Final Report on XStack: Software Synthesis for High Productivity ExaScale Computing

Principal Investigator:
Prof. Armando Solar-Lezama
Postal Address: The Stata Center, Building 32-G840, 32 Vassar Street, Cambridge, MA 02139
Telephone Number: (617) 258-9727
Email: asolar@csail.mit.edu

Partnering Institution: University of California at Berkeley (UCB)

The goal of the project was to develop a programming model that would significantly improve productivity in the high-performance computing domain by bringing together three components: a) Automated equivalence checking, b) Sketch-based program synthesis, and c) Autotuning. In more detail, we are pursuing the following efforts:

- Bringing into HPC programming the recent results from automated model-checking, specifically program equivalence checking, which we use to ascertain the semantic equivalence of an executable specification and a high-performance implementation, eliminating bugs due to performance optimizations.
- We are advancing the power of our synthesizer that works outside domains amendable to classical synthesis such as FFTW. The Sketch synthesizer, which can be guided by programmers without formal training, is used to implement high-performance implementations.
- We are developing a refinement programming methodology, wherein a high-performance implementation is obtained by successive optimizations of a specification. These refinements are checked for correctness with equivalence checking and parts of these optimizations are synthesized.

The report provides an executive summary of the research accomplished through this project. At the end of the report is appended a paper that describes in more detail the key technical accomplishments from this project, and which was published in SC 2014 [1].

Summary

Over the three years of our project, we accomplished the following milestones:

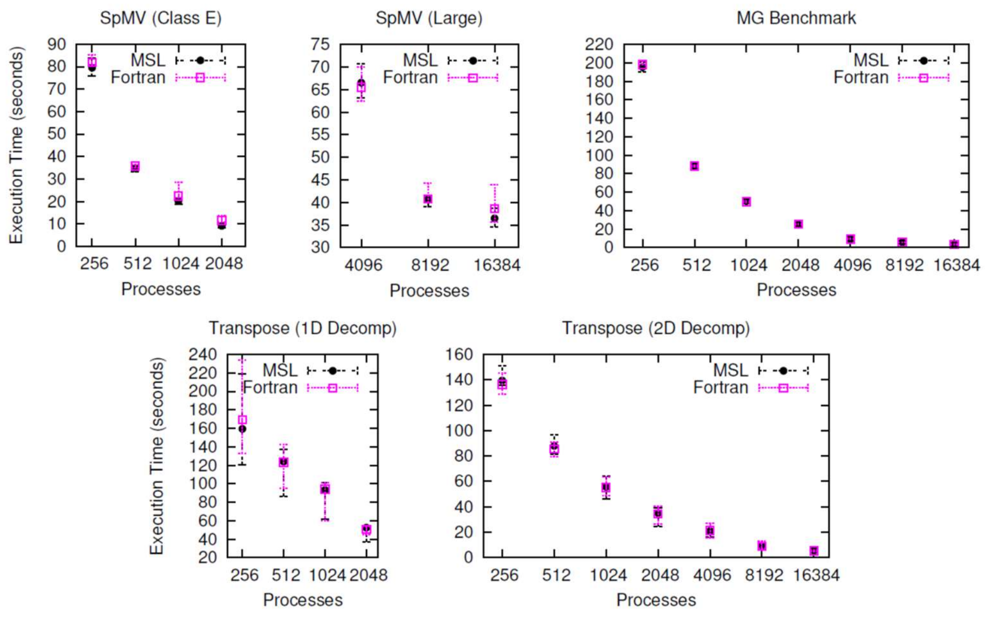
- We developed a set of algorithm to support synthesizing SPMD implementations on top of the Sketch synthesis system.
- We developed a set of high-level programming constructs that are easier to analyze compared to low-level MPI, but are still sufficiently expressive to support implementation of efficient and scalable implementations of distributed algorithms.
- We developed a new set of synthesis constructs to make it easier to reuse insights from one problem in implementing similar problems.
- We implemented a set of kernels from the NAS parallel benchmarks and showed that we could use high-level sketches to synthesize implementation details necessary to achieve performance comparable to Fortran implementations.
- We have developed a new algorithm for solving synthesis problems much more efficiently by parallelizing the search for a solution.

Sketch based Synthesis of MPI code

As part of this project, we developed a set of constructs to support synthesis of SPMD implementations. The new constructs enforce a bulk-synchronous programming model that is determinate by construction and that eliminates the need to match sends and receives. In this model, all communication happens through a set of primitives that must be called synchronously by all threads and which have barrier-like behavior. Reductions naturally have this property, but even point-to-point communication is expressed in this manner. The constraints on the communication primitives and the determinacy guarantees significantly simplify the analysis of programs written in terms of these constructs and make it possible to perform the deep semantic analysis necessary for synthesis. Our synthesizer exploits determinacy through a novel reduction that converts the SPMD synthesis problem into a sequential synthesis problem which can be solved by our off-the-shelf Sketch synthesis system.

One of our major accomplishments is to show that the new programming constructs together with the new reduction can allow us to scale synthesis to realistic computational kernels. Specifically, we have demonstrated our synthesis-based methodology through case studies implementing non-trivial distributed kernels---both regular and irregular---from the NAS parallel benchmarks. We show that our approach can automatically infer many challenging details from these benchmarks and can enable high-level implementation ideas to be reused between similar kernels. We also demonstrate that these high-level notations map easily to low-level C code and show that the performance of this generated code matches that of hand-written Fortran.

With regard to performance, we were able to show scalability up to 16384 cores on SpMV, MG and 3D transpose with 2D partitioning. In all cases, the performance matched the performance of hand-written Fortran. This showed that the higher level programming model and the resulting simplification of the analysis did not come at the expense of performance.



Improvements to the Solver Infrastructure

The Sketch synthesizer relies on a counterexample guided inductive synthesis solver (cegis) that takes synthesis problems in an intermediate representation and solves for their unknowns. Over the course of the project we made a number of changes to the underlying solver to improve its scalability and its ability to handle complex synthesis problems. Some of these changes are summarized below:

Support for arrays in the solver. The solver relies on bounded reasoning, so in order to reason about arrays, the frontend would put a bound on the maximum size of an array and treat it as a set of scalar variables. This caused a very significant blow-up on the size of the intermediate representation which would cause the solver to run out of memory on some benchmarks. By introducing arrays into the intermediate representation and adding algorithms for the solver to reason about them directly, we are able to get order of magnitude reductions in memory consumption from the cegis solver.

Abstraction refinement for recursive functions. The sketch compiler reasons about most array manipulation codes using an algorithm that translates them into systems of mutually recursive functions. In previous versions of the solver these were handled by inlining the functions a fixed number of times as directed by a compiler flag. We have implemented a new scheme that relies on abstraction refinement to automatically discover which functions need to be inlined and exactly by how much. This prevents the system from unnecessarily inlining those functions which do not require it.

Support for max in the solver. The transformation that transforms array codes into mutually recursive functions relies on solving for the maximum value i that satisfies a predicate $p(i)$ (i.e. $\max(i, p(i))$). A previous version of the algorithm used a simple algebraic manipulation routine to solve for the value of i whenever possible. When the algebraic manipulation failed, the system would fall back on an explicit search for i using a while loop. The huge performance difference between the algebraic solver and the explicit search meant that small changes in the sketch could lead to huge performance differences in the solver if the changes caused the algebraic solver to fail. Our new algorithm relies on the solver to find values of i by relying on a combination of algebraic reasoning and abstraction refinement.

Parallel synthesis. We have been able to significantly improve the scalability of the solver by parallelizing the search for a correct solution using a combination of solver-based synthesis and randomized search.

[1] "MSL: a Synthesis Enabled Language for Distributed Implementations", Zhilei Xu, Shoaib Kamil, Armando Solar-Lezama, *Published in SC 2014*

MSL: a Synthesis Enabled Language for Distributed Implementations

Zhilei Xu
MIT CSAIL
timxu@csail.mit.edu

Shoaib Kamil
MIT CSAIL
skamil@csail.mit.edu

Armando Solar-Lezama
MIT CSAIL
asolar@csail.mit.edu

***Abstract*—This paper demonstrates how ideas from generative programming and software synthesis can help support the development of bulk-synchronous distributed memory kernels. These ideas are realized in a new language called MSL, a C-like language that combines synthesis features with high level notations for array manipulation and bulk-synchronous parallelism to simplify the semantic analysis required for synthesis. The paper shows that by leveraging these high level notations, it is possible to scale the synthesis and automated bug-finding technologies that underlie MSL to realistic computational kernels. Specifically, we demonstrate the methodology through case studies implementing non-trivial distributed kernels—both regular and irregular—from the NAS parallel benchmarks. We show that our approach can automatically infer many challenging details from these benchmarks and can enable high level implementation ideas to be reused between similar kernels. We also demonstrate that these high level notations map easily to low level C code and show that the performance of this generated code matches that of hand-written Fortran.**

I. Introduction

This paper shows how new techniques from the areas of software synthesis [1], automated bug-finding [2] and generative programming [3, 4] can be combined to aid the development of high-performance computational kernels. 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 paper 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 NAS parallel benchmarks [5] 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 is not intended to develop full programs, but it 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 Fortran implementations. In the rest of this section, we provide a high level overview of MSL and the programming model that it supports.

A. *The* MSL *Programming Model*

We illustrate the MSL programming model through a simple example: transposing a three dimensional array. Below is a simple reference implementation of 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 $0 \le z < nz$, $0 \le y < ny$, $0 \le x < nx$. 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 one of its dimensions; specifically, given N processes, we partition the z dimension of array A, and the x dimension of the transposed array B. Because the dimension across which we partition is different for the two arrays, the transpose requires a re-distribution of data across the machine.

```
void dtrans(int nx, int ny, int nz, int N,
          double[nz/N, ny, nx] LA, ref double[nx/N, ny, nz] LB) {
  int bufsz = (nx/N)*ny*(nz/N);
  view LA as double[N, bufsz] abuf;
  view LB as double[N, 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 [5]. 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. **nprocs** is a pre-defined expression that refers to the total number of processes, which equals N in dtrans() here.

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

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;
  gen int num() {
    return {| an1 | an2 | an3 | bn1 | nprocs |};
  }
  gen 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;

    gen int num() {
      return {| an1 | bn1 | bn2 | bn3 | nprocs |};
    }
    gen 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()]
              // bn3/nprocs, bn2*bn3, bn3, 1
            = fin[p*dim() + i*dim() + j*dim() + k*dim()];
              // an1, bn2*bn3/nprocs, bn3/nprocs, 1
}
```

In order for the synthesizer to discover the details of these methods, the developer has to specify the relationship between the behaviors of trans() and dtrans(). To do this, the programmer must specify two things: the relationship between the computation performed by the two functions, and the relationship between the purely-local and distributed data used by each. Relating the computation is done by writing a test harness like the one 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 keyword **implements** states that tester() should perform equivalent computation to what trans() does, except that the tester works by distributing the initial arrays and then collecting the results back into the sequential one. **spmdfork** indicates that the computation is distributed to a set of parallel processes, each executing the same code. The equivalence is checked exhaustively for a bounded set of inputs satisfying assumptions at the beginning of the harness.

In addition to relating the computation, the programmer needs to provide distribute()/collect() functions that relate the distributed and sequential data layouts. The distribute()/collect() functions partition the global array ga along its slowest growing dimension. It is important to note that these functions will not be part of the kernel we are trying to synthesize; 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.

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 transpose in this section, the same generators can be reused to transpose any pair of dimensions, and can even be reused to transpose grids partitioned across two dimensions instead of one. Without synthesis, the two 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 [6]; or (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 implemetation of FT), which is also error-prone, and introduces runtime costs. 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.

B. Contributions

Overall, the paper 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 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 code, even when running across thousands of cores.

II. Key Language Features in MSL

The MSL language builds on previous work on synthesis enabled languages [7–9] and generative programming [10] to give the programmer the ability to create high level reusable abstractions with minimal runtime cost. The language also provides a clean mechanism to describe coordination and communication patterns for SPMD computation.

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 section, we describe the key design decisions that we made to navigate the tradeoff between programmability, performance and analyzability.

A. Basic Synthesis Features

The SKETCH language [7] first showed how to add synthesis capabilities to an imperative language by combining unknown integer constants with *generators*. Unknown integer constants are denoted by the token ??, which 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. Another construct is the choice expression (like {| a | b |}) which represents an unknown choice of its several branches.

Simple integer unknowns and choices can be used to define spaces of more complex program fragments by packaging them into *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. This property was used in the running example: because dim() was defined as a generator in unpack, different uses of it lead to different concretizations.

B. Array language

MSL provides a number of features to support the development of array-centric code. Arrays in MSL have copy semantics, impose aliasing restrictions when passed by reference, and must specify the size using an immutable expression. A novel feature in MSL is support for dependent array types: for any integer expression n and type T, one can use the notation T[n] as the type of arrays with n elements of type T. Multidimensional arrays are supported, but unlike languages like Java, sizes of all dimensions must be specified, resulting in more efficient, singly-indirect accesses: any multidimensional array is transformed into a (row-major) 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 check for array bounds. Note that in MSL all the assertions are checked at compile time by a bounded model checker and do not impose runtime costs.

In general, checking that all array accesses conform to the bounds declared in their type is undecidable. Nevertheless, the MSL compiler performs bounded correctness checking to identify out-of-bound accesses. This requirement of avoiding out-of-bounds accesses also serves as a lightweight specification when synthesizing details of code; in the case of the introductory example, the constraints on out-of-bound accesses played an important role in ruling out many incorrect choices for index expressions and loop iteration bounds.

MSL includes a number of syntactic conveniences around dependent-typed arrays; for example, it is common to pass a dependent-typed array and its size together to a function. MSL supports *implicit parameters* for succintness: when an **int** parameter is used as the size of a later array parameter, it can be declared as *implicit*, just as the All_to_all and pack functions in Section I-A. Also, it is convenient in many 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). 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 [11], and in distributed array libraries such as Adlib [12].

C. SPMD *programming in* MSL

MSL supports SPMD style programming, and allows programmers to relate the behavior of an SPMD implementation to that of a sequential one. In order to enable the analysis necessary to support synthesis for SPMD code, the language provides a restricted but expressive set of features for describing computation in a bulk-synchronous style. In this style of programming, algorithms iterate between a local computation phase and a communication/synchronization phase.

Communication is achieved through a set of functions that support all-to-all communication, reductions and point-to-point communication. We have seen in Section I the All_to_all function that sends data from and to all processes. It is an abstraction of the MPI_Alltoall function. MSL also 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 n−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_SUM and so on.

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 receipient. 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. Our system 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.

MSL imposes some constraints on how communication functions are used. The most important one is that all communication functions are designed to be called in bulk-synchronous fashion; in particular, when a communication function is called, the path condition cannot depend on the rank of the process—while in principle it is possible to write correct programs that violate this condition, it is generally considered good practice to avoid calling collective operations from branches that are taken by only some processes, which is why other SPMD languages such as Titanium also enforce this restriction [13]. In our case, all communication functions are considered collectives.

In the case of transfer, 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 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 SPMD nature of the program, ensure that all the messages will reach their destinations and that execution will be fully deterministic. Thus, after checking that all the conditions are satisfied, the synthesizer can reason about the parallel distributed memory program using the same tools that are used for reasoning about sequential programs, as demonstrated in Section III.

Some computations naturally can be decomposed into operations over subsets of the processes. MSL supports a restricted form of process hierarchy through process grouping. Groups are defined through *communicators* which are similar in spirit to MPI communicators but restricted to ensure analyzability. A communicator is created with a declaration of the form:

```
CommSplit c(g, i);
```

The declaration must be called collectively by all processes; each process will indicate what group g it wants to join and which id i it wants to have within that group. We enforce that two processes joining the same group cannot claim the same id, and that the set of ids and groups must be contiguous.

The in_group block takes as a parameter a communicator and defines a scope where that communicator is active.

```
CommSplit c(g, i);
in_group(c) {
  //communicator c is active here.
}
```

Within the in_group block, communication functions refer to the group-local process id, and All_to_all and reduce are done within the group, and expressions like **mypid** and nproc refer to their group-local meanings. This is useful when the programmer wants to divide the workload to a multidimensional process grid rather than a flat process row. Like CommSplit, in_group is a collective operation and must adhere to the restrictions on path conditions. Note also that every process becomes a member of some group, and that groups cannot overlap; similar restrictions are found in implementations of hierarchical communicators [14] and are useful for performance and analyzability while not presenting too onerous of a burden on the programmer.

D. *Establishing* SPMD*-sequential equivalence*

As described in the overview, the language provides a **spmdfork** construct that allows the programmer to describe how the SPMD computation relates to the sequential computation. All uses of **spmdfork** are required to have the same form as the code in the example: a call to distribute, followed by a call to the SPMD kernel, followed by a call to collect. distribute/collect are created by the user and together they define the distributed memory partitioning of data. For each piece of global data, distribute returns the partition of that data belonging to process **mypid**. collect takes the partition in process **mypid** and defines how the global data would be "updated" with the values from the partition. They are normal MSL functions, so the programmer can use generators to help derive more sophisticated partitioning

```
 v :=  x |L[t].x|G.x
 e :=  tid | v | n | e1 op e2
 s :=  v = e |assert e | s1;s2 | if (v) s1 else s2 |
   while(v){s1}| reduce(e,y) | transfer(sc,e,rid,rc,x)
 P :=  s_dist;fork{s};s_collect
```

Fig. 1: The simplified language $\mathsf{L}_{\mathsf{small}}$

schemes, for instance when the problem size is not evenly divided by the number of processes. When some details are left unspecified, the compiler will synthesize the missing details in order to let the programmer know exactly what mapping from sequential to distributed data it is assuming.

It is important to note that distribute/collect exist solely to allow the analysis engine to relate the distributed memory implementation to the original reference implementation; they do not actually perform any communication, and will not be part of the synthesized kernel. These functions enable the reasoning in the next section.

III. Reasoning About SPMD Programs

The system follows a three step process to generate efficient code from the MSL program given by the user. (1) First, a series of transformations is applied to the MSL program to create a "sequential equivalent": a sketch of a *sequential* program with unknowns and a guarantee that if the sequential program has its sketch filled in, the same details can be used to correctly implement the parallel program. (2) Taking the transformed program and its specification as input, a synthesis tool uses a counterexample guided inductive synthesis algorithm (CEGIS) to solve the sequential synthesis problem, as will be outlined in Section IV. (3) Finally, the code fragments derived from the sequential problem are filled in the original MSL program and a code generation phase performs low level optimizations such as overlapping communication with computation to produce efficient C++ code as described in Section V.

This section describes our approach to reduce SPMD programs in MSL to sequential programs in a way that captures SPMD semantics as well as the restrictions imposed by the different language constructs. More specifically, given a sequential reference implementation P_{seq}, the goal is to create a transformation that allows us to check the correctness of P_{spmd} by checking the sequential equivalence between the reduced SPMD program and P_{seq}. However, this does not mean that the semantics of MSL are sequential; the purpose of this reduction is purely for analyzability, to reduce the synthesis problem for SPMD into an analyzable sequential synthesis problem.

For space reasons, we will only outline the transformation and give a flavor as to why it works on a minimal language L_{small} that preserves the key features of MSL from the perspective of SPMD programming. Examining new features or analysis ideas in a minimal language is a standard approach that makes it possible to gain a better understanding of how those features will behave and interact with each other. In the remaining section we describe how programs in this small language can be transformed into semantically equivalent programs in a sequential language $L_{\mathrm{small}}^{\mathrm{seq}}$.

A. The L_{small} Language

The L_{small} language, shown in Figure 1, is a minimal imperative language with a few additional constructs to express bulk-synchronous SPMD computation. A program in L_{small} is a stylized version of the tester harness shown in the introduction which distributes the global state, computes in SPMD style on the distributed state and then collects the state back.

The **fork** statement spawns N processes each with a unique id t; all processes execute the body of the **fork** and **fork**s cannot be nested. Normally, we would expect statements inside **fork** to only be able to access their process-local state—because we are modeling distributed memory programs—and statements outside **fork** to be able to access only global state. However, distribute and collect need to be able to access both local and global state. For the purpose of formalization, we will assume without loss of generality that all statements before **fork** correspond to distribute and all statements after **fork** correspond to collect. Therefore, within **fork** all variables are assumed to be process-local and only process-local variables can be accessed. Outside the fork, the notation $L[t].x$ refers to process-local variable x of process t and $G.x$ refers to global variable x. In essence, we will be modeling a distributed language with a shared-memory language, while preserving the race-free characteristics of the distributed language. The purpose of this modeling is to eventually transform into a fully-serial language on which we can perform tractable synthesis.

L_{small} includes two communication primitives: reduce and transfer. reduce(e,y) evaluates an expression e at every process, adds all their values, and writes the sum to the output parameter y on every process. transfer(sc,e,rid,rc,x) sends and receives data from peer processes: rid is the process id of the receiver, sc is the send condition, e is data to be sent, rc is the receive condition, and the output parameter x stores received data.

The semantics of reduce and transfer are particularly important to our transformation. Recall that all communication operations in MSL are collective operations. The semantics of reduce and transfer are defined in terms of a channel $C[t]$ that is a temporary buffer for the communication operations. Thus, the semantics for reduce(e,y) can be specified as

```
reduce(e, y) {
  C[t] = e;
  barrier;
  L[t].y = sum(C[0:max(t)])
  barrier;
}
```

That is, reduce contains barriers that ensure no process executes past the reduction operation unless all processes have completed the reduction. This *barrier-like* property also applies to transfer, and is essential to our reduction to sequential semantics, since it ensures synchronization points for all processes after each communication operation.

B. SPMD*-to-Sequential Transformation*

The goal of the transformation process is to produce a sequential program with the following property: for any execution produced by the original program, the execution of the transformed program either produces the same final state, or causes an assertion failure if the original program failed to satisfy the rules of usage for communication primitives. The basic logic behind the transformation is simple; the goal is to sequentialize a parallel region by recursively breaking it into a sequential composition of smaller parallel regions until the parallel regions are simple enough to easily establish their equivalence to a sequential code fragment. The key for this strategy to work is to show that after every step of this process the set of possible outcomes of the execution remains unchanged—despite the fact that the set of possible interleavings has been reduced due to the introduction of sequential ordering. The formal correctness argument, which we will elide here, is based on Lipton's theory of left-mover and right-mover actions [15, 16]. Instead, we will cover a high level view of the transformations that demonstrates how they work.

Since the parallel region is contained in a **fork** statement, we can restrict our transformation to work solely on the body s of **fork**{s}. Then, the following five cases cover all the interesting possibilites for the transformation; the first two cases correspond to base cases for the transformation, while the latter three recursively sequentialize larger statements.

Private Statement If the body s of **fork**{s} does not invoke any communication functions, then it must operate only on local state. In this case, all interleaving orders are equivalent, so the transformation can choose any one of them; we choose the interleaving that executes one process at a time. As a trivial example, the statement **fork**{ x=y;} would be sequentialized into the code below; *i.e.* the local value of y on every process is copied to the local value of x following some sequential order.

```
forall t: L[t].x = L[t].y;
```

Transfer or Reduce Statement In the case of a reduction, we use the barrier-like behavior to define a straightforward translation to a sequential operation. Due to the synchronization statements, we can replace the collective **fork**{ reduce(e,y) } with the sequential statements

```
sum = 0;
forall t: sum += L[t].e;
forall t: L[t].y = sum;
```

Clearly, without the implicit barriers within the reduction, we would not be able to sequentialize the statement due to possible interleavings of operations. The argument if s corresponds to a transfer statement is similar to that of reduce; again, we use the collective nature of our communication operations to trivially reduce the operation into a sequentially-equivalent one.

Conditional Statement If s corresponds to an **if** statement that contains communication within one or both of its branches, then we must transform it to lift out the conditional from within **fork**. Our rules of usage require that the condition expression must evaluate to the same value on each process (since the path condition cannot depend on rank); this means we can consider this code equivalent to one which, after checking the condition, spawns processes to execute the appropriate branch. As long as the rest of the transformations preserve semantics, **fork**{**if** (v) s1 **else** s2} will be equivalent to **if** (v) **fork**{s1} **else** **fork**{s2}.

Loop In this case, the body must involve communication operations, otherwise the rule for a private statement would apply. As in the case for a conditional statement, the usage rules require the loop condition to always evaluate to the same value on every process , due to our restriction that path conditions cannot be rank-dependent. The case in which the number of iterations is zero is trivial. In the other case, we can transform **fork**{**while**(x) s} to **fork**{s; **while** (x) s}. Assuming that the final rule (for sequences of statements, which we will discuss below) is correct, we can further transform this into **fork**{s}; **fork**{**while**(x) s}. Recursively applying this will give a complete schedule that has turned into a sequence of **fork** statements. Thus the transformation schedules all processes to perform the first iteration before any can start the next iteration, and under our semantics, this is equivalent to any other schedule that can occur.

Statement Sequences Finally, we need to transform a statement of the form **fork**{s1; s2} into an equivalent sequential statement **fork**{s1}; **fork**{s2}. To prove this is correct, we must ensure that the transformation is semantics-preserving. Consider a trace t produced by **fork** {s1; s2}; we 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 can always swap a_2, a_1 to a_1, a_2, then we will have a strategy to convert t into a good trace t'. In the case that either of the two actions are local-only, this is trivial. If both contain communication, we can rely on the barrier-like behavior of the communication operations as well as the path condition restriction; it can never be the case that two communication operations belonging to two different statements are being executed at the same time. The transformation outlined above allows us to turn a synthesis problem involving parallelism into a semantically-equivalent serial synthesis problem.

IV. Constraint-Based Synthesis

After the transformations from Section III are applied, what is left is a synthesis problem over sequential programs. 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 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. The approach used for both of these problems builds on previous work on constraint-based synthesis [1, 7], so we only describe the algorithm at a high level here.

A. Constraint generation

We generate 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 a computation. The *source* nodes of the DAG (without incoming edges) are input values, including program inputs (corresponding to in) and control parameters (c, the unknowns 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 [2], 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) stmt_1; **if** (c) stmt_2;...; **if** (c) stmt_{LB}; **assert**(! c)}, where stmt_k 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 unknowns 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 [1], 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 and provides high confidence in the result 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.

B. *Solving for control parameters*

Our system searches for control parameters c^s such that $\forall in.Q(in, c^s)$. The counterexample guided inductive synthesis (CEGIS) approach 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 $\wedge_{in \in E_i} Q(in, c_i)$.

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.

C. *Correctness checking procedure*

The synthesizer uses a correctness checking procedure similar to SAT-based bounded model checking: 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 in.

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.

V. Code Generation

After the synthesizer discovers the details of the implementation, the last step is to generate an efficient C++/MPI implementation from the high level program. As discussed earlier, code generation relies heavily on inlining and partial evaluation of generators in order to produce code specialized to each individual use of a generator. The compiler also performs a set of standard optimizations such as unboxing of temporary structures and copy propagation.

The arrays in MSL have copy semantics, which means that when arrays are passed in as a parameter, semantically the callee gets a copy of the original array. To prevent excessive copying, arrrays 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.

Another important aspect of code generation is generating MPI code to implement the programming model. Most communication mechanisms in MSL can be translated to their MPI counterparts in a straightforward way, such as All_to_all and reduce. The function transfer is more interesting because the semantics require it to have barrier-like behavior, but that would make the resulting code too inefficient. Instead, we implement transfer using only the asynchronous MPI communications to do pairwise synchronization. Recall that the key reason for barrier-like behavior was to ensure that messages transmitted by statement s_1 could only be received by the same statement s_1 in another process, which ensured that program semantics would be unchanged as long as execution of s_1 on all process were scheduled before the execution of some subsequent statement s_2 on all processes (see Section III-B). We achieve the same goal by using a sequential tag for each message and ensuring each transfer receives only messages with the correct tag; this strategy is similar to that used in prior work on phasers [17]. Thus, transfer(scond, len, sendbuf, rid, rcond, recvbuf) translates to:

```
// msgtag is a global counter
int tag = ++msgtag;
MPI_Request req;
MPI_Status sta;
if (rcond) {
  MPI_Irecv(recvbuf, len, MPI_DOUBLE,
     MPI_ANY_SOURCE, tag, MPI_WORLD, &req);
}
if (scond) {
  MPI_Send(sendbuf, len, MPI_DOUBLE,
     MPI_ANY_SOURCE, tag, MPI_WORLD);
}
if (rcond) {
  MPI_Wait(&req, &sta);
}
```

The compiler also does automatic re-arrangement of the communication to gain some overlapping of computation and communication. To demonstrate this optimization, consider the example below, which is a simplified version of the communication in MG benchmark of NAS:

```
// double[n,m] grid;
// double[m] sbuf1, rbuf1, sbuf2, rbuf2;
sbuf1 = pack1(grid); // read grid, write sbuf1
sbuf2 = pack2(grid); // read grid, write sbuf2
transfer(sbuf1, rbuf1); // we omit other parameters for simplicity
transfer(sbuf2, rbuf2); // and assuming scond, rcond are both true
unpack(grid, rbuf1); // read rbuf1, write grid
unpack(grid, rbuf2); // read rbuf2, write grid
```

A straightforward translation will separate the computation (packing/unpacking) completely from the two stages of communication. The optimization tries to move MPI_Irecv and MPI_Send to as early as possible, and move MPI_Wait to as late as possible, while respecting the data dependencies: MPI_Irecv writes to recvbuf, so it cannot swap with any pre-communication read/write access to the receiving buffer, MPI_Send reads from sendbuf, so it cannot occur before any pre-communication write to the sending buffer, and MPI_Wait cannot swap with any post-communication read/write access to the receiving buffer. Using these simple rules, the above example is translated to:

```
int tag1, tag2;
MPI_Request req1, req2;
MPI_Status sta1, sta2;
tag1 = ++msgtag;
MPI_Irecv(rbuf1, tag1, &req1); // omit other parameters for simplicity
tag2 = ++msgtag;
MPI_Irecv(rbuf2, tag2, &req2);
sbuf1 = pack1(grid); // read grid, write sbuf1
MPI_Send(sbuf1, tag1);
sbuf2 = pack2(grid); // read grid, write sbuf2
MPI_Send(sbuf2, tag2);
MPI_Wait(&req1, &sta1);
unpack(grid, rbuf1); // read rbuf1, write grid
MPI_Wait(&req2, &sta2);
unpack(grid, rbuf2); // read rbuf2, write grid
```

Thus, for some sequences of communications, we can take advantage of overlapping communication with computation by moving the generated calls while still respecting dependencies.

VI. 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 IV-C), 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 section, 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.

A. 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 I 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 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.

B. 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 instatiations 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 [18], 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
```

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.

C. 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 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 [19]. 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.

D. 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

E. 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 2.

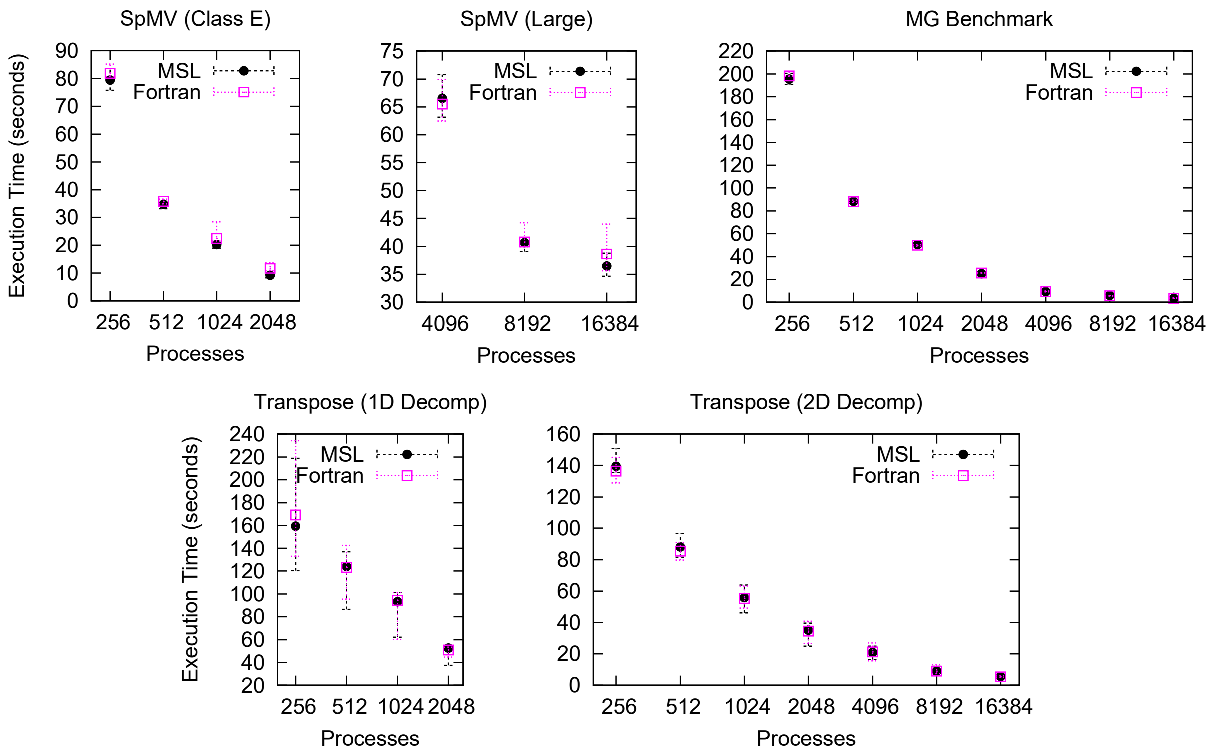


Fig. 2: Performance of our three benchmarks from the case studies. For the transpose from FT, we show performance using both a 1D and a 2D domain decomposition. For SpMV, we increase the matrix size for $P = 4096$ and larger to $4e7 \times 4e7$ and 32 nonzeros per row, running for 100 iterations.

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 2. For the transpose in a 1D decomposition, we did not scale higher than $P = 2048$ because of restrictions due to the problem size.

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.

VII. Related Work

Generative Programming Expression templates [3] are a metaprogramming technique in C++ which uses templates to create domain specific embedded languages (DSELs). 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 [4]. Modern efforts to provide fast scientific libraries have incorporated C++ template metaprogramming to provide reusable abstractions, including the Epetra and Tpetra packages in Trilinos [20] and the Matrix Template Library [21]. Macro-based metaprogramming such as that in Lisp, newer versions of Scala [22] and Terra [23] are more powerful than the mechanisms in C++; however, the metaprogramming support in MSL is particularly powerful due to its integration with synthesis mechanisms, which could also be used to enhance Scala or C++ if it were integrated with those languages.

Domain-Specific Languages for HPC To ease development of high performance parallel software, a number of domain-specific languages have been developed, including for PDE solvers [24] and stencil computations [25, 26]. 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 [27] and Delite [28] that leverage advanced metaprogramming infrastructure to reduce that effort. MSL, in contrast, combines metaprogramming 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.

Software Synthesis In general, synthesis helps derive correct and efficient programs from specifications or higher level models. High-performance libraries such as Atlas [29] and SPIRAL [30] 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 [1, 7] 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 to find function inverses [31], to reverse engineer malware [32], and even to automatically grade programming assignments [33]. There has even been recent work on frameworks to make it easier to create synthesis-enabled domain specific languages [8, 9]. 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.
There has also been much prior work synthesizing concurrent data structures [34–38]. 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.

General Purpose Languages for HPC Distributed-memory languages for high performance computing include X10 [39], a distributed language with a notion of *places* that correspond to local state in our model; Titanium [13], a high level SPMD language with Java syntax; and Universal Parallel C (UPC) [40], an extension to C for SPMD programming on distributed memory machines. These have generally similar execution models to MSL, but lack the extensive metaprogramming and synthesis capabilities of our language. Other efforts to define future parallel languages include Chapel [41], High-Performance Fortran (HPF), and ZPL [42].

Verification of Parallel Programs Automatic static verification approaches for parallel programs employ similar simplifications to those we use. PUG [43] and GPUVerify [44] 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 [45, 46].

Compiler-based Optimization There is a long history of compiler optimizations to improve performance, including the polyhedral model [47–49] for loops. 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.

Autotuning Due to the failure of general compilers to acheive 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 [25, 50, 51] have extended this to other important domains and to compilers [52, 53]. 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.

Conclusion

This paper 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.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd Milind Kulkarni for their insightful comments, and our program manager Sonia Sachs for her guidance and support. Saman Amarasinghe, Ras Bodik, Adam Chlipala, and Martin Rinard have given invaluable advice to the authors. This work is partially supported by DOE Office of Science awards #DE-SC0008923 and #DE-SC0005372.

References

[1] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California, Berkeley, 2008.
[2] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004.
[3] T. Veldhuizen, "C++ gems," S. B. Lippman, Ed., 1996, ch. Expression templates, pp. 475–487.
[4] E. Niebler, "Proto: A compiler construction toolkit for dsels," in *Symposium on Library-Centric Software Design*, ser. LCSD '07, 2007.
[5] 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.
[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *OSDI*, 2004.
[7] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.
[8] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Onward*, 2013.
[9] ——, "A lightweight symbolic virtual machine for solver-aided host languages," in *PLDI*, 2014.
[10] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.
[11] H. Richardson, "High performance fortran: history, overview and current developments," 1.4 TMC-261, Thinking Machines Corporation, Tech. Rep., 1996.
[12] B. Carpenter, "Adlib: A distributed array library to support hpf translation," in *CPC*, 1995.
[13] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick, "Titanium language reference manual," EECS Department, University of California, Berkeley, Tech. Rep., 2005.
[14] A. Kamil and K. Yelick, "Hierarchical computation in the SPMD programming model," in *LCPC*, 2013.
[15] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Comm. ACM*, 1975.
[16] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *POPL*, 2004.
[17] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: A unified deadlock-free construct for collective and point-to-point synchronization," in *ICS*, 2008.

[18] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Auto-tuning the 27-point stencil for multicore," in *iWAPT2009*, 2009.

[19] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, and L. Snyder, "The design and implementation of a parallel array operator for the arbitrary remapping of data," in *PPoPP*, 2003.

[20] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, Sep. 2005.

[21] J. G. Siek and A. Lumsdaine, "The matrix template library: A generic programming approach to high performance numerical linear algebra," in *ISCOPE*, 1998, pp. 59–70.

[22] E. Burmako, "Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming," ser. SCALA '13, 2013.

[23] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: A multi-stage language for high-performance computing," in *PLDI*, 2013.

[24] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based pde solvers," *SC Conference*, 2011.

[25] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *IPDPS*, 2010.

[26] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures." in *IPDPS*, 2011.

[27] S. A. Kamil, "Productive high performance parallel programming with auto-tuned domain-specific embedded languages," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2013.

[28] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *PACT*, 2011.

[29] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing*, 1998.

[30] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *IJHPCA*, vol. 18, no. 1, 2004.

[31] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *PLDI*, 2011.

[32] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010.

[33] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI*, 2013.

[34] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *PLDI*, 2008.

[35] M. T. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," in *PLDI*, 2008.

[36] M. Kuperstein, M. T. Vechev, and E. Yahav, "Automatic inference of memory fences," in *FMCAD*, 2010.

[37] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav, "Dynamic synthesis for relaxed memory models," in *PLDI*, 2012.

[38] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv, "Concurrent data representation synthesis," in *PLDI*, 2012.

[39] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, 2005.

[40] UPC Consortium, "Upc language specification, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[41] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *IJHPCA*, vol. 21, no. 3, 2007.

[42] B. Chamberlain, "The design and implementation of a region-based parallel language," Ph.D. dissertation, University of Washington, 2001.

[43] G. Li and G. Gopalakrishnan, "Scalable smt-based verification of gpu kernel functions," in *FSE*, 2010.

[44] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer, "Engineering a static verification tool for gpu kernels," in *CAV*, 2014.

[45] S. F. Siegel and T. K. Zirkel, "Automatic formal verification of MPI-based parallel programs," in *PPoPP '11*, C. Cascaval and P.-C. Yew, Eds. ACM, 2011.

[46] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Formal analysis of mpi-based parallel programs," *Commun. ACM*.

[47] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *PACT*, 1998.

[48] U. Bondhugula, J. Ramanujam, and et al., "Pluto: A practical and fully automatic polyhedral program optimization system," in *PLDI 08*, 2008.

[49] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *PLDI*, 2013.

[50] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *IEEE*, vol. 93, no. 2, February 2005, invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[51] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Scientific Discovery through Advanced Computing Conference, Journal of Physics: Conference Series*, 2005.

[52] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *PLDI*, 2009.

[53] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *Supercomputing*, 2012.