6-1-1990

Loop Coalescing and Scheduling for Barrier MIMD Architectures

Matthew T. O'Keefe
Purdue University

Henry G. Dietz
Purdue University, hankd@ecn.purdue.edu

Loop Coalescing and Scheduling for Barrier MIMD Architectures

Matthew T. O'Keefe
Henry G. Dietz

TR-EE 90-44
June 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Loop Coalescing and Scheduling for Barrier MIMD Architectures

Matthew T. O'Keefe and Henry G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
June 7, 1990
hankd@ecn.purdue.edu
(317) 494 3357

ABSTRACT

Barrier MIMDs are asynchronous Multiple Instruction stream Multiple Data stream architectures capable of parallel execution of variable execution time instructions and arbitrary control flow (e.g., `while` loops and calls); however, they differ from conventional MIMDs in that the need for run-time synchronization is significantly reduced. This work considers the problem of scheduling nested loop structures on a barrier MIMD. The basic approach employs *loop coalescing*, a technique for transforming a multiply-nested loop into a single loop. Loop coalescing is extended to nested *triangular loops*, in which inner loop bounds are functions of outer loop indices. Also, a more efficient scheme to generate the original loop indices from the coalesced index is proposed for the case of constant loop bounds. These results are general, and can be applied to extend previous work using loop coalescing techniques. We concentrate on using loop coalescing for scheduling barrier MIMDs, and show how previous work in loop transformations [Wol89], [Pol88] and linear scheduling theory [ShF88], [ShO90] can be applied to this problem.

Key phrases: Loop Coalescing, Loop Transformation, Barrier Synchronization, Compiler Parallelization, Compiler Optimization, Static Barrier MIMD.

1. Introduction

Parallel computer architectures hold great promise for solving large, compute-intensive problems. To fully exploit parallel machines, it is necessary to translate applications software into efficient parallel code. Most of the parallelism in programs is found in loops, and techniques are necessary to extract loop parallelism and exploit it at run-time.

This work considers loop parallelization and scheduling for a new class of parallel machines called *barrier MIMD (Multiple Instruction stream, Multiple Data stream) architectures* [DiS88], [OKD90]. Barrier MIMDs are characterized by a fast, flexible hardware barrier synchronization mechanism that executes in a few clock cycles. Barriers may be applied across any arbitrary subset of the processors. Recall that a processor performs the following steps at a barrier synchronization point:

[1] Marks itself as present at the barrier.

[2] Waits for all other *participating* processors to arrive at the barrier.

[3] After all participating processors have arrived at the barrier, it continues execution past the barrier.

In a barrier MIMD, step [3] is modified so that processors proceed past the barrier *simultaneously*. Using this property, previous work [ZaD90] has shown that for basic blocks of code executed on a barrier MIMD, static scheduling can remove many unnecessary synchronizations at compile-time.

This work considers the problem of scheduling nested loop structures on a barrier MIMD. Since the processors have separate, independent control streams, the body of the nested loops can contain subroutine calls, `IF` statements, other control flow constructs and variable-time instructions. Hence, barrier MIMDs can exploit loop parallelism that VLIW and SIMD machines, limited to a single control stream, must ignore.

The basic approach employs *loop coalescing* [Pol88], a technique for transforming a multiply-nested loop into a single loop. Loop coalescing is extended to nested *triangular loops*, in which inner loop bounds are functions of outer loop indices. Also, a more efficient scheme to generate the original loop indices from the coalesced index is proposed for the case of constant loop bounds. These results are general, and can be applied to extend previous work using loop coalescing techniques. We concentrate on using loop coalescing for scheduling barrier MIMDs, and show how previous work in loop transformations [Wol89], [Pol88] and linear scheduling theory [ShF88], [ShO90] can be applied to this problem.

This manuscript is organized as follows. In section two, some previous work in scheduling parallel, shared-memory MIMD architectures is reviewed. Section three extends the loop coalescing transformation to triangular loops, and proposes an improved technique for coalescing rectangular loops (with constant upper and lower bounds). Section four shows how a coalesced loop can be scheduled on a barrier MIMD; an algorithm for generating the proper sequence of barrier synchronizations is given. Finally, conclusions and directions for future work are given in section five.

2. Previous Work

Scheduling schemes for parallel architectures fall into two broad classes: *static* and *dynamic*. In *static scheduling*, compile-time information is used to determine a binding between tasks and processors before program execution begins; this approach has very low run-time overhead but can result in poor load balancing under certain conditions. In contrast, *dynamic scheduling* employs run-time information to perform this binding during program execution, resulting in good load balancing at the expense of high run-time overhead. Hybrid schemes between static and dynamic scheduling are also possible.

The Flow Model Processor (FMP) MIMD architecture [LuB80], [Lun87] employs static scheduling for allocating parallel loop iterations to processors. The FMP is a shared-memory MIMD notable for its fast hardware barrier synchronization mechanism and a *decentralized* approach to scheduling and control. The target application domain for the machine was computational aerodynamics, although it supports a general MIMD model.

The Flow Model Processor was programmed using an extended Fortran language that included a parallel `DO` loop construct, the `DOALL`. The `DOALL` provided the basic parallel construct for the FMP; no dependencies exist between `DOALL` iterations so they can be executed in parallel. The iteration space for the `DOALL` was described by a `DOMAIN` statement. For example, the declaration

```
DOMAIN/EYEJAY/: I=1, IMAX; J=1, JMAX
```

declares that there are `IMAX*JMAX` elements, each consisting of a pair of values for `I` and `J` in the ranges shown. Each pair of index values specifies an *instance* `IJ` of the loop body. Index sets created with `DOMAIN` statements such as `EYEJAY` are called *domains*. In the aerodynamic flow codes to be executed on the FMP only rectangular domains were considered, as these were the most common domains found in such code. Loops iterating over rectangular domains are called *rectangular loops;* they correspond to nested loops with constant upper and lower bounds.

Parallel execution of the `DOALL` iterations began when control flow in the program reached the `DOALL`. Early FMP studies considered employing a centralized control unit to compute an optimal allocation of the loop instances. However, the final design employed a decentralized mechanism for static loop scheduling[1]: processor id numbers P were assigned from 0 to $PMAX-1$, where $PMAX$ was the number of processors. Each processor was also given the maximum instance number and the number of processors executing the `DOALL`. Processor P began by executing instance number $IJ=P$. In the previous example, the index variables were `I` and `J`: each processor can determine these index variables from the instance number IJ with the following equation:

$$IJ = J * IMAX + I$$

In this case, $I = IJ \; mod \; IMAX$ and $J = IJ \; div \; IMAX$. After computing each instance, a processor increments its instance number IJ by $PMAX$ to obtain the next instance to compute. This mapping of iterations to processors is called *interleaved allocation* in this work. This continues until $IJ > IJMAX$. All processors then participate in a hardware barrier synchronization before program execution proceeds.

A centralized control mechanism is needed only at the beginning of the `DOALL` to broadcast the number of processors participating and the maximum instance number. At that point, processors can independently compute the iterations assigned to them without accessing any central control or shared variables. This avoids the contention and run-time overhead inherent in a dynamic scheduling scheme. The FMP loop scheduling technique establishes a binding at compile-time between loop iterations and a virtual machine, where each processor is given an equal number of iterations; a binding between the virtual machine and actual machine is made at run-time.

Notice that the loop iterations are divided up among the processors equally and are allocated "all at once" at the beginning of parallel execution. If loop iteration execution times vary widely, there would seem to be a danger that the processors would finish at widely different times. Detailed instruction-level simulation studies conducted during the design of the Flow Model Processor showed that the execution time of iterations was close and the amount of processor time spent waiting was small. Kruskal and Weiss [KrW85] studied this problem and showed that for a wide class of distributions for iteration execution times, allocating an equal number of iterations to each processor all at once has good efficiency.

1. Although the scheduling is static, i.e. performed at compile-time, recompilation is unnecessary if the machine configuration changes or different numbers of processors are used to execute the `DOALL`.

A dynamic scheduling scheme known as *guided-self scheduling* [Pol88] was developed by Polychronopolous and Kuck to reduce the amount of run-time overhead while still maintaining good load balancing among processors. *Loop coalescing* is applied to transform nested parallel DO loops with constant upper and lower bounds (i.e., rectangular loops) into a single parallel DO loop with a single dimension. Other transformations such as loop distribution and loop interchanging [Wol89] can be applied to transform a set of nested loops into the proper form for coalescing. In essence, loop coalescing is a compiler technique that constructs the FMP domains automatically at compile-time.

Processors obtain iterations of the coalesced loop by accessing the shared coalesced index variable; the number of iterations given at each access varys dynamically, starting out large but tapering off to a single iteration according to

$$x_i = \left\lceil \frac{R_i}{PMAX} \right\rceil, \quad R_{i+1} \leftarrow R_i - x_i$$

where R_i is the number of iterations remaining at step i (and $R_1 = N$, the total number of iterations in the loop), x_i is the number of iterations given to the processor requesting work at step i, and $PMAX$ is the number of processors. This adaptive variation in allocated work reduces the number of synchronization operations compared to allocating a single iteration at a time. The number of synchronization operations is also reduced by coalescing, since only a single index, not multiple indices as in the original loops, need be accessed. In guided self-scheduling, the processor at step i executes iterations $[N-R_i+1, \ldots, N-R_i+x_i]$. Mapping consecutive iterations to a single processor is called *consecutive allocation* in this work.

3. Generalized Loop Coalescing

In this section, a technique for coalescing triangular nested loops with inner loop bounds that are functions of the outer loop indices is proposed. An improved method for generating the original indices from the coalesced index rectangular loops is also given. Triangular loops are ubiquitous in the numerical linear algebra codes [DoM79], [GoV83] that are perhaps the most common input to vectorizing and parallelizing compilers. The new technique broadens the applicability of loop coalescing.

The approach used in the FMP to generate the original loop indices from the coalesced index can be applied to rectangular loops with nest levels greater than two. The basic idea is to coalesce starting from the innermost nest levels and proceed outward. The two innermost levels are coalesced, followed by the next innermost loop and the coalesced loop formed in the previous step, and so on until the outermost loop in the loop nest is reached and coalesced. Consider the following loop:

```
DO 10 I = 1, IMAX
   DO 20 J = 1, JMAX
      DO 30 K = 1, KMAX
            ....
```

Coalescing the two innermost loops yields

```
DO 10 I = 1, IMAX
   DO 20 JK = 1, JMAX*KMAX
      J = JK div KMAX
      K = JK mod KMAX
            ....
```

followed by the remaining loops

```
DO 10 IJK = 1, IMAX*JMAX*KMAX
   I  = IJK div JMAX*KMAX
   JK = IJK mod JMAX*KMAX
   J  = JK div KMAX
   K  = JK mod KMAX
            ....
```

Each coalescing step results in the need for one integer division[2] at run-time to generate the loop indices from the coalesced index, and this example requires two integer divisions. In contrast, Polychronopolous's scheme [Pol88] requires two integer divisions, one multiplication and one subtraction *per loop index*, resulting in six integer divisions, three multiplications, and three subtractions for this example.

Techniques for coalescing two-level *triangular loops* are now given. In a triangular nested loop, the inner loop bounds are functions of the outer loop index variable.

2. The *div* and *mod* operations have been specified in the loop body for clarity. The quotient of the integer division represents the *div* result, the remainder the *mod* result.

Consider the following triangular loop structure:

```
DO 10 J = 1, N
   DO 20 K = 1, I
     ....
```

The index set for this loop with N=5 is given in figure one.

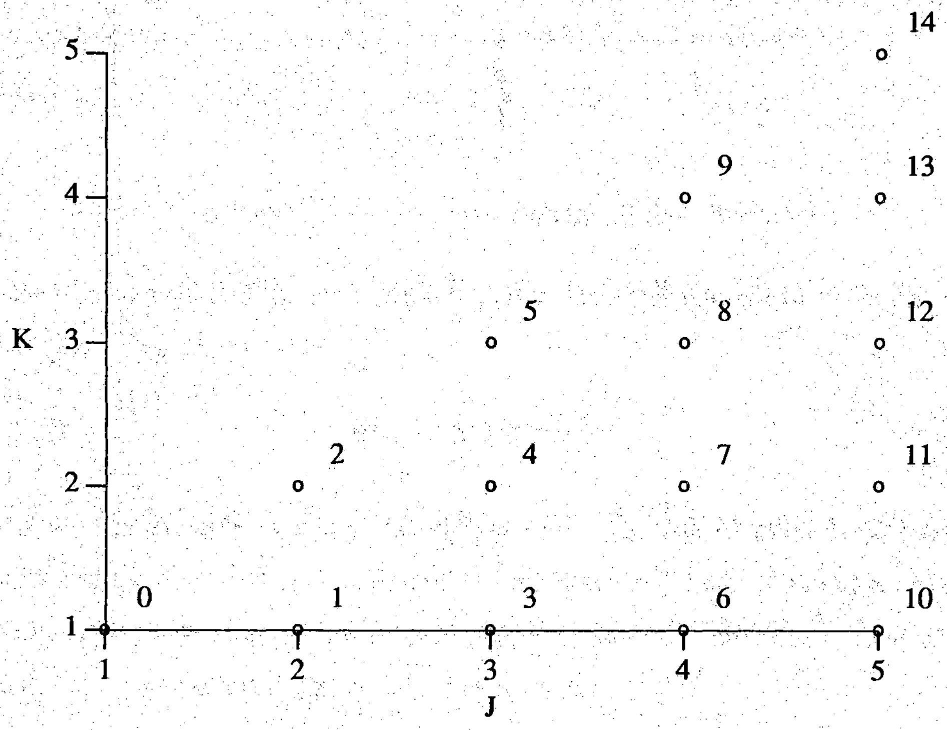

Figure 1: Example Triangular Loop with Serial Execution Order.

The iterations are labeled with their serial execution order. The total number of instances, $\tau(N)$, in the coalesced loop, is given by the expression

$$\tau(N) = \sum_{J=1}^{N} \lambda(J,N)$$

where $\lambda(J,N) = J$ for this example, which yields $\tau(N) = N(N+1)/2$. The function $\lambda(J,N)$ represents the number of iterations of the inner K loop as a function of the outer loop index J and upper bound N. Normalized inner loops have lower loop bounds and increments equal to one and $\lambda(J,N)$ reduces to the loop upper bound. In the general case, the function $\lambda(J,N)$ is given as

$$\lambda(J,N) = (\, ub(J,N) - lb(J,N) + 1) \; div \; inc(J,N)$$

where $ub(J,N)$, $lb(J,N)$, and $inc(J,N)$ are the upper bound, lower bound, and increment functions, respectively, for the inner loop.

For the example loop, the function $\tau(N) = N(N+1)/2$ is the number of iterations in the coalesced loop. The index variable for the coalesced loop will be JK, with a lower bound of 0 and upper bound of $\tau(N)-1$. The original loop indices J and K must be re-generated from the coalesced index.

Figures 2 and 3 show how J and K vary with the coalesced index JK.

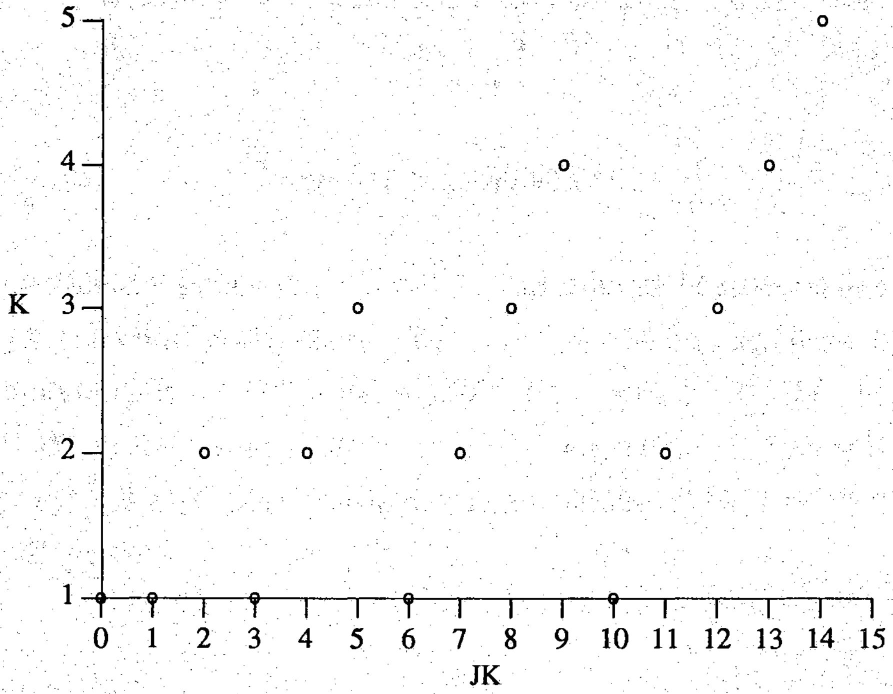

Figure 2: K as a Function of JK.

It can be seen that transitions occur at 0, 1, 3, 6 and 10. This series can be generated by a *transition function* $\iota(j)$, $0 \le j \le N-1$ where, in this example, $\iota(j) = j(j+1)/2$. The transition function can be used to determine the value of index J given coalesced index JK: $J = \min\{\, j : \iota(j) > JK \}$, i.e., the smallest j such that $\iota(j) > JK$. Hence, to determine J from JK, the function $\iota(j)$ must be computed for $j = 0,1,2,...$ until $\iota(j) > JK$. It is then straightforward to compute the inner loop index K; in the example of figure 1, $K = JK - \iota(j-1) + 1$.

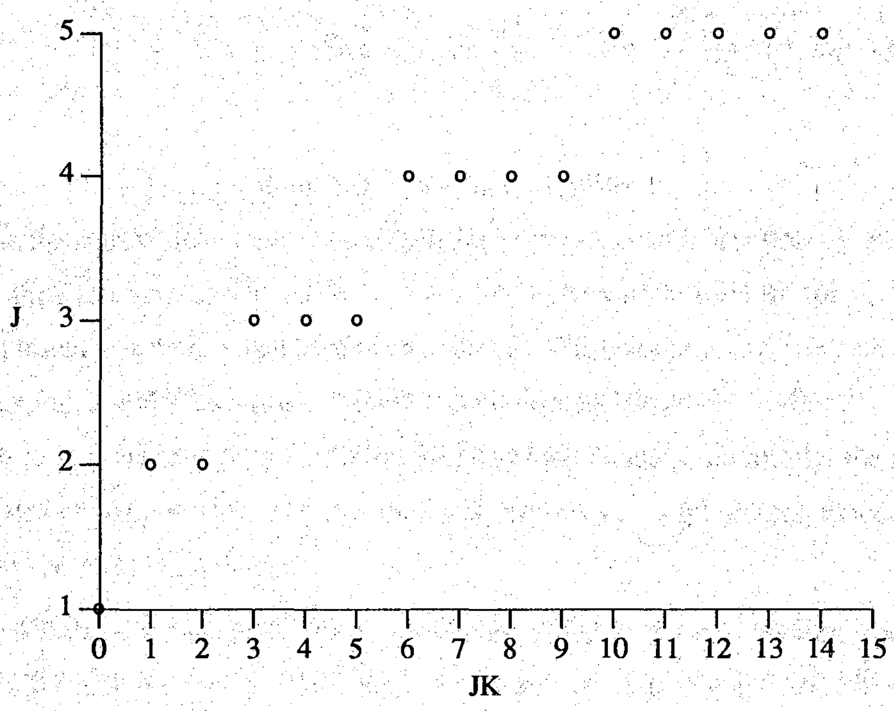

Figure 3: J as a Function of JK.

To execute the coalesced loop on a barrier MIMD, each processor independently computes the transition function for successive *j* until ι(*j*)>*JK*, where *JK* is the current instance for the processor. This gives `J` for the instance, which is then used to compute `K`. The body of the loop is executed using these generated values for `I` and `J`. The cost of these operations depends on the complexity of the transition function, which in turn depends on the form of the inner loop bounds. Alternately, the transition series could be generated at compile-time, and saved in local memory in the processors, reducing the run-time overhead at the expense of extra storage.

To generalize the approach given above for doubly-nested loops, it is necessary to determine the proper transition function for general loop bounds. In the general case, doubly-nested loops have the following form:

```
DO 10 I = M, N, P
   DO 20 J = lb(I,M,N) , ub(I,M,N), inc(J,M,N)
      ....
```

Figure 4: General Form for Doubly-Nested Loops.

The upper and lower bounds and increment of the inner loop are functions of the outer loop index and bounds. Note that the outer loop bounds and increment `M`, `N`, and `P` can be integer expressions. Loop normalization could be employed to transform the loop increment and lower bounds to one to simplify the general-form loop structure, but this increases the complexity of subscript expressions[3]. It is not used in this work.

The transition function for $\iota(j)$ for the general form doubly-nested loops is

$$\iota(j) = \begin{cases} \sum_{J=M}^{j} \lambda(J,M,N) & M \le j \le N-1 \\ 0 & j < M \end{cases}$$

where

$$\lambda(J,M,N) = \begin{cases} (ub(J,M,N) - lb(J,M,N) + 1) \ \mathbf{div}\ inc(J,M,N) & \text{if } inc(J,M,N) > 0 \\ (lb(J,M,N) - ub(J,M,N) + 1) \ \mathbf{div}\ inc(J,M,N) & \text{if } inc(J,M,N) < 0 \end{cases} .$$

The number of instances in the coalesced loop τ is then given as

$$\tau(M,N) = \iota(N)$$

A closed-form expression for $\iota(j)$ is required, and this will sometimes require manipulation of the summation. This was not the case for the example in figure one, where

$$\iota(j) = \begin{cases} \sum_{J=1}^{j} J = j(j+1)/2 & 1 \le j \le N \\ 0 & j < 1 \end{cases}$$

is a well-known form. As another example, consider the following loop structure:

```
DO 10 J = 1, N
   DO 20 K = 1, N-J+1
         ....
```

Figure 5: Example of a General Form Doubly-Nested Loop.

3. Wolfe [Wol86] recently observed that loop normalization can adversely affect the complexity of transforming loops since it typically increases the complexity of the array subscript expressions, and it can sometimes prevent a useful transformation by changing the direction vectors for a given loop nest.

In this case,

$$\iota(j) = \begin{cases} \sum_{J=1}^{j} N-J+1 & 1 \le j \le N-1 \\ 0 & j < 1 \end{cases} .$$

which can be reduced to

$$\iota(j) = \sum_{J=1}^{j} N - \sum_{J=1}^{j} J + \sum_{J=1}^{j} 1 = j(2N-j+1)/2 \quad , 1 \le j \le N-1 .$$

For the general loop form, once `J` is computed from the transition function, `K` is determined from the expression

$$K = JK - \iota(j-1) + lb(J,M,N) \quad .$$

As a more complex example, consider the loop of figure 6, which is part of Trench's algorithm for determining the inverse of a Toeplitz matrix [GoV83][4]:

```
        DO 10 J = 2, (N-1)/2 + 1
           DO 20 K = J, N - J + 1
              B(J,K) = B(J-1,K-1) +
       +                 (v[N+1-K]*v[N+1-J] - v[J-1]*v[K-1])/GAMMA;
   10      CONTINUE
   20   CONTINUE
```

Figure 6: Doubly-nested Loop Taken from Trench's Algorithm.

The iteration space for this loop (with $N=10$) is given in figure 7.

The transition function is derived as follows:

$$\iota(j) = \begin{cases} \sum_{J=2}^{j} (N-2J+2) & 2 \le j \le ((N-1)/2+1) \\ 0 & j<2 \end{cases} .$$

Distributing the summation

4. Proper synchronization for the coalesced form of this loop is considered in the next section.

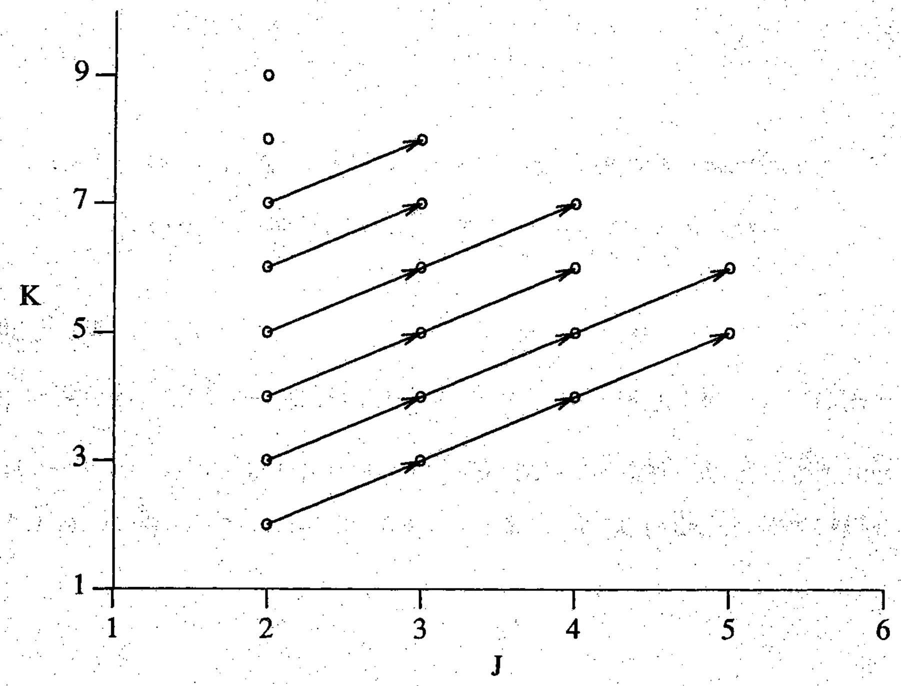

Figure 7: Iteration Space for Loop Nest from Trench's Algorithm (N=10).

$$\iota(j) = \sum_{J=2}^{j} N - \sum_{J=2}^{j} 2J + \sum_{J=2}^{j} 2 \quad , \; 2 \le j \le ((N-1)/2+1)$$

and simplifying yields

$$\iota(j) = N(j-1) - 2(j(j+1)/2-1) + 2(j-1) = -j^2 + (N+1)j - N \quad , \quad 2 \le j \le (N-1)/2 \; .$$

Table 1 shows how the coalesced loop iterations for Trench's algorithm are spread across a four-processor barrier MIMD. The original indices for the loop are given in parentheses (J,K) next to the coalesced index.

PE 0	PE 1	PE 2	PE 3
0 (2,2)	1 (2,3)	2 (2,4)	3 (2,5)
4 (2,6)	5 (2,7)	6 (2,8)	7 (2,9)
8 (3,3)	9 (3,4)	10 (3,5)	11 (3,6)
12 (3,7)	13 (3,8)	14 (4,4)	15 (4,5)
16 (4,6)	17 (4,7)	18 (5,5)	19 (5,6)

Table 1: Processor Assignment for Trench Loop Nest (4 processors)

The approach for coalescing doubly-nested triangular loops can be applied successively to coalesce multiply-nested loops. This procedure begins with the innermost loops and continues outward; for example, with a triply-nested loop, the two inner loops are coalesced, followed by the outermost loop and the new coalesced inner loop. The following multiply-nested rectangular loop will be coalesced with the triangular loop coalescing techniques. This will allow a comparison between the previous loop coalescing techniques for rectangular loops and the new, general approach described in this work.

```
DO 10 I = 1, IMAX
    DO 20 J = 1, JMAX
        DO 30 K = 1, KMAX
            ....
```

Coalescing the J and K loops yields the transition function

$$\iota_{JK}(j) = \sum_{J=1}^{j} KMAX = j*KMAX$$

and the loops now have the form

```
DO 10 I = 1, IMAX
    DO 20 JK = 0, JMAX*KMAX-1
        J = min { j : j*KMAX > JK }
        K = JK - (j-1)*KMAX +1
        ....
```

Coalescing these two loops gives the transition function

$$\iota_{IJK}(i) = \sum_{l=1}^{i} (JMAX*KMAX) = i(JMAX*KMAX)$$

and the completely coalesced loop is

```
DO 10 IJK = 0, IMAX*JMAX*KMAX-1
   I  = min { i : i*(JMAX*KMAX) > IJK }
   JK = IJK - (i-1)*(JMAX*KMAX)
   J  = min { j : j*KMAX > JK }
   K  = JK - (j-1)*KMAX +1
   ....
```

Unlike the other rectangular loop coalescing techniques, the new approach does not use integer division. In the best case the `I` and `J` computations require a single compare operation each, and `JK` and `K` computations require two integer multiplies, two subtractions, and one addition. However, on average the `I` and `J` computations will require that *i* and *j* be incremented some average amount until the inequality is satisfied.

The best approach will depend on the availability of integer division in hardware and the relative speed of integer division and multiplication, as well as the average increment per iteration in the triangular approach. Recent processor architecture designs have reduced the amount of hardware support for relatively infrequent operations such as division, and software support routines for integer division are slow. One study found that a general purpose divide routine averaged 80 cycles per divide operation [MaP88].

Notice that the need for multiplies and divides to compute indices for each iteration can in general be eliminated by using consecutive allocation (mentioned in section 2) and replicating the original looping control structure in the code for each process. This is discussed further in [Pol88]. We stress the other techniques because they efficiently support arbitrary allocations (including consecutive allocation), however, when consecutive allocation is appropriate, the use of the original looping structure may be preferable.

4. Loop Scheduling and Synchronization on Barrier MIMD Architectures

In the previous section, a generalized technique for coalescing loops was described. In this section loop coalescing is considered for static, decentralized scheduling of barrier MIMD architectures. The approach taken will be similar to that for the FMP, except the compiler will automatically construct the domain for a set of nested loops after the appropriate analysis has been performed, and the domains are not restricted to rectangular shapes. In addition, the instances of the coalesced loop may be synchronized as necessary by a barrier, so coalescing is not restricted to loops without dependencies. Loop coalescing simplifies loop scheduling since the single dimension of the coalesced iteration space can be allocated evenly among the processors with small scheduling overhead.

The basic properties of barrier MIMD architectures were mentioned in the introduction. They include a fast hardware barrier synchronization mechanism that can be applied across any subset of the processors. A *barrier processor* generates the proper sequence of barrier masks to insure correct sequencing and proper timing relationships between computational processors. It places the barriers in a *barrier synchronization buffer* where they are matched against processors waiting at a barrier, and then executed. A single WAIT line from each processor to the synchronization buffer is used to indicate that a particular processor is participating in a barrier synchronization. Thus, when scheduling a loop it is necessary to generate code for the computational processors to request a barrier and for the barrier processor to generate the proper barrier masks in the correct order.

In addition, before execution of a coalesced loop on a barrier MIMD, the barrier processor must broadcast the number of iterations in the coalesced loop and the number of processors executing the loop. Loop iterations in the coalesced index set are assigned to the computational processors using interleaved allocation, as in the FMP. This binding occurs at compile-time between loop iterations and a virtual barrier MIMD machine; the binding between the virtual and actual barrier MIMD machine occurs at run-time when the barrier processor broadcasts the number of iterations in the loop and the number of processors in the actual machine.[5]

Data dependencies [ShF88], [Wol89] between loop iterations must be considered during coalescing, and if such dependencies do exist then the resulting coalesced loop may require barrier synchronization. If no dependencies exist between iterations, then no synchronization is required and processors proceed to

5. This approach also allows the machine to be partitioned so that independent loops (or programs) may be executing simultaneously on different parts of the machine.

asynchronously execute the coalesced loop until all iterations are computed. At this point, all processors barrier synchronize before continuing execution.

Loop transformations can be used to restructure a loop nest to provide different coalescing results [Pol88]. For example, loop interchanging [Wol89] can be applied as necessary to move parallel loops to the innermost nest levels [Pol88]. Alternately, serial loops could be moved into the innermost levels, with outer parallel loops coalesced around them. Loop distribution [KuM72], [Wol89] can also be employed to transform loops into perfectly-nested form for coalescing. The best loop structure for coalescing depends on several factors, including the necessity of balancing work among processors to exploit as much parallelism as possible, the data dependence structure of the nested loops, and run-time constraints such as data locality. One major difference between barrier MIMDs and other MIMD machines is that barrier synchronization is very fast and efficient; also, the static nature of scheduling the machine makes large variations in processor execution times unlikely.

The order of loops before coalescing directly affects the allocation of loop iterations across the processors as well as the number of barriers generated. Proper execution on a barrier MIMD imposes certain constraints on this ordering. In particular, the innermost coalesced loops must not have any dependencies across loop iterations. In this work, only the outermost coalesced loop is allowed to have dependencies: the dependencies across this loop may require barriers for correct execution.

For example, consider the loop nest from Trench's algorithm (figure 7): a dependence exists between iterations (J,K) and $(J+1,K+1)$, which will be represented as the dependence vector $\bar{d} = [1 \quad 1]$ [ShF88]. This dependence vector can be seen in figure 7. From the figure, it is clear that all iterations with $J = b$, where b is a constant, can be executed in parallel, i.e., the K loop may be executed in parallel; the J loop is executed serially, and barriers can be used to enforce this ordering. The basic idea is to determine a schedule $\sigma(J,K) = \pi(J,K)+c$ that is a linear function of the loop indices so that iterations executing in parallel have $\sigma(J,K) = d$, where d is a constant.

The difference between schedule values for consecutive iterations executed on a single processor determines how many barriers that processor should execute. Table 2 shows how this approach generates barriers to enforce the proper execution order for the loop nest from Trench's algorithm. Barriers are represented as horizontal lines in the table. The linear schedule for this example is $\sigma(J,K) = J-2$.

Schedules that are linear functions of the loop indices are referred to as *linear schedules* [ShF88]. These schedules are related to the well-known *wavefront method* [KuM72], [Kuh80] but are generalized

PE 0	PE 1	PE 2	PE 3
0 (2,2) 4 (2,6)	1 (2,3) 5 (2,7)	2 (2,4) 6 (2,8)	3 (2,5) 7 (2,9)
8 (3,3) 12 (3,7)	9 (3,4) 13 (3,8)	10 (3,5) -	11 (3,6) -
16 (4,6)	17 (4,7)	14 (4,4)	15 (4,5)
18 (5,5)	19 (5,6)	-	-

Table 2: Proper Execution Ordering Enforced by Barriers.

in the sense that coefficients of the linear schedule function are not restricted to integers and may be rational numbers [ShF88]. It will be shown in this work that the wavefronts (called *hyperplanes* in [ShF88]) in a linear schedule can be implemented directly by barriers. This work will be concerned primarily with *simple linear schedules* that are functions of a single index variable although more general linear schedules are briefly considered. The schedule proposed for the loop nest of Trench's algorithm was a simple linear schedule. The wavefronts generated by this schedule can be seen in figure 8.

Linear schedules have many advantages. In a classic paper [KaM67], Karp, Miller and Winograd proved that, under certain conditions (uniform data dependencies and unit-time computations) the execution time of an optimal linear schedule and the *free* or dataflow schedule, which executes a computation as soon as its operands are available, is bounded by a constant[6]. Hence, a good linear schedule should be able to exploit most of the parallelism within a loop (or set of nested loops). Simple linear schedules have a straightforward interpretation in terms of nested loops. The outermost loop corresponds to the wavefront direction; the simple linear schedule is a function of the outermost loop index, as in the example loop nest from Trench's algorithm. The barriers that enforce the wavefront order are, in effect, enforcing the serial order inherent in the outermost loop.

6. Shang and Fortes [ShF88] sharpened this result by providing sufficient conditions for the schedules to have equal execution time.

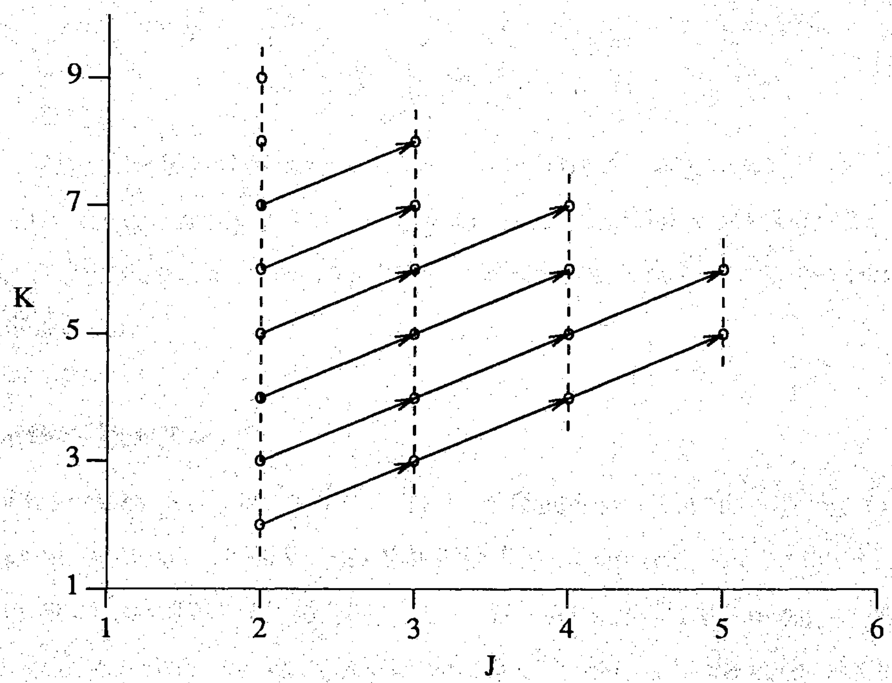

Figure 8: Wavefronts for Loop Nest from Trench's Algorithm (N=10).

The algorithm for generating barriers for simple linear schedules is now described. Each computational processor executes this algorithm to generate a proper sequence of barriers to correctly implement the simple linear schedule.

Algorithm: Barrier Generation

The *wavefront index* ω is generated from linear schedule function $\sigma(\bar{J})$, where $\bar{J} = (J_1, J_2, ..., J_n)$ are the n indices of the original nested loops that have been coalesced. The wavefront index represents the wavefront in which iteration $(\bar{J})$ is executed. Let p be the processor id number, P the number of processors executing the schedule, and let N be the number of iterations in the coalesced loop. I represents the current iteration being executed by processor p. The procedure is:

[1] [Initialize.] $\omega_o \leftarrow 0, I \leftarrow p, done \leftarrow FALSE.$

[2] [Generate indices.] Compute the indices $\bar{J}$ from the coalesced index I. (As described in the previous section.)

[3] [Calculate wavefront index.] $\omega \leftarrow \sigma(\bar{J})$, $\beta \leftarrow \omega - \omega_o$. (β gives the number of barriers before execution of iteration I.)

[4] [Generate barriers.] Execute β barrier waits before executing iteration I.

[5] [Check for completion.] If $done = TRUE$, execute one more barrier and then terminate the algorithm.

[6] [Set up for next iteration.] $\omega_0 \leftarrow \omega$, $I \leftarrow I+P$.

[7] [Check for last iteration.] if $I \geq N$, then $I \leftarrow N-1$, $done \leftarrow TRUE$.

[8] Go to [2].

The barrier processor must generate $\sigma(J_{\max}) - \sigma(J_{\min})$ barrier masks, where $J_{\max}$ and $J_{\min}$ are the maximum and minimum points for the linear schedule σ. Note that in this algorithm, the barrier mask includes all P processors, so the capability to barrier synchronize subsets of the processors is unused. More sophisticated algorithms could be developed to avoid this. In step [6], the next iteration to execute is obtained as $I \leftarrow I+P$, yielding and interleaved allocation of iterations. Consecutive allocation, as used in guided self-scheduling, is also possible with minor modifications of the algorithm.

Several examples of loop scheduling for barrier MIMDs will now be given to clarify and expand the ideas in this section. The first example code, given in figure 9, solves a lower triangular system of equations using forward elimination [GOV83].

```
          DO 10 I = 1, N
100          Y(I) = B(I)
             DO 20 J = 1, I-1
200             Y(I) = Y(I) - L(I,J)*Y(J)
20           CONTINUE
300          Y(I) = Y(I)/L(I,I)
10        CONTINUE
```

Figure 9: Forward Elimination for Triangular System Solution.

Statement `100` can be distributed out of the `I` loop, and the `I` and `J` loops interchanged, bringing the parallel loop `I` into the innermost nesting level. The restructured code is given in Figure 10:

```
           DO 10 I1 = 1, N
100            Y(I1) = B(I1)
10         CONTINUE
           DO 20 J = 1, N
300            Y(J) = Y(J)/L(J,J)
               DO I = J+1, N
200                Y(I) = Y(I) - L(I,J)*Y(J)
30             CONTINUE
20         CONTINUE
```

Figure 10: Restructured Loop before Coalescing.

The `I1` loop may be executed in parallel, with a barrier separating loops `I1` and `J`. Loops `J` and `I` can now be coalesced. Statement `300` can be moved into the inner `I` loop at the cost of computational redundancy; alternately, this statement could be executed conditionally within the inner loop, depending on the generated index values [Pol88]. The right approach depends partly upon the ability of the machine to quickly broadcast values from one processor to all others; if this capability is missing, then it may pay to compute the value locally in each processor.

The transition function for the coalesced loop is

$$\iota(j) = \begin{cases} \sum_{J=1}^{j} N-J & 1 \le j < N \\ 0 & j < 1 \end{cases}$$

which reduces to $\iota(j) = jN - j(j+1)/2$, $1 \le j < N$. The original indices can be generated from the coalesced index as follows:

$$J = \min \{ j : \iota(j) > JI \}$$

and

$$I = JI - \iota(j-1) + (J+1) \quad .$$

This transition function is rather complex, but there are several approaches to reducing the overhead in computing it. One obvious solution is to have an independent integer function unit dedicated to computing the transition function in parallel with the loop body computations. Note that the transition function can be computed in advance for increasing values of J as it is independent of the loop body. This "look-ahead" approach to computing the transition function could also be used to fill gaps in computation while a processor is waiting to barrier synchronize with other processors. Thus, it appears that the transition function overhead can be masked quite effectively.

Coalescing loops J and I yields

```
          DO 10 I1 = 1, N
100          Y(I1) = B(I1)
10        CONTINUE
          DO 20 JI = 1, N
             J = min { j : ι(j)  > JI }
             I = JI - ι(j-1) + (J+1)
200          Y(I) = Y(I) - L(I,J)*Y(J)
             IF ( I .EQ. J+1 ) Y(I) = Y(I)/L(I,I)
20        CONTINUE
```

Figure 11: Restructured Loop after Coalescing.

The simple linear schedule for this coalesced loop is $\sigma(\bar{J}) = J-1$. The barriers enforce the proper ordering between successive column computations. Row computations are executed in parallel depending on the number of processors allocated at the end of the loop. Notice how the parallelism width of the forward elimination algorithm decreases monotonically as the algorithm moves down the columns of the matrix L. This is quite common for such triangular loop structures. The barrier processor can *tune* the processor allocation for the coalesced loop by separating the computation into phases: as the parallelism width goes down (or up) for each phase, fewer (or more) processors can be allocated by the barrier processor for the current phase.

The next example considered is Gaussian elimination [GoV83]. The innermost loops for Gaussian elimination, labeled 20 and 30 in figure 12, can be coalesced and scheduled effectively on a barrier MIMD.

```
      DO 40 K = 1, N-1
C
C        code for partial (or complete) pivoting elided
C
         DO 10  P = K+1, N
            W(P) = A(K,P)
 10      CONTINUE
         DO 30  I = K+1, N
 14         COEF = A(I,K)/A(K,K)
 16         A(I,K) = COEF
            DO 20  J = K+1, N
               A(I,J) = A(I,J) - COEF*W(J)
 20         CONTINUE
 30      CONTINUE
 40   CONTINUE
```

Figure 12: Original Code for Gaussian Elimination [GoV83].

Statements `14` and `16` can be distributed out the `I` loop; since the range for the resulting loop matches that of the `DO` loop labeled `10` and no dependencies exist between these loops, they can be fused [Wol89]. The resulting code is shown in figure 13.

```
      DO 40 K = 1, N-1
C
C        code for partial (or complete) pivoting elided
C
         DO 10  P = K+1, N
            W(P) = A(K,P)
 14         COEF = A(P,K)/A(K,K)
 16         A(P,K) = COEF
 10      CONTINUE
         DO 30  I = K+1, N
            DO 20  J = K+1, N
               A(I,J) = A(I,J) - COEF*W(J)
 20         CONTINUE
 30      CONTINUE
 40   CONTINUE
```

Figure 13: Restructured Code for Gaussian Elimination.

The `P` loop may be executed in parallel; of course, since it is a single loop, no coalescing is necessary. The restructured code after coalescing inner loops `I` and `J` into index `IJ` is shown in figure 14[7]. A barrier is required between the `P` loop and the coalesced `IJ` loop, and after the `IJ` loop to enforce the proper ordering inherent in the outer loop, which is executed serially. Clearly, $\tau(N,K) = (N-K)^2$ and the functions to generate the original indices from `IJ` are

$$I = (IJ\ div\ (N-K)) + (K+1)$$

and

$$J = (IJ\ mod\ (N-K)) + (K+1) \quad .$$

Since both the `I` and `J` loops may be executed in parallel, there is no need to generate barriers to enforce a proper ordering between iterations of the coalesced loop. The code for partial or complete pivoting, if it were included in the example, could be parallelized like the `P` loop. As with the forward

7. In this rectangular loop example, the transition function has been replaced by *div* and *mod* operations.

elimination example, the barrier processor could tune the processor allocation to adapt to the monotonically decreasing parallelism as K increases.

```
      DO 40 K = 1, N-1
C
C        code for partial (or complete) pivoting elided
C
         DO 10  P = K+1, N
            W(P)  = A(K,P)
 14         COEF  = A(P,K)/A(K,K)
 16         A(P,K)  = COEF
 10      CONTINUE
         DO 30  IJ = 0, (N-K)**2 - 1
            I = (IJ div (N-K)) + (K+1)
            J = (IJ mod (N-K)) + (K+1)
            A(I,J) = A(I,J) - COEF*W(J)
 30      CONTINUE
 40   CONTINUE
```

Figure 14: Restructured Code for Gaussian Elimination.

The next example considers executing a *non-simple* linear schedule on a barrier MIMD. This and the following example show how linear schedules can exploit the maximum parallelism in nested loop structures by considering all loops simultaneously. The loop given in figure 15 implements a four-point difference problem: notice that the dependencies, shown in figure 16, preclude parallelizing either the I or J loop directly. This example is taken from [Wol89].

```
          DO 20  I = 2, N-1
              DO 10  J = 2, N-1
                  A(I,J) = (A(I-1,J) + A(I+1,J)
     +                     + A(I,J-1) + A(I,J+1))/4
10            CONTINUE
20        CONTINUE
```

Figure 15: Four-point difference problem.

The wavefront technique [KuM72], [Kuh80] was originally developed to exploit the parallelism in such loops. These loops can be coalesced and executed with a linear schedule. Loop structures such as the four-point difference problem show that parallelism in some nested loops is not inherent in one or the other loop, but can be extracted by considering both loops *simultaneously*. This simultaneous approach is natural when loop coalescing is combined with linear schedules.

A valid linear schedule for executing the loops in figure 15 is $\sigma(I,J) = I+J-4$; the wavefronts for this schedule correspond to the dashed lines in figure 16. Note that this is not a simple linear schedule, since σ is a function of both `I` and `J`. However, in this example, the coalesced loop can be executed on a barrier MIMD if the number of processors allocated is equal to $N-2$. The barrier generation algorithm will still work properly for some linear schedules if the number of processors is restricted to the range of the innermost coalesced loop. The exact conditions when it may still be applied is a current research problem. Table 3 shows the allocation of the iterations of the coalesced loop for four processors (N=6).

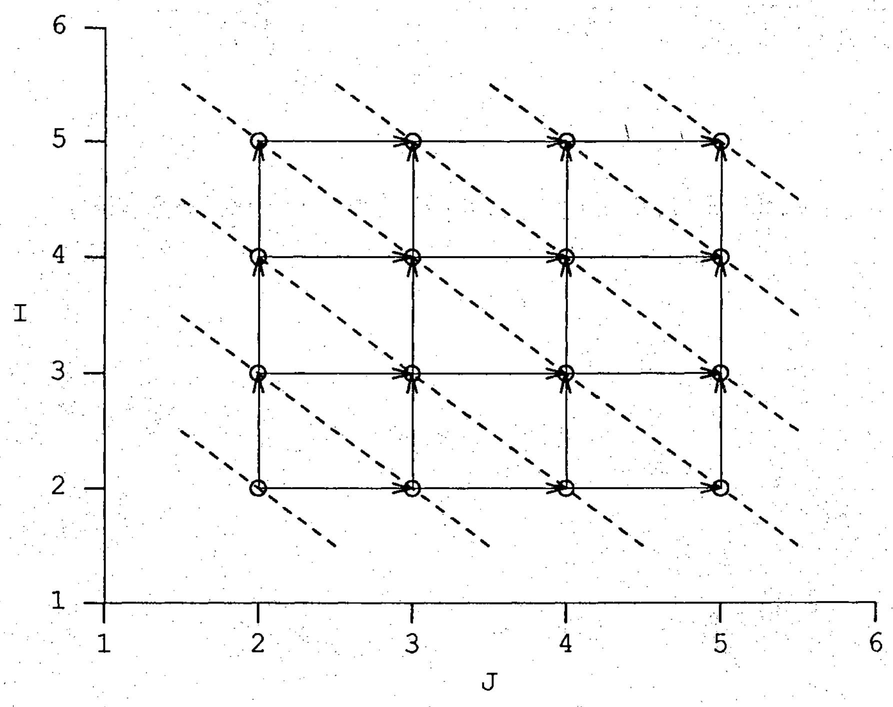

Figure 16: Index Space and Wavefronts for 4-Point Problem (N=6).

PE 0	PE 1	PE 2	PE 3
0 (2,2)	-	-	-
4 (2,3)	1 (3,2)	-	-
8 (2,4)	5 (3,3)	2 (4,2)	-
12 (2,5)	9 (3,4)	6 (4,3)	3 (5,2)
-	13 (3,5)	10 (4,4)	7 (5,3)
-	-	14 (4,5)	11 (5,4)
-	-	-	15 (5,5)

Table 3: Proper Execution Ordering Enforced by Barriers.

Another example of the interaction between loops that affects the amount of exploitable parallelism can be seen in the loop nest of figure 17. This loop nest also provides an example of a rational linear schedule [ShF88], [ShO90].

```
      DO 20  I = 1, N
        DO 10 J = 1, M
          A(I,J) = A(I-2, J) + B(I,J)*C(I) + D(J,I)
10      CONTINUE
20    CONTINUE
```

Figure 17: Loop Nest with a Rational Linear Schedule.

The `J` loop can be executed in parallel, but a dependence along `I` loop prevents parallelization; notice, however, that the dependence distance in the `I` loop is 2. This means that two wavefronts along the `J` loop can be executed in parallel: this can be realized with the rational linear schedule $\sigma(J,K) = (I-1)/2$. The loop is coalesced in the normal manner and the resulting schedule is, in fact, optimal in terms of execution time. In this example, a rational linear schedule yields twice as much parallelism compared to parallelizing the `J` loop alone. A detailed discussion of issues related to optimal linear schedules and loop scheduling can be found in [ShO90].

The following example provides insight on a subtle problem in exploiting loop parallelism and how loop coalescing can help solve the problem. The innermost loop nest for Cholesky decomposition ([GoV83], pp. 89) is shown in figure 18.

```
      DO 40 K = 1,N
         temp = 0.0
         DO 10 P = 1,K-1
10          temp =  temp + G(K,P)*G(K,P)
         G(K,K) = sqrt(A(K,K) - temp)

         DO 30 I = K+1,N
            temp = 0.0
            DO 20 P = 1,K-1
20             temp = temp + G(I,P)*G(K,P)
30          G(I,K) = (A(I,K) - temp)/G(K,K)
40    CONTINUE
```

Figure 18: Code for Cholesky Decomposition [GoV83].

Notice how the the loop limits for the inner `I` and `P` loops (labeled `20` and `30`) vary with `K`. For small values of `K`, most of the parallelism resides in the `I` loop since the `P` loop range is small; however, the situation changes as `K` approaches `N`, where the `I` loop range becomes small, and the `P` range large. Parallelism exists in both loops[8] but shifts from the `I` loop to the `P` loop as `K` moves through its range. Since loop interchange is possible, it is difficult to decide which loop should be parallelized for a machine that supports a single level of loop parallelism. If coalescing is applied to these loops the parallelism inherent in the loop structure can be exploited more effectively, since it would be inherent in the single coalesced loop.

As another example of the difficulty in effective loop parallelization, consider again the loop nest from Trench's algorithm, given in figure 6. Now assume that, instead of the loop bounds given in the

8. The parallelism in the `P` loop must be realized through and an associative reduction [Wol89].

figure, the bounds for `J` are 1 .. *N* and for `K` are 1 .. *M*. Given the dependence structure, a linear schedule can exploit parallelism in one or the other loop; the loop with maximum parallelism depends on the relative values of `M` and `N`[9]. The appropriate test can be executed at run-time to determine which loop should be parallelized: with loop coalescing, the *div* and *mod* parameter can be a variable set according to the results of this test. The result is a very efficient technique to statically generate the proper run-time test to exploit the maximum parallelism possible.

5. Conclusions

In this work, loop coalescing has been extended to apply to triangular nested loops. A new approach has been proposed for coalescing rectangular loops that is more efficient than current techniques. The new loop coalescing techniques, combined with some familiar loop transformations for parallelization, have been applied to the problem of scheduling nested loop structures on barrier MIMD architectures. Simple linear schedules have been shown to be an effective paradigm for efficiently exploiting the parallelism in nested loops. These schedules can also quite easily take advantage of parallelism that is inherent in the interaction between nested loops. Loop coalescing also has advantages in parallelizing loop structures where the parallelism shifts from one loop to another during execution, and where simple tests at run-time can determine the best loop to parallelize.

Future research effort include extending the barrier generation algorithm so that it can be applied to linear schedules in general. Current work also includes a prototype compiler that will implement several of the transformations described in this work.

9. The analysis necessary to determine such tests in the general case is given in [ShO90].

References

[DiS88]

H. G. Dietz and T. Schwederski, "Extending Static Synchronization Beyond SIMD and VLIW," *Tech. Report TR-EE 88-25,* Purdue University, School of Electrical Engineering, June 1988.

[DoM79]

J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide,* SIAM: Philadelphia, 1979.

[GoV83]

G. H. Golub and C. F. Van Loan. *Matrix Computations,* Johns Hopkins University: Baltimore, 1983.

[KaM67]

R. M. Karp, R. E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM,* Vol. 14, No. 3, pp. 563-590, July 1967.

[KrW85]

C. P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Trans. Software Eng.,* vol. SE-11, no. 10, pp. 1001-1016, October 1985.

[KuM72]

D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," *IEEE Trans. Comput.,* vol. C-21, no. 12, pp. 1293-1310.

[Kuh80]

R. H. Kuhn, *Optimization and Interconnection Complexity for Parallel Processors, Single-Stage Networks, and Decision Trees.* Ph.D. dissertation, Dept. of Comp. Science, U. of Illinois at Urbana-Champaign, February 1980.

[LuB80]

S. F. Lundstrom and G. H. Barnes. "A Controllable MIMD Architecture," *Proc. Int. Conf. on Parallel Processing,* pp. 19-27, 1980.

[Lun87]

S. F. Lundstrom, "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," *IEEE Trans. Comput.,* vol. C-36, no. 11, pp. 1292-1309, Nov. 1987.

[MaP88]

D. J. Magenheimer, L. Peters, K. W. Pettis, and D. Zuras, ''Integer Multiplication and Division on the HP Precision Architecture,'' *IEEE Trans. Comput.*, vol. C-37, no. 8, pp. 980-990, August 1988.

[OKD90]

M. T. O'Keefe and H. G. Dietz, ''Hardware Barrier Synchronization: Static Barrier MIMD,'' to appear, *1990 Int. Conf. on Parallel Processing*, St. Charles, IL.

[Pol88]

C. D. Polychronopolous, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Boston, 1988.

[ShF88]

W. Shang and J.A.B. Fortes, ''Time Optimal Linear Schedules for Algorithms with Uniform Dependencies,'' *Proceedings of Int'l Conf. on Systolic Arrays*, May 1988, pp. 393-402 (also to appear in *IEEE Trans. on Computers*).

[ShO90]

W. Shang, M. T. O'Keefe, and J. A.B. Fortes, ''On Optimal, Generalized Cycle Shrinking,'' *Technical Report*, School of Electrical Engineering, Purdue University, in preparation (May 1990).

[Wol86]

M. J. Wolfe, ''Loop Skewing: The Wavefront Method Revisited,'' *Int. Jour. of Parallel Programming*, vol. 15, no. 4, 1986.

[Wol89]

M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press: Cambridge, MA, 1989.

[ZaD90]

A. Zaafrani, H. G. Dietz, and M. T. O'Keefe, ''Static Scheduling for Barrier MIMD Architectures,'' to appear, *1990 Int. Conf. on Parallel Processing*, St. Charles, IL.