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 $\mathbf{O}(N^2)$ to $\mathbf{O}(P^2)$, 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

* *International Journal of Parallel, Emergent and Distributed Systems (JPEDS)*, 21(5):323–344, October 2006.

† Department of Computer Science, 601 S. College Rd., University of North Carolina at Wilmington, Wilmington, NC 28403, cferner@uncw.edu, 910-962-7129

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 $\mathbf{O}(N^2)$ to $\mathbf{O}(P^2)$, 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 $\mathbf{v} = [v_1, \ldots, v_n]^{\mathrm{T}}$ to represent a vector, v_i to represent the ith element of the vector $\mathbf{v}$, and $\mathbf{v}_{i:j}$ to represent the subvector from the ith through the jth element of $\mathbf{v}$. We also will use the notation $\mathbf{v} = [w, \mathbf{z}]^{\mathrm{T}}$, where w is a scalar and $\mathbf{z}$ is a vector to mean $\mathbf{v} = [w, z_1, \ldots, z_n]^{\mathrm{T}}$ (i.e. $\mathbf{v} \neq [w, [z_1, \ldots, z_n]]^{\mathrm{T}}$). 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.

```
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]
```

Figure 1: Elimination Phase of Gaussian Elimination

2.1 Definitions

Definition 2.1 *An* iteration vector $\mathbf{i} = [i_1, \ldots, i_n]^T \in \mathbb{Z}^n$ *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 $a = (X, l, \mathcal{F}, \omega, e)$ where:*

- *X is the name of the array*
- *l is the number of subscripts*
- *$\mathcal{F}(\mathbf{i}) = F\mathbf{i} + \mathbf{f}$ is an affine expression called an access function which maps an iteration instance $\mathbf{i}$ to an element of the array, where F is an $l \times n$ matrix of symbolic constants and $\mathbf{f}$ is a vector of l symbolic constants*
- *ω 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 $\mathcal{D}_s(\mathbf{i}) = D_s\mathbf{i} + \mathbf{d}_s$, where $\mathbf{i}$ is a iteration instance, D_s is a $2n \times n$ matrix of symbolic constants and $\mathbf{d}_s$ is a vector of $2n$ symbolic constants.*

The iteration instance $\mathbf{i}$ is a valid iteration instance for loop nest s iff $\mathcal{D}_s(\mathbf{i}) \geq \vec{0}$, where $\vec{0}$ 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 $\prec$ is defined recursively for two iteration vectors $\mathbf{i}, \mathbf{i}' \in \mathbb{Z}^n$ such that $\mathbf{i} \prec \mathbf{i}'$ iff*

$$i_1 < i'_1 \ \vee \ (i_1 = i'_1 \ \wedge \ \mathbf{i}_{2:n} \prec \mathbf{i}'_{2:n}).$$

The lexicographical operator provides a total ordering of the iteration instances of a loop nest such that $\mathbf{i} \prec \mathbf{i}'$ iff iteration $\mathbf{i}$ is executed prior to iteration $\mathbf{i}'$ 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 $\Phi_s(\mathbf{i}) = C_s\mathbf{i} + c_s$*, 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 $A = \{(\mathcal{D}_s(\mathbf{i}) \geq \vec{0}) \;\cup\; (p = \Phi_s(\mathbf{i}))\}$ is built from the loop bounds and the partitioning. (An equality constraint, such as $p = \Phi_s(\mathbf{i})$, can be rewritten as two inequality constraints: $p \geq \Phi_s(\mathbf{i}) \;\wedge\; p \leq \Phi_s(\mathbf{i})$.) 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 $p = \Phi_s(\mathbf{i}) = i_2$

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 $L_{a_w a_r} : \mathbb{Z}^n \longrightarrow \mathbb{Z}^n$ *is a mapping from a read array access a_r and iteration instance $\mathbf{i}_r$ to the write array access a_w and iteration instance $\mathbf{i}_w$ that produced the value required. That is, given two array references a_r, a_w of a loop nest s with access functions $\mathcal{F}_{a_r}(\mathbf{i}_r)$ and $\mathcal{F}_{a_w}(\mathbf{i}_w)$, respectively, such that $\omega_{a_r} = false$ and $\omega_{a_w} = true$, then $L_{a_w a_r}(\mathbf{i}_r) = \mathbf{i}_w$ iff*

$$\mathcal{D}_s(\mathbf{i}_w) \geq \vec{0} \ \wedge \ \mathcal{D}_s(\mathbf{i}_r) \geq \vec{0} \ \wedge \ \mathcal{F}_{a_w}(\mathbf{i}_w) = \mathcal{F}_{a_r}(\mathbf{i}_r) \ \wedge \ \mathbf{i}_w \prec \mathbf{i}_r \ \wedge$$
$$\left(\nexists\, \mathbf{i}'_w \text{ such that } \ \mathcal{D}_s(\mathbf{i}'_w) \geq \vec{0} \ \wedge \ \mathcal{F}_{a_w}(\mathbf{i}'_w) = \mathcal{F}_{a_r}(\mathbf{i}_r) \ \wedge \ \mathbf{i}_w \prec \mathbf{i}'_w \prec \mathbf{i}_r\right).$$

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.

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, $\mathbf{i}_r$, and $\mathbf{i}_w$. The system of constraints should include the loop bounds $\mathcal{D}_s(\mathbf{i}_r) \geq \vec{0}$ and $\mathcal{D}_s(\mathbf{i}_w) \geq \vec{0}$, the partitioning $p_r = \Phi_s(\mathbf{i}_r)$ and $p_w = \Phi_s(\mathbf{i}_w)$, the last write dependence (which ties the receive and send variables together) $\mathbf{i}_w = L_{a_w a_r}(\mathbf{i}_r)$, 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 $p_r, p_w, \mathbf{i}_r, \mathbf{i}_w$ for receiving data and $p_w, p_r, \mathbf{i}_r, \mathbf{i}_w$ 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 $\mathbf{i}_w$ that produces a value and an iteration instance $\mathbf{i}_r$ 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[i2][i3]`) for iteration instance $\mathbf{i}_w$ to the array reference `a[i1][i3]` for iteration instance $\mathbf{i}_r$ such that $i_{1_w} = i_{1_r} - 1$, $i_{2_w} = i_{1_r}$, and $i_{3_w} = i_{3_r}$. 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).

```
pr = p
if 3 <= pr AND pr <= N then
    for pw = 2 to pr-1 do begin
        receive from pw
        i1r = pw
        i2r = pr
        for i3r = pw to N+1 do begin
            i1w = pw-1
            i2w = i1w+1
            i3w = i3r
            unpack a[i1r][i3r]
        end
    end
```

(a) The receive loop nest

```
pw = p
if 2 <= pw AND pw <= N-1 then
    for pr = pw+1 to N do begin
        i1r = pw
        i2r = pr
        for i3r = pw to N+1 do begin
            i1w = pw-1
            i2w = i1w+1
            i3w = i3r
            pack a[i2w][i3w]
        end
        send to pr
    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 $\Psi = [p_r, p_w, \mathbf{i}_r, \mathbf{i}_w]^{\mathrm{T}}$ which contains the unknowns of A. The order of the unknowns in Ψ 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 $\mathcal{F}_{a_r}(\mathbf{i})$ and $\mathcal{F}_{a_w}(\mathbf{i})$

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 $X_{a_r}[\mathcal{F}_{a_r}(\mathbf{i}_r)]$
2. Use FME to build a loop nest from A and Ψ 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` $p_r = lb_{p_r}$ `to` ub_{p_r} `do begin ... end`" to

   $p_r = p$
   `if` $lb_{p_r} \leq p_r$ `AND` $p_r \leq ub_{p_r}$ `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 $\Psi = [p_w, p_r, \mathbf{i}_r, \mathbf{i}_w]^{\mathrm{T}}$ which contains the unknowns of A. The order of the unknowns in Ψ 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 $\mathcal{F}_{a_r}(\mathbf{i})$ and $\mathcal{F}_{a_w}(\mathbf{i})$

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 $X_{a_w}[\mathcal{F}_{a_w}(\mathbf{i}_w)]$
2. Use FME to build a loop nest from A and Ψ 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

   $p_w = p$
   `if` $lb_{p_w} \leq p_w$ `AND` $p_w \leq ub_{p_w}$ `then begin ... end`

Algorithm 3 Create Communication Loop Nests

Input:3

- 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 $\mathbf{i}_r$ and the vector of loop indexes to be used for sending $\mathbf{i}_w$
- The variable used for the receiving processor p_r and the variable used for the sending processor p_w
- The loop bounds $\mathcal{D}_s(\mathbf{i})$ for the instruction s
- The last write data dependence mapping $L_{a_w a_r}(\mathbf{i})$
- The array access functions $\mathcal{F}_{a_r}(\mathbf{i})$ and $\mathcal{F}_{a_w}(\mathbf{i})$
- The affine partitioning $\Phi_s(\mathbf{i})$

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:

$$A = \left\{ \begin{array}{lclc} \mathcal{D}_s(\mathbf{i}_r) \geq \vec{0} & \cup & \mathcal{D}_s(\mathbf{i}_w) \geq \vec{0} & \cup \\ p_r = \Phi_s(\mathbf{i}_r) & \cup & p_w = \Phi_s(\mathbf{i}_w) & \cup \\ \mathbf{i}_w = L_{a_r a_w}(\mathbf{i}_r) & \cup & p_w \neq p_r & \end{array} \right\}$$

2. If A does not have a solution, then stop (no communication is necessary)
3. Use Algorithm 1 with inputs: A, $\Psi_r = [p_r, p_w, \mathbf{i}_r, \mathbf{i}_w]^{\mathrm{T}}$, p_w, $\mathcal{F}_{a_r}(\mathbf{i}_r)$, $\mathcal{F}_{a_w}(\mathbf{i}_w)$ to create a receive loop nest
4. Use Algorithm 2 with inputs: A, $\Psi_w = [p_w, p_r, \mathbf{i}_r, \mathbf{i}_w]^{\mathrm{T}}$, p_r, $\mathcal{F}_{a_r}(\mathbf{i}_r)$, $\mathcal{F}_{a_w}(\mathbf{i}_w)$ 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 straight forward 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 $B(\mathbf{i}, A, e)$ be a mapping from an iteration instance $\mathbf{i}$ for a receive or send loop nest that unpacks or packs an array element to its offset within the message. That is,*

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

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 $\omega_{a_r} =$ false and $\omega_{a_w} =$ true. Also let $\mathbf{i}'_r$ and $\mathbf{i}'_w$ be iteration instances of instruction s such that $\mathcal{D}_s(\mathbf{i}'_r) \geq \vec{0}$, $\mathcal{D}_s(\mathbf{i}'_w) \geq \vec{0}$, $\mathbf{i}'_w = L_{a_w a_r}(\mathbf{i}'_r)$, $p'_r = \Phi_s(\mathbf{i}'_r)$, and $p'_w = \Phi_s(\mathbf{i}'_w)$. If $p'_r \neq p'_w$, then Algorithm 3 will generate loop nests such that*

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

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

The offset used to unpack the value $X[\mathcal{F}_{a_r}(\mathbf{i}'_r)]$ 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, $b_r = B([\mathbf{i}'_r, \mathbf{i}'_w]^{\mathrm{T}}, A, e_{a_r})$. Likewise, the offset used to pack the value $X[\mathcal{F}_{a_w}(\mathbf{i}'_w)]$ in the message sent by p'_w is the number of iterations of the loops created in step 2 of Algorithm 2. Therefore, $b_w = B([\mathbf{i}'_r, \mathbf{i}'_w]^{\mathrm{T}}, A, e_{a_w})$. 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 $\mathbf{i}'_r$ and $\mathbf{i}'_w$ in Ψ_r and

Ψ_w are the same for both algorithms, the upper and lower bounds of the vector $[\mathbf{i}'_r, \mathbf{i}'_w]^{\mathrm{T}}$ 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 $[\mathbf{i}'_r, \mathbf{i}'_w]^{\mathrm{T}}$ 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 $p_r, \mathbf{i}_r, p_w, \mathbf{i}_w$ and $p_w, \mathbf{i}_w, p_r, \mathbf{i}_r$ 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 $B(\mathbf{i}_w, A, e_{a_w}) = B(\mathbf{i}_r, A, e_{a_r})$.

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:

$$\mathbf{O}\left(\sum_{r=0}^{j-1}(j-r)\frac{k^{2^r}}{4^{(2^r-1)}}\right) \tag{1}$$

where j is the number of unknowns and k is the number of constraints. In the worst case scenario, the number of constaints 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 constrains is $k = 6n + 5$: the lower and upper bounds on the loop variables $\mathcal{D}_s(\mathbf{i}_r) \geq \vec{0}$ and $\mathcal{D}_s(\mathbf{i}_w) \geq \vec{0}$ each

represent $2n$ constraints; the partitioning $p_r = \Phi_s(\mathbf{i_r})$ and $p_w = \Phi_s(\mathbf{i}_w)$ each represent 2 constraints; the LWT $\mathbf{i}_w = L_{a_w a_r}(\mathbf{i}_r)$ represents $2n$ constraints; and the requirement that $p_w \neq p_r$ represents 1 constraint.

Step 1 of Algorithm 3 is $\mathbf{O}(jk)$. 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 not 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 $p \in [lb_p..ub_p]$ to physical processor ids $pid \in [0..P)$.

To use this mapping, we modify our algorithm to include two new unknowns pid_r and pid_w, add constraints for $pid_r = M_s(p_r)$ and $pid_w = M_s(p_w)$, replace the constraint $p_r \neq p_w$ with the constraint $pid_r \neq pid_w$, and use pid_r and pid_w as the variables for sending and receiving messages. Notice that the mapping function adds the implicit constraints $0 \leq pid_r < P$ and $0 \leq pid_w < P$. We can then use

Algorithm 3 to generate the message passing code if we redefine the functions. Let

$$\begin{aligned}
\mathcal{D}^*_s([p,\mathbf{i}]^{\mathrm{T}}) &= D^*_s[p,\mathbf{i}]^{\mathrm{T}} + \mathbf{d}^*_s \\
\mathcal{F}^*_{a_r}([p,\mathbf{i}]^{\mathrm{T}}) &= \mathcal{F}_{a_r}(\mathbf{i}) \\
\mathcal{F}^*_{a_w}([p,\mathbf{i}]^{\mathrm{T}}) &= \mathcal{F}_{a_w}(\mathbf{i}) \\
L^*_{a_w a_r}([p,\mathbf{i}]^{\mathrm{T}}) &= L_{a_w a_r}(\mathbf{i}) \\
\Phi^*_s([p,\mathbf{i}]^{\mathrm{T}}) &= M_s(\Phi_s(\mathbf{i}))
\end{aligned} \tag{2}$$

where,

$$D^*_s = \begin{bmatrix} \vec{0} & D_s \\ 1 & -C_s \\ -1 & C_s \end{bmatrix} \text{ and } \mathbf{d}^*_s = \begin{bmatrix} \mathbf{d}_s \\ -c_s \\ c_s \end{bmatrix}.$$

Recall from Definitions 2.3 and 2.5 that $\mathcal{D}_s(\mathbf{i}) = D_s\mathbf{i} + \mathbf{d}_s$ and $\Phi_s(\mathbf{i}) = C_s\mathbf{i} + c_s$, where D_s is a $2n \times n$ matrix, $\mathbf{d}_s$ 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 $\mathbf{d}^*_s$ is such that $\mathcal{D}^*_s([p,\mathbf{i}]^{\mathrm{T}}) \geq \vec{0}$ satisfies both the loop bound constraints as well as the partitioning constraints, since:

$$\begin{aligned}
\mathcal{D}^*_s([p,\mathbf{i}]^{\mathrm{T}}) \geq \vec{0} &\Rightarrow D^*_s[p,\mathbf{i}]^{\mathrm{T}} + \mathbf{d}^*_s \geq \vec{0} \\
&\Rightarrow (0 \cdot p + D_s\mathbf{i} + \mathbf{d}^*_s \geq \vec{0}) \ \wedge \ (1 \cdot p - C_s\mathbf{i} - c_s \geq 0) \ \wedge \ (-1 \cdot p + C_s\mathbf{i} + c_s \geq 0) \\
&\Rightarrow (D_s\mathbf{i} + \mathbf{d}_s \geq \vec{0}) \ \wedge \ (p \geq C_s\mathbf{i} + c_s) \ \wedge \ (p \leq C_s\mathbf{i} + c_s) \\
&\Rightarrow (\mathcal{D}_s(\mathbf{i}) \geq \vec{0}) \ \wedge \ (p = \Phi_s(\mathbf{i})).
\end{aligned}$$

We will then use Algorithm 3 with the variables pid_r and 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 ω_{a_r} = true and ω_{a_w} = false
- The vector of loop indexes to be used for receiving $\mathbf{i}_r$ and the vector of loop indexes to be used for sending $\mathbf{i}_w$
- 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 pid_r and the variable used for the sending physical processor pid_w
- The loop bounds $\mathcal{D}_s(\mathbf{i})$ for the instruction s
- The last write data dependence mapping $L_{a_w a_r}(\mathbf{i})$
- The array access functions $\mathcal{F}_{a_r}(\mathbf{i})$ and $\mathcal{F}_{a_w}(\mathbf{i})$
- The affine partitioning $\Phi_s(\mathbf{i})$
- 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, $[p_r, \mathbf{i}_r]^{\mathrm{T}}$, $[p_w, \mathbf{i}_w]^{\mathrm{T}}$, pid_r, pid_w, $\mathcal{D}^*_s(\mathbf{i})$, $L^*_{a_w a_r}(\mathbf{i})$, $\mathcal{F}^*_{a_r}(\mathbf{i})$, $\mathcal{F}^*_{a_w}(\mathbf{i})$, $\Phi^*_s(\mathbf{i})$.

---

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 ω_{a_r} = false and ω_{a_w} = true. Also let $\mathbf{i}'_r$ and $\mathbf{i}'_w$ be iteration instances of instruction s such that $\mathcal{D}_s(\mathbf{i}'_r) \geq \vec{0}$, $\mathcal{D}_s(\mathbf{i}'_w) \geq \vec{0}$, $\mathbf{i}'_w = L_{a_w a_r}(\mathbf{i}'_r)$, $pid'_r = M_s(p'_r = \Phi_s(\mathbf{i}'_r))$, and $pid'_w = M_s(p'_w = \Phi_s(\mathbf{i}'_w))$. If $pid'_r \neq pid'_w$, then Algorithm 4 will generate loop nests such that*

- *processor pid'_w sends a message containing $X[\mathcal{F}_{a_w}(\mathbf{i}'_w)]$ at offset b_w to processor pid'_r,*
- *processor pid'_r receives a message containing $X[\mathcal{F}_{a_r}(\mathbf{i}'_r)]$ at offset b_r from processor 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 pid'_w will send a message containing the value $X[\mathcal{F}^*_{a_w}([p'_w, \mathbf{i}'_w]^{\mathrm{T}})]$ at offset b_w to processor pid'_r, processor pid'_r will receive a message containing the value $X[\mathcal{F}^*_{a_r}([p'_r, \mathbf{i}'_r]^{\mathrm{T}})]$ at offset b_r from processor pid'_w, such that $\mathcal{F}^*_{a_w}([p'_w, \mathbf{i}'_w]^{\mathrm{T}}) = \mathcal{F}^*_{a_r}([p'_r, \mathbf{i}'_r]^{\mathrm{T}})$ and $b_w = b_r$. Since $\mathcal{F}^*_{a_w}([p'_w, \mathbf{i}'_w]^{\mathrm{T}}) = \mathcal{F}_{a_w}(\mathbf{i}'_w)$ and $\mathcal{F}^*_{a_r}([p'_r, \mathbf{i}'_r]^{\mathrm{T}}) = \mathcal{F}_{a_r}(\mathbf{i}'_r)$, 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 pid_r and pid_w. The number of constraints is now $j = 6n + 9$ because of the introduction of the constraints $pid_r = M_s(p_r)$ and $pid_w = M_s(p_w)$.

4.1.2 Example of Incorrect Ordering

We mentioned previously that the order of the unknowns for the innermost loops (inside p_w for the receive loop nest and inside p_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:

```
for i2 = 1 to N do
    for i3 = 1 to N 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 $p = \Phi_s(\mathbf{i}) = i2$ and mapping function of:

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

where $blksz = \lceil (ub_p - lb_p + 1)/P \rceil$. The mapping function $M_s(p)$ has the effect of assigning a block of partitions to each physical processor. Thus, a given processor $mypid$ will be responsible for the partitions p such that $lb_p + blksz \cdot mypid \leq p < lb_p + (mypid + 1) \cdot blksz$. 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 program 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.

```
pidr = mypid
if 1 <= pidr AND pidr <= min((-1+N)/blksz, -1+P)
                                              then begin
   for pidw = 0 to -1+pidr do begin
      receive from pidw
      for pr = 1+blksz*pidr to min(blksz+blksz*pidr, N)
                                                do begin
         i2r = pr
         for i3r = 1+blksz*pidw to blksz+blksz*pidw
                                                do begin
            pw = i3r
            i2w = pw
            i3w = -1+i2r
            unpack a[i3r][i2r - 1]
         end
      end
   end
end
```

(a) The receive loop nest

```
pidw = mypid
if 0 <= pidw AND pidw <= min(-2+P, (-1-blksz+N)/
                                   blksz) then begin
   for pidr = 1+pidw to min(-1+P, (-1+N)/blksz)
                                            do begin
      for pw = 1+blksz*pidw to blksz+blksz*pidw
                                            do begin
         i2w = pw
         for i3w = blksz*pidr to min(-1+N, -1+blksz+
                                blksz*pidr) do begin
            pr = 1+i3w
            i2r = 1+i3w
            i3r = i2w
            pack a[i2w][i3w]
         end
      end
      send to pidr
   end
end
```

(b) The send loop nest

Figure 4: Example of Incorrect Ordering of the Loops

```
...
<pid2>: a[5][1] = a[5][1] + a[1][4]
<pid2>: a[5][2] = a[5][2] + a[2][4]
<pid2>: a[5][3] = a[5][3] + a[3][4]
<pid2>: a[5][4] = a[5][4] + a[4][4]
<pid2>: a[5][5] = a[5][5] + a[5][4]
<pid2>: a[5][6] = a[5][6] + a[6][4]
<pid2>: a[5][7] = a[5][7] + a[7][4]
<pid2>: a[5][8] = a[5][8] + a[8][4]
<pid2>: a[6][1] = a[6][1] + a[1][5]
<pid2>: a[6][2] = a[6][2] + a[2][5]
<pid2>: a[6][3] = a[6][3] + a[3][5]
<pid2>: a[6][4] = a[6][4] + a[4][5]
<pid2>: a[6][5] = a[6][5] + a[5][5]
<pid2>: a[6][6] = a[6][6] + a[6][5]
<pid2>: a[6][7] = a[6][7] + a[7][5]
<pid2>: a[6][8] = a[6][8] + a[8][5]
<pid2>: pack a[5][6]
<pid2>: pack a[5][7]
<pid2>: pack a[6][6]
<pid2>: pack a[6][7]
<pid2>: send to <pid3>
...
```

```
...
<pid3>: receive from <pid2>
<pid3>: unpack a[5][6]
<pid3>: unpack a[6][6]
<pid3>: unpack a[5][7]
<pid3>: unpack a[6][7]
<pid3>: a[7][1] = a[7][1] + a[1][6]
<pid3>: a[7][2] = a[7][2] + a[2][6]
<pid3>: a[7][3] = a[7][3] + a[3][6]
<pid3>: a[7][4] = a[7][4] + a[4][6]
<pid3>: a[7][5] = a[7][5] + a[5][6]
<pid3>: a[7][6] = a[7][6] + a[6][6]
<pid3>: a[7][7] = a[7][7] + a[7][6]
<pid3>: a[7][8] = a[7][8] + a[8][6]
<pid3>: a[8][1] = a[8][1] + a[1][7]
<pid3>: a[8][2] = a[8][2] + a[2][7]
<pid3>: a[8][3] = a[8][3] + a[3][7]
<pid3>: a[8][4] = a[8][4] + a[4][7]
<pid3>: a[8][5] = a[8][5] + a[5][7]
<pid3>: a[8][6] = a[8][6] + a[6][7]
<pid3>: a[8][7] = a[8][7] + a[7][7]
<pid3>: a[8][8] = a[8][8] + a[8][7]
...
```

Figure 5: Excerpt of Debug Messages Using Incorrect Ordering

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:

```
blksz = ⌈(ub_p − lb_p + 1)/P⌉
for p = lb_p + mypid * blksz to min(ub_p, lb_p + (mypid + 1) * blksz - 1) do begin
    ...
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:

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

where $blksz = \lceil (ub_p - lb_p + 1)/P \rceil$. 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 $\mathbf{O}(N^2)$. 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 $(P-1)(P-2)/2 = \mathbf{O}(P^2)$. 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 = mypid
if 3 <= pr AND pr <= N then begin
   for pw = 2 to -1 + pr do begin
      if mypid <> (pw - 2) / blksz then begin
         receive from (pw - 2) / blksz
         i1r = pw
         i2r = pr
         for i3r = pw to 1+N do begin
            i1w = -1 + pw
            i2w = pw
            i3w = i3r
            unpack a[i1r][i3r]
         end
      end
   end
end
```

(a) The receive loop nest

```
pw = mypid
if 2 <= pw AND pw <= -1+N then begin
   for pr = 1 + pw to N do begin
      if mypid <> (pr - 2) / blksz then begin
         i1r = pw
         i2r = pr
         for i3r = i1r to 1+N do begin
            i1w = -1 + i1r
            i2w = i1r
            i3w = i3r
            pack a[i2w][i3w]
         end
         send to (pr - 2) / blksz
      end
   end
end
```

(b) The send loop nest

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

```
pidr = mypid
if 1 <= pidr AND pidr <= -1 + P then begin
   for pidw = 0 to -1 + pidr do begin
      receive from pidw
      for pr = 2 + blksz * pidr to min(N, blksz + 1 + blksz
                                                  * pidr) do begin
         for i1r = 2 + blksz * pidw to blksz + 1 + blksz
                                                   * pidw do begin
            i2r = pr
            for i3r = i1r to 1+N do begin
               pw = i1r
               i1w = -1 + pw
               i2w = pw
               i3w = i3r
               unpack a[i1r][i3r]
            end
         end
      end
   end
end
```

(a) The receive loop nest

```
pidw = mypid
if 0 <= pidw AND pidw <= -2 + P then begin
   for pidr = 1 + pidw to -1 + P do begin
      for pr = 2 + blksz * pidr to min(N, blksz + 1 + blksz
                                                  * pidr) do begin
         for i1r = 2 + blksz * pidw to blksz + 1 + blksz
                                                   * pidw do begin
            i2r = pr
            for i3r = i1r to 1+N do begin
               pw = i1r
               i1w = -1 + i1r
               i2w = i1r
               i3w = i3r
               pack a[i2w][i3w]
            end
         end
      end
      send to pidr
   end
end
```

(b) The send loop nest

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

Table 1: Execution Time in Seconds of the Compiler from Source to Executable using each Algorithm

	Gaussian Elimination	LU Decomp
Total Compile Time Using Algorithm 3	3.295	2.778
Total Compile Time Using Algorithm 4	4.892	3.786

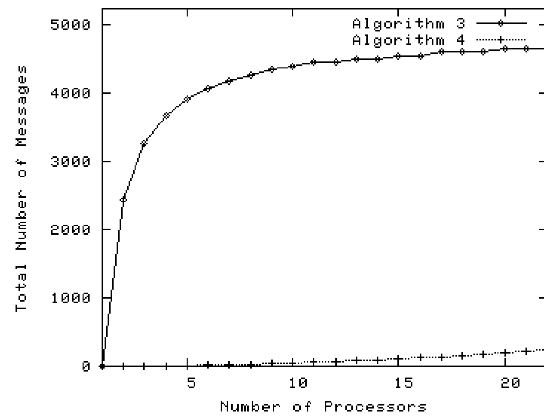

Figure 8: Total Number of Messages Send from all Processors

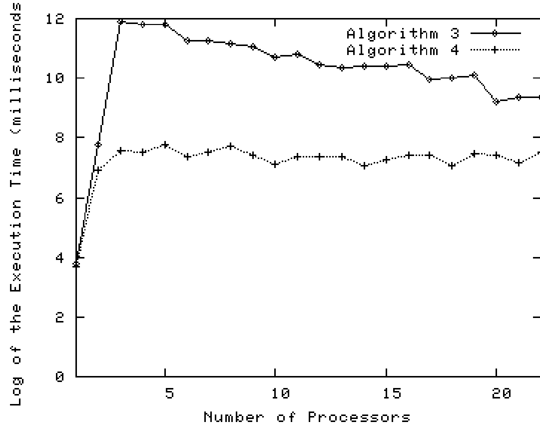

Figure 9: Execution Times for Gaussian Elimination (100x100)

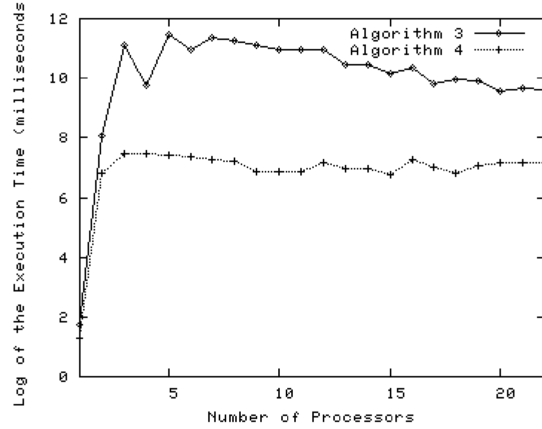

Figure 10: Execution Times for LU Decomposition (100x100)

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 $\mathbf{O}(N^2)$ to $\mathbf{O}(P^2)$, 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 heuristic 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:

$$\begin{array}{rcl} i_1 & \geq & 1 \\ i_1 & \leq & N \\ i_2 & \geq & i_1+1 \\ i_2 & \leq & N \\ i_3 & \geq & i_1 \\ i_3 & \leq & N+1 \end{array} \quad \Rightarrow \quad \begin{array}{rcl} i_1 - 1 & \geq & 0 \\ N - i_1 & \geq & 0 \\ i_2 - i_1 - 1 & \geq & 0 \\ N - i_2 & \geq & 0 \\ i_3 - i_1 & \geq & 0 \\ N - i_3 + 1 & \geq & 0 \end{array}$$

$$\Rightarrow \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix}}_{D_s} \cdot \underbrace{\begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix}}_{\mathbf{i}} + \underbrace{\begin{bmatrix} -1 \\ N \\ -1 \\ N \\ 0 \\ N+1 \end{bmatrix}}_{\mathbf{d}_s} \geq \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\vec{0}} \tag{3}$$

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:

$$\Phi_s(\mathbf{i}) = i_2 = \underbrace{\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}}_{C_s} \cdot \underbrace{\begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix}}_{\mathbf{i}} + \underbrace{0}_{c_s} \tag{4}$$

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

$$A = \{(\mathcal{D}_s(\mathbf{i}) \geq \vec{0}) \cup (p = \Phi_s(\mathbf{i}))\} = D_s \cdot \mathbf{i} + \mathbf{d} \geq \vec{0} \cup \left\{ \begin{array}{ccc} p & \geq & i_2 \\ p & \leq & i_2 \end{array} \right\}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} p \\ i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} -1 \\ N \\ -1 \\ N \\ 0 \\ N+1 \\ 0 \\ 0 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5}$$

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

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} p \\ i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} -1 \\ N \\ -1 \\ N \\ 0 \\ 0 \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{6}$$

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

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p \\ i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} -1 \\ N \\ -1 \\ N-1 \\ N \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7}$$

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

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p \\ i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} -2 \\ N \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{8}$$

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

```
for p = 2 to N do
    for i1 = 1 to p-1 do
        for i2 = p to p do
            for i3 = N+1 downto i1 do
```

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:

$$\mathbf{i}_w = L_{a_0 a_2}(\mathbf{i}_r) = \begin{cases} \begin{bmatrix} i_{1_r} - 1 \\ i_{1_r} \\ i_{3_r} \end{bmatrix} & \text{if } \mathcal{D}_s(\mathbf{i}_r) \geq \vec{0} \ \wedge \ \mathcal{D}_s([i_{1_r} - 1, i_{1_r} i_{3_r}]^{\mathrm{T}}) \geq \vec{0} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9}$$

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

Communication Code Generation

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

$$A = \left\{ \begin{array}{llll} \mathcal{D}_s(\mathbf{i}_r) \geq \vec{0} & \cup & \mathcal{D}_s(\mathbf{i}_w) \geq \vec{0} & \cup \\ p_r = \Phi_s(\mathbf{i}_r) & \cup & p_w = \Phi_s(\mathbf{i}_w) & \cup \\ \mathbf{i}_w = L_{a_r a_w}(\mathbf{i}_r) & \cup & p_w \neq p_r & \end{array} \right\}$$

$$
= \left\{ \begin{array}{l}
\left\{ \begin{array}{rcl}
i_{1_r} - 1 & \geq & 0 \\
N - i_{1_r} & \geq & 0 \\
i_{2_r} - i_{1_r} - 1 & \geq & 0 \\
N - i_{2_r} & \geq & 0 \\
i_{3_r} - i_{1_r} & \geq & 0 \\
N - i_{3_r} + 1 & \geq & 0
\end{array} \right\}
\cup
\left\{ \begin{array}{rcl}
i_{1_w} - 1 & \geq & 0 \\
N - i_{1_w} & \geq & 0 \\
i_{2_w} - i_{1_w} - 1 & \geq & 0 \\
N - i_{2_w} & \geq & 0 \\
i_{3_w} - i_{1_w} & \geq & 0 \\
N - i_{3_w} + 1 & \geq & 0
\end{array} \right\} \cup \\
\left\{ \begin{array}{rcl}
p_r - i_{2_r} & \geq & 0 \\
i_{2_r} - p_r & \geq & 0
\end{array} \right\}
\cup
\left\{ \begin{array}{rcl}
p_w - i_{2_w} & \geq & 0 \\
i_{2_w} - p_w & \geq & 0
\end{array} \right\} \cup \\
\left\{ \begin{array}{rcl}
-i_{1_w} + i_{1_r} - 1 & \geq & 0 \\
i_{1_w} - i_{1_r} + 1 & \geq & 0 \\
-i_{2_w} + i_{1_r} & \geq & 0 \\
i_{2_w} - i_{1_r} & \geq & 0 \\
-i_{3_w} + i_{3_r} & \geq & 0 \\
i_{3_w} - i_{3_r} & \geq & 0
\end{array} \right\}
\cup
\left\{ \begin{array}{rcl}
p_r - p_w - 1 & \geq & 0
\end{array} \right\}
\end{array} \right\} \quad (10)
$$

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

References

[1] S. P. Amarasinghe and M. S. Lam, Communication optimization and Code Generation for distributed memory machines, In the *Proceedings of The ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, pp. 126–138, June 1993.

[2] C. Amza, A.L. Cox, S. Dwarkada, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, TreadMarks: Shared memory computing on networks of workstations, *IEEE Computer*, pp. 18–28, February 1996.

[3] C. Ancourt and F. Irigoin, Scanning polyhedra with DO loops, In the *Proceedings of third ACM SIGPLAN Symposium on Principles & Practice of Programming Languages (PPOPP)*, Williamsburg, Virginia, April 21-24, pp. 39–50, 1991.

[4] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Kluwer Academic Publishers, Boston, MA, 1993.

[5] S.H. Bokari, On the mapping problem, *IEEE Transactions on Computers*, C-30, pp. 207-214, 1981.

[6] M.C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In Proc. of the *Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 29–38, Santa Barbara, Calif., July. 1995.

[7] M. Cosnard, E. Jeannot, T. Yang, SLC: Symbolic scheduling for executing parameterized task graphs on multiprocessors, in the *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, Wakamatsu, Japan, September 21-24, 1999.

[8] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, Evaluating the performance of software distributed shared memory as a target for parallelizing compilers, In the *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, pp. 475–482, April 1-5, 1997.

[9] C.S. Ferner, "The Paraguin compiler—Message-passing code generation using SUIF," in the *Proceedings of the IEEE SoutheastCon 2002*, Columbia, SC, pp. 1–6, April 5-7, 2002.

[10] C.S. Ferner and R.G. Babb, Automatic choice of scheduling heuristics for parallel/distributed computing, *Scientific Programming*, 7(1):47–65, 1999.

[11] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276-291, December 1992.

[12] E. Jeannot, Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs, in the *Proceedings of Parallel Computing 2001 (ParCo2001)*, Naples, Italy, September, 2001.

[13] P. J. Keleher, Update Protocols and cluster-based shared memory, *Computer Communications* 22(11):1045–1055, July 1999.

[14] C.W. Kessler, Parallel fourier-motzkin elimination, In *Proceedings of Euro-Par'96, Lyon*, France, August 1996, *Springer LNCS 1124*, pp. 66–71. (Full version of paper available at http://www.ida.liu.se/~chrke/fork95/a17.ps)

[15] A. A. Khan, C. L. McCreary, and M. S. Jones, A comparison of multiprocessor scheduling heuristics, In *Proceedings of the 23rd International Conference on Parallel Processing*, Aug. 1994.

[16] Yu-Kwong Kwok and Ishfaq Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, *ACM Computing Surveys*, 31(4):406–471, December 1999.

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

[18] D. E. Maydan, S. P. Amarasinghe and M. S. Lam, Array data-flow analysis and its use in array privatization, In the *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 10-13, pp. 2–15, 1993.

[19] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGS on multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, 446-451, April 1994.

[20] F. Quilleré, S. Rajopadhye, and D. Wilde, Generation of efficient nested loops from polyhedra, *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[21] S. Ramaswamy, S. Sapatnekar, P. Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers, *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, November 1997.

[22] D.J. Scales and M.S. Lam, An efficient shared memory layer for distributed memory machines, Computer Systems Laboratory Technical Report CSL-TR-94-627, Department of Computer Science, University of Stanford, 1994.

[23] B. Shirazi and M. Wang, Analysis and evaluation of heuristic methods for static task scheduling, *Journal of Parallel and Distributed Computing*, 10:222-232, 1992.

[24] Split-C, The Computer Science Division, University of California, Berkeley, http://www.cs.berkeley.edu/projects/parallel/castle/split-c/.

[25] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley Publishing Co., Redwood City, CA, 1996.

[26] J. Xue, *Loop Tiling for Parallelism*, Kluwer Academic Publishers, Boston, MA, 2000.

[27] T. Yang and C. Fu. Heuristic algorithms for scheduling iterative task graphs on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):608–622, June 1997.