Revisiting Communication Code Generation Algorithms for
Message-passing Systems*

Clayton Ferner†

December 23, 2006

Abstract

In this paper, we investigate algorithms for generating communication code to run on distributed-memory systems. We modify algorithms from previously published work and prove that the algorithms produce correct code. We then extend these algorithms to incorporate the mapping of virtual processors to physical processors and prove the correctness of this extension. This technique can reduce the number of interprocessor messages. In the examples that we show, the total number of messages was reduced from $O(N^2)$ to $O(P^2)$, where $N$ is the input size and $P$ is the number of physical processors.

The reason that it is important to revisit communication code generation and to introduce a formal specification of the incorporation of mapping in the communication code generation is so that we can make use of the many scheduling heuristics proposed in the literature. We need a generalized mapping function so that we can apply different mapping and scheduling heuristics proposed in the literature for each input program, therefore improving the average performance.

Keywords—parallelizing compiler, parallel computation, distributed-memory, message-passing, mapping, code generation, affine partitioning

1 Introduction

Distributed-memory parallel computer systems, such as clusters of computers or networks of workstations (NOWs) are becoming increasingly popular. These clusters are inexpensive computer systems capable of high performance parallel computation. The motivation behind these clusters is that they can be built from low cost, off-the-shelf components, which in turn allows computer users to build their own high performance
parallel computer systems at low cost. Furthermore, the distributed-memory model scales better than the shared-memory model [21]. Therefore, it is important that we continue to provide tools to assist users in the development of software that runs efficiently on distributed-memory parallel systems.

Toward that goal, Distributed Shared Memory (DSM) libraries, such as Treadmarks [2] or SAM [22], allow parallel code written for a shared-memory model to be run on a distributed-memory system. The DSM libraries have alleviated the message-passing and scheduling concerns for both the user and the compiler. There are also compilers that provide a global addressing model to the user, such as Olden [6] and Split-C [24]. However, the abstraction created by the DSM library or global addressing compiler is often a source of inefficiency [8,13]. Cox, et al. [8] showed several examples of programs where the compiler-generated message-passing solution outperformed the DSM solution. Thus, DSMs may create a convenient abstraction, but they are not necessarily the most efficient. Instead, it is important that research continue on parallelizing compilers that can generate code that is optimized for the particular architecture upon which it will execute. Specifically, it is important that the research community continue to improve the automatic generation of message-passing code that can run on distributed-memory systems.

Amarasinghe and Lam [1] proposed algorithms to generate communication code for message-passing systems, using both the owner-computes rule and the last write tree [1,18]. These algorithms create virtual processors (i.e. task partitions) whose range is unbounded, which must subsequently be mapped to physical processors whose range is bounded. In this paper, we propose an extension to their algorithms to incorporate the mapping of virtual to physical processors for two reasons: first to reduce the number of interprocessor messages and consequently improve performance, and second to generalize the mapping so that different mapping heuristics could be used by the compiler. We do not attempt to solve the partitioning problem nor the mapping problem. Instead, we want to generalize the communication code generation so that we can make use of various techniques to solve these problems.

Unfortunately, while implementing the extension we propose using the algorithms of Amarasinghe and Lam, we discovered that the implementation does not always produce communication code that works correctly. In particular, the ordering in which data are packed in a message by the sending processor is not necessarily the same order in which the receiving processor tries to unpack the data. As a result of this realization, we revisit the communication code generation algorithms proposed by Amarasinghe and Lam. In particular, we have modified the order of the nested loops that perform the communication to insure that the processors will pack and unpack the data in the same order and then prove that this is correct.

By incorporating the mapping of partitions to physical processors as part of the code generation algorithm, we were able to reduce the overall execution time of the example programs by reducing the total number of messages from $O(N^2)$ to $O(P^2)$, where $N$ is the input size and $P$ is the number of physical
processors.

The contributions of this paper are:

1. we give a formal specification of the algorithms for generating message-passing code;
2. we then extend the algorithms to incorporate the mapping of partitions to physical processors, and;
3. we prove the correctness of the algorithms.

The rest of this paper is organized as follows: section 2 gives background information and definitions; section 3 discusses the generation of message-passing code; section 4 describes how virtual processors can be mapped to physical processors as part of the code generation algorithms; section 5 shows some results of programs run on a distributed-memory system using these techniques; and section 6 gives concluding remarks and discusses future work. In addition, the appendix provides some details of the mathematics used in the algorithms.

2 Background

Throughout this paper, we use the notation $\mathbf{v} = [v_1, \ldots, v_n]^T$ to represent a vector, $v_i$ to represent the $i^{th}$ element of the vector $\mathbf{v}$, and $v_{i:j}$ to represent the subvector from the $i^{th}$ through the $j^{th}$ element of $\mathbf{v}$. We also will use the notation $\mathbf{v} = [w, \mathbf{z}]^T$, where $w$ is a scalar and $\mathbf{z}$ is a vector to mean $\mathbf{v} = [w, z_1, \ldots, z_n]^T$ (i.e. $\mathbf{v} \neq [w, [z_1, \ldots, z_n]]^T$). We use the notation $lb_x$ and $ub_x$ to be the lower and upper bounds, respectively, of a variable $x$.

The problem studied in this paper is the parallelization of a loop nest that modifies elements of an array. As is typically done, we assume that the loop nests are count-controlled loops whose lower and upper bounds are affine expressions of symbolic constants (loop invariants) and outer (containing) loop indexes, such as the example shown in Figure 1. We also assume that the array access functions are affine expressions of symbolic constants and outer loop indexes.

\begin{verbatim}
for i1 = 1 to N do
    for i2 = i1+1 to N do
        for i3 = N+1 downto i1 do
            a[i2][i3] = a[i2][i3] - a[i1][i3] * a[i2][i1] / a[i1][i1]
\end{verbatim}

Figure 1: Elimination Phase of Gaussian Elimination
2.1 Definitions

Definition 2.1 An iteration vector \( i = [i_1, \ldots, i_n]^T \in \mathbb{Z}^n \) is a vector of the indexes of \( n \) nested loops. An iteration instance of an iteration vector is an instance of the loop nest.

Definition 2.2 An array reference is represented as a tuple \( a = (X, l, F, \omega, e) \) where:

- \( X \) is the name of the array
- \( l \) is the number of subscripts
- \( F(i) = Fi + f \) is an affine expression called an access function which maps an iteration instance \( i \) to an element of the array, where \( F \) is an \( l \times n \) matrix of symbolic constants and \( f \) is a vector of \( l \) symbolic constants
- \( \omega \) is a boolean, which is set to true iff the array reference is a write operation (i.e. appears on the lhs of an assignment operator)
- \( e \) is the element size in bytes

Definition 2.3 The loop bounds of a loop nest \( s \) are given by the affine expression \( D_s(i) = D_s i + d_s \), where \( i \) is an iteration instance, \( D_s \) is a \( 2n \times n \) matrix of symbolic constants and \( d_s \) is a vector of \( 2n \) symbolic constants.

The iteration instance \( i \) is a valid iteration instance for loop nest \( s \) iff \( D_s(i) \geq 0 \), where \( 0 \) is a vector of \( 2n \) zeros. The loop bounds for the loop nest in Figure 1 are shown in equation (3) of the appendix.

Definition 2.4 The lexicographically less than operator \( < \) is defined recursively for two iteration vectors \( i, i' \in \mathbb{Z}^n \) such that \( i < i' \) iff

\[
i_1 < i'_1 \lor (i_1 = i'_1 \land i_{2:n} < i'_{2:n}) .
\]

The lexicographical operator provides a total ordering of the iteration instances of a loop nest such that \( i < i' \) iff iteration \( i \) is executed prior to iteration \( i' \) when executed sequentially on a single processor.

2.2 Affine Partitioning and Parallel Code Generation

Lim and Lam [17] developed a technique to determine the computation decomposition (or partitioning) that provides the coarsest granularity of parallelism for a given order of communication. It is claimed in [17] that their partitioning algorithm “... subsumes previously proposed loop transformation algorithms that are based on unimodular transformations, loop distribution, fusion, scaling, reindexing and statement reordering.” This
partitioning determines how the iterations of a loop nest will be divided into individual tasks, which can be executed in parallel.

**Definition 2.5** An affine partitioning \( \Phi_s(i) = C_s i + c_s \), where \( C_s \) is an \( 1 \times n \) matrix of symbolic constants and \( c_s \) is a scalar symbolic constant, is the mapping of an iteration instance of a loop nest \( s \) to a partition number.

Partition numbers are not necessarily positive, and their range may be arbitrarily large. We refer to these partitions as virtual processors, since each partition could potentially be executed by a different processor, if an unlimited number of processors were available. An example of an affine partitioning is shown in equation (4) of the appendix for the loop nest in Figure 1.

Once an affine partitioning has been derived, it determines which virtual processors will execute each iteration instance. A system of constraints \( A = \{(D_s(i) \geq 0) \cup (p = \Phi_s(i))\} \) is built from the loop bounds and the partitioning. (An equality constraint, such as \( p = \Phi_s(i) \), can be rewritten as two inequality constraints: \( p \geq \Phi_s(i) \land p \leq \Phi_s(i) \).) An example of this system of constraints is shown in equation (5) of the appendix. The system also defines a polyhedra in \( n \)-space. The order of the unknowns is important, since it determines the order of the nesting of the resulting loops. The virtual processor \( p \) should be the first (outermost loop), because this loop will be transformed to an `if` statement.

We do not discuss nor attempt to solve the problem of determining the best partitioning; instead, the reader is referred to [17]. However, once an affine partitioning is determined, the SPMD (Single Program Multiple Data) code to execute the parallel version of the loop nest \( s \) can be generated using an algorithm based on Fourier-Motzkin elimination (FME) [3]. Equations (4)-(8) of the appendix show the progression of FME as it builds the transformed loop nest shown in Example 1 of the appendix. We also refer the interested reader to [3, 4, 14, 25, 26] for a thorough discussion of FME and how it is used to build the loop nest from a system of constraints. Other methods exist to generate the loop nests from polyhedra such as [20]. However, we do not address the problem of transforming a system of constraints to a loop nest. Instead, we are more interested in how the system is built and the ordering of the loops that are generated. Once the loop nest has been created from the system of constraints, the \( p \) loop is then changed to an `if` statement since each processor will be responsible for a single iteration of that loop. We can also replace degenerate loops with single assignment statements resulting in the parallelized loop shown in Figure 2.

We make the assumption in this paper that a loop nest created from a system of constraints using FME will execute an iteration instance iff that iteration instance is a solution to the system. Xue [26] actually showed that, for integer solutions, FME is not exact. One can contrive an example where the above assumption is invalid. However, FME is widely used as an effective algorithm for generating loop nests from
if 2 <= p AND p <= N then
   for i1 = 1 to p-1 do begin
      i2 = p
   for i3 = N+1 downto i1 do
      a[i2][i3] = a[i2][i3] - a[i1][i3] * a[i2][i1] / a[i1][i1]
   end

Figure 2: Resulting Parallel Loop Nest for Gaussian Elimination Using the Partitioning \( p = \Phi_s(i) = i_2 \)

a system of constraints for real applications.

2.3 Last Write Tree

In [18], Maydan, Amarasinghe, and Lam developed the concept of a last write tree, which is a value-centric approach to data dependencies as opposed to a location-centric approach. A last write tree is a mapping from an iteration which reads a value of an array to the exact iteration which produced the value. This approach is useful for message-passing because it allows for parallelism to be determined by a computation decomposition instead of a data decomposition.

**Definition 2.6** A last write data dependence \( L_{a,w,r} : \mathbb{Z}^n \rightarrow \mathbb{Z}^n \) is a mapping from a read array access \( a_r \) and iteration instance \( i_r \) to the write array access \( a_w \) and iteration instance \( i_w \) that produced the value required. That is, given two array references \( a_r, a_w \) of a loop nest \( s \) with access functions \( F_{a_r}(i_r) \) and \( F_{a_w}(i_w) \), respectively, such that \( \omega_{a_r} = false \) and \( \omega_{a_w} = true \), then \( L_{a,w,r}(i_r) = i_w \) iff

\[
D_s(i_w) \geq 0 \quad \land \quad D_s(i_r) \geq 0 \quad \land \quad F_{a_w}(i_w) = F_{a_w}(i_r) \quad \land \quad i_w < i_r \quad \land \\
\left( \exists i_{w}' \text{ such that } D_s(i_{w}') \geq 0 \quad \land \quad F_{a_w}(i_{w}') = F_{a_w}(i_r) \quad \land \quad i_w < i_{w}' < i_r \right).
\]

This definition states that for \( i_w \) to be the last write iteration for \( i_r \), both must be valid iterations, the array elements that are referenced must be the same, \( i_w \) must execute prior to \( i_r \), and there cannot be another iteration between \( i_r \) and \( i_w \) that also modifies the same array element. Notice that the last write data dependence implies that a read array access has no more than one write access from which it gets its value. On the other hand, a write access may have many read accesses that use its value. Equation (9) of the appendix shows the last write dependence between the \( a[i2][i3] \) array access on the lhs and the \( a[i1][i3] \) array access on the rhs of the program example in Figure 1.

---

6
2.4 Building the System

In order to use the data dependence information to send and receive dependent data, we create variables for both the receiving as well as the sending processors and iteration instances: \( p_r, p_w, i_r, \) and \( i_w \). The system of constraints should include the loop bounds \( D_s(i_r) \geq 0 \) and \( D_s(i_w) \geq 0 \), the partitioning \( p_r = \Phi_s(i_r) \) and \( p_w = \Phi_s(i_w) \), the last write dependence (which ties the receive and send variables together) \( i_w = L_{a_{w,s}}(i_r) \), and finally the constraint that \( p_r \neq p_w \). This information determines the valid iterations for which there is a data dependence and for which that data is non-local. Therefore, communication is required between virtual processors \( p_r \) and \( p_w \). Equation (10) of the appendix shows the entire system of constraints for the example in Figure 1.

Note that the constraint \( p_r \neq p_w \) cannot be written in a form that is consistent with a system of inequalities, where all constraints must be satisfied. Instead, we can rewrite it as \( p_r \leq p_w - 1 \lor p_r \geq p_w + 1 \). Since this requires the \( \lor \) operator, we have to create two systems, one with each of these new constraints, and solve both. This can produce two sets of communication code for each dependency, although we have found that one of these sets of constraints will usually be inconsistent and hence, no communication will be performed. In the discussion below, we will continue to use the constraint \( p_r \neq p_w \), although it actually means there should be two systems; one with each of the constraints: \( p_r \leq p_w - 1 \) and \( p_r \geq p_w + 1 \).

In order to create the code to perform communication, we create two loop nests where the unknowns are ordered \( p_r, p_w, i_r, i_w \) for receiving data and \( p_w, p_r, i_r, i_w \) for sending data. Recall, that the order of the unknowns determines the order in which the loops are generated using FME. These loops will iterate over all pairs of virtual processors and all iterations instances (mapped to those virtual processors) such that there is a data dependence between iteration instance \( i_w \) that produces a value and an iteration instance \( i_r \) that reads a value. Each virtual processor \( p \) will execute the first loop for only those iterations where \( p = p_r \), to receive all the data that it needs to read. Likewise, virtual processor \( p \) will execute the second loop for only those iterations where \( p = p_w \), to send locally computed information to the processors that require it. The bodies of the loop nests are instructions to pack data into and unpack data out of the message buffer.

Consider the elimination phase of Gaussian elimination shown in Figure 1. There is a data dependence from the lhs of the assignment \((a[12][13])\) for iteration instance \( i_w \) to the array reference \((a[11][13])\) for iteration instance \( i_r \) such that \( i_{1w} = i_{1r} - 1, \) \( i_{2w} = i_{1r} \), and \( i_{3w} = i_{3r} \). After building the system (also see equation (10) of the appendix) and using FME, the resulting communication loop nests are shown in Figure 3. Many of the loops are degenerate and have been replaced by single assignment statements. The parallelized loop nest of Figure 2 is then inserted between the receive loop nest in Figure 3(a) and the send loop nest in Figure 3(b).
\[ px = p \]
\[ if \ 3 \leq px AND px \leq N \ then \]
\[ for \ px = 2 \ to \ px-1 \ do \ begin \]
\[ receive \ from \ px \]
\[ i1r = px \]
\[ i2r = px \]
\[ for \ i3r = px \ to \ N \ do \ begin \]
\[ i1w = px-1 \]
\[ i2w = i1w+1 \]
\[ i3w = i3r \]
\[ unpack \ a[i1x][i3r] \]
\[ end \]
\[ send \ to \ px \]
\[ end \]
\[ (a) \ The \ receive \ loop \ nest \]
\[ pw = p \]
\[ if \ 2 \leq pw AND pw \leq N-1 \ then \]
\[ for \ px = pw+1 \ to \ N \ do \ begin \]
\[ i1r = pw \]
\[ i2r = px \]
\[ for \ i3r = pw \ to \ N+1 \ do \ begin \]
\[ i1w = pw-1 \]
\[ i2w = i1w+1 \]
\[ i3w = i3r \]
\[ pack \ a[i2w][i3w] \]
\[ end \]
\[ send \ to \ px \]
\[ end \]
\[ (b) \ The \ send \ loop \ nest \]

Figure 3: Communication loop nests for the dependency \( a[i2][i3] \) to \( a[i1][i3] \) in the loop nest shown in Figure 1

The technique described above for producing communication code will create a program that must first receive all of its data before executing any computations, and then performs all of its computations before sending any data. The resulting program may very well execute sequentially and could run slower than the single-processor sequential execution. However, in previous work [9], we presented a technique to overlap the communication with computation. Here, we are primarily interested in generating a correct program.

3 Communication Loop Nest Generation

This is where the contribution of this paper begins. First we revisit the communication algorithms proposed by Amarasinghe and Lam [1]. We formalize the algorithms with a slight modification to the ordering of the unknowns. Second, we prove the correctness of the new algorithms. Then, in section 4, we extend the communication algorithms to incorporate a mapping of virtual to physical processors and prove the correctness of this extension.

3.1 Communication Code Generation Algorithms

Algorithms 1 and 2 show how the two loop nests are created to receive and send messages, respectively, for a particular data dependence, given the system of constraints and the order of the unknowns. Notice that the outermost loop is actually used as the current processor id. Each processor is only interested in participating in communication where it is either the sending processor or one of the receiving processors. It does not need
**Algorithm 1** Receive Data Loop Nest Generation

**Input:**

- A system of constraints $A$ that describes the processors and iterations that need to communicate
- Vector $\Psi = [p_r, p_w, i_r, i_w]^T$ which contains the unknowns of $A$. The order of the unknowns in $\Psi$ will be the order of the loop nests. The first unknown should be $p_r$ and is used as the current processor id.
- The unknown $p_w$ to be used as the sending processor
- The array access functions $\mathcal{F}_{a_r}(i)$ and $\mathcal{F}_{a_w}(i)$

**Output:** A loop nest for each processor to receive the value of array reference $a_r$ from the processor that computes it

**Steps:**

1. Create an instruction to unpack $X_{a_r}[\mathcal{F}_{a_r}(i_r)]$
2. Use FME to build a loop nest from $A$ and $\Psi$ from the inside out with the body being the instruction from step 1 until the variable $p_w$ is reached
3. For the $p_w$ loop, create the loop with a body that consists of:
   - (a) A receive instruction to receive a message from processor $p_w$
   - (b) The loop nest from step 2
4. Continue using FME to build a loop nest from the remaining variables with the body from step 3 until all variables have been eliminated
5. Finally, convert the outermost loop “for $p_r = l b_{p_r}$ to $u b_{p_r}$ do begin ... end” to

$$p_r = p$$

if $l b_{p_r} \leq p_r$ AND $p_r \leq u b_{p_r}$ then begin ... end
Algorithm 2 Send Data Loop Nest Generation

Input:
- a system of constraints $A$ that describes the processors and iterations that need to communicate
- vector $\Psi = [p_w, p_r, i_r, i_w]^T$ which contains the unknowns of $A$. The order of the unknowns in $\Psi$ will be the order of the loop nests. The first unknown should be $p_w$ and is used as the current processor id.
- the unknown $p_r$ to be used as the receiving processor
- the array access functions $F_{ar}(i)$ and $F_{aw}(i)$

Output: A loop nest for each processor to send the value of array reference $a_w$ to the processors that need it

Steps:
1. Create an instruction to pack $X_{aw}[F_{aw}(i_w)]$
2. Use FME to build a loop nest from $A$ and $\Psi$ from the inside out with the body being the instruction from step 1 until the variable $p_r$ is reached
3. For the $p_r$ loop, create the loop with a body that consists of:
   (a) The loop nest from step 2
   (b) A send instruction to send a message to processor $p_r$
4. Continue using FME to build a loop nest from the remaining variables with the body from step 3 until all variables have been eliminated
5. Finally, convert the outermost loop “for $p_w = lb_{p_w}$ to $ub_{p_w}$ do begin ... end” to

\[
p_w = p \\
\text{if } lb_{p_w} \leq p_w \text{ AND } p_w \leq ub_{p_w} \text{ then begin ... end}
\]
Algorithm 3 Create Communication Loop Nests

**Input:**
- Two array references $a_r$ and $a_w$ contained within a loop nest $s$ for the same array $X$ such that $\omega_{a_r} = \text{false}$, $\omega_{a_w} = \text{true}$
- The vector of loop indexes to be used for receiving $i_r$ and the vector of loop indexes to be used for sending $i_w$
- The variable used for the receiving processor $p_r$ and the variable used for the sending processor $p_w$
- The loop bounds $D_s(i)$ for the instruction $s$
- The last write data dependence mapping $L_{a_w a_r}(i)$
- The array access functions $F_{a_r}(i)$ and $F_{a_w}(i)$
- The affine partitioning $\Phi_s(i)$

**Output:**
A loop nest for each processor to receive the value of array reference $a_r$ from the processor that computes it and a loop nest for each processor to send the value of array reference $a_w$ to the processors that need it.

**Steps:**
1. Create the system of constraints:
   \[
   A = \begin{cases}
   D_s(i_r) \geq \vec{0} & \cup & D_s(i_w) \geq \vec{0} & \cup & \Phi_s(i_r) = p_r & \cup & \Phi_s(i_w) = p_w & \cup & i_w = L_{a_w a_r}(i_r) & \cup & p_w \neq p_r \\
   \end{cases}
   \]
2. If $A$ does not have a solution, then stop (no communication is necessary).
3. Use Algorithm 1 with inputs: $A$, $\Psi_r = [p_r, p_w, i_r, i_w]^T$, $p_w$, $F_{a_r}(i_r)$, $F_{a_w}(i_w)$ to create a receive loop nest.
4. Use Algorithm 2 with inputs: $A$, $\Psi_w = [p_w, p_r, i_r, i_w]^T$, $p_r$, $F_{a_r}(i_r)$, $F_{a_w}(i_w)$ to create a send loop nest.
5. Surround the parallelized loop nest $s$ with the receive loop nest and send loop nest.

To participate in communication between other processors. In addition, notice the complementary processing in step 3 of Algorithms 1 and 2. In Algorithm 1 step 3, data are received then unpacked. In Algorithm 2 step 3, data are packed then sent.

Next, Algorithm 3 shows how to take a data dependence and build the system to be used by the first two algorithms. Notice that the order of the unknowns is the same, with the exception that $p_r$ and $p_w$ are reversed, in steps 3 and 4.

### 3.2 Proof of Correctness

In this section, we prove in Theorem 3.2 that Algorithm 3 (and subsequently Algorithms 1 and 2) will correctly exchange dependent data between virtual processors. The first part of the proof is that the processors will send messages to and receive messages from the correct processors. This is a fairly straightforward part of the proof because of the way in which the system of constraints is built.

The second part of the proof is more important. That is, when a processor packs data into the buffer to send to another processor, the receiving processor must unpack the data in the correct order. We can think of the message buffer as a single dimensional array. Then to prove that the sending and receiving processors...
will pack and unpack in the same order, we need to show that any given array element in the buffer has the same offset from the point of view of either processor. This brings us to the following definition:

**Definition 3.1** Let \( B(i, A, e) \) be a mapping from an iteration instance \( i \) for a receive or send loop nest that unpacks or packs an array element to its offset within the message. That is,

\[
B(i, A, e) = \left( \sum_{j=1}^{n} (i_j - lb_{i_j}) \cdot \prod_{k=j+1}^{n} (ub_{i_k} - lb_{i_k} + 1) \right) \cdot e
\]

where \( e \) is the element size and \( lb_{i_j} \) and \( ub_{i_j} \) are the lower and upper bounds of the variables \( i_j \), respectively, derived from \( A \).

The function \( B \) is essentially the mapping of a multi-dimensional array to a single-dimensional array, where each loop index is considered another dimension.

**Theorem 3.2** Let \( a_r, a_w \) be two array references for the same array \( X \) within a loop nest \( s \) such that \( \omega_{a_r} = \text{false} \) and \( \omega_{a_w} = \text{true} \). Also let \( i'_r \) and \( i'_w \) be iteration instances of instruction \( s \) such that \( D_s(i'_r) \geq 0, D_s(i'_w) \geq 0 \), \( i'_w = L_{a_w,a_r}(i'_r), p'_r = \Phi_s(i'_r), \) and \( p'_w = \Phi_s(i'_w) \). If \( p'_r \neq p'_w \), then Algorithm 3 will generate loop nests such that

- processor \( p'_w \) sends a message containing \( X[F_{a_w}(i'_w)] \) at offset \( b_w \) to processor \( p'_r \),
- processor \( p'_r \) receives a message containing \( X[F_{a_r}(i'_r)] \) at offset \( b_r \) from processor \( p'_w \), and
- \( F_{a_w}(i'_w) = F_{a_r}(i'_r) \) and \( b_w = b_r \).

**Proof:** Algorithm 1, using FME, will generate a loop nest such that processor \( p'_r \) will receive a packet from processor \( p'_w \) and unpack the value \( X[F_{a_w}(i'_w)] \) because \( p_r = p'_r, p_w = p'_w, i_r = i'_r, \) and \( i_w = i'_w \) is a solution to \( A \). Likewise, Algorithm 2 will generate a loop nest such that processor \( p'_w \) will pack the value \( X[F_{a_w}(i'_w)] \) and send the packet to processor \( p'_r \), again because \( p_r = p'_r, p_w = p'_w, i_r = i'_r, \) and \( i_w = i'_w \) is a solution to \( A \). Also, \( i'_w = L_{a_w,a_r}(i'_r) \) implies that \( F_{a_w}(i'_w) = F_{a_r}(i'_r) \).

The offset used to unpack the value \( X[F_{a_r}(i'_r)] \) from the message received by \( p'_r \) is the number of iterations of the loops created in step 2 of Algorithm 1, since each new message will have a new offset starting at zero. Therefore, \( b_r = B([i'_r, i'_w]^T, A, e_{a_r}) \). Likewise, the offset used to pack the value \( X[F_{a_w}(i'_w)] \) in the message sent by \( p'_w \) is the number of iterations of the loops created in step 2 of Algorithm 2. Therefore, \( b_w = B([i'_r, i'_w]^T, A, e_{a_w}) \). Since \( a_r \) and \( a_w \) refer to the same array, the element size is the same. Also, since the order of the unknowns of \( i'_r \) and \( i'_w \) in \( \Psi_r \) and
Ψ_w are the same for both algorithms, the upper and lower bounds of the vector \([i'_r, i'_w]^T\) will be equal. Therefore, \(b_w = b_r\). ■

Notice from the proof of Theorem 3.2 that to insure that \(b_w = b_r\), we need the same vector \([i'_r, i'_w]^T\) for the computation of the offsets. This requires that Algorithm 3 steps 3 and 4 use the same ordering of the loop indexes that follow \(p_w\) and \(p_r\). Amarasinghe and Lam, in their algorithm, ordered the unknowns \(p_r, i_r, p_w, i_w\) and \(p_w, i_w, p_r, i_r\) in the receive and send loop nests, respectively. This ordering is different for the receive side than it is for the send side, which may produce a different offset. It is not clear if and when \(B(i_w, A, e_{a_w}) = B(i_r, A, e_{a_r})\).

We made this modification to their algorithm so that we could prove that the offsets of the same array element would be the same for both processors. In fact, after we extend the algorithm in the next section, the ordering of the unknowns proposed by Amarasinghe and Lam will produce loop nests such that the data are packed in a different order than they are unpacked. We will show an example of this incorrect ordering in the next section. On the other hand, using the ordering that we propose in Algorithm 3, the data are guaranteed to be packed and unpacked in the same order.

3.2.1 Complexity of the Algorithm

The complexity of Algorithm 3, given the inputs, is the complexity of Fourier-Motzkin Elimination. Steps 2 through 4 of Algorithms 1 and 2 perform the FME. Kessler [14] established the worst case complexity of FME to be:

\[
O\left(\sum_{r=0}^{j-1} (j - r) \left(\frac{k^2 r}{d(j - r)}\right)^2\right)
\]

(1)

where \(j\) is the number of unknowns and \(k\) is the number of constraints. In the worst case scenario, the number of constraints of the system \(A\) can double as each variable is eliminated, if the number of lower bounds and the number of upper bounds on that variable is approximate \(j/2\). However, Kessler argued that the average run time should be considerable lower for two reasons:

1. The probability that the number of lower bounds and the number of upper bounds on a variable are \(j/2\) at each step in the process is rather small.

2. A sparse matrix of coefficients will not generate many new constraints since only non-negative coefficients generate new constraints. As more variables are eliminated from the system the matrix becomes more sparse.

For Algorithm 3, the number of unknowns is \(j = 2n + 2\), where \(n\) is number of nested loops. The number of constraints is \(k = 6n + 5\): the lower and upper bounds on the loop variables \(D_s(i_r) \geq \bar{s}\) and \(D_s(i_w) \geq \bar{s}\) each
represent 2n constraints; the partitioning \( p_r = \Phi_s(i_r) \) and \( p_w = \Phi_s(i_w) \) each represent 2 constraints; the LWT \( i_w = L_{a_{w,c}}(i_r) \) represents 2n constraints; and the requirement that \( p_w \neq p_r \) represents 1 constraint.

Step 1 of Algorithm 3 is \( O(jk) \). Step 2 of Algorithm 3 does not add to the complexity since it is actually a result of FME routine. The routine either produces a solution, if it exists, or a failure is there is no solution. Step 5 of Algorithm 3 and steps 1 and 5 of Algorithms 1 and 2 are all constant operations. Therefore, the worst case complexity of Algorithm 3 is equation (1) where \( j = 2n + 2 \) and \( k = 6n + 5 \).

4 Mapping of Partitions to Physical Processors

The technique described in section 3 to generate message-passing code will produce a correct program; however, the performance may be poor. This is primarily due to the fact that messages are sent between pairs of virtual processors instead of physical processors. There can be many messages sent between different pairs of virtual processors, all of which are executed by the same pair of physical processors. A better strategy is to include the mapping of virtual to physical processors as part of the system of constraints from which the message-passing code is generated. This allows the code generator to create single messages between pairs of physical processors, greatly reducing the total number of messages.

4.1 Communication Code Generation Algorithm

To follow this strategy, we formally define a mapping of partitions to physical processors. Essentially, the mapping needs to take a set of integers that have an arbitrary range, and map it to a set of integers with a restricted range. We use the symbolic constant \( P \) to represent the total number of physical processors, which is determined at runtime. We also assume the processors have a unique id in the range \([0..P]\).

**Definition 4.1** A partition-to-physical-processor mapping \( M_s: \mathbb{Z} \to [0..P] \) is a mapping of partitions \( p \in [\text{lb}_p, \text{ub}_p] \) to physical processor ids \( \text{pid} \in [0..P] \).

To use this mapping, we modify our algorithm to include two new unknowns \( \text{pid}_r \) and \( \text{pid}_w \), add constraints for \( \text{pid}_r = M_s(p_r) \) and \( \text{pid}_w = M_s(p_w) \), replace the constraint \( p_r \neq p_w \) with the constraint \( \text{pid}_r \neq \text{pid}_w \), and use \( \text{pid}_r \) and \( \text{pid}_w \) as the variables for sending and receiving messages. Notice that the mapping function adds the implicit constraints \( 0 \leq \text{pid}_r < P \) and \( 0 \leq \text{pid}_w < P \). We can then use
Algorithm 3 to generate the message passing code if we redefine the functions. Let

\[
D_s^*([p, i]^T) = D_s^*[p, i]^T + d_s^* \\
F_{a_s}^*([p, i]^T) = F_{a_s}(i) \\
F_{a_w}^*([p, i]^T) = F_{a_w}(i) \\
L_{a_w, a_r}^*([p, i]^T) = L_{a_w, a_r}(i) \\
\Phi_s^*([p, i]^T) = M_s(\Phi_s(i))
\]

(2)

where,

\[
D_s^* = \begin{bmatrix}
0 & D_s \\
1 & -C_s \\
-1 & C_s
\end{bmatrix}
\quad \text{and} \quad d_s^* = \begin{bmatrix}
d_s \\
-c_s \\
c_s
\end{bmatrix}.
\]

Recall from Definitions 2.3 and 2.5 that \( D_s(i) = D_s i + d_s \) and \( \Phi_s(i) = C_s i + c_s \), where \( D_s \) is a \( 2n \times n \) matrix, \( d_s \) is a \( 2n \) element vector, \( C_s \) is a \( 1 \times n \) matrix, and \( c_s \) is a scalar. The notation \( D_s^* \) does not refer to a matrix with matrices nested within it, but rather refers to a matrix having the elements of \( D_s \) with \( 2n \) zeros in the left column, the elements of \( C_s \) negated with ones in the left column, and the elements of \( C_s \) and negative ones in the left column. The construction of \( D_s^* \) and \( d_s^* \) is such that \( D_s^*([p, i]^T) \geq 0 \) satisfies both the loop bound constraints as well as the partitioning constraints, since:

\[
D_s^*([p, i]^T) \geq 0 \implies D_s^*[p, i]^T + d_s^* \geq 0 \\
\implies (0 \cdot p + D_s i + d_s^* \geq 0) \land (1 \cdot p - C_s i - c_s \geq 0) \land (-1 \cdot p + C_s i + c_s \geq 0) \\
\implies (D_s i + d_s \geq 0) \land (p \geq C_s i + c_s) \land (p \leq C_s i + c_s) \\
\implies (D_s(i) \geq 0) \land (p = \Phi_s(i)).
\]

We will then use Algorithm 3 with the variables \( \text{pid}_r \) and \( \text{pid}_w \) as the processors ids. The virtual processors ids are now simply loop indexes. Algorithm 4 shows how this is accomplished, and Theorem 4.2 proves its correctness.

We are not addressing the problem of how to determine the best mapping function for a program which is NP-hard in general [5]. However, separating the mapping function from the communication loop nest generation is essential for proceeding with the mapping problem. There are many mapping, scheduling, and clustering heuristics in the literature that address the mapping of virtual processors to physical processors. (See [10, 11, 15, 16, 19, 23] for a sample of papers that have published comparisons or surveys of scheduling heuristics.) In order to make use of these heuristics, they need to be able to be inserted easily into the compiler. Generalizing the mapping function, as we have done here, is a step toward that process. The
Algorithm 4: Create Communication Loop Nests

Input:
- Two array references \( a_r \) and \( a_w \) contained within a loop nest \( s \) for the same array \( X \) such that \( \omega_{a_r} = \text{true} \) and \( \omega_{a_w} = \text{false} \)
- The vector of loop indexes to be used for receiving \( i_r \) and the vector of loop indexes to be used for sending \( i_w \)
- The variable used for the receiving virtual processor \( p_r \) and the variable used for the sending virtual processor \( p_w \)
- The variable used for the receiving physical processor \( \text{pid}_r \) and the variable used for the sending physical processor \( \text{pid}_w \)
- The loop bounds \( D_s(i) \) for the instruction \( s \)
- The last write data dependence mapping \( L_{a_w,a_r}(i) \)
- The array access functions \( \mathcal{F}_{a_r}(i) \) and \( \mathcal{F}_{a_w}(i) \)
- The affine partitioning \( \Phi_s(i) \)
- The physical processor mapping \( M_s(p) \)

Output: A loop nest for each processor to receive the value of array reference \( a_r \) from the processor that computes it and a loop nest for each processor to send the value of array reference \( a_w \) to the processors that need it

Steps:
1. Define the functions from equation [2].
2. Use Algorithm 3 with input \( a_r, a_w, [p_r,i_r]^T, [p_w,i_w]^T, \text{pid}_r, \text{pid}_w, D_s(i), L_{a_w,a_r}^s(i), \mathcal{F}_{a_r}^s(i), \mathcal{F}_{a_w}^s(i), \Phi_s(i) \).

next step, which is non-trivial, will be to adapt the heuristics to provide a generic mapping function given a description of the dependencies between virtual processors.

Theorem 4.2  Let \( a_r, a_w \) be two array references for the same array \( X \) within a loop nest \( s \) such that \( \omega_{a_r} = \text{false} \) and \( \omega_{a_w} = \text{true} \). Also let \( \mathbf{i}_r \) and \( \mathbf{i}_w \) be iteration instances of instruction \( s \) such that \( D_s(\mathbf{i}_r) \geq 0 \), \( D_s(\mathbf{i}_w) \geq 0 \), \( \mathbf{i}_w = L_{a_w,a_r}(\mathbf{i}_r) \), \( \text{pid}_r = M_s(p_r = \Phi_s(i_r)) \), and \( \text{pid}_w = M_s(p_w = \Phi_s(i_w)) \). If \( \text{pid}_r \neq \text{pid}_w \), then Algorithm 4 will generate loop nests such that

- processor \( \text{pid}_w \) sends a message containing \( X[\mathcal{F}_{a_w}(\mathbf{i}_w)] \) at offset \( b_w \) to processor \( \text{pid}_r \),
- processor \( \text{pid}_r \) receives a message containing \( X[\mathcal{F}_{a_r}(\mathbf{i}_r)] \) at offset \( b_r \) from processor \( \text{pid}_w \), and
- \( \mathcal{F}_{a_w}(\mathbf{i}_w) = \mathcal{F}_{a_r}(\mathbf{i}_r) \) and \( b_w = b_r \).

Proof: Given the inputs to Algorithm 3 in step 2, we know from Theorem 3.2 that Algorithm 3 will generate a loop nest such that processor \( \text{pid}_w \) will send a message containing the value \( X[\mathcal{F}_{a_w}^s([p_w^r,i_w^r]^T)] \) at offset \( b_w \) to processor \( \text{pid}_r \), processor \( \text{pid}_r \) will receive a message containing the value \( X[\mathcal{F}_{a_r}^s([p_r^r,i_r^r]^T)] \) at offset \( b_r \) from processor \( \text{pid}_w \), such that \( \mathcal{F}_{a_w}^s([p_w^r,i_w^r]^T) = \mathcal{F}_{a_r}^s([p_r^r,i_r^r]^T) \) and \( b_w = b_r \). Since \( \mathcal{F}_{a_w}^s([p_w^r,i_w^r]^T) = \mathcal{F}_{a_w}(\mathbf{i}_w) \) and \( \mathcal{F}_{a_r}^s([p_r^r,i_r^r]^T) = \mathcal{F}_{a_r}(\mathbf{i}_r) \), then \( \mathcal{F}_{a_w}(\mathbf{i}_w) = \mathcal{F}_{a_r}(\mathbf{i}_r) \). ✷
4.1.1 Complexity of the Algorithm

The complexity of Algorithm 4 is based on the complexity of Algorithm 3. The number of unknowns is now $j = 2n + 4$, where $n$ is number of nested loops, due to the introduction of the two variables $\pi_{d_r}$ and $\pi_{d_w}$. The number of constraints is now $j = 6n + 9$ because of the introduction of the constraints $\pi_{d_r} = M_s(\pi_r)$ and $\pi_{d_w} = M_s(\pi_w)$.

4.1.2 Example of Incorrect Ordering

We mentioned previously that the order of the unknowns for the innermost loops (inside $\pi_w$ for the receive loop nest and inside $\pi_r$ for the send loop nest) must be the same when we incorporate the mapping of virtual processors to physical processors. To demonstrate that a different ordering of the loop variables can produce code that causes the processors to pack and unpack data in a different order, we ran Algorithm 4 using the ordering proposed by Amarasinghe and Lam on the following (contrived) example:

\[
\text{for } i2 = 1 \text{ to } N \text{ do }
\]
\[
\text{for } i3 = 1 \text{ to } N \text{ do }
\]
\[a[i2][i3] = a[i2][i3] + a[i3][i2-1]\]

Figure 4 shows the resulting send and receive loop nests for the data dependency. For this example, we used a partitioning function of $p = \Phi_s(i) = i2$ and mapping function of:

\[
M_s(p) = \left\lfloor \frac{p - lb_p}{\text{blksz}} \right\rfloor
\]

where $\text{blksz} = \lceil(ub_p - lb_p + 1)/P \rceil$. The mapping function $M_s(p)$ has the effect of assigning a block of partitions to each physical processor. Thus, a given processor $\text{mypid}$ will be responsible for the partitions $p$ such that $lb_p + \text{blksz} \cdot \text{mypid} \leq p < lb_p + (\text{mypid} + 1) \cdot \text{blksz}$. Figure 5 shows an excerpt from the debugging messages when the program was executed using $N = 8$ and $P = 4$. One can see that the data are packed and unpacked in a different order producing incorrect results.

5 Results

In this section we compare the performance of two programs compiled using Algorithms 3 and 4. The two input programs that we used for the comparison are Gaussian elimination and LU decomposition. We ran these programs on a cluster of 11 dual-processor Pentium PCs connected through a FastEthernet switch using MPI as the message-passing medium.
\begin{figure}
\centering
\begin{enumerate}
\item The receive loop nest
\begin{verbatim}
pidr = mypid
if 1 <= pidr AND pidr <= min((-1+r)/blkz, -1+p)
    then begin
        for pidv = 0 to -1+pidr do begin
            receive from pidv
            for pr = 1+blkz*pidr to min(blkz+blkz+pidr, N)
                do begin
                    i2r = pr
                    i3r = 1+blkz*pidv to blkz+blkz+pidv
                    do begin
                        pv = i3r
                        i2v = pv
                        i3v = -i+2r
                        unpack a[i3r][i2r - 1]
                    jend
                end
            end
        end
end
\end{verbatim}
\end{enumerate}
\caption{Example of Incorrect Ordering of the Loops}
\end{figure}

\begin{figure}
\centering
\begin{verbatim}
\ldots
\end{verbatim}
\caption{Excerpt of Debug Messages Using Incorrect Ordering}
\end{figure}
For these two input programs, we chose to assign a block of virtual processors to each physical processor. For Algorithm 3, the compiler inserted the parallelized loop nests along with their corresponding send and receive loop nests into the following loop to perform the tiling of virtual processors:

\[
\text{blksz} = \left\lfloor \frac{(ub_p - lb_p + 1)}{P} \right\rfloor \\
\text{for } p = lb_p + \text{myid} \times \text{blksz} \text{ to } \min(ub_p, lb_p + (\text{myid} + 1) \times \text{blksz} - 1) \text{ do begin} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{...} \\
\text{end}
\]

Figure 6 shows communication loop nests generated by Algorithm 3 for the data dependence between the \(a[i2][i3]\) array access on the lhs and the \(a[i1][i3]\) array access on the rhs of the program example in Figure 1.

For Algorithm 4, we used the following mapping function:

\[
M_s(p) = \left\lfloor \frac{p - lb_p}{\text{blksz}} \right\rfloor
\]

where \(\text{blksz} = \left\lfloor (ub_p - lb_p + 1)/P \right\rfloor\). Figure 7 shows the communication loop nest for the same data dependence as Figure 6, but this time generated by Algorithm 4. The total clock time for the compiler to run from source to MPI and then to executable using each algorithm is shown in Table 1.

One can see that the total number of messages sent from all virtual processors using the loop nest in Figure 6 is proportional to \((N - 1)(N - 2)(P - 1)/(2P)\). This function asymptotically approaches \((N - 1)(N - 2)/2\) as \(P\) approaches \(N\). Since we consider \(P << N\), then we consider the number of messages sent from the loop in Figure 6 to be \(O(N^2)\). One can also see that the total number of messages sent from all physical processors using the loop nest in Figure 7 is proportional to \((P - 1)(P - 2)/2 = O(P^2)\). Figure 8 shows an actual count of the total number of messages for both loop nests where \(N = 100\).

Although the mapping of virtual processors to physical processors is the same for both algorithms, Algorithm 3 generates individual messages between pairs of virtual processors. Thus, Algorithm 3 gives rise to a parallel program that will generate numerous short messages. In contrast, since Algorithm 4 incorporates the mapping \(M_s(p)\) in the system used to generate the communication loops, messages are aggregated. Thus, Algorithm 4 gives rise to a parallel program that will generate larger but fewer messages.

Figures 9 and 10 also show the execution times of the two input programs using the two algorithms. One can see from the results that the use of the mapping function to create the message-passing code via Algorithm 4 significantly improves the performance. This is primarily due to the reduction in the overall number of messages. Although the curves for Algorithm 4 are flatter, there is still a trend downward as the
pr = mpid
if 3 <= pr AND pr <= N then begin
  for pv = 2 to -1 + pr do begin
    if mpid <> (pv - 2) / blkz then begin
      receive from (pv - 2) / blkz
      lir = pv
      i2r = pr
      for i3r = pv to 1 + N do begin
        iv = -1 + pv
        i2v = pv
        i3v = i3r
        unpack a[i1r][i3r]
      end
    end
  end
end

pu = mpid
if 2 <= pu AND pu <= -1+N then begin
  for pr = 1 + pu to N do begin
    if mpid <> (pr - 2) / blkz then begin
      ilr = pu
      i2r = pr
      for i3r = ilr to 1+N do begin
        ilv = -1 + ilr
        i2v = ilr
        i3v = i3r
        pack a[i2v][i3v]
      end
    end
  end
end

(a) The receive loop nest
(b) The send loop nest

Figure 6: Resulting Send and Receive Loop Nests from Algorithm 3

pidr = mpid
if 1 <= pidr AND pidr <= -1 + P then begin
  for pidw = 0 to -1 + pidr do begin
    receive from pidw
    for pr = 2 + blkz * pidr to min(N, blkz + 1 + blkz) + pidr do begin
      i2r = pr
      for i3r = ilr to 1+N do begin
        iv = -1 + pidw
        i2v = pidw
        i3v = i3r
        unpack a[i1r][i3r]
      end
    end
  end
end

pidw = mpid
if 0 <= pidw AND pidw <= -2 + P then begin
  for pidr = 1 + pidw to -1 + P do begin
    for pr = 2 + blkz * pidr to min(N, blkz + 1 + blkz) + pidr do begin
      i2r = pr
      for i3r = ilr to 1+N do begin
        ilv = -1 + pidr
        i2v = pidr
        i3v = i3r
        pack a[i2v][i3v]
      end
    end
  end
end

(a) The receive loop nest
(b) The send loop nest

Figure 7: Resulting Send and Receive Loop Nests from Algorithm 4

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Gaussian Elapsed Time</th>
<th>LU Decomposition Elapsed Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 3</td>
<td>3.295</td>
<td>2.778</td>
</tr>
<tr>
<td>Algorithm 4</td>
<td>4.892</td>
<td>3.786</td>
</tr>
</tbody>
</table>

Figure 8: Total Number of Messages Send from all Processors
number of processors increases.

We were not able to obtain speedup with these examples for two reasons. First, there is redundant information packed in each message between physical processors, causing the messages to be larger than necessary. Second, because the messages contain redundant information, the message buffers will overflow with larger problem sizes. One can see from Figure 7 that with each iteration of the $p_r$ loop the same values are being packed in the same message. The data that are packed are independent of the value of $p_r$. Therefore, the $p_r$ loop could be suppressed into a degenerate loop. This seems straightforward enough; however, we have not yet investigated how to determine when redundant data is being packed and how best to deal with it.

Another possibility for performance improvement is message relay. In the example programs that we used, one processor sends the same data to many others processors. In this situation, the receiving processors could relay the message, instead of the message always originating from the same processor. In hand-coded experiments, we found that, although relaying does not reduce the number of messages, relaying does significantly improve the performance. We have not yet investigated how to determine if relaying can be applied and how to implement it.

6 Conclusions and Future Work

In this paper, we revisited the algorithms proposed by Amarasinghe and Lam [1]. We made a modification to their algorithms so that we could prove the correctness of the message passing loop nests. This was necessary so that we could extend the algorithms to incorporate the mapping of virtual processors to physical processors. This extension reduces the number of messages. In the examples that we showed the extension
reduced the total number of messages from $\mathcal{O}(N^2)$ to $\mathcal{O}(P^2)$, where $N$ is the input size and $P$ is the number of processors.

The reason that it is important to introduce a formal specification of the incorporation of mapping in the communication code generation is so that we can make use of the many scheduling heuristics proposed in the literature. We showed in previous work [10] that making use of a library of scheduling heuristics can improve the average performance of the resulting programs. We need a generalized mapping function for the communication code generation so that we can employ different mapping and scheduling heuristics.

We envision a framework where the system of constraints that describes communication requirements can be used by a heuristic to produce the mapping function $M_s(p)$. How one can adapt a scheduling heuristic to this framework is an open problem. Two possible solutions to this problem may be the Iterative Task Graph (ITG) [27] and the Parameterized Task Graph (PTG) [7,12]. If we can adapt heuristics to the framework that we propose, then we can use a metaheuristic, such as in [10], to choose an appropriate heuristic for each input program, therefore improving the average performance.

Appendix

Loop Bounds

Loop bounds can be represented by a system of constraints, which can be written as an expression of matrices and vectors. The loop bounds for the loop nest in Figure 1 of the paper are:

\[
\begin{align*}
    i_1 &\geq 1 \\
    i_1 &\leq N \\
    i_2 &\geq i_1 + 1 \\
    i_2 &\leq N \\
    i_3 &\geq i_1 \\
    i_3 &\leq N + 1
\end{align*}
\]

\[
\begin{align*}
    i_1 - 1 &\geq 0 \\
    N - i_1 &\geq 0 \\
    i_2 - i_1 - 1 &\geq 0 \\
    N - i_2 &\geq 0 \\
    i_3 - i_1 &\geq 0 \\
    N - i_3 + 1 &\geq 0
\end{align*}
\]
\[
\begin{align*}
\Rightarrow & \quad 
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 0 \\
-1 & 1 & 0 \\
0 & -1 & 0 \\
-1 & 0 & 1 \\
0 & 0 & -1
\end{bmatrix}
\cdot
\begin{bmatrix}
i_1 \\
i_2 \\
i_3
\end{bmatrix}
+ 
\begin{bmatrix}
-1 \\
-1 \\
0 \\
N + 1
\end{bmatrix}
\geq 
\begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}

(3)
\end{align*}
\]

Affine Partitioning

Affine partitioning is a technique to represent the assignment of loop iterations to partitions. This is also internally stored in matrix and vector form. One possible affine partitioning for the loop nest in Figure 1 is:

\[
\Phi_s(i) = i_2 = 
\begin{bmatrix}
0 & 1 & 0 \\
0 & -1 & 0 \\
0 & 0 & -1 \\
0 & -1 & 0 \\
0 & 0 & 0 \\
1 & 0 & -1
\end{bmatrix}
\cdot
\begin{bmatrix}
i_1 \\
i_2 \\
i_3
\end{bmatrix}
+ 
\begin{bmatrix}
0
\end{bmatrix}

(4)
\]

Joining the affine partitioning with the loop bounds gives a system \( A \) such that:

\[
A = \{ (D_s(i) \geq \vec{0}) \cup (p = \Phi_s(i)) \} = D_s \cdot i + d \geq \vec{0} \cup \left\{ \begin{array}{l}
p \geq i_2 \\
p \leq i_2
\end{array} \right\}
\]

\[
A = 
\begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & -1 & 1 & 0 \\
0 & 0 & -1 & 0 \\
0 & -1 & 0 & 1 \\
0 & 0 & 0 & -1 \\
1 & 0 & -1 & 0 \\
-1 & 0 & 1 & 0
\end{bmatrix}
\cdot
\begin{bmatrix}
p \\
i_1 \\
i_2 \\
i_3
\end{bmatrix}
+ 
\begin{bmatrix}
-1 \\
0 \\
N
\end{bmatrix}
\geq 
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}

(5)
\]
Using the system from equation (5), the lower and upper bounds for \( i_3 \) are \( i_1 \leq i_3 \leq N + 1 \). Projecting away \( i_3 \) from \( A \) using FME [3] produces:

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & -1 & 1 & 0 \\
0 & 0 & -1 & 0 \\
1 & 0 & -1 & 0 \\
-1 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
p \\
i_1 \\
i_2 \\
i_3
\end{bmatrix}
\begin{bmatrix}
-1 \\
N \\
-1 \\
N \\
0 \\
0
\end{bmatrix}
\geq
\begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
\tag{6}
\]

Using the system from equation (6), the lower and upper bounds for \( i_2 \) are \( \max\{i_1, p\} \leq i_2 \leq \min\{N, p\} \). Projecting away \( i_2 \) from \( A \) using FME produces:

\[
A = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & -1 & 0 & 0 \\
1 & -1 & 0 & 0 \\
0 & -1 & 0 & 0 \\
-1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
p \\
i_1 \\
i_2 \\
i_3
\end{bmatrix}
\begin{bmatrix}
-1 \\
N \\
-1 \\
N \\
0
\end{bmatrix}
\geq
\begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}
\tag{7}
\]

Using the system from equation (7), the lower and upper bounds for \( i_1 \) are \( 1 \leq i_1 \leq \min\{N, p - 1, N - 1\} \). Projecting away \( i_1 \) from \( A \) using FME produces:

\[
A = \begin{bmatrix}
1 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
p \\
i_1 \\
i_2 \\
i_3
\end{bmatrix}
\begin{bmatrix}
-2 \\
N
\end{bmatrix}
\geq
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\tag{8}
\]

Finally, using the system from equation (8), gives the lower and upper bounds for \( p \) of \( 2 \leq p \leq N \). The resulting parallel loop nest is:

```latex
\begin{align*}
\text{for } p = 2 \text{ to } N \text{ do } \\
\quad \text{for } i_1 = 1 \text{ to } p-1 \text{ do } \\
\quad \quad \text{for } i_2 = p \text{ to } p \text{ do } \\
\quad \quad \quad \text{for } i_3 = N+1 \text{ downto } i_1 \text{ do }
\end{align*}
```
Example 1: Parallelized Gaussian Elimination

Notice that the $i_2$ loop is degenerate. Also, the $p$ loop will be converted to an if statement since each virtual processor will executed a different iteration. The final loop nest is shown in Figure 2 of the paper.

**Last Write Tree**

The last write tree between the array reference on the lhs ($a_0$) and the second array reference on the rhs ($a_2$) in the loop nest from Figure 1 are:

$$i_w = L_{a_0a_4}(i_r) = \begin{cases} 
    \begin{bmatrix} 
        i_{1r} - 1 \\
        i_{1r} \\
        i_{3r} 
    \end{bmatrix} & \text{if } D_s(i_r) \geq \vec{0} \land D_s([i_{1r} - 1, i_{1r}, i_{3r}]) \geq \vec{0} \\
    \text{undefined} & \text{otherwise}
\end{cases}$$

(9)

For example, consider the iteration instance $i_r = [1, 2, 2]^T$. The value read by $i_r$ ($a[1][2]$) is defined outside of the loop nest because iteration instance $i_w = [0, 1, 2]^T$ is outside the loop bounds. However, the iteration instance $i_r = [2, 3, 3]^T$ needs to read the value $a[2][3]$ which is last modified by the iteration instance $i_w = [1, 2, 3]^T$. An important consideration for the last write tree is that there is no other iteration instance between $i_w$ and $i_r$ that modifies the value $a[2][3]$.

**Communication Code Generation**

The system used to create the communication code in Algorithm 3 step 1 is:

$$A = \begin{cases} 
    D_s(i_r) \geq \vec{0} \cup D_s(i_w) \geq \vec{0} \cup \\
    p_r = \Phi_s(i_r) \cup p_w = \Phi_s(i_w) \cup \\
    i_w = L_{a_r,a_w}(i_r) \cup p_w \neq p_r 
\end{cases}$$
\[
\begin{align*}
&\left\{ \begin{array}{l}
i_v - 1 \geq 0 \\
N - i_v \geq 0 \\
i_v - i_u - 1 \geq 0 \\
N - i_v \geq 0 \\
i_u - i_v \geq 0 \\
N - i_u + 1 \geq 0 \\
p_v - i_v \geq 0 \\
i_v - p \geq 0 \\
-1 + i_u \\
i_u - i_v + 1 \geq 0 \\
-1 + i_v \geq 0 \\
i_v - i_u \geq 0 \\
-1 + i_v \geq 0 \\
i_u - i_v \geq 0 \\
i_v - i_u \geq 0 \\
\end{array} \right\} \cup
\left\{ \begin{array}{l}
i_w - 1 \geq 0 \\
N - i_w \geq 0 \\
i_w - i_v - 1 \geq 0 \\
N - i_w \geq 0 \\
i_v - i_w \geq 0 \\
N - i_v + 1 \geq 0 \\
p_w - i_w \geq 0 \\
i_w - p \geq 0 \\
-1 + i_v \\
i_v - i_w + 1 \geq 0 \\
i_w - i_v \geq 0 \\
-1 + i_v \geq 0 \\
i_v - i_w \geq 0 \\
i_v - i_w \geq 0 \\
\end{array} \right\} \cup
\left\{ \begin{array}{l}
p_r - p - 1 \geq 0 \\
p_r - p \geq 0 \\
\end{array} \right\}
\end{align*}
\]

The loop nests that result from this system are shown in Figure 3 of the paper.

References


