Independent Loop Suppression to Reduce Message Size for Message-Passing Code

P. Jerry Martin[1] **and Clayton S. Ferner**[1]
[1]Department Computer Science, University of North Carolina Wilmington, Wilmington, NC, USA

Abstract - *In this paper we implement and experiment with an optimization technique 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.*

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

1 Introduction

The concept of automatically parallelizing programs written for sequential execution so that they can be 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 rely on a parallelizing compiler. Nonetheless, we continue to perform research on parallelizing compilers in hope of easing the burden on the programmer of finding a parallel solution. A recent article in CACM on compiler research stated that, “Exploiting large-scale parallel hardware will be essential for improving an application’s performance or its capabilities in terms of execution speed and power consumption. The challenge for compiler research is how to enable the exploitation of the power of the target machine, including its parallelism, without undue programmer effort.” [6, p. 62]

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 is usually 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 or cores. 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 implement a technique to reduce the sizes 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 developed techniques to check for and suppress the packing of redundant information in messages as well as suppress the recreation of the same message. These optimization techniques not only reduce the size of the messages, but 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 this paper is to present one of the optimization techniques described above, discuss its implementation, and show the result of using this technique on an example application. In section 2, we discuss why we implemented only one technique. We also only consider one application in this paper: the elimination step of Gaussian Elimination shown in figure 1. 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.

This paper is organized as follows: section 2 gives some background on the message-passing code, section 3 describes the problem and the fix, section 4 describes the implementation, section 5 describes the tests that we ran, section 6 discusses the results, section 7 discusses future work, and section 8 concludes.

2 Background

We have 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 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], [3], [4], [5], [9], and [10] 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 application program 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.

```
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

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

```
//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",
                                         pidw, 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.

due to the fact that communication and computation are performed by completely separate steps. This problem is addressed by overlapping communication with computation using a technique 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. Again, we refer the reader to [1], [2], [3], [4], [5], [9], and [10] for background on message-passing code generation.

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 that the receiving processor owns (`pr`) and the iterations mapped to `pr` (`i1r`, `i2r`, `i3r`) as well as the partitions it owns (`pw`) and the iterations mapped to `pw` (`i1w`, `i2w`, `i3w`) for which there is data produced by iteration `i1w`, `i2w`, `i3w` needed by 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 sizes of messages sent between pairs of processors as well as the time to pack, transmit and unpack those 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 the 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 may appear in the message. One can also see this problem by examining 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

```
...
<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 3.

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.

The two optimization techniques that we studied for their effectiveness in improving the performance of the message-passing program are:

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` variable is equal to the lower bound within the send loop code if the receiving processors `pidr` is independent of the array element subscript variables.

The first step of this study was to modify by hand the resulting parallelized code to implement these optimizations. We reported the results of hand-coding the optimizations in a previous paper [8]. We discovered and reported in [8] that optimization (1) produced a significant improvement in reducing the sizes of the messages as well as the performance of the resulting program.

We also discovered that the second technique improved the performance only slightly and made no difference in the message sizes. Although we knew the second technique would have no impact on the sizes of the messages, we were somewhat surprised to see only a small improvement in performance. Therefore, we have chosen to implement only the first optimization technique. Although the second technique could be implemented and possibly provide some benefit to other application programs, we are concentrating our effort on the technique which we know will produce a beneficial result. Furthermore, the performance of a message-passing program is primarily dictated by the number of bytes transmitted and not by the number of instructions executed.

4 Implementation of Redundancy Elimination Technique

In order to suppress redundant data in the messages, we need to suppress the loops in the send and receive loop nest for which the data being packed are independent of that particular loop variable. Each communication loop variable L_c must be checked to determine if any of the array element subscript variables are dependent upon it for their values. If the variables are not dependent on L_c, then the L_c loop is replaced with an assignment statement.

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

We start with the system of inequalities S that represents all of the loop bounds for the receive loop nest, and separately, the send loop nest. (See [1], [2], [3], [4], and [10] for background on the system of inequalities and how these are used to represent loop nests.) We then create a new system of inequalities, S_x, for each array subscript variable X. This is accomplished by filtering through the original system of inequalities for those inequalities that contain a nonzero coefficient for X. In other words, S_x will have only the inequalities of S that involve the variable X. For each new system, the inner loop variables are projected away, since outer loop variables cannot be dependent on inner loop variables. We then perform a dependence check (see below) between each loop variable L_c on each new system S_x.

4.2 Step 2: Dependence Check

We iterate through each communication loop variable L_c starting with the innermost and moving outward until we reach a loop with either the `pidw` or `pidr` variable. A dependence check test is run between L_c and each S_x that was created in the previous step. We test dependence by filtering S_x on L_c. If the two variables are independent, then the resulting system will be empty. In other words, no inequalities of S_x involve the variable L_c. If there is at least one inequality in the resulting system, then X is dependent on L_c. The filter is a vector that contains zero values for all loop variables except for a value of 1 for L_c. Figure 5 displays this concept in pseudo code.

```
For each communication loop Lc
    boolean dependent = FALSE
    For each system of inequalities Sx
        System S'x = Filter_thru(Sx using Vector(Lc))
        If S'x is not empty then
                dependent = TRUE;
                Set upperbound(Lc) = lowerbound(Lc)
                break;
```

Figure 5. Pseudo Code for Dependency Check.

4.3 Step 3: Replace loop

If the current loop variable L_c is determined to be independent of all array element subscript variables, then that loop is forced to be a degenerate loop by making its upper bound the same as its lower bound; therefore, the loop only has a single iteration. In a latter pass of the compiler, degenerate loops are replaced with single assignment statements. For example, the loop:

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

would be changed to

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

which would then be replaced with

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

5 Tests

Three programs that implement the code of figure 1 were produced for comparison:

- Program #1 – Sequential program to be executed on a single processor (produced using the gcc compiler).
- Program #2 – The original MPI program produced by the compiler without the optimization technique described in section 4.
- Program #3 – The redundant data suppression MPI program produced by the compiler with the optimization technique described in section 4.

Comparing the performance of program #3 against #2 will show us the impact on performance the optimization technique described in the previous section has. We also include the performance of program #1 as a control.

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. The problem input size ranged from 100 to 1000 in increments of 100. Each program was run 10 times at each processor/input size combination, 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 program were also collected.

6 Results

Figure 6 shows the execution time of program #2 for various problem sizes on various numbers of processors. These times include the time to scatter the input and gather the results back to the master processor, but they do not include the time for I/O. Also shown in figure 6 is the runtime for the sequential program run on a single processor. The runtime shown for the sequential program is the time to execute the program segment of figure 1, but also does not include the time for I/O. One can see that the parallel program experienced significant slowdown. This is due to the large messages that contain redundant data and the time it takes to build and transmit those messages. Furthermore, we are trying to execute a nested FOR loop in a distributed-memory system. It is extremely difficult to parallelize a loop nest at such a fine level of granularity and obtain speedup. One can also observe in figure 6 that there are some data points missing. There were some problem size and number of processor configurations that resulted in messages so large that the program exceeded the memory capacity of the machines.

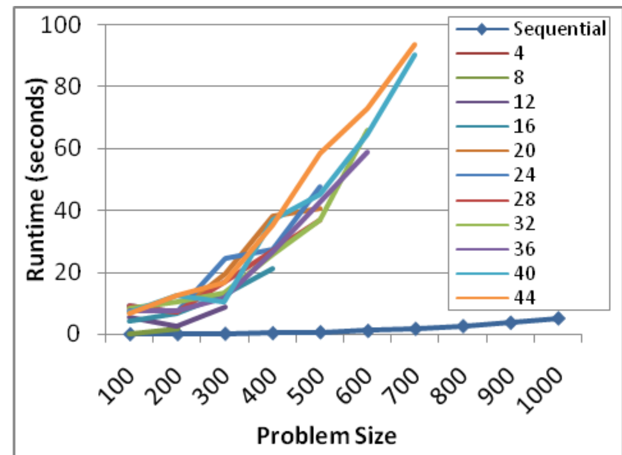

Figure 6. Runtime without Optimization.

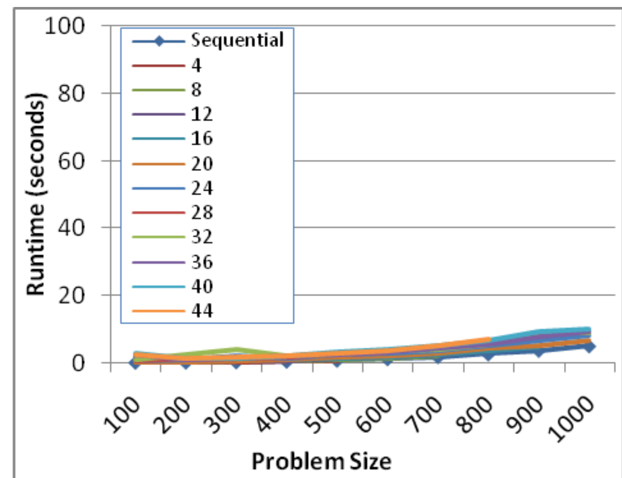

Figure 7. Runtime with Optimization.

Figure 7 shows the runtimes of program #3 with the same configurations as program #2. All times are less than 10 seconds. The improvement in performance using the optimization technique proposed here is on average 88% reduction in runtime. The performance is comparable to sequential execution. The parallel programs cannot do any better than sequential, and the reason for this is explained in section 7.

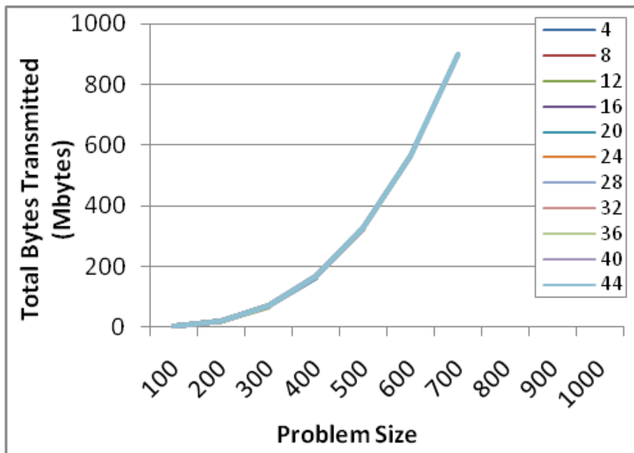

Figure 8. Total Bytes Transmitted without Optimization.

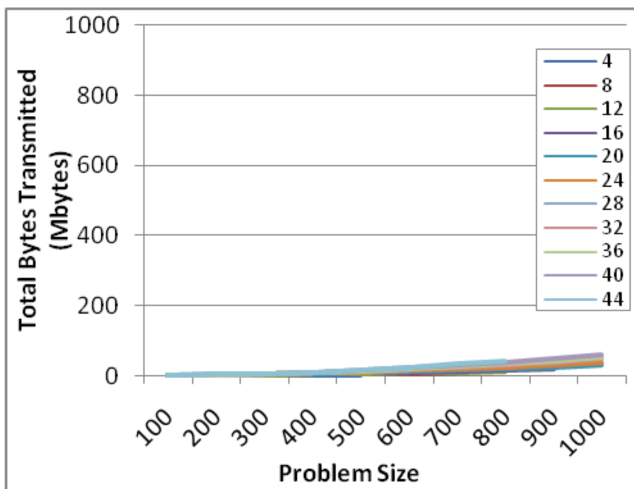

Figure 9. Total Bytes Transmitted with Optimization.

Figures 8 and 9 show the total number of bytes 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 do not exceed the memory limits of the machines. The reduction in number of bytes transmitted is about 93% on average.

7 Future Work

The reason why the parallel program cannot run faster than sequential is because we are not able to overlap communication with computation. The compiler has a pass whereby it attempts to merge the loops to allow transmission of partial results and allow the program to continue computing without waiting for more data. The way the communication loops are generated, without overlapping, each processor waits to receive all of the data it needs before beginning work, after which it transmits its data to processors that need it. Even if communication were instantaneous, the parallel program could not outperform the sequential program without this merge step. When we hand-coded our prototype optimization, which we published in [8], we knew the prototype would not outperform the sequential program. However, we expected to see speedup when this optimization was implemented in the compiler.

The problem is that the system of inequalities, which are used to generate the loops and which we also use for the optimization, are discarded after the loops are created. The pass that attempts to merge the loops tries to recreate the system of inequalities from the new loops. However, the loops have become so complex that the routines within the SUIF library that convert loops to a system of inequalities are unable to do so. Therefore, the pass that attempts to merge loops is unable to overlap communication with computation. After we fix this, we are confident that we will see real speedup for a loop nest partitioned in a very fine level of granularity on a distributed-memory parallel system.

8 Conclusions

In this paper we implemented a technique to improve the performance of parallel code generated by an automatic parallelizing compiler to run on a distributed-memory system using message passing. The optimization technique suppresses the redundant data within messages sent between processors. We showed how and why redundant information is packed in the messages. We discussed how the compiler can test for the existence of redundant information and suppress it.

The Elimination Step of Gaussian Elimination was used as the application program for testing the effects of implementing this optimization. Without the proposed optimization, the parallel code generated by the compiler produced redundant data within messages sent between processors. We created three test programs for comparison: a sequential program, a parallel program produced by the compiler without using the optimization, and a parallel program produced by the compiler using the optimization. We have shown that the performance of the program by using the new optimization improved by 88%. We also have shown that the optimization decreases the size of messages passed between processors by 93%. Furthermore, reducing the message sizes allows one to run the program on larger input sizes, which is another very important consequence of this technique.

9 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.