2010

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

Suppressing Independent Loops in Packing/Unpacking Loop Nests to Reduce Message Size for Message-Passing Code

Phillip Jerry Martin II

A Capstone Project Submitted to the
University North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2010

Approved by

Advisory Committee

Dr. Curry Guinn	Dr. Tom Janicki

Dr. Clayton Ferner, Chair

Abstract

Suppressing Independent Loops in Packing/Unpacking Loop Nests to Reduce Message Size for Message-Passing Code. Martin II, Phillip, 2010. Capstone Paper, University of North Carolina Wilmington.

Two optimization techniques are explored for a parallelizing compiler that automatically generates parallel code for distributed-memory systems. Two problems arise from message-passing code generated by the compiler: 1) messages contain redundant data, and 2) the same data is sometimes transmitted to different processors, yet the messages are repacked for each processor. In this paper, the technique that eliminates the first problem, redundant data, is found to yield benefits. The steps necessary to implement the technique into the parallelizing compiler are provided in detail. The second optimization technique yields negligible benefits and is not implemented. Testing of the implementation is described and the results of the testing are given to show the benefits achieved by the first optimization technique.

Table of Contents

Tables

Figures

Chapter 1: Introduction

A compiler that can take a computer program written for execution on a single processor and automatically create parallel code that runs on multiple processors is a useful idea. The ability to reduce the execution time of a computer program by transforming it into parallel code without having to rewrite the code for parallel execution could provide great benefit. As computing machinery becomes cheaper, the popularity of clusters with many processors is increasing. To utilize the power of multiple processors, programs may have to be rewritten to take advantage of parallelization. Writing code for execution on multiple processors can be time consuming and prone to errors. A parallelizing compiler can generate the parallel code automatically and therefore reduce programmers' work.

Much progress has been made with parallelizing compilers that produce code for shared-memory systems. These systems are often large and expensive. On the other hand, distributed-memory systems are increasing in popularity due to the decreasing cost of equipment and the ease of assembly. However, parallelizing compilers that produce code for distributed-memory systems are more complex. Not only must the compiler produce code to divide the workload between processors, but it must also generate the code to perform the necessary communication. This communication code is vital when processors do not share memory. Any values computed by a processor that are required by another processor on another machine in the system must be packaged into a message and sent to the other machine. Since communication between processors increases the execution time of a program, a compiler must find a way to reduce the number of communications as well as the sizes of those communications.

In this paper, techniques are introduced to improve the performance of a particular parallelizing compiler, the Paraguin compiler. These techniques aim to reduce the size of the messages produced by the communication code and to prevent the repacking of messages that contain the exact same data with the overall goal to reduce execution time of parallel programs generated by the Paraguin compiler. Only one of the techniques discussed will be implemented. The reason for implementing only one technique will be discussed in Chapter 2.

Background

Dr. Ferner has developed an automatically parallelizing compiler that produces message passing code using the MPI library suitable for execution on a distributed-memory system. The compiler is called the Paraguin Compiler [5] and is built using the SUIF Compiler [9]. The SUIF Compiler provides the tools necessary to construct a functional parallelizing compiler. The details of how the technology produces parallel code are beyond the scope of this paper. The interested reader may examine [1], [2], [3], [4], [5], [9] and [10] for more information on code generation. Further details on the Paraguin compiler will be discussed in Chapter 2.

In this paper, the Elimination step of Gaussian Elimination shown in Figure 1 will be used as the example program. This program will be used to demonstrate the problems that the Paraguin compiler exhibits as well as to test the effectiveness of the techniques proposed to solve the problems. Information about the example program follows.

When a program is parallelized by the Paraguin compiler, the resultant parallel code contains execution code and communication code. The execution code is merely the

```
for (i1 = 1; i1 <= N; i1++) {
   for (i2 = i1+1; i2 <= N; i2++) {
      for (i3 = N+1; i3 >= i1; i3--) {
          a[i2][i3] = a[i2][i3] - a[i1][i3]
              * a[i2][i1] / a[i1][i1];
        }
    }
}
```

Figure 1. Elimination step of Gaussian elimination

original program in parallel form. The communication code handles the passing of messages between processors. The communication code includes receiving code and sending code. The execution code is placed between these two pieces of communication code. The code is arranged in this way:

Receive Code Block

Execution Code Block

Send Code Block

When code is run in parallel on a distributed-memory system, each processor in the system is assigned a unique processor ID. Each processor executes the exact same code. Control statements within the code determine which processors will execute which portion of code in the program.

Definition of terms

Iteration space. Encompasses all legal values of the loop variables. From Figure 1, where the variables are i_1, i_2, and i_3, iteration space is I = {(1, 2, N+1), …, (1, 2, 1), (1, 3, N+1), …, (1, 3, 1), …, (N-1, N, N-1)}

Iteration instance. One point in iteration space. E.g. $\vec{\imath} = (1, 2, 10)$

Operation. An iteration instance.

Partition. A partition of the iteration space. Contains 1 or more iteration instances.

Tile/Virtual Processor. A partition.

Dependence. An operation *A* is dependent on operation *B* if it requires data computed by operation *B*.

Independence. An operation *A* is independent from operation *B* if it does not require data computed by the operation *B*.

Source program. The computer program to be transformed by a parallelizing compiler into parallel code.

Sending processor. The processor that computes data required by another processor, and sends that data to the other processor.

Receiving processor. The processor that requires data computed by another processor.

Additional background – General overview of Paraguin compiler operations

When the example code in Figure 1 is transformed into parallel code, the iteration space must be divided up, or partitioned, into a set of tasks that can be executed by the pool of available processors. The workload is known as the iteration space. The iteration space encompasses all legal values of the loop variables. In the case of Figure 1, where the variables are `i1`, `i2` and `i3`, the iteration space would be all the iterations

represented by iterations I = {(1, 2, N+1), (1, 2, N), …, (N-1, N, N-1)}. One point in the iteration space is called an iteration instance e.g. $\vec{\imath}_x$ = (2, 4, 6) or (1, 2, 3).

Using Lim & Lam's algorithm [7] to determine the best way to partition the iteration space, the code can be partitioned on loop variable `i2`. Thus, all iteration instances where `i2` is equal would belong to the same partition `p`. For example, one partition might contain *p2* = {(1,2,3), (2,2,3), (3,2,4), …} where `i2 = 2`. Figure 2 shows the resulting parallel code when the code in Figure 1 is partitioned on `i2`.

```
if (2 <= p && p <= N){
    for (i1 = 1; i1 <= p-1; i1++){
        i2 = p
        for (i3 = N+1; i3 >= i1; i3--){
            a[i2][i3] = a[i2][i3] - a[i1][i3] *
                                    a[i2][i1] / a[i1][i1];
        }
    }
}
```

Figure 2. Resulting parallel loop nest for Gaussian elimination using the partitioning `p` = `i2`

Once the iteration space is divided into partitions, then the partitions are mapped to physical processors. This mapping determines which partitions will be executed by which processors. Figure 3 shows an example where partitions *p2* and *p3* are mapped to Processor 1, and partitions *p4*, *p5* and *p6* are mapped to Processor 2.

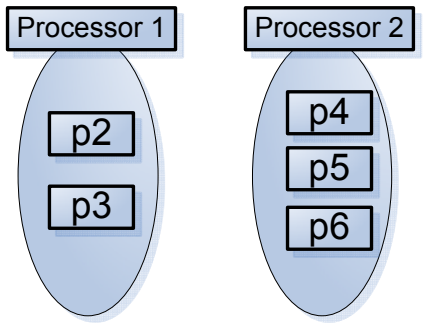

Figure 3. Partitions mapped to physical processors

Data dependence

Data dependence can exist between memory references. A data dependence exists when a program instruction `X` reads from a memory reference to which another instruction `Y` had previously written. The instruction `X` is dependent on the value written to the memory reference by the instruction `Y`, and thus it can be said that instruction `X` is dependent upon instruction `Y`.

This concept can be extended to iteration instances, since in the case of Figure 1, instructions are executed within iteration instances. If instruction `X` is dependent on instruction `Y`, then the iteration instance that contains instruction `X` is also said to be dependent on the iteration instance that contains instruction `Y`.

Furthermore, iteration instances are divided up into partitions. If a partition contains an iteration instance that is dependent upon another iteration instance in a different partition, then the former partition is said to be data dependent on the latter partition.

Finally, if a partition is allocated to a physical processor, and that partition is dependent upon another partition that was allocated to another physical processor, then it can be stated that the former processor is dependent upon the latter processor.

In the case that a processor requires data computed by another processor, communication of that data will be required. The processor that computes the needed value is known as the sending or write processor. The processor that needs the value computed is called the receiving or read processor. In Figure 4, Processor 1 executed the iterations contained in partition `p2`. Data was computed and stored in memory. Processor 2 needs the data computed in partition `p2` in order to execute the iterations contained in

partition *p4*. Partition *p4* is dependent on *p2* because *p4* requires the data computed by *p2*.

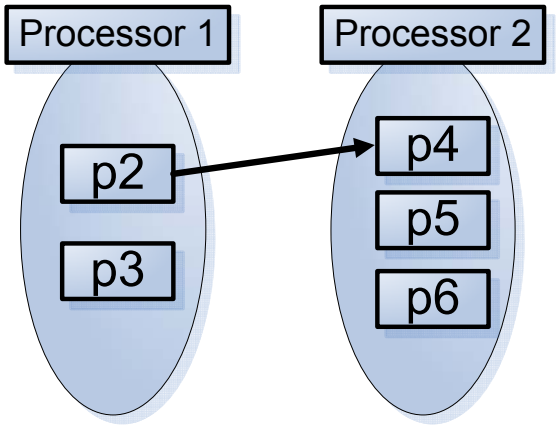

Figure 4. Dependency between *p4* and *p2*

The Paraguin compiler can automatically produce the communication code required by these data dependencies. The communication code is comprised of a sending loop, which packs data into messages and sends those messages to other processors, and a receiving loop, which receives messages from sending processors and unpacks the data into local memory. The main execution code is placed between the receive code and the send code. In this way, a processor will check for received data, execute its workload, and then send any dependent data to other processors. Figure 5 and Figure 6 show the receive and send loops, respectively, that perform the communication required for *one* of the dependencies in the program segment of Figure 1. The data dependence shown here is between the second array reference on the right-hand side of the assignment (`a[i1][i3]`) and the array reference on the left-hand side of the assignment (`a[i2][i3]`). The following section will explain the meaning of the variables in the communication loops.

```
//RECEIVE loop
pidr = mypid;
if(pidr >= 1 && pidr <= (-2+N)/blksz){
   for(pidw = 0; pidw <= -1+pidr; pidw++){
      printf ("<pid%d>: receive from <pid%d>\n", pidr, pidw);
      MPI_Recv(..., pidw, ...);
      for(pr = 2+blksz*pidr; pr <= min(1+blksz+blksz*pidr, N); pr++){
         for(i1r = max(2+blksz*pidw,2); i1r <= 1+blksz+blksz*pidw; i1r++) {
            i2r = pr;
            for(i3r = i1r; i3r <= 1+N; i3r++) {
                pw = i1r;
                i1w = -1+pw;
                i2w = 1+i1w;
                i3w = i3r;
                MPI_Unpack(..., &a[i1r][i3r], ...);
                printf("<pid%d, p%d>: unpack a[%d][%d] - Value: %f\n",
                                          pidr, pr, i1r, i3r, a[i1r][i3r]);
            }
         }
      }
   }
}
```

Figure 5. The receive loop for the elimination step of Gaussian elimination

```
//SEND loop
pidw = mypid;
if( pidw >= 0 && pidw <= (-2-blksz+N)/blksz){
   for(pidr = 1+pidw; pidr <= (-2+N)/blksz; pidr++){
      for(pr = 2+blksz*pidr; pr <= min(1+blksz+blksz*pidr, N); pr++){
         for(i1r = max(2+blksz*pidw,2); i1r <= 1+blksz+blksz*pidw; i1r++) {
            i2r = pr;
            for(i3r = i1r; i3r <= 1+N; i3r++) {
                pw = i1r;
                i1w = -1+pw;
                i2w = 1+i1w;
                i3w = i3r;
                MPI_Pack(&a[i2w][i3w], ...);
                printf ("<pid%d, p%d>: pack a[%d][%d] - Value: %f\n",
                                          pid w, pw, i2w, i3w, a[i2w][i3w]);
            }
         }
      }
      MPI_Send(... pidr ... );
      printf ("<pid%d>: send to <pid%d>\n", pidw, pidr);
   }
}
```

Figure 6. The send loop for the elimination step of Gaussian elimination

Communication code – Explanation of loop variables

Table 1 summarizes the meaning of the loop variables. Here we will look at the send loop variables in Figure 6 in more detail. Variable `pidw` is the processor ID number of the physical processor that computes the data that needs to be communicated. It is otherwise known as the write processor, hence the "w" in the variable name. Variable `pidr` is the processor ID of the receiving, or read, processor, hence the "r" in the variable name. `pidr` is the processor that depends upon the data produced by `pidw`. Variable `pr` is the partition mapped to the receiving processor, `pidr`. Variables `i1r`, `i2r` and `i3r` collectively represent the iteration instance that requires the data. Variables `i1w`, `i2w` and `i3w` collectively represent the iteration instance that computed the data required by iteration instance (`i1r`, `i2r`, `i3r`). A further breakdown of these variables follows.

`pidw`	Writing processor
`pidr`	Reading processor
`pw`	Partition assigned to writing processor
`pr`	Partition assigned to reading processor
`i1w`, `i2w`, `i3w`	Iteration instance that computes the value needed
`i1r`, `i2r`, `i3r`	Iteration instance that needs the value computed

Table 1. Loop variables and their meanings

As the sending code is executed on each physical processor, the processor assigns its processor ID to variable `pidw`:

```
pidw = mypid;
```

It then checks to see if its processor ID is in the range of write processors that produce data that is needed by another processor:

```
if( pidw >= 0 && pidw <= (-2-blksz+N)/blksz){
```

If so, then the processor loops through each receiving processor, `pidr`, that requires data produced by itself, `pidw`:

```
for(pidr = 1+pidw; pidr <= (-2+N)/blksz; pidr++){
```

The code iterates through each partition, `pr`, on receiving processor, `pidr`:

```
for(pr = 2+blksz*pidr; pr <= min(1+blksz+blksz*pidr,
  N); pr++){
```

The iteration instances (`i1r`, `12r`, `13r`) contained within each partition, `pr`, are looped through:

```
for(i1r = max(2+blksz*pidw,2); i1r <=
  1+blksz+blksz*pidw; i1r++){
       i2r = pr;
       for(i3r = i1r; i3r <= 1+N; i3r++) {
```

Note that variable `i2r` is not a loop, but simply an assignment statement. It was originally a loop that had this form:

```
for(i2r = pr; i2r <= pr; i2r++)
```

Since the lower and upper bounds of the loop are equal, it has only one iteration, which is a *degenerate* loop. The Paraguin compiler rewrites the loop as:

```
i2r = pr;
```

The code iterates through each partition, pw, on the sending processor, pidw. Partition pw represents the partition that produced the data that must be communicated. Note that this loop has also been rewritten as an assignment statement by the compiler because its lower and upper bounds are equal:

```
pw = i1r;
```

The code then iterates through the iteration instance (i1w, i2w, i3w) that computed the data that needs to be communicated. Note that these loops have also been rewritten as assignments statements by the compiler because their lower and upper bounds are equal:

```
i1w = -1+pw;
i2w = 1+i1w;
i3w = i3r;
```

Finally, the body of this loop nest is the statement that packs the data computed by iteration instance (i1w, i2w, i3w) in partition pw which is mapped to physical processor pidw. This data is needed by iteration instance (i1r, i2r, i3r) in partition pr which is mapped to physical processor pidr.

```
MPI_Pack(&a[i2w][i3w], ...);
```

Once all the dependent data needed by pidr has been packed into a message by pidw, the MPI_Send statement is executed. This statement sends the message to the receiving processor, pidr.

The packing and sending of data is done for each pidr that requires data computed by pidw.

Receiving code explanation

The variables in the receiving code in Figure 5 have the same meaning as those in the sending code. The order of the loops is different. In Figure 5, the processor assigns its processor ID to the `pidr` variable:

```
pidr = mypid;
```

The code then checks to see if the processor ID is within the range of processors that is receiving data from sending processors:

```
if(pidr >= 1 && pidr <= (-2+N)/blksz){
```

The code loops through each write processor, `pidw`, that is sending a message to this processor, `pidr`.

```
for(pidw = 0; pidw <= -1+pidr; pidw++){
```

The code pulls the message from sending processor `pidw` out of the message buffer:

```
MPI_Recv(..., pidw, ...);
```

The code then loops through each partition `pr` that requires data from `pidw`:

```
for(pr = 2+blksz*pidr; pr <= min(1+blksz+blksz*pidr,
     N); pr++){
```

The code then loops through the iteration instance (`i1r`, `i2r`, `i3r`). Note that the loop for `i2r` has been reduced to an assignment statement by the Paraguin compiler because its lower and upper bounds are equal.

```
for(i1r = max(2+blksz*pidw,2); i1r <=
   1+blksz+blksz*pidw; i1r++) {
      i2r = pr;
      for(i3r = i1r; i3r <= 1+N; i3r++) {
```

The code loops through each partition `pw` which is the partition on `pidw` that computed the data required by iteration instance (`i1r`, `i2r`, `i3r`). Once again, the loop was replaced with an assignment statement:

```
pw = i1r;
```

The final loops are the iteration instances (`i1w`, `i2w`, `i3w`) that computed the dependent data.

```
i1w = -1+pw;
i2w = 1+i1w;
i3w = i3r;
```

The code then unpacks the dependent data into the local memory of processor `pidr`:

```
MPI_Unpack(..., &a[i1r][i3r], ...);
```

Statement of the problem

Parallelizing compilers that generate code for distributed-memory systems should attempt to reduce the amount of communication and the sizes of the messages sent between processors so as to reduce the execution time of the resulting parallel program.

The Paraguin compiler exhibits two behaviors that increase the execution time of the parallel code it produces:

1. **The compiler packs redundant data**

First, the Paraguin compiler generates code that produces messages containing redundant data. That is, the compiler creates, or packs, a message with the same data two or more times. This increases the size of the messages. Time is wasted packing the redundant data and sending the larger messages to receiving processors.

Figure 7 shows an excerpt of the debugging output from running a parallel version of the Gaussian example program in Figure 1. The output shows that the array values produced by partition `p2` on processor `pid1` are packed into a message which is then sent to processor `pid2`. The figure shows that the array values have been packed into the message more than once. This problem occurs when multiple partitions that have been mapped to the same physical processor (in this case, the receiving processor) require the same data values computed by another partition mapped to a different physical processor. This is shown in Figure 8. Partitions `p4` and `p5` on the receiving processor are dependent upon the same data computed by partition `p2` on the sending processor. When the sending processor packs up the data to be sent to the receiving processor, it packs the data for both `p4` and `p5`. The problem occurs because the loops that produce the send and receive code do not take into account the fact that these partitions that require the exact same data are on the same processor, thus the data would only need to be packed once and sent to the receiving processor. Both partitions `p4` and `p5` would then have access to that data without the unnecessary packing of redundant data.

```
...
<pid1, p2>: pack a[2][2] - Value: 63.000000
<pid1, p2>: pack a[2][3] - Value: 28.000000
<pid1, p2>: pack a[2][4] - Value: 91.000000
<pid1, p2>: pack a[2][5] - Value: 60.000000
<pid1, p2>: pack a[2][6] - Value: 64.000000
...
<pid1, p2>: pack a[2][2] - Value: 63.000000
<pid1, p2>: pack a[2][3] - Value: 28.000000
<pid1, p2>: pack a[2][4] - Value: 91.000000
<pid1, p2>: pack a[2][5] - Value: 60.000000
<pid1, p2>: pack a[2][6] - Value: 64.000000
...
<pid1>: send to <pid2>
```

Figure 7. Excerpt from debugging output

This can be seen in send code in Figure 6. Observe that the data that is being packed are the values stored in the array `a[i2w][i3w]`. Because the value of `pr` does not determine the value of the array subscript variables, `i2w` and `i3w`, each iteration of the `pr` loop packs the same data in the message being assembled for transmission to processor `pidr`. For example, `i2w` is dependent on `i1w`, which is dependent on `pw`, which is dependent on `i1r`, which is dependent on `pidw`, which is the current processor id. The values of `i2w` and `i3w` are independent of the value of `pr`. Since each partition `pr` resides on processor `pidr`, the data should only be packed once and then sent. Packing the data for each `pr` on `pidr` is redundant.

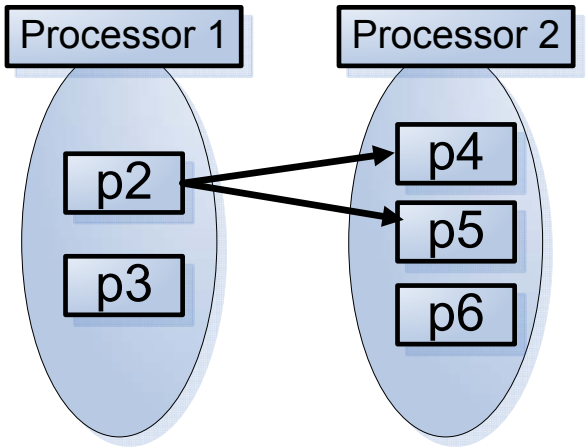

Figure 8. Multiple partitions require data

2. The compiler recreates messages with identical data

Second, the Paraguin compiler generates code that recreates messages that have identical data. For example, Processor 1 may pack a message with data that is required by Processor 2 and Processor 3. Instead of creating the message once and then sending that message to both processors, the code creates the message and sends the message to Processor 2. It then recreates the same message and sends the message to Processor 3. Time is wasted in recreating identical messages instead of packing the data into the message once and then sending that message to the various receiving processors.

This can be seen in Figure 6. Because the array subscripts (`i2w`, `i3w`) are independent of `pidr`, each iteration of the `pidr` loop creates the same message.

Chapter 2: Preliminary work / proposed solutions

To solve the two problems exhibited by the Paraguin compiler, the following techniques were proposed.

Technique to eliminate redundant data packing

Since the code packs redundant data whenever two or more partitions that are dependent on the same data are mapped to the same physical processor, the code should only pack that data once for each receiving processor instead of for each partition. Figure 6 shows that data is packed for each partition `pr` even though the value of `pr` does not determine which data `(a[i2w][i3w])` are packed. Variables `i2w` and `i3w` are independent of variable `pr`. So each time the `pr` loop runs, the same data is packed.

To eliminate redundant data, each loop variable in the communication code must be checked to determine if the array subscripts (in this case `i2w` and `i3w`) are dependent upon the value of that loop variable. In the example in Figure 6, when each variable is checked to see if its value determines the value of the subscripts of the data being packed, it is found that the subscripts are independent of `pr` and `i2r`.

This is the technique proposed to eliminate redundant data: All loop variables that do not determine which values are packed must be found. Upon discovering that a loop variable is independent from the subscripts of the data being packed, that loop can be eliminated and replaced with an assignment statement. For example, the loop for `pr`

```
for(pr = 2+blksz*pidr; pr <= min(1+blksz+blksz*pidr,
  N); pr++){
```

can be replaced with a loop where the lower bounds equals the upper bounds.

```
for(pr = 2+blksz*pidr; pr <= 2+blksz*pidr, N); pr++){
```

Since this becomes a degenerate loop, the Paraguin compiler will then reduce this loop to an assignment statement:

```
pr = 2+blksz*pidr;
```

Doing this will eliminate the unnecessary iterations of the `pr` loop which have no influence on what data is packed. The other variable in this example, `i2r`, has already been reduced to an assignment statement by the Paraguin compiler because it was previously a degenerate loop. Therefore, it does not contribute to the redundant data packing problem.

Technique to eliminate recreation of identical messages

Whenever a sending processor creates messages for receiving processors, it always creates the message from scratch even if all the messages it is sending are identical**.** The code must create and send a message for each receiving processor. There may be an occurrence where the identity of the receiving processor has no effect on what data is packed. Since the array subscripts `i2w` and `i3w` are independent from `pidr`, each iteration of the `pidr` loop will create and send the same message. The following technique is proposed to eliminate this problem: If the `pidr` loop does not affect which data values are packed into the message, then the removal of this loop would cause the message to be created only once. However, the removal of the loop would also cause the

message to be *sent* only once instead of sending it to each receiving processor, which would be an incorrect solution. To end the recreation of messages, but still allow the sending of that message to each receiving processor, an IF statement can be placed within the `pidr` loop. Such as:

```
for(pidr = 1+pidw; pidr <= (-2+N)/blksz; pidr++){

     if(pidr == 1+pidw){

          .
          .
          .

          MPI_Pack(&a[i2w][i3w], ...);

     }

     MPI_Send(... pidr ... );

}
```

The IF statement allows the packing code to be executed once, but subsequent iterations of the `pidr` loop fail the condition. The `MPI_Send` code is NOT placed within the IF block. This allows following iterations of the `pidr` loop to send the message to processor `pidr` without recreating the data within the IF statement.

In the first phase of this study [8], the parallel code in Figure 5 and Figure 6 was modified by hand to test the feasibility of the two techniques described above. Where loops were found to be independent of the array subscript variables, the loops were replaced with assignment statements to eliminate redundant data packing. Figure 9 and Figure 10 show the hand-written modifications for the first optimization. An IF statement

was placed within the `pidr` loop to eliminate message recreation as described previously.

```
//RECEIVE loop
pidr = mypid;
if(pidr >= 1 && pidr <= (-2+N)/blksz){
   for(pidw = 0; pidw <= -1+pidr; pidw++){
      printf ("<pid%d>: receive from <pid%d>\n", pidr, pidw);
      MPI_Recv(..., pidw, ...);

      pr = 2+blksz*pidr;   //loop replaced with assignment statement
         for(i1r = max(2+blksz*pidw,2); i1r <= 1+blksz+blksz*pidw; i1r++) {
            i2r = pr;
            for(i3r = i1r; i3r <= 1+N; i3r++) {
                pw = i1r;
                i1w = -1+pw;
                i2w = 1+i1w;
                i3w = i3r;
                MPI_Unpack(..., &a[i1r][i3r], ...);
                printf("<pid%d, p%d>: unpack a[%d][%d] - Value: %f\n",
                                          pidr, pr, i1r, i3r, a[i1r][i3r]);
            }
         }
   }
}
```

Figure 9. Hand-modified receive loop from Figure 5

```
//SEND loop
pidw = mypid;
if( pidw >= 0 && pidw <= (-2-blksz+N)/blksz){
   for(pidr = 1+pidw; pidr <= (-2+N)/blksz; pidr++){

      pr = 2+blksz*pidr;   //loop replaced with assignment statement
         for(i1r = max(2+blksz*pidw,2); i1r <= 1+blksz+blksz*pidw; i1r++) {
            i2r = pr;
            for(i3r = i1r; i3r <= 1+N; i3r++) {
                pw = i1r;
                i1w = -1+pw;
                i2w = 1+i1w;
                i3w = i3r;
                MPI_Pack(&a[i2w][i3w], ...);
                printf ("<pid%d, p%d>: pack a[%d][%d] - Value: %f\n",
                                            pid w, pw, i2w, i3w, a[i2w][i3w]);
            }
         }
      MPI_Send(... pidr ... );
      printf ("<pid%d>: send to <pid%d>\n", pidw, pidr);
   }
}
```

Figure 10. Hand-modified send loop from Figure 6

After testing the hand-written code, the results showed that the first technique, which eliminated redundant data packing, greatly reduced the time of execution and the amount of memory required. It also allowed the program to be run using larger input sizes.

The second technique, eliminating the recreation of identical messages, improved the performance only slightly. As was expected, it did not reduce the size of the messages, but it also did not reduce the time of execution significantly, which was not expected. This is because most of the execution time of a parallel program running in a distributed-memory environment is spent in communication not execution. The time spent recreating identical messages is very slight when compared to the time spent transmitting messages.

Due to the negligible improvement provided by the second technique, the decision was made not to implement it into the Paraguin compiler at this time. However, the first technique demonstrated that it may yield great benefits. In this paper, the implementation, testing, analysis and results of the first technique will be described in detail in Chapters 4 and 5.

Chapter 3: Literature & review background

Parallelizing compiler basics

In [13], parallelizing compiler techniques based on linear inequalities are introduced. The paper describes a parallelizing compiler created using the SUIF toolkit [9]. The Paraguin compiler incorporates concepts from [13]; loop bounds, loop partitioning and last write tree. A short overview of these concepts as implemented in the Paraguin compiler is presented below [4]. An understanding of these concepts is necessary to explain how the redundant data elimination technique is implemented.

Loop bounds

The Paraguin compiler represents loop bounds in the source program as a system of constraints, or linear inequalities. This can be represented as an expression of matrices and vectors. The loop bounds of the example program in Figure 1 are represented this way in Figure 11.

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

Figure 11. Loop bounds represented as a system of inequalities (left) and as matrices (right) [4]

Loop partitioning

The Paraguin compiler uses a technique called Affine Partitioning [7] which is a representation of the assignment of loop iteration instances to partitions. Simply, it represents how the loop iteration instances will be divided up into partitions. These partitions, or virtual processors, are later allocated to real processors.

For example, consider Figure 1. The loops could be partitioned on loop `i2`. Thus all iterations of `i2` would be in the same partition. Meaning all iterations where `i2 = 1` would be contained in one partition, all iterations where `i2 = 2` would be in another partition, etc. To get finer-grained parallelism, the loops could be partitioned on `i3` though this may decrease performance due to the increased cost of communication. Finding optimal partitioning is described in [7]. Figure 12 shows an example of partitioning a 3-dimensional iteration space on `i2`.

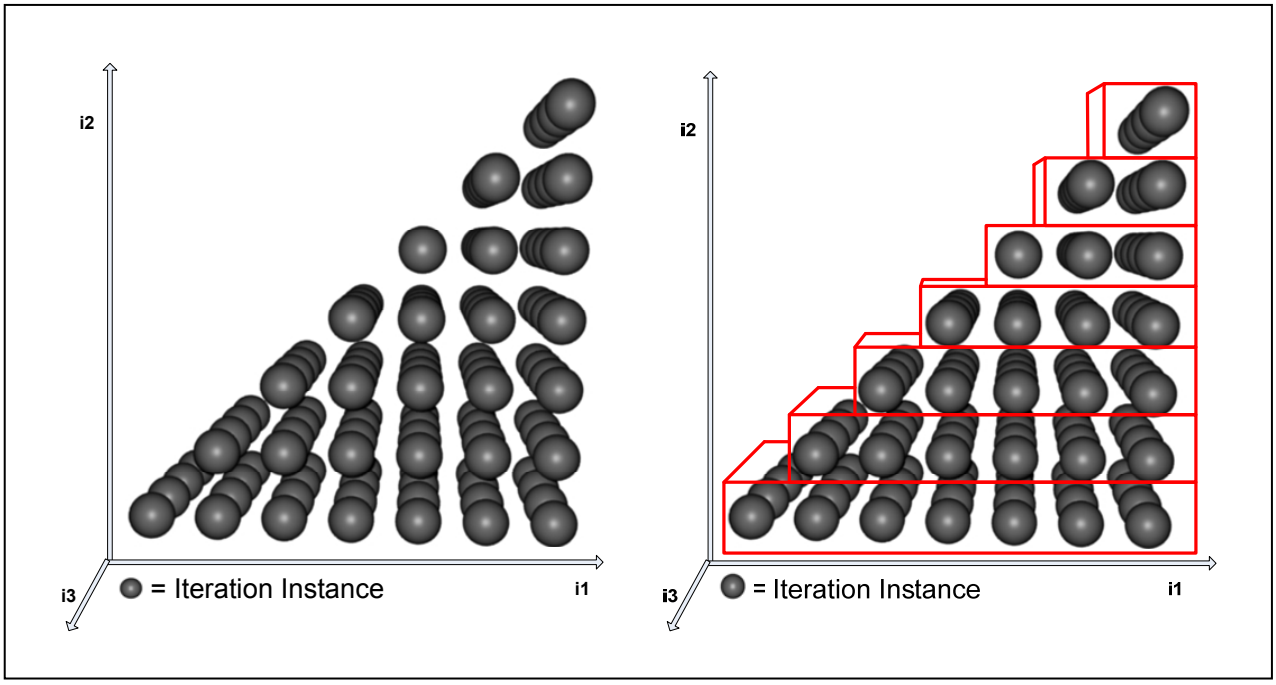

Figure 12. 3-dimensional iteration space partitioned on `i2`

The Last Write Tree [14] provides the information that describes which loop iterations are dependent on data produced by other iterations. This information can be used to determine which iteration instances require data computed by other iteration instances.

The Last Write Tree represents a mapping between an iteration instance $\vec{L}_r$ reading a memory location M and an iteration $\vec{L}_w$ that wrote to memory location M. In other words, the Last Write Tree represents the data dependency between two iteration instances. An important consideration for the last write tree is that there is no other iteration instance between $\vec{L}_w$ and $\vec{L}_r$ that modifies the value at memory location M.

Assigning partitions to real processors

The code produced by the Paraguin compiler allocates partitions to physical processors using a Block algorithm. The blocksize is a symbolic constant `blksz` in Figure 5 and Figure 6. The blocksize is determined by taking the ceiling of dividing the input problem size by the number of physical processors which is only known at runtime. For example, if the input problem size is 100 and the number of physical processors is 9, the value of the symbolic constant `blksz = ceiling(100/9) = 12`. Each processor would be allocated 12 partitions except for the last processor which would receive 4.

Redundant data packing behavior generalized

The packing of redundant data is a common behavior of parallelizing compilers that produce message passing code for distributed-memory systems for this reason: To parallelize a program, the work done by that program must be divided up into smaller

pieces and assigned to multiple processors. These partitions are usually the result of dividing the loop iterations. Since the largest amount of execution time is spent in loops, parallelizing the loops provides the largest potential increase in performance. The partitions or virtual processors are assigned to physical processors at run-time where they will be executed. How the work is partitioned and allocated to physical processors varies between parallelizing compilers, but the general concept is still the same. In the case of the Paraguin compiler, work (iterations instances) is divided up into partitions. These partitions are then assigned to physical processors.

Depending on the partitioning of the iteration space and the allocation of the partitions to real processors, a situation may arise that leads to the packing of redundant data. As Figure 8 shows, two partitions (`p4` and `p5`), are allocated to be executed on real processor `pid2` (receiving processor). These partitions are dependent upon data produced by another partition, `p2`, which is allocated to another real processor `pid1` (sending processor). After `pid1` has executed the work of `p2`, the data that was computed by `p2` must be packed and sent to `pid2` for use by the two partitions (`p4` and `p5`) that are dependent on that data. The data is packed for each dependence, and since each dependence resides on `pid2`, the same data will be packed twice by `pid2` into the message that will be sent to `pid2`. Since parallelizing compilers partition the work and assign that work to multiple real processors, redundant data packing may result.

Techniques for eliminating the amount of inter-processor communication can be applied. The amount of redundant data that is packed can be reduced by attempting to allocate dependent partitions onto the same real processor as the partition that computes

its required data. Allocation/placement methods are discussed in [3]. Nonetheless, dependencies between partitions allocated to different processors may still arise.

LooPo compiler

In [11], the LooPo parallelizing compiler is described. The compiler was studied to determine whether or not it might produce communication code that may pack redundant data. This behavior could arise in the following situation.

In the LooPo compiler, *operations* (iteration instances) are divided up and assigned to *tiles* (partitions). The tiles are then assigned to real processors. A situation may arise where two operations on two different tiles are assigned to the *same* real processor. If these operations require data computed by an operation on another real processor, then the data will be packed into the same message twice (once for each dependency).

Figure 13 shows a possible scenario of redundant data packing. Notice that Figure 13 shows a data dependence `d1` between operation 1 and operation 5, and `d2` between operation 1 and operation 6. After the value of memory cell `M` is computed by operation 1, it must be packed up and sent to any operations that are dependent on its value. The value of `M` will be packed by real processor `pid1` for the dependent operations 5 and 6. Since these operations have been assigned to the same real processor `pid2`, the value of `M` will be packed more than once into the same message by `pid1` as seen in the *writes* to the `pid2` message buffer in Figure 13.

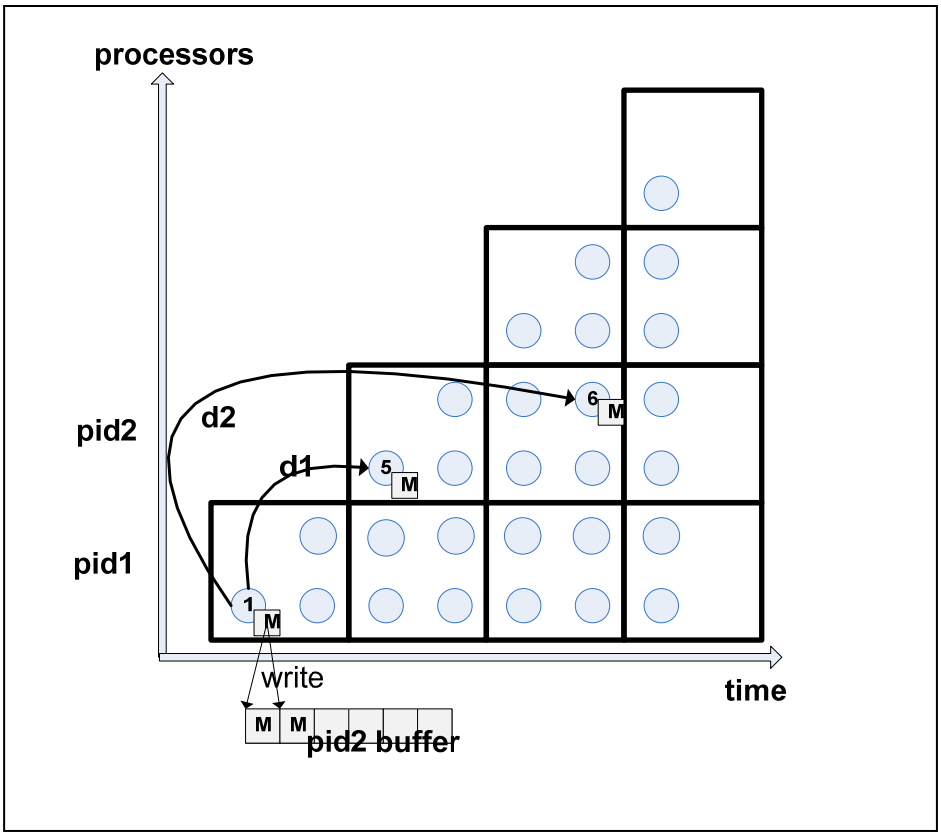

Figure 13. Dependency between processors (based on Figure 3.8 from [11])

Griebl [12] states that the "same data element may be sent multiple times: e.g., the use of the value just computed at different statements in the program is represented by different dependences, and every dependence is treated separately" [12]. The data just computed is packed for each dependence regardless of whether the data is destined for the same real processor. The author, after offering a solution, goes on to say that the solution would increase execution time of the generated code, "and in some cases this additional cost is higher than the communication cost saved due to the reduced communication volume" [12].

The solution in this paper does not increase execution time significantly. In fact, the solution reduces the number of instructions executed because loops are eliminated

within the loop nest. Chapter 6 shows that the savings of execution time in suppressing redundant data far exceed any overhead in doing so.

Chapter 4: Implementation

This chapter explains how the redundant data elimination technique has been implemented into the Paraguin compiler. As discussed in Chapter 2, redundant data is packed into messages when partitions mapped to the same physical processor are dependent on the same data computed by another physical processor. This behavior can be eliminated by replacing loops with assignment statements when the values of the subscript variables for the array elements that are being packed are independent of a particular loop's variable.

Each communication loop variable L_C must be checked to see if it affects the value of the subscript array variables of the array elements that are being packed. If the array subscript variables are not dependent upon L_C, then the L_C loop can be replaced with an assignment statement.

Step 1: Create a system of inequalities for each array element subscript variable

As discussed in Chapter 3, the Paraguin compiler represents the communication loop bounds as a system of inequalities S (Figure 14). The first column contains the constants. The second column contains the coefficients of `NP` which is a symbolic constant for the number of processors. The other columns are communication loop variables.

To discover which communication loops should be replaced with assignments statements, a new system of inequalities S_x must be generated for each array element subscript variable X that is also a communication loop variable. Variables used in the subscripts that are not loop variables do not contribute to redundant data unless they are

dependent on loop variables. Each new system S_x will be used in Step 2 to check for dependency between each subscript variable X and each loop variable L_C.

$$\begin{bmatrix}
101 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
100 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
100 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
101 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\
100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0
\end{bmatrix} \cdot \begin{bmatrix} 1 \\ \text{NP} \\ \text{pidr} \\ \text{pidw} \\ \text{pr} \\ \text{i1r} \\ \text{i2r} \\ \text{i3r} \\ \text{pw} \\ \text{i1w} \\ \text{i2w} \\ \text{i3w} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 14. System of inequalities S

The new system of inequalities S_x is generated by:

(a) creating an intermediate system S'_x by filtering *through* S and only keeping inequalities where the coefficient for X is nonzero; and

(b) filtering *away* (removing) inequalities from S'_x that contain inner loop bounds.

To perform step (1a), system S is filtered *through* to create S'_x. Filtering functions are used to filter a subset of inequalities from a given set of inequalities. A filter function requires a filter as a parameter. A filter is simply a vector containing values of zero for each variable in the system S except for a value of 1 for each variable we are interested in keeping or removing. For example, if X is `i2w` and `i2w` is the 11th level of nesting, then the filter would be $F_{\texttt{i2w}} = [000000000010]^T$. If X is the nth symbol in the list of symbols for system S, then the filter used for filtering *through* is $F_x = [\underbrace{0 \ldots 0}_{n-1} 10 \ldots 0]^T$. We can use this filter to create a system in which all inequalities where the coefficient of `i2w` is nonzero are either kept *or* removed depending upon which filter function is used.

Two filter functions are used by the algorithm: `filter_thru` and `filter_away` [9]. `Filter_thru` creates a new system which contains *only* the inequalities in S where the coefficients of the variables specified in the filter are nonzero. If the `filter_thru` function is passed a filter containing the values [001101000001], then the new system would contain *only* the inequalities where the coefficients of the 3rd, 4th, 6th and 12th variables are nonzero. All other inequalities are absent from the resulting system.

On the other hand, `filter_away` creates a new system which contains *all* the inequalities in S *except* for those inequalities where the coefficient of the variables specified in the filter are nonzero. So if `filter_away` is passed a filter containing the values [001101000001], the new system would contain all inequalities in S except for the inequalities where the coefficients of the 3rd, 4th, 6th and 12th variables are nonzero.

Simply, `filter_thru` *keeps* inequalities where the coefficients of variables specified in the filter are nonzero, and `filter_away` *throws away* inequalities where the coefficients of variables specified in the filter are nonzero.

To continue with step (1a), a filter F_X is created. System S is filtered *through* using filter F_X. This will result in a new system S'_X where the coefficient for X is nonzero for each inequality in the new system. Any inequality of S for which the coefficients for X is zero will be ignored and will not be present in S'_X. Thus system S'_X will contain only inequalities that contribute to the loop bounds of subscript variable X.

For example, in the send code in Figure 6, there are two subscript variables, `i2w` and `i3w`. Two filters are created, $F_{\texttt{i2w}} = [000000000010]^T$ and $F_{\texttt{i3w}} = [000000000001]^T$. Filtering *through* S using the filters $F_{\texttt{i2w}}$ and $F_{\texttt{i3w}}$ generates two new systems $S'_{\texttt{i2w}}$ and $S'_{\texttt{i3w}}$, respectively (Figure 15 and Figure 16).

$$\begin{bmatrix} 100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \mathrm{N} \\ \mathrm{pidr} \\ \vdots \\ \mathrm{i2w} \\ \mathrm{i3w} \end{bmatrix} \geqslant \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 15. System of inequalities $S'_{\texttt{i2w}}$

$$\begin{bmatrix} 101 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \mathrm{N} \\ \mathrm{pidr} \\ \vdots \\ \mathrm{i2w} \\ \mathrm{i3w} \end{bmatrix} \geqslant \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 16. System of inequalities $S'_{\texttt{i3w}}$

To perform step (1b) on each of these two new systems, any inner loop variables are filtered *away* (removed from the system) because the values of the variables are never dependent upon inner loops. Only outer loops have the possibility of affecting the value of the variables. The inner loop variables are unnecessary and can be removed from the system of inequalities for each variable X.

To remove inner loop inequalities from each S'_x, a filter is created where the value representing the column of each inner loop variable is 1. All other variables are 0. This filter is then used to filter *away* on S'_x. The resulting system S_x contains no inner loop inequalities.

To show an example of removing the inner loops, a different system other than those of Figure 15 and Figure 16 must be shown because filtering *away* does not actually alter the systems of Figure 15 and Figure 16. Suppose that X is `i1r` as in the receive code in Figure 5. Figure 17 shows the resulting system $S'_{\texttt{i1r}}$ after step (1a) is applied. In step (1b), the inner loops are removed from system $S'_{\texttt{i1r}}$ using the filter $\hat{F}_x = [000000111111]^T$. If X is the nth symbol in the list of symbols for system S, then the filter used for filtering *away* is $\hat{F}_x = [\underbrace{0 \ldots 0}_{n} 1 \ldots 1]^T$. The resulting system $S_{\texttt{i1r}}$ is shown in Figure 18.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 100 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \vdots \\ \texttt{i1r} \\ \texttt{i2r} \\ \texttt{i3r} \\ \texttt{pw} \\ \texttt{i1w} \\ \texttt{i2w} \\ \texttt{i3w} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 17. System of inequalities $S'_{\texttt{i1r}}$

$$\begin{bmatrix} 100 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \vdots \\ \text{i1r} \\ \text{i2r} \\ \text{i3r} \\ \text{pw} \\ \text{i1w} \\ \text{i2w} \\ \text{i3w} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figure 18. System of inequalities $S_{\texttt{i1r}}$ (inner loops removed)

Going back to the original scenario, in the case of system $S'_{\texttt{i2w}}$, all coefficients for the only inner loop variable, `i3w`, are already zero. Therefore, the filtering *away* of inner loop variables results in a new system $S_{\texttt{i2w}}$ that is identical to system $S'_{\texttt{i2w}}$ in Figure 15. Also, system $S'_{\texttt{i3w}}$ does not contain any inner loop inequalities because `i3w` is the innermost loop variable. Therefore, after filtering *away* inner loops, $S_{\texttt{i3w}}$ contains the same inequalities as the intermediate system $S'_{\texttt{i3w}}$.

$$\begin{bmatrix} 100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \text{N} \\ \text{pidr} \\ \vdots \\ \text{i2w} \\ \text{i3w} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 19. System of inequalities $S_{\texttt{i2w}}$ (same as Figure 15)

Each new system S_x is added to the set Ψ where $\Psi = \{ S_x \mid X \in$ loop variable and array subscript variable$\}$. The systems in Ψ will be examined in Step 2 to check for a dependency between variable X and each loop variable L_C. Figure 20 displays pseudocode for Step 1.

```
For each X | X ϵ array element subscript variable and loop variable){
        F_x = Create_filter(X)
        S'_x = Filter_thru(S using F_x)
        F̂_x = Create_inner_loop_filter(X)
        S_x = Filter_away (S'_X using F̂_x)
        Add S_x to Ψ
}
```

Figure 20. Create system of inequalities for each array element subscript variable and project away inner loops (Step 1 pseudocode)

Step 2: Dependence check

Each communication loop variable L_C is inspected starting with the innermost loop and moving outward until we reach a loop where the index variable is either `pidw` or `pidr`. For each variable L_C, a filter F_{L_C} must be created to filter out the expressions from each S_x that do not involve L_C. Suppose L_C is the `pr` variable. Then $F_{\texttt{pr}} = [000010000000]^T$ where `pr` is the 5th loop in the loop nest.

Filter F_{L_C} is used to filter *through* each system of inequalities S_x in Ψ. Filtering results in a new system $\hat{S}_x$. If $\hat{S}_x$ is empty, then X is independent of L_C. This means that the coefficient of L_C in each inequality in the system of inequalities S_x was zero. In other words, no inequalities of S_x involve the variable L_C and therefore X is independent of L_C.

For example, suppose the current L_C variable is `pr`. Filtering *through* system $S_{\texttt{i2w}}$ in Figure 19 using filter $F_{\texttt{pr}} = [000010000000]^T$ results in an empty system $\hat{S}_x = [\ ]$ because there are no inequalities in $S_{\texttt{i2w}}$ where the coefficient for `pr` is nonzero. Therefore the array subscript variable `i2w` is independent from the `pr` loop.

However, if after filtering *through* S_x there is at least one inequality in the resulting system $\hat{S}_x$, then X is dependent on L_C. This means that the coefficient of L_C in at least one of the inequalities in S_x was nonzero.

Now suppose that `i1r` is the current L_C variable. Filtering *through* system $S_{\texttt{i2w}}$ in Figure 19 using the `i1r` filter $F_{\texttt{i1r}} = [000001000000]^T$ results in the nonempty system $\hat{S}_{\texttt{i2w}}$ shown in Figure 21 because there is at least one inequality in $S_{\texttt{i2w}}$ where the coefficient for `i1r` is nonzero. Therefore the array subscript variable `i2w` is dependent on the `i1r` loop.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \vdots \\ \text{i1r} \\ \vdots \\ \text{i2w} \\ \text{i3w} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Figure 21. System $\hat{S}_{\texttt{i2w}}$: the result of filtering *through* $S_{\texttt{i2w}}$ by $F_{\texttt{i1r}}$

The algorithm must also handle the following case: If X is dependent on L_C, then X is transitively dependent on all outer loop variables on which L_C is dependent. The algorithm must take steps to ensure that any outer loop variables on which L_C is dependent are not replaced with an assignment statement. Therefore, Step 1 must also be applied to L_C if X is dependent on L_C. A system of inequalities S_{Lc} is generated for L_C and added to the set of systems Ψ to be checked for any dependence on outer loop variables. If this new system is not created and checked, then any outer loop variable L_O upon which L_C is dependent, when checked against X, could be found to be independent of X. Loop variable L_O would then be replaced with an assignment statement (see Step 3) and the generated code would be incorrect; a loop on which X is dependent would be

eliminated. Checking for dependencies between L_C and other communication loop variables insures that no necessary loops will be replaced with assignment statements. Figure 22 displays Step 2 pseudocode.

```
For each communication loop Lc {
    boolean dependent = FALSE
    //Step 2
    For each system of inequalities Sx ϵ Ψ {
        Ŝx = Filter_thru(Sx using FLc)
        If Ŝx is not empty then {
                dependent = TRUE;
                Create SLc by Step 1. Add SLc to Ψ
                break;
        }
    }
    //Step 3
    IF dependent = FALSE
        Set upperbound(Lc) = lowerbound(Lc)
}
```

Figure 22. Pseudocode for dependence check (Steps 2 and 3)

Step 3: Replace the loop with an assignment statement

If the current loop L_C is found to be independent of each subscript variable X, then it can be safely replaced with an assignment statement. To do so, the loop's upper bound is set equal to its lower bound; therefore, the loop has only one iteration. This is known as a degenerate loop. The Paraguin compiler later replaces degenerate loops with assignments statements. For example, the loop

```
for(pr = 2+blksz*pidr; pr <= min(1+blksz+blksz*pidr, N);
    pr++){
```

would have its upper bound set to its lower bound. The result would look like

```
for(pr = 2+blksz*pidr; pr <= 2+blksz*pidr; pr++){
```

The Paraguin compiler would later replace the loop with

```
pr = 2+blksz*pidr;
```

The replacement of a degenerate loop with an assignment statement is not essential as it provides a performance improvement of only a few machine instructions. It is performed mainly to clean up the resulting output program.

After Steps 1 through 3 have been applied for each communication loop variable L_C, all independent loops have been replaced with assignment statements, and all dependent loops have been left untouched. Appendix A contains the code that was written to implement these steps.

Chapter 5: Testing and analysis

Tests

Once the data redundancy elimination technique was implemented in the Paraguin compiler, several test programs were created to determine the effect of the technique on the performance of the parallel code generated by the compiler. Three programs that implement the code of Figure 1 were produced:

- Program 1 – A sequential program executed on a single processor (produced using the gcc compiler).
- Program 2 – An MPI program generated by the Paraguin compiler without the data redundancy elimination optimization technique described in this paper.
- Program 3 – An MPI program generated by the Paraguin compiler using the data redundancy elimination optimization technique described in this paper.

These tests measure the runtime of the programs in seconds. The runtime of the parallel programs includes the time to scatter the input (communication of the initial input data to all processors) and gather the results back to the master processor, but it does not include the time for I/O. The runtime for the sequential program includes the time to execute the program segment of Figure 1, but also does not include the time for I/O. The sum of all bytes communicated between processors during runtime of the parallel programs is also measured.

With these three tests, it can be determined not only if Program 3 provides an improvement over Program 2, but also how well a parallel program generated by the compiler performs against a sequential program.

In addition to these tests, the time it takes the Paraguin compiler with the optimization to parallelize the code in Figure 1 was measured and compared with the compile time of the Paraguin compiler with the optimization technique turned off and the compile time of the gcc compiler.

To test the programs, each parallel program was run on a cluster consisting of 4 Dell PowerEdge 1850s with 2 Intel Dual Core 2.8 GHz processors with 12 Gbytes of memory and 8 Sunfire X4100s with 2 AMD Dual Core 2.6 Ghz processors with 8 Gbytes of memory. One machine is used as the head node, which is used only for interactive use and submitting jobs. The sequential program was run on the head node. The machines are connected using a Cisco 100 Mbps switch. Programs 2 and 3 were executed using 4 through 44 processors in increments of 4 e.g. 4, 8, 12, 16, etc. The problem input size ranged from 100 to 1000 in increments of 100 e.g. 100, 200, 300, etc. Each program was run 10 times at each processor/input size combination e.g. 4/100, 8/100, 12/100, etc., and the runtime results presented in this paper are the averages of the 10 runs. The total sums in bytes of all messages passed for each run were also collected.

Results

The compile time of the code in Figure 1 averaged across 10 runs by the gcc compiler and the Paraguin compiler with and without the optimization were 1 second, 3.7 seconds and 3.5 seconds, respectively.

The following graphs display the results of the above tests. In each of the runtime graphs, the runtime for Program 1 (the sequential program) is also graphed for comparison. Figure 23 shows the runtime in seconds for Program 2. One can observe that the parallel program experienced significant slowdown as the problem size increased.

This is due to the large messages that contain redundant data and the time it takes to build and transmit those messages. One can also observe that some data points are missing for certain processor/input size combinations. These combinations resulted in messages so large that the program exceeded the memory capacity of the machines. The sequential program, without the need to communicate large messages, performed much better than Program 2.

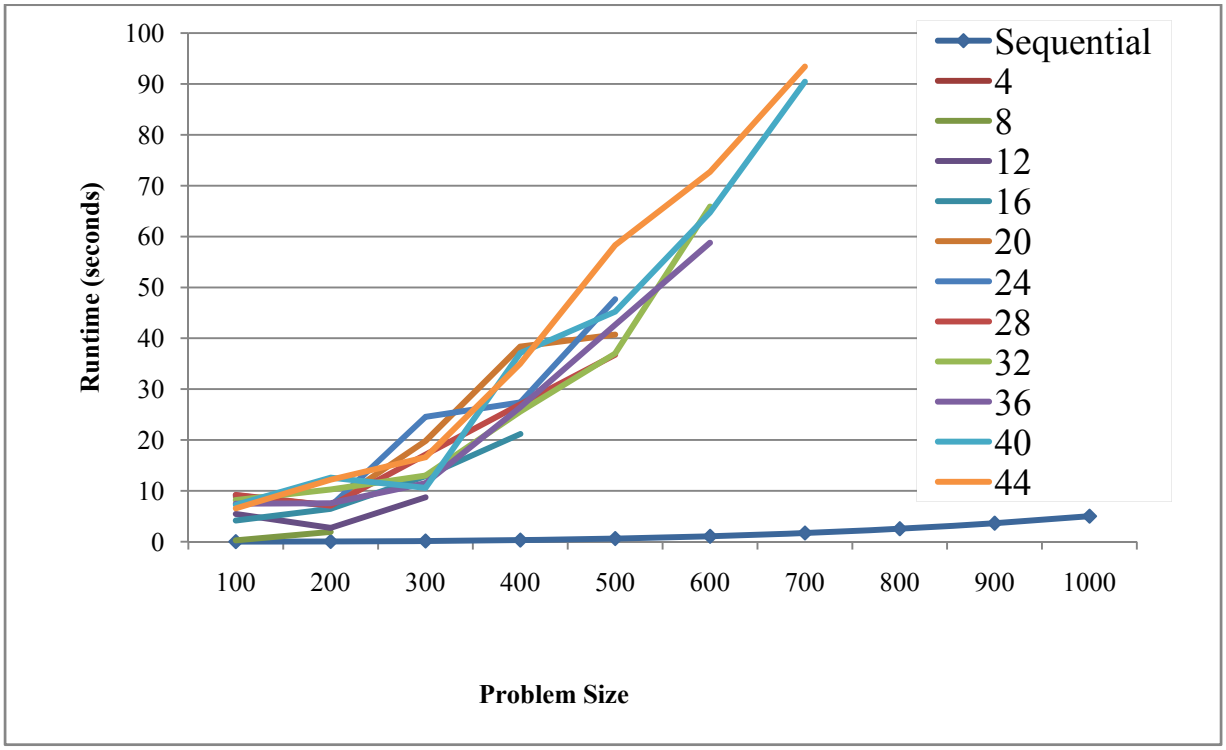

Figure 23. Runtime without optimization

Figure 24 shows the runtime of Program 3. The elimination of redundant data resulted in smaller message sizes which decreased the runtime for all processor/input size combinations to less than 10 seconds. The improvement of using the optimization technique proposed in this paper is on average an 88% reduction in runtime. The performance of Program 3 is comparable to sequential though it does not outperform

sequential. The reasons for this are analyzed further in the *Communication and Execution Overlap* section in Chapter 6.

Although hard to see in Figure 24, there are less data points missing. The reduction in message sizes also allows Program 3 to run larger problem sizes for all numbers of processors because the smaller messages do not exceed the memory capacity of the machines.

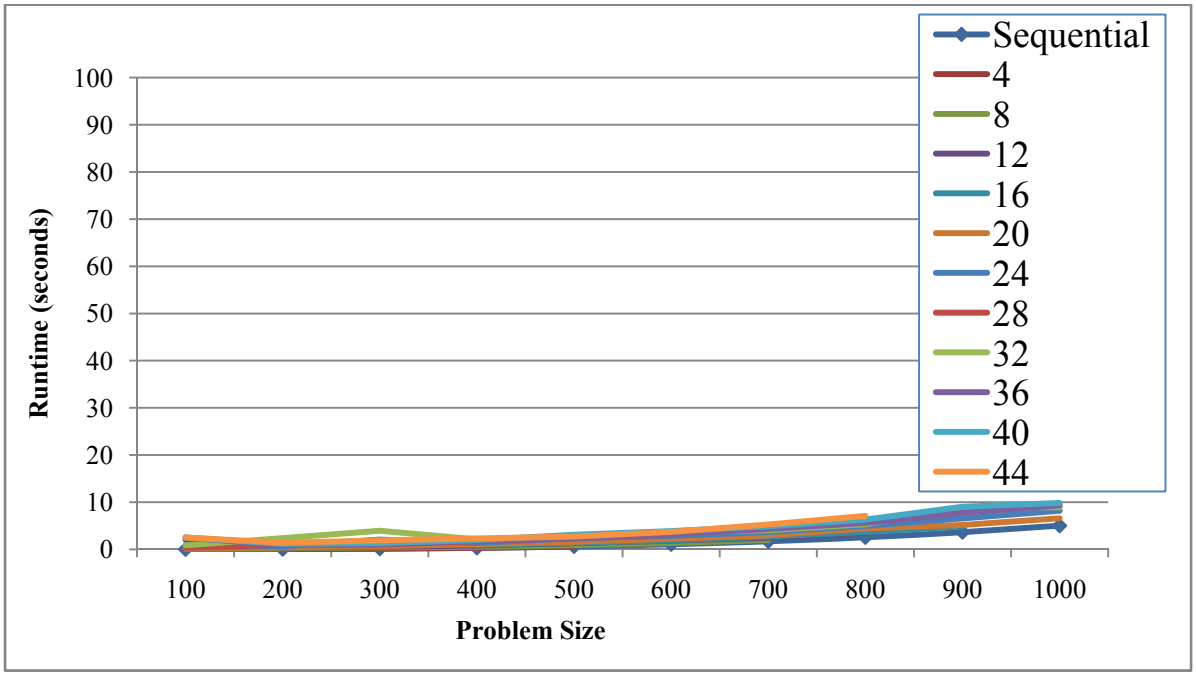

Figure 24. Runtime with optimization

Figure 25 and Figure 26 display the total number of bytes communicated between processors for Program 2 and Program 3, respectively. The redundant data elimination technique implemented in Program 3 greatly reduces the sizes of the messages which is the reason for the runtime improvement between Figure 23 and Figure 24. Furthermore, the reduction of the message sizes allows the program to run with larger problem sizes to

since the memory capacity of the machines is not exceeded. When comparing Program 3 to Program 2, the reduction in the number of bytes transmitted is about 93% on average.

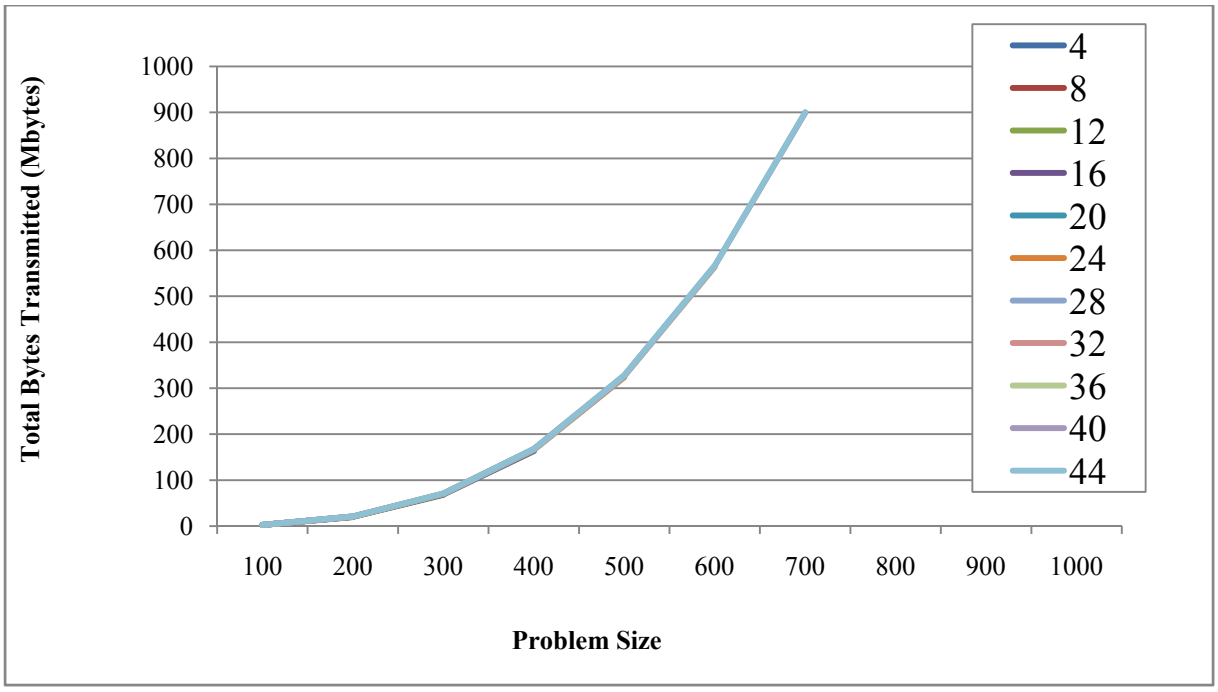

Figure 25. Total bytes transmitted without optimization

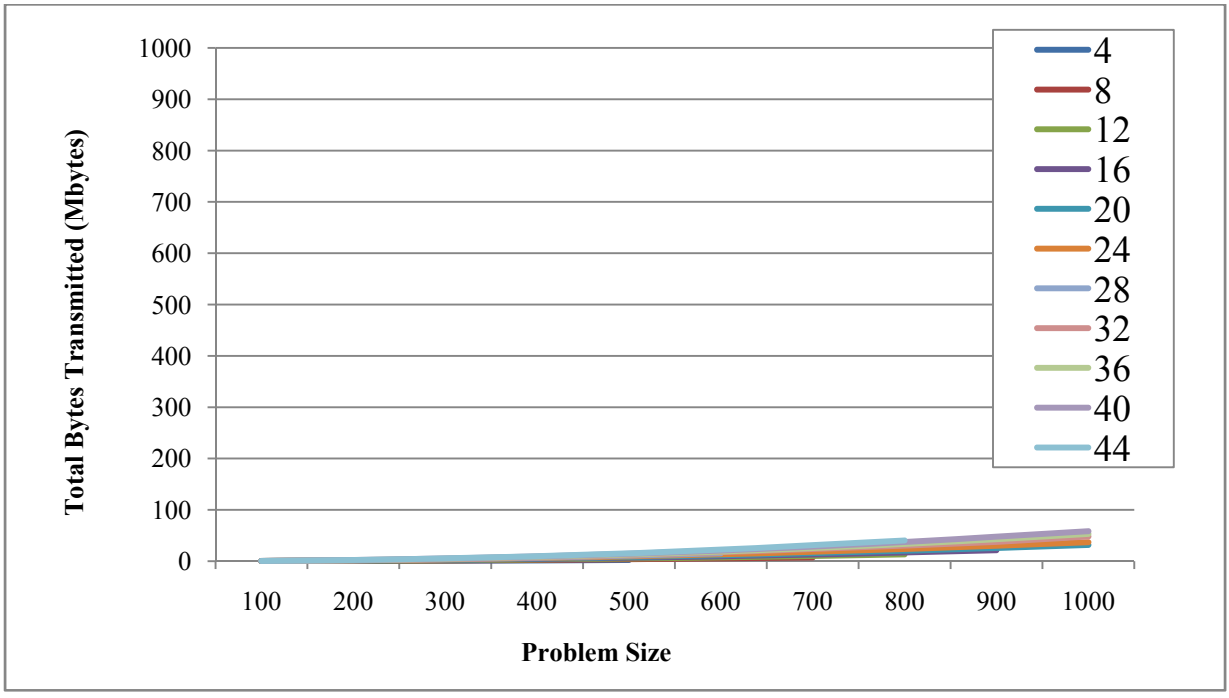

Figure 26. Total bytes transmitted with optimization

Chapter 6: Conclusions

Communication and execution overlap

The parallel program does not achieve speeds greater than the sequential program even though many more processors are utilized. The reason is that the Paraguin compiler currently does not overlap communication code with execution code. At this time, the compiler produces code where each processor must wait to receive all data it is dependent on before it can begin executing its allocated workload. Also, the entire workload for a processor must be completed before it can begin sending messages to other processors that depend upon the data that it has just computed. This kind of code arrangement (Receive data -> Execute work -> Send data) is basically sequential execution, but with the added overhead of having to communicate data. This is why the parallel program cannot currently meet or exceed the performance of a sequential program.

As the Paraguin compiler is generating the parallel code, it attempts to overlap communication with execution code. To do so, it must have access to the system of inequalities that represent the loops. The compiler then attempts to recreate the system of inequalities from the new loops. However, by this time the code has become so complex that the routines within the SUIF library that convert loops to a system of inequalities is unable to do so.

When the hand written code was created during an early phase of this project, it was known that the code would not outperform the sequential program. However, we believed that once the technique was implemented in the Paraguin compiler, the compiler would be able to overlap communication with execution code thus increasing

performance over that of sequential. That did not turn out to be the case, as the tests in this paper show.

The process of overlapping communication with computation will have to be redesigned to deal with the added complexity. We expect to see real speedup over sequential execution once overlapping is redesigned and implemented.

Conclusions

The focus of this paper is the implementation of an optimization technique in the Paraguin compiler. The Paraguin compiler is an automatically parallelizing compiler which generates parallel code given a sequential program. The compiler exhibits a problem that is addressed by this paper. That problem is that the compiler produces code which packs redundant data into messages sent to other processors. This causes the message sizes to be larger than necessary which increases the runtime of the parallel program. It also causes the program to fail when running the program using certain combinations of problem input sizes and number of processors because the size of the messages may exceed the memory capacity of the machines executing the program.

A technique is proposed in this paper to eliminate the redundant data packing. The technique replaces loops where the loops counter variable does not affect the value of the subscript variables of the array being packed. The optimization was implemented in the Paraguin compiler.

Test programs were created to observe the effect of the technique on programs generated by the Paraguin compiler. The Elimination Step of Gaussian Elimination in

Figure 1 was used as the application program for testing the effects of implementing the technique discussed in this paper.

When test results of the runtime of the parallel program were compared with the runtime of the sequential program, the parallel program was still slower than the sequential program. This is due to the fact that the Paraguin compiler failed to overlap communication code with execution code with the optimization. This results in code that runs as if it were sequential code, but with the increased overhead to communicate data between processors. In the future, once communication and execution overlap is implemented into the Paraguin compiler, the generated parallel code is expected to outperform sequential execution.

When the parallel program using the optimization proposed in this paper was compared with the parallel program without the optimization, the test results showed that the technique eliminated the redundant data being packed into messages and reduced the total number of bytes transmitted during program execution by 93%. As a consequence, the runtime of the parallel program with the technique was reduced by 88%. The reduction in the size of the messages also allowed the program to be run using combinations of problem input sizes and numbers of processors that had previously failed due to the memory capacity of the machines being exceeded.

The technique provides a significant improvement in the performance of parallel code automatically generated by the Paraguin compiler.

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)*, 126—138, Albuquerque, New Mexico, June 1993.

[2] 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)*, 39—50, Williamsburg, Virginia, April 21-24, 1991.

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

[4] C.S. Ferner, "Revisiting communication code generation algorithms for message-passing systems," *International Journal of Parallel, Emergent and Distributed Systems (JPEDS)*, Vol. 21 No. 5, 323—344, October 2006.

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

[6] M. Hall, D. Padua, and K. Pingali, "Compiler research: The next 50 years," Communications of the ACM, Vol. 52 No. 2, 60—67, February 2009.

[7] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine partitions," *Parallel Computing*, Vol. 24 No. 3-4, 445—475, 1998.

[8] P. J. Martin and C. S. Ferner, "Suppressing independent loops in packing/unpacking loop nests to reduce message size for message-passing code," in the *Proceedings of the PDPTA'07 – The 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (as part of WORLDCOMP'07)*, Las Vegas, NV, June 15-18, 2007.

[9] "The SUIF Compiler System," Computer Science Department, Stanford University, http://suif.stanford.edu/.

[10] J. Xue, "*Loop tiling for Parallelism*," Kluwer Academic Publishers, Boston, MA, 2000.

[11] M. Classen, "Automatic Code Generation for Distributed Memory Architectures in the Polytope Model", Diplomarbeit, Universitat Passau, 2005.

[12] M. Griebl, "Automatic Parallelization of Loop Programs for Distributed Memory Architectures", Habilitation Thesis, Universitat Passau, 2005.

[13] S. P. Amarasinghe, "Parallelizing Compiler Techniques Based on Linear Inequalities", Stanford University, 1997.

[14] 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 SIGPLA N-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 10-13, pp. 2-15, 1993.

Appendix A

Code for Redundancy Elimination Technique

Appendix A

For this project, 1 function was created and 3 functions were modified.

New Function: build_variable_nsli

```
/*
 * Function: build_variable_nsli
 *
 * Parameters: This function takes the system of inequalities S
 * (nsli) and the index of the variable X (sym_count)to be used
 * as the filter.
 *
 * Returns: a new system Sx that contains only inequalities
 * pertaining to the loop bounds of X, with the inner loops
 * filtered out.
 */
named_symcoeff_ineq build_variable_nsli(
                      named_symcoeff_ineq nsli, int sym_count) {

     //Create filter Fx of size N - N is # of loop variables
     constraint myfilter(nsli.n());

     //Set the variable X we want to filter for to 1
     myfilter[sym_count] = 1;

     //Create new system S'x that contains all inequalities
     named_symcoeff_ineq nsli_for_op =
                          nsli.filter_thru(&myfilter, NULL, 0);

     //Declare new system Sx
     named_symcoeff_ineq temp_nsli;

     //Reset every value in the filter to 0
     myfilter = 0;


     //Creating filter that contains all inner loops
     sym_count++;
     for(sym_count; sym_count < nsli.n();
          sym_count++){
          myfilter[sym_count] = 1;
     }

     //Remove all inner loop variables using new filter
     temp_nsli = nsli_for_op.filter_away(&myfilter, NULL, 0);

     //return the new system Sx
     return new named_symcoeff_ineq(temp_nsli);
}
```

Modified Function: build_receive_loop

```
/*
 * Function: build_receive_loop
 *
 * This function produces the receive loop communication code.
 * The code between "BEGIN NEW CODE" comments and the "END NEW
 * CODE" comments were added for this project.
 *
 *
 */
void build_receive_loop (tree_node_list * tnl, tree_node * tn,
       named_symcoeff_ineq nsli, var_sym * pidw, in_array *
           instr, block lb, block ub, int mapping, int tag, int
                numSymConst)
{
    if (~nsli) {

        return;
    }

    //Various variable declarations
    int i;
    block in_blk (instr);
    // elem_size() returns the number of BITS, not the number of
     //bytes.
    block nitem_blk((int) instr->elem_size() / 8);
    block buffer_blk(buffer_var);
    block buffer_sz_blk(GUIN_BUFFER_SZ);
    block position_blk(position_var);
    block NP_blk (NP_var);
    block blksz_blk (blksz_var);
    block Comm_blk(Comm_var);
    block Byte_type_blk(Byte_type_var);
    block Packed_type_blk(Packed_type_var);
    block any_source_blk(MPI_ANY_SOURCE);
    block status_blk(status_var);
    block mypid_blk(my_pid_var);
    block pidw_blk(pidw);
    block map_blk(pidw_blk);

    block unpack_proc_blk(unpack_proc);
    block unpack_blk (block::CALL(unpack_proc_blk,
                 buffer_blk, buffer_sz_blk, position_blk.addr(),
                    in_blk, nitem_blk, Byte_type_blk, Comm_blk));

    block code(unpack_blk);
    var_sym * loop_var;
    block lvar_b;

    i = nsli.n() - 1;
```

```
    name_table nt = nsli.cols();

// BEGIN NEW CODE //

      //count - # of symbols in array subscripts
      // used to create an operand array
      int count = 0;
      int x;

      //Loop through each dimension of the array.
      //Need to gather count of subscript variables to
      //create an operand array

      for(x=0; x < instr->dims(); x++){

            //IF the object at index x is an instruction
            //then grab number of sources and add them to count
            //ELSE IF the object at index x is a symbol
            //then increment the variable counter

            if (instr->index(x).is_instr())
                  count += instr->index(x).instr()->num_srcs();
            else if (instr->index(x).is_symbol())
                  count += 1;
      }

      operand *op_array;

      //Create operand array using 'count' from earlier
      op_array = (operand *) malloc (count * sizeof (operand));

      //Loop through array dimensions and store all subscript
      //array variables
      int temp = 0;
      int j;
      for(x = 0; x < instr->dims(); x++){
            operand op1 = instr->index(x);

            if (op1.is_instr()) { //if operand is instruction

                  instruction *ins = op1.instr();

                  //Loop through instruction sources, grab source
                  //operands, store them in operand array
                  for(j=0; j < ins->num_srcs(); j++){
                        op_array[temp++] = ins->src_op(j);
                  }

            } else if (op1.is_symbol()) { //else if symbol
                  op_array[temp++] = op1; //store operand
            }
```

```
      else
            //unknown operand
            printf("different kind of op\n");
}

boolean independent;

//Create filter with size N - N is # of loop variables
constraint myfilter(nsli.n());
myfilter = 0;

//Create a list PSI to hold new systems Sx
dependence_check_list *dcl = new dependence_check_list;


//Loop through all communication loop variables Lc to
//determine the indices of the subscript variables in
//the system of inequalities S
for(x=1; x < nsli.n(); x++){
      //Looping through each subscript operand
      for(j=0; j < count; j++){

            //If this variable in S is one of the subscript
            //operands...
            //Then call build_variable_nsli which
            //(1) makes the filter Fx,
            //(2) filters thru to obtain new system S'x,
            //(3) filters thru to remove all inner loop
            //    variables which results in new system Sx.
            //
            //Add new system Sx (temp_nsli) to PSI (dcl)

            if(nt[x].var() == op_array[j].symbol()){
                  named_symcoeff_ineq temp_nsli =
                              build_variable_nsli(nsli, x);
                  dcl->append(new
                              named_symcoeff_ineq(temp_nsli));
            }
      }
}

//Save this system because when it gets projected away, it
//seems to mess up the bounds and cause false dependence
//checks. Use this system to filter thru and create other
//systems.
named_symcoeff_ineq original_nsli = new
                              named_symcoeff_ineq(nsli);


named_symcoeff_ineq test_nsli = new
                        named_symcoeff_ineq(nsli);
```

```
// END NEW CODE //

     //Loop through all communications variables
     //starting with innermost loop variable
     while (i > numSymConst) {

// BEGIN NEW CODE //
          independent= false;
          myfilter = 0;

          //Create F_lc
          myfilter[i] = 1;

          named_symcoeff_ineq dnsli;
          named_symcoeff_ineq *temp_nsli;
          named_symcoeff_ineq retval;

          dependence_check_list_iter dcl_iter(dcl);

          //Iterate through PSI (dcl)
          while(!dcl_iter.is_empty()){
               temp_nsli = dcl_iter.step();

               //Filtering through Sx using F_lc creates
               //new system S(hat)x (dnsli).
               dnsli = temp_nsli->filter_thru(&myfilter,NULL,0);

               //IF the resulting system S(hat)x is empty
               if(dnsli.m() <= 0){//If zero, then independent
                 independent= true;

               }
               else{

                 independent = false;

                 retval = build_variable_nsli(original_nsli, i);

                 dcl->append(new named_symcoeff_ineq(retval));
                    break;
                 //Need to break, otherwise it will loop through
                 //and the next operand may set independent =
                 //true which would then remove the loop that
                 //this particular operand is dependent on
               }
          } //end 2nd while


          test_nsli.filter_away(&myfilter, NULL, 0);
// END NEW CODE //
```

```
            loop_var = nt[i].var();
            lvar_b.set (loop_var);

            if (loop_var == my_pid_var || loop_var == pidw_var) {

                  if (loop_var == pidw_var) {

                  //On the receive side, the tag should be THIS
                  //processor id (mypid), NOT pidw, because the
                  //send side will be sending
                  //the packet with a tag of the receive side
                  //pidr.

                        block tag_blk(block(tag) * (ub - lb + 1) +
                                                mypid_blk);

                        block stmts (block::CALL(receive_proc,
                              buffer_blk, buffer_sz_blk,
                              Packed_type_blk, map_blk,
                              tag_blk, Comm_blk,
                              status_blk.addr()),
                              block(position_blk = block(0)),
                              code);


                        // Suppress receiving messages from the
                        //same processor.
                        if (!Send2Self)
                              code.set(block::IF(mypid_blk !=
                                    map_blk, stmts));
                        else
                              code.set(stmts);


                        code.set(addNewLoop(&nsli, code, loop_var,
                                                      i, TRUE));

                  } else {

                  code.set(addNewLoop(&nsli, code, loop_var, i,
                                    TRUE, NULL, NULL, 0));
                  }

            } else {
// BEGIN NEW CODE - added two arguments (1, independent) //
                  //the degenerate variable is a command line flag
                  // used to turn on and off the optimization
                  if(degenerate)
                        //Call addNewLoop with independent variable
                        code.set(addNewLoop(&nsli, code, loop_var,
                          i, TRUE, NULL, NULL, 1, 1, independent));
```

```
                else //Call addNewLoop, dont use optimization
                     code.set(addNewLoop(&nsli, code, loop_var,
                                         i, TRUE, NULL, NULL, 1));
// END NEW CODE //

           }// end if-else

           nsli.project_away(immed(loop_var));

           i--;

     } //end 1st while


    tree_node_list *new_tnl = code.make_tree_node_list();
    if (tn != NULL)
        tnl->insert_before (new_tnl, tn->list_e());
    else
        tnl->append(new_tnl);
}

// END RECEIVING CODE//
```

Modified function: build_send_loop

```
/*
 * Function: build_send_loop
 *
 * This function produces the send loop communication code.
 * The code between "BEGIN NEW CODE" comments and the "END NEW
 * CODE" comments were added for this project.
 *
 *
 */
void build_send_loop (tree_node_list * tnl, named_symcoeff_ineq
          nsli, var_sym * pidr, in_array * instr, block lb,
               block ub, int mapping, int tag, int numSymConst)
{

    if (~nsli) {
        return;
    }

    //Various variable declarations
    int i;
    block in_blk (instr);
    // elem_size() returns the number of BITS, not the number of
     //bytes.
    block nitem_blk((int) instr->elem_size() / 8);
    block buffer_blk(buffer_var);
    block buffer_sz_blk(GUIN_BUFFER_SZ);
    block position_blk(position_var);
    block NP_blk (NP_var);
    block blksz_blk (blksz_var);
    block Comm_blk(Comm_var);
    block Byte_type_blk(Byte_type_var);
    block Packed_type_blk(Packed_type_var);
    block any_source_blk(MPI_ANY_SOURCE);
    block mypid_blk(my_pid_var);
    block pidr_blk(pidr);
    block map_blk(pidr_blk);

    block pack_proc_blk(pack_proc);
    block pack_blk (block::CALL(pack_proc_blk, in_blk, nitem_blk,
                    Byte_type_blk, buffer_blk, buffer_sz_blk,
                                position_blk.addr(), Comm_blk));


    block code(pack_blk);
    var_sym * loop_var;
    block lvar_b;

    i = nsli.n() - 1;
    name_table nt = nsli.cols();
```

```
// BEGIN NEW CODE //
     //count - # of symbols in array subscripts
     //used to create an operand array
     int count = 0;
     int x;

     //Loop through each dimension of the array.
     //Need to gather count of subscript variables to
     //create an operand array

     for(x=0; x < instr->dims(); x++){

          //IF the object at index x is an instruction
          //then grab number of sources and add them to count
          //ELSE IF the object at index x is a symbol
          //then increment the variable counter

          if (instr->index(x).is_instr())
               count += instr->index(x).instr()->num_srcs();
          else if (instr->index(x).is_symbol())
               count += 1;
     }


     operand *op_array;

     //Create operand array using 'count' from earlier
     op_array = (operand *) malloc (count * sizeof (operand));

     //Loop through array dimensions and store all subscript
     //array variables
     int temp = 0;
     int j;
     for(x = 0; x < instr->dims(); x++){
          operand op1 = instr->index(x);

          if (op1.is_instr()) { //if operand is instruction

               instruction *ins = op1.instr();

               //Loop through instruction sources, grab source
               //operands, store them in operand array
               for(j=0; j < ins->num_srcs(); j++){
                    op_array[temp++] = ins->src_op(j);
               }

          } else if (op1.is_symbol()) { //else if symbol
          op_array[temp++] = op1;       //store operand
          }
          else
```

```
                //else unknown operand
                printf("different kind of op\n");
        }//end for

        boolean independent;

        //Create filter with size N - N is # of loop variables
        constraint myfilter(nsli.n());
        myfilter = 0;


        //Create a list PSI to hold new systems Sx
        dependence_check_list *dcl = new dependence_check_list;



        //Loop through all communication loop variables Lc to
        //determine the indices of the subscript variables in
        //the system of inequalities S
        for(x=1; x < nsli.n(); x++){
             //Looping through each subscript operand
             for(j=0; j < count; j++){

                  //If this variable in S is one of the subscript
                  //operands...
                  //Then call build_variable_nsli which
                  //(1) makes the filter Fx,
                  //(2) filters thru to obtain new system S'x,
                  //(3) filters thru to remove all inner loop
                  //    variables which results in new system Sx.
                  //
                  //Add new system Sx (temp_nsli) to PSI (dcl)

                  if(nt[x].var() == op_array[j].symbol()){
                       named_symcoeff_ineq temp_nsli =
                                 build_variable_nsli(nsli, x);
                       dcl->append(new
                                 named_symcoeff_ineq(temp_nsli));
                  }
             }
        } //end for

        //Save this system because when it gets projected away, it
        //seems to mess up the bounds and cause false dependence
        //checks. Use this system to filter thru and create other
        //systems.
        named_symcoeff_ineq original_nsli = new
                              named_symcoeff_ineq(nsli);
// END NEW CODE //
```

```
      //Loop through all communications variables
      //starting with innermost loop variable
      while (i > numSymConst) {
// BEGIN NEW CODE //
            independent= false;
            myfilter = 0;

            //Create F_lc
            myfilter[i] = 1;

            named_symcoeff_ineq dnsli;
            named_symcoeff_ineq *temp_nsli;
            named_symcoeff_ineq retval;


            dependence_check_list_iter dcl_iter(dcl);

            //Iterate through PSI (dcl)
            while(!dcl_iter.is_empty()){
                  temp_nsli = dcl_iter.step();

                  //Filtering through Sx using F_lc creates
                  //new system S(hat)x (dnsli).
                  dnsli = temp_nsli->filter_thru(&myfilter,NULL,0);

                  //IF the resulting system S(hat)x is empty
                  if(dnsli.m() <= 0){//If zero, then independent
                        independent= true;

                  }
                  else{

                    independent = false;

                    retval = build_variable_nsli(original_nsli, i);

                    dcl->append(new named_symcoeff_ineq(retval));
                    break;
                   //Need to break, otherwise it will loop through
                    //and the next operand may set independent =
                   //true which would then remove the loop that
                    //this particular operand is dependent on
                  }
            } /end 2nd while
// END NEW CODE //


            loop_var = nt[i].var();
            lvar_b.set (loop_var);

            if (loop_var == my_pid_var || loop_var == pidr_var) {
```

```
                if (loop_var == pidr_var) {

                     // On the send side, the tag should be the
                     //receiving processor id.

                     block tag_blk(block(tag) * (ub - lb + 1) +
                                                  pidr_blk);
                     block stmts (block(position_blk =
                          block(0)), code,
                          block::CALL(send_proc, buffer_blk,
                          position_blk, Packed_type_blk,
                          map_blk, tag_blk, Comm_blk));

                     // Suppress messages that our sent from
                     // this virtual processor to itself.
                     if (!Send2Self) {
                          code.set (block::IF (mypid_blk !=
                                         map_blk, stmts));
                     } else {
                          code.set (stmts);
                     }

                     code.set(addNewLoop(&nsli, code, loop_var,
                                          i, FALSE));
                } else {

                     code.set(addNewLoop(&nsli, code, loop_var,
                               i, FALSE, NULL, NULL, 0));
                }

           } else {

// NEW CODE - added two arguments (1, independent) //
                //the degenerate variable is a command line flag
                // used to turn on and off the optimization
                if(degenerate)
                   //Call addNewLoop with independent variable
                   code.set(addNewLoop(&nsli, code, loop_var, i,
                      FALSE, NULL, NULL, 1, 1, independent));
                else //Call addNewLoop, dont use optimization
                   code.set(addNewLoop(&nsli, code, loop_var, i,
                      FALSE, NULL, NULL, 1));
// END NEW CODE //

           }//end if-else

           nsli.project_away(immed(loop_var));

           i--;
      } //end 1st while
```

```
    tree_node_list *new_tnl = code.make_tree_node_list();
    tnl->append (new_tnl);
}

//END SEND CODE//
```

Modified Function: addNewLoop

```
/*
 * Function: addNewLoop
 *
 * This function adds the generated communication loops to the
 * code. If the Lc loop is independent, the upper and lower
 * bounds are set equal to each other so that the compiler will
 * degenerate the loop in a subsequent pass.
 *
 */
block addNewLoop(named_symcoeff_ineq *nsli, const block & body,
     var_sym * idx, int i, boolean ins_before, tree_node_list
     *tnl, tree_node *tn, int dont_tighten, int step, boolean
     degenerate)//parameter added//
{
    block code;
    instruction *lb_ins;
    instruction *ub_ins;
    tree_node_list *new_tnl;
    block lvar_b (idx);
    named_symcoeff_ineq loop_bounds;


    if (! (~(*nsli)) ) {


        // Use Fourier-Motzkin elimination to tighten the bounds
        // named_sc_fm FM(*nsli);

        if (dont_tighten) {
            loop_bounds = *nsli;
        } else {

            FM.fm_bounds(i, i+1);
            FM.get(&loop_bounds);
        }


        boolean single_iter = create_instructions (loop_bounds,
                                  i, idx, &lb_ins, &ub_ins);

        block lb_blk (lb_ins);
        block ub_blk (ub_ins);
```

```
        if (step < 0) {
//NEW CODE//
            if(degenerate){ //If Lc was independent

              //Then set the lower bound equal to upper bound.
              //The fourth parameter was lb_blk (lower bound)
              //It has been set to ub_blk (upper bound).
              code.set (block::FOR(lvar_b, ub_blk, bop_geq,
                           ub_blk, block(step), body) );
            }
            else{
//END NEW CODE//
              code.set (block::FOR(lvar_b, ub_blk, bop_geq,
                           lb_blk, block(step), body) );
            }

        } else {
//NEW CODE//
            if(degenerate){ If Lc was independent

              //Then set the lower bound equal to upper bound.
              //The third parameter was ub_blk (upper bound)
              //It has been set to lb_blk (lower bound).

              code.set (block::FOR(lvar_b, lb_blk, lb_blk, body)
);
            }
            else{
              //code.set(stmts);
              code.set (block::FOR(lvar_b, lb_blk, ub_blk, body)
);
            }
// END NEW CODE //

            new_tnl = code.make_tree_node_list();
        }


        if (tnl != NULL) {
            new_tnl = code.make_tree_node_list();
            if (tn != NULL) {
                if (ins_before) {
                    tnl->insert_before (new_tnl, tn->list_e());
                } else {
                    tnl->insert_after (new_tnl, tn->list_e());
                }
            } else {
                tnl->append (new_tnl);
            }
        }
    } else {
```

```
        // printf ("No Solutions\n");
    }
    return code ;
}

//END ADD NEW LOOP//
```