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

P. Jerry Martin
Department Computer Science
University of North Carolina Wilmington
Wilmington, NC USA

Clayton S. Ferner
Department Computer Science
Company of North Carolina Wilmington
Wilmington, NC USA

Abstract - *In this paper we experiment with two optimization techniques we are considering implementing in a parallelizing compiler that generates parallel code for a distributed-memory system. We have found that there are two problems that often arise from the automatically generated message-passing code: 1) messages contain redundant data, and 2) the same data is sometimes transmitted to different processors, yet the messages are repacked for each processor.*

Our experiments demonstrate that it is indeed worthwhile suppressing the packing of redundant information in a message. Not only did it improve performance, but it allowed us to run the program on a larger input size. We also discovered that it is not worthwhile to suppress the repacking of the same message. The reason is because the size of the messages is a greater factor in the performance of a message-passing program than the number of instructions executed.

Keywords: Parallelizing compiler, message-passing, code generation.

1 Introduction

The concept of automatically parallelizing sequentially written programs to run on parallel machines has existed for several decades. The advancement in this area has been significant but not outstanding. This is mainly because there are several NP-hard problems that must be addressed when deciding how best to parallelize a sequential program. As a result, many researchers would rather hand-code their parallel solution than to rely on a parallelizing compiler. Nonetheless, we continue to perform research on parallelizing compilers in hope to ease the burden on the programmer of finding a parallel solution.

The progress made by researchers in creating parallelizing compilers for shared-memory parallel systems has been greater than the progress made by researchers of compilers for distributed-memory parallel systems. The reason is because the parallel code must make use of message-passing in order to deal with the distributed nature of the data. This adds another level of complexity to the already complex task of parallelizing sequential programs. Furthermore, most parallel programs written to run in a distributed-memory system use a language or tools that make them asynchronous. An asynchronous environment usually is a more difficult environment in which to write a correct parallel program than a synchronous environment.

However, distributed-memory systems are increasing in popularity and frequency. Shared-memory systems are traditionally large and expensive systems. On the other hand, one can put together a distributed-memory system of relatively inexpensive computers and a network switch. Furthermore, desktop computers that are manufactured today have multiple processors. For these reasons smaller schools and organizations are opting to use these systems because of their affordability and the ease with which multi-processor, distributed-memory systems can be put together. Also, the advancement of grid technology is making it possible to have a very large number of processors at one's disposal. The need is growing for tools that can assist computer programmers in developing parallel applications that can run on distributed-memory systems.

In this paper, we are considering a technique to reduce the size of the messages that are transmitted between processors. In developing our compiler and testing with some sample programs, we discovered that the techniques for passing messages can lead to redundant information being packed in a single message. Furthermore, the same message can be sent to multiple processors, yet the message will be reconstructed each time. We are developing techniques to check for and suppress the packing of redundant information in messages as well as suppress the recreation of the same message. If we are successful, then we will not only reduce the size of the messages, we will also reduce the time to create the messages. Since these techniques will involve removing loops within a loop nest, the time savings could be significant.

The purpose of the research presented in this paper is to prototype these ideas to determine if they are feasible. We present here an example program where the messages have redundant information. We generate the loop nests the way the compiler does and then modify the loops to suppress the redundant information and suppress the repacking of messages. We then compare the sizes of the messages as well as the time to execute the programs. If we are successful in decreasing the time for the program to execute, then we will implement these techniques in the compiler.

We only consider one application in this paper: the elimination step of Gaussian Elimination. Again, this paper is presenting the preliminary results of the techniques mentioned above. These preliminary results will justify continuing the effort to implement these techniques or to abandon them. Although we have seen the problem of redundant data in other applications, we do not know if this is a common problem in real scientific code. We expect this problem to be common, since the problem of redundant data is caused by multiple partitions being mapped to the same physical processor and not a characteristic of the application itself. We will elaborate further on the causes later in this paper.

2 Background

We have developed an automatically parallelizing compiler that produces message passing code using MPI suitable for execution on a distributed-memory system. The compiler is called the Paraguin Compiler [5] and is built using the SUIF Compiler [7]. The SUIF Compiler provides all the tools necessary to build a parallelizing compiler.

The technology behind the generation of parallel code and the code to transmit messages containing the dependent data is beyond the scope of this paper. We refer the interested reader to [1], [2], [4], [5], [7], and [8] for background on the details of how this is done. In this paper we present the code that the compiler produces without discussing the details of how it was produced or arguing the correctness of that code.

Figure 1 shows the elimination step of Gaussian Elimination, which we will use as our running example

```
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];   (S1)
      }
   }
}
```

Figure 1. The Elimination Step of Gaussian Elimination.

and prototype for the techniques presented in this paper. Given that the problem is partitioned such that each iteration of the `i2` loop is a separate partition, there is a data dependence between the left-hand side of the assignment of statement S1 (`a[i2][i3]`) and the second array reference (`a[i1][i3]`) on the right-hand side that crosses partitions. When these partitions are mapped to different physical processors, communication of this value is required.

Figures 2 and 3 show the receive and send loops that unpack and pack the dependent data between the reference `a[i2][i3]` on the left-hand side of the assignment and the reference `a[i1][i3]` on the right-hand side of the assignment. The send loop is executed by any processor where `pidw` is the current processor's id (`mypid`). Similarly, the receive loop is executed by any processor where `pidr` is the current processor's id (`mypid`). The parallelized execution loop is inserted between the two communication loops. The code shown here does indeed work correctly, although the performance is poor. The poor performance is due to the fact that communication and computation are performed is completely separate steps. This problem is addressed by overlapping communication with computation using a techniques described in [5]. However, this technique is out of the scope of this paper and a step that is performed *after* the creation of the communication loops. The techniques we are proposing in this paper would be performed after the creation of the communication loops but prior to the overlapping step.

Within the loop nests that send and receive messages are various loop variables. These loop variables have meaning. Take the send loop for example. Each processor executes that loop nest where the send processor id (`pidw`) is itself. It then loops through all physical processors (`pidr`) that require data computed by `pidw`. Next, the send processor loops through all partitions it owns (`pw`) and the partitions that the receiving processors owns (`pr`) for which there is

```
//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 2. 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 3. The Send Loop for the Elimination Step of Gaussian Elimination.

data produced by pw needed by pr. Then the sending processor loops through all iterations (i1w, i2w, i3w) executed by pw and all iterations (i1r, i2r, i3r) executed by pr for which iteration i1w, i2w, i3w produces data needed during iteration i1r, i2r, i3r. The receive loop can be viewed in a similar way except that it is executed by each processor where pidr is itself.

3 Statement of the Problem

The code generated by the compiler produces messages containing redundant data. This needlessly increases the size of messages sent between pairs of processors as well as the time to pack those messages. Furthermore, if multiple processors are receiving a message from the same sending processor, the sending processor may repack the same information in each message for each receiving processor, instead of packing the message once and then sending multiple times. This further increases the time spent packing messages.

Take for example the code to send and receive data from the elimination step of Gaussian Elimination shown in figures 2 and 3. A sample of the debugging statements from this sending loop of figure 3 is shown in figure 4. One can see from figure 4 that the same array location is packed multiple times in a single message. This happens when the number of partitions is larger than the number of processors, which is usually the case. Each instance of an array location in the message is destined for a separate partition. However, since each processor may be responsible for executing multiple partitions, multiple instances of the same array element will appear in the message. One can also see this in the code from the send loop in figure 3. The data that are packed are the values `a[i2w][i3w]`. Yet there is no dependence of the array element subscripts `i2w` and `i3w` and the loop variable `pr` (which corresponds to all the partitions that require the data being packed). Each iteration of the `pr` loop packs the same information. Similarly, the receive loop nest unpacks the same information for each iteration for the `pr` loop. Furthermore, the array element subscripts are not dependent on the receiving processor id (`pidr`). So each iteration of the `pidr` loop repacks the same message with the same data.

```
...
<pid0, p2>: pack a[2][2] - Value: 63.000000
<pid0, p2>: pack a[2][3] - Value: 28.000000
<pid0, p2>: pack a[2][4] - Value: 91.000000
<pid0, p2>: pack a[2][5] - Value: 60.000000
<pid0, p2>: pack a[2][6] - Value: 64.000000
...
<pid0, p2>: pack a[2][2] - Value: 63.000000
<pid0, p2>: pack a[2][3] - Value: 28.000000
<pid0, p2>: pack a[2][4] - Value: 91.000000
<pid0, p2>: pack a[2][5] - Value: 60.000000
<pid0, p2>: pack a[2][6] - Value: 64.000000
...
<pid0>: send to <pid1>
```

Figure 4. Sample from Debug Statements from Send Loop of Figure 2.

4 Ideas to Speedup Generated Code

We believe we can generate smaller messages, and reduce packing time by using the following techniques:

1. The size of the messages can be reduced by replacing FOR loops in the message passing code with assignment statements (causing them to be degenerate loops) when the loop variable is independent of array element subscript variables.

2. The time spent packing can be reduced by placing an IF statement with the condition that the `pidr` is equal to the lower bound within the send loop code if the receiving processor `pidr` is independent of the array element subscript variables.

Both of these techniques require a check of independence between the array element subscripts and the loop variable. The SUIF compiler has the tools necessary to check for dependence. We expect this to be fairly easy to implement. This could be accomplished by looping through each loop variable in the communication loops (except for `pidr`) and performing a dependence test on that loop variable and the data being packed. If the loop variable of both sides of the communication is independent of the data being packed, then the loop can be suppressed using technique (1) above. The same test can be performed on the `pidr` loop variable to determine if technique (2) applies. The tests we present in this paper are intended to investigate the effects of these techniques on the performance and whether the implication justifies continued work on these ideas.

5 Tests

To test the feasibility of our ideas, we created 5 programs that perform the elimination step of Gaussian Elimination:

- Program #1 - The Sequential Program
- Program #2 - The original MPI Program automatically generated
- Program #3 - The MPI Program with redundant data suppression
- Program #4 - The MPI Program with redundant repacking suppression
- Program #5 - The MPI Program with redundant data & repacking suppression

The message passing and main execution loops used in the prototypes were generated using Linear Inequality Calculator (LIC) included with the SUIF compiler [7]. LIC aids in rapidly prototyping systems for code generation. The loop bounds, partitioning, partition-to-processor mapping, and last-write tree for the Gaussian Elimination code of figure 1 were provided to LIC, which then generated the loops. The original communication loops for program #2 are shown in figures 2 and 3.

Program #1 is the sequential version of the program shown in figure 1. This is used as a control. Programs 3, 4, and 5 are modifications of Program #2. In Program #3, we suppress the packing of redundant data in each message by replacing the `pr` FOR loops with an assignment of `pr` to its lower bound value in both the send and receive loop nests. Specifically, the following statement was inserted to replace the `pr` FOR loop in the loops shown in figures 2 and 3:

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

This is a legal modification to the loops because the body of those loops (the packing of `a[i2w][i3w]` and the unpacking of `a[i1r][i3r])` is independent of the variable `pr`. So only one iteration of the loop is necessary.

In Program #4, we attempted to suppress the repacking of the same message for each processor to which the message is sent. We accomplish this by inserting the following IF statement inside the `pidr` loop and surrounding the packing code in the send loop nest of figure 3:

```
if(pidr == 1+pidw){
```

This has the effect of executing the packing of the message only for the first iteration of the `pidr` loop. Subsequent iterations of the loop do not repack the message, but simply send the existing message. This should be done on the send side only, because the receive loop still needs to unpack the data in the message.

In Program #5, both redundant data and redundant repacking were suppressed using the techniques of Program #3 and Program #4 together.

6 Results

To test the programs, each parallel program was run on a cluster consisting of 6 Dell PowerEdge 1850s with 2 Intel Dual Core 2.8 GHz processors with 12 Gbytes of memory and 9 Sunfire X4100s with 2 AMD Dual Core 2.6 Ghz processors with 8 Gbytes of memory. The machines are connected using a Cisco 100 Mbps switch. The sequential program was run on one Dell PowerEdge, while the parallel programs were run on 4, 8, 12, 16, and 20 processors. The problem input size ranged from 100 to 1000 in increments of 10. Each program was run 10 times at each processor/input size combination, and the runtime results presented in this section are the averages of the 10 runs. The total sums of message sizes for each program were also collected.

The performance of Program #2, the original parallel program, is shown in figure 5. The runtime of this program was significantly worse than the sequential program due to the time spent packing and sending the large messages. Furthermore, we were unsuccessful in running the program for some combinations of the number of processors and the problem size because the resulting messages were so large that they exceeded the memory capacity of the machines.

The performance of Program #3 is shown in figure 6. The technique of suppressing the packing of redundant data allowed the parallel program to run at a similar speed as the sequential program. The performance of the parallel program running on 4 processors was actually better than sequential, but this is due to the fact that those 4 processors reside on the same machine and use shared memory to implement the message passing. Although it is not easy to detect using the same scale for figure 6 as the other graphs, the small gap between the sequential runtime and the runtime of the parallel program is narrowing. If the problem size had been large enough the parallel program should have run faster than sequential.

Furthermore, we have not implemented overlapping communication with computation as discussed in [5]. The code, as described here, will essentially be executed sequentially because each processor will receive all of its required data before proceeding with execution. Likewise, each processor will not transmit its computed values to processors that need them until it has finished all of its execution. Therefore, we cannot expect to do much better than sequential execution, until we can implement the technique of suppressing redundant data in the compiler, which does overlap communication with computation.

The performance of Program #4, which suppresses the repacking of messages, is shown in figure 7. This program performed only slightly better than the original parallel program. The slight performance improvement is the result of the time savings for not having to repack the message. However, one can see that performance is influenced primarily by the time required to transmit the

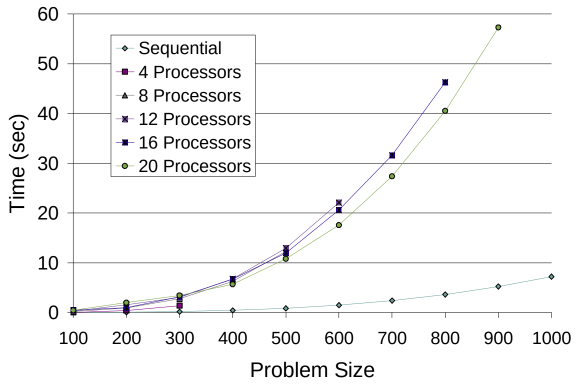

Figure 5. Runtime for Original Loops

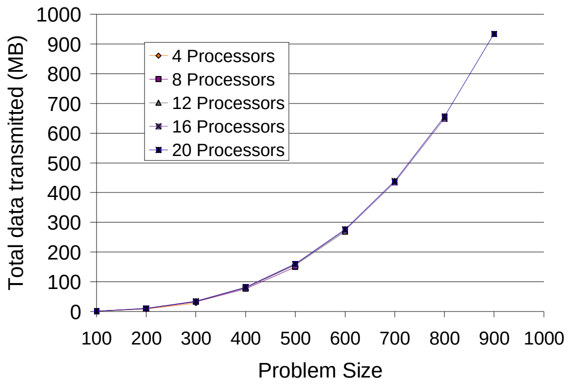

Figure 8. Total Bytes Transmitted for Original Loops

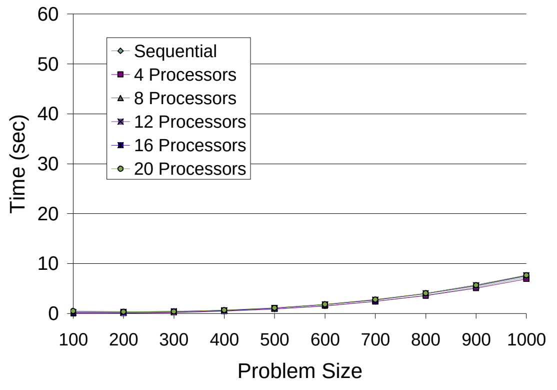

Figure 6. Runtime when Packing of Redundant Data is Suppressed.

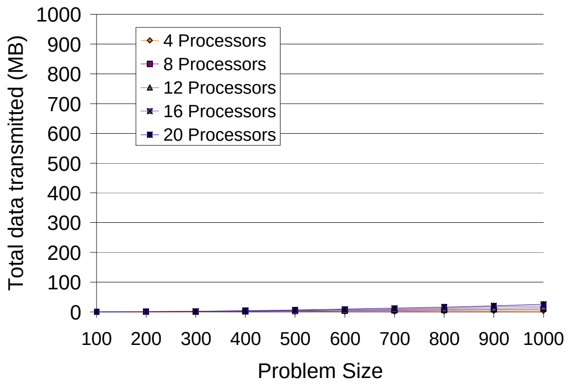

Figure 9. Total Bytes Transmitted After Suppressing Redundant Data

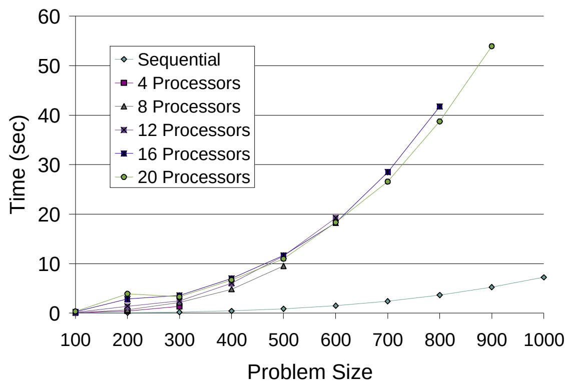

Figure 7. Runtime when Repacking of Message is Suppressed.

messages, not by the time spent executing instructions that copy values from one memory location to another, which packing does. Furthermore, the suppression is only relevant on one side of the communication (the send side).

Program #5 combined the data and repack suppression techniques. The timing results are not shown because they are almost identical to the results of Program #3. It appears that combining the two techniques does not provide a significant advantage over implementing data suppression alone.

Figures 8 and 9 show the total sizes of all messages sent between processors for the input size and number of processor combinations. Program #3 greatly reduces the size of the messages, which is the main reason for the performance improvement. It also allows the problem size to increase, since the sizes of the messages are not pushing the memory limits of the machines.

Program #4 actually increased the total bytes transmitted. This is because the last processor may have a smaller set of partitions to execute. So the messages sent to the last processor are likely be smaller. When we suppress repacking of the messages, the messages sent to the last processor may have too much data. This does not present a problem, but since the performance was only marginally improved, it is not worth it. In a distributed-memory environment, anything we can do to reduce the message size is more important than anything we can do to reduce the number of instructions executed.

Program #5 did not increase the total bytes transmitted from Program #3, but it did not decrease it either. Again, it is not worth the effort to suppress repacking when the dominant factor for performance in a distributed-memory environment is the number of bytes transmitted.

7 Conclusions

In this paper we tried two techniques to improve the performance of parallel code generated by an automatic parallelizing compiler to run on a distributed-memory system using message passing. The purpose of this experiment is to determine if there is reason to pursue these techniques and make the effort to implement them in a real parallelizing compiler. The two techniques that we presented in this paper are: 1) suppress the redundant data within messages sent between processors, and 2) suppress the repacking of the same message for different processors.

What we discovered during this experiment is that it is more important to reduce the message sizes than to reduce the amount of code executed. In a distributed-memory environment where messages are transmitted to pass dependent data between processors, the size of the message is a dominant factor on the runtime of the program. Reducing the redundant information in messages has a large impact on the performance. Furthermore, reducing the message sizes allows one to run the program on larger input sizes, which is another very important consequence of this technique.

The second technique of suppressing the repacking of the same message that is sent to multiple processors had only a marginal improvement in performance. The increased performance was due to the fact that fewer instructions are executed. However, reducing the number of instructions executed has only a small impact on performance as compared to the impact of reducing message sizes. In fact, suppressing the repacking of messages can lead to larger messages being sent to some processors. It is very important to make every effort to keep the number of bytes transmitted as small as possible, even at the expense of executing more instructions.

We anticipate the implementation of the technique to suppress the packing of redundant data in messages to be fairly straightforward. Fortunately, the SUIF compiler has techniques available to test for the dependence between two variables. This test is essential for determining if a loop can be replaced by a degenerate loop.

The results of this experiment will allow us to move forward with the implementation of the first technique. We expect to demonstrate that this is a worthwhile optimization for a parallelizing compiler to apply. Also as a result of this experiment, we will not spend time implementing the second technique, since it is unlikely to result in any significant improvement.

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

[3] Michael Classen, "Automatic code generation for distributed memory architectures in the polytope model", Ph.D. Thesis, Universitat Passau. September 30, 2005.

[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] G. Goumas, Drosinos, Athanasaki, Koziris. "Message-passing code generation for non-rectangular tiling transformations", *Parallel Computing*, Vol. 32, 711–732, 2006.

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

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