Refined Fortran: An Update

David Klappholz
Xiangyun Kong
Apostolos D. Kallis

Department of Electrical Engineering and Computer Science
Stevens Institute of Technology
Castle Point Station
Hoboken, NJ 07030
201-420-5509
klapphol@cse.ogc.edu

Abstract

Refined Languages (Refined Fortran, Refined C, etc.) are extensions of their parent languages in which it is possible to express parallelism, but impossible to create races or deadlocks. Where strictly deterministic behavior is desired, multiple executions of a Refined Fortran program with the same input data can be guaranteed to either compute the same results or terminate with the same run-time errors regardless of differences in scheduling. Where asynchronous behavior is desired, freedom from races can be guaranteed. The Refined Languages approach achieves its goal by extending sequential imperative programming languages with data- (rather than control-) oriented constructs, and by viewing the expression of parallelism in data- (rather than control-) oriented terms. Earlier versions of Refined Fortran are discussed in [1]-[2]; the present work supersedes and extends work reported in these earlier publications.

1. Motivation

If automatic parallelism detection were *always* able to recognize opportunities for safe parallel execution, then parallel algorithms could be coded in sequential languages; no parallelism-oriented language constructs would be necessary.

The major reason that no parallelism detector finds *all* safe parallelism is the un-resolvability of some of the aliases created through the use of array subscripting and pointer dereferencing. Neither traditional flow analysis [3], nor *array subscript analysis* [4]-[6], nor *pointer analysis* [7] resolves all relevant aliases, so parallelism-oriented language extensions are required.

2. The Refined Language Approach

In a Refined Language the programmer specifies potential parallel execution of code segments $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ by writing them in such way as to indicate that there are no read/write, write/read, or write/write dependences between any two of $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$.

The Refined Language extensions are nothing more than constructs which enable the programmer to provide the necessary alias-resolution information to a parallelizing compiler in cases in which it is impossible to do so in the parent language. They are used when code contains references which the programmer *thinks* will never be aliases for one another, but which automatic parallelism detection techniques would be unable to resolve.

The Refined Language compiler does not, however, take the programmer's word for anything. Before it considers generating parallel code, the compiler verifies that if the relevant references were never aliases for one another, then parallel execution would indeed be safe. If so, and if it decides that parallel execution is cost effective, it generates parallel code containing run-time checks which result in fatal errors if, in fact, those references turn out to be aliases for one another.

In order to recognize the potential parallelism intended by the programmer, a Refined Language compiler performs the same code analysis as does a parallelism detector. In the case of Refined Language compiling, though, we refer to this analysis as *parallelism recognition* rather than parallelism detection.

The Refined Language programmer specifies *potential* parallel execution because the Refined Language compiler's back end performs cost/benefit analysis to determine which of the parallelism specified by the programmer is cost effective, and will, therefore, be exploited.

The major Refined Language constructs for writing synchronous, deterministic parallel code are the *PARTITION* statement and the *DISTINCT* statement. The bulk of our presentation of Refined Fortran will consist of a discussion of these two statements. Quite recently, Refined Languages have been extended to allow the writing of asynchronous parallel code through the use of the *atom* statement which is very briefly presented in [8].

Since Refined Fortran is still in the development stage, specific details, especially questions of syntax, can be expected to change. When the 8X standard is finalized, Refined Fortran constructs for dealing with pointers will be introduced. In the meantime, the Refined Language approach to dealing with pointers may be found in [8].

3. The *PARTITION* Statement

3.1 Fixed Number of Partition Elements

Consider the Fortran code for quicksort shown in Figure 3-1. The two recursive calls to *QUICKSORT* may safely be executed in parallel with one another, but no parallelism detector that we know of will determine that this is the case. Inter-procedural subscript analysis gives false positives which indicate that no parallelism is possible.

In Refined Fortran *QUICKSORT* might be written as in Figure 3-2. Since each of the two recursive calls to *QUICKSORT* writes elements of the array *A*, the programmer

```
      SUBROUTINE QUICKSORT(A, LOW, HIGH)
            INTEGER  A(1)
            INTEGER LOW
            INTEGER HIGH
            INTEGER TEMP
            INTEGER PIVOT
            INTEGER I
            INTEGER J
            PIVOT = A((LOW+HIGH)/2)
            J = HIGH
            I = LOW
10          IF (PIVOT .LT. A(J))  THEN
                  J = J - 1
                  GO TO 10
            ENDIF
20          IF (A(I) .LT. PIVOT)  THEN
                  I = I + 1
                  GO TO 20
            ENDIF
            IF (I .LE. J)  THEN
                  TEMP = A(I)
                  A(I) = A(J)
                  A(J) = TEMP
                  J = J - 1
                  I = I + 1
            ENDIF
            IF (I .LT. J)  GO TO 10
            IF (J .GT. LOW)  THEN
                  CALL QUICKSORT(A,LOW,J)
            ENDIF
            IF (HIGH .GT. I)  THEN
                  CALL  QUICKSORT(A,I,HIGH)
            ENDIF
            RETURN
      END
```

Quicksort
Figure 3-1

indicates that these are intended to be non-overlapping subarrays of *A*. This is accomplished through

- the use of the *PARTITION* statement, which dynamically gives new names *LOWER* and *UPPER* to non-overlapping subarrays of array *A*.
- rewriting the two recursive calls to *QUICKSORT* so that they reference *LOWER*, and *UPPER* respectively

The effect of

PARTITION A(K) (LOWER(K .LE. J), UPPER(K .GE. I))

is to:

- rename as *LOWER* the set of elements, *A(K)*, of array *A* for which the predicate $K <= J$ is true
- rename as *UPPER* the set of elements, *A(K)*, of array *A* for which the predicate $K <= J$ is false *and* the predicate $K \geq I$ is true

In a *PARTITION* statement, each predicate entails the conjunction of the negations of all previous predicates; the *partition elements* (sub-data-structures) defined by these predicates are, therefore, guaranteed to be non-overlapping; *the compiler need perform no work to verify this.*

Note that, in the code of Figure 3-2, a reference to *LOWER(I)* or *UPPER(I)* is a reference to *A(I)* -- legal if *A(I)* is within the defined bounds of *LOWER* or *UPPER* respectively, and illegal otherwise. *In general, a partition element name is simply an access-restricted alias for the partitioned data structure, and can be used anywhere that a reference to the data structure is valid.*

If the *PARTITION* statement were simply a programmer's assertion which the compiler were required to trust, then freedom from races would in no way be assured. *In a Refined Language this is emphatically not the case*; a *PARTITION* statement, in addition to specifying dynamic renaming of data, contains the information which the compiler needs to insert run-time checks to guarantee that access violations, should they occur, are flagged as fatal run-time errors.

In the case of *QUICKSORT*, the *PARTITION* statement indicates that the following run-time checks are necessary:

- each reference to an element of *LOWER*, say to *LOWER(K)*, must be preceded by a test for $K <= J$
- each reference to an element of *UPPER*, say to *UPPER(K)*, must be preceded by a test for $\neg\ (K <= J) \wedge (K \geq I)$

A failed check results in a fatal run-time error rather than a race. The text of the error message indicates exactly where a race could have occurred. As *QUICKSORT* is called recursively, the amount of required checking grows logarithmically. It is, however, quite simple to optimize these checks to at most a

```
      SUBROUTINE QUICKSORT(A, LOW, HIGH)
            INTEGER A(1)
            INTEGER LOW
            INTEGER HIGH
            INTEGER TEMP
            INTEGER PIVOT
            INTEGER I
            INTEGER J
            PARTITION ELEMENTS(LOWER, UPPER)
            PIVOT = A((LOW+HIGH)/2)
            J = HIGH
            I = LOW
10          IF (PIVOT .LT. A(J)) THEN
                  J = J - 1
                  GO TO 20
            ENDIF
20          IF (A(I) .LT. PIVOT) THEN
                  I = I + 1
                  GO TO 20
            ENDIF
            IF (I .LE. J) THEN
                  TEMP = A(I)
                  A(I) = A(J)
                  A(J) = TEMP
                  J = J - 1
                  I = I + 1
            ENDIF
            IF (I .LT. J) GO TO 10
            PARTITION A(K) (LOWER(K .LE. J), UPPER(K .GE. I))
            IF (J .GT. LOW) THEN
                  CALL QUICKSORT(LOWER,LOW,J)
            ENDIF
            IF (HIGH .GT. I) THEN
                  CALL QUICKSORT(UPPER,I,HIGH)
            ENDIF
            RETURN
      END
```

Parallel Quicksort in Refined Fortran
Figure 3-2

```
      SUBROUTINE GJRED(A,M,N,K)
         REAL A(M,N)
         DO 10 I = 1, M
            IF (I.EQ.K) GOTO 10
            DO 20 J = 1, N
               A(I,J) = A(I,J) - A(I,J)*(A(K,J)/A(K,K))
20          CONTINUE
10       CONTINUE
      END
```

Figure 3-3
Inner Loops of Gauss Jordan Elimination

single upper and lower bound check with no loss of determinacy.

Consider, as a second example, the code of Figure 3-3, which constitutes the inner loops of Gauss-Jordan elimination. The iterations of both the *DO 10* and the *DO 20* loop may safely be executed in parallel with one another. A parallelism detector would, however, have a relatively hard time detecting that this is the case since the GCD and Banerjee tests[4], [6] result in false dependences for both loops.

Most parallelism detectors use the GCD and Banerjee tests only to determine whether there is a dependence; they don't do the additional, time-intensive, work of computing the solutions of the dependence equations. Doing this extra work would reveal that the only dependence is in the case of $I = K$; knowing this, however, is still insufficient. A parallelism detector would still have to recognize that the earlier statement *IF (I.EQ.K) GOTO 10* rules out the case of $I = K$ as an actual dependence by eliminating the iteration for $I = K$.

A Refined Fortran version of the same parallel algorithm is shown in Figure 3-4.

```
      SUBROUTINE GJRED(A,M,N,K)
         PARTITION ELEMENTS(PIVROW, REST)
         REAL A(M,N)
         PARTITION A(I,J)(REST(I.NE.K), PIVROW)
         DO 10 I = 1, M
            IF (I.EQ.K) GOTO 10
            DO 20 J = 1, N
               REST(I,J) = REST(I,J) -
                  REST(I,J)*(PIVROW(K,J)/PIVROW(K,K))
20          CONTINUE
10       CONTINUE
      END
```

Refined Fortran Version of Gauss Jordan Elimination
Figure 3-4

```
            S = 1
            DO 10 I = 1, LG(N)
               S = 2*S
               DO 20 J = 1, N/S
                  X(J*S) = X(J*S) + X(J*S - (S/2))
20          CONTINUE
10       CONTINUE
```

Summing the Elements of an Array
Figure 3-5

```
      PARTITION ELEMENTS(SEGMENT[])
         :
         :
      S = 1
      DO 10 I = 1, LG(N)
         S = 2*S
         PARTITION X(K) SEGMENT[J]
            ((J - 1)*S .LT. K .AND. K .LE. J*S), J = 1, N/S)
         DO 20 J = 1, N/S
            SEGMENT[J](J*S) = SEGMENT[J](J*S)
                              + SEGMENT[J](J*S- (S/2))
20       CONTINUE
10    CONTINUE
```

Refined Fortran Code
for Summing the Elements of an Array
Figure 3-6

The programmer, "knowing" that there will not be a *K*-th iteration of the *DO 10* loop, tells the compiler as much in the *PARTITION* statement by indicating that a reference to *REST(K,J)* would be illegal for any value of J. (Alternatively, the programmer is telling the compiler that should there be a *K*-th iteration, the compiler is to declare a fatal error). The compiler, seeing that on the *(I,J)* iteration of the nest of loops:

- REST(I,J) is read
- REST(I,J) is written
- PIVROW is read -- where PIVROW does not overlap with REST

recognizes that all iterations of the two loops may be executed in parallel with one another.

3.2. Variable Number of Partition Elements

In the preceding section we dealt with the partitioning of arrays into fixed numbers of partition elements. In the present section we consider the case in which the data partitioning which justifies an algorithm's parallel execution involves a run-time-decided number of partition elements.

Consider the code of Figure 3-5 which sums the elements of an *N*-element one-dimensional array *X*, leaving the sum in *X(N)*: This code utilizes the standard log-time parallel algorithm for solving first-order linear recurrences, successively dividing the array into halves, quarters, eighths, etc.; after each division the individual segments may be processed in parallel with one another, i.e., the iterations of the *DO 20* loop may be executed in parallel with one another. As of the moment we are aware of a number of automatic vectorizers/parallelizers which do not detect the indicated

```
      SUBROUTINE GAUSS( A, M, N, IROW )
         REAL A(M,N)
         REAL X(M)
         INTEGER IROW(M)
         DO 5 I = 1, M
            IROW(I) = I
 5          CONTINUE
         DO 10 K = 1, M
            BIG = 0
            DO 20 I = K, M
               IF (ABS(A(IROW(I),K)).GT.BIG) THEN
                  BIG = ABS(A(IROW(I),K))
                  IBIG = I
               ENDIF
 20         CONTINUE
C           **********************
C           swap IROW(BIG) and IROW(K)
C           **********************
            ITEMP = IROW(IBIG)
            IROW(IBIG) = IROW(K)
            IROW(K) = ITEMP
C           **********************
            PIVOT = A(IROW(K),K)
            A(IROW(K),K) = 1.0
            DO 40 J = K+1, N
               A(IROW(K),J) = A(IROW(K),J) / PIVOT
 40         CONTINUE
            DO 50 I = K+1, M
               X(I) = A(IROW(I),K)
               A(IROW(I),K) = 0.0
               DO 60 J = K+1, N
                  A(IROW(I),J) = A(IROW(I),J) - X(I) * A(IROW(K),J)
 60            CONTINUE
 50         CONTINUE
 10      CONTINUE
      END
```

Gaussian Elimination with Partial Pivoting
Figure 4-1

parallelism, and unaware of any automatic vectorizer / parallelizer which does; the problem is that, without taking into account the way in which the value of *S* varies from iteration to iteration, the GCD and Banerjee tests report non-existent dependences. While a parallelism detector could *attempt* to trace the sets of possible values of *all* variables occurring in subscript expressions, possibly through the use of symbolic execution, it is not clear that the attempt would succeed sufficiently frequently to justify the apparently enormous compile-time overhead.

Figure 3-6 shows Refined Fortran code for the parallel array summation algorithm. On each iteration of the *DO 10* loop, array *X* is partitioned into the number of segments appropriate to that iteration through the use of an implied-do-loop style *PARTITION* statement. Since the *J*-th iteration of the *DO 20* loop references only *SEGMENT[J]*, the Refined Fortran compiler recognizes that all iterations may be executed in parallel with one another.

Note that, unlike the situation in the *QUICKSORT* code, the repeated partitioning of array *X* in the current code is not an instance of partition refinement. In *QUICKSORT* the initial partition is successively refined at each level of the recursion after the first because a partition element (*LOWER* or *UPPER*) is being partitioned. In the current code it is the entire array *X* which is being re-partitioned on each iteration of the *DO 10* loop rather than a partition element created on the previous iteration. In fact, each partitioning of array *X* is actually a coarsening of the previous partitioning. Run-time overhead can be optimized to at most a lower and upper bounds check per array reference.

4. The *DISTINCT* Statement

The attribute of memory locations which is of primary interest where parallelism is concerned is *disjointness*. If the members of a collection of memory locations are pairwise disjoint, then they may be processed in parallel with one another.

If we think in terms of the addresses of those memory locations rather than in terms of the locations themselves, then the attribute of primary interest becomes *distinctness*. If addresses $\alpha_1, \alpha_2, \ldots, \alpha_k$ are pairwise distinct, then the memory locations which they refer to are pairwise disjoint.

In a Refined Language the programmer is able to declare a set of values pairwise distinct within a defined scope. As with the partitioning of data, the declaration of distinctness is verifiable through a combination of static and run-time checks.

```
      SUBROUTINE GAUSS( A, M, N, IROW )
         REAL A(M,N)
         REAL X(M)
         INTEGER IROW(M)
         PARTITION ELEMENTS(PIVROW, REST)
         DISTINCT ROW_INDEX(IROW(I) = I, I = 1,M)
         DO 10 K = 1, M
            BIG = 0
            DO 20 I = K, M
               IF (ABS(A(IROW(I),K)).GT.BIG) THEN
                  BIG = ABS(A(IROW(I),K))
                  IBIG = I
               ENDIF
20          CONTINUE
            SWAP(IROW(IBIG), IROW(K))
            PIVOT = A(IROW(K),K)
            A(IROW(K),K) = 1.0
            DO 40 J = K+1, N
               A(IROW(K),J) = A(IROW(K),J) / PIVOT
40          CONTINUE
            PARTITION A(I,J)(REST(I.NE.IROW(K)),PIVROW)
            DO 50 I = K+1, M
               X(I) = REST(IROW(I),K)
               REST(IROW(I),K) = 0.0
               DO 60 J = K+1, N
                  REST(IROW(I),J) = REST(IROW(I),J)
                                  - X(I) * PIVROW(IROW(K),J)
60             CONTINUE
50          CONTINUE
10       CONTINUE
         END DISTINCT ROW_INDEX
      END
```

Refined Fortran Code for
Gaussian Elimination with Partial Pivoting
Figure 4-2

```
      SUBROUTINE GAUSS( A, M, N, IROW, JCOL )
         REAL A(M,N)
         INTEGER IROW(M)
         INTEGER JCOL(N)
         PARTITION ELEMENTS(PIVROW, REST)
         DISTINCT ROW_INDEX(IROW(I) = I, I = 1,M)
         DISTINCT COL_INDEX(JCOL(J) = J, J = 1, N)
                  :
                  :
         SWAP(IROW(IBIG), IROW(K))
         SWAP(JCOL(JBIG), JCOL(K))
         PARTITION A(I,J)(REST(I.NE.IROW(K)),PIVROW)
         DO 50 I = K+1, M
            X(I) = REST(IROW(I),JCOL(K))
            REST(IROW(I),JCOL(K)) = 0.0
            DO 60 J = K+1, N
               REST(IROW(I),JCOL(J)) = REST(IROW(I),JCOL(J))
                  - X(I) * PIVROW(IROW(K),JCOL(J))
60          CONTINUE
50       CONTINUE
         END DISTINCT COL_INDEX
         END DISTINCT ROW_INDEX
      END
```

Refined Fortran Code for
Gaussian Elimination with Full Pivoting
Figure 4-3

Any of the following may be declared distinct through the use of a *DISTINCT* statement:

- the values (contents) of a collection of scalar variables, e.g.
  - *DISTINCT* *<identifier>*$(A_1, A_2, \ldots, A_k)$
- the values (contents) of the elements of an array along all or part of one dimension, e.g.
  - *DISTINCT* *<identifier>*$(B(I), I = I_1, I_2)$, where $I_1..I_2$ is either part or all of B's extent
  - *DISTINCT* *<identifier>*$(C(I,J), J = J_1, J_2)$ where $J_1..J_2$ is either all or part of C's extent along its second dimension
- the values (contents) of the elements of an array along all or part of two or more dimensions, e.g.
  - *DISTINCT <identifier>* $((D(I,J,K), I = I_1, I_2), K = K_1, K_2))$ where $I_1..I_2$ is either part or all of D's extent along its first dimension, and $K_1..K_2$ is either all or part of D's extent along its third dimension

The effect of a *DISTINCT* statement which declares distinct the contents of a collection of memory locations $\alpha_1, \alpha_2, \ldots, \alpha_k$ is the insertion of code to check that $\alpha_i \neq \alpha_j$ for $i \neq j$, $1 <= i,j <= k$. A failed check results in a fatal error.

The scope of a *DISTINCT* statement extends from the statement itself to the corresponding *END DISTINCT* statement. Because declarations of distinctness need not be properly nested, *DISTINCT* statements are named in order to enable matching of *DISTINCT* declarations with *END DISTINCT* declarations.

If, at any point between a *DISTINCT* statement and the corresponding *END DISTINCT*, any α_i is written, code is inserted immmediately after the write to check that $\alpha_i \neq \alpha_j$ for $i \neq j$, $1 <= j <= k$. A failed check results in a fatal error.

If the number of memory locations whose contents are declared distinct is relatively small, then the checking overhead is not excessive. If, on the other hand, the number of memory locations is any greater than "relatively small," then the amount of checking required can easily get out of hand, unless it can be optimized out.

If we step back for a moment to survey what we have been doing with the *PARTITION* and *DISTINCT* constructs, we see that, at least informally, the appropriate "data structure" for writing parallel code is the *set*: a list of objects containing no repeated objects. Rather than introduce the *set*, with its likely attendent overhead as an actual data structure, we continue to use Fortran's's intrinsic data structures in writing code, and introduce constructs to inform the compiler of the data's *set*-like nature; we thus make use of the *set* as the conceptual data structure and implement it using (hopefully) efficient intrinsic data structures.

When, for whatever reasons, the implementation data structure must be modified, the Refined Language compiler must be informed, in a verifiable fashion, that it retains its *set*-like properties. For this purpose, Refined Fortran contains a small number of intrinsic functions such as the *SWAP* function employed in the following example.

Consider the code of Figure 4-1, which performs Gaussian elimination with partial pivoting.

On each iteration of the *DO 10* loop, that row of *A* which has not yet been used as the pivot row, and whose pivot column element is the largest among all such rows is chosen as the pivot row. As each pivot row is chosen, it is moved upward in array A, so that at the end those rows which have been used as pivot row appear in the order in which they were so used. In the above code rows are not actually moved. Rather, elements of *IROW* are swapped to reflect the desired movement of rows.

In order to recognize the fact that the *DO 40*, *DO* 50 and *DO* 60 loops may be parallelized it is necessary to see that the values of *IROW(1)*, *IROW(2)*, ... , *IROW(M)* are pairwise distinct, and that, as a result, memory locations *A(IROW(1))*, *A(IROW(2))*, ... , *A(IROW(M))* are disjoint. The reason they are is that:

- they are initialized to 1, 2, ... , M respectively in the *DO 5* loop
- the only write to any element of *IROW* is a part of a swap of two elements of *IROW*

In the particular case of the code of Figure 4-1, if a parallelism detector "knew" to look for distinctness of the elements of IROW, it might determine initial distinctness by examining the *DO 5* loop. It would be more difficult to recognize the swap which preserves distinctness.

Figure 4-2 shows Refined Fortran code in which the programmer has indicated distinctness in a verifiable manner. The *DISTINCT* statement indicates to the compiler that the elements of *IROW* are intended to be distinct, and initializes them in such way that the compiler can determine, with very little effort, that they are initially distinct. The fact that swapping of elements of *IROW* is accomplished via a call to the intrinsic function *SWAP* enables the compiler to determine that they remain distinct, and to parallelize the *DO 40*, *DO 50*, and *DO 60* loops.

The case of Gaussian elimination with full pivoting, in which both rows and columns are swapped is shown in Figure 4-3.

The *DISTINCT* statement, like many conventional constructs, gives the programmer the power to create arbitrarily large amounts of overhead. Its intended use is in cases in which there is little or no overhead. The initialization of variables within the *DISTINCT* statement, together with the introduction of a small number of functions such as *SWAP*, makes the desired reduction of overhead -- to zero -- possible in a large variety of actual codes.

5. Odds and Ends

In a short presentation such as this, many questions are raised whose answers cannot, due to space limitations, be immediately answered. In the case of the present discussion of Refined Fortran, the list includes at least the following items:

- *formal* specification of the syntax/semantics of Refined Fortran extensions (but see Appendix A)
- specifics of the optimization of run-time partition-element membership checking and distinctness checking
- directives to override the compiler's decisions as to which potential parallelism is to be exploited and which not

All of these questions, and many additional ones, have been considered, and will be answered in subsequent publications.

6. Conclusions

The Refined Language approach makes no claim to simplifying or automating the process of designing novel parallel algorithms or that of parallelizing existing sequential

algorithms which are not automatically parallelizable. These remain (sometimes) difficult intellectual tasks, of the same nature as the by-now-better-understood task of designing and debugging sequential algorithms/programs.

What it does simplify is the task of debugging and maintaining parallel code. Moreover, it does so without greatly complicating the task of the programmer. Conceptual "partitioning" of data structures is a fundamental aspect of the design of parallel algorithms. The programmer who has designed a parallel algorithm or understood someone else's design understands this "partitioning" perfectly well, but neither conventional programming languages nor their control-parallelism-extended dialects provide the programmer with a means to tell the compiler about it.

Refined Languages enable the programmer to give the compiler information, already known to the programmer, which the compiler can use to determine if parallelization is actually justified, and to generate debuggable parallel code when the safety of parallelization is conditional upon factors determinable only at run time.

In most cases of actual codes, we have been able to reduce checking, where it is required, to at most an upper bound and a lower bound check. Even if this amount of checking is objectionable, though, the Refined Language approach can be used as a debugging aid. As either a compilation or a run-time option, checks can be turned off. If an anomaly is perceived on an execution with a particular data set, checks can be turned back on, and the code rerun.

The annotation provided by the programmer through the use of Refined Language constructs can be just as useful to the reader as to the compiler. Writing a parallel algorithm in a Refined Language can render the algorithm far more understandable to the human reader than would be a control-parallelism extended representation; the former contains explicit justification of the intended parallel execution.

References

[1] D. Klappholz, H. G. Dietz, K. Stein, H. C. Park, and X. Kong, "Refined Languages: An Evolutionary Approach to the Use of Sequential Languages for Programming Parallel(MIMD) Machines," in Parallel Processing: State of The Art Report, Pergamon-Infotech, Maidenhead, Berkshire, U.K., 1987, pp. 59-70

[2] Henry Dietz, and David Klappholz, "Refined FORTRAN: Another Sequential Language for Parallel Programming," in Proc. Int'l Conference on Parallel Processing, August, 1986

[3] Aho, A., Sethi, R., and Ullman, J., Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986

[4] Wolfe, Michael, Optimizing Supercompilers for Supercomputers, Pitman/MIT Press, 1989

[5] Allen, J.R., Dependence Analysis for Subscripted Variables and its Application to Program Transformations, Ph.D. Thesis, Rice University, April, 1983

[6] Banerjee, Utpal, Dependence Analysis for Supercomputing, Kluwer Academic Publishers, 1988

[7] Horwitz, S., P. Pfeiffer, and T. Reps, "Dependence Analysis for Pointer Variables," in Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, 1989.

[8] Klappholz, D., Kallis, A., and Kong, X., "Refined C - An Update" in Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing, Urbana, IL, August 1-3, 1989, MIT Press

Appendix: Partition Statement

A-1. Declaration, Scope, and Invocation

If data is to be partitioned, the names of the partition elements must first be declared in a *PARTITION ELEMENTS* statement. Names of partition elements may not appear in *COMMON* or *EQUIVALENCE* statements.

Within a subprogram, a particular name may be used in only one *PARTITION ELEMENTS* declaration. The following, therefore, constitutes a compile-time error:

```
PARTITION ELEMENTS(X,Y)
PARTITION ELEMENTS(V,Y)
```

A partition is activated through the execution of a *PARTITION* statement, which specifies defining predicates for a list of partition elements which have previously been declared in a single *PARTITION ELEMENTS* statement. A partition remains in effect until either the end of the scope of the partition elements' names or the invocation of a different partition which utilizes the same set of partition elements. Thus, in the code of Figure A1 the first partition remains in effect until the second *PARTITION* statement is executed.

```
PARTITION ELEMENTS(B,C)
        :
PARTITION A(K)(B( π11 ), C( π12 ))
        :
STRAIGHT LINE CODE
        :
PARTITION A(K)(B( π21 ), C( π22 ))
```

Re-Partitioning of an Array
Figure A1

Multiple partitions of the same data structure involving different sets of partition elements may be in effect concurrently. Thus, in the code of Figure A2 after both *PARTITION* statements have been executed, *B*,*C*,*D*, and *E* are all legal names for parts of *A*. The Refined Language compiler understands that *B* and *C* are disjoint, that *D* and *E* are disjoint, and that, for example, *B* and *D* need not be.

```
PARTITION ELEMENTS(B,C)
PARTITION ELEMENTS(D,E)
        :
PARTITION A(K)(B( π11 ), C( π12 ))
        :
STRAIGHT LINE CODE
        :
PARTITION A(K)(D( π21 ), E( π22 ))
```

Multiple Concurrent Partitions of an Array
Figure A2

The same set of partition element names may be used to partition different data structures. The code of Figure A3 is, therefore, legal. References to the different *A*'s and the different *B*'s are disambiguated by referencing them as *X.A*, and *Y.A* or *X.B* and *Y.B* respectively.

```
PARTITION ELEMENTS(A,B)
        :
PARTITION X(K)(A( π11 ), B( π12 ))
        :
STRAIGHT LINE CODE
        :
PARTITION Y(K)(A( π21 ), B( π22 )
```

Partitions of Different Arrays Using Same Partition Element Names
Figure A3

A data structure may be partitioned differently on different arms of a conditional. Thus, in the code of Figure A4 whichever partitioning of *A* was actually executed will be in effect immediately following the *ENDIF*.

```
PARTITION ELEMENTS(FIRST, LAST)
        :
INTEGER A(50)
        :
IF (X = 0)
    THEN PARTITION A(K)(FIRST(K .LE. 25), LAST)
    ELSE PARTITION A(K)(FIRST(K .LE. 40), LAST)
ENDIF
```

Alternative Partitionings of an Array on Alternate Arms of a Conditional
Figure A4

If a partition element is referenced, but no partition which uses it is in effect, the result is a fatal error. The code of Figure A5 results in a run-time fatal error if the *THEN* arm of the conditional is executed, and *A* has not previously been partitioned into *LOW* and *HIGH*.

```
PARTITION ELEMENTS(FIRST, LAST)
PARTITION ELEMENTS(LOW, HIGH)
        :
INTEGER A(50)
    :
IF (X = 0)
    THEN PARTITION A(K)(FIRST(K .LE. 25), LAST)
    ELSE PARTITION A(K)(LOW(K .LE. 40), HIGH)
ENDIF
    :
STRAIGHT LINE CODE
        :
LOW(M) = 33
```

Potential Reference to an Inactive Partition
Figure A5

A-2. Partition Predicates

The values of variables used in evaluating partitioning predicates are the values of those variables immediately before the *PARTITION* statement is executed. Thus, in the code of Figure A6 *B* would consist of *A(1)* thru *A(24)*, and *C* would consist of *A(25)* thru *A(50)*, while in the code of Figure A7 *B* would consist of *A(1)*, and *C* would consist of *A(2)* thru *A(50)*.

```
PARTITION ELEMENTS(B,C)
INTEGER A(50)
        :
J = 25
PARTITION A(K)(B(K .LT. J), C)
```

Illustration of Evaluation of Partition Predicate
Figure A6

```
PARTITION ELEMENTS(B,C)
INTEGER A(50)
        :
J = 2
PARTITION A(K)(B(K .LT. J), C)
```

Illustration of Evaluation of Partition Predicate
Figure A7

The partition elements into which a data structure is partitioned need not be contiguous in the original data structure. Thus, in the code of Figure A8 *EVEN* consists of the even-numbered elements of *A*, and *ODD* consists of the odd-numbered elements.

```
PARTITION ELEMENTS(B,C)
INTEGER A(50)
        :
PARTITION A(K)(EVEN((K/2)*K = K), ODD)
```

Non-Contiguous Partition Elements
Figure A8

After an array has been partitioned, members of the partition elements are referenced in the same way as elements of the original array. In the *QUICKSORT* code, for example, a reference to *UPPER*(1) is a reference to *A*(1) -- legal if 1 satisfies the predicate which defines *UPPER*, and illegal otherwise.

Partitions are invoked dynamically and can be refined dynamically, i.e., a partitioned data structure can be further partitioned. In the *QUICKSORT* code, for example, the partition element *UPPER* defined at one level of recursion is further partitioned into *UPPER* and *LOWER* at the next level of recursion.

Finally, a Refined Language *partition* need not be a partition in the formal sense. Formally, the elements of a partition must be non-empty, must be pairwise disjoint, and must exhaust the set which they partition. The elements of a Refined Language *partition* need only be pairwise disjoint; empty *partition* elements are allowed as is a set of *partition* elements which do not exhaust the data structure which they *partition*. If a partition element is empty, then any reference to it results in an error.