The Paraguin Compiler—Message-passing Code Generation Using SUIF

Clayton S. Ferner; University of North Carolina at Wilmington; Wilmington, NC

Keywords: Parallelizing compilers, Parallel/Distributed systems, Open-source, Message-passing

ABSTRACT

In this paper, we introduce the *Paraguin* project at the University of North Carolina at Wilmington. The goal of the Paraguin project is to build an open source message-passing parallelizing compiler for distributed-memory computer systems. We discuss the progress we have made in developing this compiler as well as mention the parts that have not yet been developed. It is our intent that, by providing an open source compiler, we will stimulate research in automatic message-passing parallelism and encourage collaboration.

We demonstrate a technique to improve the performance of a message-passing program by overlapping communication with computation. Although the original concept was introduced in previous work [1], the algorithm was not developed nor shown to provide benefit. Our preliminary results indicate that the technique does significantly improve the performance. We were able to reduce the running time of our test program by 4 to 65 percent.

1. INTRODUCTION

Beowulf-class clusters of computers are becoming more and more popular every year. These clusters are inexpensive computer systems capable of parallel computation. The philosophy behind Beowulf clusters is that they can be built from low cost, off-the-shelf components. This allows small companies or departments to build their own high performance parallel computer system. Unfortunately, there are not many options available to a user of a Beowulf system for automatically generating parallel code.

The scientific community spends much time learning how to use parallel systems and how to program in parallel. This is time that is *not* spent on the particular problems the user needs to solve. Parallelizing compilers represent one option for the scientific programmer who does not wish to also become a computer scientist. However, most parallelizing compilers today will generate parallel code that assumes a shared-memory model of the underlying system. There are only a few that can generate message-passing code that can be run on a Beowulf without the aid of a Distributed-Shared Memory (DSM) library. A few compilers that we found that claim to do this are: Paradigm, VAST-HPF, Bert 77, and PGI CDR-16. Not only is the list short, but also the purchase of one of these compilers will represent a large percentage of the cost of the entire system. This does not seem to fit well with the philosophy of a Beowulf cluster.

Fortunately for users of distributed systems, a DSM library, such as Treadmarks [2] or SAM [3], will allow parallel code written for a shared-memory model to be run on a distributed-memory system. The DSM packages have alleviated the message-passing and scheduling concerns for both the user and the compiler. There are also compilers that provide a global addressing model to the user, such as Olden [4] or Split-C [5]. However, the abstraction created by the DSM library is a source of inefficiency [6,7]. The paper by Cox, et. al. [6] showed several examples of regular programs where the compiler-generating message passing solution outperformed the DSM solution. Although DSMs create a convenient abstraction, they are not necessary the best option.

The research for shared-memory systems is more advanced than that for distributed-memory. There are several reasons for this. First, parallel computation for distributed-memory systems appears to be more complex than for shared-memory. Second, DSMs have given users another option other than using a parallelizing compiler to generate message-passing code. Third, the lack of open-source parallelizing compilers has hindered research in automatic parallelization for distributed-memory. If someone wishes to conduct research in this area, there are few open source compilers from which they can start. As a result, much of the work done in this area, such as static scheduling heuristics, has used simulation or a small number of inputs. The Suif compiler [8] from Stanford University is an example of the benefit of having an open-source compiler. Much research has been conducted using Suif as the base compiler, not only at Stanford but also around the world. This has been very beneficial for advancing compiler technology. Smaller research institutions, that do not have the resources of a school like Stanford, can now conduct research in compiler technology without making the enormous investment of building the infrastructure.

This paper presents a progress report from a project called *Paraguin* at the University of North Carolina at Wilmington. We are building an open-source message-passing automatic parallelizing compiler based on the Suif compiler. The goal of this paper is three-fold: we would like to raise awareness of the Paraguin project and hopefully stimulate interest as well as collaboration, discuss where we are in the project and where we need to go, and present some preliminary results on work to overlap communication with computation.

The rest of this paper is organized as follows: Section 2 gives an overview of the project and discusses the background of message-passing code generation, Section 3 describes the

0-7803-7252-2/02/$10.00 © 2002 IEEE Proceedings IEEE SoutheastCon 2002

work we have done to overlap communication with computation, Section 4 presents some results, and Section 5 provides some concluding remarks.

1.1. Notation

$\vec{i} = [i_1, \ldots, i_n]^T$	vector of loop indices
p	virtual processor or partition
p_r, p_w	virtual processor for receiving and sending
lb_i, ub_i	lower and upper bounds for loop i
$f(c, i_1, \ldots, i_n)$	affine function of loop indices and symbolic constants

2. BACKGROUND

2.1. Overview of the Paraguin Project

The Paraguin Project is a compiler pass written using Suif version 1.3. It will generate a program in C with calls to the MPI library [9] to send and receive data between processors. The work that we have done is largely based on work presented by Amarasinghe and Lam [1]. The reason we must reproduce much of that work is because their compiler pass is no longer available. However, we do present new work in this paper that was not given by Amarasinghe and Lam. The work they presented shows how to build a system of linear inequalities from a loop nest, given the *computation decomposition* (mapping of iterations to processors) and the exact data-flow analysis. This system is then used to generate a loop nest for each read array access to receive and unpack a message from the processor that generated that value. It is also used to generate a loop nest for each write array access to pack and send a message to the processors that need it. Amarasinghe and Lam assume that the computation decomposition and the exact data-flow analysis are given.

We have also implemented in our compiler pass the construction of this system of linear inequalities and the generation of the Receive and Send loop nests. However, we have not yet implemented the data decomposition and the exact data-flow analysis. For the results presented in this paper, we have provided this information to the compiler by means of pragma statements in the source code. The dataflow analysis is a well-studied problem, and we plan to implement this so that the compiler can automatically perform as much of this analysis as possible. However, in the case of irregular loop nests, we will require the user to provide the dataflow analysis to assist the compiler in generating efficient code.

We expect the data decomposition should be provided in a later version of Suif. Work done by Lim and Lam [10] shows how to take a loop nest as input and determine the computation decomposition that provides the maximum degree of parallelism while minimizing the degree of synchronization. They also show that this decomposition provides the coarsest granularity. Lim and Lam make the claim that their algorithm "[...] subsumes previously proposed loop transformation algorithms that are based on unimodular transformations, loop distribution, fusion, scaling, reindexing and statement reordering." [10, p. 445]. The promise of this research is great, and we hope that, when this is available, we can then generate message-passing code that is much more efficient than was previously possible with a parallelizing compiler.

2.2. Message-Passing Code Generation

The algorithm to generate message-passing code is described in detail in [1]. We will only mention briefly how the algorithm works. Suppose we have a loop nest such as that shown in Figure 1, which contains read and write accesses to some array. The domain of our input programs is limited to loops whose bounds and array accesses are affine functions of the symbolic constants and the loop indices of outer loops, as is done in the research upon which our work is based. The iteration space of the loop nest is spanned by the vector $\vec{i} = [i_1, \ldots, i_n]^T$. The iterations will be mapped to virtual processors (or partitions), designated by p. This mapping is called the *computation decomposition*. The virtual processors will later be mapped to physical processors. Our compiler can support either cyclic or block mapping of virtual to physical processors.

```
for i1 = l1(c) to u1(c)
 for i2 = l2(c,i1) to u2(c,i1)
  ...
    for in = ln(c,i1,...,in-1) to un(c,i1,...,in-1)
      ...
      ... = ... X[ f(c,i1,...,in) ] ...
      X[ g(c,i1,...,in) ] = ...
      ...
```

Figure 1: Example Loop Nest

We use $\vec{i}_r$ to denote the iteration instance of a statement that contains a read access for a particular array value and $\vec{i}_w$ to denote the iteration instance of a statement that contains a write access for a particular array value. We also use p_r and p_w to denote the virtual processor to which $\vec{i}_r$ and $\vec{i}_w$ are mapped. A read access for iteration $\vec{i}_r$ can be mapped to the exact iteration $\vec{i}_w$ where the write access generated the value used. This mapping is provided by a structure known as the *Last Write Tree (LWT)* [11].

The original loop nest is transformed into a SIMD or MIMD code by creating a system of linear inequalities that represent the loop bounds for $\vec{i}$ and the computation decomposition. The new loop nest is generated using the algorithm from [12] based on Fourier-Motzkin elimination. We will refer to the transformed loop nest as the *Execution* loop nest (as opposed to the *Send* and *Receive* loop nests). The outermost loop of the Execution loop nest will have an index of p, such as the second loop nest shown in Figure 2. The p loop will later be turned into an if statement of the form: `if (`lb_0`<=` p `&&` p `<=` ub_0`)`. This will then be placed inside another loop, where p will iterate over all the virtual processors that are mapped to the physical processor number `mypid`.

```
/* Receive Loop Nest */
for p_r = lb_{r_0} to ub_{r_0}
 for i_{r_1} = lb_{r_1} to ub_{r_1}
  ...
   for i_{r_n} = lb_{r_n} to ub_{r_n}
    for p_w = lb_{w_0} to ub_{w_0} {
     receive packet from processor p_w
     for i_{w_1} = lb_{w_1} to ub_{w_1}
      ...
       for i_{w_n} = lb_{w_n} to ub_{w_n}
        unpack (X[f(c,i_{w_1},...,i_{w_n})])
    }

/* Execution Loop Nest */
for p = lb_0 to ub_0
 for i_1 = lb_1 to ub_1
  ...
   for i_n = lb_n to ub_n
    ...
    ... = ... X[f(c,i_1,...,i_n)] ...
    X[g(c,i_1,...,i_n)] = ...
    ...

/* Send Loop Nest */
for p_w = lb_{w_0} to ub_{w_0}
 for i_{w_1} = lb_{w_1} to ub_{w_1}
  ...
   for i_{w_n} = lb_{w_n} to ub_{w_n}
    for p_r = lb_{r_0} to ub_{r_0} {
     for i_{r_1} = lb_{r_1} to ub_{r_1}
      ...
       for i_{r_n} = lb_{r_n} to ub_{r_n}
        pack (X[g(c,i_{r_1},...,i_{r_n})])
     send packet to processor p_r
    }
```

Figure 2: Before Overlapping Communication with Computation

For each read access to array location $X[f(c,i_1,\ldots,i_n)]$, we can generate a loop nest that will receive and unpack the data from the processor that created the value, such as the first loop nest shown in Figure 2. This is done by building a system of linear inequalities from the loop bounds, the LWT mapping, the computation decomposition for $\vec{i}_r$ and $\vec{i}_w$, and the constraint $p_r \neq p_w$. The loop nest is then generated from this system, such that the order of the loop indices is $(p_r, \vec{i}_r, p_w, \vec{i}_w)$, from outermost to innermost. The body of the loop nest is the statement to unpack the data. The statement to receive the packet is placed immediately following the determination of p_w. We will refer to this loop nest as the *Receive* loop nest.

The *Send* loop nest for each write access to array location $X[g(c,i_1,\ldots,i_n)]$ is generated is a similar fashion, except that the loop indices are ordered $(p_w, \vec{i}_w, p_r, \vec{i}_r)$, from outermost to innermost. The body of the loop nest is the statement to pack the data. The statement to send the packet is placed at the end of the p_r loop body. We will refer to this loop nest as the *Send* loop nest. Many of the innermost loops will be degenerate loops (containing only one iteration). By definition, the p_w and $\vec{i}_w$ loops of the Receive loop nest will be degenerate. The degenerate loops are later replaced by simple assignment statements.

3. OVERLAPPING COMMUNICATION WITH COMPUTATION

When the Receive loop is generated from an Execution loop, it will be a loop nest representing for the vector $[p_r, i_{r_1}, \ldots, i_{r_n}, p_w, i_{w_1}, \ldots, i_{w_n}]^T$. We replace the p_r and $\vec{i}_r$ loop indices with p and $\vec{i}$. Similarly, the send loop represents the vector $[p_w, i_{w_1}, \ldots, i_{w_n}, p_r, i_{r_1}, \ldots, i_{r_n}]^T$. We replace the p_w and $\vec{i}_w$ loop indices with p and $\vec{i}$. This causes the Receive, Send, and Execution loops to have the same indices for the outermost loops. This will allow us to combine the loops such that the loop bodies that are executed with the same values for p and $\vec{i}$, will be inside a single loop.

Figure 3 shows an example of loop nests that can be merged so that the communication overlaps with the computation. Notice that the outermost loops have the same loop index. The intersection of these iterations is [2 … N -1]. We take the three loop bodies and put them inside a new loop that iterates across this intersection, as shown in Figure 4. Then we need to insert new loops for the iterations that are left over.

```
for i_k = 2 to N
        Receive Loop Body
for i_k = 1 to N
        Execution Loop Body
for i_k = 1 to N-1
        Send Loop Body
```

Figure 3: Example of Loops That Can be Merged

```
for i_k = 1 to 1
        Execution Loop Body
        Send Loop Body
for i_k = 2 to N - 1
        Receive Loop Body
        Execution Loop Body
        Send Loop Body
for i_k = N to N
        Receive Loop Body
        Execution Loop Body
```

Figure 4: After Merging Loops

The advantage of this is that we can overlap communication with computation. The reason it is legal to do this is because

the iteration instance $\vec{i}_r$ of the Receive loop nest that generates the array value $X[f(c,i_1,\ldots,i_n)]$, used during iteration instance $\vec{i}$ of the Execution loop nest is such that $\vec{i}_r = \vec{i}$. Likewise for the Send loop nest. We can then merge the loops within these loops recursively to further improve performance.

Amarasinghe and Lam mentioned merging these loop nests. However, they did not provide any details of how it was implemented and whether it proves to be a useful technique. We provide an algorithm in this section that attempts to merge the Receive, Execution, and Send loop nests when it is possible to do so. We also provide some preliminary results that indicate that it can provide an improvement in performance.

The algorithm shown in Figure 5 is our algorithm to merge the loops. It should be applied to the loop nests after renaming the outermost loops indices to *p* and $\vec{i}$, but *before* translating the *p* loop to an if statement. This is a greedy algorithm, in the sense that it tries to find the first set of loops for which there is an intersection of iterations. *M* is used for the system of inequalities that represents the loop bounds of the current intersection. `combLoops` and `Body` are used to store the loops and their respective bodies that are in the current intersection. The algorithm steps `currentLoop` through the list of for loops (line 8). If the iterations of the `currentLoop` also intersect with the iterations in *M*, then it is added to the intersection (lines 10-14). Once we find a loop that does not intersect or has a different index, then we will either start over with the current loop, if the intersection is only a single loop (lines 16-21), or we will merge the loops in `combLoops`.

Once we have a set of loops whose iterations intersect, we will then proceed to merge these (lines 23-43). We create a new loop *L*, whose bounds satisfy the constraints of *M* and whose body is `Body`, and insert this into the code (lines 24-27). We then remove the loops that are being merged (line 29). Next we need to insert loops for the iterations that are left over. This is done by taking each of these loops and checking to see if its set of iterations intersects with the inverse of the bounds of *L* (i.e. `index` < *lb* or `index` > *ub*). If it does intersect with either of these, then there are iterations that are not included in *L*. Therefore we must add a new copy of this loop with the remaining iterations (lines 30-37). After we have merged a set of loops, we start `currentLoop` from the beginning (lines 39-43).

We stop this process when we cannot find any consecutive loops in the list of loops that have any common iterations (line 47). Then we recursively attempt to merge the inner loops (lines 48-49), as long as the bodies are lists of for loops only whose indices are the same variable.

4. RESULTS

We ran our compiler pass on the LU decomposition kernel shown in Figure 6. This is the same program segment that Amarasinghe and Lam used to demonstrate their compiler pass. The mapping of virtual processors to physical processors was done using a block mapping. After the Send and Receive loop nests are generated, the entire set of loops is placed inside a loop that maps the virtual processors to physical.

The two resulting programs (before and after merging the loops) were then compiled with a native compiler, linked with the MPI library, and run on the Beowulf cluster at UNCW. This Beowulf cluster consists of twelve dual-processor 1 GHz Pentium III processors, connected together by a 10/100 Fast Ethernet switch. The head machine has 512 Mbytes of memory and 75 Gbytes of disk space, and the client machines each have 256 Mbytes of memory and 30 Gbytes of disk space. We also ran these programs on an IBM RS/6000 SP at the North Carolina Supercomputing Center, which has 180 Winterhawk II nodes, each of which has four 375 MHz Power 3 processors, 2 Gbytes memory, 8 Mbytes L2 cache, and 64 Kbytes L1 cache.

Figure 7 shows the running times of these programs on the Beowulf cluster with a matrix size of 256x256. Overlapping the communication with computation by combining the loops in the manner that we described previously produced a significant improvement in performance. The improvement ranges between approximately 5 and 65 percent. The results for the IBM SP are shown in Figure 8. We were not able to run with a larger input size than 150x150, because the message buffers for MPI on that system are set so that we could not send very large packets. The results for the IBM range from 4 to 22 percent.

Although showing the results from one sample program does not prove that our algorithm always improves the performance, we present this data to suggest that this technique can make a difference. We do not yet know if this technique or the compiler can produce similar results on other types of problems. Our experience tells us that we will find that the compiler will work well for some problems (or problem domains) and poorly for others. We emphasize that this work is only preliminary. More experimentation is needed. However, the results presented here provide the motivation to continue this research.

5. CONCLUSIONS

Distributed-memory computer systems are increasing in popularity. However users of these systems must either write programs with explicit message-passing calls or use a Distributed Shared Memory (DSM) library. Although DSMs provide a convenient abstraction, Cox, et. al. [6] have shown that a compiler generated message-passing program can outperform the shared-memory version of the same program running with a DSM. This means that DSMs should not be the only option for the user.

Unfortunately, compiler technology for distributed-memory computer systems has lagged behind the technology for shared-memory computer systems. The reason is three fold: distributed-memory parallelism is more complex, there is a lack of open source parallelizing compilers that can generated message-passing code, and DSMs have given users another choice. In this paper, we introduced the *Paraguin* project at the University of North Carolina at Wilmington. The goal of

```
Input:   InLoops :  List of for loops
Output: List of merged for loops
Let:     M, c1, c2 : set of inequalities
         combLoops : list of for loops
         Body : list of instructions
         currentLoop, loop, L : for loop
         index : loop index variable
         lb, ub : expression

1   Repeat
2       M ← Ø
3       combLoops ← null
4       Body ← null
5       currentLoop ← InLoops.first()
6       index ← currentLoop.index ()
7
8       While ( NOT currentLoop.isPastEnd() ) do
9
10          If index = currentLoop.index() AND (M = Ø  OR  currentLoop.bounds() ∩ M ≠ Ø ) then
11              M ← M ∪ currentLoop.bounds()            /* Add this loop to the intersection */
12              combLoops.append( currentLoop )
13              Body.append( currentLoop.body() )
14              currentLoop ← currentLoop.next()
15
16          Else if | M | ≤ 2 then                      /* M contains bounds for only 1 for loop, so start over */
17              M ← currentLoop.bounds()
18              combLoops ← currentLoop
19              Body ← currentLoop.body()
20              index ← currentLoop.index()
21              currentLoop ← currentLoop.next()
22
23          Else                                        /* Merge the current set of loops */
24              lb ← makeLowerBound( M )
25              ub ← makeUpperBound( M )
26              L ← new ForLoop( index, lb, ub, Body )
27              InLoops.insertBefore( L, currentLoop )
28
29              For each loop ∈ combLoops do InLoops.remove( loop ) Endfor
30              For each loop ∈ combLoops do
31                  c1 ← loop.bounds() ∩ (index < lb)
32                  c2 ← loop.bounds() ∩ (index > ub)
33                  If | c1| ≠ 0 InLoops.insertBefore( new ForLoop( index,
34                           makeLowerBound( c1 ), makeUpperBound( c1 ), loop.body() ), L )
35                  If | c2| ≠ 0 InLoops.insertAfter( new ForLoop( index,
36                           makeLowerBound( c2 ), makeUpperBound( c2 ), loop.body() ), L )
37              Endfor
38
39              M ← Ø
40              combLoops ← null
41              Body ← null
42              currentLoop ← InLoops.first()
43              index ← currentLoop.index ()
44
45          Endif
46      Endwhile
47  Until M = Ø
48  For each loop ∈ InLoops do
49      Recursively apply this algorithm to loop.body()
50  Endfor
51  Return InLoops
```

Figure 5: Algorithm to Merge Loop Nests

```
for (i1 = 0; i1 <= N; i1++)
  for (i2 = i1 + 1; i2 <= N; i2++)
    X[i2][i1] /= X[i1][i1];
    for (i3 = i1 + 1; i3 <= N; i3++)
      X[i2][i3] -= X[i2][i1] * X[i1][i3];
  }
```

Figure 6: Sequential Version of LU Decomposition

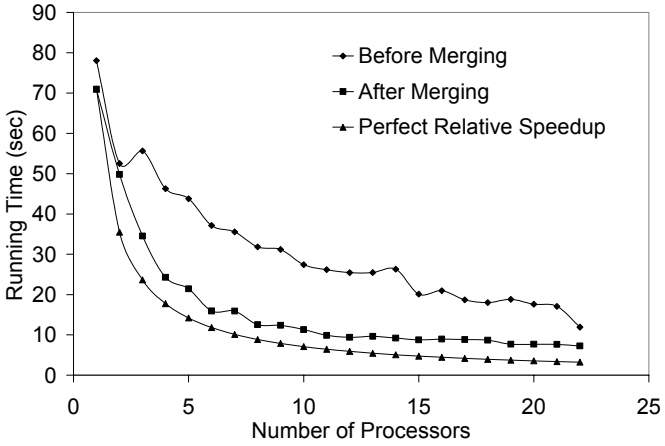

Figure 7: Results from the Beowulf Cluster. Input size is 256x256.

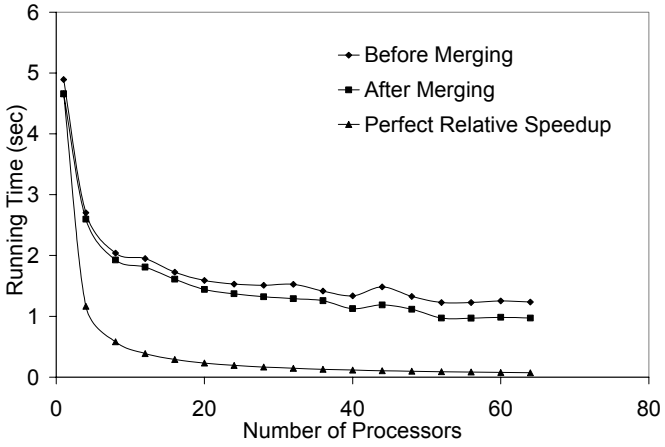

Figure 8: Results from the IBM SP. Input size is 150x150.

the Paraguin project is to build an open source message-passing parallelizing compiler for distributed-memory computer systems. We have discussed the progress we have made in developing this compiler and mention the parts that have not yet been developed. It is our intent that, by providing an open source compiler, we will stimulate research in automatic message-passing parallelism and encourage collaboration.

We demonstrated a technique to improve the performance of the resulting program by overlapping communication with computation. Although this concept was introduced in previous work [1], the algorithm was not developed nor shown to provide benefit. Our results indicate that the technique does significantly improve the performance. However, these results are only preliminary. Further experimentation is needed. These results indicate that our technique has merit, and that we should continue our investigations.

6. ACKNOWLEDGEMENTS

I would like to thank the North Carolina Supercomputing Center for providing access and time to their IBM SP. I would also like to thank UNCW and the Department of Computer Science for providing the Beowulf cluster.

REFERENCES

1. S. P. Amarasinghe and M. S. Lam, "Communication optimization and Code Generation for distributed memory machines," In the *Proc. of The ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, June 1993, pp. 126-138.
2. C. Amza, A.L. Cox, S. Dwarkada, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, Feb. 1996, pp. 18-28.
3. D.J. Scales and M.S. Lam, "An efficient shared memory layer for distributed memory machines," Computer Systems Laboratory Technical Report CSL-TR-94-627, Department of Computer Science, University of Stanford, 1994.
4. M.C. Carlisle and A. Rogers. "Software caching and computation migration in Olden." In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, Santa Barbara, Calif., July 1995, pp. 29-38.
5. Split-C, The Computer Science Division, University of California, Berkeley, http://www.cs.berkeley.edu/projects/parallel/castle/split-c/.
6. A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, "Evaluating the performance of software distributed shared memory as a target for parallelizing compilers," In the *Proc. of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997, pp. 475-482.
7. P. J. Keleher, "Update Protocols and cluster-based shared memory," *Computer Communications*, vol. 22, no.11, July 1999, pp.1045-1055.
8. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, no. 12, Dec. 1996, pp. 84-89.
9. P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1996.
10. A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine partitions," *Parallel Computing*, vol. 24, no. 3-4, 1998, pp. 445-475.
11. D. E. Maydan, S. P. Amarasinghe and M. S. Lam, "Array data-flow analysis and its use in array privatization," In the *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, Jan. 1993, pp. 2-15.
12. C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," In the *Proceedings of third ACM SIGPLAN Symposium on Principles & Practice of Programming Languages (PPOPP)*, Williamsburg, Virginia, Apr. 1991, pp. 39-50.

BIOGRAPHIES

Clayton S. Ferner
Department of Computer Science
University of North Carolina at Wilmington
601 S. College Rd.
Wilmington, NC 28403 USA

e-mail: cferner@uncwil.edu

Clayton Ferner is an Assistant Professor in the Department of Computer Science at the University of North Carolina at Wilmington. His research interests are parallel computing, compilers for parallel and distributed computing, and networks. Before joining the faculty at UNCW, Ferner was a member of the Technical Staff at Lucent Technologies. Ferner holds and MS and Ph.D. in Computer Science from the University of Denver. Ferner is a Member of IEEE and ACM.