*to appear in "Foundations of Dependable Computing: Vol III, System Implementation", Gary Koob editor, Kluwer Academic Publishers, 1*

# COMPILER SUPPORT

## 3.1 SPECULATIVE EXECUTION

## 3.2 COMPILER ASSISTED SYNTHESIS OF ALGORITHM-BASED CHECKING IN MULTIPROCESSORS

to appear in "Foundations of Dependable Computing: Vol III, System Implementation", Gary Koob editor, Kluwer Academic Publishers, 1

Section 3.2

# Compiler Assisted Synthesis of Algorithm-Based Checking in Multiprocessors[1]

**Prithviraj Banerjee[2]**
**Vijay Balasubramanian[3]**
**Amber Roy-Chowdhury[2]**

*Abstract*

*In this section we describe a compile-time approach to synthesizing algorithm-based checks for numerical programs. The compiler is used to identify linear transformations within loops and for restructuring nonlinear program statements to introduce more linearity into the program. The data manipulated linear statements are then checked by the introduction of checksums, which can often be done more cheaply than replication. We discuss the implementation of a source-to-source restructuring compiler based on the above approach, and present results of applying this compiler to routines from LINPACK, EISPACK and the Perfect Benchmark Suite.*

## 3.2.1 Introduction

The advent of cost-effective VLSI components in the past few years has made feasible the commercial development of multiprocessor systems. Since the probability of one or more processors failing in such multiprocessor systems is quite large, it is desirable to build some fault tolerance feature into them. In this section we will discuss some cost-effective techniques for fault tolerance in multiprocessor architectures. The requirements for high performance and fault tolerance are seemingly contradictory: parallel architectures and algorithms

[1]This research was supported in part by the Office of Naval Research under contract N00014-91-J-1096

[2]Center for Reliable and High Performance Computing, University of Illinois, Urbana, IL 61801

[3]Computer Science Dept., Xavier University, New Orleans, LA 70125

developed for high performance attempt to achieve maximum utilization of each of the processors, while fault tolerance requires redundant computations and checks to ensure that the results of the computations are correct. The result is that conventional fault tolerance techniques are very expensive when applied to multiprocessor architectures.

There are two basic approaches to achieving fault tolerance in multiprocessors: *static or masking redundancy*, and *dynamic or standby redundancy*.

In the *static or masking redundancy* approach, one uses N copies of a module and votes on the results. One can combine the scheme with the use of a disagreement detector and a switching unit to produce a hybrid redundant system. Clearly, in a multiprocessor, one can apply this at several levels. One can replicate each processor and vote on the result of each processor's computation, or one can replicate the entire multiprocessor and vote on the combined result. A third option is to divide the P processors of the multiprocessor into P/N groups of N processors, each group voting on its results before communicating to other groups. The first two options incur a large hardware overhead (at least 200% if using triple modular redundancy, N = 3), while the third option has a large time overhead (upto 200% in time for N =3). In order to provide error masking, all critical transactions must be replicated and voted upon, where the minimum degree of replication is triplication. Such an approach has been used in the FTMP[1], the C.VMP[2], and the SIFT multiprocessor[3]. The FTMP and C.VMP performed the voting on triplicated set of computations in hardware, while the SIFT performed the voting on triplicated set of computations in software.

A lower cost fault tolerance technique in multiprocessors is to use *dynamic or standby redundancy*, where one first uses some technique to detect a faulty processor and reconfigures the system by bringing in a spare processor. In this approach, a mechanism has to be provided to produce an indication of an error occurrence during the system operation. This can be achieved using either hardware or time redundancy. Once the presence of a faulty processor is detected, a fault location or diagnosis procedure is used. The faulty processor is now replaced by a spare processor through reconfiguration techniques. Finally, error recovery is performed, whereby the spare processor takes over the computations of the faulty processor from where it left off using some checkpointed information. This approach is usually the most cost effective approach to achieving fault tolerance in general purpose multiprocessors.

This section deals with low cost techniques for detecting errors in multiprocessors in a dynamic redundancy approach. The issues of reconfiguration and

to appear in "Foundations of Dependable Computing: Vol III, System Implementation", Gary Koob editor, Kluwer Academic Publishers, 1S

recovery are also important, but not discussed in this section. There are several approaches to performing the error detection in multiprocessors. (1) Use of standard circuit level or module level coding and self-checking techniques on components of a multiprocessor. (2) Duplication and comparison using space and time redundancy. (3) Algorithm-based methods. Use of the first techniques is well-known [4] and will not be discussed in this section. We will now review the second and third techniques.

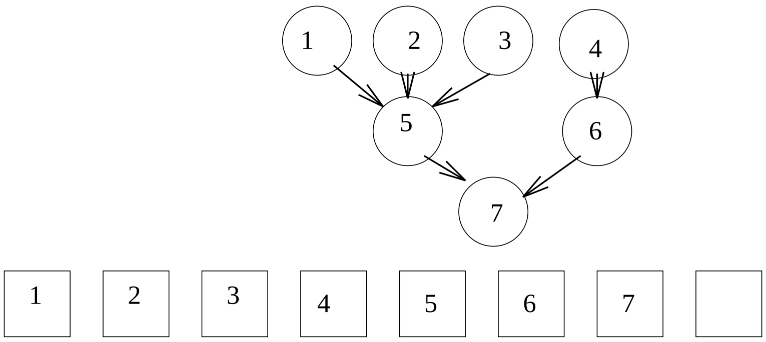
## *Error Detection through Duplication and Comparison*

If the multiprocessor system does not provide any lower level techniques for error detection, one can resort to some general system level techniques for error detection using duplication and comparison. In order to provide on-line error detection, all critical transactions performed in a multiprocessor can be duplicated and compared. The duplication and comparison operations can be performed using space or time redundancy. There are several ways this can be achieved. One can duplicate each processor of the multiprocessor and compare the results before communicating to the processor pairs. For example, duplication and comparison in hardware is performed in the Stratus multiprocessor[5], and in the Intel 432 system [6].

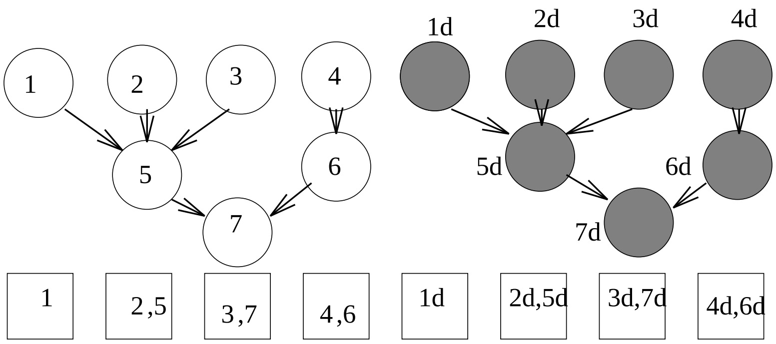
An equivalent way of viewing this option is to divide the $P$ processors of a multiprocessor into $P/2$ pairs. The global common memory consisting of $M$ memory modules can again be divided into $M/2$ pairs. Comparators can be kept inside each processor and memory module and results of both computations have to match for an operation to be executed. The comparison of results of computations can be performed in hardware at each machine cycle, during each memory operation. If an error is detected by a processor pair, both processors of the pair are powered off, and the computations can proceed on the $P-2$ remaining processors, configured as $(P-2)/2$ pairs of processors.

A second approach is to duplicate the entire multiprocessor, and compare the results of the parallel computations. This option is really not used in practice.

In the third approach, one uses duplication and comparison using time redundancy. Such a technique is needed when one does not wish to spend the 100% overhead that is needed in the space redundancy approaches. Specifically, the $P$ processors duplicate the same parallel computation in time, on a different set of processors, such that each processor only affects one copy of the result. This can readily be achieved by dividing the $P$ processors into $P/2$ groups of




(a) Original task graph mapping




(b) Duplicated task graph mapping

**Figure 3.2.1** Example of mapping duplicated task graphs on disjoint sets of processors

two processors (pairs), each of which compares the results of the pair before communicating with other processors.

Figure 3.2.1 shows an example task graph that is duplicated and compared for detection of faults. Case (a) shows that original task graph that is mapped onto 8 processors. Case (b) shows that the the task graph is duplicated, and each copy is mapped onto a disjoint set of 4 processors. The results of the two copies are compared twice, each time by a processor from each subgroup. The above approach in the worst case may require 100% time overhead. However, it may actually result in less time overhead than that due to idleness in processors in the original task mapping. Clearly, if there were no task dependencies in the original task graph, one can map tasks efficiently onto all processors of a multiprocessor, and keep all processors busy all the time. Hence it is possible to get a perfect speedup of $P$ on $P$ processors on that task graph. However, in the presence of task dependencies, one often finds processors that are idle, since there are no ready tasks. Such situations give rise to speedups that are less than $P$. In such situations, one can map the original task graph on $P/2$ processors (as shown in Case (b)), get better processor utilization, and use the remaining $P/2$ processors to perform the duplicate computation of the task graph. Hence, in real task graphs, one can observe less than 100% time overhead.

The comparison operation can also be performed in software, except that now, the granularity of computation before a comparison can be performed has to be larger than a machine cycle since software comparison takes longer than hardware comparisons. The duplication of tasks and their allocation on different processors of a multiprocessor can be performed such that the duplicated tasks and their comparison tasks get executed on different processors.

## *Algorithm-Based Error Detection*

From the above discussions, it is clear that general purpose error detection techniques in multiprocessors can require an overhead of about 100% in hardware or time. This is because when one uses $P$ processors in a multiprocessor to get $P$ times the throughput of a uniprocessor, in order to achieve fault detection, one is either forced to use twice the number of processors, to keep the same throughput, or use the same number of processors and get half the throughput. For certain specific parallel applications running on these multiprocessors, it is often possible to use some algorithm-based schemes for error detection at much lower cost in hardware or time. Another advantage of algorithm-based techniques is that it is possible to gain a lot of fault tolerance at low cost with off-the-shelf hardware that has little or no concurrent error detection capability, and most multiprocessors that are currently built fit this class.

Algorithm-based fault tolerance (ABFT) was proposed originally by Huang and Abraham [7] for matrix computations on array processors. Since then, algorithm-based fault tolerance (ABFT) and algorithm-based error detection (ABED) techniques have been proposed for numerous applications using array processors and systolic arrays. All of them deal with low cost fault tolerance/error detection techniques specific to the algorithms being executed.

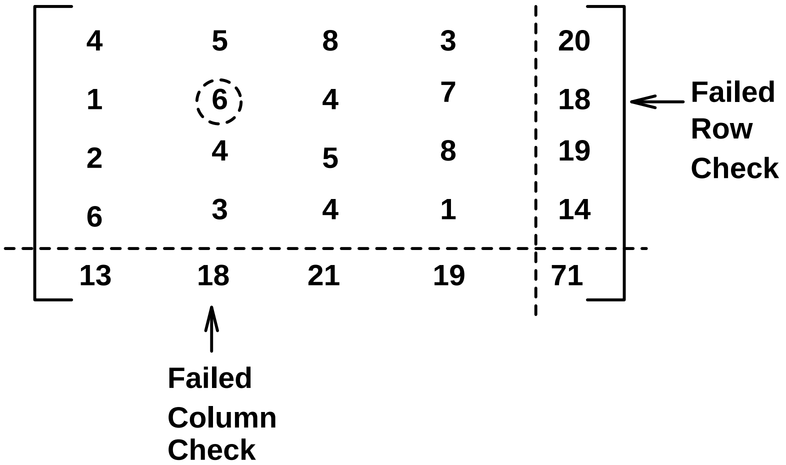



**Figure 3.2.2** Checksum matrix encoding

We will now provide a brief review of the algorithm-based fault tolerance method. Conventional data encoding is done at the word level in order to protect against errors which affect bits in a word. Since a faulty processor in a multiprocessor could affect all the bits of a word it is operating on, we need to encode the data at a higher level. This can be done by considering the set of input data to the algorithm and encoding this set. The original algorithm must then be redesigned to operate on this encoded data and to produce encoded output data. The redundancy in the encoding would enable the correct data to be recovered or, at least, to recognize that the data are erroneous.

We illustrate the application of an algorithm-based checking technique by an example: the multiplication of two $N \times N$ matrices [7]. In the checksum encoding, an extra row and an extra column is appended to the original matrix which are the sums of the elements of the columns and rows, respectively [7]. For each row or column in the matrix, considered as a vector, $(a_1 a_2 \cdots a_N)$, the check elements $CS$ is appended to the vector where

$$CS = \sum_{i=1}^{N} a_i \tag{3.2.1}$$

Figure 3.2.2 shows an example of a $4 \times 4$ matrix, augmented by the row and column checkum encoding. When two such matrices are multiplied, the resultant matrix preserves the checksum property. If there is an error in the result matrix element $(i, j)$, it will be identified by verifying the equality of the sum of the row elements with the checksum for row $i$, and by verifying the equality of the sum of the column elements with the checksum for column $j$. Suppose the matrix in Figure 3.2.2 is the result of a matrix multiplication, and element (2,2) is erroneous. Then the error will be detected in a failed row 2 check, and a failed column 2 check. Once the erroneous element is identified, the correct element can be reconstructed by taking the sum of all elements of that row (column) except the erroneous element and subtracting this sum from the row (column) checksum.

This is illustrated in Figure 3.2.3 which shows a $5 \times 4$ row checksum encoded matrix multiplied by a $4 \times 5$ column checksum encoded matrix on a $5 \times 5$ processor array having row and column broadcasting capability to produce a $5 \times 5$ full checksum matrix. The operation of the array is as follows. For the matrix elements $a_{i,j}$ on the left-hand side of the array, the top four rows consist of the information part of the matrix $A$, and the fifth row is the summation vector; the elements $a_{i,j}$ are broadcast from the left boundary of the array to processors in the $i$th row at time $j$. For the matrix elements $b_{j,k}$ on the top of the array, the leftmost columns are the information part of the matrix $B$ and the fifth column is the summation vector. The elements $b_{j,k}$ are broadcast from the top boundary of the array to processors in the $k$th column at time $j$. At time $j$, the processor $P_{i,k}$ performs the product of $a_{i,j}$ with $b_{j,k}$, and accumulates the product in a register. Thus after $n$ steps, each processor calculates an element of the $C$ matrix. Subsequently, one performs a verification of the checksum property for each row $i$ and each column $j$. The verification of elements in row $i$ is performed by processors in row $i + 1$. If the processor $(i, j)$ in the array is faulty, it will affect the output element $(i, j)$ of the matrix $C$, hence the checks will fail for row $i$ and column $j$, correctly identifying the faulty processor. The erroneous data element $(i, j)$ can be corrected by subtracting the sum of all elements of the row $i$ except element $(i, j)$ from the row checksum for row $i$.

For an $N \times N$ array of processors, the original matrix multiplication can be performed in n time steps. The fault tolerant matrix multiplication needs an $(N+1)\times(N+1)$ array. The checksum verification can be performed in parallel among processors of the rows and columns using a tree based summation, which would take $2log(N)$ steps. The redundancy overhead is $2/N$ in hardware and $2/log(N)$ in time. For large values of $N$ ($> 100$), the overheads are less than 5-10%.

The above scheme assumed a processor array of size $N \times N$ for performing a matrix multiplication of matrices of size $N \times N$. In reality, since the number of

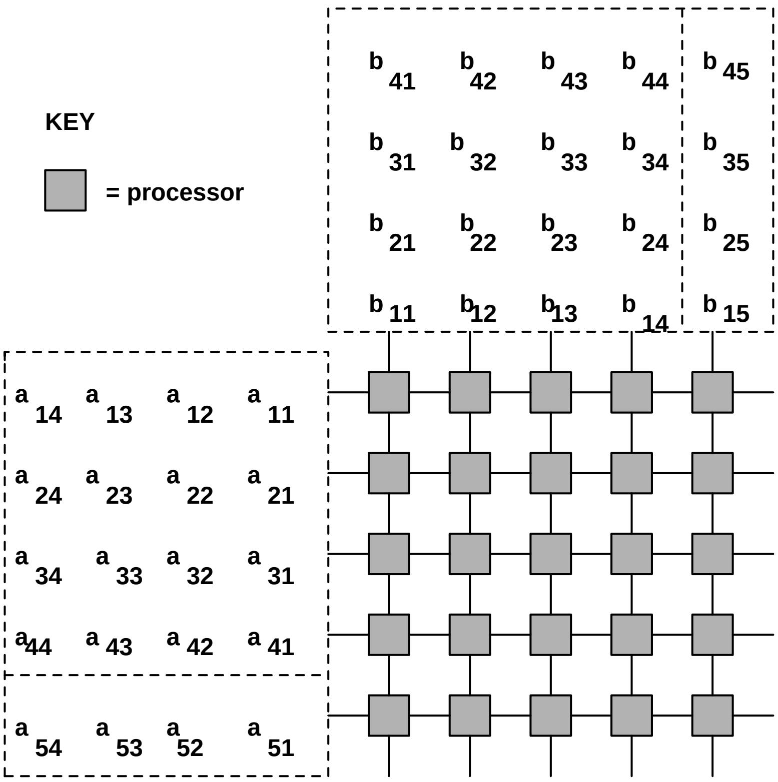



**Figure 3.2.3** Checksum matrix multiplication on an orthogonal array

processors is limited (to say $P \times P$ array), a partitioned matrix multiplication scheme can be used. The basic idea involves the partitioning of each matrix into an $N \times P - 1$ submatrix, and performing multiplication of partitioned submatrices with checksum rows or columns for each partition. In order to avoid the effect of a single faulty processor corrupting multiple data elements in the same row or column of the larger result matrix, each partitioned matrix multiplication is performed by rotating the matrix one row and column position at each step. Details of the algorithm can be found in [7].

The idea of algorithm-based fault tolerance was proposed initially for systolic arrays. While these proposed techniques were interesting, none of the results were practically applicable since there were not many commercially available systolic array processors for actual evaluation of the schemes. Recently, Banerjee, et. al., have reported on the application of the ABFT techniques to general purpose multiprocessors such as hypercube multiprocessors [8]. While previous techniques for ABFT ignored the effects of finite precision roundoff errors on the encodings, the actual studies of ABFT reported on an Intel iPSC hypercube measured the real error coverage in the presence of finite precision arithmetic. More detailed evaluations with regard to error coverages, timing overheads, and error latency were reported in [24]. The schemes did not involve any hardware modifications or overhead.

We now describe the application of algorithm-based fault tolerance techniques for the matrix multiplication problem on a general purpose hypercube multiprocessor. The host computer of the hypercube partitions the matrix $A$ into a number of rectangular strips by rows equal to the number of processors in the hypercube and sends one strip to each processor. The complete matrix $B$ is also sent to each processor. Each processor $P_i$ performs the submatrix multiplication $C_i = A_i \times B$ using a sequential algorithm. At the end, each processor sends the submatrices of the result matrix back to the host.

The algorithm is modified to include system level checks based on the checksum matrix encoding as shown in Figure 3.2.4. The elements of each column of the matrix $A$ are summed together to form the row $CC(A)$, called the column checksum of $A$. The matrix multiplications $C = A \times B$ and $D = CC(A) \times B$ are performed. The column checksum of matrix $C$, $CC(C)$, is computed and compared to $D$. In the absence of faults and round-off errors, $CC(C)$ and $D$ should be identical.

The $A$ strips are duplicated among processor pairs, such that the two processors (hereafter, referred to as nodes) in a pair are mutual neighbors (or mates) in the hypercube. Node $i$ computes the column checksum of the $A$ strip of its neighbor

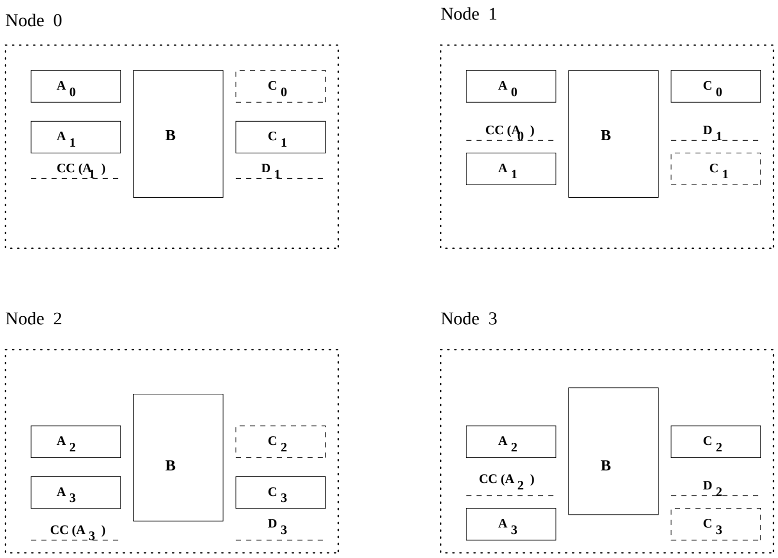



**Figure 3.2.4** Algorithm-based fault detection for matrix multiplication on a general purpose multiprocessor

$CC(A_{mate(i)})$, and then the $D$ strip of its mate, $D_{mate(i)} = CC(A_{mate(i)}) \times B$. Node $i$ then obtains the $C$ strip of its mate, $C_{mate(i)}$. Node $i$ checks its mate by comparing $CC(C_{mate(i)})$ to $D_{mate(i)}$ and sends the result (pass or fail) to the host. Finally, the host judges the computation error-free if all nodes "pass"; otherwise, it judges the computation erroneous.

If there is a fault in a node during the regular matrix multiplication computation, it will be detected by the row check with a high probability, since the checksum row for a strip $C_i$ is calculated by the node that is the neighbor of Node $i$. The above checksum encoding scheme was implemented on a 16 processor Intel iPSC/1 hypercube for cubes of various sizes (4, 8 and 16 nodes), using randomly generated $64 \times 64$ matrices. It gave high error coverage (75-96% for bit level errors, and 100% for word level errors) while maintaining low time overheads (10-30%). Details of the implementation as well as the experimental results can be found in [8].

Algorithm-based fault tolerance techniques have been applied to a wide range of applications including matrix vector multiplication [9], matrix solvers [10], Fast Fourier Transform [9, 11, 12] QR Factorization [12], and adaptive filtering [13].

to appear in "Foundations of Dependable Computing: Vol III, System Implementation", Gary Koob editor, Kluwer Academic Publishers, 1

### 3.2.2 Compiler Assisted Synthesis of Algorithm-based Checking

Although algorithm-based fault tolerance (ABFT) and algorithm-based error detection (ABED) are viable approaches in view of their low time and memory overhead, their practical application is limited in that all the parallel applications developed so far have to be reprogrammed manually to provide error detection. It would be desirable to have an automated (e.g. compiler-based) approach to the synthesis of ABED schemes to relieve the user of the task of devising these checks. The basic idea is to have the compiler analyze high-level language programs, identify linear transformations in DO loops, and automatically insert system-level checks based on this property. Most of the ideas regarding compiler-assisted generation of algorithm-based checks were developed in [14].

The target programs for this approach of automatic check insertion by the compiler are parallel Fortran programs, since there is a large body of existing software written in Fortran and it continues to be the preferred language for scientific programs. In subsection 3.2.8 we discuss the implementation of a compiler pass for automatic check insertion using Parafrase-2 [15] as a core. Parafrase-2 is an existing source-to-source parallelizing and vectorizing package developed at the University of Illinois.

In this section, we discuss rules by which one may identify linear transformations in loop operations encountered in Fortran programs. Although the implementation discussed here is specifically for Fortran programs written for the Intel iPSC/2 hypercube multicomputer, the theory and implementation are general enough to be applied to parallel programs written for other general purpose multicomputers. Subsection 3.2.3 describes the linearity property and its identification. Subsection 3.2.4 presents compiler transformations to introduce linearity in statements. Subsection 3.2.5 discusses the exploitation of the linearity property to produce system level checks for a multiprocessor system. Subsection 3.2.6 discusses the concepts of symbolic linearity and program linearity. Subsection 3.2.7 discusses various tradeoffs involved in linearity-based checking. Subsection 3.2.8 discusses the implementation of a compiler-assisted check generator. Subsection 3.2.9 presents the results of running several programs through the automatic check generator. Subsection 3.2.10 presents a summary of the discussion in this section.

```
DO I = 1,M
    DO J = 1,N
        DO K = 1,L
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
        ENDDO
    ENDDO
ENDDO
```

**Figure 3.2.5** Matrix multiplication code

### 3.2.3 Linear Transformations in Fortran Loops

In this section, we address the issue of linear transformations in Fortran DO loops and their identification at a symbolic level. We restrict ourselves to operations on two-dimensional arrays in order to relate to practical numerical algorithms; the extension to multi-dimensional arrays is very simple, and will be discussed later. We consider indexed program statements of the form $S(I^1, I^2, \ldots, I^k)$, where $k$ denotes the number of distinct loops surrounding $S$ and $I^j$ is the index of the $j$th loop. For example, in the matrix multiplication code shown in Fig. 3.2.5, the single indexed statement is of the form $S(I, J, K)$ and has $k = 3$.

#### *The Linearity Property*

We would like to identify statements $S$ which are linear transformations, i.e., satisfy the linearity property. In order to present the most general form of such statements, we introduce some notation. We assume that the output is a two dimensional array involving loop indices $I$ and $J$.

(1) $A$[I, J] denotes a two dimensional *array variable* whose subscripts are real functions of $I$ or $J$; hence, it is equivalent to $A(f_1(I), f_2(J))$ or $A(f_2(J), f_1(I))$, where $f_1$ and $f_2$ are arbitrary real functions. Examples of such variables would be $A(I + 3, 2J + 1)$ and $A(J^2, I)$.

(2) **A**[I, J] denotes a set of such variables, $\{A_1$[I, J], $A_2$[I, J],$\ldots\}$.

(3) $A$[I] denotes an array variable which involves only $I$ as a subscript. It is equivalent to $A(f(I), K)$, $A(K, f(I))$, $A(f_1(I), f_2(I))$, or $A(f(I))$, where $K$ can be a constant or a function not involving $I$ or $J$. $A$[J] is similarly defined.

(4) **A**[I] and **B**[J] denote sets of array variables $A$[I] and $B$[J], respectively.

The general form of the potentially linear statement can now be written as

$$D[\mathrm{I},\mathrm{J}] = f(I, J, \mathbf{A}[\mathrm{I}], \mathbf{B}[\mathrm{J}], \mathbf{C}[\mathrm{I},\mathrm{J}]) \qquad (3.2.2)$$

where $f$ is some real-valued function involving array variables similar to the ones defined above, and possibly, the variables $I$ and $J$, independently. Note that the above form does not exclude $D$[I, J] from appearing in the R.H.S. as well, which would then be a part of **C**[I, J]. When this statement is placed inside a multiply nested DO loop structure with loops having $I$ and $J$ as indices, array $D$ gets updated according to the values of $f$ for allowed $I$ and $J$ values (and values of indices corresponding to other loops, if present). As an example, consider the following statement

$$D(I+3, J) = A(I, 2) * B(L, J+4) + (B(I, 2I))^2 / C(J, I) \qquad (3.2.3)$$

This is consistent with the general form where $D$[I, J] = $D(I+3, J)$, **A**[I] = $\{A(I,2), B(I,2I)\}$, **B**[J] = $B(L, J{+}4)$, and **C**[I,J] = $C(J, I)$. It can be similarly verified that the indexed statement in matrix multiplication code of Fig. 3.2.5 conforms to the general form.

Let $I_k$, $1 \le k \le m$ be values (not necessarily all distinct) that $I$ takes in course of the loop iterations, and let $w_k$, $1 \le k \le m$, be arbitrary real numbers. Then $f$ is *linear in I* if

$$\sum_{k=1}^{m} w_k f(I, J, \mathbf{A}[\mathrm{I_k}], \mathbf{B}[\mathrm{J}], \mathbf{C}[\mathrm{I_k},\mathrm{J}]) = f(\sum_{k=1}^{m} w_k I_k, J, \sum_{k=1}^{m} w_k \mathbf{A}[\mathrm{I_k}], \mathbf{B}[\mathrm{J}], \sum_{k=1}^{m} w_k \mathbf{C}[\mathrm{I_k},\mathrm{J}]) \qquad (3.2.4)$$

In the above equation, summation over an index of a set, say **A**, is assumed to yield a set whose elements are values obtained by summations over the index under consideration of the arrays belonging to set **A**. Note that the L.H.S. of the above equation is nothing but $\sum_{k=1}^{m} w_k D[\mathrm{I_k},\mathrm{J}]$. In some algorithms, $I$ may be restricted to the first dimension of all array variables; it is then a *row* variable and $I$-linearity then translates to *row*-linearity. We can similarly define $J$-linearity (or *column*-linearity in specific cases). $I+J$-linearity (or *row-column*-linearity in specific cases) means that the function $f$ is linear both in $I$ and $J$.

For example, matrix multiplication is a linear operation, as the statement inside the triple loop satisfies the linearity test for $I$, with $m = M$, as shown below

$$\sum_{k=1}^{M} w_k(C(I_k, J) + A(I_k, K) * B(K, J)) = \sum_{k=1}^{M} w_k C(I_k, J) + (\sum_{k=1}^{M} w_k A(I_k, K)) * B(K, J) \quad (3.2.5)$$

The above relation holds because of the distributivity of multiplication over addition. It can be similarly shown that the statement is $J$-linear as well. This $I + J$-linearity translates to row-column linearity as $I$ is limited to the first dimension and $J$ to the second.

In general, $m$ equals the number of $I$-loop iterations, i.e., the linearity property is applied across all allowable $I$ values. In addition, if $m$ is equal to the size of the input and output arrays (indexed by $I$) in the $I$-dimension, then this is called *full linearity*. As an example, the linearity test for matrix multiplication shown above employs full linearity as the number of $I$-loop iterations is equal to the row size of matrices $A$ and $C$. However, for a particular algorithm we might need to apply the linearity property only for a subset of the $I$ values; we are then using a *sub-linearity* property. This is true if $I$, in course of its iterations, covers only part of the relevant arrays in the $I$-dimension. Such a case often arises when the $I$ loop bound(s) is not a constant but depends on an outer loop index or the loop stride is not unity. The extent of linearity is also determined by the range of $J$ values for which $f$ is linear in $I$; usually this range covers all the values that $J$ takes in course its loop iterations.

It can be easily seen that the application of the linearity concept to statements involving multi-dimensional arrays requires just a simple extension of the above definitions. In this case, the output variable can have an arbitrary number of loop indices in its subscripts and the statement is linear with respect to those indices that satisfy the linearity property as defined above. Note that a statement which writes to a single dimensional array is just a special case of the form of Eq. 3.2.4 where one of the loop indices in the output variable is replaced by a constant.

## *Detection of Linearity*

One can determine whether a given transformation is linear by applying the above linearity test to it. However, the following similar looking statement

$$D(I,J) = C(I,J) + A(I,K) * A(I,J+2) \qquad (3.2.6)$$

is not $I$-linear since both $A(I,K)$ and $A(I,J+2)$ vary with $I$ and hence the distributive law is no longer applicable.

A brute force application of the linearity test is not feasible for lengthy and complex statements; moreover, it involves algebraic manipulations and consumes valuable time. We therefore discuss *automatic and fast detection of linearity by the compiler.* We thus need to address the problem at a *symbolic (structural) level.* Let $exp_1$ and $exp_2$ be two expressions occurring as parts of the R.H.S. expression of an assignment. Using the commutative and associative properties of addition, and the distributivity of multiplication over addition, one can translate the linearity test to the following three *recursive linearity rules*:

(R1) $exp_1 \pm exp_2$ is linear in $I(J)$ if and only if both $exp_1$ and $exp_2$ are linear in $I(J)$.

(R2) $exp_1 * exp_2$ is linear in $I(J)$ if and only if either $exp_1$ is linear in $I(J)$ and $exp_2$ does not involve $I(J)$, or vice versa.

(R3) $exp_1/exp_2$ is linear in $I(J)$ if and only if $exp_1$ is linear in $I(J)$ and $exp_2$ does not involve $I(J)$.

In addition, we have the following four fairly obvious linearity rules for simple expressions and functions:

(R4) A constant is not linear in any variable.

(R5) A simple variable is linear in itself.

(R6) An array variable is linear in its subscript variables.

Finally, we have the following linearity rule for functions:

(R7) A function of an expression is linear in the variables the expression is linear in, if and only if the function itself is linear. Examples of non-linear functions are trigonometric functions, square roots, etc.

In order to determine whether a particular statement is linear, the compiler needs to check the satisfiability of the above rules $R1 - R7$ by the statement. For example, the statement

$$C(I,J) = A(J) + D(I+2,J) \tag{3.2.7}$$

is not $I$-linear (from rule $R1$), since $A(J)$ does not involve $I$ and so is not $I$-linear (from rule $R6$). However, the statement is $J$-linear as the R.H.S. expression satisfies rule $R1$ w.r.t. $J$. The statement

$$C(I,J) = A(I,J) * B(I) + D(I) \tag{3.2.8}$$

is neither $I$ nor $J$-linear. $A(I,J)$ and $B(I)$ belong to the same product term and both involve $I$. Hence, this product term violates rule $R2$. Moreover, $D(I)$ does not involve $J$ at all. It can be similarly verified that statement 3.2.3 is neither $I$ nor $J$-linear (rule $R3$ violated), statement 3.2.6 is $J$ but not $I$-linear and the matrix multiplication assignment statement is linear in both $I$ and $J$.

### 3.2.4 Compiler Transformations to Introduce Linearity

Many assignment statements encountered in Fortran programs do violate rules $R1 - R7$ for both $I$ and $J$ and hence are not linear. However, it is possible to restructure a non-linear statement such that it satisfies these rules for at least one loop index which then qualifies it as a linear transformation. Again, this restructuring needs to be done at a symbolic level. The following four simple techniques can be used by the compiler to introduce linearity into statements

(T1) If an expression of the form $exp_1 + exp_2$ is such that one component does not contain the loop index in question, the variable can be expanded to accommodate that index. This creates an instance of the variable for every iteration of the DO loop corresponding to that index. For example, statement 3.2.8 can be made $J$-linear by expanding $D(I)$ to $D(I,J)$. The trade-off here is the introduction of linearity at the cost of additional memory. However, scalar expansion is a technique employed by most parallelizing compilers [16] to reduce edges in the dependence graph, so in fact, the task check generation at compile time may be made easier after parallelization by a parallelizing compiler. Thus, a parallelizing compiler often unknowingly introduces linearity in some statements.

(T2) If an expression of the form $exp_1 * exp_2$ is such that both $exp_1$ and $exp_2$ contain the index in question, variable substitution can be performed by substituting this expression by a single variable containing the relevant loop indices. An additional statement needs to be placed before the restructured statement that simply assigns the multiple variable term to the single variable. For example, the statement

$$\begin{aligned} E(I,J) &= B(I,K)*C(K,J)+B(I,J)*(D(L,J))^2 \\ &\quad +A(I,K)*A(I,K+2) \end{aligned} \tag{3.2.9}$$

is not $I$-linear due to the presence of the expression $A(I,K)*A(I,K+2)$ which has both components involving $I$. Restructuring this statement produces the following code where the second statement is now $I$-linear

$$F(I,K) = A(I,K)*A(I,K+2) \tag{3.2.10}$$

$$E(I,J) = B(I,K)*C(K,J)+B(I,J)*(D(L,J))^2+F(I,K) \tag{3.2.11}$$

Very often, the extra computation due to the substitution process is small compared to what is saved by introduction of linearity since this can avoid duplication of the original statement.

(T3) If an expression of the form $exp_1/exp_2$ violates rule $R3$, then variable substitution can be performed similar to the one described above, which removes variables containing the loop index in question from the denominator. For example, statement 3.2.3 can be made $J$-linear by substituting $1/C(I,J)$ by say, $E(I,J)$. Again an additional statement needs to be introduced which effects this substitution.

(T4) Violation of rule $R7$ can be similarly taken care of by substituting the non-linear expression by a simple array variable.

It may be noted that technique $T1$ is also applicable to expressions where the non-linear component is not a simple expression. In such cases, the above techniques can be applied to this component. Such a recursive approach is discussed in detail in subsection 3.2.8.

It is important to note that restructuring techniques $T2$, $T3$ and $T4$ might increase the program length (and hence computation) by an amount sufficient to offset any gain that can be obtained by introduction of linearity. Hence, they should be used with discretion.

### 3.2.5 Linearity-Based System-Level Checking

In this section, we will discuss how linearity can be exploited to provide low-cost system-level checks for concurrent error detection in multiprocessors by the use of the checksum encoding.

#### *The Checksum Encoding Scheme*

Once a statement is identified to be a linear transformation (or is restructured to satisfy the linearity property), it is possible to use this information to develop system-level checks for detecting errors during computations. Once an index in which a statement is linear has been discovered by the compiler, or the statement has been restructured so that it is linear in a particular index, we may use Eq. 3.2.4 to compute and compare the L.H.S. and R.H.S. values. In the absence of errors, these values should be equal. As an example, let the statement be $I$-linear. First compute $\sum_{k=1}^{m} w_k D[\mathrm{I_k}, \mathrm{J}]$, for a particular value of $J$. We will denote this as the L.H.S. computation. Also compute $f(\sum_{k=1}^{m} w_k I_k, J, \sum_{k=1}^{m} w_k \mathbf{A}[\mathrm{I_k}], \mathbf{B}[\mathrm{J}], \sum_{k=1}^{m} w_k \mathbf{C}[\mathrm{I_k}, \mathrm{J}])$, by first performing the summations of the array variables belonging to sets $\mathbf{A}$ and $\mathbf{C}$ and possibly the summation of the $I$ values, and then computing the value of the function $f$ for the same $J$. This is the R.H.S. computation. These values are then compared with each other; a mismatch indicates error(s) during normal computations. This procedure is repeated for other values of $J$. The role of the compiler is to detect program statements which are linear, restructure some of the non-linear ones and then insert the above system-level checks automatically into the program.

The sums of the array variables computed above, can be regarded as encodings on the input/output data, and are referred to as *weighted checksums*. It is customary to use unity weights $w_k$ for computing these checksums, which are then known as *unity checksums* or just checksums. If we restrict $I$ as a row variable and $J$ as a column variable, then summations which exploit the $I$-linearity property would involve array values belonging to the same column and are called *column checksums* (since $I$ values change but $J$ remains the same); summations for $J$-linearity would involve values in the same row and are called *row checksums*. These terms carry over directly from the checksum encoding proposed for matrix multiplication [7].

Sometimes a statement may fail the linearity condition only because of the presence of a constant in the R.H.S. expression (rule $R4$). A straightforward application of restructuring technique $T1$ will assign the constant to a single dimensional array. During the checking phase, the array values will be summed over 1 to $m$. A more efficient way of performing this computation is to simply multiply the constant by $m$; hence, technique $T1$ need not be applied at all for such non-linear statements.

## *Parallelization Issues*

We first briefly review some basic concepts regarding data dependencies in program statements. Let $S_1$ and $S_2$ be two statements inside a loop. $S_1$ and $S_2$ are said to be involved in a flow dependence (denoted by $S_1 \delta S_2$), if the result of $S_1$ is used by $S_2$ in the current or a subsequent iteration. If in addition, we have $S_2 \delta S_1$, then these two statements are involved in a cycle and the loop has to be executed in a serial fashion. In general, cycles involve other kinds of dependencies as well; details of the dependence notions can be found in [17]. Absence of cycles enables the iterations of the loop to be executed completely in parallel. Note that these concepts can be extended to statement blocks also [16].

Once parallelism has been discovered in a loop, it can be marked as DO, DOALL or DOACROSS, depending on whether the iterations need to be executed serially, can be executed completely in parallel or have a serial execution with some degree of overlap between successive iterations [17]. DOALL and DOACROSS loops can be executed in parallel machines by partitioning the set of iterations and assigning a different set to each processor. This is called loop concurrentization [18]. We refer to the concurrent loop index as the *parallelization variable*. Concurrentization often requires the insertion of synchronization instructions within the body of the concurrent loop [19]. These might be needed in critical regions of the loop body where processors access shared data, or for ensuring proper ordering of the loops during execution. In distributed memory machines synchronization is performed by communication of proper data among processors.

We will focus on parallelization and data integrity checking for a distributed memory multicomputer. A very important issue in parallelization is the consistency of data distribution with the parallel loop constructs in the program. This is especially true for programs with multiple DO loops one after the other. The parallelization of one loop might demand distribution of the input matrices by rows whereas it might be necessary to have a column distribution for another loop. A simple solution is to have the input data distribution governed by the largest parallel DO loop in the program.

```
. . .
DOALL I = 1,M
    . . .
    DO J = 1,N
        . . .
        D[I,J] = f ( A [I], B [J], C [I,J])
        . . .
    ENDDO
    . . .
ENDDOALL
. . .
```

**Figure 3.2.6** Parallel loop under study

## *Checking on a Distributed Memory Multicomputer*

Let us now apply the checksum encoding scheme discussed earlier to a specific distributed memory multicomputer using unity checksums. The procedure will be different depending on whether the checking variable and the parallelization variable are the same or not. In order to clearly illustrate the concepts involved, we assume a multiple loop structure where the $I$-loop is normalized (the lower bound and loop stride are both unity) and is completely parallelizable. In fact, compilers do normalize loops before optimizing them [16]. We further assume without loss of generality, that the $J$-loop is inside the $I$-loop. The parallel loop structure is shown in Fig. 3.2.6. The outer DOALL loop is concurrentized so that the iterations are assigned to the processors in a wraparound fashion; here $I$ is the parallelization variable. The input data is distributed accordingly. Hence, if there are $P$ processors numbered 0, 1, ..., $P$-1 assuming that $M$ is divisible by $P$, processor $K$ gets $M/P$ iterations, and **A**[I] and **C**[I, J] values, corresponding to $I = K+1, K+P+1, \ldots, K+M-P+1$. For the following discussion we let $Z$ denote the set 1, 2, ..., M, and let $Z_K$ denote the set $K+1, K+P+1, \ldots, K+M-P+1$.

### *I-linearity based Checking*

Now suppose that the statement inside the $J$-loop is detected to be $I$-linear. We employ $I$ as the checking variable. The sublinearity property may be used

```
Processor 0

     I-iterations   1,5

Normal computations:

     DO J = 1,N
         D[1,J] = f{A[1],B[J],C[1,J]}
          D[5,J] = f{A[5],B[J],C[5,J]}
     ENDDO

Check computations:

     DO J = 1,N
          E(J) = D[1,J] + D[5,J]
          F(J) = f{A[2]+A[6],B[J],C[2,J]+C[6,J]}
     ENDDO
```

```
Processor 1

     I-iterations  2,6

Normal computations:

     DO J = 1,N
          D[2,J] = f{A[2],B[J],C[2,J]}
          D[6,J] = f{A[6],B[J],C[6,J]}
     ENDDO

Check computations:

     DO J = 1,N
          E(J) = D[2,J] + D[6,J]
          F(J) = f{A[1]+A[5],B[J],C[1,J]+C[5,J]}
     ENDDO
```

**Figure 3.2.7** An I-linearity based checking scheme

to perform the checking. On processor $K$, we compute $\sum_{I \in Z_K} D[\mathrm{I},\ \mathrm{J}]$, and $f(\sum_{I \in Z_K} \mathbf{A}[\mathrm{I}], \mathbf{B}[\mathrm{J}],\ \sum_{\mathrm{I} \in \mathrm{Z_K}} \mathbf{C}[\mathrm{I}, \mathrm{J}])$, for different values of $J$. The check computations are split across processor pairs, preferably neighboring processors; they cooperate with each other to obtain and compare the checksums.

Let us illustrate the sub-linearity based checking scheme for $M = 8$ and $P = 4$. Fig. 3.2.7 shows the normal and check computations performed in processors 0 and 1, with respect to the linear statement only. The execution of other statements proceeds as in the normal algorithm, and is not explicitly shown here. We consider a row cyclic mapping, where processors 0 and 1 get data corresponding to iterations $\{1,5\}$ and $\{2,6\}$,respectively. Each of these data sets is operated on by a set of checksum calculations. The L.H.S and R.H.S. calculations for the sets $\{1,5\}$ and $\{2,6\}$ need to be performed on different processors. In this case, the L.H.S calculation for set $\{1,5\}$ is performed on processor 0; processor 1 performs the R.H.S. calculation. The roles are reversed for set $\{2,6\}$. This requires both the processors to have each other's data sets before the check computations can proceed; this requires a modified mapping which duplicates data in pairs of processors. The values at the end of the checking phase are then mutually exchanged and compared, which requires additional communications. Suppose a fault occurred in processor 0 during these computations. It produces an error with high probability in $D[1, \mathrm{J}] + D[5, \mathrm{J}]$ and/or $f\{\mathbf{A}[2] + \mathbf{A}[6], \mathbf{B}[\mathrm{J}], \mathbf{C}[2, \mathrm{J}] + \mathbf{C}[6, \mathrm{J}]\}$. Since their counterparts in processor 1 are calculated correctly, the comparisons in that processor will produce a mismatch and signal a fault. The role of the compiler is to automate the checking process by inserting statements inside the node program that perform the check computations and communications together with the comparison checks.

The time required for the check computations is much lower than the normal computation time, especially for large input arrays (high $M$ and $N$) and functions $f$ which require a significant amount of computation. Each function $f$ is computed $N * M/P$ times during the normal computation phase in each processor, whereas it is computed only $N$ times during the checking phase. This is evident from Fig. 3.2.7, although the overhead for this example is considerable since we have chosen small dimensions for ease of explanation. Two extra communication steps are required for exchanging the $E$ and $F$ arrays. Thus, the relative overhead for the $f$ computations is $O(P/M)$, which is very low for high $M$. The time required for the computation of the various checksums, which consist of addition operations, is usually much less than the time consumed by the $f$ computations which may involve several floating point operations.

Let us illustrate these ideas for matrix multiplication. Fig. 3.2.5 shows that the $I$ ($J$) iterations are completely independent of each other; hence, we can employ $I$ ($J$) as the parallelization variable. Let us assume without loss of generality that the $I$ iterations are performed in parallel. Concurrentizing the $I$-loop is equivalent to distributing the $A$ matrix by rows to the hypercube processors. Also, the output $C$ matrix resides in the processors in a row distributed fashion. This is because the $A$ and $C$ array variables are functions of $I$. The $B$ matrix is replicated on all the processors; note that the $B$ variable does not involve $I$. If there are $P$ processors (assume that $M$ is divisible by $P$), then each processor gets $M/P$ iterations of the $I$ loop, i.e., a submatrix of $A$ of size $M/P \times L$. Each processor then multiplies its own submatrix with the complete $B$ matrix in parallel, and produces a submatrix of $C$ of size $M/P \times N$. These submatrices are then sent to the host, which reorganizes them to generate the complete $C$ matrix.

$$
\begin{array}{rc}
 & \left[\begin{array}{cccc}
A_{1,1} & A_{1,2} & \cdots & A_{1,16} \\
+ & + & & + \\
A_{5,1} & A_{5,2} & \cdots & A_{5,16} \\
+ & + & & + \\
\vdots & \vdots & & \vdots \\
+ & + & & + \\
A_{13,1} & A_{13,2} & \cdots & A_{13,16}
\end{array}\right] \\
\hline
CC(A) = & \Sigma A \quad \Sigma A \qquad\qquad \Sigma A
\end{array}
\left[\begin{array}{ccc}
B_{1,1} & \cdots & B_{1,16} \\
\vdots & & \vdots \\
B_{16,1} & \cdots & B_{16,16}
\end{array}\right]
=
\begin{array}{rc}
 & \left[\begin{array}{ccc}
C_{1,1} & \cdots & C_{1,16} \\
+ & & + \\
\vdots & & \vdots \\
+ & & + \\
C_{16,1} & \cdots & C_{16,16}
\end{array}\right] \\
\hline
CC(C) = & \Sigma C \qquad\qquad \Sigma C
\end{array}
$$

**Figure 3.2.8** Column checksum encoding for matrix multiplication

The normal algorithm requires no row communications; hence, there is no particular advantage in using row checksums. We then opt for an $I$ sub-linearity based column checksum approach, for purposes of illustration. Let $M = N = L = 16$, and $P = 4$. Fig. 3.2.8 shows the computations (normal and check) for the submatrix multiplication in processor 0, which receives the $I$-iterations 1, 5, 9, and Processor 0 calculates the column checksums of the submatrix of $C$ and stores them in an array $CC(C)$. The column checksums of the submatrix of $A$ are calculated in the neighboring processor of $A$, and stored in an array $CC(A)$. The neighbor also performs the multiplication $CC(A) \times B$ and stores the resultant vector in array $D$. The arrays $CC(C)$ and $D$ are then compared element by element; any mismatch indicates errors during computation.

The normal computation in each processor requires $O(MNL/P)$ operations, whereas the check computation requires only $O(ML/P + LN)$ operations. A constant number of communication steps are needed for the exchange operations for comparison. The relative overhead is, thus, O($1/N$+$P/M$). Hence, the checksum encoding scheme described here is, indeed, a low overhead scheme.

*J-linearity based Checking*

Suppose that the statement inside the $J$-loop is linear in $J$. One could then use $J$ as the checking variable and employ a $J$-linearity based checking scheme. Using a full linearity property necessitates the computations $\sum_{J \in Y} D[\mathrm{I}, J]$, and $f(\mathbf{A}[\mathrm{I}], \sum_{\mathrm{I} \in \mathrm{Y}} \mathbf{B}[\mathrm{J}], \sum_{\mathrm{J} \in \mathrm{Y}} \mathbf{C}[\mathrm{I}, \mathrm{J}])$, for different values of $I$. Here, $Y$ = 1, 2, ..., N is the set of all values that $J$ takes in course of its iterations. Since, our loop distribution scheme assigns all iterations of $J$ for a particular value of $I$ to the same processor (as is clear from Fig. 3.2.6, the entire check computation for that value of $I$ can be performed in the same processor. However, a fault in the processor will invalidate the results. Hence, the L.H.S and the R.H.S.

to appear in "Foundations of Dependable Computing: Vol III, System Implementation", Gary Koob editor, Kluwer Academic Publishers, 1

part of the computation is split among neighboring processors and the results at the end are exchanged and compared. It can be easily seen that the relative overhead is $O(1/N)$, and is low for high $N$.

Since the matrix multiplication assignment statement is linear in $J$, the above scheme is applicable. This results in a row checksum encoding of the matrices $B$ and $C$ where each processor computes the row checksums of its $B$ matrix and its $C$ submatrix. The rest of the computations are very similar to the column-checksum approach and are not discussed here.

It is worthwhile mentioning here that it is possible to have loop constructs where the parallelization variable is neither $I$ nor $J$, which means that the data distribution is governed by some other loop index. The $I$ ($J$) iterations may or may not be distributed across processors which again depends on the relative location of the $I$ ($J$) loop with respect to the parallel loop. But the techniques illustrated above are still applicable where the extent of linearity and the number of processors participating in the check depends on the checking variable loop distribution.

The checking schemes described above are applicable to program statements which are linear in either $I$ or $J$ (or are restructured to be linear). Statements not satisfying the linearity property are duplicated on neighboring processors, and the results are exchanged and compared. In any case, duplication of a particular segment of the program might be preferred over linearity-based checking (if applicable) when that code is executed only in certain processors in the normal algorithm and other processors are idle at that time. This is often true for program segments outside the DOALL loops. In such a situation, duplication does not incur any computation overhead though some communication needs to be done to exchange results of code.

### 3.2.6 Symbolic Linearity and Program Linearity

The successful application of the checksum encoding scheme to a linear statement in a program depends not only on the statement's structure, but also on the program context of the statement. This is best illustrated by a simple example. Let us consider the $I + J$ linear statement in the double loop structure of Fig. 3.2.9. Using $I$ as the checking variable, one cannot perform the summation of the $B$ array values for different values of $I$ since $K$ is also dependent on $I$ (the distributive law is not applicable). However, the use of $J$

```
DO K = 1,L
  . . .
  DOALL I = 1,M
      . . .
      DO J = 1,N
          . . .
          D[I,J] = f ( A [I], B [J], C [I,J])
         . . .
      ENDDO
     . . .
  ENDDOALL
  . . .
ENDDO
```

**Figure 3.2.9** Example loop illustrating program linearity

as the checking variable circumvents the above problem as $K$ is independent of $J$.

We introduce the concept of program linearity of a statement to address such situations. A *symbolically linear* statement is said to be *program linear* in the context of a particular program, if this property can be directly exploited to develop a checksum-based encoding scheme for on-line error detection. Thus, our example statement, although symbolically linear in both $I$ and $J$, is program linear only in $J$ but not $I$. In the future sections, unless otherwise indicated, we use the term "linearity" to indicate symbolically linear statements which are program linear as well. Typically, we check a statement in a loop body for symbolic linearity with respect to the loop index variable. However, the statement may involve additional variables which are different from the loop index (potentially linearizing) variable but are nevertheless dependent on it, perhaps because they are altered within the body of the loop. Such variables are called induction variables. In essence, the idea behind checking for program linearity is to replace all induction variables by the loop index variable and checking the resulting statement for symbolic linearity with respect to the induction variable in the usual manner. If the transformed statement is found to be symbolically linear, the original statement is program linear.

We discuss a procedure for determining whether a statement is program linear in a later subsection.

### 3.2.7 Trade-offs in Algorithm-based Checking

This section discusses various trade-offs involved in checking related to the nature and extent of the checksum encodings. These trade-offs are essentially options to the compiler and depend on the program structure as will be described below. The objectives are: (1) high error coverage, (2) low performance overhead, and (3) low error latency. These objectives are mutually conflicting: realization of (2) can only be achieved at the expense of (1) and (3), and vice-versa. For a better understanding of the issues involved, we assume that $I$ is a row variable and $J$ is a column variable, which is indeed true for many algorithms.

#### *Location of Encoding*

The location of encoding depends on the nature of linearity in the statements inside the multiple loop structure. If the statement is $I$-linear, then a column checksum encoding would be employed. On the other hand, $J$-linearity implies that a row checksum encoding scheme be used. As before we assume that $I$ is the parallelization variable which means that the input arrays are distributed by rows to the processors. Computation of column checksums in general, require elements from different processors. Moreover, the computed values might have to be exchanged. All this implies additional communication. However, error propagation is higher as a fault in any processor can manifest itself as an error in the checksum value. In contrast, row checksums can be computed within processors. In fact, they can be treated as additional array elements and appended to the rows. The advantage is that if the normal algorithm requires communication of rows between processors, then these checksum values can be communicated along with them thus saving in start-up time. The disadvantage is *error masking*: during a checksum computation, an erroneous processor might create errors in opposite directions which could cancel out and produce a correct checksum value. However, the possibility of such error masking is quite low in realistic situations.

## *Extent of Checking*

One could use the full linearity property (if the checking variable covers the full range in its dimension) or a sub-linearity property. Again we assume that $I$ is the parallelization variable so that the input matrices are distibuted by rows. This is particularly relevant in the case of column checksum encodings. Full linearity requires that all processors contribute to the checksum values thus increasing error propagation, as described earlier. However, additional communication is required. Sub-linearity restricts the summations to elements within processors. There is a trade-off here in the form of reduced communication overhead at the expense of error masking. A sub-linearity based column checksum encoding behaves very similar to a row checksum encoding in terms of fault propagation and the time overhead values. Note that sub-linearity based encodings also require communication for the exchange operations, but its localized nature suggests a lower overhead than that for the full linearity based encodings.

Sometimes, sub-linearity is the only solution when conditional statements are present inside loops. It is possible that the execution of a linear statement inside a loop structure is governed by a conditional prior to the statement but inside the loop structure. This implies that the statement might be executed only for certain values of the checking variable. Hence, the extent of checking is automatically limited and sub-linearity needs to be employed.

The extent of checking is also determined by the number of column (row) checksums computed during checking; the encoding can be applied to all columns (rows) or to particular columns (rows) only. Moreover, if some statements are both $I$ and $J$ linear then $I + J$ linearity can be used, and both row and column checksums can be employed. The trade-off here is an increase in fault detection due to the additional checks at the expense of greater overhead.

## *Frequency of Checking*

It is possible that the checking variable loop occurs inside a serial loop in which case this loop needs to be executed repeatedly inside all processors. The frequency of checking then refers to how often the checksum calculations are done during the course of the algorithm. They can be performed after each iteration of the loop or at the end of some iterations only. The higher the frequency of checking, the higher the fault detection capability and the higher the time overhead. Let $J$ be the checking variable in the example loop structure of Fig. 3.2.10. Then the check computation could be done after the normal

```
DO K = 1,L
   . . .
   DOALL I = 1,M
        . . .
        DO J = 1,N
             . . .
             D[I,J] = f ( A [I], B [J], C [I,J])
            . . .
        ENDDO
      . . .
  ENDDOALL
  . . .
ENDDO
```

**Figure 3.2.10** Example loop illustrating checking frequency

computation for every iteration of the $K$ loop; the other extreme is when the checking is performed at the end i.e., after the normal computation in the $L$th iteration.

## *Generation of Encoding*

Many programs have multiple loop structures containing cycles where the output array variable in a statement is also present as input. For example, consider the loop structure of Fig. 3.2.10 again where $D$[I,J] $\in$ **C**[I,J], and is involved in a cycle with respect to loop index $K$. We might desire to perform checking after every iteration of the $K$ loop. Instead of computing the input row checksums for the $D$ array for each iteration, it is desirable to store the output checksums computed in the previous iteration and then use them for the current iteration. Thus, the $D$ array could be expanded to accommodate the row checksum column as $D[\mathrm{M}+1,\ \mathrm{J}] = f(\mathbf{A}[\mathrm{I}],\ \sum_{J=1}^{N} \mathbf{B}[\mathrm{J}],\ \sum_{J=1}^{N} \mathbf{C}[\mathrm{I},\ \mathrm{J}])$. The checksums are treated as part of the input array and included in the normal computations. Contrast this to the schemes described earlier, where the checksums are computed explicitly.

to appear in "Foundations of Dependable Computing: Vol III, System Implementation", Gary Koob editor, Kluwer Academic Publishers, 1

Another advantage of storing the checksums is that checking need not be performed after every iteration and yet errors during normal computation for any iteration can be detected. This is because any error during normal computation destroys the linearity property and invalidates some (or all) of the checksums; note that if this computation is performed in a single processor, a fault in it could cause error masking and keep the checksums validated which means that the error will not be detected. The checksums remain invalidated for subsequent iterations where there are no errors and (absence of) linearity is preserved. This can be detected when a checking operation is performed after some iteration. Two errors in two different iterations can mask each other and revalidate the checksums; however, the probability of such an event is low.

### *Memory Requirements*

The classic trade-off between memory required and time taken is evident in variable expansion for program restructuring and checksum storage. Variable expansion of a variable introduces linearity into statements, which can be exploited to reduce the checking time overhead, but increases the dimension of the variable by one, which increases storage requirements. For example, expansion of $A(I)$ to $A(I,J)$ adds $M*(N-1)$ elements to the original array $A$, where $M$ and $N$ are the upper bounds of the $I$ and $J$ loops respectively. Variable expansion is a practical optimization only as long as the added memory is within permissible limits. On a similar note, checksum storage to reduce computation time is feasible as long as the memory requirements are reasonable.

## 3.2.8 Implementation of a Compiler for Synthesis of Algorithm-Based Checks

In this section, we discuss the implementation of a compiler for automatic detection of linear statements in Fortran programs, restructuring of non-linear statements, and subsequent insertion of suitable code to perform on-line checking, based on the concepts discussed in the previous sections. This compiler is essentially an enhanced version of Parafrase-2 [15][20], an existing vectorizing/parallelizing compiler developed at the University of Illinois, which has been implemented as a source to source code restructurer. This compiler, termed CRAFT (for CompileR Assisted Fault-Tolerance) has been discussed by Balasubramanian in [14, 21]. The results of application of this package to two sample routines each from the LINPACK and EISPACK libraries, and to two Perfect

Benchmark routines, are presented. The results of experimental evaluation are presented on an Intel iPSC/2 hypercube.

## *Review of Parafrase-2*

Parafrase-2 provides a reliable, portable and efficient research tool for experimentation with program transformations and other compiler techniques for parallel supercomputers. One of the major features is the ability to allow different source languages as input to the compiler. This is accomplished by means of an intermediate representation used by the core of Parafrase-2 and hence transparent to the user. A property of the intermediate representation is the ability to recreate the source program. In order to make this intermediate representation transparent to the user, a preprocessor is used to transform each input language to the intermediate representation.

The input to Parafrase-2 is a set of data structures, organized in the form of a parse tree [22] so as to allow recreation of source code. This set of data structures is constructed from the input program by the preprocessor associated with the corresponding language. A *pass* of Parafrase-2 operates on the data structures to transform the program to a form suitable for parallel execution. Passes can be executed in a number of orders. This is achieved by insisting that the form of the system data structures be left invariant by each pass. The output of Parafrase-2 is the modified intermediate form, which is acted upon by a post-processor to produce the requisite output language.

Access to any fields of the standard data structures is exclusively via macros [20].

Of particular interest to us are Fortran assignment statements and the expressions and macros associated with them. As an example, the expression corresponding to the statement

$$K = K + L * M \tag{3.2.12}$$

is stored as the expression tree shown in Fig. 3.2.11.

Each node in the tree is an expression and points to zero or more children. The children are ordered and stored in a linked list, and are referred to as operands of the expression. For example, the operands of the parent expression of this tree are $K$ and $K + L * M$, where the second expression again has two operands.

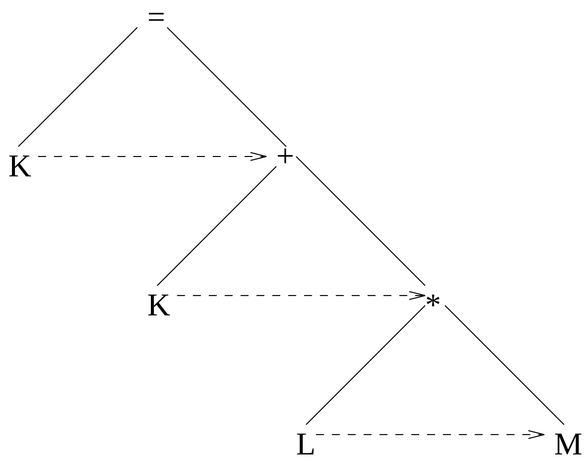



**Figure 3.2.11** Example expression tree

## *The LINTEST Pass*

We now describe the LINTEST pass, which performs automatic detection of linear statements in a given program followed by linearization of other statements. Since it operates actively only on statements inside DO loops, it needs information about the nesting level of each statement. This information is supplied by the 'donest' pass of Parafrase-2 [20]. LINTEST is executed in two phases or sub-passes. The first of these is the GETLIN phase, which goes through the entire program and identifies symbolically linear statements. The second phase uses the above information and calls one of the following two procedures, depending on the nature of the statement that is being currently processed: (1) the PROGLIN procedure, which determines whether a symbolically linear statement is program linear as well; (2) the MAKELIN procedure, which selectively linearizes a non-linear statement.

## *The GETLIN Phase*

This phase performs step-by-step detection of symbolic linearity for all assignment statements inside loops. The recursive nature of the expression tree corresponding to a particular statement suggests a direct application of the recursive linearity rules discussed earlier. Note that rule $R1$ also applies to assignment expressions i.e., $exp_1 = exp_2$ is linear in a particular variable if and only if $exp_1$ and $exp_2$ are individually linear in that variable. Each node of the

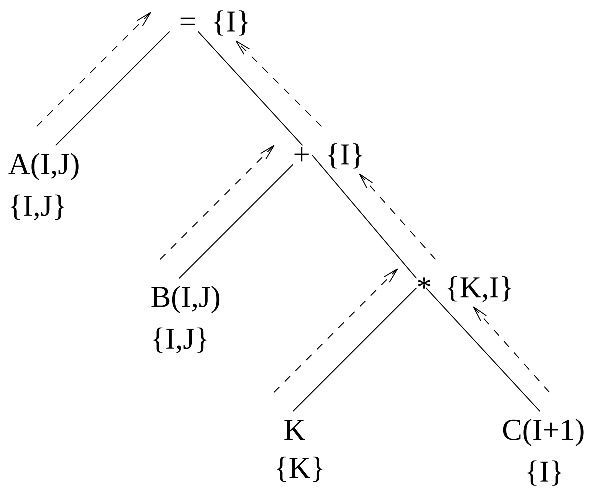



**Figure 3.2.12** Example expression tree for illustrating GETLIN phase

expression tree has a *linearity set* associated with it, which consists of all simple variables that the node expression depends on, including additional information about the linearity of this expression with respect to these variables. First, the linearity sets of the leaves are generated by using the linearity rules $R4 - R6$. This requires accesses to the leaf expressions which is performed in a top-down manner starting from the root of the expression tree. The linearity sets for the non-leaf nodes are generated by an appropriate *merge* of the linearity sets of its children. This merge depends on the type of expression associated with the node and the corresponding linearity rule to be selected from $R1 - R3$. Consider the following statement

$$A(I, J) = B(I, J) + K * C(I + 1) \qquad (3.2.13)$$

Fig. 3.2.12 illustrates the above bottom-up approach adopted for $I$-linearity detection of this statement.

Shown in the figure is a subset of the linearity set at each node, to indicate only the *linearity variables* i.e., the variables in which the expression is linear in.

The linearity set generated for the parent expression of each assignment statement, is saved for further use by the remaining phase of the LINTEST pass and by other passes. Another activity that is concurrently performed with symbolic linearity detection is the generation of an *outset* for each loop in the program, which consists of the output variables of all assignment statements inside that particular loop. These outsets are employed for program linearity detection, as will be clear from the description below.

### *The PROGLIN Procedure*

We return to the problem of determining which symbolically linear statements are also program linear. Recall that we introduced the notion of program linearity of a statement in subsection 3.2.6. The key step in the identification of program linearity of a symbolically linear statement is to determine whether the input variables to the statement are functions of the iteration variables (the loop indices) associated with loops that the statement is nested in. This, in the strictest sense, requires a detailed flow analysis of the entire loop body for each of these loops. Parafrase-2, although targeted to achieve this objective, currently does not implement certain necessary optimizations such as forward substitution, induction variable substitution, etc. However, we observe that, in general, output variables of assignment statements inside a particular loop do depend on that loop index, unless they are assigned to a constant expression; this is a relatively rare occurrence. Based on this observation, the outset of each loop is generated in the GETLIN phase itself. Any variable occurring in the outset of a particular loop is assumed to be dependent on the iteration variable of that loop.

The PROGLIN procedure operates on each statement that has been identified to be symbolically linear by the GETLIN phase. It uses linearity set of the statement currently under test, and the outsets of the various loops to determine the program linearity of the statement. The basic steps are as follows:

(1) For each linearity variable of the statement, access the body of the innermost loop in which the statement is nested in, to check for program linearity in relation to the corresponding loop index.

(2) Proceed only if the linearity variable is an induction variable i.e., is or depends on, the loop index. Otherwise, access the next surrounding loop, if present.

(3) Determine whether an input variable to the statement (other than the linearity variable) is, or depends on, the loop index.

(4) If the above condition is satisfied, check whether the input variable occurs independently of the loop index in the statement. If this is true, the statement

```
        K = 1
        DO I = 1,M                              [L,A,B,K]
            L = I*2                             {L,I}
            DO J = 1,N                          {A,B}
S1              A(I,J) = A(I,J) + K*C(I)        { I ,J,K}
S2              A(K,J) = B(K,J) + C(J)*D(K,L)   { K,J ,L}
            ENDDO
            K = K + 2                           {K}
        ENDDO
```

**Figure 3.2.13** Example loop for illustrating PROGLIN procedure

is not program linear in relation to the current loop index and the next outer loop is considered in turn.

(5) However, if the condition in (3) is not satisfied, check whether the arrays, if any, in which this variable occurs independently of the loop index, depend on the iteration variable. Again, only a positive result indicates that the statement is not program linear and the next loop is accessed.

To facilitate the execution of steps (4) and (5), each linearity set contains additional information about arrays in which the elements of the set are involved. This is obtained in the GETLIN phase.

Consider the loop structure of Fig. 3.2.13.

Indicated on the right corner are the linearity sets of the relevant assignment statements and the outsets of the two loops with the linearity variables in boldface type, Statement $S1$ is symbolically linear in $I$ (and not $J$, $K$), as shown. However, $K$ varies with $I$, which is indicated by its presence in the outset of the $I$-loop. In fact, $K = 2.I + 1$, since $K$ is incremented by two at the end of every iteration of the $I$-loop. Thus, statement $S1$ is not program linear in $I$. Statement $S2$, on the other hand, is program linear in both $K$ and $J$. Although $K$ and $L$ both depend on $I$, $L$ occurs in the *same* array as $K$, thus preserving the program linearity with respect to $K$. The linearity with respect to $J$ is not affected by the independent presence of either $K$ or $L$ (in array $D$), as both remain constant while the $J$-loop is being executed.

### *The MAKELIN Procedure*

This procedure operates actively on all assignment statements which are not symbolically linear, except those that satisfy two conditions simultaneously: (1) their R.H.S. expressions are non-linear, and (2) they write to scalars or non-linear expressions. In other words, if the L.H.S. of an assignment statement is a scalar or an array expression involving only constants (e.g. A(2)), then it will be ignored, unless its R.H.S. expression is linear in some variable; in such a case, the output variable is expanded to accommodate this *linearizing variable.* If the output variable is, however, a linear array expression, the R.H.S. expression is linearized with respect to the linearity variable. If multiple linearity variables exist in either of these cases, then the one associated with the innermost surrounding loop of the parent statement is chosen.

The linearizing technique adopted, directly exploits the recursive nature of the R.H.S. expression tree to search for the smallest sub-expression (expression associated with a node furthest away from the root) that when linearized, guarantees the linearity of the parent expression. At any stage, the transformation techniques $T1 - T4$, as discussed in subsection 3.2.4, are used to determine which of the two operands of the current expression needs to be selected for linearization. If this search process ends in a simple expression (i.e. at the leaf), variable expansion is performed. Otherwise, the selected expression is substituted by a single array involving the linearizing variable and other variables obtained from the inset of the substituted expression. This is followed by the generation of a new statement denoting the substitution. Note that this substitution is aborted if the expression so chosen is the R.H.S. expression itself.

This procedure is illustrated on the expression tree of Fig. 3.2.14, which corresponds to the following non-linear statement:

$$C(J) = C(J) + A(I,J) * B(J) * 2 \qquad (3.2.14)$$

Since the output variable is linear in $J$, the R.H.S. expression is chosen to be linearized with respect to $J$. The first operand of this sum expression viz., $C(J)$, is already linear in $J$ and so, the second operand is selected for linearization. The expression tree of this product term indicates that it is sufficient to linearize the first operand viz. $A(I,J) * B(J)$ since the other operand does not involve $J$. This follows from the linearity rule $R2$ described in subsection 3.2.3. At this stage, transformation technique $T2$ is directly applied to substitute the expression by an array expression involving $I$ and $J$ (shown as $TEMP(I,J)$ in

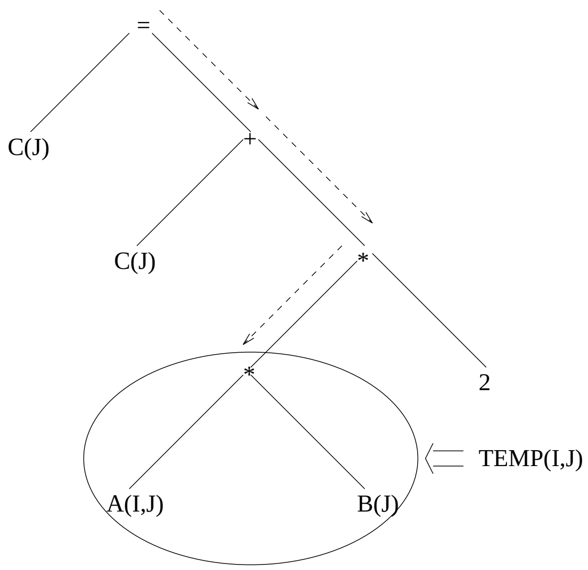



**Figure 3.2.14** Example expression tree for illustrating MAKELIN phase

the figure). The pair of statements, thus, generated are

$$TEMP(I,J) = A(I,J) * B(J) \tag{3.2.15}$$

$$C(J) = C(J) + TEMP(I,J) * 2 \tag{3.2.16}$$

Variable expansion or expression substitution requires modification of the expression tree corresponding to the original statement. List manipulation and expression building functions, provided by Parafrase-2 [20], are used here. The actual modification is carried out in the following manner:

(1) First, the operand list of the new array is appropriately generated to include the linearizing variable as well as the input variables of the the expression to be expanded or substituted.

(2) Next, the expression corresponding to the array is constructed by using the above operand list and the symbol table pertaining to the symbol of this array. Note that substitution might first require the generation of a new symbol table entry for the array.

(3) Finally, a new statement to denote the substitution is constructed and appropriately included within the program.

The linearized statement at the end of step (2) is tested for program linearity by re-executing the various steps of the PROGLIN procedure. If it fails the test, the MAKELIN procedure is termed to be unsuccessful and the modification of step (2) is undone. Also, step (3) is no longer executed.

From the description of the MAKELIN phase, thus, it is apparent that information about the linearity of the sub-expressions at various nodes of the statement expression tree, as well as the their insets, is needed. This is generated by using the appropriate routines of the GETLIN phase, as and when required.

## *The ADDCHECK Pass*

The ADDCHECK pass operates on the linearized version of the original program and generates suitable check code for providing on-line detection of errors that may occur during actual execution of the program. This pass is placed after the LINTEST pass in the passlist. Currently, the pass operates actively on all program linear and program linearized statements that have at least one linearity variable occurring as the iteration variable of some surrounding loop. In the event that multiple linearity variables exist, the one associated with the innermost surrounding loop is selected to act as the checking variable. Thus, each loop having the checking variable as its index plays a key role in this pass and will be subsequently referred to as the *check loop*.

The check code for each program linear (or linearized) statement, includes three sets of statements: (1) *summation statements*, each denoting the summation of the various values of an array variable (or scalar) involving the linearity variable, for different values of this iteration variable, (2) *check statements*, that denote the modified computation of the output using these summations, and the comparison that follows, and (3) *initialization statements*, that denote the proper initialization of the summation variables before they are used to store the summation values. We now describe the generation of each set.

### *Summation Statements*

The generation of summation statements first involves identification of the leaves of the statement expression tree and selection of only those expressions that involve the linearity variable. For each such expression (represented generically by *exp*), a summation statement is built and included in the program, as follows:

(1) Assume for the moment that *exp* is an array. First, a suitable operand list for the summation variable is created. This consists of all variables that occur along with the linearity variable in *exp*, but which are associated only with loops inside the check loop. Note that this operand list would be empty if *exp* was just a scalar, or if the check loop was the innermost loop itself as is true for many of the loop structures encountered in actual programs.

(2) Next, an expression corresponding to the new summation variable, say *sum*, is built. The nature of the expression building function used for this purpose depends on the status of the operand list created above. In other words, if the operand list is empty, a scalar expression is created; otherwise the resulting expression is an array with the given operand list.

(3) Using the *exp* and *sum* expressions, a summation statement of the form $sum = sum + exp$ is built. Note that the occurrences of *sum* on the L.H.S. and on the R.H.S. actually correspond to different expressions that share a common symbol table entry. This requires a duplication of the *sum* expression created in (2), before the statement can be built.

(4) Finally, the summation statement is inserted just before or immediately after the linear statement in question, depending on whether *exp* corresponds to an input variable or an output variable, respectively. Such an insertion guarantees that the linear statement shares all its parent conditional statements with the summation statement, so that there is a one-to-one correspondence between the executions of the linear and summation statement. For example, if the linear statement was to be executed only during certain iterations of the check loop, then the summation statement would also be executed only in these iterations, thus, preserving the validity of the summation values. Another major advantage in having the summations thus placed is that variable expansions can be totally avoided in the MAKELIN procedure, since the the value of each such variable in the current iteration of the check loop will be included in the corresponding summation before it acquires a possible new value for the next iteration.

Fig. 3.2.15 shows the check code inserted by the ADDCHECK pass for statement $S2$ in the loop structure of Fig. 3.2.13.

Note that $J$ has been employed as the checking variable, since it is associated with the inner loop. Statement $S2$ has three array expressions that involve

```
        DO I = 1,M
        . . .
I1          SUM0 = 0
I2          SUM1 = 0
I3          SUM2 = 0
            DO J = 1,N
                . . .
SM1             SUM1 = SUM1 + B(K,J)
SM2             SUM2 = SUM2 + C(J)
S2              A(K,J) = B(K,J) + C(J) * D(K,L)
SM3             SUM0 = SUM0 + A(K,J)
            ENDDO
C1          SUM3 = SUM1 + SUM2 * D(K,L)
C2          CALL COMPARE(SUM3, SUM0)
            . . .
        ENDDO
```

**Figure 3.2.15** Illustration of the ADDCHECK pass on the loop of Fig. 3.2.13

$J$ viz., $A(K, J)$, $B(K, J)$, and $C(J)$. Hence, three summation statements are inserted, each denoting the summation of values of one of the three arrays for $1 = J <= N$. These are numbered $SM1 - SM3$ in the figure.

*Check and Initialization Statements*

We now describe the construction of the check statements. The first of these denotes the alternate computation of the output summation using the input summations. The structure of this statement is similar to that of the original linear statement, but for the replacement of each input array (or scalar) involving the checking variable with the corresponding summation variable, and the presence of a new summation variable at the output. This is shown in Fig. 3.2.15 where check statement $C1$ is obtained from $S1$ by replacing $B(K, J)$ and $C(J)$ with $SUM1$ and $SUM2$, respectively. The recursive generation of the expression tree for this statement is performed concurrently with the exploration of the original expression tree, which is needed anyway to generate the summation statements. The next check statement is actually a subroutine call for comparison of the two output summations to check for possible mismatches. This is indicated as $C2$ in the figure. It is possible that the check loop is not the innermost loop in a multiply nested loop structure, in which case, all loops inside the check loop that surround the linear statement, need to be duplicated and placed around the check statements to ensure the validity of the comparisons. Such a situation is illustrated in the next section which discusses the check code for matrix multiplication.

The generation of initialization statements is fairly simple. This involves a duplication of the expressions corresponding to the various summation variables so that they may be used in the construction of assignment statements having their R.H.S. expressions set to the constant, zero. These statements are placed before the check loop and are indicated as $I1 - I3$ in the figure. As in the case of check statements, all loops occurring inside the check loop that affect the linear statement must be duplicated and placed around the initialization statements.

## *Example Application of Passes to Matrix Multiplication*

We now illustrate the LINTEST and ADDCHECK passes on the node program for parallel matrix multiplication on the Intel iPSC/2 hypercube. Fig. 3.2.16 shows the relevant code for the node program obtained by using $I$ as the paral-

```
        DO I = 1,M  / P
            DO J = 1,N
                DO K = 1,L
S1                  C(I,J) = C(I,J) + A(I,K) * B(K,J)
                ENDDO
            ENDDO
        ENDDO
```

**Figure 3.2.16** Parallel node program for matrix multiplication

lelization variable in the program of Fig. 3.2.5. Note that each processor would execute $M/P$ iterations of the $I$-loop.

The assignment statement $S1$ is identified to be symbolically $I+J$-linear by the GETLIN phase; the recursive linearity rules $R1$ and $R2$ are used here. This statement is then acted upon by the PROGLIN procedure, which concludes that it is also program linear in both $I$ and $J$ since the only output variable occurring in this loop structure viz., the $C$ array, involves both $I$ and $J$. Note that the MAKELIN procedure is not employed at all.

Fig. 3.2.17 shows the same program with on-line summation, check and initialization statements.

The scheme employs $J$ as the checking variable. The summation statement for $B(K,J)$ viz., $SM1$, is generated and inserted. The values stored in the $SUMK$ array at the end of the $J$-iterations are nothing but the row checksums of the $B$ array. The procedure for $C(I,J)$ is, however, modified to exploit a special property of the loop structure viz., repeated update and write operations on the $C$ array. A summation statement is generated for *only the output* $C(I,J)$ and is placed *outside* the $K$-loop as indicated by $SM2$. Hence, errors during the actual execution of the entire $K$-loop (for a particular value of $J$), will be reflected in the value stored in $SUM0$. In fact, this method aims to limit the amount of checking that needs to be done, and is, therefore, a frequency-related optimization. Note that after the $J$-loop has been completely executed, the values stored in the $SUM2$ array are nothing but the row checksums of the $B$ matrix; $SUM1$ holds the row checksum of the $I$th row of the $C$ submatrix.

```
        . . .
        DO I = 1,M/P
            DO K = 1,L
I1              SUM2(K) = 0
            ENDDO
I2          SUM0 = 0
            DO J = 1,N
                DO K = 1,L
SM1                 SUM2(K) = SUM2(K) + B(K,J)
S1                  C(I,J) = C(I,J) + A(I,K)*B(K,J)
                ENDDO
SM2             SUM0 = SUM0 + C(I,J)
            ENDDO
            DO K = 1,L
C1              SUM1 = SUM1 + A'(I,K)*SUM2(K)
            ENDDO
            CSEND(...SUM1...)
            CRECV(...SUM1'...)
C2          CALL COMPARE(SUM1',SUM0)
        ENDDO
        . . .
```

**Figure 3.2.17** Parallel node program for matrix multiplication with checks

The procedure for the check statements is modified accordingly. Statement $C1$ is obtained from $S1$ by replacement of $B(K, J)$ with its summation variable, and the output and input $C(I, J)$'s by a new variable. It is then placed outside the $J$-loop with a $K$-loop around it. The complete execution of the $K$-loop would then result in an alternate computation of the output checksum viz., the R.H.S. computation as per the notation adopted in subsection 3.2.5. Since, the program under consideration is a node program, $A(I, K)$ is replaced by $A'(I, K)$, where the $A'$ array stores the $A$ submatrix values for a neighboring processor. This results in the computation being performed for this neighbor. After an exchange of the R.H.S. computation values with the neighbor, the comparison routine is finally called.

The initialization statements $I1$ and $I2$ are inserted as shown in the figure and are obtained by a straightforward implementation of the steps discussed in the previous section. It is worthwhile mentioning here that the complete check code is placed inside the $I$-loop and is, therefore, executed repeatedly for all the $I$-iterations.

### 3.2.9 Results of Applying the Compiler on Real Programs

This section presents the results of application of the compiler to real scientific Fortran programs. Six sample routines from the LINPACK and EISPACK libraries, and from the Perfect Club Benchmark Suite, are considered first. These routines are essentially sequential programs meant to run on uniprocessors. However, results for sequential programs can be expected to carry over to their parallel versions since the fraction of linearizable statements remains relatively unchanged. We then consider parallel versions of two of the above programs developed for a hypercube multiprocessor, by running the compiler on the corresponding node programs.

#### *Results on Sequential Programs*

Table 3.2.1 shows the results of applying the compiler passes for detection and generation of linear statements to six application programs. The first two routines are from the LINPACK library. DGEFA, factors a real matrix by Gaussian Elimination, which is then used by DGESL to solve a corresponding system of linear equations. The results presented here are for expanded versions of the above two routines. This was done to expose the linearity property of

**Table 3.2.1** Linearity characteristics of six application programs

| Program | Number of Statements | | | | | | % lin-checkable | |
|---|---|---|---|---|---|---|---|---|
| | Exec | Assign | Lin | Pglin | Mklin | Pslin | Static | Dynamic |
| DGEFA | 43 | 16 | 4 | 4 | 5 | 5 | 56.3 | 98.0 |
| DGESL | 49 | 18 | 3 | 3 | 9 | 9 | 66.7 | 74.2 |
| TRED2 | 92 | 46 | 12 | 12 | 16 | 16 | 60.9 | 66.4 |
| TQL2 | 84 | 47 | 3 | 2 | 14 | 14 | 34.0 | 95.0 |
| TRFD | 67 | 23 | 3 | 2 | 1 | 0 | 12.5 | 57.4 |
| INTGRL | 28 | 11 | 0 | 0 | 2 | 2 | 18.2 | 38.1 |
| OLDA | 100 | 35 | 6 | 6 | 13 | 13 | 54.3 | 97.1 |
| MDG | 100 | 6 | 2 | 2 | 2 | 2 | 66.7 | 99.9 |
| MDMAIN | 61 | 11 | 3 | 3 | 0 | 0 | 27.3 | 30.0 |
| BNDRY | 13 | 6 | 0 | 0 | 6 | 6 | 100.0 | 100.0 |
| INITIA | 97 | 34 | 0 | 0 | 12 | 12 | 35.3 | 41.4 |
| CORREC | 10 | 4 | 0 | 0 | 2 | 2 | 50.0 | 50.0 |
| INTRAF | 59 | 47 | 4 | 4 | 5 | 4 | 18.8 | 29.7 |
| INTERF | 108 | 81 | 9 | 9 | 2 | 2 | 13.6 | 11.0 |
| POTENG | 69 | 43 | 0 | 0 | 2 | 2 | 4.7 | 10.9 |

certain statements inside the expanded subroutines, so that they could be easily detected by the compiler. Next we have a routine form EISPACK , TRED2, which reduces a real symmetric matrix to a symmetrical tridiagonal matrix using and accumulating orthogonal similarity transforms. This matrix is then used by the EISPACK routine, TQL2, to find the eigenvalues and eigenvectors. The remaining two programs are from the Perfect Club Benchmark Suite. The program TRFD simulates the computational aspects of a two-electron integral transformation. It employs two subroutines, INTGRL and OLDA, the results for which are also provided in the table. The code for MDG provides a molecular dynamics model for water molecules in the liquid state at room temperature and pressure. Again, the results for most of the important subroutines used in these programs are also are shown in the table.

The next six columns of the table show the results obtained at the end of the LINTEST pass for each of the above six programs along with their subroutines, if present. For a particular program:

(1) *Exec* indicates the total number of executable statements inside the program. Hence, declarations and other similar statements are excluded.

(2) *Assign* denotes the number of assignment statements present within loops. These are the key statements for the LINTEST pass.

(3) *Lin* indicates the number of assignments (out of (2)) that were detected to be symbolically linear by the GETLIN phase.

(4) *Pglin* pertains to candidate statements for the PROGLIN procedure, and shows the number of symbolically linear statements that are program linear as well.

(5) *Mklin* represents the results of the MAKELIN procedure, and shows the number of statements from (2) (but not in (3)) that were transformed to be symbolically linear and were program linear as well.

(6) *Pslin* denotes the number of statements (out of (5)) that are attributed to variable expansions. Such expansions can be avoided because of the way in which the summations are carried out during actual checking. Hence, these statements represent a special class of non-linear statements that are amenable to checking without the necessity for any restructuring, and are called *psuedo-linear*.

The last two columns portray the combined picture that emerges from the above six columns. These results are a measure of the amount of linearity that is present in, or can be successfully extracted from, the original programs using our compiler. Here, the term *lin-checkable* is used to denote all such statements which lend themselves to linearity-based checking. Note that statements that are not lin-checkable can be checked by the simple duplication-and-comparison approach. The total number of lin-checkable statements per row is nothing but the total number of program linear and linearized statements ( (3) + (4) ) for the same row. Each number in the *static* column is obtained by dividing this number by the total number of assignments statements that were present initially or created during substitutions ( (2) + (5)-(4) ). In other words, the static results show the total number of lin-checkable statements relative to the total number of assignment statements in the final linearized versions of the original programs, and are, thus, indicative of the inherent structure of these programs.

The dynamic results present a more realistic picture as they are an estimate of the total number of executions of all lin-checkable statements with respect to the total number of executions of all assignment statements, for a given input data size. Hence, programs with a high dynamic percent count would most likely have a high percentage of lin-checkable statements inside multiple loops. This is indeed true for highly structured programs such as the DGEFA, TQL2, and OLDA routines, which show a dramatic rise in their dynamic counts when compared to the static counts. In fact, the OLDA subroutine dominates the execution of the entire TRFD program with the result that the net dynamic count is around 93%.

The above values allow us to make the following inferences. Since any error that occurs during the execution of a lin-checkable statement is detected by the corresponding check code with a high probability, one can expect programs having reasonably high dynamic counts (the first five programs) to give high error coverages while executing on real error-prone machines. Correspondingly, programs with very low dynamic counts such as some subroutines in the MDG program cannot be expected to give high error coverages, if linearity-based checking were the sole concurrent error detection scheme to be used. The coverages for on-line error detection of such programs can be boosted up by the intelligent use of duplication techniques. The results for DGEFA, DGESL and TQL2 were estimated using an input matrix size of 100, whereas an input size of 1000 was used for the TRED2 routine. The parameter values for the TRFD and MDG routines were defaults provided by the Benchmark Suite.

**Table 3.2.2** Linearity characterisitcs of two parallel node programs for iPSC/2

| | Number of statements | | | | | | % lin-checkable | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Dynamic | | |
| Program | Exec | Assign | Lin | Pglin | Mklin | Pslin | Static | P=4 | P=8 | P=16 |
| PDGEFA | 72 | 25 | 5 | 5 | 10 | 10 | 60.0 | 93.3 | 89.1 | 84.1 |
| PTRED2 | 362 | 138 | 8 | 8 | 32 | 32 | 29.0 | 58.9 | 56.5 | 52.4 |

## *Results for Parallel Programs*

This section presents results for parallel versions of the DGEFA and TRED2 routines. The former is the Fortran version of the parallelized DGEFA program which has been implemented on an Intel iPSC-2 hypercube and is described in [14]. The program for the TRED2 routine is a parallel implementation on the iPSC-2 hypercube using a row cyclic distribution of the input matrices, and is described in [23]. Table 3.2.2 shows the results for the processor node programs viz., PDGEFA and PTRED2, presented in a manner similar to the previous table.

Since DGEFA is a highly structured program as mentioned in the previous section, the static count for PDGEFA remains more or less the same indicating the fact that the linearity of the program as a whole has not been affected by parallelization. The dynamic count drops down a little bit as expected. Since the program parallelization causes the number of iterations of the parallel loop to decrease by a factor equal to the number of processors, the relative number of executions of lin-checkable statements inside the parallel loop also goes down by the same factor. Parallelization of the TRED2 routine, however, introduces lots of non-linear statements because of the somewhat loosely structured nature of the original program. Hence, the static percent is decreased quite a bit as is evident from the Tables 3.2.1 and 3.2.2. The drop in the dynamic count is comparatively lower, as many of these non-linear statements occur outside the most frequently executed loops in the program.

## *Results on Error Coverages and Time Overheads*

In order to observe error coverage results of the compiler-assisted data integrity checking scheme, a version of the DGEFA routine with data integrity checks

**Table 3.2.3** Percent error coverage in the DGEFA checking scheme

| Error type | Unit | Number of processors 4 | 8 | 16 |
|---|---|---|---|---|
| Transient bit error | adder | 92.97 | 91.41 | 88.28 |
| | multiplier | 86.72 | 89.84 | 84.38 |
| Transient word error | adder | 100.00 | 100.00 | 100.00 |
| | multiplier | 100.00 | 100.00 | 100.00 |
| Permanent bit error | adder | 92.97 | 92.97 | 90.63 |
| | multiplier | 89.84 | 89.84 | 86.72 |
| Permanent word error | adder | 100.00 | 100.00 | 100.00 |
| | multiplier | 100.00 | 100.00 | 100.00 |

was implemented for the Intel iPSC/2 hypercube multicomputer. Experiments were run on randomly generated 100x100 matrices for hypercubes with 4, 8 and 16 processors. Errors were injected at the source code level and could be of four types - bit-level transient errors, bit-level permanent errors, word-level transient errors and word-level permanent errors. The errors were assumed to affect either floating point additions or floating point multiplications. Injecting these errors at the source code level involved replacing the output of every floating point addition or multiplication by a call to an appropriate error injection routine, which would then corrupt the result in accordance with the error being simulated. Details of this method may be found in [8, 24]. Error coverages are illustrated in Table 3.2.3. These are quite high and in particular are always 100% for word-level errors, which can be expected to cause more data corruption than bit-level errors.

Time overheads are illustrated in Fig. 3.2.18. Timing overheads decrease with an increase in problem size since normal computations increase much more rapidly than check computations as problem size increases.

### 3.2.10 Summary

In this section, we have addressed the task of synthesizing algorithm-based checking techniques for general applications. We have approached the problem at the compiler level by identifying linear transformations in Fortran DO loops, restructuring program statements to convert non-linear transformations

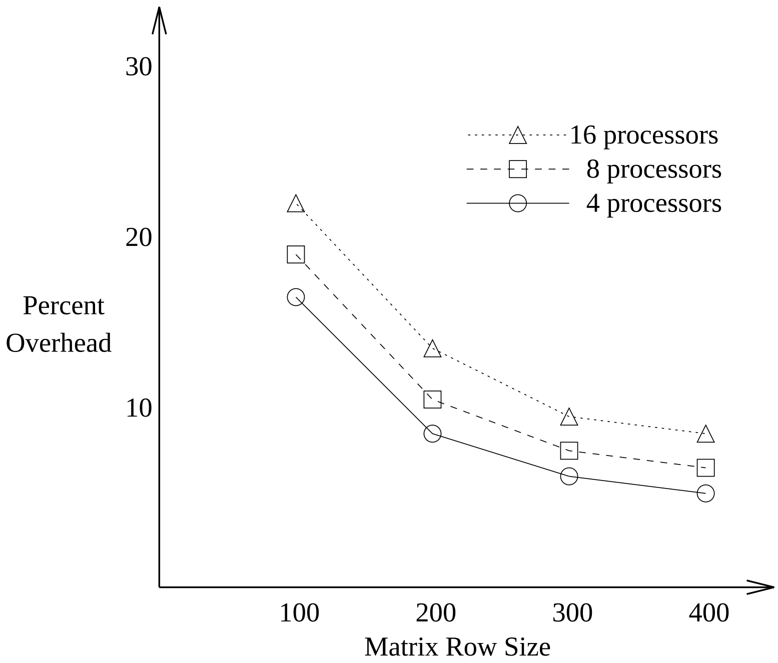



**Figure 3.2.18** Time overhead of the DGEFA checking scheme

to linear ones, and have proposed system-level checks based on this property. We have illustrated our approach on the example problem of matrix multiplication. We have discussed certain trade-offs involved in algorithm-based checking which are related to the nature and extent of the checksum encodings.

Next, we have discussed the implementation of a source-to-source restructuring compiler that can synthesize suitable low-cost checks for providing on-line detection of errors occurring during execution of a variety of numerical Fortran programs. This compiler consists of two passes. The first of these is the LINTEST pass which performs automatic detection of linear statements and selective linearization of others. The second pass is the ADDCHECK pass which performs actual insertion of check code using the information generated in the previous pass. We have illustrated these passes on the node program for matrix multiplication on the hypercube to generate a version with on-line checks. We have then presented the results of application of this compiler to six important routines from the LINPACK and EISPACK libraries, and the Perfect Benchmark Suite. We have also presented error coverage and time overhead results for a selected parallel prgram.

# REFERENCES


[1] A. L. Hopkins, I. T. B. Smith, and J. H. Lala, "FTMP: A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE*, vol. 66, pp. 1221–1239, Oct. 1978.

[2] D. P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Hsao, "A Case Study of C.mmp, Cm*, and C.vmp. I. Experiences with Fault Tolerance in Multiprocessor Systems," *Proc. IEEE*, vol. 66, pp. 1178–1199, Oct. 1978.

[3] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE*, vol. 66, no. 10, pp. 1240–1255, Oct. 1978.

[4] D. P. Siewiorek and R. S. Swartz, *Theory and Practice of Reliable System Design.* Bedford, MA: Digital Press, 1982.

[5] O. Serlin, "Fault-Tolerant Systems in Commercial Applications," *IEEE Computer*, pp. 19–30, August 1984.

[6] D. Johnson, "The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems," *IEEE Computer*, pp. 40–48, August 1984.

[7] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, pp. 518–528, June 1984.

[8] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Trans. Comput.*, vol. 39, pp. 1132–1145, September 1990.

[9] J.-Y. Jou and J. A. Abraham, "Fault-tolerant matrix operations on multiple processor systems using weighted checksums," *SPIE Proceedings*, vol. 495, August 1984.

[10] F. Luk, "Algorithm-Based Fault Tolerance for Parallel Matrix Solvers," *Proc. SPIE Real-Time Signal Processing VIII*, vol. 564, 1985.

[11] M. Malek and Y. H. Choi, "A fault-tolerant fft processor," *IEEE Trans. Comput.*, May 1988.

[12] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection techniques in signal processing applications," *IEEE Trans. Comput.*, vol. 39, pp. 1304–1308, October 1990.

[13] R. B. Mueller-Thuns, D. McFarland, and P. Banerjee, "Algorithm-Based Fault Tolerance for Adaptive Least Squares Lattice Filtering on a Hypercube Multiprocessor," *Proc. Int. Conf. on Parallel Processing*, pp. 177–189, Aug. 1989.

[14] V. Balasubramanian, "*The Analysis and Synthesis of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors*." Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, February 1991. Tech. Report no. CRHC–91–6, UILU–ENG–91–2210.

[15] W. Harrison, "*An Overview of the Structure of Parafrase*." Univ. of Illinois, Urbana-Champaign, July 1985. CSRD Tech. Report no. 501, PR-85-2, UILU–ENG–85–8002.

[16] B. Leasure, "*The Parafrase project's Fortran analyzer major module documentation*." Univ. of Illinois, Urbana-Champaign, July 1985. CSRD Tech. Report no. 504, PR-85-5, UILU–ENG–85–8005.

[17] C. D. Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design," *IEEE Trans. Comput.*, vol. 37, pp. 991–1004, August 1988.

[18] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, pp. 1184–1201, December 1986.

[19] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Trans. Comput.*, vol. C-36, pp. 1485–1495, December 1987.

[20] C. D. Polychronopoulos, M. Girkar, M. R. Hagighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2 manual." CSRD, Univ. of Illinois, Urbana-Champaign, 1990.

[21] V. Balasubramanian and P. Banerjee, "CRAFT: A Compiler for Synthesizing ALgorithm-Based Fault Tolerance in Hypercube Multiprocessors," *Proc. Int. Conf. Parallel Processing (ICPP-91)*, Aug. 1991.

[22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison-Wesley, 1988.

[23] M. Gupta, "Automatic data partitioning on distributed memory multiprocessors." Univ. of Illinois, Urbana-Champaign, October 1990. Tech. Report no. CRHC–90–14.

[24] A. Roy-Chowdhury, "*Evaluation of Algorithm Based Fault-Tolerance Techniques on Multiple Fault Classes in the Presence of Finite Precision Arithmetic.*" M.S. Thesis, Univ. of Illinois, Urbana-Champaign, August 1992. Tech. Report no. CRHC–92–15, UILU–ENG–92–2228.