivial because those instructions only access and affect local state and thus will always be left-movers and right-movers with respect to instructions executed by different processes.

On the other hand, it can never be the case that both a_2 and a_1 are one of $\{tin, trn, rin, red\}$, because that would require process 2 to be executing a `reduce` or a `transfer` from statement s_2 while process 1 is executing a `reduce` or a `transfer` from statement s_1 , which cannot happen due to the barrier-like behavior of both of those statements. The only combinations left are those involving `wait` but it is easy to see that those will not cause problems because `wait` has no side effects, thus is left-mover and right-mover with respect to all other instructions. □

The WHL rule

$$\begin{array}{l}
\hline
s \text{ is collective, } v \text{ is a fresh variable name, } s' = \langle \text{fork}\{s\} \rangle_{seq} \\
\hline
\langle \text{fork}\{ \text{while}(e) s \} \rangle_{seq} = G.v = (e)_{L[0]}; \\
\qquad \forall pid: \text{assert}((e)_{L[mypid]} == G.v); \\
\qquad \text{while}(G.v) \{ \\
\qquad \qquad s'; \\
\qquad \qquad G.v = (e)_{L[0]}; \\
\qquad \qquad \forall pid: \text{assert}((e)_{L[mypid]} == G.v) \\
\qquad \qquad \}
\end{array}$$

is similar to IF, but more complex. Given that the loop body s is collective, the transformed program first checks that the loop condition expression evaluates to coherent value across all processes, and then uses a *while* loop to execute s' multiple times, where s' is the reduced program obtained by recursively applying the $\langle \cdot \rangle_{seq}$ transformation on $\text{fork}\{s\}$. Note that after each loop iteration, the value of the condition expression might change. As the bulk-synchronous constraints require that the condition is coherent for each iteration. the condition agreement check is also done in the loop body, after each s' .

This transformation will schedule all processes to perform the first iteration before any process can start the second iteration. The soundness of this rule can be proved by induction on the number of iterations based on the soundness of the IF and SEQ rules.

Theorem 4.3.2. For any non-negative integer T , if the transformed sequential program in the WHL rule of Figure 4-11 executes the *while* loop for T times in total, and no assertion failure is triggered, this sequential program is semantically equivalent to the original parallel program in the WHL rule.

Proof. Base case: $T = 0$. For the sequential program, the loop condition $G.v$ is false, so $(e)_{L[p]}$ is false for all process ID p , because no assertions are violated. Then for the parallel program: similar to the argument we have done for the IF rule, for any execution trace, we can always move the actions that evaluate the loop condition e to the beginning of the trace and preserve the semantics, which means all the evaluations of $e_{L[p]}$ must be just like what they are evaluated in the sequential program, *i.e.* the

loop condition e evaluates to false on all subprocesses. Thus for all the subprocesses, the loop body is never executed in the parallel program, just like the case in the sequential program, proving the base case of Theorem 4.3.2.

Inductive case: suppose Theorem 4.3.2 is correct for $T = k$, we prove that it is also correct for $T = k + 1$.

Observe that $while (e) \{ s \}$ is equivalent to $if (e) \{ s; while (e) \{ s; \} \}$ for any e and s . This is obvious for sequential execution, and it can be easily proved for parallel execution, too. Using this fact, We expand both sides of the transformation in the WHL rule, and our goal is now to prove that

$$fork\{ if (e) \{ s; while (e) \{ s; \} \} \} \quad (4.3.1)$$

is semantically equivalent to

$$\begin{aligned}
&G.v = (e)_{L[0]}; \\
&\forall pid: assert((e)_{L[mypid]} == G.v); \\
&if(G.v) \{ \\
&\quad s'; \\
&\quad G.v = (e)_{L[0]}; \\
&\quad \forall pid: assert((e)_{L[mypid]} == G.v); \\
&\quad while(G.v) \{ \quad //(*) \\
&\quad\quad s'; \\
&\quad\quad G.v = (e)_{L[0]}; \\
&\quad\quad \forall pid: assert((e)_{L[mypid]} == G.v) \\
&\quad\quad \} \\
&\quad \} \\
&\}
\end{aligned}$$

(4.3.2)

when no assertion failure is triggered. But at this time, the *while* loop annotated with $(*)$ is only executed for $T = k$ iterations in the sequential program.

From the soundness of the IF rule, we know that to prove the equivalence of 4.3.1 and 4.3.2, we only need to prove the equivalence of $fork\{ s; while (e) \{ s; \} \}$ and

```

s';
G.v = (e)L[0];
∀pid: assert((e)L[mypid] == G.v);
while(G.v) {      //(*)
  s';
  G.v = (e)L[0];
  ∀pid: assert((e)L[mypid] == G.v)
}
}

```

Because of the **SEQ** rule, that boils down to proving: $fork\{ s; \}$ is semantically equivalent to s' (the condition in the **WHL** rule) and $fork\{ while\ (e)\ \{ s; \}\}$ is semanti-

cally equivalent to

```

G.v = (e)L[0];
∀pid: assert((e)L[mypid] == G.v);
while(G.v) {      //(*)
  s';
  G.v = (e)L[0];
  ∀pid: assert((e)L[mypid] == G.v)
}

```

(our inductive hypothesis).

Thus the whole chain of proof for Theorem 4.3.2 is finished. As the theorem is true for any T , it implies the soundness of the **WHL** rule. \square

This completes the proof for the transformations in Figure 4-11. The actual transformation done by MSL is only slightly more complex than what we have described so far because it must deal with a number of additional language features such as heap-allocated objects, arrays and functions, but the high-level ideas for the SPMD-to-sequential transformation are the same. By applying this reduction, we are able to transform the synthesis problem for SPMD programs into a synthesis problem over sequential implementations, allowing us to leverage existing synthesis infrastructure.

Chapter 5

Synthesis and equivalence checking

After the transformations described in Chapter 4 are applied, what is left is a synthesis problem over sequential programs. This is a problem already solved by SKETCH [84, 87], a constraint-based software synthesis language. so we provide only a brief discussion here as background.

At a high level, the synthesis problem is solved in two phases. In the first phase, the requirements that the program is equivalent to its reference implementation, together with the requirement of avoiding all assertion failures—including assertions added implicitly to check for array-bounds violations and assertions added during the transformation—get encoded into a predicate $Q(in, c)$ that is true iff the execution of the program on input in will satisfy all requirements when all unknowns in the implementation are completed as described by a control parameter c . In the second phase, the synthesizer searches for a control parameter c^s such that $\forall in. Q(in, c^s)$. In practice, the equation is usually checked only on a bounded set of inputs in ; the set is bounded by restricting the number of bits for values in . We describe the algorithm at a high level in the rest of this chapter.

5.1 Constraint generation

MSL (based on SKETCH) generates the predicate $Q(in, c)$ by performing symbolic execution over the sequentialized program, and transform it into a large DAG (Directed

Acyclic Graph) where each node is a symbolic representation of an intermediate value in program execution. The *source* nodes of the DAG (without incoming edges) are input values, including program inputs (corresponding to *in*) and control parameters (*c*, the unknown holes in the sketch); the *sink* nodes of the DAG (without outgoing edges) are all the asserted conditions; all other nodes are intermediate states that pass dependencies from sources to sinks. Straight line code maps to its representation directly, but loop iterations and function calls do not.

Like other automated bug-finding tools based on bounded model-checking [20], we handle loops and function calls by unrolling and inlining respectively. Our system will unroll a loop of the form `while(c) stmt` into a series of conditionals and an assertion `{if (c) stmt1; if (c) stmt2; ...; if (c) stmtLB; assert(! c)}`, where `stmtk` represents the *k*-th unrolled instance. The loop is unrolled up to *LB* levels, a configurable parameter to the synthesizer; asserting the negation of the loop condition at the end guarantees that insufficient unrolling is reported as an error and a larger *LB* must be tried. This restriction makes the transformed DAG finite and only works for finite input space programs. Similarly, function calls are all inlined, with a configurable bound on recursive call depth; when a generator function is inlined, the unknown holes inside it are cloned to different copies under different calling contexts.

The equivalence of the program and the reference implementation is checked by adding extra assertions that their outputs are equal. Then the DAG is turned into a boolean circuit by representing each node with a number of bits, and each edge with a number of constraints that describes the corresponding dependency between the incoming and outgoing nodes (bits). The final predicate $Q(in, c)$ is obtained by building a conjunction of all assertion nodes in the boolean circuit.

Like previous synthesis work [84], we represent floating point values as elements in a small finite field— this allows us to separate issues of floating point accuracy from programming errors while keeping the analysis tractable by reducing the number of bits required to reason about arithmetic operations. A consequence of this is that the equivalence check assumes algebraic properties like commutativity and associativity. This allows the programmer to leverage the synthesis and automated bug-finding

features even with implementations that will produce different outputs on IEEE 754 floats.

5.2 Solving for control parameters

Our system searches for control parameters c^s such that $\forall in.Q(in, c^s)$. This existential-universal predicate equation belongs to the 2QBF (Quantified Boolean Formula with two quantifiers) problem, which is usually hard to tackle. SKETCH employs a counterexample guided inductive synthesis (CEGIS) approach [84] to solve this.

The key is to avoid the universal quantifier when searching for c^s by instead solving a simpler problem: given a small set of representative inputs E_i , the system finds a c_i such that $\bigwedge_{in \in E_i} Q(in, c_i)$.

The CEGIS algorithm starts with a set E_0 containing a single random input. A constraint solver solves the problem above for a c_0 which is then passed to a checking procedure to check if the resulting program is indeed correct. If it is not, then a counterexample input in_i is found, and the process is repeated with a new set $E_{i+1} = E_i \cup in_i$. In this manner, the algorithm discovers the solution to the synthesis problem. By gradually building the representative input sets E_i 's, the CEGIS algorithm avoids the costly \forall quantifier when searching for c^s .

5.3 Correctness checking procedure

The synthesizer uses a correctness checking procedure similar to SAT-based bounded model checking [63]: for any candidate control parameter c_i , $Q(in, c_i)$ is a boolean predicate with only in as its input vector. To check whether the resulting program is indeed correct, a constraint solver tries to find a counterexample in_i such that $\neg Q(in_i, c_i)$. If no in_i can be found, the program is correct for the input space.

For our analysis to be tractable, we only use finite bits to represent in , thus provide only *bounded* correctness guarantee, which is useful to prevent many bugs like out-of-bound array access, wrong loop iteration bounds, and wrong indexing expressions.

This complements normal random testing mechanisms because the constraint solver exhaustively checks the input space and covers all corner cases.

5.4 Completing the sketch

After the synthesizer discovers the unknown parts of the program, the MSL compiler uses that information to complete the original SPMD sketch. This basically involves two steps in our system: *hole concretization* and *partial evaluation*, which are briefly explained in this section. Further details can be found in [84, 85].

As we have presented in Section 3.2, the only essential unknown values that need to be discovered by the synthesizer are the integer holes (written as ?? in the source code), which are the *control parameters* of the boolean predicate in the above discussion. Once the synthesizer finds out the values of the control parameters, it fills the integer holes based on them: each integer hole in the program is replaced by its corresponding integer constant. This simple step is called *hole concretization*. After this, the MSL program becomes complete.

Then some parts of the program that depend on the integer holes may now become fully determined and can be executed (*evaluated*) at compile time. MSL has a cleanup process to simplify or eliminate those parts, based on a standard compiler technique called *partial evaluation* [5, 21, 48]. It can be seen as a combination of very aggressive compiler optimizations such as constant propagation, transitive assignment elimination, and dead code elimination.

For example, in Section 3.2 we have stated that the choice expression is just a convenient syntactic sugar on top of the integer holes, thus

```
int x = { | a | b | 5 | i-1 |};
```

is mechanically transformed to:

```
int x;  
switch(??) {  
  case 0: x = a; break;  
  case 1: x = b; break;  
  case 2: x = 5; break;  
  default: x = i-1;  
}
```

Suppose the synthesizer concretizes the hole to 1, then the whole `switch` statement becomes useless, and the partial evaluation simplifies it to “`int x = b;`”. Furthermore, if `x` is later used in the program, before being modified, that use of `x` will be replaced by a direct use of `b`.

The purpose of the partial evaluation process is two-fold: it makes the program much cleaner and more readable, because many of the complex program constructs, both user-written and generated by the compiler, are eliminated during the partial evaluation; this simplified MSL program also results in a C++ program that is easier for the C++ compiler to optimize. Note that MSL can do more aggressive optimizations that C++ compilers cannot do because the programming model of MSL is more restrictive.

Chapter 6

Code generation

Now that the MSL compiler gets a completed SPMD program, the last step is to generate an efficient C++/MPI implementation from the high-level MSL program. As discussed in Section 5.4, the compiler relies on partial evaluation to clean up and optimize the completed program, and it also performs a set of standard optimizations such as unboxing of temporary structures and copy propagation.

In this chapter, we mainly discuss the important optimizations being used when translating MSL code to C++ code, with focus on implementing the SPMD model on top of the MPI library, which seems easy at first glance, as all the relevant concepts in MSL have direct mappings in MPI. But actually it is non-trivial, because MSL builds upon this idea of bulk-synchronous, barrier-like synchronization model, even for point-to-point communications, and if we translate the `transfer` operation naïvely with a barrier to guarantee synchrony, the performance and scalability will be limited. Thus we introduce to MSL an optimization that removes the bulk-synchronous burden from the implementation of `transfer`.

6.1 From bulk-synchrony to point-to-point communications

The semantics of `transfer` modeled in MSL have barrier-like behavior. A straightforward implementation is to simulate `transfer` by `send` and `recv` operations together with a `barrier` in MPI, as depicted in Figure 6-1(a). It uses `MPI_Isend` because at least one of the `send` and `recv` needs to be asynchronous in order to avoid deadlock. Figure 6-1(b) illustrates the runtime behavior of this implementation. Note that `MPI_Wait` is omitted in the picture.

```
(a) inline void transfer(int len, bool scond, double * sendbuf, int rid, bool rcond, double * recvbuf) {
    MPI_Request req;
    MPI_Status status;
    if (scond) {
        MPI_Isend(sendbuf, len, MPI_DOUBLE, rid, MPI_ANY_TAG, MPI_WORLD, &req);
    }
    if (rcond) {
        MPI_Recv(recvbuf, len, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_WORLD);
    }
    if (scond) {
        MPI_Wait(&req, &status);
    }
    MPI_Barrier(MPI_WORLD);
}
```

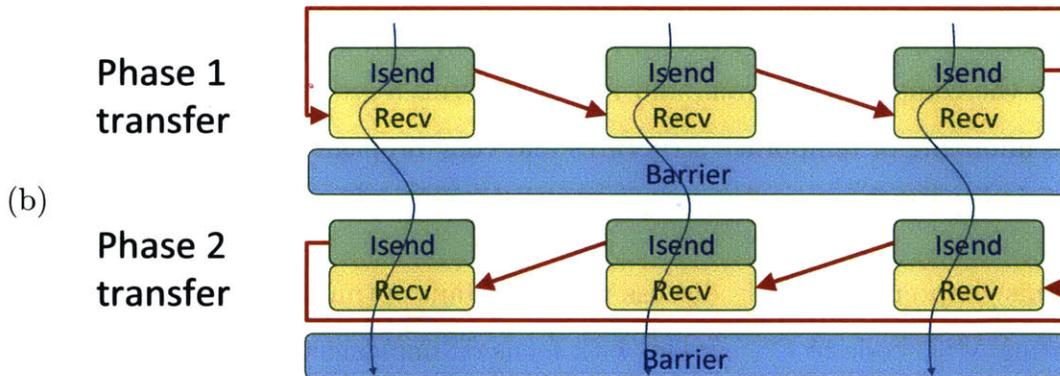


Figure 6-1: (a) the straightforward MPI implementation for `transfer` and (b) an illustration of its runtime behavior

The bulk-synchronous model of `transfer` (just as all the synchronization / communication mechanisms in MSL) allows us to talk about `transfer` in phases, because it must be called by all processes in tandem. The barriers after each `transfer` are necessary to enforce the correct behavior: before each `barrier`, all the outstanding messages are

guaranteed to come from one phase of `Isend`, so they are received by the `Recv` from the same phase. If we naïly remove the barriers as shown in Figure 6-2, buggy behavior can happen: Here process 0 executes faster and marches to Phase 2 ealier, and the other processes are still at Phase 1. The Phase 2 message sent by process 0 reaches process 2 before the Phase 1 message sent by process 1, so the Phase 2 message is received by process 2's Phase 1 `recv`, violating the semantics of `transfer`.

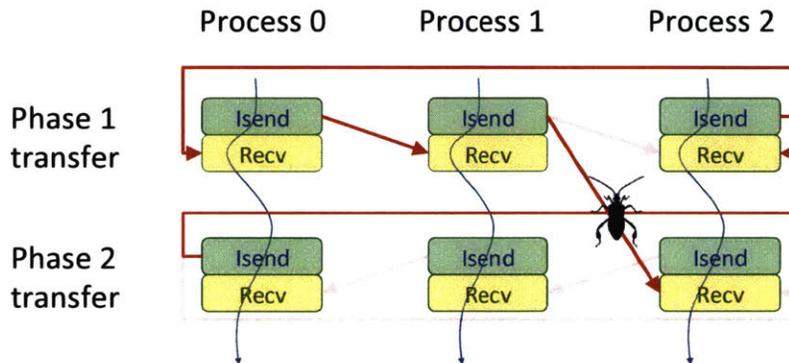


Figure 6-2: Buggy behavior after removing barriers naïvely

To prevent such bugs without using heavy-weight barriers, we exploit the *message tagging* feature of MPI: each `send` operation can tag the message being sent with an integer; each `recv` operation can specify its expected tag, and only the incoming message with a matching tag can fulfill that `recv` operation. Because logically all the `transfer` operations happen in phases, we maintain a counter of the current phase number, and tag messages with this counter. This new optimized implementation is shown in Figure 6-3. Without the barriers, there might be outstanding messages coming from the `MPI_Isend` of multiple phases at any moment, but the `MPI_Recv` is also tagged, so it only accepts the correct message from its own phase, and is shielded from mismatched `MPI_Isend`, as shown in Figure 6-4.

With the message tags taking over the role of the barriers, this optimization is intuitively correct. To formally prove it, we first model the implementation in Figure 6-3 with a new set of semantics that replaces the old semantics for `transfer` established in Figure 4-6 and Figure 4-9. Using the new formal semantics, we will prove that the SPMD-to-sequential transformation process presented in Section 4.3 still holds, so that

```

// initialization of the phase counter
int phase = 0;

inline void transfer(int len, bool scond, double * sendbuf, int rid, bool rcond, double * recvbuf) {
    int tag = ++phase;
    MPI_Request req;
    MPI_Status status;
    if (scond) {
        MPI_Isend(sendbuf, len, MPI_DOUBLE, rid, tag, MPI_WORLD, &req);
    }
    if (rcond) {
        MPI_Recv(recvbuf, len, MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_WORLD);
    }
    if (scond) {
        MPI_Wait(&req, &status);
    }
}

```

Figure 6-3: an optimized implementation for `transfer` (slightly simplified)

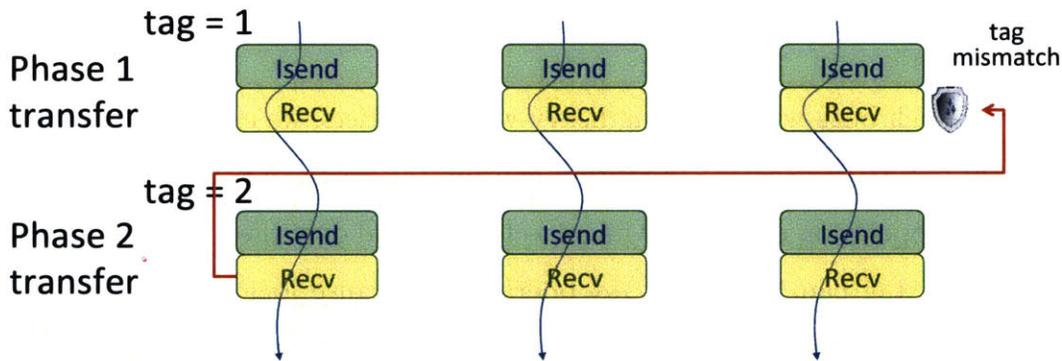


Figure 6-4: Using tagging to prevent mismatched messages

any MSL program with the new `transfer` semantics can still be sequentialized by the same transformation rules listed in Figure 4-11. Hence the behavior of the parallel program is equivalent to the sequentialized program, which is deterministic, regardless of which set of semantics we use for `transfer`, old or new, so the new implementation of `transfer` must preserve the program meaning.

6.2 Formally proving the correctness of the optimized point-to-point transfer

Figure 6-5 shows the new formal semantics for `transfer`: `transfer` is translated into three new finer-grained instructions, $t\text{send}(s\text{cond}, e, r\text{id})$, $t\text{recv}(r\text{cond}, x)$ and $t\text{out}(s\text{cond}, r\text{id})$, whose execution semantics are also listed.

$$\llbracket \text{transfer}(s\text{cond}, e, r\text{id}, r\text{cond}, x) \rrbracket_{Tr} = t\text{send}(s\text{cond}, e, r\text{id}); t\text{recv}(r\text{cond}, x); t\text{out}(s\text{cond}, r\text{id})$$

(a) the new translation from `transfer` to finer-grained instructions

$$\frac{\begin{array}{l} I(pc) : t\text{send}(s\text{cond}, e, r\text{id}) \\ m = L[p].\text{tag} + 1, n = (r\text{id})_{L[p]} \\ L'[p].\text{tag} = m \\ (s\text{cond})_{L[p]} \Rightarrow B'[n][m] = (e)_{L[p]} \end{array}}{\sigma \xrightarrow{t\text{send}(s\text{cond}, e, r\text{id})}_p \sigma'}$$

$$\frac{\begin{array}{l} I(pc) : t\text{recv}(r\text{cond}, x) \\ m = L[p].\text{tag}, n = B[p][m] \\ (\neg r\text{cond})_{L[p]} \vee n \neq \perp \\ (r\text{cond})_{L[p]} \Rightarrow L'[p].x = n \\ (r\text{cond})_{L[p]} \Rightarrow B'[p][m] = \top \end{array}}{\sigma \xrightarrow{t\text{recv}(r\text{cond}, x)}_p \sigma'}$$

$$\frac{\begin{array}{l} I(pc) : t\text{out}(s\text{cond}, r\text{id}) \\ m = L[p].\text{tag}, n = (r\text{id})_{L[p]} \\ (\neg s\text{cond})_{L[p]} \vee B[n][m] = \top \end{array}}{\sigma \xrightarrow{t\text{out}(s\text{cond}, r\text{id})}_p \sigma'}$$

(b) execution semantics of the new instructions

Figure 6-5: The new semantics of the optimized `transfer`

In this new semantics model, we will assume a local variable $L[p].\text{tag}$ that is initialized to 0 on each process p . The communication channel for `transfer` now needs to be modeled as $B[p][m] : \text{Pid} \rightarrow (\text{Int} \rightarrow (\text{Int} \cup \{\perp, \top\}))$, an ordered array of mailboxes (indexed by m , the message tag) for each process p . It is initialized to $(\lambda p. \lambda m. \perp)$. $B[p][m] = \perp$ means position m of the buffer at process p has no incoming data yet, and $B[p][m] = \top$ means that the receiving process p has already copied the m -th incoming message to its local variable, and whoever has sent the m -th message can stop waiting for its own `MPI_Isend` to complete.

Figure 6-6 compares the usage of the old channel and the new channel. With the old `transfer` semantics and the old channel, there is only one cell (mailbox) per each subprocess. Conceptually there is a barrier after each phase of `transfer`. The `tin` instruction just writes to the receiving process's cell, and each subprocess can re-use its only mailbox across phases, because the barrier-like behavior of `transfer` requires

every subprocess to retrieve the incoming message from its mailbox (if there is any) before all processes can move to the next phase. For the same reason, there is no need to use special values \perp and \top to indicate if a mailbox is filled and if the message has already been retrieved, because the strict timing of the *tin*, *trn*, and *wait* instructions guarantees that a message is always written by some *tin* before *trn* is executed, and an incoming message is always retrieved by some *trn* before the sender process exits the transfer statement (by executing its *wait*).

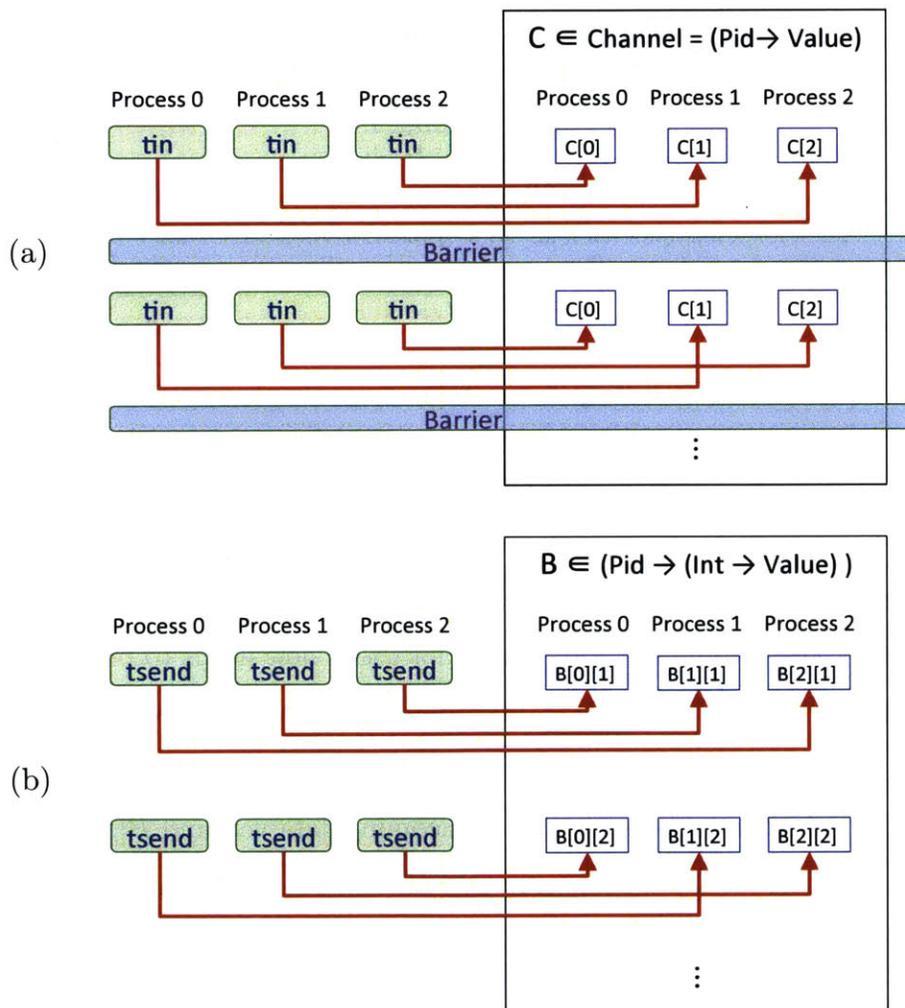


Figure 6-6: Illustration of transfer with (a) the old channel and (b) the new channel

With the new transfer semantics and the new channel, there is no longer the barrier-like behavior and the strict timing constraints, so each process cannot re-use a single mailbox for multiple phases of transfers. Instead, there must be a separate cell $B[p][m]$

reserved for every phase m per process p . Then the $t\text{send}$ from phase m can write to the m -indexed mailbox of the receiving process, without conflicting with other $t\text{send}$ from a different phase. The special value \perp is also needed to indicate whether a mailbox is filled with data or not, and the special value \top is used to indicate whether an incoming message has been retrieved by its receiver or not.

Using the new channel definition, the three new instructions $t\text{send}$, $t\text{recv}$, and $t\text{out}$ fit the behavior of the implementation in Figure 6-3 closely. We contrast them with the old transfer instructions $t\text{in}$, $t\text{rn}$, and $t\text{wait}$ by putting the new and old execution rules side by side:

$$\begin{array}{c}
 \boxed{
 \begin{array}{l}
 I(pc) : t\text{send}(s\text{cond}, e, rid) \\
 m = L[p].tag + 1, n = (rid)_{L[p]} \\
 L'[p].tag = m \\
 (s\text{cond})_{L[p]} \Rightarrow B'[n][m] = (e)_{L[p]} \\
 \hline
 \sigma \xrightarrow{t\text{send}(s\text{cond}, e, rid)}_p \sigma'
 \end{array}
 }
 \quad \text{v.s.} \quad
 \boxed{
 \begin{array}{l}
 I(pc) : t\text{in}(s\text{cond}, e, rid) \\
 \forall u. PC(u) \in \{pc, pc + 1\} \\
 (s\text{cond})_{L[p]} \Rightarrow C'[(rid)_{L[p]}] = (e)_{L[p]} \\
 PC'(p) = pc + 1 \\
 \hline
 \sigma \xrightarrow{t\text{in}(s\text{cond}, e, rid)}_p \sigma'
 \end{array}
 }
 \end{array}$$

The main difference is: the old $t\text{in}$ needs to do barrier-like waiting for other processes to come to the same place before moving, while the new $t\text{send}$ does not. After the waiting, the process executing $t\text{in}$ just write the data to the recipient process's mailbox, knowing that the message will later be retrieved by some $t\text{rn}$ from the same phase as itself. Instead of waiting, $t\text{send}$ needs to increment the phase counter and tag the data being sent with the counter value m (putting it into the m -indexed mailbox of the recipient process). This mimics the behavior of the asynchronous `MPI_Isend` with tagging. It does not wait to check if the message has been received (copied to a local variable) by the receiver.

$$\begin{array}{c}
 \boxed{
 \begin{array}{l}
 I(pc) : t\text{recv}(r\text{cond}, x) \\
 m = L[p].tag, n = B[p][m] \\
 (\neg r\text{cond})_{L[p]} \vee n \neq \perp \\
 (r\text{cond})_{L[p]} \Rightarrow L'[p].x = n \\
 (r\text{cond})_{L[p]} \Rightarrow B'[p][m] = \top \\
 \hline
 \sigma \xrightarrow{t\text{recv}(r\text{cond}, x)}_p \sigma'
 \end{array}
 }
 \quad \text{v.s.} \quad
 \boxed{
 \begin{array}{l}
 I(pc) : t\text{rn}(r\text{cond}, x) \\
 \forall u. PC(u) \in \{pc, pc + 1\} \\
 (r\text{cond})_{L[p]} \Rightarrow L'[p].x = C[p] \\
 PC'(p) = pc + 1 \\
 \hline
 \sigma \xrightarrow{t\text{rn}(r\text{cond}, x)}_p \sigma'
 \end{array}
 }
 \end{array}$$

The main difference is again that $t\text{rn}$ needs to do barrier-like waiting while $t\text{recv}$ does not. Once the barrier-like wait finishes, the process executing $t\text{rn}$ can just copy the data in its mailbox to its local variable x , because it knows that this must be sent by some $t\text{in}$ during the same phase. On the other hand, the process executing $t\text{recv}$ needs to pick the Phase m data from its m -indexed mailbox, where m is its local phase

counter, and it needs to wait for that mailbox to be filled by some *tsend* in Phase m ($n = B[p][m], (\neg rcond)_{L[p]} \vee n \neq \perp$). *trecv* corresponds to the `MPI_Recv`: it waits for the data with the correct tag to be available, copies the data to a local variable, and acknowledges the delivery of the message ($(rcond)_{L[p]} \Rightarrow B'[p][m] = \top$).

$$\boxed{\begin{array}{l} I(pc) : tout(scond, rid) \\ m = L[p].tag, n = (rid)_{L[p]} \\ (\neg scond)_{L[p]} \vee B[n][m] = \top \\ \hline \sigma \xrightarrow{tout(scond, rid)}_p \sigma' \end{array}} \text{ v.s. } \boxed{\begin{array}{l} I(pc) : wait \\ \forall u. PC(u) \neq pc - 1 \\ PC'(p) = pc + 1 \\ \hline \sigma \xrightarrow{wait}_p \sigma' \end{array}}$$

Again *wait* does a barrier-like synchronization. *tout*, on the other hand, corresponds to the `MPI_Wait` doing only pairwise synchronization. The process executing *tout* just waits for its message to be received by the recipient process ($(\neg scond)_{L[p]} \vee B[n][m] = \top$).

To show that the new implementation of *transfer* actually preserves the meaning of the program from the old implementation, we just need to prove that any parallel program under the new scheme of semantics can still be sequentialized by the same transformation rules shown in Figure 4-11. Because the sequentialized program is deterministic for a given parallel program, this will guarantee that the original program has the same behavior regardless of which *transfer* implementation it uses. We will not repeat the same proof steps already discussed in Section 4.3, because most part of the original proof still holds without any change, but the proof of the `SEQ` rule needs to adapt to the new instructions *tsend*, *trecv*, and *tout*. The only relevant part of the proof that might be affected is Lemma 4.3.1, which relies on the barrier-like property of the communication operations, which in turn, is provided by Theorem 4.2.1. So the only interesting task is to provide a new theorem that substitute the role of Theorem 4.2.1, under the new scheme of semantics for *transfer*.

Definition 6.2.1. We call a *tsend* action with message tag m a *tsend_m* action. Simillary we define *trecv_m* and *tout_m* actions.

Theorem 6.2.2. For any execution trace t , we can re-arrange its actions (preserving its semantics) into a new trace t' so that for any m , the *tsend_m* actions from all subprocesses are grouped together in t' . The similar facts hold for *trecv* and *tout*.

Its proof builds on Lemma 6.2.3 that allows us to swap consecutive actions.

Lemma 6.2.3. If there are two consecutive actions x, y in t executed by different subprocesses, where x is a $t\text{send}_m$ action, and y is any action except for $t\text{send}_m$, $t\text{recv}_m$, or $t\text{out}_m$ actions, then we can always swap them to be y, x and get an equivalent trace.

Proof. It is easy to see that the actions other than $t\text{send}$, $t\text{recv}$, and $t\text{out}$ are all right-movers with respect to a $t\text{send}_m$ event, so we only need to consider the case where y is $t\text{send}_n$, $t\text{recv}_n$, or $t\text{out}_n$ with $n \neq m$. In this case, their message tags are different, so x and y access non-overlapping portions of the system state, thus are movers to each other. \square

Using Lemma 6.2.3, for any $t\text{send}_m$ we can always find its first occurrence in trace t , and suppose it is action x executed by process p , all the other $t\text{send}_m$ actions executed by other processes are after x , but maybe not immediately following x , but we can always swap actions such that all the $t\text{send}_m$ actions are grouped together. Similar arguments extend to the $t\text{recv}$ and $t\text{out}$ actions, and this completes the proof of Theorem 6.2.2. It shows that although the new `transfer` does not enforce barrier-like property at individual instruction level, the resulting execution trace is always equivalent to a “good” trace in which all three steps of `transfer` do exhibit barrier-like behaviors.

Then we can just focus on the “good” traces when we re-apply the proof of Lemma 4.3.1, which provides the basis for proving the SEQ rule, thus completing the proof of SPMD-to-sequential transformation rules in Figure 4-11, under the new scheme of `transfer` semantics.

Therefore any parallel program under the new scheme of `transfer` semantics can be sequentialized by the same transformation process as under the old semantics, so the two schemes must be semantically equivalent to begin with.

6.3 Other implementation details

The other communication mechanisms in MSL are translated to their MPI counterparts in a straightforward way, such as `All_to_all` and `reduce`. The special values `mypid` (the process

id) and `nprocs` are obtained at the beginning of the code by calling `MPI_Comm_rank` and `MPI_Comm_size`.

Aside from the SPMD programming features, the MSL compiler does some other optimizations when generating C++ code. One of them worth mentioning is with arrays — the arrays in MSL have copy semantics, which means that when an array is passed in as a parameter, semantically the callee gets a copy of the original array. To prevent excessive copying, arrays are copied lazily: the compiler will only generate the code that copies the input array if the array is modified by the callee. If an array is used as a reference parameter, the callee is free to modify the array without copying it. These optimizations, together with aggressive inlining and copy propagation, eliminate most unnecessary array copying.

Chapter 7

Combining distributed kernels

The previous chapters focus on how individual kernels are written, synthesized, and checked in MSL. This chapter discusses another important issue: how to combine multiple MSL kernels together to form a bigger application, without losing the correctness benefits provided by MSL for each individual kernel.

For discussion purpose here, kernels are just functions. Consider a very simple example of composing two distributed kernels f and g , shown in Figure 7-1. The `main` calls both f and g as subroutines, and information (data) flows through input/output relationships among the subroutines.

```
spmd void main(int n,  
               int[n/nprocs] a, ref int[n/nprocs] b) {  
    int[n/nprocs] c = f(n, a);  
    g(n, c, b);  
}  
  
spmd int[n/nprocs] f(int n, int[n/nprocs] x);  
  
spmd void g(int n,  
            int[n/nprocs] x, ref int[n/nprocs] y);
```

```
void main'(int n,  
           int[n] a, ref int[n] b) {  
    int[n] c = f'(n, a);  
    g'(n, c, b);  
}  
  
int[n] f'(int n, int[n/nprocs] x);  
  
void g'(int n,  
        int[n/nprocs] x, ref int[n/nprocs] y);
```

Figure 7-1: Simple example of composing distributed kernels

Side by side, we have shown another *sequential* function `main'` that calls two sequential subroutines f' and g' . The structure of `main` closely resembles `main'`, but `main` and its subroutines execute in SPMD manner and the data they use (a , b , c) have the “/nprocs” factor in their array lengths, reflecting the distributed nature of the program. Checking the correctness of `main` against `main'` can be done as before — writing a test harness

and distribute / collect functions, and relying on the solver to check the test harness. However, when both `f` and `g` are complex functions, the overall test harness is even more complex and harder for the solver. In the real applications where many kernels are composed together, the synthesis time of the solver grows exponentially as the program gets bigger, essentially forbidding the approach of checking the whole program. If we know that both `f` and `g` can be checked separately against `f'` and `g'`, which is tractable for the solver as we have discussed before, can we leverage that extra information to guarantee the correctness of `main` without checking the giant, monolithic program?

When the code uses raw data types like arrays freely as shown in Figure 7-1, it is not easy for the compiler to reason about the structural resemblance of `main` and `main'`. However, when the data types are abstracted and used in a more restricted manner, as shown in Figure 7-2, the structural equivalence manifests itself quite obviously: here, if we can establish that `Grid` and `Grid'` are just two different implementations of the same abstract data type, and that `f / g` and `f' / g'` are corresponding implementations of the same operations over the data type, then without going deeply into the semantic details, we can establish the correspondence between `main` and `main'` at a syntactic level.

| | |
|--|--|
| <pre> spmd void main(Grid a, ref Grid b) { Grid c = f(a); g(c, b); } spmd Grid f(Grid x); spmd void g(Grid x, ref Grid y); </pre> | <pre> void main'(Grid' a, ref Grid' b) { Grid' c = f(a); g(c, b); } Grid' f'(Grid' x); void g'(Grid' x, ref Grid' y); </pre> |
|--|--|

Figure 7-2: Re-organizing the example with abstract data type

This chapter builds upon that high-level idea and describes how MSL solves the challenges of composing and checking kernels:

1. The first challenge is, for a “driver” function like `main` in the example, how to let it freely choose from different sets of implementations of data representations and functions (one set is the SPMD version that operates on partitioned, local states, and the other set is the sequential version that operates on non-partitioned state)? The MSL solution is to provide a package system to facilitate data representation abstraction. A *package* is a standard ADT (abstract data type) [69] implementation

like the module systems in Haskell and OCaml. It wraps a set of data type definitions and functions together, and allows the writer to control the *visibility* of each type or function. In particular, she can make some types *opaque*. The definitions of the opaque types are encapsulated inside the package, and these types can only be read and written by the functions in that package. From outside, the client code only uses the names of the opaque types symbolically: it never manipulates the values of those types directly, but only through the package’s public functions. With these means, opaque types and the operations (functions) over them constitute an *abstract interface* to the client. The client program, once written according to the restrictions, can freely choose between different implementations provided by different packages without code change, a benefit of “programming against the interface instead of the implementation” that has long been advocated.

2. In addition to that, can we *check* that two packages with the same interface actually implement the same functionality? This is not covered by standard ADT systems, as it has to rely on semantic analysis. We will present the mechanism in MSL that checks if a package P_2 is a *refinement* of P_1 by checking each public function separately. The refinement relationship holds for the pair of packages regardless of the client code — actually MSL checks for the *most general client*, thus provides the most general guarantee. This is based on a concept we call *box / unbox* function pairs, which is similar to and a generalization of the *distribute / collect* functions in the prior chapters. We give a soundness proof of this equivalence checking system, which is quite involved. This checking system relies on separate, local checks for each individual public functions, and because those local checks are bounded model checking, the overall system only provides bounded guarantees for the global correctness. However, if the local checks are done by sound methods, the whole equivalence checking system will also be sound.

3. Finally, does the equivalence checking system adds extra runtime overhead? We will see that those *box / unbox* functions, just like the *distribute / collect* functions we have discussed before, are *ghost code* [32, 105]. They vanishes at runtime, and the system only executes the actual SPMD kernel code efficiently.

7.1 Package System in MSL

In MSL, the programmer can wrap type and function definitions inside a *package*, and control the visibility of the types and functions.

Each type can be defined (currently MSL only allows one kind of type definition: `struct`) as `private` (the default case), `opaque`, or `public`. The outside user cannot see or use a `private` type; she can see the full definition of any `public` type, and use it just as any normal `struct` type: for example, she can inspect its `struct` members, and construct a fresh value of the `public` type using `new`. The `opaque` types are special: from outside, the user can only see the name of the `opaque` type, and use it to declare the type of some variables, but she cannot see the definition of the `opaque` type, cannot inspect its internal members, and cannot construct new values. The only way she can manipulate values of a `opaque` type is through the `public` functions of the package: those functions may take `opaque` typed parameters as input, and may output `opaque` typed values. Each function can be defined as either `private` (the default case) or `public` in a package. `Private` function can only be called by functions inside the package. `Public` functions can be seen and called from outside the package, and the signature of any `public` function cannot involve `private` types.

Below is an example package:

```
package P1 {
  opaque struct Pair {
    int x;
    int y;
  }
  public Pair construct(int a, int b) {
    return new Pair(x=a, y=b);
  }
  public int sum(Pair u) {
    return u.x + u.y;
  }
  public Pair prod(Pair u, Pair v) {
    int a = u.x * v.x;
    int b = u.y * v.y;
    return new Pair(x=a, y=b);
  }
}
```

In package `P1`, the programmer has defined a `opaque` type `Pair`, a pair of integers. The only way the package user can deal with any data of type `Pair` is to call the `public`

functions `construct`, `sum`, and `prod` (which returns a new `Pair`, the element-wise product of the two input `Pair` objects). An example of how to use the package is shown below:

```
int user1(int[2] a, int[2] b) {
    using P1;
    Pair X = construct(a[0], b[0]);
    Pair Y = construct(a[1], b[1]);
    Pair Z = prod(X, Y);
    int c = sum(Z);
    return c;
}
```

A package's *interface* is the collection of the names and signatures of its public functions, the names of its opaque types, and the names and definitions of its public types. From outside, a user can only observe the interface of a package. A nice thing about the package abstraction is that packages with the same interface can replace each other. When the programmer wants to use another package `P2` with the same interface as `P1`, she only needs to change one single `using` statement, as shown below:

```
int user2(int[2] a, int[2] b) {
    using P2;
    Pair X = construct(a[0], b[0]);
    Pair Y = construct(a[1], b[1]);
    Pair Z = prod(X, Y);
    int c = sum(Z);
    return c;
}
```

where `P2` can be a different implementation than `P1`. A possible implementation of `P2` is shown below:

```
package P2 {
    opaque struct Pair {
        int x;
        int y;
        int s; // caches x+y
    }
    public Pair construct(int a, int b) {
        return new Pair(x=a, y=b, s=a+b);
    }
    public int sum(Pair u) {
        return u.s;
    }
    public Pair prod(Pair u, Pair v) {
        int a = u.x * v.x;
        int b = u.y * v.y;
        return construct(a, b);
    }
}
```

Note that the type `Pair` has a different definition in `P2` than in `P1`. In `P2`, instead of calculating the sum each time, functions that may write to a `Pair` calculate and cache the sum inside `Pair.s`, and the `sum` function avoids re-calculation and just returns the cached sum. The user of the packages does not observe the implementation difference: the very code snippet that uses `P1` is still valid without modification when `P1` is replaced by `P2`; not only is it still valid, it must produce the same result as before. It is easy to verify that the functions `user1` and `user2` always produce the same results given the same input values, while this is true for any complex usage of packages `P1` and `P2`. In this case, we say that `P2` is a *refinement* of `P1`, or simply `P2` *refines* `P1`.

Now comes an interesting part: if we want the functions in `P2` to perform correctly, the `prod` function in `P2` must (directly or indirectly) calculate and cache the sum for the newly created `Pair` object. If we incorrectly define the `prod` function and get something like the package `P3`, things will go wrong.

```
package P3 {
  ... // all definitions are the same as in P2, except for the prod function
  public Pair prod(Pair u, Pair v) {
    ... // all lines are the same as in P2, except for the last line
    return new Pair(x=a, y=b);
  }
}
```

In `P3`, the `prod` function just returns a new `Pair` without calculating `Pair.s`, either directly or indirectly, and by default `Pair.s` gets 0 value. Then the following calls to the `sum` function on the returned `Pair` object will go wrong, thus a code snippet that uses `P1` is not guaranteed to produce the same result if the user replaces `P1` with `P3`. We say in this case that package `P3` does not refine `P1`, although syntactically `P3` can replace `P1` perfectly.

Note the problem here is *non-local*: it is not that the `prod` function has calculated the element-wise product incorrectly, but it's some invariant being broken, which can only be observed later when the user calls the `sum` function. In the next sections we will show how we catch this kind of bugs by the equivalence checking mechanism already built in MSL.

The package system we have introduced above, and the mechanism to check package refinement, are useful to a wide range of programming problems, and in this thesis

we will see that they nicely provide a ground for checking SPMD against sequential implementations: when P_1 contains only sequential kernels, and P_2 contains SPMD kernels that manipulate distributed data, establishing refinement relation between P_1 and P_2 naturally guarantees that the SPMD implementation behaves equivalently to the sequential one.

7.2 Defining the refinement relation

First we define what we mean by package *refinement*. In previous chapters we introduced the definition of equivalence at the *function* level: if two functions have the same function signature, and they produce the same output for all possible input, then they are functionally equivalent, even if they have vastly different implementations. Similarly, we define the refinement relationship among packages by their behaviors observed from outside. Recall that a package's *interface* is the collection of the names and signatures of its public functions, the names of its opaque types, and the names and definitions of its public types. For package P_2 to refine P_1 , they must have the same interface. Furthermore, P_2 must exhibit the same behavior as P_1 when used by any client. In fact, we just need to require that P_2 behave equivalently to P_1 when used by the *most general client*, a concept widely used [11, 93, 98, 99] in the Programming Languages and Software Engineering areas, which is an over-approximation of all legal clients of the package, and is used to exhibit all possible behaviors of the package.

Definition 7.2.1. An n -step *most general client* of an interface I is a program that:

1. Initially (at step 0), the program state may contain any number of variables and heap cells of arbitrary values, but none of those values can be of opaque types in I .
2. At each step i , the program may choose to execute any action, which can be a normal MSL statement or a call to some public function in I . The input to this action must be chosen from the values that the program has after step $i - 1$, or any primitive constants. Note that executing the action can create new values which may be of opaque types in I .
3. The program executes for n steps, and the *final state* is the collection of all the

variables and heap cells after step n .

As we can see, the most general client is defined against an abstract interface I rather than a concrete package, so that it can be applied to different packages implementing the same interface. We say that a package P_2 refines P_1 if it has the same interface as P_1 , and for any n , the n -step most general client cannot distinguish between P_1 and P_2 , *i.e.*, there is no observable difference in the final state between the case of using P_1 and the case of using P_2 .

To make that high-level intuition rigorous, we first define formally some concepts. We consider an execution of a program (including the most general client) as applying a sequence of *actions* on a *state*. For our discussion in this chapter, we define a *state* S to be a collection of variables and heaps, $S = (V, H, O)$, where V refers to the collection of all typed variables (each variable may be either public typed or opaque typed), H refers to the *public heap* (whose cells are all public typed), and O refers to the *opaque heap* (whose cells are all opaque typed).

The value of any variable or heap cell can be a primitive value like integer and float, or a structure, or a pointer (which points to either the public heap or the opaque heap). Additionally there is a value \perp to indicate the “uninitialized value”. We want to be able to compare two values for equality, and when they are both primitives, things are easy, but when they are pointers, we need to deal with the whole heap. Given two public heaps H_1 and H_2 , and a bijection ϕ from H_1 to H_2 , (*i.e.* given a heap cell $c \in H_1$, $\phi(c) \in H_2$), we say that H_1 *conforms to* H_2 under ϕ (written as $H_1 \sqsubseteq_\phi H_2$) iff the check below passes:

Step 1. For any cell $x \in H_1$, let $y = \phi(x)$ denote its corresponding cell in H_2 . they must have the same type.

Step 2. If x and y are both primitive, they must have the same primitive value.

Step 3. If x and y are both pointers, and they are both pointing to public heaps: suppose x points to c , then y must points to $\phi(c)$. If x and y are both pointing to opaque heaps, then there is no restriction. If one of them points to a public heap, and the other points to an opaque heap, then the check fails.

Step 4. If x and y are both structures of the same type, then the check is recursively

done for each corresponding field.

Definition 7.2.2 (Observational Similarity). We say that the two states S_1 and S_2 are *observationally similar* when they contain the same collection of variable names, the same number of initialized public heap cells, and there exists a bijection ϕ from H_1 to H_2 such that $H_1 \sqsubseteq_{\phi} H_2$, $H_2 \sqsubseteq_{\phi^{-1}} H_1$, and for all variables x , the pair of values $V_1[x]$ and $V_2[x]$ passes the above check.

It is important to note that observational similarity is restricted to the public (*observable*) parts of the state, and does not imply that the two states are considered equivalent. Equivalence is a much stronger property: for two states S_1 and S_2 to be equivalent, they must be observationally similar, *and* when the execution start from S_1 and S_2 in sync, no matter how many actions are executed, the resulting states never diverge in their observable parts, as we will see later.

An *action* a is either some normal MSL statement (like an assignment or a call to a non-packaged function) that involves solely public types, or a call to a public function (which may involve opaque types) of an interface I . It has the effect of transforming the state, *i.e.* given a state S , $a(S)$ produces a new state S' by reading and changing some of the variables and / or heaps of S . We do assume every action to be deterministic.

A *trace* is just a finite sequence of actions. When a trace t includes calls to public functions of an interface, it is *abstract* because an interface does not tell the implementation detail (the exact effect of the actions), so we use $t@P$ to denote a *concrete trace* in which all calls to the public functions should resolve to functions in package P , and then the effect of each action as well as the whole trace is determined. For example, for the (abstract) trace

$$t = \{Pair\ x = construct(5, b)\}, \{int\ c = sum(x)\}, \{int\ d = c + 1\}$$

and a package $P1$ (as defined in Section 7.1), the concrete trace is:

$$t@P1 = \{Pair@P1\ x = construct@P1(5, b)\}, \{int\ c = sum@P1(x)\}, \{int\ d = c + 1\}$$

Definition 7.2.3 (Package Refinement). Executing the most general client for an interface I (which packages $P1$ and $P2$ both implement) can generate an infinite set of (abstract) traces. We define that $P2$ refines $P1$ iff for every trace t generated by such n -step most general client (where n can be any positive integer), $t@P1$ and $t@P2$ produce *observationally similar* final states (similar as defined in Definition 7.2.2).

7.3 Establishing equivalence between packages

We cannot check for refinement between two packages by naively following Definition 7.2.3, because that requires us to work with infinitely many traces, each of which may involve arbitrarily many steps. The usual way of dealing with infinity is by induction. Suppose we are given some oracle that can tell not only if two states are *observably similar*, but also whether two states (one with package $P1$, the other with package $P2$) are essentially *equivalent*. With that, we want to prove for any possible action a , if we prepare two equivalent states S_1 and S_2 before executing a , and apply $a@P1$ to S_1 , $a@P2$ to S_2 , the resulting states S'_1 and S'_2 are guaranteed to be still equivalent; then mathematical induction will extend this property to arbitrarily long traces.

To get that oracle, MSL requires the package writer to provide an *unbox* function, which converts an opaque value to an observable public representation, such that the checker can reason about equivalence between states; also the package writer needs to provide a *box* function, which constructs an opaque value from public values, such that the checker can prepare equivalent states when needed.

This is required for every opaque type τ in a package. A box function for τ takes as input some public values, and returns a value of type τ . An unbox function does the opposite: it takes as input a value of type τ , and outputs some public values.

Below is an example of the box/unbox functions for the packages $P1$ and $P2$ defined in Section 7.1.

```
package P1 {
  opaque struct Pair {
    int x;
    int y;
```

```

}

box Pair bf(int a, int b) {
  return new Pair(x = a , y = b);
}

unbox void ubf(ref int a, ref int b, Pair v) {
  a = v.x;
  b = v.y;
}
...
}

package P2 {
  opaque struct Pair {
    int x;
    int y;
    int s; // caches x+y
  }
  box Pair bf(int a, int b) {
    return new Pair(x = a , y = b, s = a + b);
  }
  unbox void ubf(ref int a, ref int b, Pair v) {
    assert v.s == v.x + v.y;
    a = v.x;
    b = v.y;
  }
  ...
}

```

Notice the “**assert**” statement in the unbox function `ubf` in package `P2`: it states the *representation invariants* [45] that has to be maintained by any valid value of type `Pair@P2`. The nice thing is that in MSL, the invariants can be expressed using the same language as the normal code.

Without loss of generality, from now on we assume that the box / unbox functions all take a single input and produce a single output, because multiple public values can always be wrapped in a public struct. Then a box function is just a mapping from a public type to an opaque type, and an unbox function is a mapping from an opaque type to a public type. We further require that: any box function must be *surjective* (onto function); any unbox function must be *injective* (one-to-one function). This can be checked by the solver for each box / unbox function separately.

We can efficiently check the equivalence of two packages `P1` and `P2` with the same interface if they satisfy Condition 7.3.1.

Condition 7.3.1. For any opaque type τ , defined in both `P1` and `P2`, there is a box

function for τ with the same signature in both $P1$ and $P2$, and there is an unbox function for τ with the same signature in both $P1$ and $P2$. The box functions are all surjective, and the unbox functions are all injective.

Let's assume that the box function is $\tau_{box_T(\alpha_T)}$, i.e. it takes as input a public type α_T (might be a public struct type), and returns a type τ ; the unbox function is $\beta_T unbox_T(\tau)$, i.e. it takes as input a type τ , and returns a public type β_T (not necessarily the same as α_T). The pair of functions box_T and $unbox_T$ are defined with the same signatures in both packages $P1$ and $P2$. Of course the type τ means different things from $P1$ to $P2$, but the public types α_T and β_T must have the same meaning across packages.

If the two packages satisfy Condition 7.3.1, we can check for their equivalence *locally*: we do not need to enumerate traces, but we only need to check the equivalence of each public function separately, by the following procedure:

Procedure 7.3.2 (Package refinement check).

(a) For every public function f of the common interface implemented by both packages $P1$ and $P2$, f may take some inputs, and write to some outputs. Suppose the inputs to f involve opaque types T_1, T_2, \dots, T_m , and the outputs of f involve opaque types S_1, S_2, \dots, S_n . We replace each T_i with α_{T_i} , and each S_i with β_{S_i} , and get another function signature, which only involves public types. With this new signature, we define two functions $f1$ and $f2$:

$f1$ first calls each $box_{T_i}@P1$ to transform the input of public type α_{T_i} into the opaque type $\tau_i@P1$, and then uses them to call the original public function $f@P1$, and afterwards calls each $unbox_{S_i}@P1$ to transform each output of opaque type $s_i@P1$ into public type β_{S_i} .

$f2$ first calls each $box_{T_i}@P2$ to transform the input of public type α_{T_i} into the opaque type $\tau_i@P2$, and then uses them to call the original public function $f@P2$, and afterwards calls each $unbox_{S_i}@P2$ to transform each output of opaque type $s_i@P2$ into public type β_{S_i} .

Now $f1$ and $f2$ have the same signature involving only public types, and they call

$\mathfrak{f}@P1$ and $\mathfrak{f}@P2$ respectively. We check that \mathfrak{f}_1 and \mathfrak{f}_2 are functionally equivalent using the mechanism discussed in Chapter 5.

(b) For every opaque type T , we check that for any input X , $unbox_T@P1(box_T@P1(X))$ generates the same output as $unbox_T@P2(box_T@P2(X))$.

Theorem 7.3.3. For two packages $P1$ and $P2$ with the same interface satisfying Condition 7.3.1, if all the checks in Procedure 7.3.2 pass, then $P1$ refines $P2$ (refinement as defined in Definition 7.2.3).

The proof is based on the induction strategy we described earlier, and a starting point is to lift the box and unbox functions to deal with states.

Definition 7.3.4 (Unbox and box function for the state). Given a packages P satisfying Condition 7.3.1, we lift the unbox function from values to the whole state: For any state $S = (V, H, O)$ (used with P), its *unboxed state*, written as $unbox(S) = (V', H', O')$ is obtained by unboxing every opaque value in S using the unbox functions, *i.e.*, for any initialized opaque cell $c \in O$, we call the corresponding unbox function based on c 's type, the resulting value is put into the public heap H' , and any reference to c in the old state S is replaced by a reference to the new value in S' ; and we do the similar thing for any opaque value in V ; the new opaque heap O' is empty because every opaque value is unboxed into the public heap.

Similarly, we lift the box function to work on states. The box and unbox functions over states is defined for each package. Hereafter, we will use $box_1 / unbox_1$ to denote the box / unbox function over states using package $P1$, and $box_2 / unbox_2$ to denote the box / unbox function over states using package $P2$.

Having defined that, we can prove Lemma 7.3.5 below, which implies Theorem 7.3.3.

Lemma 7.3.5. For two packages $P1$ and $P2$ with the same interface satisfying Condition 7.3.1, and all the checks in Procedure 7.3.2 pass, then for a starting state S_0 without any opaque value, and any trace t of length N , if we apply $t@P1$ to S_0 and get S_1 , and apply $t@P2$ to S_0 and get S_2 , then $unbox_1(S_1)$ and $unbox_2(S_2)$ are also the same.

Proof. The proof is by induction on N , the length of the traces under consideration.

Base case: when $N = 1$, the trace has only one action a , and when a is not a call to a public function of the interface, this lemma obviously holds; when a is a call to a public function f , the check in Procedure 7.3.2 implies this lemma.

Inductive case: suppose Lemma 7.3.5 is correct for trace length $N = k$, now we prove it is correct for trace length $N = k + 1$. A trace t of length $k + 1$ is composed of a trace t' of length k followed by an additional action a . For the same starting state S_0 , suppose applying $t'@P1$ gets T_1 , and applying $t'@P2$ gets T_2 ; then we know that applying $a@P1$ to T_1 gets S_1 , and applying $a@P2$ to T_2 gets S_2 . From the inductive hypothesis, $unbox_1(T_1)$ and $unbox_2(T_2)$ are the same, so the public parts of T_1 and T_2 are the same.

When a is not a call to a public function of the interface, it is trivial to show that S_1 and S_2 are still the same, because a must only modify the public parts of the states in the same way.

When a is a call to a public function f of the interface, let

$$X_1 = unbox_1(S_1) = unbox_1(a@P1(T_1))$$

and

$$X_2 = unbox_2(S_2) = unbox_2(a@P2(T_2))$$

We want to prove that X_1 and X_2 are the same. First, because Condition 7.3.1 requires all box functions to be surjective, we can always find some Y_1 without opaque values such that $T_1 = box_1(Y_1)$. Let $Z_1 = unbox_1(T_1) = unbox_1(box_1(Y_1))$ (note that Z_1 is not necessarily the same as Y_1 , because unbox and box functions are not guaranteed to be inverse of each other), $Z_2 = unbox_2(T_2)$. From the inductive hypothesis, Z_1 and Z_2 must be the same.

Now let $T_3 = box_2(Y_1)$ (note that this is possible because Y_1 has no opaque values), $Z_3 = unbox_2(T_3) = unbox_2(box_2(Y_1))$. Then because of the check in Procedure 7.3.2(b), Z_3 and Z_1 must be the same, which is in turn, the same as Z_2 . So we get $unbox_2(T_2) = unbox_2(T_3)$, which means that T_2 and T_3 must be the same, because Condition 7.3.1

requires all unbox functions to be injective. That means $X_2 = \text{unbox}_2(a@P2(T_2))$ must be the same as $\text{unbox}_2(a@P2(T_3))$, which is just $\text{unbox}_2(a@P2(\text{box}_2(Y_1)))$. From the check in Procedure 7.3.2(a), we know that this is the same as $\text{unbox}_1(a@P1(\text{box}_1(Y_1)))$, which is just X_1 . So we have proven that X_1 is the same as X_2 , completing our inductive case. □

Note that Lemma 7.3.5 implies Theorem 7.3.3, so we have proven that the package refinement check can be done separately for individual functions.

7.4 SPMD programming with packages

MSL allows a functions f to be tagged with `spmd`, including functions inside a package. This tells the compiler that the execution of f should be done by parallel processes. MSL also allows a data type τ to be tagged with `spmd`, to tell the compiler that such data should be distributed to parallel processes, and can only be used by functions tagged with `spmd`. An example package with `spmd` data and functions is shown below:

```
package P1 {
  opaque spmd struct Vector {
    int len;
    int[len] arr;
  }

  public spmd Vector newVec(int N) { ... }
  public spmd Vector sum(Vector a, Vector b) { ... }
  public spmd int norm(Vector v) { ... }

  public struct PubVec {
    int N;
    int[N] arr;
  }

  box spmd void bf1(PubVec u, ref Vector v) { ... }
  unbox spmd void ubf1(ref PubVec u, Vector v) { ... }
}
```

Package `P1` shows the high-level idea of a distributed vector implementation: `Vector` is an opaque data type which is distributed to parallel processes. The user of the package cannot manipulate `Vector` directly, but have to call the public functions, which are all tagged with `spmd`. In addition to that, a public type `PubVec` is defined to enable

expressing the box and unbox functions for `Vector`. `PubVec` is *not* tagged with `spmd`, so it is a normal non-partitioned data type, which can be used by normal functions. The box function `bf1` takes a `PubVec` and produces one `Vector` per process (note that `bf1` is tagged with `spmd`, so it is executed by parallel processes); the unbox function `ubf1` takes one `Vector` per process, and updates a global `PubVec` based on the partitioned data. We can see that in the SPMD setting, box / unbox functions specialize into the *distribute / collect* functions we have discussed earlier. To see how the checking works, we define another package below:

```
package P2 {
  opaque struct Vector {
    int len;
    int[len] arr;
  }

  public Vector newVec(int N) { ... }
  public Vector sum(Vector a, Vector b) { ... }
  public int norm(Vector v) { ... }

  public struct PubVec {
    int N;
    int[N] arr;
  }

  box void bf2(PubVec u, ref Vector v) { ... }
  unbox void ubf2(ref PubVec u, Vector v) { ... }
}
```

P1 implements P2;

Package P2 has no `spmd` components: all its data types and functions are sequential. Following Procedure 7.3.2, we need to check each individual public function. Let's take `sum` as an example: First we construct a new function `sum2` as below:

```
PubVec sum2(PubVec a, PubVec b) {
  PubVec c;

  Vector@P2 x, y;
  bf2@P2(a, x);
  bf2@P2(b, y);

  Vector@P2 z = sum@P2(x, y);

  ubf2@P2(c, z);

  return c;
}
```

Then we construct a new function `sum1` in a similar way, but this time the functions in `P1` need to be executed by parallel processes, thus they are put inside an `spmdfork`:

```
PubVec sum1(PubVec a, PubVec b) {
  PubVec c;

  spmdfork {
    Vector@P1 x, y;
    bf1@P1(a, x);
    bf1@P1(b, y);

    Vector@P1 z = sum@P1(x, y);

    ubf1@P1(c, z);
  }
  return c;
}
```

Finally we can check whether `sum1` and `sum2` are functionally equivalent using the mechanisms discussed in Chapter 5.

From the example we can see that the issue of different data representations and the issue of different execution modes are handled orthogonally in MSL: The `spmd` keyword takes care of the parallel execution mode, and as we have seen in Chapter 4, every `spmd` function is deterministic, thus from the caller's point of view, an `spmd` function can be treated the same way as a normal function (modulo a wrapper `spmdfork`). This determinism also lends itself to our package refinement checking procedure, because it requires all actions to be deterministic.

It is important to note that the `box / unbox` functions are only used for analysis (just as `distribute / collect` functions). At runtime, only the actual kernels are executed, thus no additional overhead is imposed by our packaging system. Theorem 7.3.3 guarantees that the the SPMD package will generate the same observable effects as the sequential package, given that they are checked for refinement relation using Procedure 7.3.2.

Chapter 8

Case studies

We have implemented three distributed kernels from the NAS parallel benchmarks using MSL: NAS MG (multigrid), which exercises both short and long distance regular communication, a 3D matrix transpose library which uses all-to-all communication, and a Sparse Matrix-Vector multiplication (SPMV) kernel which uses irregular long distance communication and reductions; the latter two are essential components of the FT and CG benchmarks.

In all three kernels, to make the time spent in synthesis tractable (as discussed in Section 5.3), the synthesizer restricts all integer inputs to be 5 bits, and all floats (both input and intermediate float values) to be simulated by elements from a finite field of size 7. While these may seem small, checking even a simple algorithm that takes a single 1-dimensional array as input under this setting involves exhaustively checking about 10^{28} distinct inputs, which is much more than one can test with traditional mechanisms, leading to high confidence in the correctness of the synthesized code. The generated C++ code does not restrict integers to be 5 bits. Note that one effect of this approach is that the problem sizes and numbers of processes are not hard-coded at compile time, unlike the Fortran reference implementations.

In this chapter, we describe how MSL makes implementing these benchmarks easier, by assisting the programmer in writing difficult code and by enabling that code to be reused easily without performance loss.

8.1 3D Transpose from FT

The NAS FT kernel finds the solution to a partial differential equation using Fast Fourier Transforms (FFTs), and much of the time is spent redistributing data among the processes using transposes of different kinds. In Section ?? we demonstrated how MSL allows us to write a generic distributed transpose kernel that adapts itself to different specifications. Suppose that we want to implement transpose under a 2D process grid partitioning: there are $N \times M$ processes and the matrix is partitioned along its slowest growing dimension into N portions and then partitioned along its second slowest growing dimension into M portions. To implement a distributed transpose kernel under this new partitioning scheme, we only need to change `dtrans` slightly, and `pack` and `unpack` need not change at all. The modified code is shown below.

```
void dtrans(int nx, int ny, int nz, int N, int M,
            double[nz/N, ny/M, nx] LA,
            ref double[nx/N, ny/M, nz] LB) {
    int bufsz = (nx/N)*(ny/M)*(nz/N);
    view LA as double[N, bufsz] abuf;
    view LB as double[N, bufsz] bbuf;

    CommSplit c(mypid/N, mypid%N);
    in_group(c) {
        pack(LA, bbuf);
        All_to_all(bbuf, abuf); // re-distribute
        unpack(nx, ny, nz, abuf, LB);
    }
}
```

The main change is that now processes are partitioned to different groups and transpose is independently done by each group. There are some minor changes to `tester` and `distribute/collect`, all very natural and not shown here. `pack` and `unpack` need not be changed at all, and the generators inside them adapt to match the new specification.

8.2 The MG Benchmark

The NAS MG benchmark performs a V-cycle multigrid to solve a discrete Poisson problem on a 3D grid with periodic boundary conditions using several 27-point stencils. The 3D grid is partitioned in all three dimensions over the processes, and, during the

communication phase, each process must communicate with its logical neighbors in the grid to exchange data. Each neighbor is sent a different portion of the local grid, and therefore up to 12 functions are used in the exchange (6 for packing and sending in each direction, and 6 for unpacking). MSL simplifies this by enabling us to write a single packing function and a single unpacking function that can automatically be specialized for each case with no performance degradation:

```

gen void pack(int nx, int ny, int nz, double[nx, ny, nz] u,
  ref int len, ref double[MAXLEN] buf) {
  gen int p() {
    return { | ?? | ({| nx | ny | nz |} -??) |};
  }
  len = 0;
  for (int i = p(); i < p(); i++) // -X: 0, nz; +Z: 0, ny
    for (int j = p(); j < p(); j++) // -X: 0, ny; +Z: 0, nx
      buf[len++] =
        u[{| i | j | p()|}, {| i | j | p() |}, {| i | j | p()|}];
    // -X: [i, j, 1]; +Z: [nz-2, i, j]
}

```

`pack` uses generators and choice expressions to give the synthesizer freedom to instantiate the correct combination for each communication, resulting in different instantiations for each. The comment shows two such concretizations, one for the neighbor along the X axis in the negative direction, and one for the neighbor along the Z axis in the positive direction. Thus, we can reduce six functions spanning 50 lines of Fortran into 9 lines of MSL expressing a single function.

MSL also helps implement optimizations that are tedious or fragile to implement by hand. Prior work on 27-point stencils has shown that compilers do not effectively apply common subexpression elimination (CSE) to these kernels [23], and by-hand CSE is necessary for obtaining optimal performance. Though the result lacks bit-level accuracy with the original implementation, the CSE-enabled code generally does not drastically alter numerics.

The `psinv` kernel in MG, for the input 3D array $r_{x,y,z}$ and a kernel smoother $c[3]$, computes a 3D array

$$u_{x,y,z} = \sum_{d=|x'-x|+|y'-y|+|z'-z|<3} c[d] * r_{x',y',z'}$$

For the `psinv` kernel, instead of performing CSE ourselves, we can implement the sketch of an idea that MSL can synthesize based on finding equivalence to the original kernel. The basic idea is that the programmer has some intuition about which points are repeated, and decides to try precomputing some of them into a temporary array. With the assistance of the MSL synthesizer, the difficult index calculations are taken care of:

```

for (int k=??; k < nz-??; k++) // 1, nz-1
  for (int j=??; j < ny-??; j++) { // 1, ny-1
    gen int jk() { return { | j | k | } + { | ?? | -?? | }; }
    for (int i = ??; i < nx-??; i++) { // 0, nx-0
      r1[i] = 0; minrepeat { r1[i] += r[jk(), jk(), i]; }
      // r1[i] = r[k, j-1, i]+r[k, j+1, i]+r[k-1, j, i]+r[k+1, j, i]
      r2[i] = 0; minrepeat { r2[i] += r[jk(), jk(), i]; }
      // r2[i] = r[k-1, j-1, i]+r[k-1, j+1, i]+r[k+1, j-1, i]+r[k+1, j+1, i]
    }
    for (int i = ??; i < nx-??; i++) { // 1, nx-1
      gen int ii() { return { | i + { | ?? | -?? | } | }; }
      double t1 = 0, t2 = 0;
      minrepeat { t1 += { | r[jk(), jk(), ii()] | r1[ii()] | r2[ii()] | }; }
      // t1 = r[k, j, i-1]+r[k, j, i+1]+r1[i]
      minrepeat { t2 += { | r[jk(), jk(), ii()] | r1[ii()] | r2[ii()] | }; }
      // t2 = r2[i]+r1[i-1]+r1[i+1]
      u[k, j, i] = c[0]*r[k, j, i] + c[1]*t1 + c[2]*t2;
    }
  } // comments above show synthesized code, inferred by synthesizer

```

In the above code, `minrepeat` is another language construct that repeats its body an unknown number of times, and the synthesizer tries to find the minimum time that makes the program correct. The comments illustrate the difficult tedious calculations that the programmer would otherwise need to do by hand, if not for MSL.

For the MG benchmark, MSL assists in writing difficult code, but also helps reduce the overall complexity by abstracting multiple functions into a single function. Thanks to the synthesis occurring on a call-site-dependent basis, there is no loss of performance due to this abstraction.

8.3 Specialized Reduction for SpMV from CG

The NAS CG benchmark implements the conjugate gradient algorithm on a symmetric sparse matrix that is distributed across the processes using a 2D distribution. The

computation at the heart of the algorithm is a sparse matrix-vector multiply, or SpMV: given a sparse matrix M and a vector v , compute the vector $u = M \times v$. The dense sequential implementation of this is quite trivial, but adding a 2D partitioning as well as using a sparse format introduces complications to the algorithm.

The algorithm can be understood as operating in three phases after the input vector has been divided *per-column* (*i.e.* processes with the same column id have the same portion of v):

1. A local submatrix-subvector multiplication: each process calculates some local portion of u .
2. Allreduce-like reduction: processes in the same row add their local portion of u together and the point-wise sum is distributed to all processes in the same row.
3. Replicated vector transpose: processes exchange their portion of u with peers so that the per-column partitioning is recovered for u .

All three stages are tricky and MSL eases development: for local multiplication, MSL helps check that SpMV behaves equivalently to dense matrix-vector multiplication by using the dense version as a specification for the sparse version, and for the latter two phases, MSL assists in implementing customized exchange plans consisting of individual point-to-point communications, which outperform standard MPI collective routines by taking advantage of characteristics of the partitioning [24]. In the case of CG, these point-to-point communications build a tree to perform the all-to-all operation. Below is a portion of the code that synthesizes the custom reduction plan:

```
void setup_reduce_plan(int log2m, int[log2m] reduce_peer) {
    // log2m = log_2(m), m is the number of processes in a row
    int d = 1;
    for (int i=0; i<log2m; i++, d*=2)
        reduce_peer[i] = expr({d, mypid, m}, {PLUS, TIMES, DIV, MOD});
    // (mypid+d)%(d*2)+mypid/(d*2)*(d*2)+mypid/m*m
}
```

In the above code, `expr()` is a library generator in MSL that takes some operands and some operators and generates an expression constructed from those building blocks. The programmer knows that there should be $\log_2 m$ phases and that m is always divisible by two due to the partitioning. She knows that during each phase of the exchange, each process should exchange with a different process, and that at each

phase, the level of the communication tree increases. Putting these insights into an MSL program allows her to rely on the synthesizer to find the exact details of the addressing.

8.4 Time cost of synthesis step

The table below shows the time spent on the synthesis step:

| Benchmark | # generator instances | Synthesis Time |
|-----------|-----------------------|-----------------------|
| Transpose | 15 | 9 minutes 54 seconds |
| SPMV | 9 | 14 minutes 31 seconds |
| MG | 60 | 35 minutes 26 seconds |

8.5 Performance of Case Studies

We evaluate the efficiency of the code generated from MSL by comparing performance of a port to MSL against the hand-written official Fortran+MPI implementations. The ported code makes liberal use of synthesis to ease programming, as described above. Note that our goal here is not to demonstrate performance improvements, but rather to show that despite the restricted programming model, using MSL results in code of similar performance as well-written distributed code. For each benchmark, we run both MSL and Fortran implementations at different scales of parallelism (varying from 256, the minimum due to memory requirements, to 16384 cores), on the NERSC Hopper machine (a Cray XE6 with 2 AMD 12-core MagnyCours 2.1GHz processors per node). We use 16 cores per node for our experiments, and compile the code using Intel's compiler suite because it has excellent performance for both Fortran and C++ on this machine. For each configuration we run 7 times and report the min, median, and max in Figure 8-1. When possible, we use the standard NAS Class E problem parameters; for the SpMV, the problem becomes too small to run at larger concurrencies, so we scale up the matrix after $P = 2048$, with the parameters described in Figure 8-1. For the transpose in a 1D decomposition, we did not scale higher than $P = 2048$ because

of restrictions due to the problem size.

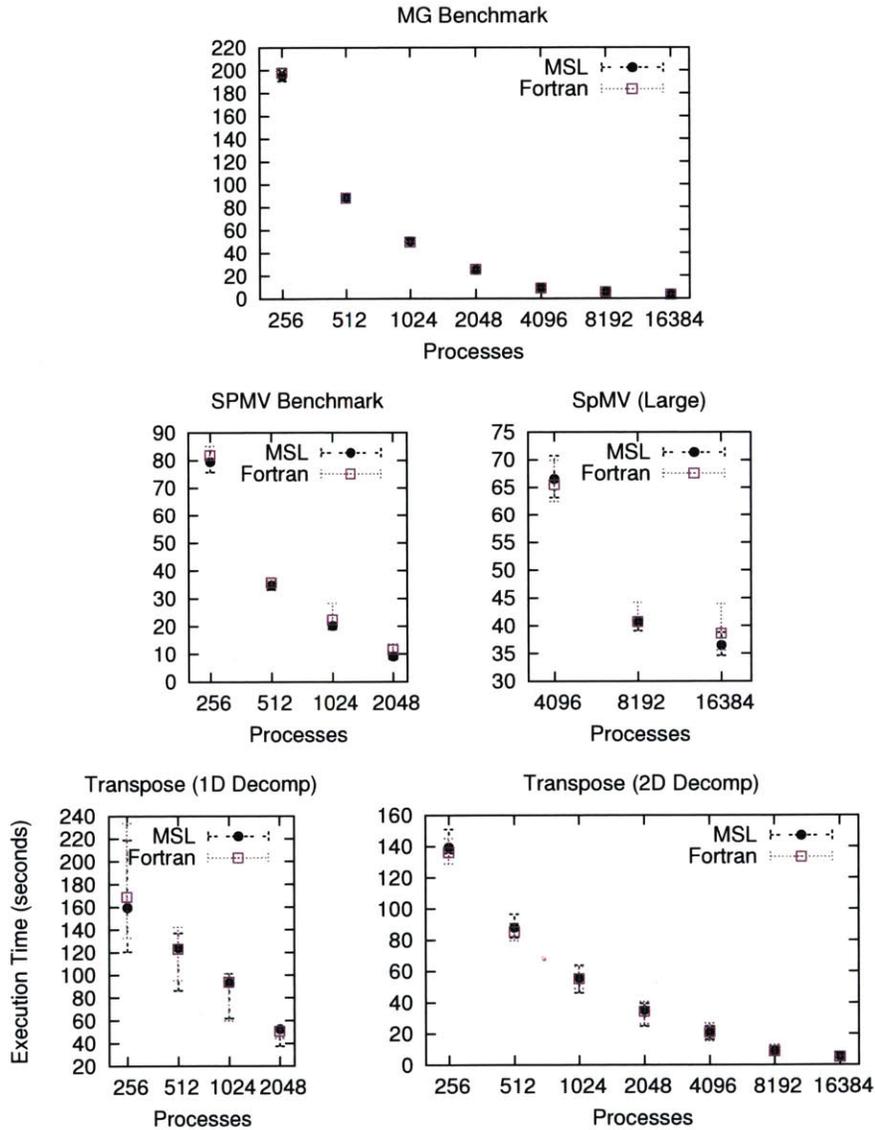


Figure 8-1: Performance of our three benchmarks from NAS. For the transpose from FT, we show performance using both a 1D and a 2D domain decomposition. For SpMV, we increase the matrix size for $P = 4096$ and larger to $4e7 \times 4e7$ and 32 nonzeros per row, running for 100 iterations.

For all experiments, the performance of MSL-generated code is within $\pm 5\%$ of the official implementation, demonstrating that programmers can take advantage of synthesis features without giving up performance. Our code generation strategy effectively eliminates possible sources for slowdown due to the restricted memory model. Overall, we see that MSL assists programmers by helping them write difficult

code and by enabling higher level generalizations of code that would otherwise need to be replicated with slight modifications, and that these features do not come with large performance costs.

Chapter 9

Conclusion

This thesis presents MSL, a new language to aid in the development of high-performance kernels on distributed memory machines. Combining generative programming and synthesis, MSL simplifies the development process and allows programmers to package complex implementation strategies behind clean high-level reusable abstractions. We have shown that MSL can automatically infer many challenging details and allow for high-level implementation ideas to be reused easily, and the generated code obtains the same performance as hand-written Fortran reference implementations.

Bibliography

- [1] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.
- [2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.
- [3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. In *The International Journal of Supercomputer Applications*, 1991.
- [4] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for gpu kernels. In *CAV*, 2014.
- [5] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319 – 357, 1976.
- [6] Gordon Bell. Supercomputers: The amazing race (a history of supercomputing, 1960-2020). Technical report, San Francisco, CA, 2014.
- [7] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.
- [8] Dan Bonachea. GASNet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.
- [9] Uday Bondhugula, J. Ramanujam, and et al. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI 08*, 2008.
- [10] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

- [11] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. *Concurrent Library Correctness on the TSO Memory Model*, pages 87–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [12] Eugene Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. SCALA '13, 2013.
- [13] B. Carpenter. Adlib: A distributed array library to support hpf translation. In *CPC*, 1995.
- [14] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM.
- [15] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3), 2007.
- [16] Bradford Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, 2001.
- [17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 2005.
- [18] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.
- [19] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, 2011.
- [20] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.
- [21] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 493–501, New York, NY, USA, 1993. ACM.
- [22] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

- [23] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Auto-tuning the 27-point stencil for multicore. In *iWAPT2009*, 2009.
- [24] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *PPoPP*, 2003.
- [25] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 5:1–5:12, New York, NY, USA, 2016. ACM.
- [26] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of mpi programs with intel® message checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.
- [27] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. In *PLDI*, 2013.
- [28] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. *SC Conference*, 2011.
- [29] Jack Dongarra. Trends in high-performance computing. *IEEE Circuits & Devices Magazine*, January/February 2006.
- [30] Yuri Dotsenko. *Expressiveness, Programmability and Portable High Performance of Global Address Space Languages*. PhD thesis, Rice University, Houston, TX, USA, 2007. AAI3340461.
- [31] A. I. El-Nashar and N. Masaki. To parallelize or not to parallelize, bugs issue. *International Journal on Intelligent Cooperative Information Systems (IJICIS)*, 10(2), July 2010.
- [32] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. *The Spirit of Ghost Code*, pages 1–16. Springer International Publishing, Cham, 2014.
- [33] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
- [34] High Performance Fortran Forum. High performance fortran. [Online; accessed 20-June-2016].
- [35] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

- [36] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *IEEE*, 93(2), February 2005. Invited paper, special issue on “Program Generation, Optimization, and Platform Adaptation”.
- [37] Rajat P. Garg and Ilya Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall Professional Technical Reference, 2002.
- [38] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of mpi-based parallel programs. *Commun. ACM*.
- [39] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *PACT*, 1998.
- [40] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A performance model for x10 applications: What’s going on under the hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop, X10 ’11*, pages 1:1–1:8, New York, NY, USA, 2011. ACM.
- [41] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.
- [42] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3), September 2005.
- [43] Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual. Technical report, EECS Department, University of California, Berkeley, 2005.
- [44] Forrest Hoffman. Message passing for master/slave programs. *Linux Magazine*, March 2003.
- [45] Daniel Jackson and Srinivas Devadas. Lecture note 6 — laboratory in software engineering, 2005. [Online; accessed 24-August-2016].
- [46] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.
- [47] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. Automatic vectorization of tree traversals. In *PACT*, 2013.
- [48] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

- [49] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Supercomputing*, 2012.
- [50] JuliaLang. Julia manual — parallel computing, 2016. [Online; accessed 25-April-2016].
- [51] JuliaParallel. Distributedarrays.jl — distributed arrays for julia, 2016. [Online; accessed 25-April-2016].
- [52] Amir Kamil and Katherine Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 185–199, Berlin, Heidelberg, 2006. Springer-Verlag.
- [53] Amir Kamil and Katherine Yelick. Enforcing textual alignment of collectives using dynamic checks. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 368–382, Berlin, Heidelberg, 2010. Springer-Verlag.
- [54] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS*, 2010.
- [55] Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.
- [56] Sam Kamin, María Jesús Garzarán, Baris Aktemur, Danqing Xu, Buse Yilmaz, and Zhongbo Chen. Optimization by runtime specialization for sparse matrix-vector multiplication. In *GPCE*, 2014.
- [57] Rainer Keller, Shiqing Fan, and Michael Resch. Memory debugging of MPI-parallel Applications in Open MPI. In G.R. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bucker, P. Gibbon, and B. Mohr, editors, *Proceedings of ParCo'07*, Julich, Germany, Sep 2007.
- [58] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, March 2016.
- [59] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I.W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A Language for Physical Simulation. Technical report, Massachusetts Institute of Technology, May. 2015.
- [60] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, June 2014.

- [61] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *PLDI*, 2013.
- [62] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. JavaScript as an Embedded DSL. In James Noble, editor, *ECOOP 2012 and Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 409–434, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [63] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 389–391, 2014.
- [64] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. In *FMCAD*, 2010.
- [65] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, September 2011.
- [66] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *FSE*, 2010.
- [67] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.
- [68] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Comm. ACM*, 1975.
- [69] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.
- [70] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.
- [71] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pages 135–146, New York, NY, USA, 2010. ACM.
- [72] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *ACM*

SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, June 2015.

- [73] Eric Niebler. Proto: A compiler construction toolkit for dsels. In *Symposium on Library-Centric Software Design*, LCS D '07, 2007.
- [74] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in scala: Towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 125–134, 2013.
- [75] The Apache Project. Apache Giraph.
- [76] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *IJHPCA*, 18(1), 2004.
- [77] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, June 2013.
- [78] Harvey Richardson. High performance fortran: history, overview and current developments. Technical report, 1.4 TMC-261, Thinking Machines Corporation, 1996.
- [79] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.
- [80] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of MPI-based parallel programs. In Calin Cascaval and Pen-Chung Yew, editors, *PPoPP '11*. ACM, 2011.
- [81] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE*, pages 59–70, 1998.
- [82] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.
- [83] Armando Solar-Lezama. The sketch programming manual.
- [84] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.
- [85] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

- [86] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI*, 2008.
- [87] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
- [88] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.
- [89] Olivier Tardieu. APGAS programming in x10, 2016. [Online; accessed 18-June-2016].
- [90] Emina Torlak and Ras Bodik. Growing solver-aided languages with rosette. In *Onward*, 2013.
- [91] Emina Torlak and Ras Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.
- [92] UPC Consortium. Upc language specification, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
- [93] Viktor Vafeiadis. *RGSep Action Inference*, pages 345–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [94] Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, 2008.
- [95] Todd Veldhuizen. C++ gems. chapter Expression templates, pages 475–487. 1996.
- [96] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC '00*, Washington, DC, USA, 2000. IEEE Computer Society.
- [97] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Scientific Discovery through Advanced Computing Conference, Journal of Physics: Conference Series*, 2005.
- [98] Chao Wang, Yi Lv, Gaoang Liu, and Peng Wu. *Quasi-Linearizability is Undecidable*, pages 369–386. Springer International Publishing, Cham, 2015.
- [99] Chao Wang, Yi Lv, and Peng Wu. *TSO-to-TSO Linearizability Is Undecidable*, pages 309–325. Springer International Publishing, Cham, 2015.
- [100] Richard Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing*, 1998.
- [101] Wikipedia. Fork-join model — Wikipedia, the free encyclopedia, 2016. [Online; accessed 1-August-2016].

- [102] Wikipedia. History of supercomputing — Wikipedia, the free encyclopedia, 2016. [Online; accessed 25-April-2016].
- [103] Wikipedia. Thread pool — Wikipedia, the free encyclopedia, 2016. [Online; accessed 1-August-2016].
- [104] Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. Msl: A synthesis enabled language for distributed implementations. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.
- [105] Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. *A Structural Approach to Prophecy Variables*, pages 61–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.