John D. McGregor
and
Arthur M. Riehl
Guest Editors

Using an Architectural Knowledge Base to Generate Code for Parallel Computers

The authors present a reconfigurable compiler for distributed memory parallel computers that performs automatic program partitioning, mapping, and communication code generation under the guidance of directives supplied by the programmer.

Anthony E. Terrano, Stanley M. Dunn, and Joseph E. Peters

Generating code for existing parallel computers involves several nontrivial operations that have no analogue in code generation for uniprocessor architectures. In particular, the work must be partitioned into schedulable units, the resulting computational units must be assigned to specific processors, and additional code must be generated to support the resulting interprocessor communication and synchronization. These steps must be performed in order; the mapping cannot be undertaken until the problem has been partitioned, and the details of the interprocessor communication are not known until the mapping has been completed. The options for partitioning and mapping depend on the problem being solved and the algorithm and architecture being used. For each new combination of problem, algorithm and architecture, a new partitioning and mapping must be created and evaluated based on the actual communication costs incurred.

The efficiency of the resulting program is determined by the quantity and locality of the interprocessor communication: partitionings that result in small amounts of distributed communication will result in shorter execution times than ones that require large amounts of global communication. The actual efficiency is also strongly influenced by details of the architecture. Some architectures incur large overheads to initiate each individual communication, so the ability to aggregate messages may have a significant impact on the efficiency, while other architectures have large memory access latencies. If a particular choice of partition and map is found to be too inefficient when the communication cost is analyzed, it is usually necessary to repeat the code generation process from the beginning.

This research supported by the National Science Foundation under contract NSF MIP87-10829 and by ITT Defense Communications Division.

At present, the partitioning, mapping, and communication instructions are embedded in the source code for the problem. Changing any of them can involve substantial rewriting of the program. It is desirable to separate the programmatic statement of an algorithm from the machine-dependent partitioning, mapping, and communication specifications. Ideally, they would simply be orthogonal and a complete program would consist of an algorithm and a partitioning strategy. Changing the partitioning strategy would not require any modification of the statement of the algorithm.

In this article, we present a compiler for distributed memory parallel computers which performs automatic program partitioning, mapping, and communication code generation. The prototype compiler is implemented in Prolog, with the code generation advice incorporated as a set of Prolog rules. The compiler presents the programmer with a global address space programming model. A complete prototype system is presently able to compile standard C code employing global arrays of data into code suitable for execution on a generic distributed memory parallel computer. All partitioning and mapping is performed automatically, and communication primitives are generated.

SYSTEM ARCHITECTURE

Until now, there has been little or no need to include expert knowledge on the architecture of the target machine(s). In general, optimizing compilers are built following the methodology described in [8] and generate code in two steps. A divide and conquer control strategy is used to partition the compilation into small program fragments, and code is generated for each fragment independently by pattern matching [3]. Some inefficiencies are then removed by peephole optimizations. These compilers generate good code for serial machines, implicitly using information on the architec-

ture of the machine. The problem arises now because high performance machines are inherently parallel and often contain more than one processor or computing element.

Applying the serial methodology to these new high performance architectures results in inefficient code generation and/or undue hardships for the programmer. To develop efficient software, programmers must optimize programs by partitioning the data, setting up the communication, and optimizing the code for the architecture of the computing element. For example, Rymarczyk [5] gives guidelines for hand coding pipelined computers at the machine level.

Our goal is to automate these steps. Much of this information can be codified in partition directives for problem decomposition and rule bases that describe the processing element architecture and interconnection topology. The code to run on these computing elements is generated with the advice given by these directives and the knowledge of the architecture of a processing element and the topology of the interconnection network.

All compilers use processor-specific information during code generation and optimization. However, the organization of the compiler differs from other compilers because the architectural information is explicitly contained in a machine-specific rule base. The separation of architectural information from the compiler yields many advantages, such as simplifying compiler retargeting and supporting code generation by more sophisticated problem reduction control strategies.

Retargetable compilers, i.e., compilers that can generate code for several processors, also use architectural information during code generation. In general, retargetable compilers contain more explicit information; differences between target processors must be indicated to the compiler. However, retargetable compilers such as PQCC [2] or the Amsterdam Compiler Kit [6] also contain implicit information, targeting machines with similar architectures or architectures within a processor family.

Our design philosophy differs from these retargetable compiler designs in that all machine-specific information is explicitly defined in a rule base. This organization has many advantages:

1. Code generation for any processor. It is possible to generate code for any machine by simply defining a new rule base for the target processor. Furthermore, retargeting requires less time and effort, since no compiler source code has to be modified.
2. Transportability. Code is easily transported between processors with defined rule bases. Recompiling the source code with a different back end will yield assembly code for a different processor.
3. Modifications in processor design easily updated. If the manufacturer makes modifications in the processor design, only the rules affected by the modifications need to be modified. No compiler source code has to be modified.
4. Upward compatibility easily described. Upward compatible models of a processor family can be described simply by augmenting the existing rule base. The rule bases for the TMS320 family of digital signal processors were developed in this manner.
5. Differences in architectures evident. Differences in architectures are readily discernible by examining the rule bases.

The rule base is consulted by the two code generators of the compiler. The first code generator partitions data and generates code for control flow and the second generates code for the computation. The computation code generator consists of eight phases: atomic assignment evaluation, automatic assignment generator, data-flow analysis, expression evaluator, peephole optimizations on intermediate code, translation of the intermediate code into machine-specific mnemonics, machine-specific peephole optimizations, and the post-processor.

The function of the atomic assignment evaluator is to preserve high-level parallelism at the register level. An atomic assignment evaluation is performed on each concurrent assignment to determine if it is atomic, i.e., a high-level language assignment that can be translated into a single low-level assembly instruction on the target processor. Atomic assignments can contain high-level information on register-level parallelism that could not be otherwise described to the compiler. The atomic assignment evaluator consults the rule base, comparing each concurrent assignment against a list of the processor's atomic instructions.

If a concurrent assignment is not atomic, the rule base is consulted to determine if it is a concurrent assignment exhibiting register-level parallelism (CAP). A CAP is similar to an atomic assignment, except it is translated into a sequence of low-level instructions. As in the case of the atomic assignment, the CAP contains high-level information on register-level parallel operations.

A distinction is made between an atomic assignment and a CAP since it is possible for a concurrent assignment to be either type of instruction. If this is the case, the atomic assignment representation is preferred over the CAP representation of the concurrent assignment since a more efficient code sequence will be generated.

The rule base is consulted by comparing each concurrent assignment of the source code to a list of atomic assignments and CAPs. Concurrent assignments that do not appear in the list are passed on to the automatic assignment generator, while atomic assignments and CAPs are preserved.

The purpose of the automatic assignment generator is to break the concurrent assignment into a set of equivalent atomic assignments, using heuristics described by Mills [4]. These heuristics were originally used as design rules for generating sequential Pascal statements.

The output of the automatic assignment generator is three-address code [1], which is atomic for most processors. Three-address code specifies that an arithmetic

operation be performed on one or two operands, and stored in a destination register. Thus, assignments with complex expressions are broken down into a set of assignments with simple expressions.

The data-flow analysis phase performs global optimization on the three-address code. The rule base is consulted to determine if there are features on the target processor that allow optimization. For example, the rule base should be consulted to determine if the processor is pipelined. If it is pipelined, the data-flow analysis phase will perform pipeline reorganization on the code, using information from the rule base such as the number of pipeline stages.

The expression evaluator receives the code from the data-flow analysis phase and generates assembly code with generalized mnemonics. The expression evaluator references the rule base to determine whether the target processor is a 0-, 1-, 2-, or 3-address machine. The rule base must also be referenced to determine the available addressing modes on the target processor.

After peephole optimizations are made on the generalized assembly code, the generalized mnemonics are translated into machine-specific mnemonics. The rule base is consulted to determine how to translate the mnemonics.

The machine-specific peephole optimizer performs optimizations on the machine-specific assembly code. It must reference the rule base to determine what optimizations can be performed. For example, if the processor is pipelined, the rule base will supply information such as number of stages in the pipeline, and instructions that can take advantage of pipelining. Optimizations can then be performed, such as replacing branch institutions with delayed branches.

The rule base is also referenced to determine what type of reduction-in-strength strategies are efficient for the architecture. For example, the operation x^2 may be replaced by the operation $x \times x$ if it is a less expensive operation on the architecture. Another example might be the reduction of the expression $4 \times x$ into the instruction x SHR 2 (shift x right 2 positions) if it is a less expensive operation to implement on the architecture.

PROGRAMMING STRATEGY

As a concrete example of a partitioning strategy, we consider the iterative solution of partial differential equations (PDEs). These algorithms typically involve updating the current value of the solution at each grid point by replacing it with a simple linear combination of the values of nearby points. This updating procedure is repeatedly applied to all of the points in the grid until the change in the values drops below a predetermined threshold. The set of neighboring points whose values are used for the updating procedure is called a stencil: different algorithms employ different stencils. Using these algorithms, many grid points may be updated simultaneously. The only constraint is that, for each point being updated, none of the other points in its stencil can be updated at the same time.

The best mapping of this problem onto a distributed memory architecture depends on the number of grid points N_g and the number of processors N_p. If we assume that typical problem sizes range upward from hundreds of grid points per side, with thousands of points being more nearly ideal, then a generous lower bound for N_g will be 10^4 for two-dimensional problems and 10^6 for three-dimensional ones. Most distributed memory computers have N_p ranging up to 10^4; the largest values are still less than 10^5. We make the reasonable assumption that $N_g > N_p$. Under this assumption, the most natural mapping of the problem into the architecture can be constructed as follows:

1. Embed a plane into the multiprocessor communication network.
2. Tile the domain of the PDE problem with N_p identical polygons.
3. Assign one tile to each processor, with neighboring tiles assigned to neighboring processors under the embedding in step 1.

With this mapping, each processor will be solving a smaller, equivalent PDE boundary value problem, with the boundary conditions at each stage determined by the solutions obtained thus far on the neighboring tiles. The interprocessor communication load is determined by the need to keep the boundary conditions current on each tile.

The best tiling is algorithm dependent: for different stencils, different tiles will minimize the communication. It has been shown [7] that a simple geometrical construction may be used to generate the optimal tile for a given stencil: identify the points that are the farthest from the center and connect them with line segments (Figure 1). The resulting shape will be a convex polygon, with a specific orientation relative to the stencil and to the coordinate axes. Such a polygon is called the convex hull of the stencil. If the polygon can be used to tile the plane, while preserving the required orientation, it will be the optimal tile for the stencil.

For example, diamond tiling is optimal for the five- and nine-point cross stencils. It consists of squares oriented at 45° with respect to the coordinate axes. It is useful to parameterize the tile by the length of its diagonal k, which must be an even divisor of n. The number of points enclosed is $k^2/2$. For the five-point cross, each of the points in the perimeter must be communicated to the adjoining tiles, as indicated in Figure 2. The values of the remote points that must be exchanged to update all of the points in the tile are also indicated in Figure 2. The points at the top and the bottom of the tile adjoin three neighbors; the remaining points just one. Thus the total number of transfers is given by $T = perimeter + 4 = 2k + 2$.

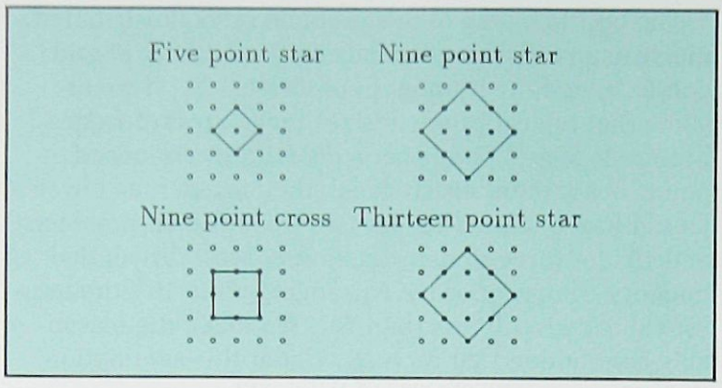

FIGURE 1. Some Common Stencils and their Convex Hulls

The size of the appropriate tile, as parameterized by k, is determined by the number of available processors and the number of grid points. Once the tile size has been computed, an origin for a global coordinate system is chosen and the coordinates of each tile are computed (see Figure 3). Next, the program code loop is unrolled for values of the left-hand side of the assignment lying in a single tile. The array references on the right-hand side may involve both local and nonlocal variables. The indices for the local variables are converted into relative coordinates with respect to the global coordinate system. The nonlocal variables references are converted into interprocessor communication instructions. The remote tile involved is identified; its global coordinates are used to label the communication channel to be used. The array indices are converted into local coordinates with respect to the remote tile, giving the local address of the variable being communicated.

ITERATIVE PARTIAL DIFFERENTIAL EQUATION SOLVER

The following program is an example employing the partitioning strategy given in the previous section for solving PDEs on a distributed memory architecture. We have removed the loop controlling the convergence criterion to better illustrate the automatic data partitioning and generation of communication primitives. The program is

```
#define NX 12
#define NY 12
#define NP 8
```

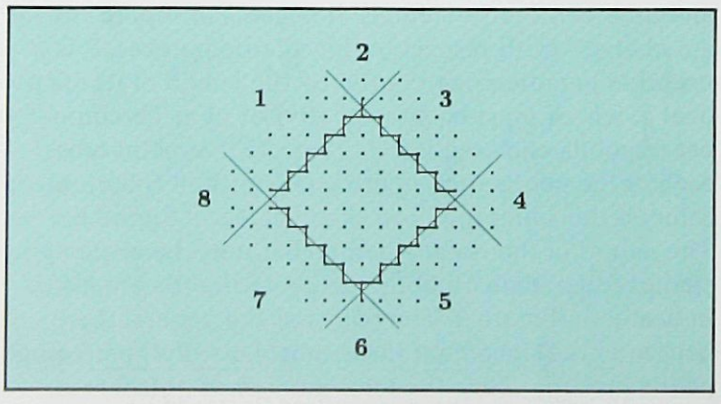

FIGURE 2. Five-Point Star Stencil for $k = 6$

```
#pragma processors NP
double phi[NX][NY];
#pragma tile phi[i][j] ((0 1) (1 0) (0−1) (−1 0))
int i, j;

main( ) {
forall(i := 0; i < NX; i++ : j := 0;
                              j < NY; j++ : (i+j) % 2 == 0)
  phi[i][j] := (phi[i][j+1] + phi[i+1][j]
                              + phi[i][j−1] + phi[i−1][j])/4
forall(i := 0; i < NX; i++ : j := 0;
                              j < NY; j++ : (i+j) % 2 != 0)
  phi[i][j] := (phi[i][j+1] + phi[i+1][j]
                              + phi[i][j−1] + phi[i−1][j])/4
}
```

The *#pragma* directive is the proposed ANSI preprocessor directive specifying compiler-dependent instructions. The tile command is a list of the points used in the update rule in terms of coordinates relative to each point. The processor directive defines the number of processors in the array. With the code given, the number of processors can be specified on the compiler command line, allowing the program to be compiled for an arbitrary number of processors without editing the source.

The *forall* statements are unordered loops. The construction consists of two pieces: a set of control statements enclosed in parentheses and a body consisting of a statement. As in a conventional loop, the *forall* is an operator that applies the function defined by its body on the domain specified by the control statements. The *forall* operator does not specify the order in which the domain is enumerated. Rather, it asserts that the function applications are independent and can be performed in any serial order or even simultaneously. In each application of the function, the statements must be executed in the order specified.

The control arguments of the *forall* are set membership specifications and these are separated by colons. Two types of specifications can be made. The first is a set membership function for a single index and takes the same form as a *for* statement, specifying a lower bound, an upper bound, and a rule for updating the index. The second is a characteristic function of an arbitrary subset of the Cartesian product of the indices. The domain of the loop is the intersection of the sets of indices specified by each of the set membership functions.

After scanning and parsing, the partitions and resulting communication primitives are generated by the control flow code generator. The control flow code generator has five phases of execution. In the first phase, the array points are partitioned among the processors. In phase two, the domain of each *forall* statement is computed. In phase three, the body of each forall is unrolled. In phase four, the concurrent assignments are partitioned into intraprocessor concurrent assignments. In the final phase, each global address is translated into a processor number and local address, and the neces-

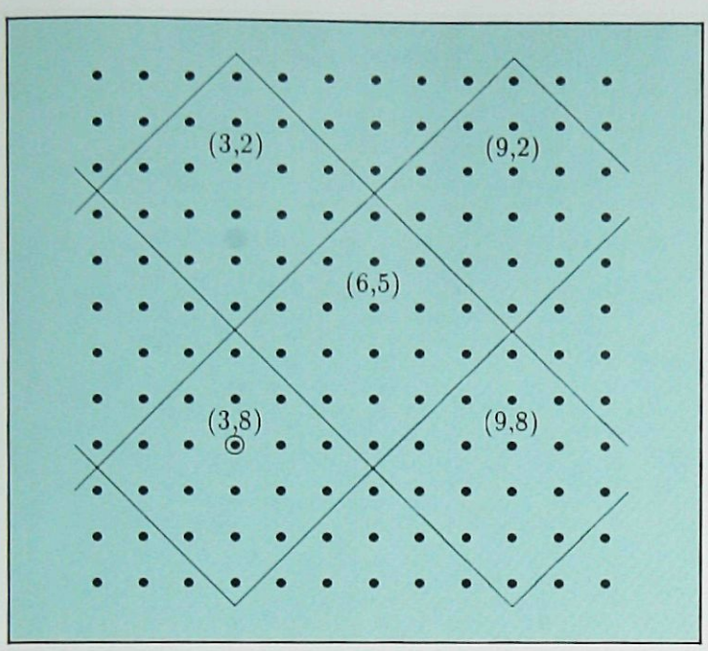

FIGURE 3. Global Coordinate System for Tiles (The origin for the center tile is circled)

sary interprocessor communication instructions are generated.

We have used the syntax from Occam for the interprocessor communication primitives:

port_a_b ? array [x y] for read
port_a_b ! array [x y] for write

where *_a_b* are the global grid coordinates of a processor, and [*x y*] are the local array indices of the word to be transferred.

In Figure 4, we show the intermediate assembly language code generated by the partitioning and control flow code generator. The assembly language code is then translated by the computation code generator which uses the rule base describing the architecture of the individual processors to generate machine instructions for the target machine. As described earlier, standard code optimizations are performed at this time. In particular, redundant communication instructions will be detected and removed with common subexpression combination techniques.

FAST FOURIER TRANSFORM

As a second example demonstrating the flexibility of the approach, we present the source and resulting intermediate code for a fast Fourier transform (FFT). The algorithm consists of two nested loops: an outer loop over the iterations of the transform and an inner loop in which the components of intermediate transforms are computed. The outer loop is necessarily serial. However, the inner loop can be directly converted into a *forall* loop. Furthermore, the two assignments in the body of the inner loop are independent, and may be executed as a single concurrent assignment. The resulting parallel code is:

```
#define N 16
#define lg(N) 4
#define NP 4
#pragma processors NP

float phi_r[N], phi_i[N], w_r[N/2], w_i[N/2];
#pragma tile phi_r[ j] (j % NP);
#pragma tile phi_i[ j] (j % NP);
float a, b;
int i, j, k;

main( ) {
  a, b := N/2, 1;
  for(i := 0; i < lg(N); i++) {
    a, b := a/2, b*2;
    forall(j := 0; j < a; j++ : k := 0; k < N; k += N/b)
      phi_r[j+k], phi_i[j+k],
                          phi_r[j+k+a], phi_i[j+k+a] :=
        phi_r[j+k] + phi_r[j+k+a],
        phi_i[j+k] + phi_i[j+k+a],
        (phi_r[j+k] − w_r[j*b] * phi_r[j+k+a]
        − phi_i[j+k] + w_i[j*b] * phi_i[j+k+a]),
        (phi_r[j+k] − w_i[j*b] * phi_r[j+k+a]
        + phi_i[j+k] − w_r[j*b] * phi_i[j+k+a]);
  }
}
```

Two partitioning strategies may be employed, depending on whether the communication occurs at the end of the calculation or at the beginning. Let N_p denote the number of processing elements. In the first case, the calculation is organized so that no communication is required until the final $\log_2(N_p)$ iterations. To

```
BEGIN
phi [3 1] := ( phi [3 2] + phi [4 1] + phi [3 0] + phi [2 1]) / 4
phi [2 2] := ( phi [2 3] + phi [3 2] + phi [2 1] + phi [1 2]) / 4
phi [4 2] := ( phi [4 3] + phi [5 2] + phi [4 1] + phi [3 2]) / 4
port_3_8 ! phi [4 1]
port_3_8 ! phi [3 0]
phi [1 3] := ( (port_9_2 ? phi [4 1]) + phi [2 3] + phi [1 2] + (port_9_2 ? phi [3 0])) / 4
phi [3 3] := ( phi [3 4] + phi [4 3] + phi [3 2] + phi [2 3]) / 4
port_9_8 ! phi [2 1]
port_9_8 ! phi [3 0]
phi [5 3] := ( (port_3_2 ? phi [2 1]) + (port_3_2 ? phi [3 0]) + phi [5 2] + phi [4 3]) / 4
port_3_8 ! phi [5 2]
port_3_8 ! phi [4 1]
phi [2 4] := ( (port_9_2 ? phi [5 2]) + phi [3 4] + phi [2 3] + (port_9_2 ? phi [4 1])) / 4
port_9_8 ! phi [1 2]
port_9_8 ! phi [2 1]
phi [4 4] := ( (port_3_2 ? phi [1 2]) + (port_3_2 ? phi [2 1]) + phi [4 3] + phi [3 4]) / 4
port_0_11 ! phi [3 0]
port_9_8 ! phi [1 2]
port_3_8 ! phi [5 2]
phi [3 5] := ( (port_0_11 ? phi [3 0]) + (port_3_2 ? phi [1 2]) + phi [3 4] +
                                                    (port_9_2 ? phi [5 2])) / 4
port_9_2 ! phi [1 3]
port_0_11 ! phi [3 5]
port_3_2 ! phi [5 3]
phi [3 0] := ( phi [3 1] + (port_3_8 ? phi [1 3]) + (port_0_11 ? phi [3 5]) +
                                                    (port_9_8 ? phi [5 3])) / 4
port_3_2 ! phi [5 3]
port_3_2 ! phi [4 4]
phi [2 1] := ( phi [2 2] + phi [3 1] + (port_9_8 ? phi [5 3]) + (port_9_8 ? phi [4 4])) / 4
port_9_2 ! phi [2 4]
port_9_2 ! phi [1 3]
phi [4 1] := ( phi [4 2] + (port_3_8 ? phi [2 4]) + (port_3_8 ? phi [1 3]) + phi [3 1]) / 4
port_3_2 ! phi [4 4]
port_3_2 ! phi [3 5]
phi [1 2] := ( phi [1 3] + phi [2 2] + (port_9_8 ? phi [4 4]) + (port_9_8 ? phi [3 5])) / 4
phi [3 2] := ( phi [3 3] + phi [4 2] + phi [3 1] + phi [2 2]) / 4
port_9_2 ! phi [3 5]
port_9_2 ! phi [2 4]
phi [5 2] := ( phi [5 3] + (port_3_8 ? phi [3 5]) + (port_3_8 ? phi [2 4]) + phi [4 2]) / 4
phi [2 3] := ( phi [2 4] + phi [3 3] + phi [2 2] + phi [1 3]) / 4
phi [4 3] := ( phi [4 4] + phi [5 3] + phi [4 2] + phi [3 3]) / 4
phi [3 4] := ( phi [3 5] + phi [4 4] + phi [3 3] + phi [2 4]) / 4
HALT
```

FIGURE 4. Intermediate Code Generated by the Control Flow Code Generator

accomplish this, those elements of phi whose indices are equivalent modulo N_p must be assigned in the same partition. The tile directive specifies this particular rule. In the second case, the communication occurs over the first $\log_2(N_p)$ iterations, and phi is partitioned into contiguous pieces of length N/N_p.

In Figure 5, we show the output for the case $N_p = 4$ and $N = 16$. As expected, the communication occurs only in the final two iterations. As in the preceding example, no optimizations have been performed. Conventional common subexpression elimination will remove the redundant read instructions. Note that the processors communicate pairwise in each iteration. The communications can be grouped into a single concurrent assignment for each iteration, possibly eliminating communication overhead latencies. Such an optimization is straightforward to implement once the target computer has been specified.

TARGET ARCHITECTURES

The intermediate language code produced by the compiler is at the level of conventional high-level programming languages for message passing computers: all data has been assigned to individual memories; it is exchanged only through explicit communication instructions; and the communication primitives support a completely connected virtual machine. The read and write primitives can be directly mapped onto the operating system calls provided on such computers as the Ncube/10 and the iPSC.

The resulting code may not be very efficient, however. These computers have fairly large communication latencies, and exchanging short messages will be slow. In general, the average message length will have to be increased by combining shorter ones. Some combining can be carried out automatically by performing conventional live/dead analysis on the variables in the communication instructions and moving the instructions into concurrent assignments. Further optimization will require global analysis, and cannot be performed automatically at this time. However, the intermediate code does provide a useful starting point for further optimizations.

The system presently generates machine code for several uniprocessor CPUs with multiple functional units, including the Weitek WTL 3364, the Texas Instruments TMS320 family, the NEC 7720, and the AT&T DSP32. Whenever possible, concurrent assignments are executed in parallel. It is also the compiler for the Coherent Parallel Computer presently under construction. This system is a massively parallel, synchronous MIMD computer. The startup overhead for an interprocessor message is only 2 cycles, and the throughput of each channel is one word per cycle. For this system, communication code will require only peephole optimizations to be efficient.

In the Coherent Parallel Computer, the computation units are distinct from the memory and communication system and can be customized for particular applica-

```
Code for Processor 0

BEGIN
phi_r [0] , phi_i [0] , phi_r [2] , phi_i [2] := ( phi_r [0] + phi_r [2] ) , ( phi_i [0] - phi_i [2] ) ,
        ( ( phi_r [0] - phi_r [2] ) * w_r [0] - ( phi_i [0] - phi_i [2] ) * w_i [0] ) ,
        ( ( phi_r [0] - phi_r [2] ) * w_i [0] +( phi_i [0] - phi_i [2] ) * w_r [0] ) ;
phi_r [1] , phi_i [1] , phi_r [3] , phi_i [3] := ( phi_r [1] + phi_r [3] ) , ( phi_i [1] - phi_i [3] ) ,
        ( ( phi_r [1] - phi_r [3] ) * w_r [4] - ( phi_i [1] - phi_i [3] ) * w_i [4] ) ,
        ( ( phi_r [1] - phi_r [3] ) * w_i [4] +( phi_i [1] - phi_i [3] ) * w_r [4] ) ;

phi_r [0] , phi_i [0] , phi_r [1] , phi_i [1] := ( phi_r [0] + phi_r [1] ) , ( phi_i [0] - phi_i [1] ) ,
        ( ( phi_r [0] - phi_r [1] ) * w_r [0] - ( phi_i [0] - phi_i [1] ) * w_i [0] ) ,
        ( ( phi_r [0] - phi_r [1] ) * w_i [0] +( phi_i [0] - phi_i [1] ) * w_r [0] ) ;
phi_r [2] , phi_i [2] , phi_r [3] , phi_i [3] := ( phi_r [2] + phi_r [3] ) , ( phi_i [2] - phi_i [3] ) ,
        ( ( phi_r [2] - phi_r [3] ) * w_r [0] - ( phi_i [2] - phi_i [3] ) * w_i [0] ) ,
        ( ( phi_r [2] - phi_r [3] ) * w_i [0] +( phi_i [2] - phi_i [3] ) * w_r [0] ) ;


( 2 ! phi_r [0] ) ; ( 2 ! phi_i [0] ) ; ( 2 ! phi_r [0] ) ; ( 2 ! phi_i [0] ) ;
phi_r [0] , phi_i [0] := ( phi_r [0] + ( 2 ? phi_r [0] ) ) , ( phi_i [0] - ( 2 ? phi_r [0] ) ) ;
( 2 ! phi_r [1] ) ; ( 2 ! phi_i [1] ) ; ( 2 ! phi_r [1] ) ; ( 2 ! phi_i [1] ) ;
phi_r [1] , phi_i [1] := ( phi_r [1] + ( 2 ? phi_r [1] ) ) , ( phi_i [1] - ( 2 ? phi_r [1] ) ) ;
( 2 ! phi_r [2] ) ; ( 2 ! phi_i [2] ) ; ( 2 ! phi_r [2] ) ; ( 2 ! phi_i [2] ) ;
phi_r [2] , phi_i [2] := ( phi_r [2] + ( 2 ? phi_r [2] ) ) , ( phi_i [2] - ( 2 ? phi_r [2] ) ) ;
( 2 ! phi_r [3] ) ; ( 2 ! phi_i [3] ) ; ( 2 ! phi_r [3] ) ; ( 2 ! phi_i [3] ) ;
phi_r [3] , phi_i [3] := ( phi_r [3] + ( 2 ? phi_r [3] ) ) , ( phi_i [3] - ( 2 ? phi_r [3] ) ) ;

( 1 ! phi_r [0] ) ; ( 1 ! phi_i [0] ) ; ( 1 ! phi_r [0] ) ; ( 1 ! phi_i [0] ) ;
phi_r [0] , phi_i [0] := ( phi_r [0] + ( 1 ? phi_r [0] ) ) , ( phi_i [0] - ( 1 ? phi_r [0] ) ) ;
( 1 ! phi_r [1] ) ; ( 1 ! phi_i [1] ) ; ( 1 ! phi_r [1] ) ; ( 1 ! phi_i [1] ) ;
phi_r [1] , phi_i [1] := ( phi_r [1] + ( 1 ? phi_r [1] ) ) , ( phi_i [1] - ( 1 ? phi_r [1] ) ) ;
( 1 ! phi_r [2] ) ; ( 1 ! phi_i [2] ) ; ( 1 ! phi_r [2] ) ; ( 1 ! phi_i [2] ) ;
phi_r [2] , phi_i [2] := ( phi_r [2] + ( 1 ? phi_r [2] ) ) , ( phi_i [2] - ( 1 ? phi_r [2] ) ) ;
( 1 ! phi_r [3] ) ; ( 1 ! phi_i [3] ) ; ( 1 ! phi_r [3] ) ; ( 1 ! phi_i [3] ) ;
phi_r [3] , phi_i [3] := ( phi_r [3] + ( 1 ? phi_r [3] ) ) , ( phi_i [3] - ( 1 ? phi_r [3] ) ) ;
HALT
```

(a)

```
Code for Processor 1

BEGIN
phi_r [0] , phi_i [0] , phi_r [2] , phi_i [2] := ( phi_r [0] + phi_r [2] ) , ( phi_i [0] - phi_i [2] ) ,
        ( ( phi_r [0] - phi_r [2] ) * w_r [1] - ( phi_i [0] - phi_i [2] ) * w_i [1] ) ,
        ( ( phi_r [0] - phi_r [2] ) * w_i [1] +( phi_i [0] - phi_i [2] ) * w_r [1] ) ;
phi_r [1] , phi_i [1] , phi_r [3] , phi_i [3] := ( phi_r [1] + phi_r [3] ) , ( phi_i [1] - phi_i [3] ) ,
        ( ( phi_r [1] - phi_r [3] ) * w_r [5] - ( phi_i [1] - phi_i [3] ) * w_i [5] ) ,
        ( ( phi_r [1] - phi_r [3] ) * w_i [5] +( phi_i [1] - phi_i [3] ) * w_r [5] ) ;

phi_r [0] , phi_i [0] , phi_r [1] , phi_i [1] := ( phi_r [0] + phi_r [1] ) , ( phi_i [0] - phi_i [1] ) ,
        ( ( phi_r [0] - phi_r [1] ) * w_r [2] - ( phi_i [0] - phi_i [1] ) * w_i [2] ) ,
        ( ( phi_r [0] - phi_r [1] ) * w_i [2] +( phi_i [0] - phi_i [1] ) * w_r [2] ) ;
phi_r [2] , phi_i [2] , phi_r [3] , phi_i [3] := ( phi_r [2] + phi_r [3] ) , ( phi_i [2] - phi_i [3] ) ,
        ( ( phi_r [2] - phi_r [3] ) * w_r [2] - ( phi_i [2] - phi_i [3] ) * w_i [2] ) ,
        ( ( phi_r [2] - phi_r [3] ) * w_i [2] +( phi_i [2] - phi_i [3] ) * w_r [2] ) ;


( 3 ! phi_r [0] ) ; ( 3 ! phi_i [0] ) ; ( 3 ! phi_r [0] ) ; ( 3 ! phi_i [0] ) ;
phi_r [0] , phi_i [0] := ( phi_r [0] + ( 3 ? phi_r [0] ) ) , ( phi_i [0] - ( 3 ? phi_r [0] ) ) ;
( 3 ! phi_r [1] ) ; ( 3 ! phi_i [1] ) ; ( 3 ! phi_r [1] ) ; ( 3 ! phi_i [1] ) ;
phi_r [1] , phi_i [1] := ( phi_r [1] + ( 3 ? phi_r [1] ) ) , ( phi_i [1] - ( 3 ? phi_r [1] ) ) ;
( 3 ! phi_r [2] ) ; ( 3 ! phi_i [2] ) ; ( 3 ! phi_r [2] ) ; ( 3 ! phi_i [2] ) ;
phi_r [2] , phi_i [2] := ( phi_r [2] + ( 3 ? phi_r [2] ) ) , ( phi_i [2] - ( 3 ? phi_r [2] ) ) ;
( 3 ! phi_r [3] ) ; ( 3 ! phi_i [3] ) ; ( 3 ! phi_r [3] ) ; ( 3 ! phi_i [3] ) ;
phi_r [3] , phi_i [3] := ( phi_r [3] + ( 3 ? phi_r [3] ) ) , ( phi_i [3] - ( 3 ? phi_r [3] ) ) ;


( 0 ! phi_r [0] ) ; ( 0 ! phi_i [0] ) ;
phi_r [0] , phi_i [0] :=
        ( ( ( 0 ? phi_r [0] ) - phi_r [0] ) * w_r [0] - ( ( ( 0 ? phi_i [0] ) - phi_i [0] ) * w_i [0] ) ,
        ( ( ( 0 ? phi_r [0] ) - phi_r [0] ) * w_i [0] + ( ( ( 0 ? phi_i [0] ) - phi_i [0] ) * w_r [0] ) ;
( 0 ! phi_r [1] ) ; ( 0 ! phi_i [1] ) ;
phi_r [1] , phi_i [1] :=
        ( ( ( 0 ? phi_r [1] ) - phi_r [1] ) * w_r [0] - ( ( ( 0 ? phi_i [1] ) - phi_i [1] ) * w_i [0] ) ,
        ( ( ( 0 ? phi_r [1] ) - phi_r [1] ) * w_i [0] + ( ( ( 0 ? phi_i [1] ) - phi_i [1] ) * w_r [0] ) ;
( 0 ! phi_r [2] ) ; ( 0 ! phi_i [2] ) ;
phi_r [2] , phi_i [2] :=
        ( ( ( 0 ? phi_r [2] ) - phi_r [2] ) * w_r [0] - ( ( ( 0 ? phi_i [2] ) - phi_i [2] ) * w_i [0] ) ,
        ( ( ( 0 ? phi_r [2] ) - phi_r [2] ) * w_i [0] + ( ( ( 0 ? phi_i [2] ) - phi_i [2] ) * w_r [0] ) ;
( 0 ! phi_r [3] ) ; ( 0 ! phi_i [3] ) ;
phi_r [3] , phi_i [3] :=
        ( ( ( 0 ? phi_r [3] ) - phi_r [3] ) * w_r [0] - ( ( ( 0 ? phi_i [3] ) - phi_i [3] ) * w_i [0] ) ,
        ( ( ( 0 ? phi_r [3] ) - phi_r [3] ) * w_i [0] + ( ( ( 0 ? phi_i [3] ) - phi_i [3] ) * w_r [0] ) ;
HALT
```

(b)

FIGURE 5. Intermediate Code for 4-Processor, 16-Point Fast Fourier Transform

```
                    Code for Processor 2

BEGIN
phix [0] , phiz [0] , phix [2] , phiz [2] := ( phix [0] + phix [2] ) , ( phiz [0] - phiz [2] ) ,
        ( ( phix [0] - phix [2] ) * wx [2] - ( phiz [0] - phiz [2] ) * wz [2] ) ,
        ( ( phix [0] - phix [2] ) * wz [2] +( phiz [0] - phiz [2] ) * wx [2] ) ;
phix [1] , phiz [1] , phix [3] , phiz [3] := ( phix [1] + phix [3] ) , ( phiz [1] - phiz [3] ) ,
        ( ( phix [1] - phix [3] ) * wx [6] - ( phiz [1] - phiz [3] ) * wz [6] ) ,
        ( ( phix [1] - phix [3] ) * wz [6] +( phiz [1] - phiz [3] ) * wx [6] ) ;

phix [0] , phiz [0] , phix [1] , phiz [1] := ( phix [0] + phix [1] ) , ( phiz [0] - phiz [1] ) ,
        ( ( phix [0] - phix [1] ) * wx [4] - ( phiz [0] - phiz [1] ) * wz [4] ) ,
        ( ( phix [0] - phix [1] ) * wz [4] +( phiz [0] - phiz [1] ) * wx [4] ) ;
phix [2] , phiz [2] , phix [3] , phiz [3] := ( phix [2] + phix [3] ) , ( phiz [2] - phiz [3] ) ,
        ( ( phix [2] - phix [3] ) * wx [4] - ( phiz [2] - phiz [3] ) * wz [4] ) ,
        ( ( phix [2] - phix [3] ) * wz [4] +( phiz [2] - phiz [3] ) * wx [4] ) ;


( 0 ! phix [0] ) ; ( 0 ! phiz [0] ) ;
phix [0] , phiz [0] :=
        ( ( ( 0 ? phix [0] ) - phix [0] ) * wx [0] - ( ( 0 ? phiz [0] ) - phiz [0] ) * wz [0] ) ,
        ( ( ( 0 ? phix [0] ) - phix [0] ) * wz [0] + ( ( 0 ? phiz [0] ) - phiz [0] ) * wx [0] ) ,
( 0 ! phix [1] ) ; ( 0 ! phiz [1] ) ;
phix [1] , phiz [1] :=
        ( ( ( 0 ? phix [1] ) - phix [1] ) * wx [0] - ( ( 0 ? phiz [1] ) - phiz [1] ) * wz [0] ) ,
        ( ( ( 0 ? phix [1] ) - phix [1] ) * wz [0] + ( ( 0 ? phiz [1] ) - phiz [1] ) * wx [0] ) ;
( 0 ! phix [2] ) ; ( 0 ! phiz [2] ) ;
phix [2] , phiz [2] :=
        ( ( ( 0 ? phix [2] ) - phix [2] ) * wx [0] - ( ( 0 ? phiz [2] ) - phiz [2] ) * wz [0] ) ,
        ( ( ( 0 ? phix [2] ) - phix [2] ) * wz [0] + ( ( 0 ? phiz [2] ) - phiz [2] ) * wx [0] ) ;
( 0 ! phix [3] ) ; ( 0 ! phiz [3] ) ;
phix [3] , phiz [3] :=
        ( ( ( 0 ? phix [3] ) - phix [3] ) * wx [0] - ( ( 0 ? phiz [3] ) - phiz [3] ) * wz [0] ) ,
        ( ( ( 0 ? phix [3] ) - phix [3] ) * wz [0] + ( ( 0 ? phiz [3] ) - phiz [3] ) * wx [0] ) ;


( 3 ! phix [0] ) ; ( 3 ! phiz [0] ) ; ( 3 ! phix [0] ) ; ( 3 ! phiz [0] ) ;
phix [0] , phiz [0] := ( phix [0] + ( 3 ? phix [0] ) ) , ( phiz [0] - ( 3 ? phiz [0] ) ) ;
( 3 ! phix [1] ) ; ( 3 ! phiz [1] ) ; ( 3 ! phix [1] ) ; ( 3 ! phiz [1] ) ;
phix [1] , phiz [1] := ( phix [1] + ( 3 ? phix [1] ) ) , ( phiz [1] - ( 3 ? phiz [1] ) ) ;
( 3 ! phix [2] ) ; ( 3 ! phiz [2] ) ; ( 3 ! phix [2] ) ; ( 3 ! phiz [2] ) ;
phix [2] , phiz [2] := ( phix [2] + ( 3 ? phix [2] ) ) , ( phiz [2] - ( 3 ? phiz [2] ) ) ;
( 3 ! phix [3] ) ; ( 3 ! phiz [3] ) ; ( 3 ! phix [3] ) ; ( 3 ! phiz [3] ) ;
phix [3] , phiz [3] := ( phix [3] + ( 3 ? phix [3] ) ) , ( phiz [3] - ( 3 ? phiz [3] ) ) ;
HALT
```

(c)

```
BEGIN                   Code for Processor 3
phix [0] , phiz [0] , phix [2] , phiz [2] := ( phix [0] + phix [2] ) , ( phiz [0] - phiz [2] ) ,
        ( ( phix [0] - phix [2] ) * wx [3] - ( phiz [0] - phiz [2] ) * wz [3] ) ,
        ( ( phix [0] - phix [2] ) * wz [3] +( phiz [0] - phiz [2] ) * wx [3] ) ;
phix [1] , phiz [1] , phix [3] , phiz [3] := ( phix [1] + phix [3] ) , ( phiz [1] - phiz [3] ) ,
        ( ( phix [1] - phix [3] ) * wx [7] - ( phiz [1] - phiz [3] ) * wz [7] ) ,
        ( ( phix [1] - phix [3] ) * wz [7] +( phiz [1] - phiz [3] ) * wx [7] ) ;

phix [0] , phiz [0] , phix [1] , phiz [1] := ( phix [0] + phix [1] ) , ( phiz [0] - phiz [1] ) ,
        ( ( phix [0] - phix [1] ) * wx [6] - ( phiz [0] - phiz [1] ) * wz [6] ) ,
        ( ( phix [0] - phix [1] ) * wz [6] +( phiz [0] - phiz [1] ) * wx [6] ) ;
phix [2] , phiz [2] , phix [3] , phiz [3] := ( phix [2] + phix [3] ) , ( phiz [2] - phiz [3] ) ,
        ( ( phix [2] - phix [3] ) * wx [6] - ( phiz [2] - phiz [3] ) * wz [6] ) ,
        ( ( phix [2] - phix [3] ) * wz [6] +( phiz [2] - phiz [3] ) * wx [6] ) ;

( 1 ! phix [0] ) ; ( 1 ! phiz [0] ) ;
phix [0] , phiz [0] :=
        ( ( ( 1 ? phix [0] ) - phix [0] ) * wx [0] - ( ( 1 ? phiz [0] ) - phiz [0] ) * wz [0] ) ,
        ( ( ( 1 ? phix [0] ) - phix [0] ) * wz [0] + ( ( 1 ? phiz [0] ) - phiz [0] ) * wx [0] ) ,
( 1 ! phix [1] ) ; ( 1 ! phiz [1] ) ;
phix [1] , phiz [1] :=
        ( ( ( 1 ? phix [1] ) - phix [1] ) * wx [0] - ( ( 1 ? phiz [1] ) - phiz [1] ) * wz [0] ) ,
        ( ( ( 1 ? phix [1] ) - phix [1] ) * wz [0] + ( ( 1 ? phiz [1] ) - phiz [1] ) * wx [0] ) ,
( 1 ! phix [2] ) ; ( 1 ! phiz [2] ) ;
phix [2] , phiz [2] :=
        ( ( ( 1 ? phix [2] ) - phix [2] ) * wx [0] - ( ( 1 ? phiz [2] ) - phiz [2] ) * wz [0] ) ,
        ( ( ( 1 ? phix [2] ) - phix [2] ) * wz [0] + ( ( 1 ? phiz [2] ) - phiz [2] ) * wx [0] ) ;
( 1 ! phix [3] ) ; ( 1 ! phiz [3] ) ;
phix [3] , phiz [3] :=
        ( ( ( 1 ? phix [3] ) - phix [3] ) * wx [0] - ( ( 1 ? phiz [3] ) - phiz [3] ) * wz [0] ) ,
        ( ( ( 1 ? phix [3] ) - phix [3] ) * wz [0] + ( ( 1 ? phiz [3] ) - phiz [3] ) * wx [0] ) ,


( 2 ! phix [0] ) ; ( 2 ! phiz [0] ) ;
phix [0] , phiz [0] :=
        ( ( ( 2 ? phix [0] ) - phix [0] ) * wx [0] - ( ( 2 ? phiz [0] ) - phiz [0] ) * wz [0] ) ,
        ( ( ( 2 ? phix [0] ) - phix [0] ) * wz [0] + ( ( 2 ? phiz [0] ) - phiz [0] ) * wx [0] ) ;
( 2 ! phix [1] ) ; ( 2 ! phiz [1] ) ;
phix [1] , phiz [1] :=
        ( ( ( 2 ? phix [1] ) - phix [1] ) * wx [0] - ( ( 2 ? phiz [1] ) - phiz [1] ) * wz [0] ) ,
        ( ( ( 2 ? phix [1] ) - phix [1] ) * wz [0] + ( ( 2 ? phiz [1] ) - phiz [1] ) * wx [0] ) ;
( 2 ! phix [2] ) ; ( 2 ! phiz [2] ) ;
phix [2] , phiz [2] :=
        ( ( ( 2 ? phix [2] ) - phix [2] ) * wx [0] - ( ( 2 ? phiz [2] ) - phiz [2] ) * wz [0] ) ,
        ( ( ( 2 ? phix [2] ) - phix [2] ) * wz [0] + ( ( 2 ? phiz [2] ) - phiz [2] ) * wx [0] ) ;
( 2 ! phix [3] ) ; ( 2 ! phiz [3] ) ;
phix [3] , phiz [3] :=
        ( ( ( 2 ? phix [3] ) - phix [3] ) * wx [0] - ( ( 2 ? phiz [3] ) - phiz [3] ) * wz [0] ) ,
        ( ( ( 2 ? phix [3] ) - phix [3] ) * wz [0] + ( ( 2 ? phiz [3] ) - phiz [3] ) * wx [0] ) ;
HALT
```

(d)

FIGURE 5. *Continued*

tions. We are presently considering three implementations of the machine. The first employs commercial floating point chips and will be suitable for problems involving linear algebra, iterative and time-evolution PDE solvers, probabilistic relaxation labeling, low-level image processing applications such as edge, boundary, and region detection algorithms and spectral algorithms in computational fluid dynamics. Dedicated computers for other applications can be constructed by combining the communication and memory system with custom computational units. A programmable computer for DNA matching and sequencing can be implemented with fast string pattern matchers. And a massively parallel lattice gas simulation machine can be implemented by using a programmable finite automaton for the computational unit.

CONCLUSION

To make effective use of a parallel computer, the user is required to break the program into subtasks, devise a mapping of the subtasks onto processing elements, and finally add interprocessor communication instructions as required. None of these tasks are trivial, and the efficiency of the resulting code depends on decisions made at each stage. At present, it is possible only to solve these problems by trial and error.

In this article, we have presented a rapid prototyping and retargetable compiler system which addresses these programming tasks. By employing partitioning directives, the system automatically decomposes the problem into an appropriate set of subtasks and generates any necessary intertask communication instructions. With this system, the user is presented with a single global address space. As examples, we have shown the compilation from C into an intermediate assembly language for a point-iterative elliptic partial differential equation solver and for a fast Fourier transform.

Code generation is an optimization process. For von Neumann architectures, a divide and conquer approach is adequate, since the individual operations which comprise a computation are essentially independent. The inefficiencies that result from generating a single machine instruction at a time can largely be corrected with subsequent peephole optimizations. For other architectures however, especially those with a multitude of independent functional units, as in parallel or pipelined computers, the instructions are much more interdependent, and generating good code requires consideration of significantly different execution orders. In this situation, more sophisticated optimization techniques are demanded. We have presented a compiler which uses a combination of both problem reduction and heuristic search to generate code. The problem reduction strategy consists of identifying maximal concurrent assignments rather than individual operations. Heuristic search is then used to find the locally optimal code for each concurrent assignment and to combine the partial solutions to produce globally (nearly) optimal code.

Partitioning information should be incorporated into type declarations. In general, type information is the

specification of the representation of data for a particular operation. Viewed from this perspective, the partitioning of data is a particular form of typing. When implemented as type declarations, partitioning rules can be extended to encompass dynamically allocated variables. In addition, complex communication operations can be specified as a typecasting, without requiring the introduction of auxiliary variables or temporary storage. Furthermore, by means of a simple generalization of the principle of reduction in strength, the techniques described here can be employed to solve the problems of dereferencing an arbitrary pointer and performing pointer arithmetic. We are presently extending the system to permit the specification of partitioning information as a generalization of a type declaration and to handle arbitrary pointer operations.

REFERENCES

1. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.
2. Cattell, R.G.G., Newcomer, J.M., and Leverett, B.W. Code generation in a machine-independent compiler. *ACM SIGPLAN Notices 14*, 8 (Aug. 1979), 65–74.
3. Glanville, R.S., and Graham, S.L. A new method for compiler code generation. In *5th Annual ACM Symposium on Principles of Programming Languages* (Tucson, Ariz., Jan. 23–25, 1978), pp. 231–240.
4. Mills, H., et al. *Principles of Computer Programming: A Mathematical Approach.* Allyn & Bacon, Newton, Mass., 1987.
5. Rymarczyk, J.W. Coding guidelines for pipelined processors. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar. 1–3, 1982), pp. 12–19.
6. Tanenbaum, A.S., et al. A practical tool kit for making portable compilers. *Commun. ACM 26*, 9 (Sept. 1983), 654–660.
7. Terrano, A.E. On the grain-size dependence of interprocessor communication demand. Submitted to *IEEE Trans. Comput.*
8. Wulf, W., et al. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Distributed memory, parallel computing, parallelizing compiler, retargetable compiler for novel architectures, virtual global address space

ABOUT THE AUTHORS:

ANTHONY E. TERRANO holds a Ph.D. in theoretical physics from Caltech. As an assistant professor at Columbia University, he was one of the principal designers for the QCD Machine, a parallel supercomputer built to address numerical problems in theoretical physics. His research interests include parallel computer architecture, application-specific computer design, parallel languages and algorithms and compiler design. Author's Present Address: Department of Electrical and Computer Engineering, Drexel University, Philadelphia, PA 19104. terrano@occlusal.rutgers.edu.

STANLEY M. DUNN is an assistant professor of biomedical engineering and a member of the Laboratory for Computer Science Research at Rutgers University. He is also a research associate professor at the University of Medicine and Dentistry of New Jersey. His research interests are in computer vision, image understanding, and parallel computing environments for vision and image understanding systems. Author's Present Address: Department of Biomedical Engineering, Rutger's University, P.O. Box 909, Piscataway, NJ 08855-0909. smd@occlusal.rutgers.edu.

JOSEPH E. PETERS is a Ph.D. candidate and research assistant in the Computer Science Department at Rutgers University, where he earned both his B.S. and M.S. degrees in electrical engineering. His research interests include compilers and parallel processing. Author's Present Address: Department of Computer Science, Hill Center, Busch Campus, Rutgers University, Piscataway, NJ 08855. peters@occlusal.rutgers.edu.