Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures

Uday Bondhugula
Indian Institute of Science
Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, India
uday@csa.iisc.ernet.in

ABSTRACT

We present new techniques for compilation of arbitrarily nested loops with affine dependences for distributed-memory parallel architectures. Our framework is implemented as a source-level transformer that uses the polyhedral model, and generates parallel code with communication expressed with the Message Passing Interface (MPI) library. Compared to all previous approaches, ours is a significant advance either (1) with respect to the generality of input code handled, or (2) efficiency of communication code, or both. We provide experimental results on a cluster of multicores demonstrating its effectiveness. In some cases, code we generate outperforms manually parallelized codes, and in another case is within 25% of it. To the best of our knowledge, this is the first work reporting end-to-end fully automatic distributed-memory parallelization and code generation for input programs and transformation techniques as general as those we allow.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code generation*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Polyhedral model, distributed-memory, code generation, affine loop nests, parallelization

1. INTRODUCTION AND MOTIVATION

Shared memory for multiple processing elements is a useful abstraction for parallel programmers. However, due to limitations in scaling shared memory to a large number of processors, the compute power of shared-memory multiprocessor systems is limited. To obtain greater processing power, multiple processing nodes are connected with a high performance interconnect such as InfiniBand or 10 Gigabit Ethernet to form a cluster. Each node has its own memory space that is not visible to other nodes. The only way to share data between nodes is by sending and receiving messages over the interconnect. The Message Passing Interface (MPI) [28] is the current dominant parallel programming model used to program compute-intensive applications on such distributed-memory clusters.

Distributed memory makes parallel programming even harder from many angles. A programmer has to take care of distribution and movement of data in addition to distribution of computation. Data distribution and computation distribution are tightly coupled – changing the data distribution in a simple way often requires a complete rewrite of compute and communication code. Debugging multiple processes that send and receive messages to and from each other is also significantly more difficult. Parallelizing even simple regular loop nests for distributed memory can be very error-prone and unproductive. In addition, whenever pipelined parallelism exists, i.e., not all processors are active to start with, or when there is a significant amount of discontiguous data to be transferred to multiple nodes, which is often the case, MPI parallelization can be a nightmare. Hence, a tool that can automatically parallelize for distributed-memory parallel architectures can provide a big leap in productivity.

In this paper, we propose techniques and optimizations for automatic translation of regular sequential programs to parallel ones suitable for execution on distributed-memory machines: typically, a cluster of multicore processors. We use the polyhedral compiler framework to accomplish this in a portable and efficient manner. As a result, we are able to handle sequences of arbitrarily nested loops with regular (affine) accesses, also known as *affine loop nests*. Affine loop nests appear in scientific and embedded computing domains in applications such as dense linear algebra codes, stencil computations, and image and signal processing applications. We would like to emphasize that distributed-memory compilation of even this restricted class of codes is very challenging and no automatic solution exists despite decades of research. Larger programs that comprise both affine and non-affine parts would also benefit from such techniques for the affine portions.

The problem of distributed memory parallelization requires a solution to several sub-problems. New techniques presented in this paper are for efficient communication code generation, i.e., when a transformed parallelized code is given as input. Hence, approaches that determine computation or data partitioning are orthogonal to it. A characteristic of the system we developed is that it is not driven by a data distribution and no data distributions need to be specified. Data moves from processor to another in a manner completely determined by the computation partitioning and data dependences, and there exists no fixed owner for data. An initial data distribution can be specified, but it would only affect communication at the start

SC13 November 17-21, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503289

and at the end.

For experimental evaluation, we coupled our proposed code generation technique with a computation partitioning-driven polyhedral parallelizer Pluto [9, 30]. Our framework is thus implemented as a source-level transformer that generates parallel code using the MPI library as its communication backend. No pragmas, directives, or distributions are provided to our system, i.e., it is fully automatic. Code we generate is parametric in the number of processors and other problem sizes, and provably correct for any number of MPI processes. Besides parallelizing for distributed memory, code we generate is also optimized for locality on each core.

Our contributions over previous works are one or more of the following: (1) handling imperfect loop nests with affine dependences, (2) significantly less communication in the presence of parametric problem sizes and number of processors, and (3) fully automatic end-to-end capability for distributed-memory parallelization.

The rest of this paper is organized as follows. Section 2 provides background and notation. Section 3 provides some more detail on the problem and challenges. Section 4 and Section 5 describe our solution and optimizations. Section 6 provides experimental results. Discussion of related work is presented in Section 7 and conclusions are presented in Section 8.

2. BACKGROUND AND NOTATION

The polyhedral compiler framework is an abstraction for analysis and transformation of programs. It captures the execution of a program in a static setting by representing its instances as integer points inside parametric polyhedra. Most publicly available tools and compilers that use this framework extract such a representation from C, C++, and Fortran programs.

Polyhedral representation of programs: Let S_1, S_2, ..., S_n be the statements of the program. Each dynamic instance of a statement, S, is identified by its iteration vector $\vec{i}$ that contains values for indices of the loops surrounding S, from outermost to innermost. Whenever the loop bounds are affine functions of outer loop indices and program parameters, the set of iteration vectors belonging to a statement form a convex polyhedron called its *domain* or *index set*. Let I_S be the index set of S and let its dimensionality be m_S. Let $\vec{p}$ be the vector of program parameters. Program parameters are not modified anywhere in the portion of code we are trying to model.

A function f on a domain I_S is called an affine function if it can be represented in the following form:

$$f(\vec{i}) \quad = \quad [c_1 \ \ldots \ c_{m_S}] . \left(\ \vec{i} \ \right) + c_0, \ \ \vec{i} \in I_S$$

Regular data accesses in a statement are represented as multi dimensional affine functions of domain indices. Codes that satisfy these constraints are also known as *affine loop nests*.

Polyhedral dependences: The data dependence graph (DDG) is a directed multi-graph with each vertex representing a statement, and an edge, $e \in E$, from node S_i to S_j representing a polyhedral dependence from an iteration of S_i to an iteration of S_j: it is characterized by a polyhedron, D_e, called the *dependence polyhedron* that captures exact dependence information corresponding to e. The dependence polyhedron is in the sum of the dimensionalities of the source and target iterations spaces, and the number of program parameters. At least one of the source and target accesses has to be a write. Data dependence polyhedrons are important for our approach since they dictate what communication needs to occur.

For example, for the code in Figure 1, the dependence between the write $a[i][j]$ at $\vec{s} = (t, i, j)$ and the read at $\vec{t} = (t', i', j')$ at $a[i'-1][j'-1]$ is given by the dependence polyhedron, $D_e(\vec{s}, \vec{t}, \vec{p}, 1)$,

```
for (t=0; t<=T−1; t++)
  for (i=1; i<=N−2; i++)
    for (j=1; j<=N−2; j++)
      a[i][j] = (a[i−1][j−1] + a[i−1][j] + a[i−1][j+1]
                + a[i][j−1] + a[i][j] + a[i][j+1] +
                a[i+1][j−1] + a[i+1][j]+ a[i+1][j +1])/9.0;
```

Figure 1: Seidel-style code

which is a conjunction of the following equalities and inequalities:

$$i' = i + 1, \quad j' = j + 1, \quad t' = t,$$
$$0 \ \leq \ t \leq \ T - 1, \quad 1 \ \leq \ i \leq \ N - 3, \quad 1 \ \leq \ j \leq \ N - 3$$

3. PROBLEM AND CHALLENGES

When compiling for shared memory, synchronization primitives take care of preserving data dependences when dependent iterations are mapped to different processors. Shared memory support provided by hardware takes care of transparently providing data that had been written to by one processor before a synchronization point, to another one after it. However, in case of distributed-memory systems, this movement of data has to be performed in software via communication over the interconnect.

Code parametric in problem sizes and number of processors can be very important for portability. For proprietary software that is distributed, a developer or vendor would not be able to provide binaries for each problem size and processor configuration, and a user would not be able to recompile. A number of difficulties arise in compilation when the number of processors or problem size symbols are not known at compile time, and have to be treated as parameters. This is not an issue with shared-memory auto-parallelization. OpenMP support takes care of partitioning a parallel loop with a choice of strategies, and since no software data transfer is performed, the two steps of generation of parallel code (marking a loop as parallel) and that of distributing the parallel loop across processors are decoupled. No matter how the parallel iterations are scheduled across processors, the hardware would transparently guarantee visibility of correct data after synchronization points.

Plugging in the number of processors as a parameter in the polyhedral representation does not help since it introduces non-affine expressions. For example, a simple loop of N iterations, when divided across $nprocs$ processors in a block manner would lead to the following SPMD code (even if N to be a multiple of $nprocs$ below):

```
for (i=my_rank∗N/nprocs; i<= (my_rank+1)∗N/nprocs−1; i++)
```

As can be seen, the number and identity of communication partners as well as communication data may often depend on the total number of processors as well as other program symbols, actual values of which will only be known at runtime. This is because dependences, which are responsible for communication, will cross over to an unknown number of processors, when viewed at compile time. Thus, there is no trivial way to use polyhedral machinery to provide a precise solution to this problem. As we will see, existing works that are able to handle affine loop nests as well as symbols lead to significant redundant communication. We provide a much more efficient solution to this problem. We will also see that the proposed solution is more efficient than previous schemes even when these parameters are known at compile time. Moreover, the above discussion assumes a block distribution. A block-cyclic or a customized allocation that is determined at runtime (a-priori or

on-the-fly) makes the problem even harder.

In addition, before generation of distributed-memory code, a sequence of complex transformations may need to be applied to parallelize the code or to improve its locality. Such transformations have already been shown to improve sequential performance significantly [25, 13, 10, 32]. Applying such transformations provides a distributed-memory code generator with only more complex input and the ability to deal with it seamlessly is crucial to generate high-performance code. The solution we propose is thus designed to deal with any arbitrary composition of affine transformations applied on to a sequential affine loop nest.

4. DISTRIBUTED MEMORY CODE GENERATION

In this section, we describe all steps involved in obtaining communication code given the original program and a transformation or computation partitioning for it.

4.1 Dependences and communication

When code is partitioned across multiple processing elements, any communication required arises out of data dependences. Recall that there are primarily three types of data dependences: flow (Read-after-Write or RAW), anti (Write-after-Read or WAR), and output (Write-after-Write dependences). It is interesting to contrast the effect of these dependences when compiling for shared-memory versus for distributed-memory systems. Anti and output dependences merely exist because the same memory location is being reused. In case of shared memory auto-parallelization, anti and output dependences are still important – this is because when iterations that are dependent via such a dependence are mapped to different processors, owing to the same shared memory location they access, synchronization is needed. However, in case of distributed memory, each processor has its own address space. This coupled with the fact that there is no flow of data associated with anti and output dependences implies that they neither lead to communication nor synchronization.

Note that our goal is to generate a distributed-memory program that preserves semantics of the original sequential program. Once the parallelized portion of the input code finishes execution, all results are to be available at a single process, say, the master process. Thus, even in the absence of any dependences, communication is needed to make sure that all results will have been gathered at the master process by the time all parallel processes have finished executing. We show that this communication code can be generated efficiently using output (WAW) dependences.

A loop that can be placed at any level (outermost to innermost) and marked parallel is called an outer parallel or a communication-free parallel loop – it will have no dependence components along it. Outer parallelism will require no communication except a gather of results at the master process. Wherever pure inner parallelism exists, i.e., communication cannot be avoided via transformation, generating efficient communication code is crucial. Note that inner parallelism, wavefront parallelism, and pipelined parallelism can all be converted into inner parallelism, i.e., one parallel loop followed by a synchronization call when running on shared memory, or communication code in case of distributed memory.

4.2 Computing communication sets

In the rest of this section, by *tile* we refer to the portion of computation under a given iteration of the parallel dimension, i.e., all dimensions surrounding it and including itself serve as parameters for the tile. It may or not have been obtained as a result of loop tiling. It is the smallest piece of computation for which we will define communication sets. It is important to note that constraints that describe a tile's domain are affine at compile time. We classify communication data for a tile into two classes:

1. Writes to locations that are read at another process either the next time the same loop is run in parallel, i.e., for another iteration of the surrounding (outer) sequential loop if any, or at a subsequent parallel loop if any.

2. All results or the last writes need to be available at a root node (or across nodes if an initial data distribution has been specified) once parallelized computation has finished executing, i.e., final writes for all data spaces need to be aggregated or rearranged.

We show that by computing two sets for each data variable, one for each of the above cases, one can determine all that has to be sent out from a process after it has finished executing a tile. We call these the *flow-out* set and the *write-out* set. Each of these sets can be a union of convex polyhedra. The integer points in these polyhedra yield actual data elements to be communicated.

Running example: We use the code in Figure 2 as an example to demonstrate all steps, showing results and code they yield at each step. This is a typical Jacobi-style stencil with time along the vertical axis and space along the horizontal. For simplicity, assume that all dimensions are tiled by a factor of 32. Tiling serves a number of purposes in our context: increasing granularity of parallelism and reducing the frequency of communication, improving locality, and bounding buffer sizes by a factor proportional to tile size where possible.

In the rest of this section, whenever we refer to a set, we mean a union of convex polyhedra with integer points enclosed by them being of interest to us. Whenever a set of linear equalities and inequalities are listed, they represent a conjunction of those. Recall notation introduced in Section 2. $\setminus$ is used as the set difference operator. In addition, some polyhedral operations are notated as below:

$project\_out(D, p, n)$:		eliminates n dimensions from set D starting from p^{th} dimension ($p \geq 1$)
$\mathcal{I}_p(M, \mathcal{D}, l)$	:	image of $\mathcal{D}$ under a multi-dimensional affine function M while treating l outermost dimensions as parameters

Algorithms for projection are provided by polyhedral libraries. For parametric image, the chosen parameters are not projected out of the image. In particular, if one needs data accessed for a given set of outer loops through an access function M, the outer loops are to be treated as parameters just like other program parameters appearing in loop bounds, and M is used as the function for the image operation. As an example, if $\mathcal{D}$ is

$$1 \leq i \leq N-1, \quad 1 \leq j \leq N-1$$
$$32i_T \leq i \leq 32i_T + 31, \quad 0 \leq j - 32j_T \leq 31$$

and $M = (i-1, j-1), l = 1$. Then, $\mathcal{I}_p(M, \mathcal{D}, l)$ is

$$0 \leq d_0 \leq N-2, \quad 32i_T \leq d_0 \leq 32i_T + 31$$
$$0 \leq d_1 \leq N-2$$

Let e be a RAW dependence from S_i to S_j. We introduce the following notation and symbols used in the algorithms that follow.
D_e^T: dependence polyhedron for edge e in the transformed space
$I_{S_i}^T, I_{S_j}^T$: domains of S_i and S_j in the transformed space with dimensionalities $m_{S_i}^T$ and $m_{S_j}^T$ respectively

```
for (i=1; i<=T-1; i++){
  for (j=1; j<=N-1; j++){
    u[i%2][j] = 0.333*(u[(i-1)%2][j-1]
        + u[(i-1)%2][j] + u[(i-1)%2][j+1]);
      }
  }
```

Figure 2: Jacobi-style code

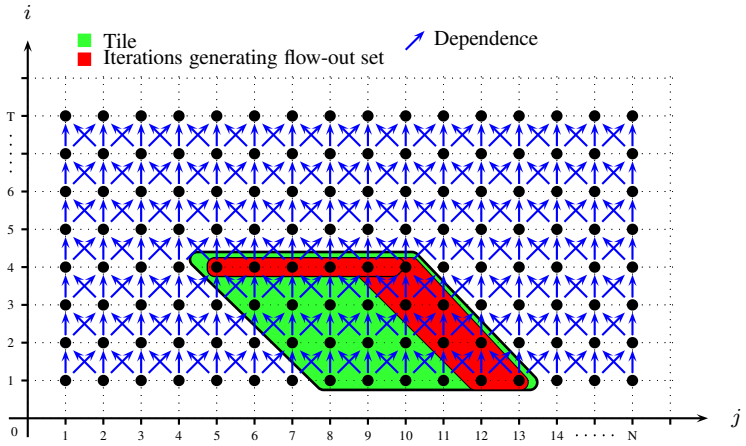


Figure 3: Iterations generating flow-out set for one tile of code in Figure 2

I_e^t: iterations inside the tile reading data written to in the tile as a result of dependence e
O_e^t: iterations of the tile whose (source) writes are read by iterations outside the tile through dependence edge e
t_k^i: k^{th} dimension of $I_{S_i}^T$

Transformed dependence polyhedra, D_e^T (and similarly transformed index sets, I_S^T) can be generated by taking the original dependence polyhedra and augmenting them with transformation functions (also called affine schedules) that map old iterators to new ones, and then projecting out the old iterators. For a dependence edge e, this yields the dependence relation between iterations in the transformed space D_e^T. Note that this is how algorithms proposed below will work transparently in conjunction with any arbitrary composition of affine transformations on the original polyhedral representation that was extracted, i.e., the code being generated is for the transformed program (if transformations were applied) and not the one extracted. Since multidimensional affine schedules capture distribution and fusion within themselves [8], any sequence of imperfect loop nests or transformations that lead to a sequence of imperfect loop nests are naturally handled this way. Thus, there is no explicit view of nesting in the algorithms below. Note that the source and the target of a dependence could belong to statements that appear in different loop nests, and communication across different parallel loop nests gets generated in this manner.

4.2.1 *Flow-out set*

The *flow-out* set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile. The first step is that of finding the subset of the transformed dependence polyhedron that has both its source and target iterations in the same tile. If l is the depth of the loop dimension being parallelized, this can be obtained by intersecting D_e^T with a set of equalities equating the first l dimensions of $I_{S_i}^T$ to those of $I_{S_j}^T$. Recall again the connection, introduced in the beginning of Section 4.2, between a *tile* and the loop dimension being parallelized. A single iteration of the loop being parallelized is the tile for which communication sets are being constructed. Let E_l be that set of equalities. Then,

$$E_l = \left\{ t_1^i = t_1^j \ \wedge \ t_2^i = t_2^j \ \wedge \ \ldots \ \wedge \ t_l^i = t_l^j \right\}$$

We now obtain the set of all iterations of the tile that write to values that are later read within the same tile through dependence edge e:

$$\begin{aligned} C_e^t &= D_e^T \cap E_l \\ I_e^t &= project\_out\left(C_e^t, m_{S_i}+1, m_{S_j}\right) \end{aligned}$$

Next, subtracting I_e^t from the set of all source dependence iterations in the tile yields those source dependence iterations whose writes are read outside the tile:

$$O_e^t = project\_out\left(D_e^T, m_{S_i}+1, m_{S_j}\right) \setminus I_e^t$$

Now, computing the image of the source write access function, M_w, on O_e^t yields the flow-out set for this particular write access and dependence.

$$F_{out}^x = \mathcal{I}_p(M_w^{S_i}, O_e^t, l)$$

Algorithm 1 computes the entire flow-out set for a particular variable. Note that the write access/statement pairs provided to it as input are from statements that fall within the loop dimension being parallelized.

Algorithm 1 Computing flow-out set for variable x

INPUT Depth of parallel loop: l; set $\mathbf{S_w}$ of $\langle$write access, statement$\rangle$ pairs for variable x
$F_{out}^x = \emptyset$
for each $\langle M_w, S_i \rangle \in \mathbf{S_w}$ **do**
  for each dependence $e(S_i \rightarrow S_j) \in E$ **do**
    if e is of type RAW and source access of e is M_w **then**
      $E_l = \left\{ t_1^i = t_1^j \ \wedge \ t_2^i = t_2^j \ \wedge \ \ldots \ \wedge \ t_l^i = t_l^j \right\}$
      $C_e^t = D_e^T \cap E_l$
      $I_e^t = project\_out\left(C_e^t, m_{S_i}+1, m_{S_j}\right)$
      $O_e^t = project\_out\left(D_e^T, m_{S_i}+1, m_{S_j}\right) \setminus I_e^t$
      $F_{out}^x = F_{out}^x \cup \mathcal{I}_p(M_w^{S_i}, O_e^t, l)$
    end if
  end for
end for
OUTPUT F_{out}^x

Since anti and output dependences are ignored in the above construction, multiple copies of the same location may exist with different processors during the parallelization. However, when a later read to the same location happens, the correct written value would end up being transferred due to the presence of a flow edge between the write and the read.

4.2.2 Write-out set

The *write-out* set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile. We compute this by looking for any WAW edges leaving the tile. If they do, subtracting the sources of those edges from the set of all points written to in the tile in an iterative manner across all WAW dependences leaves us with locations that have been "finalized" by computation in the tile. A union has to be taken across all write accesses to a given variable in a tile. For edge e associated with variable x, let:
F_e^t: iterations inside the tile that write to locations that will again be written to outside the tile through edge e,
W^x: write-out set due to a given write access for variable x, and
W_{out}^x: write-out set for variable x.
Algorithm 2 below computes the write-out set for a variable.

Algorithm 2 Computing write-out set for variable x

INPUT Depth of parallel loop: l; set $\mathbf{S_w}$ of $\langle$write access, statement$\rangle$ pairs for variable x

$W_{out}^x = \emptyset$
for each $\langle M_w, S_i \rangle \in \mathbf{S_w}$ **do**
  $W^x = \mathcal{I}_p(M_w, I_{S_i}^T, l)$
  for each dependence $e(S_i \rightarrow S_j) \in E$ **do**
    if e is of type WAW and source access of e is M_w **then**
      $E_l = \left\{ t_1^i = t_1^j \ \wedge \ t_2^i = t_2^j \ \wedge \ \ldots \ \wedge \ t_l^i = t_l^j \right\}$
      $C_e^t = D_e^T \setminus E_l$
      $F_e^t = project\_out\left(C_e^t, m_{S_i} + 1, m_{S_j}\right)$
      $W^x = W^x \ \setminus \ \mathcal{I}_p(M_w, F_e^t, l)$
    end if
  end for
  $W_{out}^x = W_{out}^x \cup W$
end for

OUTPUT W_{out}^x

4.2.3 Example

For the code in Figure 2, Figure 3 shows the flow-out set (with tile size reduced to 4x6 from 32x32 for easier illustration). It is obtained as a union of the following two polyhedra:

$$1 \ \leq \ i \leq \ T-2, \quad 1 \ \leq \ j \leq \ N-2$$
$$32i_T + 30 \ \leq \ d_0 + d_1 \leq \ 32i_T + 31$$
$$32i_T \ \leq \ d_0 \leq \ 32i_T + 31$$

$$i \ = \ 32 * i_T + 31, \ \ 1 \ \leq \ i \leq \ T-2$$
$$1 \ \leq \ j \leq \ N-1, \ \ 32i_T \ \leq \ d_0 + d_1 \leq \ 32i_T + 31$$

The second one corresponds to the horizontal line, while the first to the two oblique lines. d_0 and d_1 will index the array dimensions in the copy-out code. Note that for simplicity, the above constraints are expressed in terms of source iterators. They are actually computed in the space of transformed iterators, i.e., in terms of (t_1, t_2) where $t_1 = i$, $t_2 = i + j$, since tiling has been performed here after a skewing of the space dimension. As for the write-out set, writes that are part of the last two iterations of the outer loop here are last writes, and only the tiles containing those iterations will have a non-empty write-out set.

4.3 Packing and unpacking communication sets

With MPI, it is easy to transfer data from, and receive into, contiguous buffers. However, in most cases, we require discontiguous data to be sent and copied back on the receiver side. Hence, after the above communication sets are computed, one has to, (1) pack data to be sent in a contiguous buffer, (2) map to communication library calls, (3) unpack data at receiver side, and (4) determine send and receive buffer sizes for allocation. We construct additional statements to add to the polyhedral representation of the source program for copying out (pack) and copying back (unpack). The flow-out and write-out sets serve as the domains for the copy statements. The pack and the corresponding unpack statements have identical domains with the copy assignment reversed. An alternative to packing and unpacking is to use MPI derived data types. However, we find automating with MPI derived data types to be harder than with a simple linearized pack, and we do not explore it further.

Reasonably tight upper bounds on send and receive buffer sizes can be determined from tile constraints; we do not present details on it here due to space constraints. Write-out sets are gathered at the master process – in our case, this can be chosen to be the MPI process with rank 0.

A naive approach: A naive approach would be for each processor to send its flow-out set to all other processors. This means that all of the data to be sent is sent, but not just to the processors that need them. Hence, a processor may receive more data than necessary, and a processor that need not receive any data may receive some. Obviously, this leads to a large amount of redundant communication. However, it provides a simple clean way to generate communication code. Two of MPI's collectives, `MPI_Allgatherv` (all-to-all broadcast) and `MPI_Gatherv` perfectly fit. Allgather can be used to broadcast flow-out sets to all processors. The Gather call with process 0 as root is used to collect write-out sets. The next section presents a more precise scheme.

5. OPTIMIZING COMMUNICATION CODE

Recall that communication sets were defined per 'tile', for which we have affine constraints at compile time. This allowed us to use polyhedral machinery to compute them in the first place. Multiple such tiles may get mapped to a single physical processor and communication is done only after all of these tiles have been executed, for every iteration of immediately surrounding sequential loop, if any.

5.1 Precise determination of communication partners

The naive scheme mentioned at the end of the previous section broadcasts flow-out sets to all processors. In cases, where we have inner parallelism, depending on communication latencies and bandwidth, this will likely lead to a bottleneck. Recall again that the problem in determining communication partners was that the allocation of tiles to processors is not known at compile time. Consider the simple scenario when the number of communication partners itself depends on the total number of processors. Long dependences may traverse any number of processors. However, in many cases such as in the presence of uniform dependences, only near-neighbor communication is needed. Even in these cases, if iteration spaces are shaped peculiarly, one cannot predict near-neighbor communication just based on dependence distances. Hence, even for uniform dependences, the number and identity of communication partners cannot be determined at compile time.

We describe a solution below that achieves the following: the flow-out set is not sent to processors that do not need any value from this flow-out set. More precisely, we guarantee the following:

1. Every element in the flow-out set sent by a processor is needed by at least one other processor
2. Only processors that expect to receive at least one value from

another processor receive the flow-out set

We define two functions as part of the output code for each data variable, x, that can be a multidimensional array or a scalar. If $t_1, t_2, \ldots, t_l$ are the sequential dimensions surrounding the parallel dimension t_p, i.e., $(t_1, t_2, \ldots, t_l)$ form a prefix for the complete iteration vector, the functions are:

1. $\pi(t_1, t_2, \ldots, t_l, t_p)$: rank of processor that executes iteration $(t_1, t_2, \ldots, t_l, t_p)$
2. $\sigma_x(t_1, t_2, \ldots, t_l, t_p)$: set of processors that need the flow-out set of $(t_1, t_2, \ldots, t_l, t_p)$ for data variable x.

Generating π and σ: Code for π and σ functions is meant to be generated and added to output code. Constructing π is straightforward. It only requires the lower and upper bound expressions for t_p, and the number of processors. π is also used in computing σ_x. σ_x can be expressed as follows.

$$\sigma_x(t_1, t_2, \ldots, t_l, t_p) = \{\pi(t'_1, t'_2, \ldots, t'_l, t'_p) \mid \exists e \in E \text{ on } x, D_e^T(t_1, t_2, .., t_p, .., t'_1, t'_2.., t'_p, .., \vec{p}, 1)\}$$

σ can be constructed as follows for each variable x. For each relevant RAW dependence polyhedron in the transformed space, we eliminate all dimensions that are inner to t_p. We then scan the dependence polyhedron to generate loops for the target iterators while treating source iterators as parameters, i.e., running the generated loop nest at run-time will enumerate all dependent tiles, $(t'_1, t'_2, ..., t'_l, t'_p)$, given the coordinates of the source tile. However, our goal is not to enumerate dependent tiles, but to determine processors they are mapped to. Hence, σ makes use of π to aggregate a set of distinct values corresponding to processor ranks that the target tiles were mapped to. The overhead of evaluating σ at runtime is minimal since a call to it is made only once per all computation for a given t_p.

Send-synchronous scheme: With σ and π functions, generating more accurate communication code for a parametric number of processors now becomes possible. Processes send out data to the set of processor ranks returned by σ, and receivers are *forced* to receive them. The receivers will use received data in one or more future iterations. Hence, sends and receives are posted in a synchronous manner, relatively speaking. Non-blocking sends and receives are used so that simultaneous progress is made on all sends across all data variables when possible. We wait for their completion (`MPI_Waitall`) and copy-back received data to the right place before the next iteration of the sequential loop outer to the parallel one starts.

An arbitrary allocation: A powerful feature of this scheme is that an arbitrary π function can be used. So far, we have only alluded to a block scheduling of the parallel loop. However, π can be generated to achieve a block-cyclic scheduling, or any custom allocation as long as the allocation is known a-priori at runtime, i.e., the allocation of tiles to processors is not determined on-the-fly during execution of a parallelized loop. Since σ makes use of π, π has to be known at runtime a-priori, and it can be set appropriately to achieve the desired allocation. In addition, it is easy to use a multidimensional π whenever there is more than one parallel dimension. Such mappings to higher dimensional processor spaces are known to achieve better computation to communication ratios, for a given number of processors and problem size.

Figure 4 and Figure 5 shows the code for the Floyd-Warshall algorithm and its communication pattern. If the flow-out set and sigma is computed for each of the points (which in itself could be a tile), and aggregated for all points mapped to a processor, we end up broadcasting the flow-out set to processors executing tiles along the same row and along the same column.

```
for (k=0; k < N; k++) {
  for (y=0; y < N; y++) {
    for (x=0; x < N; x++) {
      pathDistanceMatrix [y][x] = min(pathDistanceMatrix [y][k] +
              pathDistanceMatrix [k][x], pathDistanceMatrix [y][x]);
    }
  }
}
```

Figure 4: Floyd-Warshall algorithm

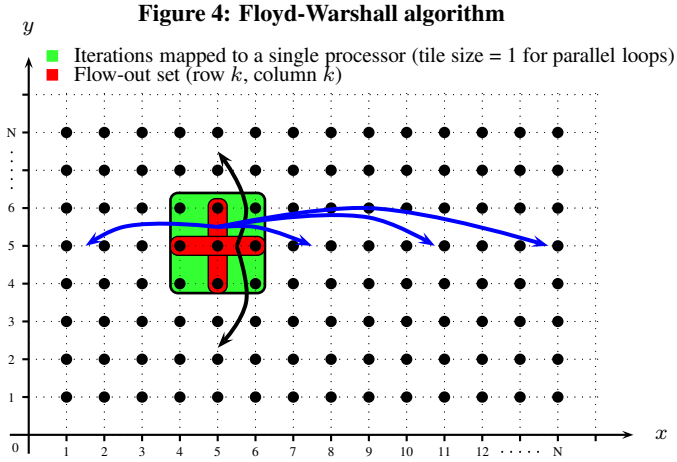


Figure 5: Communication for Floyd-Warshall: at outer loop iteration $k-1$, processor(s) updating the k^{th} row and k^{th} column broadcast them to processors along their column and row respectively.

5.2 Transitivity in dependences

We know that dependences that lead to communication of flow-out sets are RAW dependences. If RAW dependences are transitively covered by other dependences, one would end up communicating from several sources instead of just the last one, i.e., the one that writes last to the location as per the original program. We use a dependence tester that can compute such last writers or the exact data flow so that RAW dependence polyhedra do not contain any redundancy.

Interestingly, one observes the converse of the above effect when dealing with write-out sets. Note that transitivity can be eliminated from write-after-write dependences as well. A WAW dependence can be covered by other WAW dependences as well as through a combination of RAW and WAR dependences. If transitivity is eliminated for WAW dependences and with Algorithm 2 only looking at WAW dependences, one would miss writes that happen outside the tile and to the same locations written to in a tile. This leads to a write-out set much larger than the actual one, often, almost the entire set of locations written to. Hence, Algorithm 2 will only be exact if all transitively covered WAW dependences are preserved. In summary, one has to rely on the right dependence testing and analysis techniques. ISL [37] provides functions to compute last writers as desired as part of its dependence testing interface and we use it.

5.3 Putting all communication code together

We compute domains, schedules for the additional copy-out, copy-back, and communication statements, and they have all components in the polyhedral representation just like the original compute

statements. The new program comprising these added statements is given to the code generator, to generate final code in one pass.

Our framework is implemented as part of the publicly available source-to-source polyhedral tool chain. Clan [8], ISL [37], Pluto [30], and Cloog-isl [12] are used to perform polyhedral extraction, dependence testing, automatic transformation, and code generation, respectively. The Pluto scheduling algorithm [9, 10] is first used to determine a parallelizing transformation, i.e., a computation partitioning. Such a parallelizing transformation is a composition of several simpler affine transformations (including loop tiling) that specify a new execution order completely and identify loops to be parallelized. To implement polyhedral operations for all computations in Section 4 and Section 5, Polylib [31] was used. A powerful feature of our framework is that it will work with any other algorithm for transformation and detection of parallelism. As a result of using the polyhedral representation as both the input and output of our scheme, code can be generated after any sequence of valid transformations have been applied. Our scheme can thus be used in a system that specifies data and computation distributions in a different way as long as these mappings can be expressed as affine functions.

5.4 Data distribution free

The execution model of our distributed-memory parallelization tool chain described above is computation-driven. Data moves from one processor to another in a manner completely determined by the computation partitioning and data dependences. There exists no owning processor for data. A scalable execution model would require an initial data distribution to be specified due to memory space constraints at a single node. Such a data distribution would only impose one-time communication at the start with our model. A read-in set analogous to the write-out set can be constructed to bring in only the first read data. Similarly, last writes can be put back as per the initial distribution instead of being gathered at the root node. The computation partitioning itself that was obtained through an objective function (Pluto) to minimize communication and maximize locality does not change. However, if the one-time first read and last write communication costs are to be minimized, a data distribution that aligns better with the computation partitioning has to be specified.

With our current implementation, no pragmas, directives, or distributions are provided to the system, i.e., it is fully automatic with initial data assumed to be present on all nodes. Generated code is SPMD, and we choose to gather all results at process '0' only to provide exactly the same behavior as the unmodified sequential input program for easy testing. Handling an initial data distribution automatically would also require a data allocation and indexing technique for computation on each node. We do not discuss details on allocation of data in this paper, but techniques that deal with this problem already exist in the literature [6, 19], and we plan to integrate one in the future.

5.5 On communication optimality

In spite of the techniques proposed in this section, the amount of data communicated is not optimal when different parts of the flow-out set have different σs, i.e., different lists of receiving processors. An approach explored by a recent work based on ours [15] defines a "flow-in" set analogous to the flow-out set and intersects the flow-out set of a sender with the receiver's flow-in set to exactly determine data required by a receiver tile. Though it would appear to make the communication set more accurate, it introduces a problem that our current scheme does not exhibit. If two receiving tiles map to the same processor, one would end up sending necessary data multiple times. Hence, a different kind of redundancy of duplicate communication is introduced. This would be common in broadcast or multicast style communication patterns and when the number of tiles is larger than the number of processors – larger the number of tiles, greater the duplicate communication. One would need a combination of multiple techniques, possibly including runtime ones, to deal with this problem. This is also the reason that in spite of precise polyhedral dependences, owing to unknown problem sizes and number of processors, it is hard to achieve optimal communication volume. Dathathri et al. [15] explores related problems and some solutions in more detail. An optimal decomposition of flow-out sets if at all possible at compile time is still left for future research.

5.6 Improvement over previous schemes

In this section, we describe in detail how our scheme improves over existing ones for communication code generation. We consider three past works that subsume others in the literature. These are that of Amarasinghe and Lam [2], Adve and Mellor-Crummey [1], and Classen and Griebl [11]. The above schemes have the following limitations that we have overcome:

1. All three approaches used a virtual processor to physical processor mapping to deal with symbolic problem sizes and number of processors. Communication finally occurs between virtual processors that do not map to the same physical processor. In spite of this, if multiple receiving virtual processors map to the same physical processor and data being sent to two or more of these is not disjoint, the receiving physical processor ends up receiving necessary data multiple times. For example, if a virtual processor V_i is mapped to physical processor P_i and two other virtual processors V_{j1} and V_{j2} are both mapped to physical processor P_j ($P_i \neq P_j$), and $V_i(P_i)$ sends the same data or a large portion of the same data to both $V_{j1}(P_j)$ and $V_{j2}(P_j)$. Avoiding it is not trivial since one has to look for commonality in data being sent out across a set of receiving virtual processors as well as determine the list of receivers – these are only known at runtime if the number of processors and problem sizes are parametric. The sigma function-based solution presented in this section provided a solution to this problem.

2. [1] determine communication sets by directly looking at read and write accesses as opposed to data dependences. Communication is only needed at the last write before a read if pushing data, or at the first read after a write if using a pull-based mechanism. Algorithms presented in [1] do not appear to consider this issue. Since our approach relies on dependences, this requirement is easily captured in the lastwriter property of flow dependences.

Note that the second limitation also compounds redundancy created due to the first. Not eliminating transitive relations leads to more one-to-many patterns and such one-to-many patterns that in turn leads to greater redundant communication with a simple virtual to physical processor model. The approach of Classen and Griebl [11] does not suffer from the second limitation since it is based on dependences like ours. Communication polytopes are constructed for *each* flow dependence, and so communication code is generated dependence-wise. Since communication sets for multiple dependences may often refer to the same values, a new source of redundant communication is added. Their work was thus preliminary and conceptual, and reported very limited implementation and experimental evaluation.

6. EXPERIMENTAL EVALUATION

Setup: We conducted experiments on a 32-node InfiniBand cluster of dual-SMP Xeon servers. Each node comprises two quad-core Intel Xeon E5430 2.66 GHz processors with a 12 MB L2 cache and 16 GB of main memory. The InfiniBand host adapter is a Mellanox MT25204. All run Linux 2.6.18 64-bit. MVAPICH2-1.8.1 [29] (MPI over InfiniBand) is the MPI implementation used. On this cluster, it provides a point-to-point latency of 3.36 μs, unidirectional and bidirectional bandwidths of 1.5 GB/s and 2.56 GB/s respectively. All codes were compiled with Intel C/C++ compiler (ICC) version 11.1 with option -fast (implies '-O3 -ipo -static' on 64-bit Linux). Portland Group's compiler pghpf 12.1 (with -O4 -Mmpi) was used where a comparison with HPF was performed – it was the only publicly available HPF compiler we could find.

Input sequential code without any modification is taken in by our system and compilable MPI code is generated fully automatically in all cases. For the π function, a simple block scheduling is used for rectangular iteration spaces and block-cyclic for non-rectangular ones. The entire framework runs fast and the increase in source-to-source transformation time due to distributed memory compilation is less than 1.5s in all cases. We thus did not pay particular attention to optimize compilation time at this point.

Benchmark	Problem size
strmm	10000
trmm	8000
dsyr2k	4096
covcol	N = 8192
seidel	N = 10000, T = 600
jac-2d	N = 10000, T = 1000
fdtd-2d	N = 6000, T = 256
2d-heat	N = 10000, T = 1000
3d-heat	N = 512, T = 256
lu	N = 4096
floyd-warshall	N = 8192

Table 1: Problem sizes used

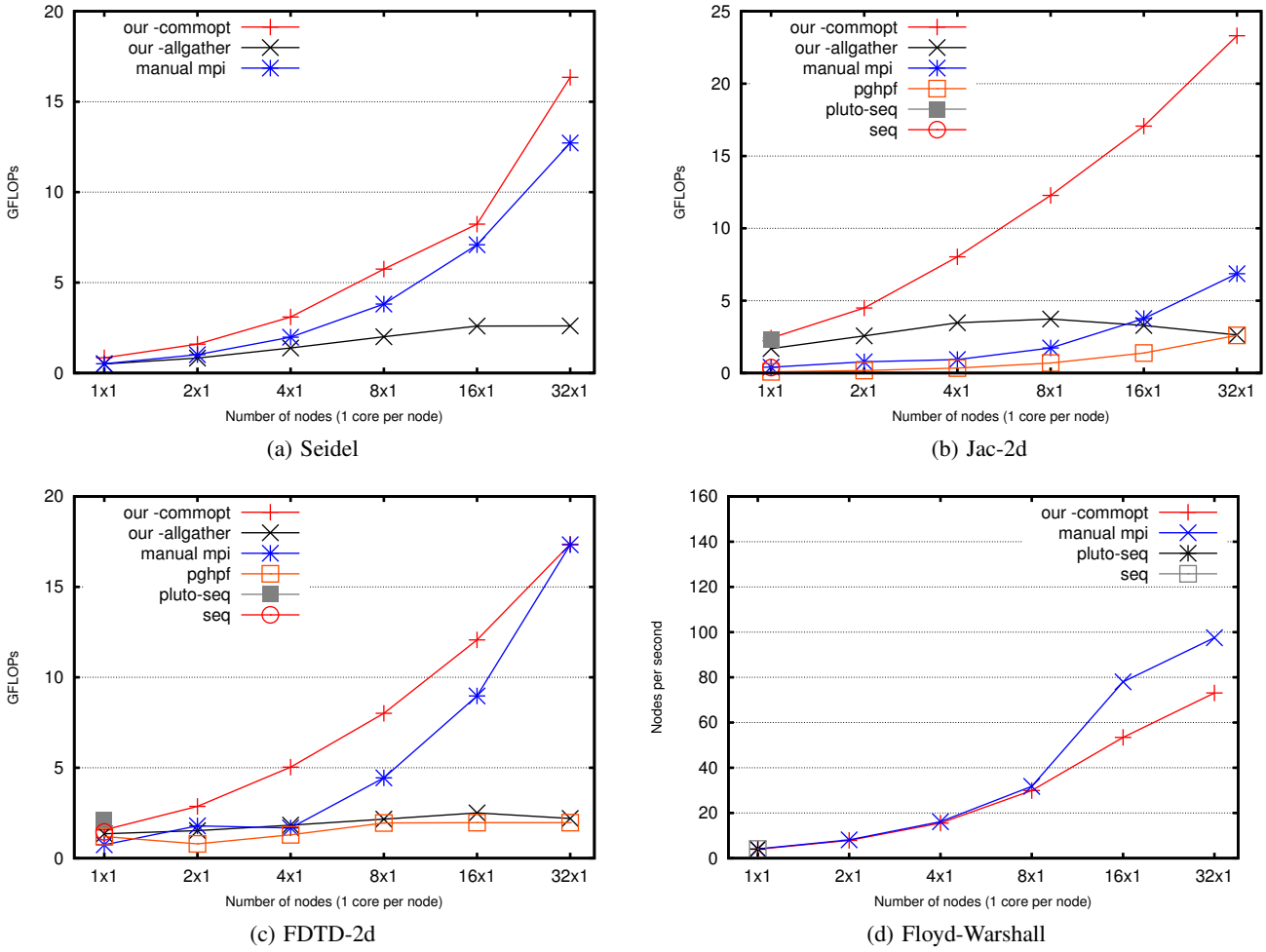
Benchmarks: We evaluate performance on selected commonly used routines and applications from dense linear algebra and stencil computations. 2d-heat and 3d-heat are part of the Pochoir suite [35]. The rest are from the Polybench suite [36]. The reason for selecting this set of benchmarks is two-fold. (1) A number of benchmarks in polybench exhibit communication-free parallelism – though manual distributed-memory parallelization of these is still difficult due to the need for locality optimization in conjunction, handling non-rectangular iteration spaces in some cases, and aggregating results – all of them would show a similar pattern. Hence, most benchmarks chosen were naturally those that had no communication-free parallelism. (2) The chosen benchmarks exhibit different communication patterns on parallelization. Stencils require near-neighbor communication while floyd-warshall and lu require broadcast and multicast style communication. All computations use double precision floating point operations. Problem sizes used are given in Table 1. All results are with strong scaling.

Comparison: Regarding experimental comparison with previous approaches, we were unable to find a publicly available system that could perform such code generation. A number of techniques from the literature only a address part of the problem, and rebuilding an end-to-end system with them is unfeasible. We believe that the detailed discussion provided in Section 5.6 and related work demonstrates our contributions. Comparison is thus provided with manually parallelized MPI versions of these codes, and with HPF where possible. Though each node has eight cores and our tool is able to generate MPI+OpenMP code, in order to focus on the distributed-memory part, we run only one OpenMP thread per process, and one MPI process per node. In these figures, *seq* refers to original code compiled with *icc* with flags mentioned earlier. *our-commopt* refers to our tool with the optimization described in Section 5. *our-allgather* refers to the naive all-to-all broadcast-based communication scheme described in Section 4.3 – this is reported only to support the importance of determining communication partners precisely. *manual-mpi* refers to hand-parallelized MPI version of codes we developed. In the case of `floyd-warshall`, the manual code was the best one selected from submissions for a course assignment part of an advanced parallel programming course taken by 12 students. The students were given two months to develop the code.

For the first four codes that exhibit outer parallelism, the only communication that occurs is that of write-out sets. We see close to ideal speedup for these. Results with *our-allgather* and *manual-mpi* are not shown since they would yield the same performance. Due to all of these codes involving non-rectangular iteration spaces, manual parallelization still involves significant effort. *pghpf* was unable to correctly compile HPF versions of these – further experimentation revealed that non-rectangularity was the most likely cause.

Figure 6 show GFLOPs performance and scalability on the cluster for codes that do incur flow-out communication as well. All x-axes are on a logarithmic scale. Table 2 shows the actual execution times and speedup factors. For `seidel`, the original loop nest has no parallel loops. Our approach includes automatic application of such a transformation and then performing distributed memory code generation. With approaches such as HPF, this code cannot be parallelized unless the programmer manually transforms it first before providing additional directives. Performing manual MPI parallelization for it is extremely cumbersome, even without tiling the time loop. Time tiling is almost never done manually for these. As can be seen, automatically generated code performs much better as a result of it being fully tiled (both space and time dimensions) which in turn leads to better locality and a reduced frequency of communication. It realizes a pipelined parallelization of 3-d tiles. The same is also true for `jac-2d` and `fdtd-2d`, improved locality and reduced frequency of communication leads to a better solution. This explanation is also supported by the fact that 'manual-mpi' exhibits super-ideal improvement when going from 16 to 32 processors (for `fdtd-2d`), and in general performs relatively better with higher number of processors – a decrease in working set size hides poor locality for 'manual-mpi'. Manually parallelized code for `jac-2d` performs significantly poorer due to lesser computation per communication call when compared to `fdtd-2d` for example. Our code shows uniformly good scalability throughout. The difference between *seq* and *pluto-seq* is as a result of locality transformations performed by Pluto. Being able to perform distributed memory code generation on input that has complex transformation expressed on it thus a key strength of our tool. For `floyd-warshall`, code we generate performs within 25% of hand-tuned code while running on 32 nodes (Figure 6(d)). Though we achieve optimal communication volume here and so does the manually parallelized code, the manual developed code overlaps computation with communication and pack/unpack.

Figure 7 shows the split between compute time and other overhead, i.e., time spent in communication, in packing to and unpacking from communication buffers, and in computing the sigma function. Results from our fully optimized codes ('our-commopt') were used for this plot. The sigma function computation itself takes



(a) Seidel (b) Jac-2d

(c) FDTD-2d (d) Floyd-Warshall

Figure 6: Performance of parallelized code on a 32-node cluster

Benchmark	*seq* (icc)	*pluto-seq*	Execution time for *our-commopt* (number of procs)						Speedup: *our-commopt-32* over	
			1	2	4	8	16	32	*seq*	*our-commopt-1*
strmm	30.4m	247s	240s	124.6s	63.5s	33.6s	17.3s	9.4s	194	26.3
trmm	35.5m	91.8s	96.4s	51.3s	27.4s	15.3s	7.14s	3.74s	570	24.5
dsyr2k	127s	39s	38.8s	22.4s	13.5s	6.80s	3.80s	1.57s	80.8	24.7
covcol	462s	30.9s	30.7s	16.7s	8.8s	4.60s	2.48s	1.30s	355	23.8
seidel	17.3m	643.5s	692s	338.7s	174.3s	94s	65.6s	33.0s	31.0	20.8
jac-2d	21.9m	206.7s	218s	111.2s	62.3s	40.7s	29.3s	21.5s	61.3	9.6
fdtd-2d	139s	129.7s	95.2s	70.7s	40.3s	25.3s	16.8s	11.7s	11.9	11.0
2d-heat	19m	266s	280s	157s	81s	52s	33s	24.0s	47.5	11.7
3d-heat	590.6s	222s	236s	118s	68.7s	41.5s	26.3s	18.8s	31.4	12.6
lu	82.9s	28s	29.5s	18.8s	9.28s	5.67s	4.3s	3.9s	21.3	7.56
floyd-warshall	2012s	2012s	2062s	1041s	527s	273s	153s	112s	18.0	18.0

Table 2: Summary of performance: execution times and improvement factors

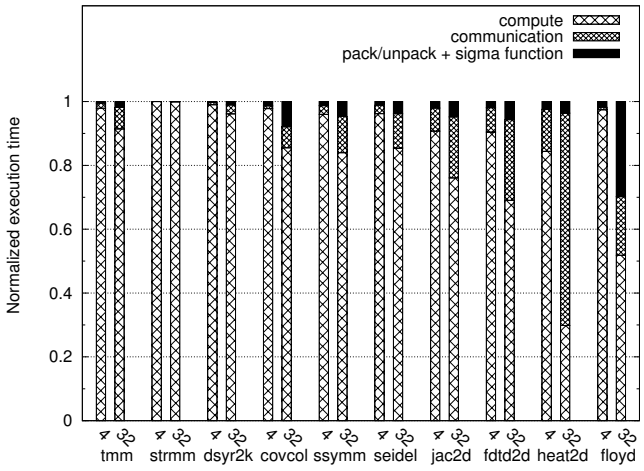


Figure 7: Breakdown of running times of parallelized codes on 4 and 32 processors (*our-commopt-4*, *our-commopt-32*)

much less time than packing and unpacking. The graph sheds further light on the cause for less than ideal scaling where seen. Since the presented results were for strong scaling, even an optimally efficient oracle scheme will be communication bound for a sufficiently large number of processors, given a problem size. Also, InfiniBand interconnect hardware used by our cluster is much older than the latest available. For example, MVAPICH2 1.8.1 [29] reports 1.2 μs of point-to-point latency, 12.6 billion B/s unidirectional, and 20.5 billion B/s bidirectional bandwidth for the latest InfiniBand hardware; these are about three times better than those for our interconnect (reported under Setup), and will cut down our communication time to a third in all cases.

We do not discuss the optimality of the communication code, in terms of volume or otherwise, in this paper. It is possible that a better partitioning of the flow-out sets leads to better communication. Studying this along with the question as to in which cases (dependence patterns) compile-time optimality could be achieved for criteria such as communication volume, given a particular execution order, is the subject of future work.

7. RELATED WORK

Several attempts have been made at achieving distributed memory parallelization. Most works [4, 2, 3, 34, 5, 38, 17] addressed the problem in a limited way with the following limitations: (1) applicable to restricted input such as perfectly nested loops with uniform dependences (only near-neighbor communication), (2) address only a few steps of the actual parallelization and code generation problem, (3) lead to redundant communication with a symbolic number of processors or problem sizes (Section 5.6).

Researchers have looked at the steps of data decomposition and computation decomposition while addressing distributed memory compilation [33, 3, 20]. Computation transformation approaches in the polyhedral framework have themselves undergone advances through [16, 24, 23, 9] that result in better parallelization for shared memory. Note that the affine partitioning works related to SUIF [24, 23] do not address distributed-memory code generation – they are transformation and parallelization algorithms. The Pluto scheduling algorithm has been shown to be a significant improvement on those [9, 10], and we use it to apply parallelizing and locality enhancing transformations before communication code generation techniques described in this paper are applied. As we have shown, generating efficient communication code on top of any automatic transformation algorithm involves a number of non-trivial problems. Without a good scheme, even the best computation partitioning is unlikely to provide good parallel speedup.

Since the contributions of this paper are on communication code generation as opposed to computation or data transformations, the closely related works from literature are those of Amarasinghe and Lam [2], Adve and Mellor-Crummey [1], and Classen and Griebl [11]. [2] handled only perfectly nested loops, while [1] and [11] are based on the polyhedral framework. As explained in Section 5.6, all of these works result in a significantly large amount of redundant communication than ours, in particular with parametric problem sizes and number of processors. However, dHPF [27] implements a number of optimizations (such as multipartitioning [14]) that are useful for any distributed-memory compilation system. Our system does not implement such an allocation scheme yet, but can do so. The discussion at the end of Section 5.1 provides this evidence.

Griebl [18] provides a discussion on distributed-memory parallelization using the polyhedral framework. The work proposes a technique for scheduling and allocation keeping distributed memory architectures in mind. However, communication code generation is not discussed.

Works that translate OpenMP to MPI address a subset of problems that we addressed. The latest among them is [22]. Unlike our work, it is restricted to a subset of affine loop nests that transfer the same set of data every invocation of the parallel loop, and communication set construction is primarily done at runtime. In addition, with OpenMP to MPI approaches, one may have to provide an optimized/transformed OpenMP code to get good performance, adding significant complexity to input taken in by such systems. A future comparison with it if available will be interesting.

Baskaran et al. [7] presented a compiler-assisted dynamic scheduling scheme that constructs and schedules the inter-tile dependence graph on a multicore. Our communication code optimization scheme in Section 5.1 can be viewed as a compiler-assisted scheme to determine communication partners at runtime. Kim et al. [21] present automatic pipelined parallelization for distributed memory with speculation. Their scheme is completely orthogonal to ours in the kind of codes it is applicable to and beneficial for, and the way parallelism is extracted. The R-STREAM compiler provides some support for distributed memory execution [26]. However, due to its reliance on PGAS as its target instead of a message passing one, it does not have to deal with communication code generation.

Recent concurrent work of Dathathri et al. [15] is based on concepts introduced here – usage of flow-out sets and determining communication partners using sigma and pi functions. It proposes refinements to the scheme presented here and reports improved performance on heterogeneous systems comprising CPU/multi-GPU systems as well as distributed-memory clusters.

8. CONCLUSIONS

We presented techniques and optimizations for translation of sequential affine loop nests to code suitable for execution on distributed-memory parallel architectures. Communication code generation and optimizations to minimize associated overhead were the key problems addressed. The scheme we proposed constructs communication sets while completely relying on data dependences. Helper routines generated by the compiler by scanning dependence relations and evaluation of those routines at runtime provided an efficient way to determine communication partners in the presence of symbolic problem sizes or number of processors, and for arbitrary allocations. These techniques were developed within a polyhedral abstraction of the input program allowing sequences of complex

transformations to be automatically applied before code was generated. We have implemented them in a source-to-source transformation tool for an end-to-end fully automatic application. Experiments conducted on a 32-node InfiniBand cluster demonstrated good results. In some cases, performance of our automatically generated code exceeded what could be achieved manually, while in another case it was close to that achieved manually with a significant development effort. A beta release of our tool is publicly available at [30].

ACKNOWLEDGMENTS

I would like to acknowledge the Department of Science and Technology (DST), India for a grant to Computer Science and Automation, Indian Institute of Science (IISc) under the DST FIST program. Most of the infrastructure required for this work was acquired through this grant. I would like to thank the reviewers of SC 2013 very much for their detailed and insightful reviews. I would also like to thank Roshan Dathathri and Chandan Reddy from IISc for their comments.

9. REFERENCES

[1] V. S. Adve and J. M. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 186–198, 1998.

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 126–138, 1993.

[3] J. Anderson, S. Amarasinghe, and M. Lam. Data and Computation Transformations for Multiprocessors. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 166–178, July 1995.

[4] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 112–125, 1993.

[5] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, 1995.

[6] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

[7] M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 219–228, 2009.

[8] C. Bastoul. Clan: The Chunky Loop Analyzer. Documentation.

[9] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (ETAPS CC)*, pages 138–146, Apr. 2008.

[10] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 101–113, June 2008.

[11] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2006.

[12] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[13] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing (ICS)*, pages 151–160, June 2005.

[14] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *Journal of Parallel and Distributed Computing*, 63:887–911, Sep 2003.

[15] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *International conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2013.

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[17] G. I. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Message-passing code generation for non-rectangular tiling transformations. *Parallel Computing*, 32(10):711–732, 2006.

[18] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation thesis.

[19] A. Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *International conference on Compiler Construction (CC)*, pages 236–250, 2009.

[20] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[21] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2010.

[22] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff. A hybrid approach of OpenMP for clusters. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 75–84, 2012.

[23] A. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM International conference on Supercomputing (ICS)*, pages 228–237, 1999.

[24] A. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[25] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2001.

[26] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for

pgas platforms with the R-Stream compiler. In *Workshop on Asynchrony in the PGAS Programming Model*, 2009.

[27] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarria-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the Rice dHPF compiler. *Concurrency: Practice and Experience*, pages 741–767, 2002.

[28] MPI: A Message-Passing Interface Standard - version 2.2. http://www.mpi-forum.org/docs/.

[29] MVAPICH: MPI over InfiniBand, 10 GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu.

[30] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[31] PolyLib - A library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.

[32] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Supercomputing (SC'10)*, New Orleans, LA, Nov. 2010.

[33] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, 1991.

[34] P. Tang and J. N. Zigman. Reducing data communication overhead for doacross loop nests. In *International conference on Supercomputing (ICS)*, pages 44–53, 1994.

[35] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 117–128, 2011.

[36] Polybench. http://polybench.sourceforge.net.

[37] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[38] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.