MANUAL AND COMPILER ASSISTED METHODS FOR GENERATING FAULT-TOLERANT PARALLEL PROGRAMS

BY

AMBER ROY-CHOWDHURY

B.Tech., Indian Institute of Technology, Kharagpur, 1990
M.S.,University of Illinois, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

MANUAL AND COMPILER ASSISTED METHODS
FOR GENERATING ERROR DETECTING PARALLEL PROGRAMS

Amber Roy-Chowdhury, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Prithviraj Banerjee, Advisor

Algorithm-based fault-tolerance (ABFT) is an inexpensive method of incorporating fault-tolerance into existing applications. Applications are modified to operate on encoded data and produce encoded results which may then be checked for correctness. An attractive feature of the scheme is that it requires little or no modification to the underlying hardware or system software. Previous algorithm-based methods for developing reliable versions of numerical programs for general-purpose multicomputers have mostly concerned themselves with error detection. A truly fault-tolerant algorithm, however, needs to locate errors and recover from them once they have been located. In a parallel processing environment, this corresponds to locating the faulty processors and recovering the data corrupted by the faulty processors. In this dissertation, we first present a general scheme for performing fault-location and recovery under the ABFT framework. Our fault model assumes that a faulty processor can corrupt all of the data it possesses. The fault-location scheme is an application of system-level diagnosis theory to the ABFT framework, while the fault-recovery scheme uses ideas from coding theory to maintain redundant data and uses this to recover corrupted data in the event of processor failures. Results are presented on implementations of three numerical algorithms on a distributed memory multicomputer, which demonstrate acceptably low overheads for the single- and double-fault location and recovery cases.

For a class of algorithms performing affine transformations, we automate the process of generating an error-detecting version at compile time. The compiler is used to identify loops that perform affine transformations on array elements. These loops are then checked by computing a checksum over the array elements being transformed and transforming the checksums appropriately, which typically results in much smaller overheads than checking the entire code by duplication. Portions of code in the program that are not affine transformations are checked by duplication. An existing source-to-source compiler, Parafrase-2, has been modified to take in programs written in High Performance Fortran (HPF) and output an error-detecting version of the same. Data distributions for the new arrays and checksums introduced are specified by inserting additional HPF directives in the program. The modified program can then be input to a parallelizer for distributed memory machines, such as PARADIGM, to obtain an error-detecting parallel program. We demonstrate

results on three numerical programs by executing the error-detecting versions generated by our compiler on a distributed memory multicomputer.

DEDICATION

To my parents

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Prithviraj Banerjee. His excellent guidance, easy accessibility, and congenial nature made it a pleasure and a privilege to work for him. Thanks also to Professors W. Kent Fuchs, Ravi Iyer, Michael Loui and Constantine Polychronopoulos for serving on my committee and providing useful comments and suggestions.

I am also grateful for the support of the Office of Naval Research and the Joint Services Electronics Program, who have funded this research.

Over the five years I spent at Illinois as a graduate student, my co-workers moved from being just that to becoming good friends. In particular, John Chandy, Buck Hodges, Antonio Lain, Dan Palermo, Shankar Ramaswamy and Ernesto Su were invaluable in providing technical help.

Carolin Tschopp was efficient and cheerful in providing secretarial help often beyond the call of duty, and for this I would like to thank her.

Lastly, I would like to thank my parents for affording me the opportunity to avail of the best education available, for the sacrifices they made, and for their unquestioning love and affection.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Massively parallel computers are being increasingly used to solve numerical problems with extremely large problem sizes. Despite the enormous computing power provided by these machines, the problem sizes are often so large that hours or even days may pass before a solution is obtained. Due to the large amounts of hardware involved, failures during the course of a computation are becoming increasingly likely. Some fault-tolerance measures are therefore needed to handle these failures.

Algorithm-based fault tolerance (ABFT) is a method in which the algorithm is modified to detect errors introduced by faults in the underlying hardware. In many cases, it is possible to achieve fault tolerance with no modifications to the hardware or system software. Also, ABFT may be used to make an algorithm execute reliably on a computer that provides little support for fault tolerance in the hardware or system software. ABFT is also very effective in detecting transient or intermittent faults through their effects on data computed by the algorithm. These types of faults may be hard or impossible to detect through off-line testing. The basic approach is to apply some encoding to the data being operated on by the algorithm, modify the encoded data concurrently with the original data, and check that the encoding is preserved at various points during the execution of the algorithm.

As an example, consider the problem of matrix multiplication $C = AB$. The simple algorithm may be made error-detecting by the addition of an extra row to A that is computed by taking the sum of all other rows of A. The product of the extra row of A with B yields an extra row in the product matrix C, which should equal the sum of all of the other rows of C in the absence of errors (due to roundoff errors, a small tolerance has to be allowed in the comparison). This matrix multiplication is illustrated in Fig. 1.1. This idea can be extended in an obvious manner in a multiprocessor environment with each processor checking the data on another processor, as is illustrated in Fig. 1.2. Note that for $n \times n$ matrices, only $O(n^2)$ operations are required for creating, manipulating, and comparing checksums, while $O(n^3)$ operations are used to compute the matrix multiplication.

ABFT versions have been developed for several numerical algorithms. Early ABFT work involved modifying numerical algorithms executing on systolic hardware. These schemes required

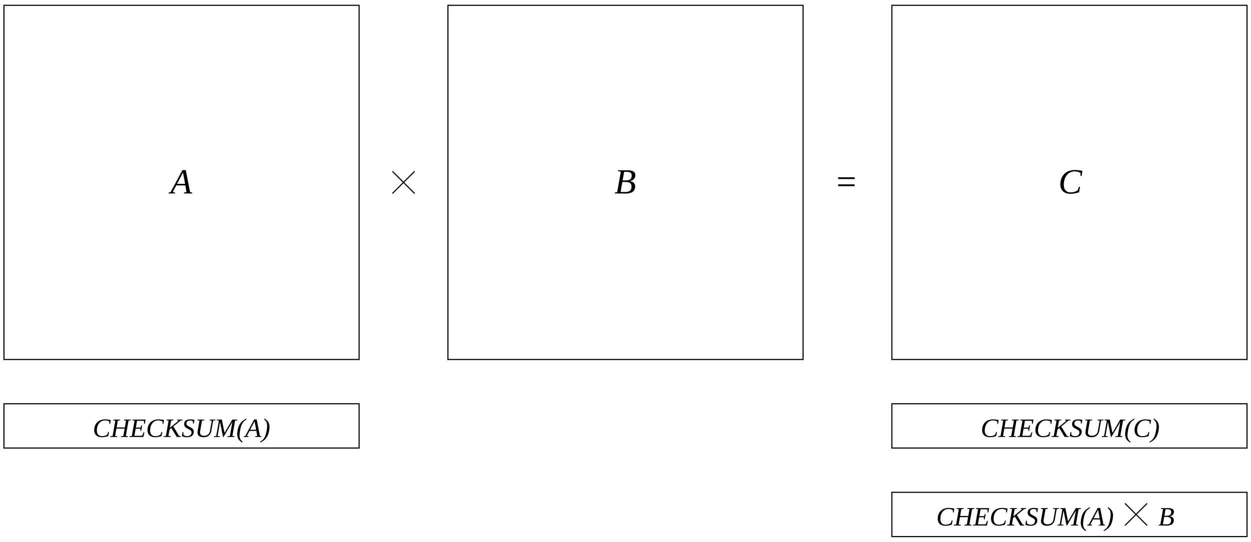


Figure 1.1 Illustration of matrix multiplication with checksums for error detection.

the addition of extra hardware to the systolic array to perform computations on redundant data [1, 2, 3, 4]. Later, ABFT schemes were designed for parallel algorithms executing on general-purpose multicomputers that required no modification of the underlying hardware [5, 6]. Although some of the ABFT schemes for systolic hardware possessed some limited capability for error location and correction, most of the suggested schemes for general-purpose multiprocessors have been confined to error detection only. A few schemes for error location and correction [7, 8, 9] have been suggested, but many of these are quite theoretical in nature, are targeted toward systolic algorithms, or suffer from other drawbacks, and would be difficult to implement on a real multiprocessor system. Also, none of these schemes give a general methodology for error location as well as error correction for an arbitrary number of errors.

We attempt to address this gap by proposing fault-location and -recovery schemes that are easily applied to the ABFT framework. The fault-location strategy is an application of system-level diagnosis theory to the ABFT framework. The fault-recovery strategy is an application of coding theory to maintaining redundant data, which is used to recover corrupted data if faults occur. We demonstrate the practicality of our ABFT schemes by presenting results on implementations of the three parallel matrix algorithms modified to perform single-fault location and recovery on a distributed-memory multicomputer. To demonstrate that even the multiple-fault location and recovery problem need not impose inordinately large overheads, we present results for the multiple-fault recovery case for one parallel matrix algorithm.

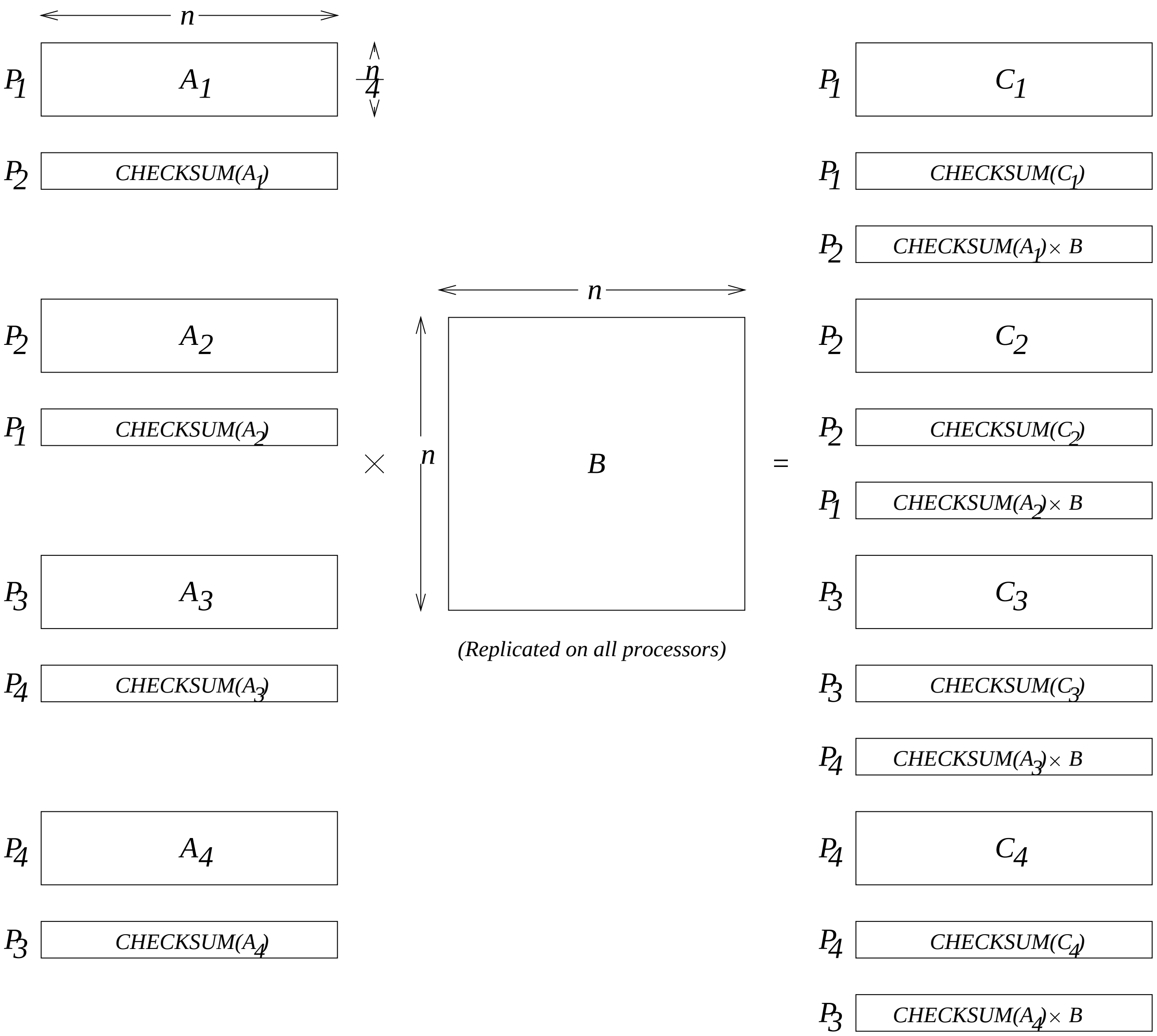


Figure 1.2 Parallel implementation of matrix multiplication with checksums for error detection.

For a class of algorithms performing linear or affine transformations on the data, a natural encoding to choose is the checksum encoding [2], where a checksum is computed of the data being operated on by the algorithm. The checksum is then transformed concurrently with the computations on the data elements, and at suitable points during the execution, the data elements are summed and compared with the transformed checksum.

We have developed an automated, compile-time approach for generating error-detecting parallel programs based on the above idea. The compiler is used to identify statements implementing affine transformations within the program and automatically insert code for computing, manipulating, and comparing checksums in order to check the correctness of the code implementing affine transformations. Statements that do not implement affine transformations are checked by duplication. Checksums are reused from one loop to the next if this is possible, rather than recomputing checksums for every statement. A global dataflow analysis is performed in order to determine points at which checksums must be recomputed. We also use a novel method of specifying the data distributions of the check data using directives provided by the High Performance Fortran (HPF) [10] standard so that the computations on the original data and the corresponding check computations are performed on different processors.

The rest of this dissertation is organized as follows. Chapter 2 discusses prior work in the areas of ABFT and compiler-assisted generation of error-detecting programs. Chapter 3 discusses an extension of the algorithm-based fault tolerance methodology to perform multiple-fault location and recovery in a parallel processing environment. Chapter 4 describes the modification of three numerical algorithms using the ideas of Chapter 3 in order to make them fault tolerant and presents overhead and fault-coverage results on implementations of the fault-tolerant algorithms on a distributed-memory multicomputer. Chapter 5 discusses the design and implementation of our automated compile-time approach for generating error-detecting parallel programs. Chapter 6 describes three programs that were used to demonstrate the generation of error-detecting code by our compiler and presents overhead results on implementations of these on a distributed-memory multicomputer. Finally, we present conclusions and future work in Chapter 7.

CHAPTER 2

RELATED WORK

We describe related work in the field of algorithm-based fault tolerance and also describe compile-time approaches that have been taken by previous researchers in order to generate error detecting programs.

2.1 Algorithm-Based Fault Tolerance

The original paper on ABFT was by Huang and Abraham [2]. This paper introduced the concept of real number codes that were used to encode the data (mostly using checksumming). Transformed data was also expected to preserve the property of the code; i.e., the sum over the transformed data should also equal the transformed checksum. Checksum coding schemes were devised for matrix multiplication executing on systolic hardware. Redundant processors needed to be added to the hardware to operate on the coded data. Location and correction of errors in a single matrix element were discussed. Partitioning schemes for achieving single error location and correction were discussed when the matrix size exceeded the number of processors in the systolic array. Applications of ABFT to LU factorization and matrix inversion were discussed; however, these latter schemes were limited to error detection only. Encoding the input data by using weighted checksums was introduced in [3]. By using this scheme, single error detecting and correcting schemes were devised for matrix multiplication, LU factorization, and matrix inversion executing on systolic hardware. A new form of encoding, the sum-of-squares (SOS) encoding was introduced for the FFT and QR factorization in [11]. Two algorithms for the FFT performing on-line error detection by employing checksums and fault location by performing data retry were discussed in [1, 4]. A fault-tolerant algorithm for solving partial differential equations (specifically, Laplace's equation) on a mesh connected processor array was discussed in [12].

More recent research efforts have been to devise ABFT schemes for general-purpose multiprocessors. Unlike schemes developed for systolic algorithms, these latter schemes do not require hardware modifications and only perform error detection. ABFT schemes for matrix multiplication, Gaussian elimination, and the FFT for a hypercube multicomputer have been discussed in [6]. An ABFT version of a Givens QR factorization for a hypercube multicomputer has been discussed in [13] and for the singular value decomposition (SVD) in [5]. A fault-tolerant bitonic sorting algorithm based on verifying that intermediate sequences sorted by a processor are themselves bitonic and contain exactly the expected elements has been discussed in [14]. Error-detecting parallel algorithms for iterative solvers for partial differential equations (PDEs) have been discussed in [15] and [16].

Practical ABFT techniques for numerical applications typically check the integrity of floating-point computations by verifying the equality of two quantities, which are theoretically identical but have been computed in two different ways. Due to differences in the manner in which roundoff accumulation occurs in the computation of these theoretically identical quantities, a small tolerance has to be allowed during their comparison. For a practical implementation of an ABFT scheme, close attention has to be paid to choosing this tolerance, because too large a value could cause errors caused by hardware faults to go undetected, while too small a value could cause roundoff errors to trigger false alarms. Finding a good value for the tolerance is by no means an easy task since the accumulated roundoff error is dependent not only on the algorithm and problem size, but also on the specific data set. An experimental approach for choosing the tolerance was adopted in [5, 6]. This approach requires rerunning the experiments for computing the tolerance value each time the characteristics of the data set, such as the size and the range, change significantly. An approach based on extracting the mantissas of the floating-point numbers into integers and applying integer operations (for which no roundoff errors occur) has been described in [17]. However, this method could only check floating-point multiplications using a checksum approach. Floating-point additions had to be checked against a tolerance, as before. In [18], error expressions were derived for the quantities being compared. These error expressions were used at runtime to keep track of roundoff error accumulation. Although the error computation was sensitive to the data set and problem size, the expressions needed to be derived separately for each algorithm under consideration.

The problem of analyzing ABFT schemes with a view to determining their error-detecting and locating capabilities was introduced in [19]. A tripartite graph model was proposed with nodes corresponding to processors, data, and checks, with edges between processors and data or between data and checks, formalizing the relationship between processors and the data elements produced by them and checks and the data elements checked by them. Lower and upper bounds were obtained on the number of processors necessary and sufficient to achieve a certain level of fault detection or location. These bounds were further improved in [20]. The same graph model has been used to study the design problem for ABFT, that is, to achieve a certain specified level of error or fault detection or location by introducing as few checks as possible [21, 22, 23, 24]. The restriction that each check be responsible for checking a constant number of elements was removed in [21];

however, this general design problem was shown to be NP-hard even when only one fault needed to be detected. However, using the above models to analyze or synthesize ABFT versions of parallel algorithms executing on general multicomputers is a difficult task. Because in a parallel realization of an algorithm on a message-passing multicomputer, the relationships between the processors, data elements, and checks are hard to formalize into the above graph model. Data may move around in messages, and data from other processors may be used for updating local data; it is not clear how to map this to a model that assumes that a fixed relationship is imposed by the algorithm between processors, data, and checks. None of the researchers have attempted to present an implementation of a fault-tolerant parallel algorithm by applying the above design techniques, and there are also no reports on mapping a parallel algorithm of reasonable complexity into the graph-based model.

Most of the literature on ABFT has concentrated on the problem of error detection rather than correction. In the early papers on ABFT [2], error location and correction were achieved for a matrix multiplication algorithm by the addition of row and column checksums to the matrix. This method could detect and correct a single erroneous element in the matrix and was not extended to deal with locating and correcting multiple erroneous elements. The problem of locating and correcting a single faulty element for the result of a matrix multiplication algorithm was addressed in [3] by the introduction of weighted checksums. This scheme was extended in [8] to perform single error location and correction for LU decomposition and QR factorization algorithms. The problem of choosing weights for the weighted checksum scheme in order to detect and correct more than a single error was studied in [7]. The weights for the weighted checksums were chosen from a Vandermonde matrix, and the decoding scheme was able to locate and correct double errors in a matrix. The ingenious decoding procedure, however, was not generalized for locating and correcting an arbitrary number of errors. Also, since entries in a Vandermonde matrix typically span wide ranges for even modest matrix dimensions, it is possible that the decoding process could run into numerical difficulties. A probabilistic method for determining data-check assignments to obtain a t-error locating algorithm, where t can be arbitrary, was given in [9]. However, the problem of error correction was not addressed.

In this dissertation, we devise a general method for error location and correction for an algorithm executing on a multiprocessor system where up to t processors can be faulty. The ideas for error location are borrowed from system-level diagnosis theory. The idea for error correction is based on weighted checksums and is somewhat similar to the idea developed in [7]. However, since we decouple the error location and correction problems, we are able to give a general procedure for recovering from errors introduced by t faulty processors, where t is arbitrary. Also, our weights are chosen from the parity check matrix of a Reed-Solomon code. For the same dimension, the elements of such a matrix would span a much smaller range than a Vandermonde matrix and would be less likely to cause numerical problems. In contrast to the t-error locating method developed in [9], our approach is deterministic rather than probabilistic. Another difference of our work from the work of [9] is that in the latter approach, the data that a particular check is assigned to check can be somewhat arbitrary because of the probabilistic approach used for data-check assignments. In a multiprocessor system, the computation of the checks would lead to a large number of messages being exchanged in an irregular pattern, with potentially heavy communication overheads. In the approach we describe in this dissertation, heavy communication overheads are usually not a problem for programs exhibiting regular communication behavior.

2.2 Compiler-Assisted Generation of Error-Detecting Programs

Other researchers have also looked at the problem of automatic generation of error-detecting code at compile time. The approach of [25] and [26] was to utilize the Very Long Instruction Word (VLIW) compiler to insert redundant operations into functional units that would otherwise be idle. Fault diagnosis could also be done by analyzing functional unit mismatches. This approach required hardware modification in the form of comparators to compare the outputs of the functional units. Also, it was tied to a particular kind of processor architecture, viz., VLIW processors. This technique could be used in conjunction with ours, since duplicated code is produced by our compiler for portions of code that do not implement affine transformations. The duplicated instructions could then be scheduled to utilize idle slots in the functional units of the VLIW processor.

Another approach to compiler-assisted fault detection for parallel programs was discussed in [27, 28]. Here, statements were duplicated in Single Program Multiple Data (SPMD) parallel programs and executed on processors that would otherwise be idle. However, in cases where the overhead for duplication would be too great, for example in loops that could be executed in parallel keeping each processor busy, only the last statement executed by each processor was duplicated and compared on another processor. While this may suffice in detecting permanent faults, it is not adequate for transient fault detection. Again, this technique could be used in conjunction with ours for portions of code that would be checked by duplication using our approach.

An automated approach for identifying linear transformations in a program and generating the code for computing and transforming checksums at compile time was first proposed in [29, 30].

Our approach builds on this idea, while improving on it in several ways to make it viable. The approach of [29] analyzed one statement at a time instead of the entire program, leading to potential inefficiencies in checksum computation. Also a full-fledged implementation based on their ideas was not performed, leaving the feasibility and usefulness of their approach unresolved. In the work reported in this dissertation, we improve on the ideas suggested in [29] in several ways and implement them by augmenting a state-of-the-art parallelizing compiler.

The major improvements of our work over [29] are as follows. We are able to generate checksum-based checks for code that performs affine transformations, which are more general than linear transformations. To be able to generate a checksum-based check for a statement, it must possess a suitable syntactic structure, and additionally satisfy some dependence conditions. We attempt to reuse checksums from one loop to the next instead of recomputing checksums for every statement. To determine if this is possible, we perform a dataflow analysis on the entire program. Finally, we use data distribution information provided by the original program through HPF directives to specify data distributions for the checksums (or any other extra data that may be needed to check the original computation) in such a manner that a checksum and the portion of data being checked reside on different processors. Such data distribution specifications, together with the owner-computes rule [31], ensures that the check for the data owned by one processor is performed on a different processor, thus increasing the likelihood of detecting single-processor failures.

CHAPTER 3

ALGORITHM-BASED FAULT LOCATION AND RECOVERY

Most ABFT versions for general-purpose multicomputers have performed only error detection. However, in the case of intermittent or permanent errors, it is also desirable to locate the faulty processor so that it may be repaired or replaced. Furthermore, in the case of a real-time application, it is desirable that the correct data be recovered by performing very few additional operations (without having to restart and rerun the entire application), even though this might mean some extra overhead during normal operation. In this chapter we describe a method for locating and recovering from t faults, where t is a design parameter. Section 3.1 describes the location strategy while Section 3.2 describes the recovery strategy. Section 3.3 presents an example for multiple-fault location and recovery for a generic algorithm.

3.1 Location

We describe a methodology for error location that is suitable for application to linear algebra applications. In particular, we have demonstrated it on three parallel numerical algorithms - matrix multiplication, QR factorization, and Gaussian elimination. The location method is directly derived from the theory of system-level diagnosis introduced in [32]. It is well-known that a one-step t-diagnosable system must satisfy the following two constraints:

a. There must be at least $2t+1$ nodes in the system.

b. Each node must be diagnosed by at least t other nodes.

Before we proceed further, we recapitulate the definition of a $D_{\delta,t}$ system [33, 34].

Definition 1 *A $D_{\delta,t}$ system is a directed graph $G = (V, E)$ where an edge $e_{ij} \in E$ exists from a vertex $v_i \in V$ to a vertex $v_j \in V$ if and only if $j = (i + \delta m) \bmod n$ where n is the number of vertices in G, δ is an integer, and $m = 1, 2, \ldots, t$.*

It is well-known [33, 34] that for a class of $D_{\delta,t}$ systems with $n = 2t+1$ and in which δ and n are relatively prime, the conditions for t-fault diagnosability stated above are not only necessary but also sufficient if we assume that the vertices of the graph represent the processing nodes in the system and the edges represent the testing links. Such systems are thus optimal both with respect

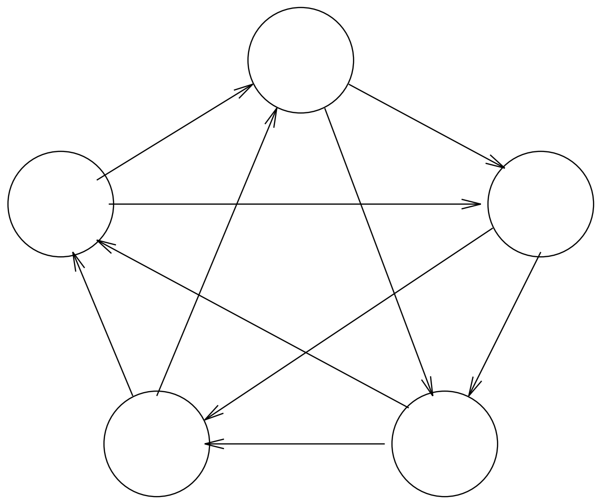


Figure 3.1 An optimal 2-fault diagnosable system.

to the number of processing nodes as well as to the testing links. An example of a $D_{1,2}$ system with $n = 5$, i.e., an optimal configuration for one-step 2-fault diagnosability, is shown in Fig. 3.1.

An ABFT system may be modified to perform t-fault location as follows. The set of p processors is grouped into $\frac{p}{2t+1}$ disjoint sets consisting of $2t+1$ processors each. Each such set will be referred to as a check group. The case when p is not divisible by $2t+1$ is easy to handle by making some of the check groups contain $2t+2$ processors. The checking assignments for a check group may be chosen to correspond to the checking assignments of a $D_{\delta,t}$ system. In an ABFT system, however, it is possible that a processor may need to use data from other processors either to update its own data or the encoded data required to check the computations of the other processors it is assigned to check. In a message-passing distributed system such as the one on which we conducted our experiments, use of data on other processors corresponds to communication of the required data via a message. Assume that processor $proc$ receives a message from processor $proc'$ containing data needed by $proc$ for its own updates. Before using this data in its own updates, processor $proc$ needs to subject this data to a check. However, checks on this data can be conducted on the t processors checking the data of processor $proc'$ (to achieve this, it is necessary that the data to be checked be communicated to the relevant processors by either $proc$ or $proc'$). Processor $proc$ proceeds to use this data in its updates only if all t checks pass. If t faults affect the checking processors in such a way that all of the checks pass, $proc'$ is fault-free by the t fault assumption, and thus the data communicated by $proc'$ is uncorrupted and may still be used by $proc$ in its updates. If a fault affects processor $proc'$ and causes the communicated data to be corrupted and at most t faults occur, then at least one check of the communicated data is performed by a fault-free processor and is guaranteed to fail. In this case, normal computations are stopped, and a checking phase is initiated in which each processor checks the data of the processors it is assigned to check. Unless the error in the communicated data was due to a transient error, it is highly likely that faulty processors would have corrupted some of their data. (If all checks pass, the communicated data was likely corrupted by a transient fault while in transit, and the data may be recommunicated). Since the syndrome corresponding to any t or fewer faulty processors is guaranteed to be unique in a $D_{\delta,t}$ system, the t faulty processors may be identified. (For simplicity of implementation, we assume the existence of a reliable host processor that receives and interprets the syndrome via a table lookup, although any of the several more sophisticated centralized or distributed diagnosis algorithms in the literature [34] could be used instead). The normal ABFT checks are carried out as usual at appropriate points in the algorithm.

3.2 Recovery

The extension to perform recovery from t faults uses ideas from the theory of error correcting codes [35] and is primarily applicable to algorithms that perform linear operations on data. A wide class of numerical algorithms falls into this category, such as matrix multiplication, Gaussian elimination, QR factorization, FFT, iterative linear system solvers, and so forth. We first introduce the following lemma which is very useful to demonstrate the applicability of coding theoretic results over certain finite fields to the field of real numbers. In the following lemma, $GF(q)$ denotes the finite field with q elements.

Lemma 1 *Vectors that are linearly independent over $GF(q)$ (q prime) are also linearly independent over the field of real numbers.*

Proof: Refer to [36]. □

Let us consider a system with a total of p processors. We partition the set of p processors into two sets, one consisting of processors 0 through $p-t-1$, which we denote by $\mathcal{P}$, and the other consisting of processors $p-t$ through $p-1$, which we denote by $\mathcal{E}$. In the rest of the section we will refer to the processors in set $\mathcal{P}$ as computation processors while we will refer to the processors in set $\mathcal{E}$ as check processors.

The method of data distribution and the fault-recovery procedure described in the remainder of the section guarantee recovery from t failures if all failures are confined to set $\mathcal{P}$, and from

$2(\sqrt{t+1}-1)$ failures if failures can occur in both sets $\mathcal{P}$ and $\mathcal{E}$. We would like to mention at the outset that, in fact, many correctable failure patterns exist where the number of failures exceeds the lower bound for the second case mentioned, and, in fact, our correction algorithm is able to correct any correctable failure pattern involving fewer than t faulty processors even when faults can affect processors from both sets $\mathcal{P}$ and $\mathcal{E}$.

Let us denote the data owned by the ith computation processor by C_i and the data owned by the ith check processor by S_i. Here C_i is assumed to be an $m \times n$ matrix. A large class of problems, and in particular each of the problems mentioned at the start of this section, can be structured to fall into this category. We assume that at all steps during the computation, the following invariant is maintained:

$$S_j = \sum_{i=0}^{p-t-1} w_{ji} C_i, \ 0 \leq j \leq t-1 \tag{3.1}$$

In other words, the data on the check processors is a weighted sum of the data on the computation processors. For an algorithm that performs linear operations on the data, the above invariant can be maintained by distributing weighted sums of the input data to the check processors at the start of the algorithm, and then having the check processors perform the same linear transformations on their data as the computation processors. We now indicate a suitable choice of weights to maximize chances of recovery following processor faults. We introduce the following notation:

$$\begin{aligned} S^T &= (S_0 \ S_1 \ \dots \ S_{t-1}) \\ C^T &= (C_0 \ C_1 \ \dots \ C_{p-t-1}) \\ W &= \begin{pmatrix} w_{00} & w_{01} & \cdots & w_{0\,p-t-1} \\ w_{10} & w_{11} & \cdots & w_{1\,p-t-1} \\ \vdots & \cdots & \cdots & \vdots \\ w_{t-1\,0} & w_{t-1\,1} & \cdots & w_{t-1\,p-t-1} \end{pmatrix} \end{aligned} \tag{3.2}$$

We have the following result.

Lemma 2 *Let α be a primitive element over $GF(q)$, where q is any prime greater than $p-t$. Let the entries of W be chosen so that $w_{ij} = \alpha^{ij} \bmod q$. Consider any submatrix W_{SM} of W consisting of any c consecutive rows of W. Then every c columns of W_{SM} are linearly independent over the field of real numbers.*

Proof: (For the properties used in the following proof, the reader is referred to [35].)

Replacing each w_{ij} by α^{ij} in W_{SM}, we find that W_{SM} consists of the first $p-t$ columns of the parity check matrix of a $(q, q-c)$ Reed-Solomon code. A $(q, q-c)$ Reed-Solomon code has minimum distance $c+1$, and so any c columns of its parity check matrix $H_{RS(q,q-c)}$ are linearly independent over $GF(q)$. Since W_{SM} consists of the first $p-t$ columns of $H_{RS(q,q-c)}$, any c columns of W_{SM} are also linearly independent over $GF(q)$. By Lemma 1, any c columns of W_{SM} are linearly independent over the field of real numbers as well. □

The following two corollaries follow immediately from Lemma 2.

Corollary 1 Every t columns of W are linearly independent over the field of real numbers.

Corollary 2 Every $c \times c$ submatrix of W_{SM} is of full rank.

As in the algorithm for the location of multiple faults described in Section 3.1, the p processor system is partitioned into $\frac{p}{2t+1}$ disjoint check groups consisting of $2t+1$ processors each, with the checking assignments in each check group chosen to correspond to an optimal $D_{\delta,t}$ system. Note that as far as the checking assignments for fault location are concerned, no distinction is made between check processors and computation processors; i.e, the redundant data on the check processors is subjected to the same ABFT checks as the original data on the computation processors. As before, all communicated data is subjected to a check before its use in the manner of Section 3.1. The invariant of Eq. (3.1) is thus correctly maintained on nonfaulty processors as long as the checks on the communicated data pass. However, if either a check on the communicated data or a regular ABFT check fails, the fault-location algorithm is executed to determine the set of faulty processors that have corrupted their data. Once the faulty processors have been located, data recovery may be initiated as follows.

Let the set of faulty processors be denoted by $\mathcal{F}$. Let us define two subsets $\mathcal{F}_P = \mathcal{F} \bigcap \mathcal{P}$ and $\mathcal{F}_E = \mathcal{F} \bigcap \mathcal{E}$. Let $|\mathcal{F}_P| = \nu_P$ and $|\mathcal{F}_E| = \nu_E$, where we use the $|X|$ notation to denote the cardinality of a set X. Let the indices of the processors in $\mathcal{F}_P$ be $f_{P_0}, f_{P_1}, \ldots, f_{P_{\nu_P-1}}$, the indices of the processors in $\mathcal{F}_E$ be $f_{E_0}, f_{E_1}, \ldots, f_{E_{\nu_E-1}}$, the indices of the processors in $\mathcal{P} - \mathcal{F}_P$ be $g_{P_0}, g_{P_1}, \ldots, g_{P_{p-t-\nu_P-1}}$, and the indices of the processors in $\mathcal{E} - \mathcal{F}_E$ be $g_{E_0}, g_{E_1}, \ldots, g_{E_{t-\nu_E-1}}$. Let us consider the system of matrix equations which may be derived from the system of matrix equations in Eq. (3.1) by deleting equations corresponding to indices in $\mathcal{F}_E$ and moving matrix terms corresponding to indices in $\mathcal{P} - \mathcal{F}_P$ and $\mathcal{E} - \mathcal{F}_E$ to the right.

$$\begin{aligned} \sum_{i=0}^{\nu_P-1} w_{f_{P_i} g_{E_j}} C_{f_{P_i}} &= S_{g_{E_j}} - \sum_{i=0}^{p-t-\nu_P-1} w_{g_{P_i} g_{E_j}} C_{g_{P_i}}, \\ & \quad 0 \le j \le t - \nu_E - 1 \end{aligned} \tag{3.3}$$

We notice that in Eq. (3.3), the left hand side involves C_i's which are unknown since they were to have been computed by the processors in the faulty set $\mathcal{F}_P$, while the right hand side involves known C_i's and S_j's since these were computed by processors in nonfaulty sets $\mathcal{P}-\mathcal{F}_P$ and $\mathcal{E}-\mathcal{F}_E$. Thus we have a system of $t-\nu_E$ matrix equations in ν_P matrix unknowns that may be solved if there exist at least ν_P linearly independent equations involving the unknowns in the system.

Let us further denote by $C_{\mathcal{F}_P}$ the matrix consisting of only those C_i's in C with indices in $\mathcal{F}_P$ and by $C_{\mathcal{G}_P}$ the remaining C_i's in C. Let us denote by W_R the reduced matrix constructed by deleting from W all rows corresponding to indices of processors in $\mathcal{F}_E$ and a reduced matrix S_R by deleting from S all S_j's corresponding to indices of processors in $\mathcal{F}_E$. Let us now define W_{R_f} to be the matrix consisting of only those columns of W_R with indices corresponding to processors in $\mathcal{F}_P$ and W_{R_g} to be the matrix consisting of the remaining columns of W_R. Then, Eq. (3.3) may be represented more succinctly in matrix notation by

$$W_{R_f} \otimes_{m\times n} C_{\mathcal{F}_P} = S_R - W_{R_g} \otimes_{m\times n} C_{\mathcal{G}_P} \tag{3.4}$$

In Eq. (3.4), the $\otimes_{m\times n}$ notation is used to indicate that each element of W_{R_f} and W_{R_g} multiplies an entire $m\times n$ block of $C_{\mathcal{F}_P}$ and $C_{\mathcal{G}_P}$, respectively, unlike normal matrix multiplication, where each element multiplies a single element (i.e., $\otimes_{1\times 1}$ is equivalent to normal matrix multiplication).

W_{R_f} is a matrix of dimensions $(t-\nu_E)\times\nu_P$. The system represented by Eq. (3.4) possesses a unique solution if and only if the rank of W_{R_f} equals ν_P. Eq. (3.4) can be constructed using only data from nonfaulty processors and leaving the data from faulty processors as unknowns to be solved for. Both sides of Eq. (3.4) are premultiplied by $W_{R_f}^T$ to get the new system

$$(W_{R_f}^T W_{R_f}) \otimes_{m\times n} C_{\mathcal{F}_P} = W_{R_f}^T \otimes_{m\times n} (S_R - W_{R_g} \otimes_{m\times n} C_{\mathcal{G}_P}) \tag{3.5}$$

An LU decomposition of the $\nu_P\times\nu_P$ matrix $(W_{R_f}^T W_{R_f})$ is then performed. (Note that the system defined by Eq. (3.5) is symmetric, so that its LU decomposition may be obtained by using only half of the operations and memory for a general unsymmetric system). If the rank of $(W_{R_f}^T W_{R_f})$ is ν_P, i.e., the matrix is of full rank, then its LU decomposition does not lead to any 0 elements occurring on the diagonals of either triangular matrix in the decomposition of $(W_{R_f}^T W_{R_f})$. (If roundoff errors are of concern, one may attempt to first perform the LU decomposition over $GF(q)$ and proceed with the LU decomposition over the field of real numbers only if the decomposition over $GF(q)$ succeeds, since by Lemma 1, linear independence over $GF(q)$ guarantees linear independence over the field of real numbers). All unknown elements can then be recovered by backsolves. The matrix $(W_{R_f}^T W_{R_f})$ is of full rank if and only if $W_{R_f}^T$ has rank ν_P [37]. So in these cases, the recovery from the pattern of faulty processors is possible.

We now examine in which cases the matrix $(W_{R_f}^T W_{R_f})$ is of full rank since in these cases, corrupted strips of C may be recovered.

Theorem 1 *If only processors in $\mathcal{P}$ fail, then t faults can be tolerated by the algorithm for multiple fault recovery.*

Proof: If only nodes in $\mathcal{P}$ fail, then the set $\mathcal{F}_E$ is empty. Thus $W_R = W$, and therefore W_{R_f}, for this fault pattern consists of ν_P columns of W, where $\nu_P \leq t$. By Corollary 1 to Lemma 2, the columns of W_{R_f} are linearly independent. Thus $(W_{R_f}^T W_{R_f})$ has full rank and Eq. (3.5) may be solved to recover all corrupted strips of C. $\square$

Theorem 2 *The algorithm for recovery from multiple faults can tolerate any pattern involving $2(\sqrt{t+1}-1)$ or fewer processors.*

Proof: Assume that a total of $\nu = \nu_P + \nu_E$ faults have occurred. Consider the matrix W_R, which is constructed by deleting all rows in W corresponding to indices in the set $\mathcal{F}_E$. Since W has t rows, no matter which ν_E rows of W are deleted, W_R will contain at least $\frac{t-\nu_E}{(\nu_E+1)}$ consecutive rows that were also consecutive in W. (The minimum occurs when the deleted rows are evenly spaced). Let the matrix formed by these consecutive rows be denoted by W'. By Lemma 2, every $\frac{t-\nu_E}{(\nu_E+1)}$ columns of W' are also linearly independent. Thus if $\nu_P \leq \frac{t-\nu_E}{(\nu_E+1)}$, the ν_P columns of W_{R_f} are guaranteed to be linearly independent, and thus W_{R_f} has rank ν_P. Hence, the matrix $(W_{R_f}^T W_{R_f})$ is of full rank, and Eq. (3.5) may be solved to recover the corrupted C_i's. Thus in the event that ν_E check processors have failed, the complete recovery can be performed provided fewer than $\frac{t-\nu_E}{(\nu_E+1)}$ computation nodes have failed; i.e., it can tolerate a total number of failed nodes $\nu \leq \nu_E + \frac{t-\nu_E}{(\nu_E+1)}$. Treating ν as a function of ν_E, we find that it possesses a minimum at $\nu_E = \sqrt{t+1}-1$ and the value of ν at this minimum is $2(\sqrt{t+1}-1)$. Thus any fault pattern involving $2(\sqrt{t+1}-1)$ or fewer nodes can be tolerated. $\square$

Note that the proofs of Theorems 1 and 2 suggest a simplification of the error recovery strategy outlined earlier. In the event that ν_P faults affect the set $\mathcal{P}$, and the total number of faults ν is less than $2(\sqrt{t+1}-1)$, we are guaranteed to find ν_P consecutive nonfaulty processors in the set $\mathcal{E}$. By restricting our attention to the portion of W on these processors only, we find that the columns corresponding to the indices in $\mathcal{F}_P$ form a $\nu_p \times \nu_p$ submatrix, which, by Corollary 2 to Lemma 2, is of full rank. Thus we may obtain ν_P equations involving the corrupted data as unknowns that are guaranteed to be linearly independent without having to construct $(W_{R_f}^T W_{R_f})$. In the event that faults affect only $\mathcal{P}$ and $\nu \leq t$ such faults occur, a linearly independent system involving the corrupted data as unknowns may be obtained by simply utilizing the weighted checksums on any ν consecutive processors in $\mathcal{E}$. However, in the event that the number of faults exceeds the bounds of Theorems 1 and 2, the general procedure outlined earlier may be used, since it may turn out that $(W_{R_f}^T W_{R_f})$ is of full rank, and Eq. (3.5) may be solved to recover the corrupted data.

Once the corrupted data has been recovered, the computations of the faulty processors are taken over by the nonfaulty check processors, which then perform normal computations instead of check computations. This recovery strategy is fast and simple, but leaves the system vulnerable to further failures. At the expense of complicating the recovery procedure, future failures may also be tolerated if the surviving processors are again partitioned into check and computation processors, the data partitioned and redistributed to the computation processors, and the weighted sums of the data on the computation processors distributed to the check processors.

3.3 Example

To clarify the methodology for algorithm redesign to perform multiple-fault location and recovery, let us consider a generic algorithm $\mathcal{G}$ designed for execution on a 14-processor system. Assume that 3-fault location and 2-fault recovery are required. The 14 processors in the system are partitioned into two subsets, with processors 0 through 10 comprising the set of computation processors and processors 11 through 13 comprising the check processors. Assume that the initial data distribution consists of $m \times n$ matrices A_0 through A_{10} on processors 0 through 10, respectively. Processors 11, 12, and 13 compute weighted checksums S_0, S_1, and S_2 of the data on the other processors by using the following equations:

$$\begin{aligned} S_0 &= A_0 + A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + A_9 + A_{10} \\ S_1 &= A_0 + 2A_1 + 4A_2 + 8A_3 + 3A_4 + 6A_5 + 12A_6 + 11A_7 + 9A_8 + 5A_9 + 10A_{10} \\ S_2 &= A_0 + 4A_1 + 3A_2 + 12A_3 + 9A_4 + 10A_5 + A_6 + 4A_7 + 3A_8 + 12A_9 + 9A_{10} \end{aligned} \tag{3.6}$$

The weights chosen comprise the first 11 columns of a (13,10) Reed-Solomon code. Since we want triple-fault location, the 14 processors are grouped into two sets, one containing processors 0 through

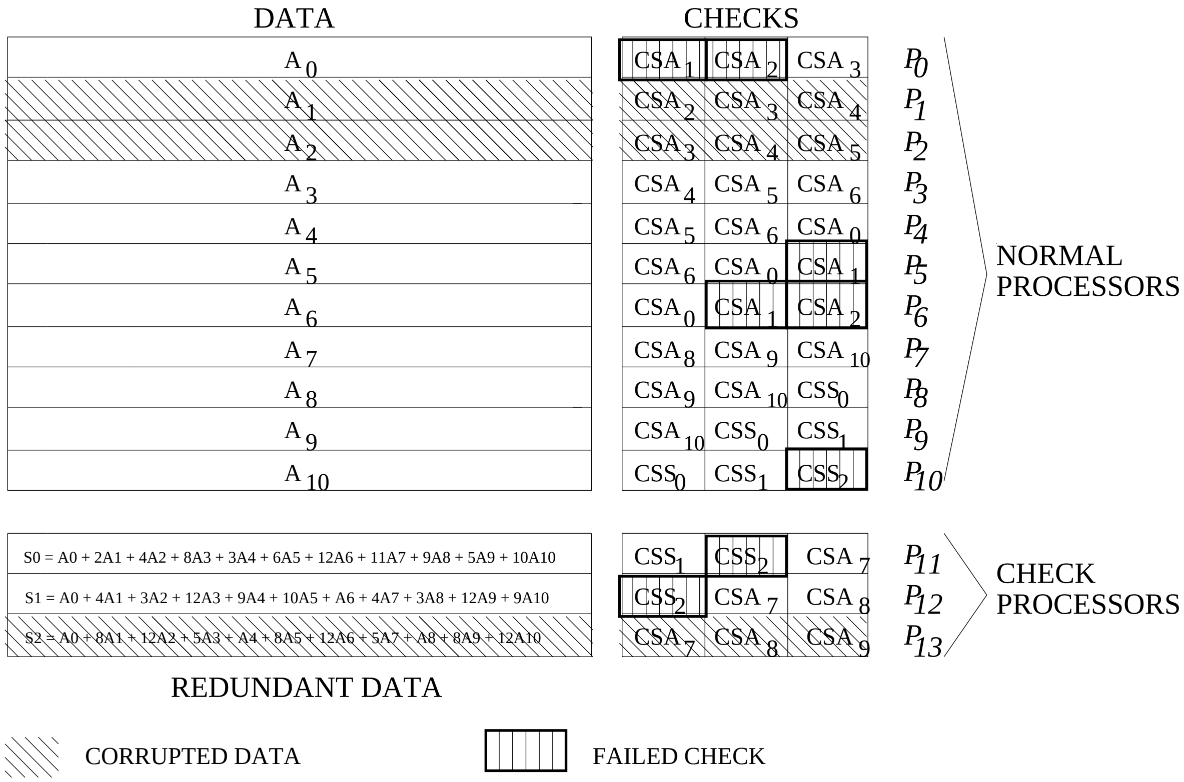


Figure 3.2 Data distribution and error-pattern example.

6 and the other containing processors 7 through 13. Each processor keeps checks on the data on three other processors (these checks can be checksums or some other appropriate encoding of the data that can be easily maintained), where the processors checked by each processor are assigned in correspondence with the edges of a $D_{1,3}$ system. Now suppose that after some transformations have taken place, processors 1, 2, and 13 fail. Checks on processor 1's data fail on processors 0, 6, and 5, checks on processor 2's data fail on processors 0 and 6, and checks on processor 13's data fail on processors 10, 11, and 12, while the outcomes of the checks on processors 1, 2, and 13 can either pass or fail. However, the syndrome uniquely identifies processors 1, 2, and 13 as the faulty processors. The data distribution, corrupted data and failed checks are shown in Fig. 3.2. Now processors 11 and 12 form a pair of consecutive nonfaulty check processors. Considering only the checksums on processors 11 and 12, we may construct the following linear system:

$$\begin{aligned} A_1 + A_2 &= S_0 - (A_0 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + A_9 + A_{10}) \\ 2A_1 + 4A_2 &= S_1 - (A_0 + 8A_3 + 3A_4 + 6A_5 + 12A_6 + 11A_7 + 9A_8 + 5A_9 + 10A_{10}) \end{aligned}$$

where we abuse the notations A_i and S_i by using them to represent the transformed data as well as the original data. The right-hand side of the above equation consists of matrices computed by nonfaulty processors, and thus the corrupted data A_1 and A_2 may be recovered by solving the above linear system. Once the corrupted matrices have been recovered, processors 11 and 12 take over the computations of processors 1 and 2 for the rest of the computation. This makes the system susceptible to further failures; however, if fault tolerance is desired during the remainder of the computation, three of the surviving processors may be designated as check processors, the data may be redistributed on the rest of the surviving processors, and new weighted checksums may be computed before proceeding with the rest of the computation.

3.4 Summary

In this chapter, we have discussed a general methodology to be used in conjunction with the ABFT framework in order to perform error location and recovery of up to t faults, where t is a design parameter. Owing to our use of well-known results from the theory of system-level diagnosis and coding theory, our approach is considerably simpler to implement, as well as being more general, than earlier approaches for error location and correction in the ABFT framework. In the next chapter, we will demonstrate the practicality of our approach by designing error-locating and correcting versions of three parallel numerical algorithms using the methodology devised in this chapter.

CHAPTER 4

IMPLEMENTATION AND RESULTS FOR ALGORITHM-BASED FAULT LOCATION AND RECOVERY

To more clearly demonstrate the applicability of the proposed ABFT techniques for fault location and recovery discussed in Chapter 3, we discuss the modification of three parallel numerical algorithms (matrix multiplication, QR factorization, and Gaussian elimination) to achieve single-fault location and recovery and present results for each algorithm on a distributed-memory multicomputer.

4.1 Fault-Tolerant Algorithm Description

4.1.1 Matrix multiplication

We consider the parallel execution of $AB = C$, where A, B, and C are dense $n \times n$ matrices. The basic serial code for matrix multiplication is shown in Fig. 4.1. We assume that A is distributed blockwise by rows and B is replicated on all processors. Other data distributions (such as one in which B is distributed blockwise by columns) can also be easily handled. We also assume that n is divisible by $p-1$. We denote the quantity $\frac{n}{p-1}$ by m. We assume that the numbering of both of the processors as well as the rows and columns of the matrices starts from 0. We designate the $p-1$th processor to be a check processor and the rest of the processors to be computation processors, using the terminology introduced in Chapter 3. Let us denote the strip of A owned by the ith computation processor by A_i. A_i consists of rows mi through $m(i+1)-1$. Prior to the start of the

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  {
    C[i][j] = 0;
    for(k=0;k<n;k++)
      C[i][j] = C[i][j] + A[i][k] * B[j][k];
  }
```

Figure 4.1 Matrix multiplication program.

execution of the matrix multiplication algorithm, an additional strip $A_{p-1} = \sum_{i=0}^{p-2} A_i$ is computed and communicated to the check processor (note that if only single fault location is desired, this step may be omitted, and instead, A may be distributed over p instead of $p-1$ processors). We assume for clarity of presentation that p is divisible by three. The set of p processors is now divided into $\frac{p}{3}$ check groups of three processors each. Within each check group, the processors are ordered according to processor identifiers. Suppose check group i contains processors $g_0^{(i)}, g_1^{(i)}$, and $g_2^{(i)}$. The processors in each check group are logically configured in a directed cycle. We shall speak of succeeding or preceding processors given a particular processor *proc*, with the ordering being implied by the directed cycle of the check group to which *proc* belongs. Prior to the start of the execution of the matrix multiplication algorithm, each processor receives and computes the checksum of the rows of the strip of A belonging to its succeeding processor. Thus, in matrix notation, processor $g_j^{(i)}$ computes

$$(acs_{g^{(i)}_{(j+1)\bmod 3}})^T = e^T A_{g^{(i)}_{(j+1)\bmod 3}} \tag{4.1}$$

where e^T denotes the all-1's row vector of length m, and $(acs_{g_k^{(i)}})^T$ denotes the checksum row of $A_{g_k^{(i)}}$. The check processor is grouped into a check group and participates in the checksum computation in the same manner as the computation processors. Note that the checksum assignments in a check group actually correspond to the check assignments of a $D_{1,1}$ system. The data distributions of the matrices and checksums on a 6-processor system are shown in Fig. 4.2. The processors then proceed to execute the matrix multiplication algorithm in parallel. Thus the following computation is performed by processor $g_j^{(i)}$:

$$\begin{pmatrix} C_{g_j^{(i)}} \\ (ccs_{g^{(i)}_{(j+1)\bmod 3}})^T \end{pmatrix} = \begin{pmatrix} A_{g_j^{(i)}} \\ (acs_{g^{(i)}_{(j+1)\bmod 3}})^T \end{pmatrix} B \tag{4.2}$$

Following the computation of $C_{g_j^{(i)}}$, processor $g_j^{(i)}$ also computes the checksum of $C_{g_j^{(i)}}$, which we denote by $(\widehat{ccs}_{g_j^{(i)}})^T$. In matrix notation, we have

$$(\widehat{ccs}_{g_j^{(i)}})^T = e^T C_{g_j^{(i)}} \tag{4.3}$$

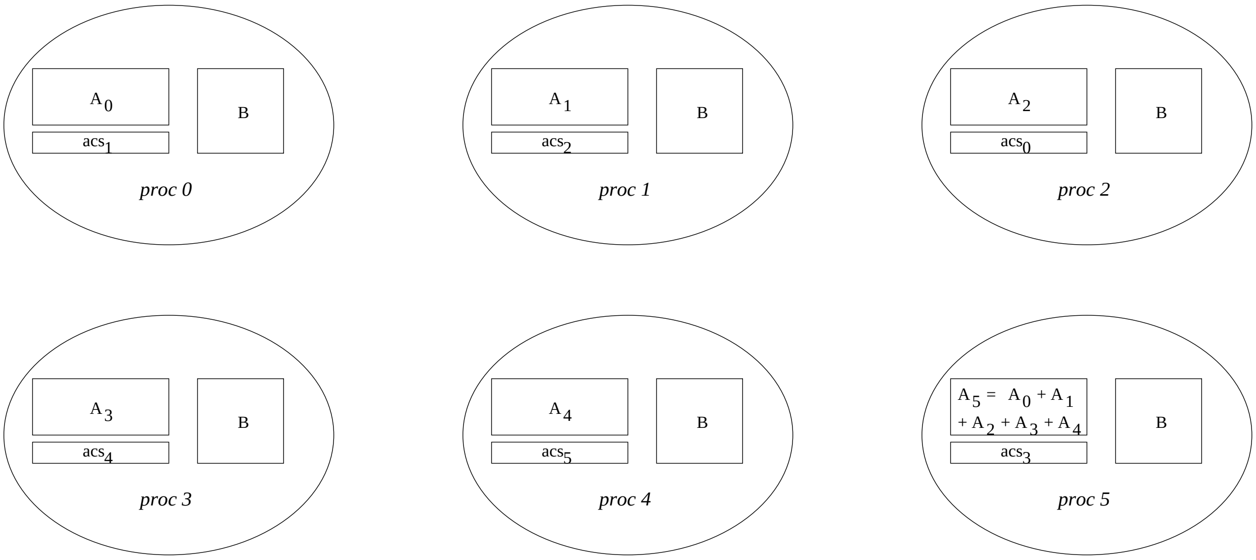


Figure 4.2 Data distribution for fault-tolerant matrix multiplication.

Next, processor $g_j^{(i)}$ sends $\widehat{ccs}_{g_j^{(i)}}$ to processor $g_{(j-1)\bmod 3}^{(i)}$. After processor $g_j^{(i)}$ receives $\widehat{ccs}_{g_{(j+1)\bmod 3}^{(i)}}$ from processor $g_{(j+1)\bmod 3}^{(i)}$, it compares $\widehat{ccs}_{g_{(j+1)\bmod 3}^{(i)}}$ and $ccs_{g_{(j+1)\bmod 3}^{(i)}}$. In the absence of processor failures, $\widehat{ccs}_{g_{(j+1)\bmod 3}^{(i)}}$ must equal $ccs_{g_{(j+1)\bmod 3}^{(i)}}$ to within a tolerance. (The tolerance is necessary due to the accumulation of roundoff errors in the computation of $\widehat{ccs}_{g_{(j+1)\bmod 3}^{(i)}}$ and $ccs_{g_{(j+1)\bmod 3}^{(i)}}$. For a discussion of a tolerance determination methodology, see [38]). In the event of a single processor failure, the checksum test for the strip of C computed by the faulty processor fails on the processor preceding it in its check group. The checksum test on the faulty processor itself may fail or pass. In either case, the test on the processor immediately preceding the faulty processor fails, and so the syndrome for any single fault within a check group is unique. The identity of the faulty processor may thus be determined by the host after receiving the results of the checksum tests of each processor. In fact, if the fault pattern is such that only one processor in each check group is faulty, such fault patterns can also be detected and located. In the event of a single processor failure, if the failed processor happens to be the check processor, nothing needs to be done as the entire matrix C may be assembled from the remaining processors. In the event the failed processor is one of the computation processors, say processor f ($f \neq p-1$), the corrupted strip C_f may be recovered by cooperation between the remaining processors by using the following equation:

$$C_f = C_{p-1} - \sum_{i=0, i\neq f}^{p-2} C_i \tag{4.4}$$

4.1.2 QR factorization

The problem of QR factorization is to factorize an $n \times n$ matrix A into an orthogonal matrix Q and an upper triangular matrix R, i.e, $A = QR$, which is achieved by premultiplying A by a series of orthogonal matrices $O^{(1)}, O^{(2)}, \ldots, O^{(N)}$. Two common orthogonal transforms are Givens and Householder transforms [39]. Givens transforms zero out a single element at a time while Householder transforms zero out an entire column at a time. We developed a parallel algorithm based on Householder transforms. In this case, we denote the matrices $O^{(k)}$ by $H^{(k)}$, with $N = n-1$, where the $H^{(k)}$'s are called Householder matrices. At the start of the algorithm, the initializations $A^{(0)} = A$ and $Q^{(0)} = I_{n \times n}$ are performed. These matrices are successively transformed to obtain R and Q^T. The kth Householder transform may be represented as $A^{(k)} = H^{(k)} A^{(k-1)}$ and $Q^{(k)} = H^{(k)} Q^{(k-1)}$. We first describe the kth Householder vector $v^{(k)}$ as

$$v^{(k)} = a^{(k-1)}_{k-1 \ldots n\; k-1} + \| \, a^{(k-1)}_{k-1 \ldots n\; k-1} \, \|_2 \; e_1 \tag{4.5}$$

where $a^{(k-1)}_{k-1 \ldots n\; k-1}$ denotes the vector formed by the elements in rows $k-1$ through n of column $k-1$ of $A^{(k-1)}$, $\| \; \|_2$ denotes the 2-norm, and e_1 is the column vector of length $n-k+1$ with a 1 as its first element and 0's elsewhere. We may define an $n-k \times n-k$ matrix $H^{(k)}_{sm}$ by the following equation:

$$H^{(k)}_{sm} = I_{n-k+1 \times n-k+1} - 2 \frac{v^{(k)} (v^{(k)})^T}{(v^{(k)})^T v^{(k)}} \tag{4.6}$$

The matrix $H^{(k)}$ is obtained by replacing the lower right $n-k+1 \times n-k+1$ submatrix of $I_{n \times n}$ by $H^{(k)}_{sm}$, as shown below.

$$H^{(k)} = \begin{bmatrix} I_{n-k+1 \times n-k+1} & \bigcirc \\ \bigcirc & H^{(k)}_{sm} \end{bmatrix} \tag{4.7}$$

It is easy to verify that $H^{(k)}$ is orthogonal and that it zeroes out the elements below the diagonal of the $k-1$th column of $A^{(k-1)}$. It is to be noted that a Householder transform would not actually be performed by constructing the Householder matrix and then performing a matrix multiplication. The actual implementation would be based on the code shown in Fig. 4.3, which has the same effect. Orthogonal transforms have the property that 2-norms of the columns being transformed are preserved. To devise fault-locating and fault-tolerant versions of the QR algorithm, error detection must first be performed. Error detection is achieved by maintaining two invariants, the sum-of-squares of each column of $A^{(k)}$ and $Q^{(k)}$ as well as the row checksum of each row of

```
/* Q is initialized to I */

for(k=0;k<n-1;k++)
{

  /* Compute Householder vector */
  vTv = 0;
  for(i=k;i<n;i++)
  {
    v[i] = A[i][k];
    vTv = vTv + A[i][k] * A[i][k];
  }
  v[k] = v[k] + sqrt(vTv);

  /* Update A */
  for(j=k;j<n;j++)
  {
    vTa = 0;
    for(i=k;i<n;i++)
      vTa = vTx + v[i] * A[i][j];
    for(i=k;i<n;i++)
      A[i][j] = A[i][j] - 2.0 * vTa * v[i] / vTv;
  }

  /* Update Q */
  for(j=0;j<n;j++)
  {
    vTq = 0;
    for(i=k;i<n;i++)
      vTq = vTq + v[i] * Q[i][j];
    for(i=k;i<n;i++)
      Q[i][j] = Q[i][j] - 2.0 * vTq * v[i] / vTv;
  }
}
```

Figure 4.3 QR factorization program.

$A^{(k)}$ and $Q^{(k)}$. The parallel single-fault tolerant QR algorithm is based on a column cyclic data distribution of the matrices $Q^{(0)}$ and $A^{(0)}$ over $p-1$ processors (one less than the total available), with processor $p-1$ being assigned the task of transforming a redundant strip of data. Thus processor i $(0 \leq i \leq p-2)$ obtains columns $i, i+p-1, i+2(p-1), \ldots, i+n-p+1$ of both $A^{(0)}$ and $Q^{(0)}$ prior to performing the first Householder transform. Processor $p-1$, designated the check processor, computes two redundant strips of data $S_A^{(0)}$ and $S_Q^{(0)}$. The ith column of $S_A^{(0)}$ $(S_Q^{(0)})$ is obtained by summing columns $i\frac{n}{p-1}$ through $(i+1)\frac{n}{p-1}-1$ of $A^{(0)}$ $(Q^{(0)})$. In matrix notation, we have

$$\begin{aligned} S_A^{(0)} &= A^{(0)}E \\ S_Q^{(0)} &= Q^{(0)}E \end{aligned} \tag{4.8}$$

where E is an $n \times \frac{n}{p-1}$ matrix with column i of E having 1's in rows $i\frac{n}{p-1}$ through $(i+1)\frac{n}{p-1}-1$ and 0's elsewhere. Let us denote the matrix $A^{(0)}$ augmented by appending the columns of $S_A^{(0)}$ to it by $A'^{(0)}$, i.e., $A'^{(0)} = [A^{(0)}|S_A^{(0)}]$. Analogously, the augmented matrix $Q'^{(0)}$ is defined as $Q'^{(0)} = [Q^{(0)}|S_Q^{(0)}]$. The matrices $A'^{(0)}$ and $Q'^{(0)}$ are thus $n \times \frac{np}{p-1}$. As before, the p processors are divided into check groups of 3 with each processor assigned to check the sum-of-squares of the succeeding processor in its check group. Also as earlier, we denote the members of the ith check group by $g_0^{(i)}, g_1^{(i)}$, and $g_2^{(i)}$. To enable processor $g_j^{(i)}$ to compute the sum-of-squares of the columns owned by processor $g_{j+1 \bmod 3}^{(i)}$, processor $g_j^{(i)}$ is communicated the columns owned by processor $g_{j+1 \bmod 3}^{(i)}$ along with its own columns at the start of the algorithm. Processor $g_j^{(i)}$ computes the sum-of-squares of the columns owned by $g_{j+1 \bmod 3}^{(i)}$ prior to performing the first Householder transform. Note that the redundant strips $S_A^{(0)}$ and $S_Q^{(0)}$ are treated no differently from the rest of the data; i.e., the sum-of-squares of S_A and S_Q are computed by the processor preceding processor p in its check group. Row checksums are also computed over all of the rows of the augmented matrices but are maintained on only one of the processors, say processor 0. Thus, if we denote the column sum-of-squares of $A'^{(0)}$ and $Q'^{(0)}$ by $sos_{A'}^T$ and $sos_{Q'}^T$ and the row checksums of the same matrices by $cs_{A'}^{(0)}$ and $cs_{Q'}^{(0)}$, then the computation of the sum-of-squares is indicated by the equations

$$\begin{aligned} (sos_{A'})_i^T &= (a'^{(0)}_{.i})^T(a'^{(0)}_{.i}) \quad 0 \leq i < \frac{np}{p-1} \\ (sos_{Q'})_i^T &= (q'^{(0)}_{.i})^T(q'^{(0)}_{.i}) \quad 0 \leq i < \frac{np}{p-1} \end{aligned} \tag{4.9}$$

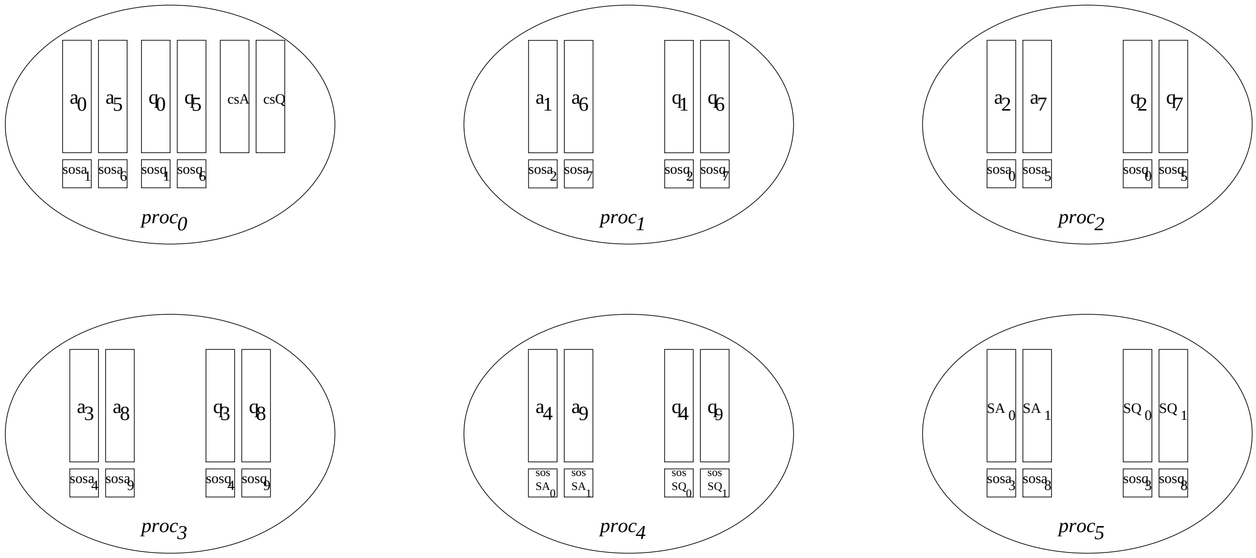


Figure 4.4 Data distribution for fault-tolerant QR factorization.

and the computation of the row checksums is indicated by the equations

$$
\begin{aligned}
cs_{A'}^{(0)} &= A'^{(0)}e \\
cs_{Q'}^{(0)} &= Q'^{(0)}e
\end{aligned}
\tag{4.10}
$$

Here, $a'^{(0)}_{.i}$ and $q'^{(0)}_{.i}$ denote the ith columns of $A'^{(0)}$ and $Q'^{(0)}$, respectively; the subscript i for the sum-of-squares vectors denotes the ith element, while e denotes the all-1's column vector with $\frac{np}{p-1}$ elements. An example data distribution for the fault-tolerant QR algorithm on a 6-processor system for a 10×10 matrix A is shown in Fig. 4.4. Once the initial row checksums and column sum-of-squares have been computed, the Householder transforms proceed in the normal manner except that the transforms are now applied to the augmented matrices and the row checksums. The sum-of-squares vectors are also updated in a manner discussed below, and we use the notation $(sos_{A'}^{(k)})^T$ and $(sos_{Q'}^{(k)})^T$ to indicate the sum-of-squares vectors after k Householder transforms. If we denote the augmented matrices after $k-1$ Householder transforms by $A'^{(k-1)}$ and $Q'^{(k-1)}$, then the kth Householder step in the fault-tolerant algorithm may be summarized by the equations

$$
\begin{aligned}
[A'^{(k)}|cs_{A'}^{(k)}] &= H^{(k)}[A'^{(k-1)}|cs_{A'}^{(k-1)}] \\
[Q'^{(k)}|cs_{Q'}^{(k)}] &= H^{(k)}[Q'^{(k-1)}|cs_{Q'}^{(k-1)}]
\end{aligned}
\tag{4.11}
$$

This transformation has the effect of preserving the row checksums of the augmented matrices. Note that since orthogonal transforms preserve 2-norms, and thus sum-of-squares, the vectors $sos_{A'}^T$ and $sos_{Q'}^T$ continue to maintain the sum-of-squares of the matrices $A'^{(k)}$ and $Q'^{(k)}$. However, note that the kth Householder transform only operates on rows $k-1$ through n of the matrices $A'^{(k-1)}$ and $Q'^{(k-1)}$. In fact, since the elements below the diagonal in the first $k-1$ columns of $A'^{(k-1)}$ have been zeroed out, the kth Householder transform only updates the lower-right $n-k+1 \times n-k+1$ submatrix of $A'^{(k-1)}$. This, coupled with the 2-norm preserving property of orthogonal transforms, implies that the sum-of-squares of the transformed submatrices of $A'^{(k-1)}$ and $Q'^{(k-1)}$ are preserved, which may be achieved by simply subtracting off the square of the elements in the $k-1$th row of $A'^{(k-1)}$ and $Q'^{(k-1)}$ from $(sos_{A'}^{(k-1)})^T$ and $(sos_{Q'}^{(k-1)})^T$; i.e., we have

$$\begin{aligned} (sos_{A'}^{(k)})_i^T &= (sos_{A'}^{(k-1)})_i^T - (a'^{(k-1)}_{k-1\,i})^2 \quad k \le i < \frac{np}{p-1} \\ (sos_{Q'}^{(k)})_i^T &= (sos_{Q'}^{(k-1)})_i^T - (q'^{(k-1)}_{k-1\,i})^2 \quad 0 \le i < \frac{np}{p-1} \end{aligned} \tag{4.12}$$

To update its local portion of the sum-of-squares vectors, each processor p needs elements of the $k-1$th rows of $A'^{(k-1)}$ and $Q'^{(k-1)}$ which reside on the succeeding processor in its check group. Each processor thus has to communicate the elements it owns in the $k-1$th rows of $A'^{(k-1)}$ and $Q'^{(k-1)}$ to the processor preceding it in its check group. Note that since elements $a'^{(k-1)}_{k-1\,0}$ through $a'^{(k-1)}_{k-1\,k-1}$ are 0's, these need not be communicated. Once this transfer has been achieved, the transferred data is subjected to a row checksum check on processor 0. By the single-fault assumption, data corrupted by a single faulty processor may be detected as long as the faulty processor is not processor 0. If, instead, processor 0 is the faulty processor, the check could pass. However, in this case, only the elements communicated by processor 0 could be faulty. Since these are only used to update the sum-of-squares values on the processor responsible for checking processor 0's columns, these could take on incorrect values. However, the next column sum-of-squares check would still correctly identify processor 0 as the faulty processor.

If the row checksum check fails, then each processor subjects the columns owned by its succeeding processor to a sum-of-squares check. This check is over all of the elements of the column, so that the original sum-of-squares vectors $sos_{A'}$ and $sos_{Q'}$ are used in the checks. The data corrupted by the faulty processor will lead to a failed check on the preceding processor, and the syndrome will correctly identify the faulty processor. Also note that prior to performing the kth Householder update on its columns, each processor needs to compute the kth Householder vector $v^{(k)}$ and the kth

Householder matrix $H^{(k)}$. Recall that the elements in rows $k-1$ through $n-1$ of the $k-1$th column of $A^{(k-1)}$ are required to compute the kth Householder vector $v^{(k)}$. This column is broadcast to all processors by its owner to enable them to compute the Householder vector and the Householder matrix for the iteration using Eqs. (4.6) and (4.5) and then perform the Householder transform. However, due to the update of the sum-of-squares vectors after each Householder transform, the sum-of-squares of the broadcasted column (which is denoted by $(sos_{A'}^{(k-1)})_{k-1}^{T}$) is already available on a different processor. The owner of $(sos_{A'}^{(k-1)})_{k-1}^{T}$ broadcasts the value to all processors, which compare $(a'^{(k-1)}_{k-1\ldots n\ k-1})^{T}(a'^{(k-1)}_{k-1\ldots n\ k-1})$ with $(sos_{A'}^{(k-1)})_{k-1}^{T}$ and proceed with the transformation step only if the check passes on all processors. If the check fails on any processor, each processor performs a sum-of-squares check on the columns owned by its succeeding processor. If the faulty processor has corrupted any of the columns owned by it, the syndrome correctly identifies it. Once a faulty processor has been identified, it is easy to recover the corrupted data. If processor $p-1$ is the faulty processor, computations may simply be continued on the remaining processors. If processor $f \neq p-1$ is the faulty processor, the remaining processors recover the corrupted columns of $A^{(k-1)}$ and $Q^{(k-1)}$ by using the equations

$$\begin{aligned} A^{(k-1)}G &= S_A^{(k-1)} - A^{(k-1)}F \\ Q^{(k-1)}G &= S_Q^{(k-1)} - Q^{(k-1)}F \end{aligned} \tag{4.13}$$

where G is an $n \times \frac{n}{p-1}$ matrix with the ith column having a 1 in position $\frac{in}{p-1}+f$ and 0's elsewhere, and $F = E - G$. The recovered data is then stored on processor $p-1$, which then takes over the computations of processor f for the remainder of the algorithm. After $n-1$ Householder steps, $Q = Q^{(n-1)}$ and $R = A^{(n-1)}$ are cyclically distributed on all processors. Note that if it is desired to be able to recover from further faults after recovery from the first fault, one of the surviving processors needs to be designated as the check processor for the remainder of the computation. A redundant strip of data is computed and transformed as before on the check processor, while the remaining processors continue to perform the normal Householder transforms.

4.1.3 Gaussian elimination

A system of linear equations $Ax = b$, where A is a dense $n \times n$ matrix, b is a column vector with n elements, and x is a column vector of n unknowns, is usually solved by applying Gaussian elimination to A and b, which results in the transformation of A into an upper triangular matrix U

and b into a modified right-hand side r. The solution of the unknowns may be obtained by solving the new system $Ux = r$, which, because of the upper triangular nature of U, is considerably easier to solve by a process known as back substitution. A total of $n-1$ linear transformations are applied to A as part of the Gaussian elimination process. If we denote the matrix A and the right-hand side vector b after $k-1$ transformation steps by $A^{(k-1)}$ and $b^{(k-1)}$, the kth transformation step may be represented as

$$[A^{(k)}|b^{(k)}] = M_k P_k [A^{(k-1)}|b^{(k-1)}] \tag{4.14}$$

where P_k is a permutation matrix responsible for switching the kth row of $A^{(k-1)}$ with one of the rows between k and n, and M_k is defined as

$$M_k = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & & 1 & 0 & & 0 \\ 0 & & -\frac{a_{k\ k-1}^{(k-1)}}{a_{k-1\ k-1}^{(k-1)}} & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\frac{a_{n-1\ k-1}^{(k-1)}}{a_{k-1\ k-1}^{(k-1)}} & 0 & \cdots & 1 \end{bmatrix} \tag{4.15}$$

M_k is called a Gauss transformation, and has the effect of zeroing out elements below the diagonal of the $k-1$th column of $A^{(k-1)}$. P_k is chosen so that among rows $k-1$ through n, the row whose $k-1$th element is the largest in modulus is swapped with the $k-1$th row. This process is known as partial pivoting, and is used to limit growth of roundoff errors. The code used to perform Gaussian elimination is shown in Fig. 4.5. Our fault-tolerant parallel implementation of Gaussian elimination based on the code of Fig. 4.5 used a row cyclic data distribution for A and b. As before, in a p processor system, the data was distributed over processors 0 through $p-2$ while processor $p-1$ computed an extra strip of data by summing rows of A and b; i.e., while processor i $(0 \leq i \leq p-2)$ received rows $i, i+p-1, i+2(p-1), \ldots, i+n-p+1$ of A and b, an $\frac{n}{p-1} \times (n+1)$ matrix S_A was computed and stored on processor $p-1$ according to the following equation:

$$S_A = E^T[A|b] \tag{4.16}$$

```
/* b is stored in column n of A */
for(k=0;k<n-1;k++)
{
  /* Find pivot */
  max = abs(A[k][k]); pivot = k;
  for(i=k;i<n;i++)
    if(abs(A[i][k])>max)
    {
      max = abs(A[i][k]); pivot = i;
    }
  /* Check if singular */
  if(max==0)
  /* Matrix is singular */
  exit();
  /* Swap pivot row with kth row */
  for(j=k;j<n+1;j++)
  {
    tmp = A[k][j];
    A[k][j] = A[pivot][j];
    A[pivot][j] = tmp;
  }
  /* Perform elimination */
  for(i=k;i<n;i++)
    for(j=k;j<n+1;j++)
      A[i][j] = A[i][j] - A[i][k] * A[k][j] / A[k][k];
}
```

Figure 4.5 Gaussian elimination program.

where E is the matrix introduced in Subsection 4.1.2. We define the augmented matrix A' as follows:

$$A' = \begin{bmatrix} A|b \\ S_A \end{bmatrix} \tag{4.17}$$

As before, processors were grouped into check groups of three with each processor computing checksums over the rows of A' owned by the processor succeeding it in its check group. To accomplish this step, processor $g_k^{(i)}$ (which, as before, denotes the kth processor in the ith check group), had to be communicated its own rows as well as its successor's rows at the start of the computation. This step may be represented as

$$rcs_{A'} = A'e \tag{4.18}$$

Here, e denotes the all-1's column vector with $n+1$ elements; $rcs_{A'}$ is thus an $\frac{np}{p-1}$ column vector. A column checksum is also computed over the matrix A' as follows:

$$ccsL_{A'}^T = f^T A' \tag{4.19}$$

where f^T denotes the all-1's row vector with $\frac{np}{p-1}$ elements and $ccsL_{A'}^T$ is a row vector with $n+1$ elements. The row vector $ccsL_{A'}^T$ is computed and stored on one of the processors, say processor 0. For reasons that we explain below, another row vector with $n+1$ elements called $ccsU_{A'}^T$ is also maintained on processor 0, which is initialized to all zeroes. We also need to maintain a $\frac{n}{p-1} \times (n+1)$ matrix SU_A on processor $p-1$, which is also initialized to all zeroes. The data distribution for a system of 10 linear equations on a 6-processor system is shown in Fig. 4.6. Once row and column checksums have been computed, Gauss transforms may be applied to A' and the row checksums in the usual manner, with one caveat. The rows of S_A are not included in the pivoting process to prevent an inadvertent introduction of linear dependence into the system. The kth transformation step may be represented as

$$[A'^{(k)}|rcs_{A'}^{(k)}] = M_k P_k [A'^{(k-1)}|rcs_{A'}^{(k-1)}] \tag{4.20}$$

where, as before, the superscript (k) denotes that k transforms have taken place. Elements $k-1$ through $n-1$ of row $\lfloor \frac{k-1}{p-1} \rfloor$ of SU_A are also updated by adding the corresponding elements of the

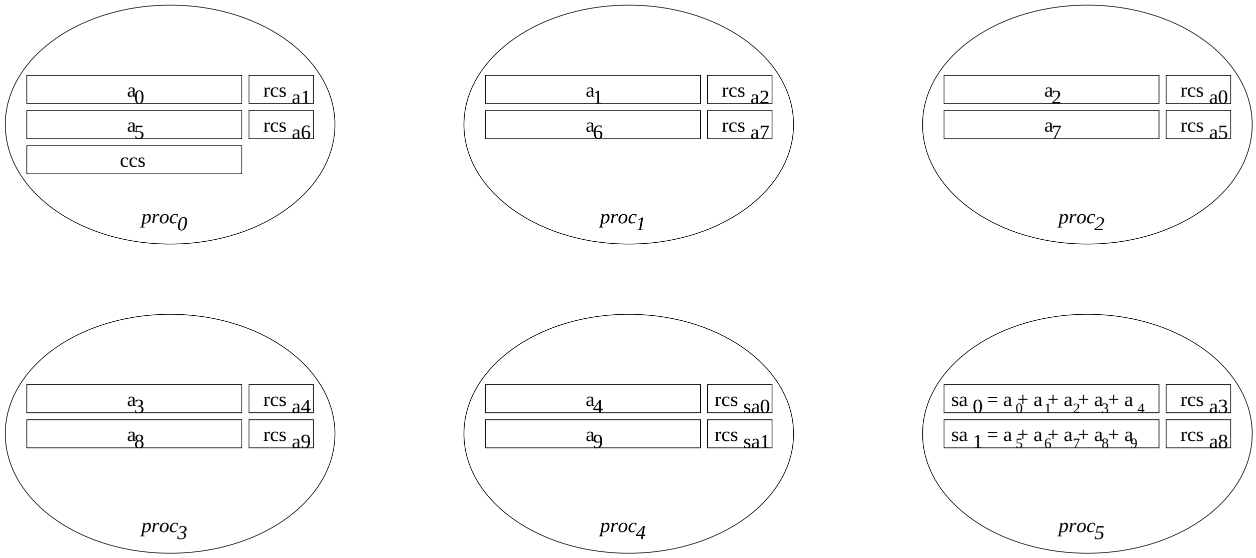


Figure 4.6 Data distribution for fault-tolerant Gaussian elimination

pivot row to it; i.e., processor $p-1$ carries out updates according to the following equation:

$$(SU_A^{(k)})_{\frac{k-1}{p-1}\ k-1\ldots n-1} = (SU_A^{(k-1)})_{\frac{k-1}{p-1}\ k-1\ldots n-1} + pr^{(k-1)}_{\frac{k-1}{p-1}\ k-1\ldots n-1} \tag{4.21}$$

where $pr^{(k-1)}$ denotes the pivot row for the $k-1$th iteration. It may be verified that $SU_A + S_A$ equals the sum of the row strips of A after each iteration. The pivoting needs to be performed carefully to prevent a faulty processor from suggesting a poor pivot. Each processor thus chooses its local pivot row and communicates it not only to the owner of the $k-1$th row, say *owner*, but also to the predecessor of *owner* in its check group, say $pred_{owner}$. The predecessors of each processor also communicate the row checksums of the local pivot rows of their successors to both *owner* and $pred_{owner}$. The integrity of the local pivots is checked on both *owner* and $pred_{owner}$ by comparing with their row checksums, and then the global pivot row is computed from the local pivot rows on both processors and communicated to all processors. The identity of the owner of the global pivot, say *gpowner*, is also computed on both processors. The two copies of the chosen pivot row may be compared on each processor for equality. Processor *gpowner* as well as the predecessor of *gpowner*, say $pred_{gpowner}$, then receive the $k-1$th row from *owner* and the row checksum of the $k-1$th row from $pred_{owner}$. If the row checksum check for the $k-1$th row passes on both processors, the row checksum for the pivot row is replaced by the row checksum for the $k-1$th row on processor $pred_{gpowner}$, while the pivot row is replaced by the $k-1$th row on processor *gpowner*. If at any stage of the pivot computation a check fails on any processor, normal computations are halted, and each processor uses its row checksums to check the data on its succeeding processor. Note that if the pivot and $k-1$th rows were covered by different rows in S_A, appropriate additions and subtractions need to be made to the relevant rows in S_A as well as the checksums of the corresponding rows, and thus both copies of the $k-1$th row need to be sent to processor $p-1$ and the processor maintaining the row checksums of S_A as well. For example, suppose the ith row is chosen to be the pivot row and exchanged with the $k-1$th row. Then, the ith row needs to be subtracted off row $\lfloor\frac{i}{p-1}\rfloor$ and added to row $\lfloor\frac{k-1}{p-1}\rfloor$ of S_A while the $k-1$th row needs to be added to row $\lfloor\frac{i}{p-1}\rfloor$ and subtracted off row $\lfloor\frac{k-1}{p-1}\rfloor$ of S_A. The corresponding adjustments need to be made to the row checksums as well.

If all checks during the pivot determination process and the comparison check of the two copies of the broadcasted pivot pass on each processor, each processor uses Eq. (4.20) to update its local rows of $A'^{(k-1)}$ and $rcs_{A'}^{(k-1)}$. The column checksums $ccsL_{A'}^{(k-1)}$ and $ccsU_{A'}^{(k-1)}$ are also modified as follows:

$$\begin{aligned}
(ccsU_{A'}^{(k)})^T_{k-1\ldots n+1} &= (ccsU_{A'}^{(k-1)})^T_{k-1\ldots n+1} + a'^{(k-1)}_{k-1\ k-1\ldots n+1} \\
(ccsL_{A'}^{(k)})^T_{i} &= (ccsL_{A'}^{(k-1)})^T_{i} - \frac{a'^{(k-1)}_{k-1\ i}}{a'^{(k-1)}_{k-1\ k-1}}(ccsL_{A'}^{(k-1)})^T_{k}\ \ k-1 \leq i \leq n
\end{aligned} \tag{4.22}$$

As a result of the update in Eq. (4.22), the sum of $(ccsU_{A'}^{(k)})^T_i$ and $(ccsL_{A'}^{(k)})^T_i$ equals the sum of the elements in the ith column of $A'^{(k)}$. Note that in order to update the row checksum for the ith row, element $a'^{(k-1)}_{i\ k-1}$ is required by the processor responsible for checking the owner of the ith row. Element $a'^{(k-1)}_{i\ k-1}$ is present on the succeeding processor. Thus elements $a_{k-1\ldots n-1\ k-1}$ need to be communicated to preceding processors by their owners to enable them to perform row checksum updates as described by Eq. (4.20). After these elements are received, these elements are communicated to processor 0, which compares the values of $(ccsL_{A'}^{(k-1)})^T_{k-1}$ with $e^T_{k-1\ldots n-1}a_{k-1\ldots n-1\ k-1}$. The results of these checks are communicated to all other processors, which perform the next step of the update using Eq. (4.20) only if the check passes. If a processor other than processor 0 has communicated incorrect data values for the row checksum update, this will likely be detected by the check on processor 0 due to the single-fault assumption. If processor 0 is faulty and has communicated incorrect data values, the check might still pass. However, by the single-fault assumption, the data values communicated by all other processors are error-free. Thus only processor 0's row checksums, which are updated on its check processor, may be updated incorrectly. However, this incorrect update would still lead to processor 0 being identified as the single faulty processor the next time a row checksum check was applied. If either check fails, the normal execution of the algorithm is halted and each processor proceeds to check the rows of its succeeding processor by performing a row checksum check. If a single processor is faulty and has corrupted its data, the syndrome may be used to identify the faulty processor f. Subsequently, if $f \neq p-1$ (i.e., the faulty processor is not the check processor), the check strips $S_A^{(k-1)}$ and $SU_A^{(k-1)}$ may be used to recover the corrupted data by using

$$G^T S_A^{(k-1)} = S_A^{(k-1)} + SU_A^{(k-1)} - F^T A^{(k-1)} \tag{4.23}$$

where G and F are as defined in Subsection 4.1.2. The check processor then takes over the computations of the faulty processor for the remainder of the algorithm. If the check processor was the faulty processor in the first place, the remaining processors execute the algorithm to completion after location of the faulty processor. As in the previous two algorithms, if recovery from subsequent failures is desired, then one of the surviving processors needs to be designated as a check processor and the recovered data needs to be redistributed on the rest of the surviving processors. A new redundant strip is computed by the check processor, and the algorithm proceeds as before.

4.2 Experimental Results

As noted earlier, ABFT schemes are attractive since they can be implemented on general-purpose multiprocessors without requiring extra hardware or system software modifications. In this section we present performance overheads for the matrix multiplication, QR factorization, and Gaussian elimination algorithms with single-fault recovery discussed in Section 4.1. We also present results for the QR algorithm with double-fault recovery to indicate that the overhead for the multiple-fault case is not prohibitive and tends to a small constant overhead. Furthermore, we present results on single-fault locating (but not correcting) versions of the same applications, which may be achieved by doing away with the check strip and the check processor and instead, using all of the processors for computations on the original data. For the QR algorithm, we also present results for the double-fault locating case. The testbeds used were a 16-processor Intel iPSC/2 hypercube and a 15-processor Intel Paragon (both distributed-memory message-passing multicomputers). Our encodings guarantee that single faults can be located or recovered from (depending on which algorithm is being run) in the event that the faults are detected in the first place. Non-detection of faults can occur if they lead to compensating errors, or if the magnitude of the data corruption due to the fault is so small that the deviation from the nonfaulty results is not greater than the tolerance.

4.2.1 Timing Overheads

Figures 4.7, 4.8 and 4.9 show the timing overheads for the algorithms for single-fault location and correction over the basic algorithms on various matrix dimensions. The experiments were performed on a 16-processor Intel iPSC/2. The overhead for the algorithms for single-fault correction is shown for two cases: the case when no fault occurred, when only a diagnosis of the system state had to be performed, and the case when a single fault was actually injected, when, following the diagnosis of the faulty processor, the corrupted data had to be recovered by using the good data and the check strip. The overhead for the single-fault location algorithm becomes very low for large matrix sizes. Overheads for the single-fault location algorithms asymptotically go to zero with increased matrix size. Overheads for the single-fault correction algorithm are somewhat higher but are also very modest for moderately large matrix sizes. For each of the algorithms, it can be shown that the overheads asymptotically tend to $\frac{1}{p-1}$ for a p-processor system. Since we used a 16-processor system, the overhead for the single-fault correction algorithm can be expected to approach 7% for large matrix sizes. We observe from the figures that even for the modest matrix sizes on which we obtained timing results, the overhead for the single-fault correction algorithm is only around 10-15%. Another point to note is that there is very little extra overhead for the recovery phase, especially for large matrix sizes. This result is not surprising since the fractional overhead contributed by the recovery phase decreases linearly with n, the matrix dimension.

In Fig. 4.10, we indicate the timing overhead for the QR factorization algorithm with double-fault location and recovery. The experiments were performed on a 15-processor Intel Paragon. In the double-fault location case, as for single-fault location, the extra computation introduced is still $O(n^2)$ compared to $O(n^3)$ for the original algorithm. Thus the overhead can be expected to become negligible for large matrix sizes. For the double-fault recovery case, the asymptotic overhead can be expected to approach $\frac{t}{p-t}$, where t is the number of processors operating on redundant data, and p is the total number of processors. We used 15 processors, two of which were used to compute on redundant data (this is sufficient for recovering from any pattern of two faults), and thus the asymptotic overhead could be expected to approach approximately 15%. We observe that for matrices of size 500×500, the overhead is already close to 15%. Although the recovery phase, as in the single-fault case, adds only $O(n^2)$ operations and can be expected to contribute a negligible

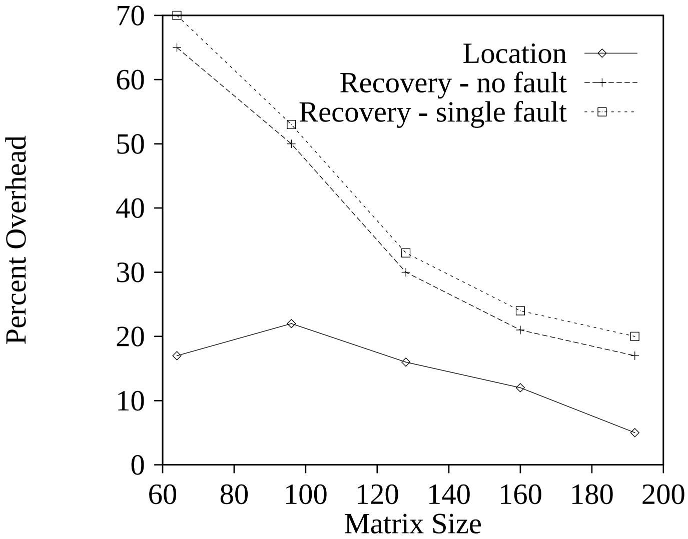


Figure 4.7 Timing overhead for matrix multiplication for the single-fault case.

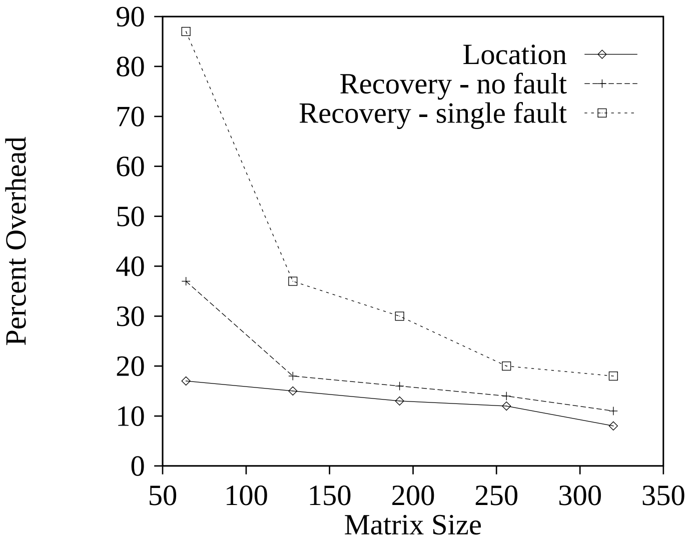


Figure 4.8 Timing overhead for QR factorization for the single-fault case.

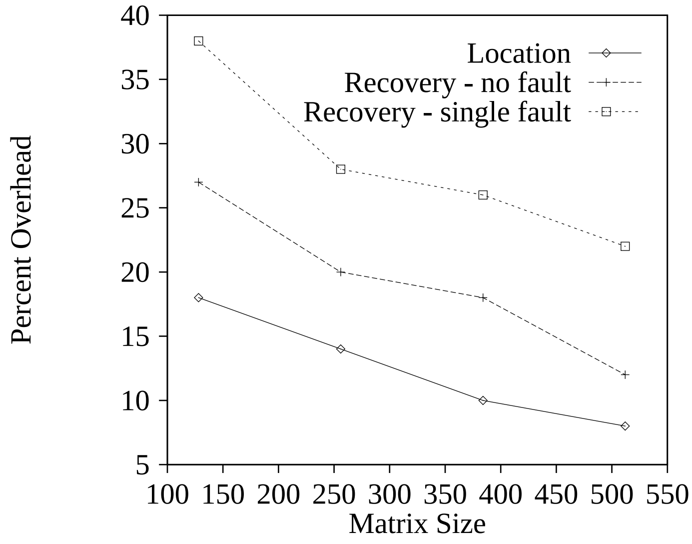


Figure 4.9 Timing overhead for Gaussian elimination for the single-fault case.

overhead for large enough problem sizes, for the problem sizes in our experiments, the recovery phase still represented a significant overhead whenever faults occurred during the run. However, the overhead is around 35% for two-fault recovery for the largest problem sizes we considered. Note that if the number of processors in the system were greater, the overheads for each case would drop further.

4.2.2 Fault coverage

To determine fault-coverage results, we injected transient and permanent bit- and word-level faults in floating-point computations and computed the percentage of times these were detected. We determined the threshold to be used by the simplified error analysis method suggested in [38]. Since most of the undetected faults were not detected due to their very marginal effects on the data (such as affecting the least significant 5 bits in a 23-bit mantissa), we determined new coverage results for faults causing errors that we classified as *significant errors*, which may be loosely defined as errors whose magnitude was more than 10 times the roundoff error which would normally occur on a fault-free system. For details of the definitions and exact methodology of determining error coverage we refer the reader to [38].

The fault-coverage results are summarized in Tables 4.1, 4.2, and 4.3. The coverages for location and recovery are reported for only those cases when the injected faults were detected. Note that in some cases, the error location coverage is less than 100%. An incorrect diagnosis can occur due

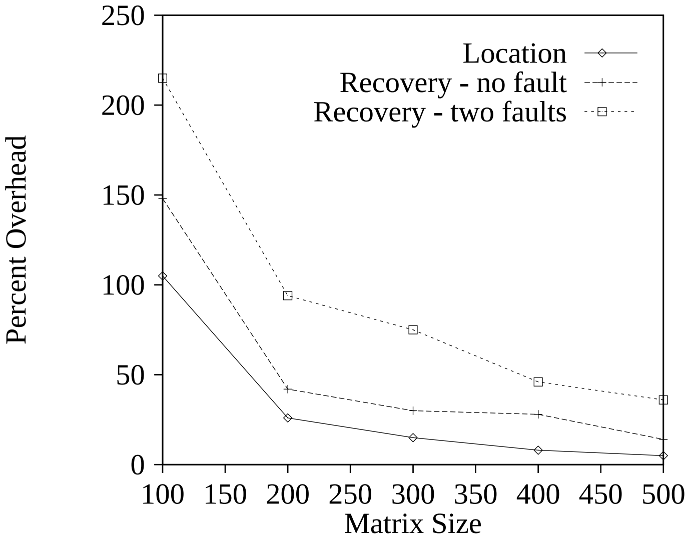


Figure 4.10 Timing overhead for QR factorization for the double-fault case.

Table 4.1 Single-fault coverage for matrix multiplication.

Fault Types	Error Detection Coverage	Significant Error Coverage	Fault Location Coverage	Fault Recovery Coverage
Transient Bit-level	73	94	100	100
Transient Word-level	100	100	100	100
Permanent Bit-level	78	95	100	100
Permanent Word-level	100	100	100	100

to a transient fault corrupting a checksum, rather than any of the original data elements, so that the faulty processor flags an error, but its check processor does not, since all of the data elements communicated to it by the faulty processor are error-free. In these cases, we may incorrectly flag as faulty the processor being checked by the faulty processor, but since the data computed by the faulty processor is error-free, the recovery phase goes through correctly, and we may restart the computation from the point of failure. Error coverage results for the fault-recovery algorithms are always 100% since our algorithms for reconstructing the data yield correct results if the data involved in the reconstruction is fault-free.

Table 4.2 Single-fault coverage for QR factorization.

Fault Types	Error Detection Coverage	Significant Error Coverage	Fault Location Coverage	Fault Recovery Coverage
Transient Bit-level	70	90	100	100
Transient Word-level	100	100	100	100
Permanent Bit-level	74	86	100	100
Permanent Word-level	100	100	100	100

Table 4.3 Single-fault coverage for Gaussian elimination.

Fault Types	Error Detection Coverage	Significant Error Coverage	Fault Location Coverage	Fault Recovery Coverage
Transient Bit-level	86	100	98	100
Transient Word-level	100	100	98	100
Permanent Bit-level	93	99	100	100
Permanent Word-level	100	100	100	100

4.3 Summary

In this chapter, we have described the modification of three parallel numerical algorithms to make them error locating and correcting. The redesign employs the basic error location and recovery framework described in the previous chapter. The checks themselves are algorithm-specific and impose very low overheads while also providing high error coverage, as demonstrated by our results.

CHAPTER 5

COMPILER-ASSISTED GENERATION OF ERROR-DETECTING PARALLEL PROGRAMS

We have developed an automated, compile-time approach to generating error-detecting parallel programs. The compiler is used to identify statements implementing affine transformations within the program and automatically insert code for computing, manipulating, and comparing checksums in order to check the correctness of the code implementing affine transformations. Statements that do not implement affine transformations are checked by duplication. Checksums are reused from one loop to the next if this is possible, rather than recomputing checksums for every statement. A global dataflow analysis is performed to determine points at which checksums need to be recomputed. We also use a novel method of specifying the data distributions of the check data using directives provided by the High Performance Fortran (HPF) [10] standard so that the computations on the original data and the corresponding check computations are performed on different processors.

5.1 A Motivational Example

We will use the code fragment in Fig. 5.1 to illustrate the development of an error-detecting parallel program. The code fragment solves a system of equations using the Jacobi iterative technique. This and other similar code fragments occur in numerical routines designed to solve partial differential equations used in modeling physical phenomena. Our eventual output is designed to be an error-detecting parallel program computing the same results as the serial program. The serial program is augmented with HPF data distribution directives to aid in the generation of the parallel program; this information is also used in generating the error-detecting version. Note that in the absence of the data distribution annotations, our compiler would be able to generate a serial version of the program useful for detecting transient errors.

Before the actual generation of the checksum-based checks is performed, a version of the original program is created by duplicating all array assignments in the program. For the Jacobi example, this duplication results in the code fragment shown in Fig. 5.2.

The next step in the process is to determine duplicate assignment statements in the code that implement affine transformations. These can be identified by examining the syntactic structure

```
      PROGRAM jacobi
      INTEGER p(4,4)
      REAL a(1000,1000)
      REAL b(1000,1000)
      INTEGER k, j, i

!HPF$ PROCESSORS :: p(4,4)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO p :: a, b

      DO k = 1,100
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1       ) / 4
          END DO
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 5.1 Code fragment implementing Jacobi's iterative technique.

```
      DO k = 1,100
        DO j = 2,999
          DO i = 2,999
            $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j + 1)
     1      ) / 4
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO j = 2,999
          DO i = 2,999
            $b(i,j) = $a(i,j)
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 5.2 Jacobi kernel with duplicate array assignments.

of the statement and by verifying that the dependences in which the statement is involved satisfy some additional conditions. Both the assignment statements that were newly introduced into the example Jacobi program satisfy the criteria for being affine transformations. These statements are then replaced by statements that transform checksums on array elements rather than transforming the array elements themselves. The exact methodology for determining when to introduce and transform checksums is explained later and forms the bulk of this chapter. One of the array dimensions is chosen to compute the checksums over (the dimension subscripted by `j` for our example), and the loop traversing this dimension in the original code is deleted. The code after introducing checksum transformations is shown in Fig. 5.3.

Information about available checksums is then propagated across statements in the program. This information is used to recompute checksums at points where checksum values are required but are not available. For the code shown in Fig. 5.3, the second checksum statement requires the values of `$cs2_a(i)` for $2 \leq$ `i` ≤ 999, but information propagation across statements is able to determine that these values are already available when the second assignment statement is encountered. However, checksums `$cs2_a` and `$cs2_b` need to be computed prior to the start of the `k` loop since these are required within the loop body. Checks are generated at intermediate points in the program only where necessary (this will be elaborated on in Section 5.3) and also at the end

```
      DO k = 1,100
        DO i = 2,999
          $cs2_a(i) = ($cs2_b(i - 1) + $cs2_b(i + 1) + ($cs2_b(i) - b(i
     1    ,999) + b(i,1)) + ($cs2_b(i) + b(i,1000) - b(i,2))) / 4
        END DO
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO i = 2,999
          $cs2_b(i) = $cs2_a(i)
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 5.3 Jacobi kernel after introduction of checksum statements.

of the program. The program after information propagation and checksum and check generation have been performed is shown in Fig. 5.4.

We assume that the program being transformed has been annotated with data distribution information (i.e., information about how arrays in the program are to be distributed over a specified processor topology) to aid in parallelization. The data distribution information about the original arrays is used to choose a suitable data distribution for the extra data that was introduced in the form of checksums and possibly some extra arrays. This step may necessitate introducing an extra dimension for a checksum variable so that each new checksum now covers a smaller portion of the array than the old checksum, which may also require the checksum transformation statements to be modified accordingly. Data distribution information is then specified for checksums and any extra arrays that may have been introduced so that computations on the original data and the corresponding check data are performed on different processors. The code after data distribution information has been introduced for checksums is shown in Fig. 5.5.

Finally, a parallelizing compiler for a distributed-memory machine (in our case, Paradigm [40, 41, 42]) is used to generate an error-detecting parallel program based on the code incorporating checks and data distribution information.

We would like to point out at this point that the parallel version of the code in Fig. 5.1 would require $O(kn^2)$ computations in all, where n is the matrix size and k is the number of iterations executed, while the parallel version of the code in Fig. 5.5 would perform $O(n^2+kn)$ extra operations due to the checksum computation, updates, and comparison. Thus the overhead due to the check operations can be expected to be small for large problem sizes. By contrast, a straighforward duplication and check approach, such as one based on the code of Fig. 5.2, would require more than double the number of operations as the original program. The approach discussed in [29] would also be able to check the bodies of the two loops of the Jacobi code by checksum manipulations since these happen to be performing linear transformations. However, since information about available checksums is not computed across statements, checksums would be regenerated prior to each loop nest enclosing a checksum manipulation statement for all checksums required by the statement. Following the execution of the checksum code and the original loop being checked by it, a checksum-based check, which is illustrated in Fig. 5.6, would be generated for the array elements being assigned by the loop being checked. Note that recomputing the checksums and generating the checks for each loop incurs $O(n^2)$ overhead for each iteration of the `k` loop. By contrast, our approach, which performs dataflow analysis to determine that `$cs2_b` is available upon each entry

```
  DO $i1 = 2,999
    DO $i2 = 2,999
      $a($i1,$i2) = a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    DO $i2 = 1,1000
      $b($i1,$i2) = b($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    $cs2_a($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $cs2_a($i1) = $cs2_a($i1) + a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    $cs2_b($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $cs2_b($i1) = $cs2_b($i1) + b($i1,$i2)
    END DO
  END DO
  DO k = 1,100
    DO i = 2,999
      $cs2_a(i) = ($cs2_b(i - 1) + $cs2_b(i + 1)
1+ ($cs2_b(i) - b(i,999) + b(i,1)) + ($cs2_b(i)
2+ b(i,1000) - b(i,2))) / 4
    END DO
    DO j = 2,999
      DO i = 2,999
        a(i,j) = (b(i - 1,j) + b(i + 1,j) +
1b(i,j - 1) + b(i,j + 1)) / 4
      END DO
    END DO
    DO i = 2,999
      $cs2_b(i) = $cs2_a(i)
    END DO
    DO j = 2,999
      DO i = 2,999
        b(i,j) = a(i,j)
      END DO
    END DO
  END DO
  DO $i1 = 2,999
    $T($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $T($i1) = $T($i1) + a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    IF (compare($T($i1),$cs2_a($i1)) .EQ. 1)
1CALL error_handler()
  END DO
  DO $i1 = 2,999
    $T_0($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $T_0($i1) = $T_0($i1) + b($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    IF (compare($T_0($i1),$cs2_b($i1)) .EQ. 1)
1CALL error_handler()
  END DO
  END
```

Figure 5.4 Jacobi code with checks.

```
      PROGRAM jacobi
      DOUBLE PRECISION $T_0(1000,4)
      DOUBLE PRECISION $T(1000,4)
      INTEGER $p
      INTEGER $i2
      INTEGER $i1
      DOUBLE PRECISION $cs2_a(1000,4)
      DOUBLE PRECISION $cs2_b(1000,4)
      DOUBLE PRECISION $a(1000,1000)
      DOUBLE PRECISION $b(1000,1000)
      REAL a(1000,1000)
      REAL b(1000,1000)
      INTEGER k, j, i

!HPF$ PROCESSORS :: p(4,4)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO p :: a, b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, BLOCK) ONTO p :: template$0(1000, 1000)
!HPF$ ALIGN (hpf$0,hpf$1) WITH template$0(hpf$0 + 250,hpf$1) WRAP :: $a
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, BLOCK) ONTO p :: template$1(1000, 1000)
!HPF$ ALIGN (hpf$0,hpf$1) WITH template$1(hpf$0 + 250,hpf$1) WRAP :: $b
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO p :: TEMPLATE$2(1000,4)
!HPF$ ALIGN $cs2_a(hpf$0,hpf$1) WITH TEMPLATE$2(hpf$0+250,hpf$1+1) WRAP
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO p :: TEMPLATE$3(1000,4)
!HPF$ ALIGN $cs2_b(hpf$0,hpf$1) WITH TEMPLATE$3(hpf$0+250,hpf$1+1) WRAP

      DO $i1 = 2,999
        DO $i2 = 2,999
          $a($i1,$i2) = a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,1000
          $b($i1,$i2) = b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          $cs2_a($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $cs2_a($i1,1) = $cs2_a($i1,1) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $cs2_a($i1,$p) = $cs2_a($i1,$p) + a($i1,($p - 1) * 250 + $i
     1        2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $cs2_a($i1,4) = $cs2_a($i1,4) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          $cs2_b($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $cs2_b($i1,1) = $cs2_b($i1,1) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $cs2_b($i1,$p) = $cs2_b($i1,$p) + b($i1,($p - 1) * 250 + $i
     1        2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $cs2_b($i1,4) = $cs2_b($i1,4) + b($i1,$i2)
        END DO
      END DO
      DO k = 1,100
        DO i = 2,999
          $cs2_a(i,1) = ($cs2_b(i - 1,1) + $cs2_b(i + 1,1) + $cs2_b(i,1
     1    ) + (b(i,2 - 1) - b(i,250)) + $cs2_b(i,1) + (b(i,250 + 1) - b
     2    (i,2))) / 4
          DO $p = 2,3
            $cs2_a(i,$p) = ($cs2_b(i - 1,$p) + $cs2_b(i + 1,$p) + $cs2_
     1      b(i,$p) + (b(i,250 * ($p - 1)) - b(i,250 * ($p - 1) + 250))
     2       + $cs2_b(i,$p) + (b(i,250 * ($p - 1) + 251) - b(i,250 * ($
     3      p - 1) + 1))) / 4
          END DO
          $cs2_a(i,4) = ($cs2_b(i - 1,4) + $cs2_b(i + 1,4) + $cs2_b(i,4
     1    ) + (b(i,751 - 1) - b(i,999)) + $cs2_b(i,4) + (b(i,999 + 1) -
     2     b(i,751))) / 4
        END DO
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO i = 2,999
          $cs2_b(i,1) = $cs2_a(i,1)
          DO $p = 2,3
            $cs2_b(i,$p) = $cs2_a(i,$p)
          END DO
          $cs2_b(i,4) = $cs2_a(i,4)
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $T($i1,1) = $T($i1,1) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $T($i1,$p) = $T($i1,$p) + a($i1,($p - 1) * 250 + $i2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $T($i1,4) = $T($i1,4) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          IF (compare($T($i1,$p),$cs2_a($i1,$p)) .EQ. 1) CALL error_han
     1    dler()
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          $T_0($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $T_0($i1,1) = $T_0($i1,1) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $T_0($i1,$p) = $T_0($i1,$p) + b($i1,($p - 1) * 250 + $i2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $T_0($i1,4) = $T_0($i1,4) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          IF (compare($T_0($i1,$p),$cs2_b($i1,$p)) .EQ. 1) CALL error_h
     1    andler()
        END DO
      END DO
      END
```

Figure 5.5 Jacobi code incorporating checks and data distribution specifications.

```
      DO k = 1,100
C Recompute checksums of b
        DO i = 2,999
          $cs2_b(i) = 0
          DO j = 2,999
            $cs2_b(i) = $cs2_b(i) + b(i,j)
          ENDDO
        ENDDO
        DO i = 2,999
          $cs2_a(i) = ($cs2_b(i - 1) +
     1$cs2_b(i + 1) + ($cs2_b(i) - b(i,999)
     2+ b(i,1)) + ($cs2_b(i) + b(i,1000) -
     3b(i,2))) / 4
        END DO
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) +
     1b(i,j - 1) + b(i,j + 1)) / 4
          END DO
        END DO
C Check array a using $cs2_a
        DO i = 2,999
          T = 0
          DO j = 2,999
            T = T + a(i, j)
          ENDDO
          IF (compare(T,$cs2_a(i)) .EQ. 1) CALL
     1error_handler()
        ENDDO
C Recompute checksums of a
        DO i = 2,999
          $cs2_a(i) = 0
          DO j = 2,999
            $cs2_a(i) = $cs2_a(i) + a(i,j)
          ENDDO
        ENDDO
        DO i = 2,999
          $cs2_b(i) = $cs2_a(i)
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
C Check array b using $cs2_b
        DO i = 2,999
          T = 0
          DO j = 2,999
            T = T + b(i, j)
          ENDDO
          IF (compare(T,$cs2_b(i)) .EQ. 1) CALL
     1error_handler()
        ENDDO
      END DO
      END
```

Figure 5.6 Transformed Jacobi kernel with regeneration of checksums before each loop.

into the `k` loop and that `$cs2_a` is available prior to the second manipulation statement, does not need to regenerate checksums and perform checks for each loop. Thus only $O(n)$ overhead is incurred for checksum manipulation for each iteration of the `k` loop.

5.2 System Overview

In this section we give an overview of the modules comprising our system. The input set accepted by our compiler consists of Fortran programs with HPF data distribution annotations [10]. Parafrase-2 [43, 44], a parallelizing compiler for shared-memory machines developed at the University of Illinois, is used as a front-end module to parse in the input program, build an abstract syntax tree representation of the program, perform dependence analysis, and build the flowgraph. The original compiler was not able to utilize information provided by HPF data distribution information and has been modified to do so [41, 45]. Apart from being a state-of-the-art optimizing and parallelizing compiler, Parafrase-2 has also been designed as a developmental tool. The compiler may be easily augmented by the addition of passes to use and modify the information stored in its internal data structures. Several passes have been added to achieve our goal of generating error-detecting versions of programs. The first pass is responsible for generating duplicate statements corresponding to the statements in the program that operate on arrays. These statements perform the same transformations as the original statements, but on different arrays, which we refer to as shadow arrays. The second pass then attempts to replace duplicate statements by statements that transform checksum variables computed by summing over one of the array dimensions wherever possible. The third pass performs information propagation and check code insertion. Information about available checksums and shadow arrays is propagated across statements in the program. If it is determined that a checksum is needed but not available at a point in the program, it is regenerated. Along with regenerating the checksum, checks are generated comparing the elements being summed with the shadow elements, if the latter are available at that point. Similarly, at points where shadow elements are required but are not available, they are copied over from the corresponding original array values. If checksums covering these array values are available, a check is also generated to check the elements being copied over. The fourth pass is responsible for taking data distribution information into consideration and specifying suitable data distributions for the check data that was introduced. This step may also involve expansion of certain dimensions of the checksum variables that were introduced and consequent modification of some of the statements transforming the checksums. Data distribution information is used to specify distributions for checksums and shadow arrays in such a manner that an original array element and the corresponding shadow array element or checksum variable that checks it reside on different processors. The modified program is input to Paradigm [40], a distributed-memory parallelizer developed at the University of Illinois, which can generate message-passing code for a variety of target multicomputers. The final output is an error-detecting parallel program for distributed-memory multicomputers. The various modules in our system and their interactions are illustrated in Fig. 5.7.

5.3 Algorithms for Check Code Generation

5.3.1 Statement duplication

The pass for duplicating statements that operate on arrays is fairly straightforward. However, not only does the pass create duplicate assignment statements for those which assign to or use

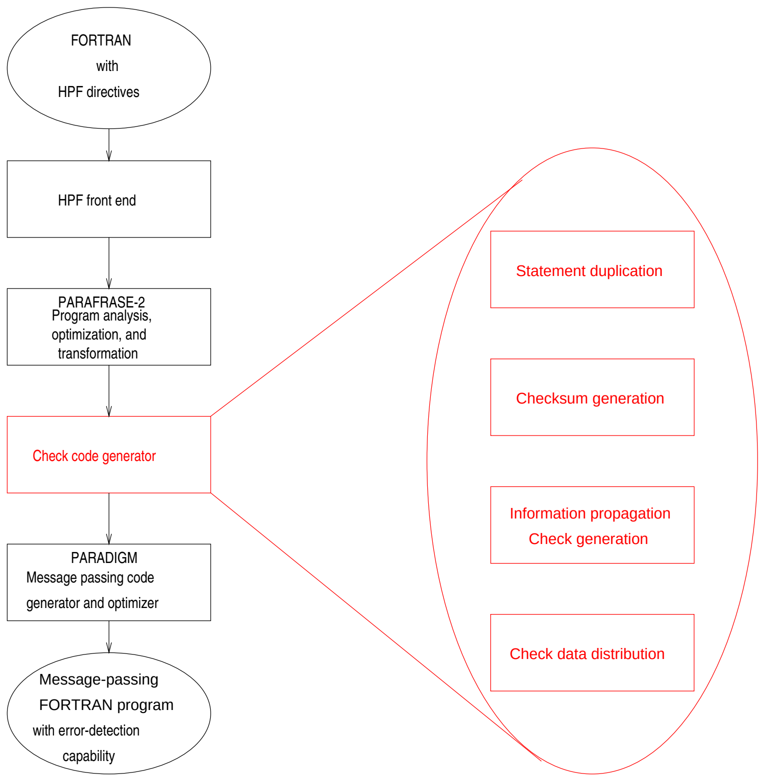


Figure 5.7 Overall organization of system for generating error-detecting parallel code.

arrays directly, but also it duplicates statements that use array elements indirectly. For example, a statement that used a scalar to which was assigned an array value prior to reaching the statement would also be duplicated. Also, if loop expressions or `IF` conditionals depend directly or indirectly on array values, then the entire loop or `IF` statement, including its body, is duplicated. By duplication of a statement, we mean that a second statement is created performing the same transformations as the original statement, but with array elements and array-dependent variables replaced by different elements that we refer to as shadows. To perform statement duplication according to the rules described, it is necessary to determine which scalar variables use array values in their definition. This problem can be solved as a standard reaching definitions problem [46]. An example output of this pass is shown in Fig. 5.2 for the code in Fig. 5.1.

5.3.2 Checksum introduction

Once statement duplication has been performed, the pass for determining affine transformations and replacing array elements by checksums is run. However, prior to this, a loop distribution pass is run to separate out duplicate statements operating on shadow elements and the corresponding original elements into separate loops. The loop distribution pass also has the effect of separating out different duplicate statements into different loops, which increases opportunities for checksum introduction, as we explain later in this section. Loop distribution on the code in Fig. 5.2 yields the code shown in Fig. 5.8.

Given an array $A(l:u)$ which is transformed by a function f satisfying the following property

$$\sum_{i=l}^{u} f(A(i)) = f(\sum_{i=l}^{u} A(i)) \tag{5.1}$$

we say that the array $A(l:u)$ undergoes a linear transformation. Suppose we have another function g where $g(x) = f(x) + c$, where c is a constant, then clearly we have

$$\sum_{i=l}^{u} g(A(i)) = g(\sum_{i=l}^{u} A(i)) + (l-u)c \tag{5.2}$$

In this case, we say that the array $A(l:u)$ undergoes an affine transformation. For example, the code in Fig. 5.11 performs an affine transformation, while the transformation performed by the code in Fig. 5.9 is not affine. Our approach to generating checksum-based checks is based on identifying duplicated assignment statements that perform a linear or affine transformation on some shadow

```
  DO k = 1,100
    DO j = 2,999
      DO i = 2,999
        $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j
1        + 1)) / 4
      END DO
    END DO
    DO j = 2,999
      DO i = 2,999
        $b(i,j) = $a(i,j)
      END DO
    END DO
  END DO
  DO k = 1,100
    DO j = 2,999
      DO i = 2,999
        a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
1       ) / 4
      END DO
    END DO
    DO j = 2,999
      DO i = 2,999
        b(i,j) = a(i,j)
      END DO
    END DO
  END DO
  END
```

Figure 5.8 Jacobi kernel with duplicated statements after loop distribution.

```
C AVAILARRAY = {$B(1:5,1:10)}, AVAILCS = {$CS2_B(6:10,1:10)}
C REQDARRAY = {$B(1:10,1:10)}, GENARRAY = {$B(6:10,1:10)}

      DO I = 1,10
        DO J = 1,10
          $A(I,J) = $B(I,J)*$B(I,J)
        ENDDO
      ENDDO

      DO I = 1,10
        DO J = 1,10
          A(I,J) = B(I,J)*B(I,J)
        ENDDO
      ENDDO
```

Figure 5.9 Code fragment for shadow array regeneration showing *AVAILARRAY*, *AVAILCS*, *REQDARRAY* and *GENARRAY* sets.

array. The statement is then replaced by one that transforms checksum values rather than the elements of the shadow array. The expression for transforming the checksum values is derived from Eq. (5.1) or (5.2), as the case may be. The loop traversing the array dimension that was summed becomes redundant and may be removed. This removal of the loop often dramatically reduces the overheads contributed by the checksum statement over the statement that it checks.

We first determine perfect loop nests whose bodies consist solely of duplicate statements that are also assignment statements. (Change of flow of control within the loop body due to the presence of an `IF` statement, for example, is not allowed. This is a conservative criterion chiefly designed to make the task of the propagation pass easier.) For the code in Fig. 5.8, both the loop nests enclosing duplicate statements with `j` as the outer loop variable are such loop nests. Next, it is determined if the set of variables used by the subscript expressions in the block is a subset of the loop index variables of the perfect nest enclosing the block. (This restriction is also made in order to make the task of the propagation pass easier.) If these conditions are satisfied, the loop indices of the perfect nest are called the *potential checksum indices* for the statements in the block. For example, the potential checksum indices for both the duplicate assignment statements in Fig. 5.8 are `i` and `j`. Note that loop distribution increases the number of statements enclosed in perfect nests as well as the number of loops in each perfect nest, which results in an increase in the number of potential checksum indices for each statement. This increase benefits later stages of the pass.

Once the potential checksum indices have been determined for each duplicate assignment statement, the syntactic structure of each such statement is examined to determine a subset of the potential checksum indices such that the statement could possibly be replaced by a checksum manipulation by computing checksums over the array dimensions involving these indices. The indices chosen are called the *candidate checksum indices* in order to distinguish them from the potential checksum indices determined earlier. The candidate checksum indices are computed by traversing the syntax tree associated with the statement under consideration in a bottom-up fashion and updating two sets, called the $AFFINE$ and $NOTAFFINE$ sets, for each node, depending on the values of these sets at its children. The $AFFINE$ set at the root of the tree contains the candidate checksum indices for the statement. The rules for updating the $AFFINE$ and $NOTAFFINE$ sets upon traversing the syntax tree in bottom-up fashion are given in Fig. 5.10. Note that the condition for suitability of subscript expressions is primarily to aid the propagation pass and could be made less restrictive if the propagation pass were made more sophisticated.

We illustrate the application of the rules in Fig. 5.10 to the assignment statement shown at the top of Fig. 5.13. This statement is assumed to be enclosed in a perfect loop nest involving loop variables `i` and `j`, similar to the code in Fig. 5.11. The syntax tree and the computation of the $AFFINE$ and $NOTAFFINE$ sets for this statement are shown at the bottom of Fig. 5.13.

Once the set of candidate checksum indices has been computed for each check statement, some additional conditions pertaining to the dependences in which the statements are involved need to be verified before a candidate checksum index can actually be chosen as the index to compute the checksum over. A candidate checksum index that passes all additional tests to determine its validity as a checksum index is called a *valid checksum index*. Once the set of valid checksum indices has been determined for all check statements, one of these may be picked as the variable to sum over. This index will be referred to as the *chosen checksum index*.

The first condition involves dependence cycles that the statement may be involved in. A thorough treatment of dependence analysis of array statements within loops and detailed descriptions of algorithms to determine when various loop transformations and optimizations are valid may be found in [47, 48, 49]. As an example, consider the loop nest shown in Fig. 5.14, which is similar to the first assignment statement in Fig. 5.8 except that the left-hand side array has been changed from `$a` to `$b`. Both `i` and `j` are candidate checksum indices for the statement; however, neither is a valid checksum index since the statement is involved in dependence cycles carried by both the `i` loop as well as the `j` loop. To see this, consider the code that would be generated if `j` were the

(1) Unary expressions $u = \pm e$ where $\pm$ denotes a generic unary operator

$$\begin{aligned} AFFINE(u) &\leftarrow AFFINE(e) \\ NOTAFFINE(u) &\leftarrow NOTAFFINE(e) \end{aligned}$$

(2) Binary addition or subtraction $b = e_1 + e_2$ or $b = e_1 - e_2$

$$\begin{aligned} AFFINE(b) &\leftarrow (AFFINE(e_1) - NOTAFFINE(e_2)) \cup (AFFINE(e_2) - NOTAFFINE(e_1)) \\ NOTAFFINE(b) &\leftarrow NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \end{aligned}$$

(3) Binary multiplication $b = e_1 * e_2$

$$\begin{aligned} AFFINE(b) &\leftarrow ((AFFINE(e_1) - NOTAFFINE(e_2)) \cup (AFFINE(e_2) - NOTAFFINE(e_1))) \\ &\quad -(AFFINE(e_1) \cap AFFINE(e_2)) \\ NOTAFFINE(b) &\leftarrow NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \cup (AFFINE(e_1) \cap AFFINE(e_2)) \end{aligned}$$

(4) Binary division $b = e_1 / e_2$

$$\begin{aligned} AFFINE(b) &\leftarrow AFFINE(e_1) - (NOTAFFINE(e_2) \cup AFFINE(e_2)) \\ NOTAFFINE(b) &\leftarrow NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \cup AFFINE(e_2) \end{aligned}$$

(5) Constant c

$$\begin{aligned} AFFINE(c) &\leftarrow \emptyset \\ NOTAFFINE(c) &\leftarrow \emptyset \end{aligned}$$

(6) Variable i. L is the set of potential checksum indices.

$$\begin{aligned} AFFINE(i) &\leftarrow \emptyset \\ NOTAFFINE(i) &\leftarrow \{i\}, \quad i \text{ is a loop variable} \\ &\quad\ \cup_j \{j \in L\}, \quad \text{otherwise} \end{aligned}$$

(7) Array A. L is as before and S is the set of variables appearing in the array subscripts. A subscript expression is suitable if (a) A appears on the left-hand side, it is of the form i, else it is of the form $i \pm c$, where i is a loop variable, c is a constant, and i does not appear in the subscript expression for any other dimension; or (b) it is of the form c, where c is a constant.

$$\begin{aligned} AFFINE(A) &\leftarrow \cup_{i \in S \cap L}\{i\}, \quad \text{all subscript expressions are suitable} \\ &\quad\ \emptyset, \quad \text{otherwise} \\ NOTAFFINE(A) &\leftarrow \emptyset, \quad \text{all subscript expressions are suitable} \\ &\quad\ \cup_{j \in L}\{j\}, \quad \text{otherwise} \end{aligned}$$

(8) Assignment a of the form $e_1 \leftarrow e_2$

$$\begin{aligned} AFFINE(a) &= AFFINE(e_1) - NOTAFFINE(e_2) \\ NOTAFFINE(a) &= NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \end{aligned}$$

Figure 5.10 Rules for computing $AFFINE$ and $NOTAFFINE$ sets in bottom-up fashion.

```
    DO j = 2,999
      DO i = 2,999
        $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j + 1)
1       ) / 4 + 10
      END DO
    END DO
```

Figure 5.11 Code fragment illustrating affine transformation.

```
    DO i = 2,999
      $cs2_a(i) = ($cs2_b(i - 1) + $cs2_b(i + 1) + ($cs2_b(i) - b(i
1     ,999) + b(i,1)) + ($cs2_b(i) + b(i,1000) - b(i,2))) / 4 + 10 * 998
    END DO
    DO j = 2,999
      DO i = 2,999
        $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j + 1)
1       ) / 4 + 10
      END DO
    END DO
```

Figure 5.12 Checksum code fragment illustrating affine transformation.

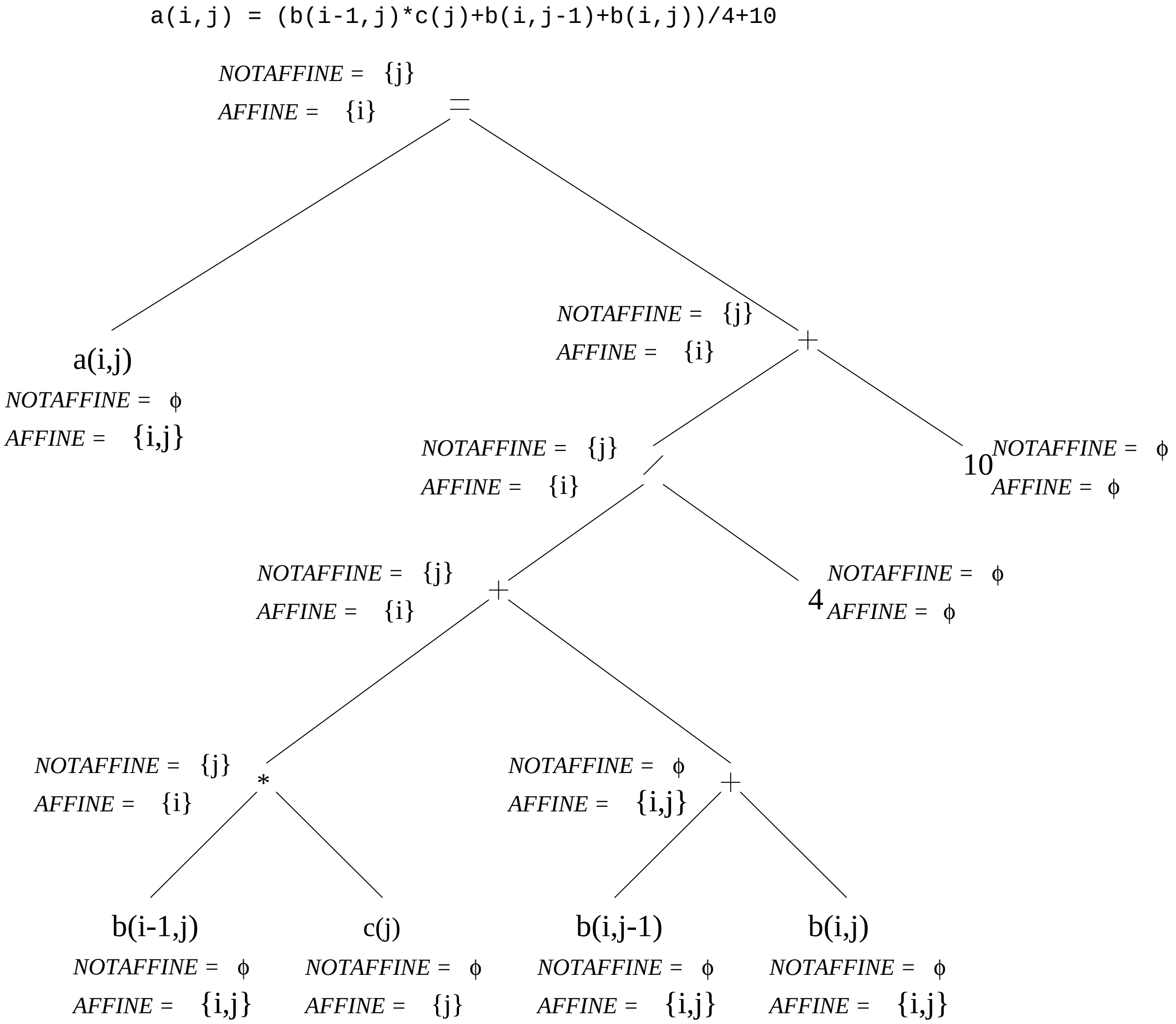


Figure 5.13 Syntax tree showing $AFFINE$ and $NOTAFFINE$ sets for assignment statement in Fig. 5.11.

```
      DO j = 2,999
        DO i = 2,999
          $b(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j
     1      + 1)) / 4
        END DO
      END DO
```

Figure 5.14 Code fragment illustrating dependence cycle

```
      DO i = 2,999
        $cs2_b(i) = ($cs2_b(i - 1) + $cs2_b(i + 1) + ($cs2_b(i) - b(i
     1    ,999) + b(i,1)) + ($cs2_b(i) + b(i,1000) - b(i,2))) / 4
      END DO
      DO j = 2,999
        DO i = 2,999
          $b(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j
     1      + 1)) / 4
        END DO
      END DO
```

Figure 5.15 Checksum code illustrating problem caused by dependence cycle (incorrect code).

chosen checksum index, which is shown in Fig. 5.15. The old value of `$cs2_b(i)` is used for both accesses `b(i,j+1)` as well as `b(i,j-1)`. However, `b(i,j-1)` actually uses values modified during the current iteration of the `i` loop, and thus the old checksum value does not correctly represent the sum over these elements.

Now we state and prove some theorems that indicate when a candidate checksum index is also a valid checksum index for the case of a single statement enclosed in a perfect nest of loops. This case actually covers a large fraction of the situations in practice since loop distribution is applied to the original code, which separates out statements not involved in dependence cycles into separate loop nests.

Theorem 3 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement S. Suppose I_1, the loop variable for the outermost loop L_1, is a candidate checksum index for S. Then I_1 is a valid checksum index for S if there is no flow dependence from S to itself.*
Proof: Since there are no flow dependences from S to itself, all values used by S are assigned prior

```
                          DO L1
                           DO L2
                             :
Candidate checksum index --> DO Lc
                               :
                              DO Lm
                               S
                              ENDDO
                               :
                             ENDDO
                             :
                           ENDDO
                          ENDDO
```

Figure 5.16 Loop nest with single assignment statement in loop body.

to the loop nest. Thus the required checksums of the right-hand side elements may be computed prior to entering the loop nest and may be transformed to generate the new checksum. □

For the next two theorems, a loop nest of the form shown in Fig. 5.16 is considered, with the loop variable for the cth loop L_c being a candidate checksum index. If this were chosen as the checksum index without verifying dependence conditions, the code shown in Fig. 5.17 would be generated, with CS being the checksum statement that would be generated for S. The next two theorems characterize when the code of Fig. 5.17 correctly updates the checksums for checking the code in Fig. 5.16 when dependences from S to itself are taken into account.

Theorem 4 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement S. Suppose I_c, the loop variable for the cth loop L_c, is a candidate checksum index for S. Then I_c is a valid checksum index for S if there is no flow dependence from S to itself carried by loops at level c or greater.*

Proof: We unroll the first $c-1$ loops, creating a separate loop nest for each value taken by I_j, $1 \leq j < c$. In each instance of the unrolled loop nests, we may apply Theorem 3 to conclude that I_c is a valid checksum index. Generating the checksum statement for each instance of the unrolled loops and rerolling to obtain the original loop ordering proves the theorem. □

In the following theorem, condition (3) is only violated in rare cases. We mention when this happens and what to do in this case after the theorem.

```
DO L1
 DO L2
   ⋮
  DO Lc-1
   DO Lc+1
    ⋮
    DO Lm
     CS
    ENDDO
    ⋮
   ENDDO
   DO Lc
    DO Lc+1
     ⋮
     DO Lm
      S
     ENDDO
     ⋮
    ENDDO
   ENDDO
  ENDDO
  ⋮
 ENDDO
ENDDO
```

Figure 5.17 Check code for loop nest of Fig. 5.16.

Theorem 5 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement S. Suppose I_c, the loop variable for the cth loop L_c, is a candidate checksum index for S. Let CS be the checksum statement corresponding to S with I_c chosen as the checksum index, and suppose the following conditions hold:*

(1) L_c does not carry any flow dependences.

(2) There is some valid reordering of the loops such that L_c can be moved inside all loops carrying flow dependences.

(3) There are no dependences between CS and S in the reordered loops.

Then I_c is a valid checksum index for S enclosed within the original loop nest.

Note that this theorem implies that it is sufficient for the reordering of condition (2) to exist, but it need not be applied.

Proof: It is clear that if the reordering of condition (2) exists, then Theorem 4 applies, and I_c is a valid checksum index for the reordered loops. In the reordered loops, there are two types of dependences: dependences from S to itself and dependences from CS to itself. Dependences from CS to S and dependences from S to CS are excluded by condition (3) of the theorem. It is clear that if the loops are reordered back to their original ordering after CS has been introduced, then the dependences from S to itself are not violated. Dependences from CS to itself are caused by the reading and writing of checksum variables. A checksum variable in CS is derived from the corresponding array variable in S and has identical subscript expressions except for the subscript involving I_c, which is missing. Thus the dependence vectors from CS to itself are identical to the dependence vectors from S to itself except for the component corresponding to the loop L_c, which is missing. Thus if interchanging loops I_j, I_k does not violate dependences from S to itself, then it also does not violate dependences from CS to itself. Thus reordering the loops to obtain the original ordering after CS has been introduced does not violate dependences from CS to itself. Thus reordering loops back to their original ordering after checksum introduction is valid since no dependences are violated. □

Condition (3) of Theorem 5 may be violated only if the original statement assigns to and accesses the same array with different subscript expressions for the subscript involving the checksum index, as in the code in Fig. 5.18, which results in some of the array elements assigned to by the original code being added and subtracted off the checksum statement. This introduces flow dependences between the original statement and the checksum statement and antidependences from the latter to the former, which may prevent the reordering of the loops back into their original ordering after the

```
DO I = 1,100
  DO J = 1,100
    A(I,J) = A(I+1,2) + A(I,J-1) * B(J)
  ENDDO
ENDDO
```

Figure 5.18 Code fragment illustrating necessity of loop reordering.

```
DO J = 1,100
  CS1_A(J) = CS1_A(2) + A(101,2) - A(1,2) + CS1_A(J-1) * B(J)
  DO I = 1,100
    A(I,J) = A(I+1,2) + A(I,J-1) * B(J)
  ENDDO
ENDDO
```

Figure 5.19 Checksum introduction for the code in Fig. 5.18 after reordering (correct).

checksum statements have been introduced. However, in this case, we may physically reorder the loops to obtain an ordering in which loops enclosed within the loop whose index is the candidate checksum index do not carry flow dependences, and then introduce the checksum statements. The point is illustrated in Figs. 5.18, 5.19, and 5.20. The code in Fig. 5.18 has a flow dependence carried by the `J` loop. However, the loops may be validly reordered to move the `J` loop outwards and `I` may be chosen as the checksum index to obtain the code in Fig. 5.19. An attempt to obtain the original loop ordering after checksums have been introduced results in the code in Fig. 5.20. However, this clearly does not yield the same results as the code in Fig. 5.19 since the value of `A(1,2)` used by the checksum statement in the first case is the value assigned in the first iteration of the previous `I` loop, while the value used in the second case is the old value of `A(1,2)` before executing any iterations of the `I` loop.

Now we state and prove some theorems indicating when a candidate checksum index is a valid checksum index for a block of assignment statements enclosed within a perfect loop nest. However, before we can do this, we need to point out problems that can be caused by backward dependences. This problem is illustrated by the code fragment shown in Fig. 5.21. Here, there is a backward dependence (actually, an antidependence) from the second assignment statement to the first. Also, the loop variable `i` is a candidate checksum index for both assignment statements. Introducing checksum manipulations after choosing `i` as the checksum index, we obtain the code in Fig. 5.22.

```
DO J = 1,100
  CS1_A(J) = CS1_A(2) + A(101,2) - A(1,2) + CS1_A(J-1) * B(J)
ENDDO
DO I = 1,100
  DO J = 1,100
    A(I,J) = A(I+1,2) + A(I,J-1) * B(J)
  ENDDO
ENDDO
```

Figure 5.20 Checksum introduction for the code in Fig. 5.18 without reordering (incorrect).

```
DO i = 2,100
  a(i) = a(i) + c(i-1)
  b(i) = b(i) + a(i+1)
END DO
```

Figure 5.21 Code fragment illustrating backward dependence.

However, the checksum manipulations do not yield the desired checksum values. This is because `b(i)` uses values of `a` computed prior to the loop, while `$cs1_b` uses the newly transformed value of `$cs1_a`. Thus a spurious flow dependence exists between the two checksum statements, while no such flow dependence exists between the two assignment statements in the original code fragment.

First, we state an analog of Theorem 3 for the case when a block of statements is enclosed within a perfect loop nest. After proving the theorem, we indicate when condition (3) is violated and what to do in this case.

Theorem 6 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement $S_1, S_2, \ldots, S_n$, with S_i lexically preceding S_j if $i < j$. Suppose I_1, the loop variable*

```
$cs1_a = $cs1_a + $cs1_c + c(1) - c(100)
$cs1_b = $cs1_b + $cs1_a + a(101) - a(2)
DO i = 2,100
  a(i) = a(i) + c(i-1)
  b(i) = b(i) + a(i+1)
END DO
```

Figure 5.22 Incorrect checksum code for code fragment in Fig. 5.21 illustrating problem caused by backward dependence.

```
DO L1
  DO L2
    S1
    S2
  ENDDO
ENDDO
```

Figure 5.23 Loop nest with multiple assignment statements in loop body.

for the outermost loop L_1, is a candidate checksum index for each S_i. Let CS_i be the checksum statement corresponding to S_i with I_1 chosen as the checksum index. Then I_1 is a valid checksum index for each S_i if all of the following conditions are satisfied:

(1) No S_i is involved in a dependence cycle.

(2) There is no dependence from S_i to S_j if $i > j$.

(3) CS_i does not access any array elements assigned by S_j for any $1 \leq i, j \leq n$.

Proof: The proof is sufficiently illustrated by the case when $n = 2$; i.e., there are two statements enclosed within the loop nest. This case is illustrated in Fig. 5.23. Since S_1 and S_2 are not involved in a dependence cycle by condition (1), and all dependences are from S_1 to S_2 by condition (2), loop distribution may be applied to yield the loop nests in Fig. 5.24. Theorem 3 then applies to each loop nest in Fig. 5.24. Thus I_1 is a valid checksum index for each loop nest in Fig. 5.24. Introduction of the checksum statements for the loop nests in Fig. 5.24 yields the code in Fig. 5.25. By condition (3), no dependences exist between the CS_i's and S_j's in Fig. 5.25. Applying loop fusion to Fig. 5.25 yields the code in Fig. 5.26. Since this code is precisely what would be generated upon choosing I_1 as the checksum index, we conclude that I_1 is a valid checksum index. □

Condition (3) may be violated if some of the checksum statements need to add or subtract off array variables in order to adjust the checksum value, and these variables are assigned to by some of the original statements in the loop. In this case, however, loop distribution could be used to separate out the statements in the body of the loop nest into separate loop nests, and Theorem 3 could be applied to generate checksum statements for each of the loop nests. This would result in code resembling that in Fig. 5.25, with checksum statements alternating with original statements.

Theorem 7 *Consider a block of statements $S_1, S_2, \ldots, S_n$ enclosed within a nest of loops $L_1, L_2, \ldots, L_m$. Suppose the loop index of the cth loop, I_c, is a candidate checksum index for each statement S_i, $1 \leq i \leq n$. Let CS_i be the checksum statement corresponding to S_i with I_c chosen as the checksum index. Then I_c is a valid checksum index if the following three conditions hold:*

```
DO L1
  DO L2
    S1
  ENDDO
ENDDO

DO L1
  DO L2
    S2
  ENDDO
ENDDO
```

Figure 5.24 Loop distribution applied to loop nest of Fig. 5.23.

```
DO L2
  CS1
ENDDO

DO L1
  DO L2
    S1
  ENDDO
ENDDO

DO L2
  CS2
ENDDO

DO L1
  DO L2
    S2
  ENDDO
ENDDO
```

Figure 5.25 Introduction of checksum statements for loop nests of Fig. 5.24.

```
DO L2
 CS1
 CS2
ENDDO

DO L1
 DO L2
  S1
  S2
 ENDDO
ENDDO
```

Figure 5.26 Loop fusion applied to loop nests of Fig. 5.25

(1) No S_i is involved in a dependence cycle involving dependences carried solely by loops L_j, $c \leq j \leq m$.

(2) There is no dependence from S_i to S_j that is loop independent or carried by loops L_k, $c \leq k \leq m$, if $i > j$.

(3) CS_i does not access any array elements assigned by S_j for any $1 \leq i, j \leq n$.

Proof: The proof proceeds by first unrolling the first $c-1$ loops in the loop nest and then applying Theorem 6 to the resulting loop nests. Condition (3) is then used to separate out the CS_i's to a separate loop nest, and loop fusion is applied to recover the original loop nest enclosing the S_i's. □

As before, if condition (3) is violated, loop distribution may be applied to the loops at level c and deeper, and checksum statements may be generated embedded within the outer $c-1$ levels of loops.

Often, even if the conditions stated in Theorems 4 and 7 are violated because of cycles of dependences carried by some loops inner to the loop whose index variable is the candidate checksum index (which we will refer to as the *candidate checksum loop*), it may be possible to use loop reordering to move the candidate checksum loop inside all loops carrying dependences. The candidate checksum index then becomes a valid checksum index for the reordered loops.

Similarly, the condition prohibiting backward dependences may be enforced by reordering some statements in the loop body. Thus the statements in the `i` loop in Fig. 5.21 may be validly reordered since there is a loop carried antidependence from the second statement to the first, but no dependence from the first statement to the second. Generating the checksum statements for the reordered code yields the correct results since the checksum statements are also generated in a reordered fashion.

Once the set of valid checksum indices has been determined for each statement, one of them is chosen as the index to compute the checksum over. For example, in the code in Fig. 5.3, the chosen checksum index is `j`, which corresponds to the second dimension of the arrays `$a` and `$b`. Once a valid checksum index has been chosen as the index to sum over for a statement S, the corresponding checksum statement is generated in the following manner. The intermediate nodes for the syntax tree for S are annotated with the $AFFINE$ and $NOTAFFINE$ sets, which were computed while computing the candidate checksum indices for S while traversing the tree in bottom-up fashion. Now, the syntax tree is traversed in top-down fashion in order to determine subexpressions that do not involve the checksum index `j`. This is indicated by the fact that the $AFFINE$ set associated with the node in the syntax tree that is the root of the subexpression does not contain `j`. The entire subexpression is then multiplied by the number of times the `j` loop is executed. The algorithm for expanding constants in expressions is shown in Fig. 5.27. Only the rules for the commonly occurring operators are shown for brevity. As an example, on using EXPAND_CONSTANTS on the tree of Fig. 5.13, the only node encountered with `j` absent from its $AFFINE$ set is the node for 10. Thus this results in the expression 10 * 998 in the corresponding checksum statement, which is shown in Fig. 5.12.

After constants have been expanded in affine expressions, arrays used by the expression that involve the checksum index as a subscript need to be replaced by checksums. These arrays may be found by a top-down traversal of the syntax tree and replaced by a checksum variable with the same subscript expressions in all dimensions except the one involving the checksum index, which vanishes. However, a correction needs to be made to the checksum variable in the event that the subscript involving the checksum index, say `j`, is of the form `j+c` or `j-c`, where `c` is a positive constant. This point is illustrated by the code in Fig. 5.11 and the corresponding check code in Fig. 5.12. We assume that upon entering the `j` loop, the checksum of `b(i,j)`, for `j` ranging from 2 to 999 (the values taken by the `j` loop), is available. However, the checksum over `b(i,j+1)` is required. This checksum may be derived from the checksum over `b(i,j)` by subtracting and adding one element, as illustrated by the code in Fig. 5.12. The other accesses to `b` in the right-hand side expression are similarly replaced by checksums incorporating the addition and subtraction of some extra elements. The information propagation pass is responsible for making available the checksums over `b(i,2:999)` at the entry to the `j` loop.

```
EXPAND_CONSTANTS(expr, ci, numiter)
  /* expr is an expression
   ci is a checksum index
   numiter is the no. of iterations */
 if !IN_SET(ci, AFFINE_SET(expr))
   then return BUILD_BINARY_OP(TIMES, numiter, expr) /* returns a new expr which is
            the old expr times numiter */
 switch
   case BIPLUS :
   case BIMINUS :
        return BUILD_BINARY_OP(TYPE(expr),
        EXPAND_CONSTANTS(LEFT_OP(expr), ci, numiter),
        EXPAND_CONSTANTS(RIGHT_OP(expr), ci, numiter))
   case BITIMES :
        if IN_SET(ci, AFFINE_SET(LEFT_OP(expr)))
          then return BUILD_BINARY_OP(BITIMES,
                 EXPAND_CONSTANTS(LEFT_OP(expr), ci, numiter), RIGHT_OP(expr))
          else  return BUILD_BINARY_OP(BITIMES,
                 LEFT_OP(expr), EXPAND_CONSTANTS(RIGHT_OP(expr), ci, numiter))
   case BIDIVIDE :
        return BUILD_BINARY_OP(BIDIVIDE,
        EXPAND_CONSTANTS(LEFT_OP(expr), ci, numiter), RIGHT_OP(expr))
   case ARRAY_REF :
   case VARIABLE :
        return COPY_EXPR(expr)
```

Figure 5.27 Algorithm for expanding constants in affine expressions.

```
GENERIC_DATAFLOW()
  change ← TRUE
  while change = TRUE
        do change ← FALSE
           for i ← 0 to nfgblks
           do in[i] ← ⋂_{j∈FGPRED(i)} out[j]
              oldout ← out[i]
              out[i] ← gen[i] ⋃ (in[i] − kill[i])
              if SETS_NOT_EQUAL(out[i], oldout)
                then change ← TRUE
```

Figure 5.28 Outline of generic iterative dataflow algorithm.

5.3.3 Information propagation and check generation

After checksum manipulation statements have been introduced, the information propagation pass is run. The pass may be divided into two stages. In the first stage, an iterative dataflow algorithm is executed to determine the checksum and array values available at various points in the program. In the second stage, the information about available checksums and arrays is used to regenerate checksums and arrays as required. We now explain each of these stages in detail.

The outline of the basic iterative dataflow algorithm is shown in Fig. 5.28. For a more detailed description of the iterative dataflow approach, see [46, 49].

Now we discuss the specifics of the algorithm as applied to our problem, viz., computing the ranges of checksums and arrays available at each block in the flowgraph.

The flowgraph for the code in Fig. 5.3 is shown in Fig. 5.29. Note that the fact that the loops are nonzero-trip has been used in constructing the flowgraph. Also, a dummy start node has been inserted.

We introduce two sets, called *AVAILARRAY* and *AVAILCS*, with every node in the flowgraph. *AVAILARRAY* and *AVAILCS* store the ranges of the shadow arrays and checksums that are available at the end of the block of statements comprising the flowgraph node. In the case of *AVAILARRAY*, when we say that a shadow array in the set is available, we mean that the values stored in that array match the corresponding values in the original array that the shadow array is supposed to check. Similarly in the case of *AVAILCS*, when we say that a checksum variable or array is available, we mean that if a new checksum is computed over the original array that this checksum is supposed to check, then the values of the two checksums should match. Associated with every node that is a loop header are two more sets, called *AVAILARRAY_ON_DO_EXIT* and

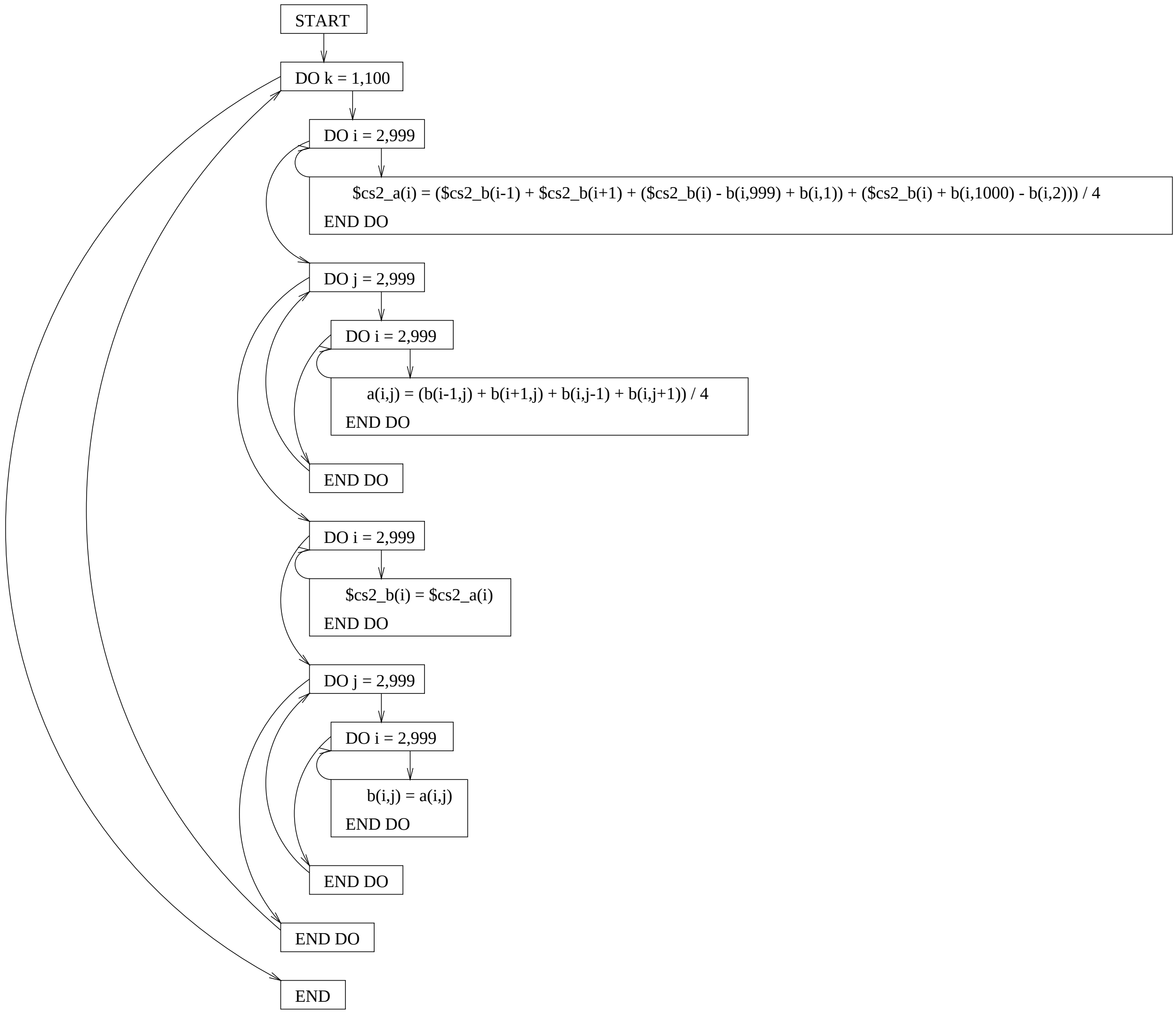


Figure 5.29 Control flow graph for Jacobi code with checksums.

AVAILCS_ON_DO_EXIT, which store the arrays and checksums available when the loop terminates. The latter two sets are introduced to ensure a less conservative computation of available arrays and checksums than would otherwise occur.

To conveniently propagate range information, we ensure that there is only one entry per variable in each of the above sets, and the ranges covered by any entry cover a contiguous portion of the array. We ensure this by making conservative choices, if necessary, on updating the sets.

Prior to executing the iterative algorithm of Fig. 5.28, the above mentioned sets are initialized for every node in the flowgraph. Essentially, the statements in a basic block are traversed in lexical order and the sets updated for each statement as it is encountered. Initially, at the start of each basic block, the sets are initialized to the empty set. We make an exception in the case of the start node in the flowgraph. For this node, the *AVAILARRAY* set is initialized to include all of the shadow arrays that have been introduced for the program. The *AVAILCS* set is initialized to include all of the checksum variables that occur in the program, with the ranges computed from the first occurrence of the variable as the program is traversed in lexical order.

For each iteration of the dataflow algorithm, all of the nodes in the flowgraph, except the dummy start node, are traversed one after the other. Upon entry to a basic block associated with a flowgraph node, the initial values of *AVAILARRAY* and *AVAILCS* are computed by taking an intersection of the *AVAILARRAY* and *AVAILCS* sets arriving along the incoming edges to the node. (The *AVAILARRAY* and *AVAILCS* values at the end of a node are propagated along all of the outgoing edges of the node.) However, an exception is made in the case in which the node is one that follows an exit from a loop. In this case, instead of the *AVAILARRAY* and *AVAILCS* sets associated with the loop header, the *AVAILARRAY_ON_DO_EXIT* and the *AVAILCS_ON_DO_EXIT* sets are used in computing the intersection. Similarly, in the event that the node under consideration is a loop header itself, and the loop is not zero-trip, the *AVAILARRAY_ON_DO_EXIT* and the *AVAILCS_ON_DO_EXIT* are set to the *AVAILARRAY* and *AVAILCS* sets, respectively, which are available along the backedge. In the event that we cannot determine if the loop is always non-zero-trip, the *AVAILARRAY_ON_DO_EXIT* and the *AVAILCS_ON_DO_EXIT* sets are set to the *AVAILARRAY* and *AVAILCS* sets associated with the loop header, which are computed by taking intersections of all the *AVAILARRAY* and *AVAILCS* on the incoming edges (which also include the backedge).

For each statement encountered in a basic block, the sets are updated in the following manner. Only check statements result in the sets being updated, and different actions are taken for check statements that are checksum statements and statements which are duplicates of the original statement, operating on shadow arrays. First, we discuss the update equations for the sets when a checksum statement *cksumstmt* is encountered. Let the checksum variable assigned to by *cksumstmt* be `$cs1_a`. Let the original array variable corresponding to `$cs1_a` be `a`. First, entries for all checksum variables corresponding to `a`, except possibly an earlier entry for `$cs1_a`, are removed from the *AVAILCS* set. An earlier entry for `$cs1_a` will also be removed if the ranges covered by the checksum dimension in the set and in the statement are not identical. Next, the ranges for each dimension are computed from the bounds of the loops enclosing *cksumstmt* and the subscript expressions for `$cs1_a`. Recall that the subscript corresponding to the checksum index is removed from the checksum variable; this subscript expression is determined from the variable being assigned to by the original statement corresponding to this check statement. The set *AVAILCS* is updated to include `$cs1_a` if it does not already contain an entry for `$cs1_a`. If *AVAILCS* already contains an entry for `$cs1_a`, then the newly computed ranges are merged with the old range information. If the two ranges are disjoint, then the new entry replaces the old only if it covers a larger portion of the array. This is a conservative criterion enforced due to our requirement that there be a single entry for each variable in each set at any time. Also, if the set *AVAILARRAY* contains the shadow array variable, say `$a`, corresponding to `$cs1_a`, then the ranges covered by `$cs1_a` that were entered into *AVAILCS* are removed from the ranges covered by `$a` in *AVAILARRAY*. If this results in an empty range in some dimension or in fragmentation of the ranges, then the entry for `$a` is removed from *AVAILARRAY*.

If instead of a checksum statement, a duplicate statement is encountered, then two cases need to be distinguished. The first case occurs when all enclosing loop bounds are constants, and all subscript expressions occurring in the statement are of the form i, $i + c$ or $i - c$, where i is a variable and c is a constant (recall that these same restrictions must be satisfied by a checksum statement). In this case, the range covered by the left-hand side variable, say `$a`, is determined. The variable `$a` and the range covered by it are entered into *AVAILARRAY* or are merged with the range information already in *AVAILARRAY* if there is already an entry in *AVAILARRAY* for `$a`. If there is an entry for a checksum variable corresponding to `$a` (such as `$cs1_a` or `$cs2_a`, for example) in *AVAILCS*, then the ranges covered by `$a` that were added to the *AVAILARRAY* set are removed from the corresponding checksum variable entry in *AVAILCS*. If this leads to some dimension becoming empty or fragmented, then the checksum variable entry is removed from *AVAILCS*.

Update rules upon entering node n of the flowgraph

$$\begin{aligned} AVAILCS(n) &\leftarrow \cap_{j \in fgpred(n)} AVAILCS(j) \\ AVAILARRAY(n) &\leftarrow \cap_{j \in fgpred(n)} AVAILARRAY(j) \end{aligned}$$

Update rules for a checksum statement CS

$GENCS$	$\leftarrow$	all checksum elements accessed by CS
$AVAILCS$	$\leftarrow$	$AVAILCS \cup GENCS$
$KILLARRAY$	$\leftarrow$	all shadow array elements covered by checksum elements assigned to by CS
$AVAILARRAY$	$\leftarrow$	$AVAILARRAY - KILLARRAY$

Update rules for a duplicate check statement DC

$GENARRAY$	$\leftarrow$	all shadow array elements accessed by DC
$AVAILARRAY$	$\leftarrow$	$AVAILARRAY \cup GENARRAY$
$KILLCS$	$\leftarrow$	all checksum elements covering any portion of the shadow array elements assigned to by DC
$AVAILCS$	$\leftarrow$	$AVAILCS - KILLCS$

Figure 5.30 Rules for updating *AVAILARRAY* and *AVAILCS* sets.

The second case occurs when the duplicate statement's subscript expressions or the bounds of the loops enclosing it do not satisfy the conditions mentioned earlier. In this case, all arrays accessed by the statement are added to the *AVAILARRAY* set, with the ranges covering the entire array (Code copying the entire original array into the corresponding shadow array will be generated just prior to the execution of the statement by the second stage of the propagate pass). All checksum variables corresponding to the shadow arrays added to *AVAILARRAY* are removed from *AVAILCS*.

The rules for updating the *AVAILARRAY* and *AVAILCS* sets in each iteration of the flowgraph are shown in Fig. 5.30. Some of the details discussed in the previous paragraphs have been omitted from the figure.

Once the dataflow algorithm has converged, the second stage of the pass, which involves regeneration of checksums and shadow elements, is performed. For example, if a statement within a loop needs the checksum value over a certain array, but this is not available at that point (which can be determined by examining the *AVAILCS* set just before the statement), a loop needs to be inserted prior to the loop enclosing the statement in which the required checksum is recomputed by summing the appropriate array elements. As we explain later, we also attempt to check that the elements being summed to regenerate the checksum have not been corrupted by errors. If a statement within a loop requires shadow array values that are not available (which can be determined by examining the *AVAILARRAY* set just before the statement), these are regenerated by inserting a loop nest prior to the loop enclosing the statement, which copies the corresponding original array values into the required shadow array values. As we explain later, we also generate a check to determine if the elements being copied over are correct, if possible.

First, one more pass over the flowgraph is used to compute the *AVAILARRAY* and *AVAILCS* sets for each individual statement in the program, rather than just the final values at the end of each basic block. The list of statements comprising the program is then traversed in lexical order. Recall that checksum statements are enclosed in perfect loop nests whose bodies consist solely of checksum statements. When such a loop nest is encountered, the checksums that are used by each statement in the body are determined by traversing the syntax tree representing the right-hand side of each statement, collecting the checksum variables which appear, and computing the ranges covered from the subscript expressions and the enclosing loop bounds. The set of these checksums is denoted by *REQDCS*. The values that actually need to be regenerated at the start of the loop nest (which we denote by *GENCS*) are determined by subtracting the *AVAILCS* set at the entry to the loop nest enclosing the checksum statements from the *REQDCS* set. Once *GENCS* has been determined for the loop body, code for recomputing these checksums is inserted at the beginning of the loop body. Also, it is determined if the shadow array values for the array values that are summed to regenerate the checksums are available upon entry to the loop nest. If so, code for performing a comparison check of the array values being summed and the corresponding shadow array values is inserted prior to the loop nest. These rules are summarized in Fig. 5.31.

An example code fragment with values of the various sets is shown in Fig. 5.32, and the check code that would be generated for it is shown in Fig. 5.33.

Loop nests that enclose check statements (but not checksum statements) are handled in a different manner since these check the original statements by performing duplicate operations rather than checksum manipulations and thus affect the values of the *AVAILCS* and *AVAILARRAY* sets in a different manner. As before, the entire body of the loop nest is traversed. For each assignment statement encountered in the loop nest, the shadow array variables and ranges are computed from the expression tree for the statement, the subscript expressions and the loop bounds. If the ranges cannot be computed due to the loop bounds or subscript expressions being complicated, or because the statement is not enclosed in a perfect loop nest, it is assumed that the entire array is used by

For a loop nest enclosing checksum statements

$$\begin{aligned}
REQDCS &\leftarrow \text{all checksum elements used by all checksum} \\
&\quad\;\text{statements in loop body} \\
GENCS &\leftarrow REQDCS - AVAILCS \\
SUMVALUES &\leftarrow \text{array values covered by } GENCS \\
CHECKVALUES &\leftarrow SUMVALUES \cap AVAILARRAY
\end{aligned}$$

Regenerate checksum elements in *GENCS* by summing the corresponding array values.

Compare shadow array values in *CHECKVALUES* with original array values.

For a loop nest enclosing duplicate check statements

$$\begin{aligned}
REQDARRAY &\leftarrow \text{all shadow array elements used by all check} \\
&\quad\;\text{statements in loop body} \\
GENARRAY &\leftarrow REQDARRAY - AVAILARRAY \\
SUMCS &\leftarrow \text{all checksums that can be generated from } GENARRAY \\
CHECKCS &\leftarrow SUMCS \cap AVAILCS
\end{aligned}$$

Regenerate shadow array elements in *GENARRAY* by copying the corresponding array values.

Generate the checksums in *CHECKCS* by summing the original array values they cover.

Compare the checksums in *CHECKCS* with the corresponding checksums in *AVAILCS*.

Figure 5.31 Rules for regenerating checksums and shadow arrays.

```
C AVAILARRAY = {$B(6:10,1:10)}, AVAILCS = {$CS2_B(1:5,1:10)}
C REQDCS = {$CS2_B(1:10,1:10)}
C GENCS = REQDCS - AVAILCS = {$CS2_B(6:10,1:10)}

      DO I = 1,10
        $CS2_A(I) = $CS2_B(I) + 10*10
      ENDDO

      DO I = 1,10
        DO J = 1,10
          A(I,J) = B(I,J) + 10
        ENDDO
      ENDDO
```

Figure 5.32 Code fragment for checksum regeneration showing *AVAILARRAY*, *AVAILCS*, *REQDCS*, and *GENCS* sets.

the statement. The array elements accessed by check statements within the loop nest are stored in a set called *REQDARRAY*. The array elements that are actually required by the statement (which we store in a set called the *GENARRAY*) are computed by subtracting the entries in the *AVAILARRAY* set for the statement from the *REQDARRAY* set for the loop nest. Code for copying over the values of the corresponding original array elements into the shadow array elements in *GENARRAY* is then generated prior to entering the loop nest. The *AVAILCS* set for the loop header is examined to determine if any checksum variables are available to check the array elements being copied over. If this is the case, then code is also inserted to sum the elements being copied over and perform a comparison check against the available checksum values. The code fragment in Fig. 5.9 shows the *AVAILARRAY*, *AVAILCS*, *REQDARRAY*, and *GENARRAY* sets associated with a loop nest enclosing a nonlinear check statement, and the check code corresponding to this code fragment is shown in Fig. 5.34.

As an example, the values of *AVAILARRAY* and *AVAILCS* after convergence are shown in Fig. 5.35 for selected edges of the control flow graph of Fig. 5.29.

5.4 Data Distribution Specification for Check Data for Distributed-Memory Parallel Programs

The algorithms described in the previous section result in the generation of a serial program with checks that is capable of detecting transient errors. However, permanent errors may still go

```
C AVAILARRAY = {$B(6:10,1:10)}, AVAILCS = {$CS2_B(1:5,1:10)}
C REQDCS = {$CS2_B(1:10,1:10)}
C GENCS = REQDCS - AVAILCS = {$CS2_B(6:10,1:10)}

C REGENERATE CHECKSUMS

       DO I = 6,10
         $CS2_B(I) = 0
         DO J = 1,10
           $CS2_B(I) = $CS2_B(I) + B(I,J)
         ENDDO
       ENDDO

C CHECK ELEMENTS WHICH WERE ADDED

       DO I = 6,10
         DO J = 1,10
           IF (COMPARE($B(I,J),B(I,J)) .EQ. 1)
             CALL ERROR_HANDLER
         ENDDO
       ENDDO

       DO I = 1,10
         $CS2_A(I) = $CS2_B(I) + 10*10
       ENDDO

       DO I = 1,10
         DO J = 1,10
           A(I,J) = B(I,J) + 10
         ENDDO
       ENDDO
```

Figure 5.33 Checksum regeneration for code fragment in Fig. 5.32.

```
C AVAILARRAY = {$B(1:5,1:10)}, AVAILCS = {$CS2_B(6:10,1:10)}
C REQDARRAY = {$B(1:10,1:10)}, GENARRAY = {$B(6:10,1:10)}

C COPY ARRAY ELEMENTS INTO SHADOW ARRAYS

       DO I = 6,10
         DO J = 1,10
           $B(I,J) = B(I,J)
         ENDDO
       ENDDO

C CHECK ELEMENTS WHICH WERE COPIED

       DO I = 6,10
         T = 0
         DO J = 1,10
           T = T + B(I,J)
         ENDDO
         IF (COMPARE($CS2_B(I),T) .EQ. 1)
           CALL ERROR_HANDLER()
       ENDDO

       DO I = 1,10
         DO J = 1,10
           $A(I,J) = $B(I,J)*$B(I,J)
         ENDDO
       ENDDO

       DO I = 1,10
         DO J = 1,10
           A(I,J) = B(I,J)*B(I,J)
         ENDDO
       ENDDO
```

Figure 5.34 Shadow array regeneration for code fragment in Fig. 5.9.

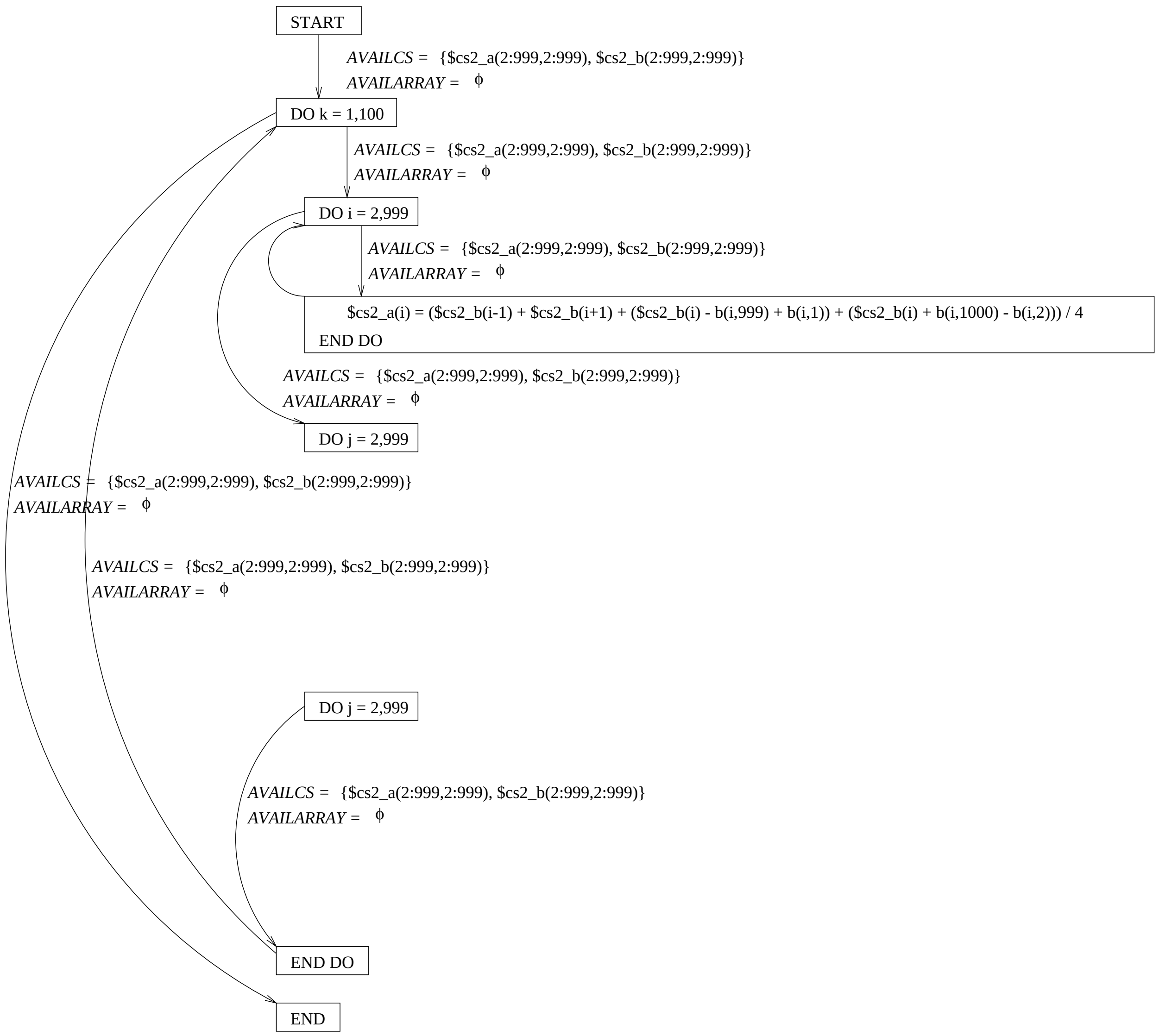


Figure 5.35 Final values of *AVAILARRAY* and *AVAILCS* on selected edges of the flowgraph of Fig. 5.29.

undetected if such errors affect the check phase in such a manner as to cause it to pass. In a parallel processing environment, one approach to guaranteeing high error coverage would be to have each processor check not its own, but a neighboring processor's data. One approach that has been used to aid compilers in the generation of parallel programs has been to specify the distribution of the arrays accessed by the program over the set of processors using data distribution directives and have the compiler automatically insert the message calls needed to transfer non-local data to a processor whenever such data is accessed. The basis for communication generation is the owner-computes rule, where a processor that owns the element being assigned to in an assignment statement is responsible for executing that statement, while if any of the elements being accessed by the statement are not local to the processor performing the assignment, these are communicated via messages. We leverage off the compiler efforts in this direction by deriving the data distributions for the extra arrays and checksums introduced by our compiler from the data distributions in the original program in such a manner that for each original data element, the data element checking it resides on a different processor. The parallel code generated by the compiler then has all of the message communication required for updating the check data and performing the comparison checks in place.

In the rest of this section, we discuss some features of HPF used by our compiler to specify data distributions and discuss how our compiler specifies data distributions to achieve our goal of having each processor's data checked by a different processor.

5.4.1 High Performance Fortran

High Performance Fortran (HPF) [10] is an extension of Fortran 90 [50] that allows portable parallel programs to be written. HPF directives begin with `!HPF$` and are treated as comments by any compiler that does not have the capacity to recognize and use HPF information. The HPF directives which are important from our perspective are the `DISTRIBUTE`, `ALIGN`, and `PROCESSORS` directives.

The `DISTRIBUTE` directive allows the programmer to specify data distributions for objects that may either be arrays or templates. Templates are placeholders that occupy no memory; different arrays may be mapped to the same template. The data distribution of the template is then used for all arrays mapped to it. Two general kinds of distributions are *block* and *cyclic*. Block distributions specify that each processor gets a contiguous piece of the object, while cyclic distributions specify that each processor gets a piece of the object at regular intervals throughout the object. Some example distributions and the syntax used for specifying them are shown in Fig. 5.36. The symbol * in some of the data distribution specification indicates that the corresponding array dimension resides on a single processor. In this case, we say that the array dimension has been *sequentialized.*

The `ALIGN` directive is used to align one array with another. An element that is aligned with another element resides on the same processor. Some examples of array alignment and the syntax used to achieve them are specified in Fig. 5.37.

The `PROCESSORS` directive is used to specify an abstract topology which can be a multidimensional mesh. The processor arrangement can then be used in distribute directives to specify data distributions across processor configurations.

5.4.2 Data distribution specifications for check data

In most of the following examples, we will concentrate on block distributed arrays; the ideas behind handling arrays that are distributed in a cyclic or block-cyclic fashion are similar.

Data distribution specification for check data (checksums and shadow arrays) needs to be specified so that the original data and the data checking it reside on different processors. This, together with the owner-computes rule, ensures that each data element is subjected to a check on a different processor, thus increasing the likelihood of detecting single processor faults. Essentially, in the case of shadow arrays, a distribution is chosen that is almost identical to the distribution of the corresponding original array, except that the data elements in one of the distributed dimensions are shifted cyclically so that a data element and the corresponding shadow element reside on different processors. This is indicated for a block distributed array in Figs. 5.38 and 5.39. Note that the `WRAP` directive is used to specify that the elements that "fall off the end" of the template are to wrap around to the first processor. `WRAP` is not a standard HPF directive; however, the same effect can be achieved in a somewhat roundabout manner by using only standard HPF. Instead, we use `WRAP` for brevity.

To determine how a checksum variable is to be distributed, we first determine how the shadow array variable corresponding to the original array being checked by the checksum would be distributed. Two cases are distinguished. The first corresponds to the case when the dimension being summed over is sequentialized. In this case, the other dimensions of the checksum are distributed in a manner identical to the distribution of the shadow array. This case is illustrated in Figs. 5.40 and 5.41.

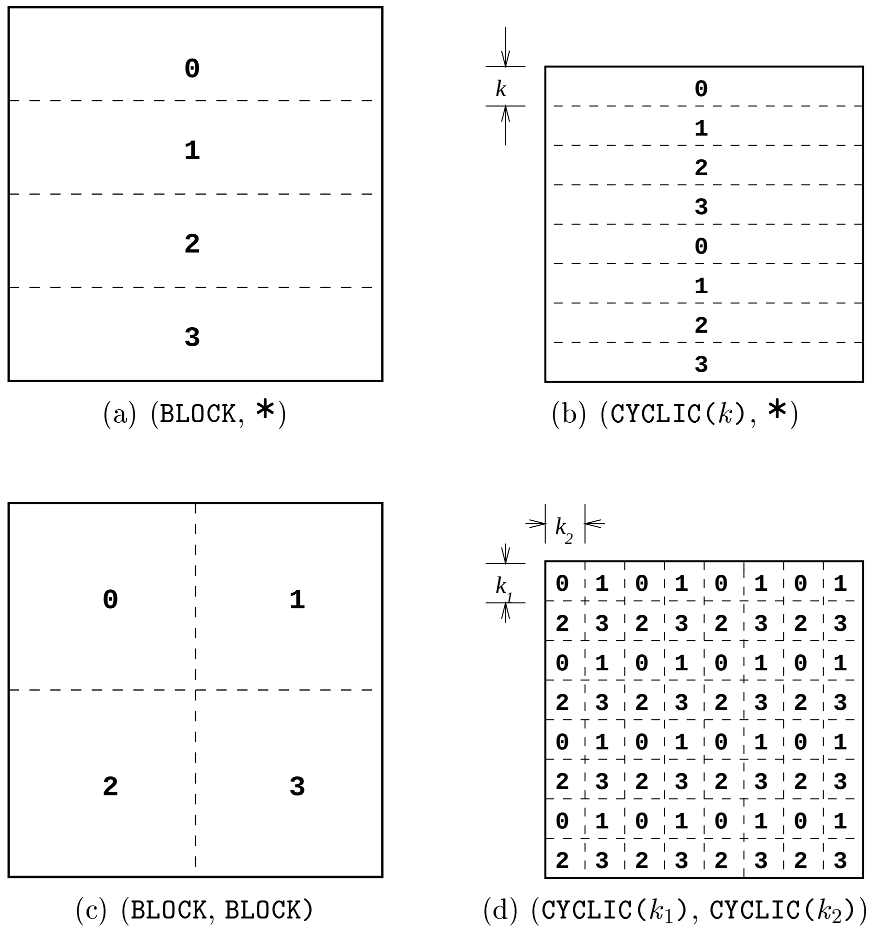
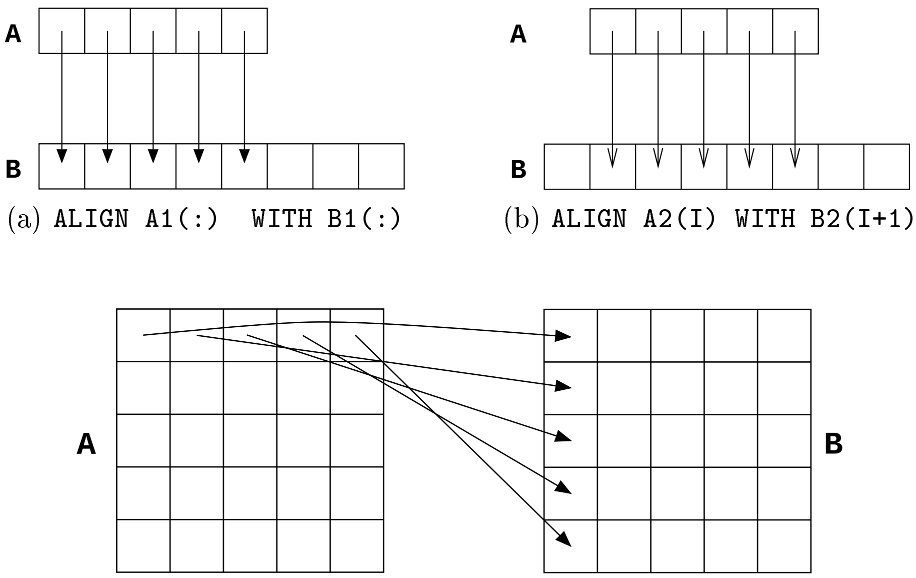


Figure 5.36 Examples of data distributions for a two-dimensional array onto a four-processor arrangement.



(a) ALIGN A1(:) WITH B1(:)   (b) ALIGN A2(I) WITH B2(I+1)

(c) ALIGN A3(I, J) WITH B3(J, I)

Figure 5.37 Examples of alignments.

```
      DOUBLE PRECISION b(64,64)
      DOUBLE PRECISION $b(64,64)

!HPF$ PROCESSORS :: p(4)
!HPF$ DISTRIBUTE (*, BLOCK) ONTO p :: b
!HPF$ TEMPLATE, DISTRIBUTE (*, BLOCK) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,:) WITH template$0(:,:) :: b
!HPF$ ALIGN (:,hpf$0) WITH template$0(:,hpf$0 + 16) WRAP :: $b
```

Figure 5.38 Data distribution specification for a block distributed array.

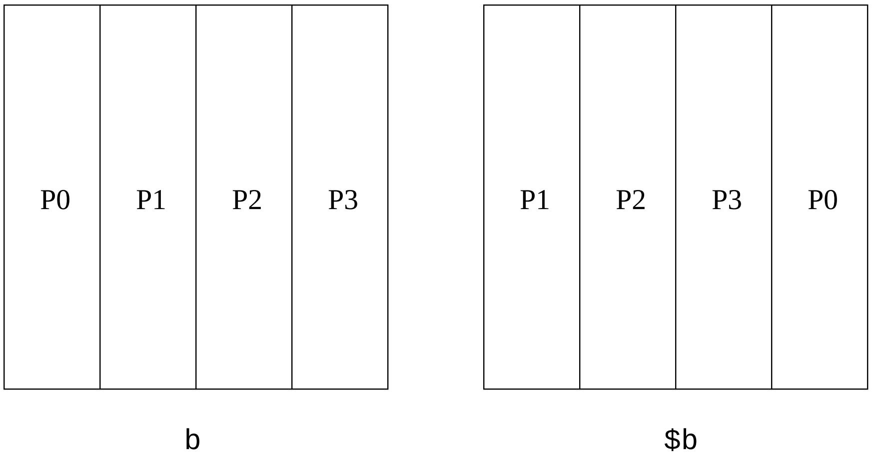


Figure 5.39 Illustration of data distribution for declaration in Fig. 5.38.

```
      DOUBLE PRECISION c(64,64)
      DOUBLE PRECISION $c(64,64)
C $cs2_c is obtained by summing the second dimension of c
      DOUBLE PRECISION $cs2_c(64)

!HPF$ PROCESSORS :: p(4)
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,:) WITH template$0(:,:) :: c
!HPF$ ALIGN (hpf$0,:) WITH template$0(hpf$0 + 16,:) WRAP :: $c
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO p :: TEMPLATE$1(64)
!HPF$ ALIGN $cs2_c(hpf$0) WITH TEMPLATE$1(hpf$0+16) WRAP
```

Figure 5.40 Checksum data distribution when the dimension being summed over is sequentialized.

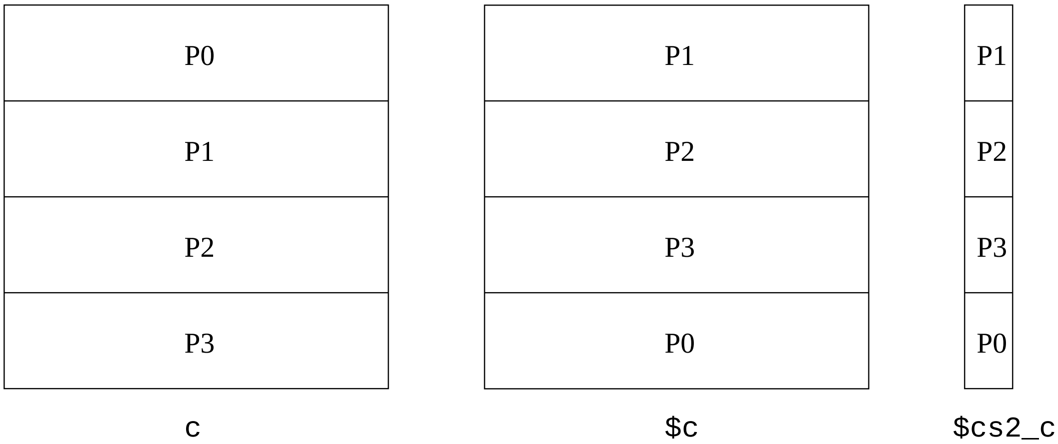


Figure 5.41 Illustration of data distribution for declaration in Fig. 5.40.

```
      DOUBLE PRECISION c(64,64)
      DOUBLE PRECISION $c(64,64)
C $cs1_c is obtained by summing the first dimension of c
      DOUBLE PRECISION $cs1_c(4,64)

!HPF$ PROCESSORS :: p(4)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO p :: c
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,:) WITH template$0(:,:) :: c
!HPF$ ALIGN (hpf$0,:) WITH template$0(hpf$0 + 16,:) WRAP :: $c
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ONTO p :: TEMPLATE$1(4,64)
!HPF$ ALIGN $cs1_c(hpf$0,:) WITH TEMPLATE$1(hpf$0+1,:) WRAP
```

Figure 5.42 Checksum data distribution when the dimension being summed over is distributed.

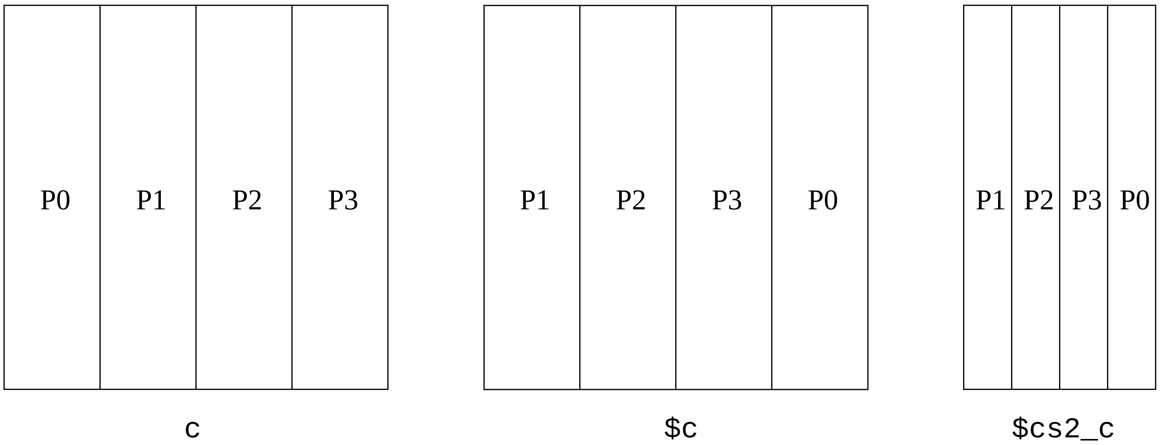


Figure 5.43 Illustration of data distribution for declaration in Fig. 5.42.

The second case occurs when the dimension that was summed over was not sequentialized but distributed. In this case, the checksum is expanded to include an extra dimension corresponding to the dimension being summed over, with the number of elements in the expanded dimension being equal to the number of processors in that dimension. The expanded dimension is then distributed so that each processor gets one element in each dimension, with the `WRAP` directive being specified if the corresponding shadow array would have been wrapped around for this dimension. This case is illustrated in Figs. 5.42 and 5.43.

In the case of a block distribution, each checksum element now stores the sum over a contiguous block of elements in the dimension being summed over, instead of storing the sum over the entire dimension. This may require the replacement of checksum manipulation statements in the code by statements manipulating the new checksums. Usually, this replacement takes the form of a prologue, a body, and an epilogue. This transformation may be illustrated by comparing the checksum manipulation statements in Figs. 5.4 and 5.5.

5.5 Summary

In this chapter, we have discussed the algorithms used by a compiler pass that generates error-detecting code based on checksum-based checks. Identification of loops that perform affine transformations on arrays involves examining the syntactic structure of the statements within the loop and the data dependences within the loop. Checksum-based checks are then generated for such loops while the remainder of the code is checked by duplicating the computations on separate copies of the original arrays. An information propagation pass based on a dataflow analysis framework is used to propagate information about available checksums and shadow arrays throughout the program so that checksums and shadow arrays are regenerated only as required. In the interest of high error coverage, the elements used to regenerate checksums are checked against their shadow values, if available, and conversely, the elements used to regenerate shadow array values are checked against available checksum values.

In a parallel processing environment, it is desirable to perform the checking of one processor's data on a different processor; we achieve this by deriving the data distributions of the check data from the data it checks in such a manner that an original data element and the element that checks it reside on different processors. A parallelizing compiler for distributed-memory machines is then used to automatically generate a parallel program in which the checks on the data owned by one processor are performed on a different processor, with the communication required to achieve this being automatically inserted by the compiler.

In the next chapter, we present results on applying our compiler to three programs and demonstrate the overheads of the error-detecting parallel program over the non-error-detecting version.

CHAPTER 6

RESULTS FOR COMPILER-GENERATED ERROR-DETECTING PARALLEL PROGRAMS

To illustrate the working of our compiler passes, we ran several Fortran programs annotated with HPF data distribution directives through it. We present the overhead results of the parallel error-detecting versions over the parallel versions with no error-detection for three of the applications here. The three applications are a parallel matrix multiplication routine computing $AB = C$ where A and C are distributed blockwise by rows and B is distributed blockwise by columns on a linear array, a Jacobi iterative solver with the grid points distributed blockwise in both dimensions on a 2-D mesh, and an ADI (Alternating Direction Implicit) solver with the grid points distributed blockwise by rows on a linear array. The first two programs consist of only linear statements, while the last consists of a mixture of linear and nonlinear loops. Two versions of each of these programs are listed in the Appendix; the first is the input program to our compiler, and the second is the output generated by our compiler.

The speedups on 16 processors for the three programs considered are shown in Figs. 6.1, 6.2, and 6.3.

Apart from presenting results on the overhead incurred because of our error detection mechanism, we also wanted to illustrate that the reuse of checksums across loops facilitated by our information propagation pass would result in lower overheads than the approach proposed in [5], which would cause checksums required by each loop to be regenerated prior to the loop. To illustrate this point we also implemented a version of the parallel, error-detecting Jacobi solver, which regenerated checksums prior to each loop instead of reusing available checksum values.

The testbed on which our experiments were performed was a 16-processor Intel Paragon.

6.1 Time Overhead

The overhead results are shown in Figs. 6.4, 6.5, and 6.6. The overheads diminish with an increase in problem size for the first two applications. However, the overhead for the ADI application is more than double that of the original program, because, for the first two applications, all of the statements within the main loops are linear and are therefore checkable by only checksum

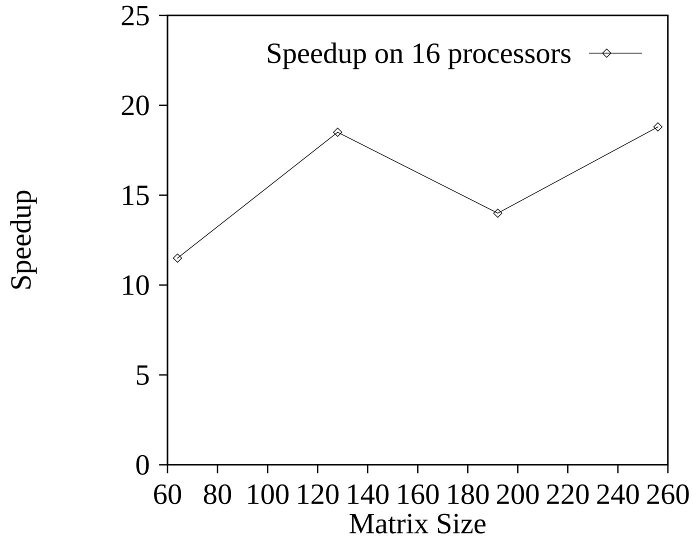


Figure 6.1 Speedup of matrix multiplication on 16 processors.

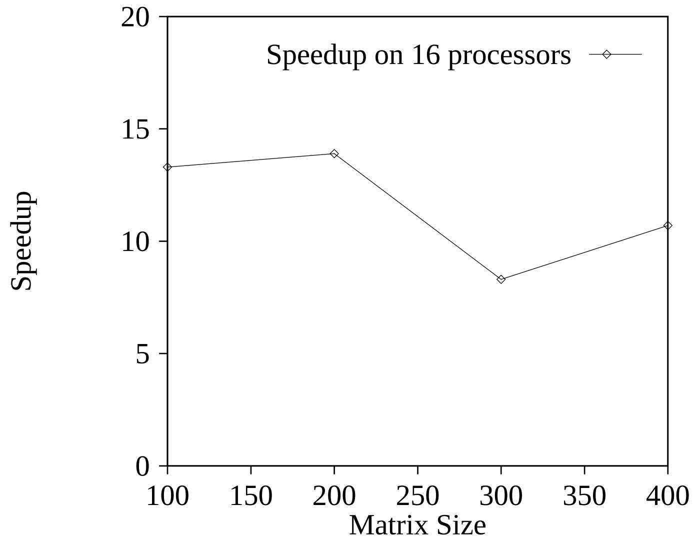


Figure 6.2 Speedup of Jacobi solver on 16 processors.

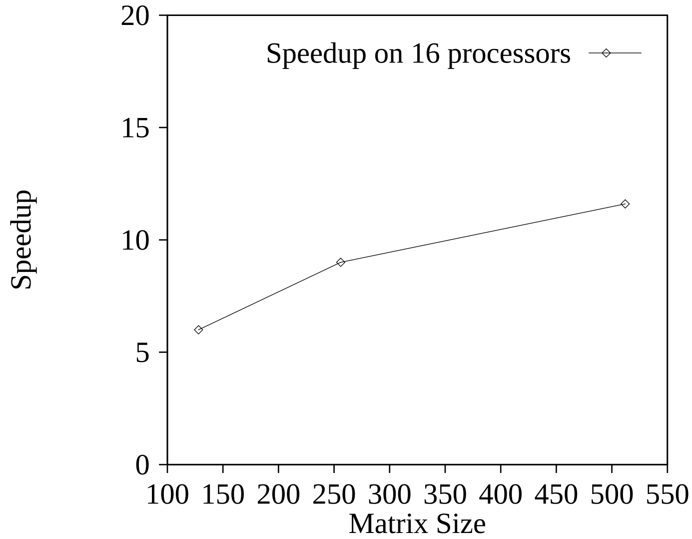


Figure 6.3 Speedup of ADI solver on 16 processors.

manipulations. However, the main loop of ADI integration contains interspersed linear and non-linear statements. This results in switching between checksum manipulations and duplicating the computations on shadow arrays, which also requires checksum or shadow array regeneration and checking of the data used in the regeneration, as described in Chapter 5. Since the shadow arrays and checksums are maintained on different processors than the data they check, the regeneration step also results in communication of large amounts of data.

6.2 Error Coverage

In addition to the time overheads, additional experiments were run in order to evaluate the effectiveness of the error-detecting parallel programs generated by our compiler in actually detecting errors. Following each statement in the parallel program that performed floating-point operations, an error injection routine was called, which replaced the result of the statement by a random word with a probability of 0.01. A hundred runs were performed for each of the three applications described earlier; in each case, the errors were detected. Thus for the set of experiments we performed, error coverage was 100%. The results are summarized in Table 6.1.

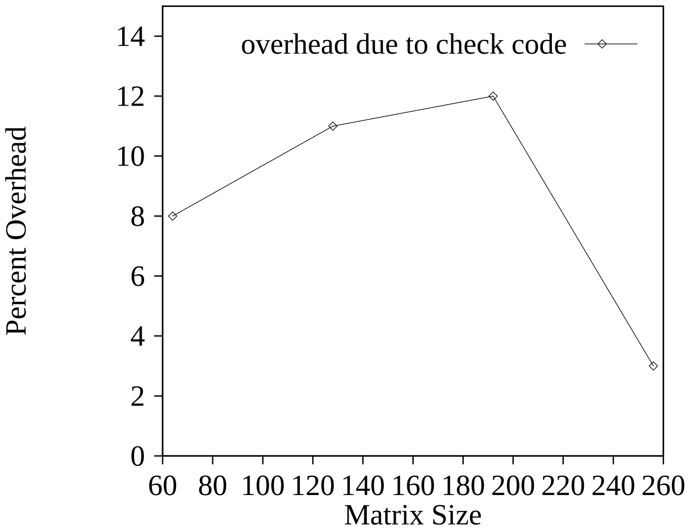


Figure 6.4 Overhead of check code for matrix multiplication.

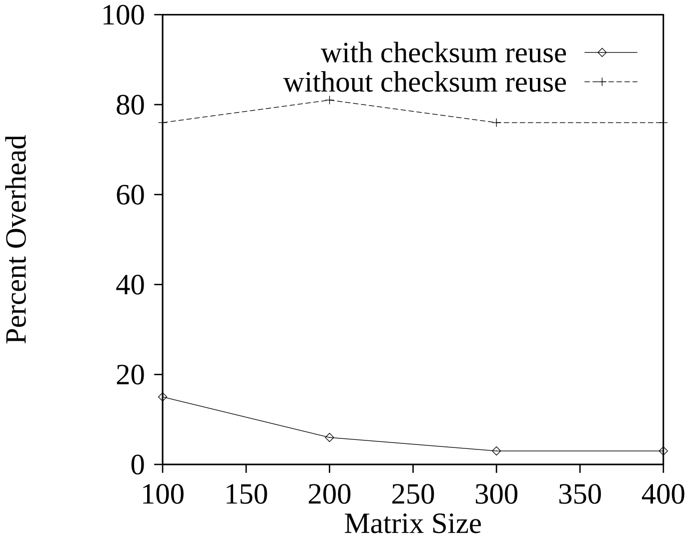


Figure 6.5 Overhead of check code for Jacobi solver.

Table 6.1 Error detection coverage for transient word-level errors.

	Mmul	Jacobi	ADI
Percent Coverage	100	100	100

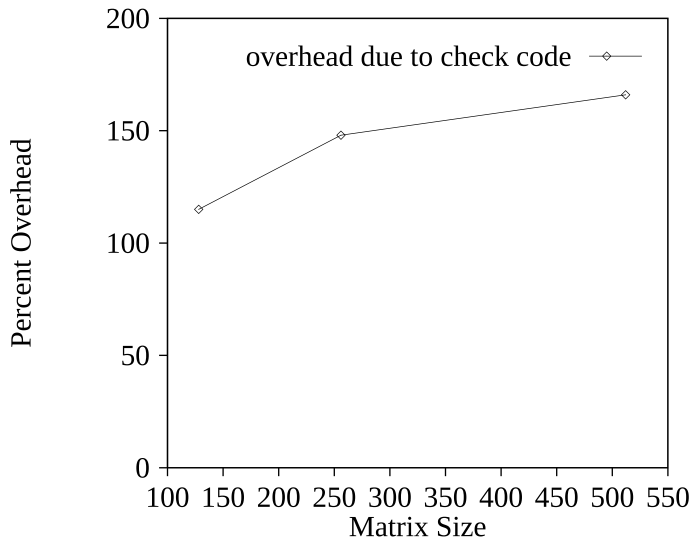


Figure 6.6 Overhead of check code for ADI integration.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation we have contributed to both the theory and implementation of error-detecting programs using the algorithm-based fault tolerance methodology.

We have, for the first time, proposed a general methodology for locating and recovering from t-faults in a parallel processing environment using the algorithm-based approach. In contrast with earlier researchers in the area, we have demonstrated the practicality of our approach by designing fault-locating and correcting versions of three numerical algorithms and implementing them on a distributed-memory parallel computer.

One direction for future work would be to examine the sharpness of the bounds for the error-correction capabilities of our method. Our error recovery algorithm guarantees recovery from t or fewer failures if check processors are not affected, but if check processors are affected, our recovery algorithm guarantees recovery only if a total of $2(\sqrt{t+1}-1)$ or fewer processors have failed. Since our recovery methodology uses coding-theoretic techniques, we feel intuitively that we should be able to recover from $\lfloor \frac{t-1}{2} \rfloor$ faults because a maximum-distance code [35] (such as the Reed-Solomon code, which we use for our recovery mechanism) with t check bits can recover from $\lfloor \frac{t-1}{2} \rfloor$ errors. It would be interesting to study if our present bound on recovery could be improved to this value and to give a simple recovery algorithm which was capable of recovering from this number of faults. (Recall that our general recovery algorithm can recover from any recoverable fault pattern, but is rather complex. The simplifications which apply to the case when at most $2(\sqrt{t+1}-1)$ faults have occurred do not apply to the general case).

The second contribution of our dissertation has been to demonstrate that the checksumming approach to checking linear and affine transformations, which has been heavily used in the design of ABFT programs, can be successfully incorporated into a compiler. Although this idea was first investigated in [5], it was not carried to full implementation. In our dissertation, we demonstrate that apart from the syntactic structure, one also must examine the dependencies in which a statement is involved to determine whether it performs an affine transformation. This aspect was ignored in [5]. We have established sufficient conditions for when a candidate checksum statement (one which possesses the necessary syntactic structure) is actually a valid checksum statement (one that can actually be checked using checksum manipulations). Ignoring the dependence conditions may lead to incorrect code in some cases. Another important improvement is the introduction of a dataflow analysis phase that inserts checks and recomputes checksums only where necessary, as opposed to the earlier approach, which would recompute checksums prior to and generate checks after each loop nest.

At present, our compiler is only able to generate checksum-based checks for loops implementing linear or affine transformations that additionally satisfy the restrictions of constant loop bounds and array subscripts of a very simple form. These restrictions are not dictated by the rules for determining affine transformations but by the fact that the information propagation pass performs set operations that can be done in a particularly simple manner if the present restrictions on the loop bounds and array subscripts are imposed. At present, the set operations involve sets that consist of ranges of integers specified by a constant upper bound and a constant lower bound. Removing these restrictions would require the information propagation pass to perform set operations on sets of integers with a more general form. For example, the upper and lower bounds of a set may be described by symbolic expressions, and the elements in the set may not include all of the integers between these bounds. Thus, improving the information propagation pass would represent a serious challenge and would probably require the use of extensive symbolic manipulation techniques of the sort implemented in Mathematica$^{\mathrm{TM}}$.

REFERENCES

[1] Y.-H. Choi and M. Malek, "A fault-tolerant FFT processor," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 617–621, May 1988.

[2] K-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, no. 3, pp. 518–528, June 1984.

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

[4] J.-Y. Jou and J. A. Abraham, "Fault-tolerant FFT networks," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 617–621, May 1988.

[5] 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. Rep. CRHC–91–6, UILU–ENG–91–2210.

[6] 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, no. 9, pp. 1132–1145, September 1990.

[7] C. J. Anfinson and F. T. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1599–1604, December 1988.

[8] F. T. Luk and H. Park, "Fault tolerant matrix triangularizations on systolic arrays," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1434–1438, November 1988.

[9] R. Sitaraman and N. K. Jha, "Optimal design of checks for error detection and location in fault-tolerant multiprocessor systems," *IEEE Trans. Comput.*, vol. 42, pp. 780–793, July 1993.

[10] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, and M. E. Zosel, *The High Performance Fortran Handbook*, Cambridge, MA: MIT Press, 1994.

[11] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, no. 10, pp. 1304–1308, October 1990.

[12] K.-H. Huang, "Fault-tolerant algorithms for multiple processor systems," Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, November 1983, Tech. Rep. CSG-20.

[13] V. Balasubramanian and P. Banerjee, "Tradeoffs in the design of efficient algorithm-based error detection schemes for hypercube multiprocessors," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 183–194, February 1990.

[14] B. M. McMillin and L. M. Ni, "Reliable distributed sorting through the application-oriented fault tolerance paradigm," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, no. 4, pp. 411–420, July 1992.

[15] A. Roy-Chowdhury and P. Banerjee, "A fault-tolerant parallel algorithm for iterative solution of the Laplace equation," *Proc. 22nd Int. Conf. Parallel Proc.*, pp. III–133–III–140, August 1993.

[16] A. Roy-Chowdhury, N. Bellas, and P. Banerjee, "Algorithm-based error detection schemes for iterative solution of partial differential equations," *IEEE Trans. Comput.*, accepted June 1995.

[17] F. T. Assaad and S. Dutt, "More robust tests in algorithm-based fault-tolerant matrix multiplication," *Proc. 22nd Int. Symp. Fault Tolerant Comput.*, pp. 430–439, June 1992.

[18] A. Roy-Chowdhury and P. Banerjee, "Tolerance determination for algorithm-based checks using simplified error analysis techniques," *Proc. 23rd Int. Symp. Fault Tolerant Comput.*, June 1993.

[19] P. Banerjee and J. A. Abraham, "Bounds on algorithm-based fault tolerance in multiple processor systems," *IEEE Trans. Comput.*, vol. C-35, pp. 296–306, April 1986.

[20] D. J. Rosenkrantz and S. S. Ravi, "Improved upper bounds for algorithm-based fault tolerance," *Proc. 26th Allerton Conf. Comm. Cont. and Comput.*, pp. 388–397, September 1988.

[21] D. Gu, D. J. Rosenkrantz, and S. S. Ravi, "Design and analysis of test schemes for algorithm-based fault tolerance," *Proc. 20th Int. Symp. Fault Tolerant Comput.*, pp. 106–113, June 1990.

[22] V. S. S. Nair, "Analysis and design of algorithm-based fault-tolerant systems," Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, August 1990, Tech. Rep. CRHC–90–3.

[23] B. Vinnakota and N. K. Jha, "A dependence graph based approach to the design of algorithm-based fault tolerant systems," *Proc. 20st Int. Symp. Fault Tolerant Comput.*, pp. 122–129, June 1990.

[24] B. Vinnakota and N. K. Jha, "Design of multiprocessor systems for concurrent error detection and fault diagnosis," *Proc. 21st Int. Symp. Fault Tolerant Comput.*, June 1991.

[25] D. M. Blough and A. Nicolau, "Fault tolerance in super-scalar and VLIW processors," *Proc. IEEE Workshop on Fault Tolerant Parallel Distrib. Syst.*, 1992.

[26] J. G. Holm and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions," *Proc. 21st Int. Conf. Parallel Proc.*, August 1992.

[27] C. Gong, R. Melhem, and R. Gupta, "Compiler assisted fault detection for distributed-memory systems," *Proc. Scalable High Perf. Comput. Conf.*, May 1994.

[28] C. Gong, R. Melhem, and R. Gupta, "Replicating statement execution for fault detection on distributed memory multiprocessors," *Proc. IEEE Workshop on Fault Tolerant Parallel Distrib. Syst.*, June 1994.

[29] V. Balasubramanian and P. Banerjee, "Compiler-assisted synthesis of algorithm-based checking in multiprocessors," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 436–446, April 1990.

[30] P. Banerjee, V. Balasubramanian, and A. Roy-Chowdhury, "Compiler assisted synthesis of algorithm-based checking in multiprocessors," in *Foundations of Dependable Computing*, G. Koob, Ed., vol. III: System Implementation, ch. 3, pp. 159–211. Boston, MA: Kluwer Academic Publishers, 1994.

[31] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD distributed memory machines," *Comm. Assoc. Comput. Mach.*, vol. 35, no. 8, August 1992.

[32] F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Elec. Comput.*, vol. EC–16, no. 6, pp. 848–854, December 1967.

[33] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital System Testing and Testable Design*, New York, NY: Computer Science Press, 1990.

[34] D. K. Pradhan, *Fault-Tolerant Computing: Theory and Techniques, Vol. II*, Englewood Cliffs, NJ: Prentice Hall, 1986.

[35] R. E. Blahut, *Theory and Practice of Error Control Codes*, Reading, MA: Addison-Wesley, 1984.

[36] V. S. S. Nair and J. A. Abraham, "Real number codes for fault-tolerant matrix operations on processor arrays," *IEEE Trans. Comput.*, vol. 39, no. 4, pp. 426–435, April 1990.

[37] G. Strang, *Linear Algebra and its Applications*, San Diego: Harcourt Brace Jovanovich, Publishers, 1988.

[38] A. Roy-Chowdhury, "Evaluation of algorithm based fault-tolerance techniques on multiple fault classes in the presence of finite precision arithmetic," M.S. thesis, Dept. of Elec. and Comput. Engg., Univ. of Illinois, Urbana-Champaign, August 1992, Tech. Rep. CRHC–92–15, UILU–ENG–92–2228.

[39] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore: Johns Hopkins University Press, 1987.

[40] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Multicomputers," *IEEE Computer*, vol. 28, no. 10, pp. 37–47, October 1995.

[41] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM compiler for distributed-memory message passing multicomputers," *Proc. First International Workshop on Parallel Processing*, December 1994.

[42] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, "Compiler optimizations for distributed-memory multicomputers used in the PARADIGM compiler," *Proc. 23rd Int. Symp. on Parallel Proc.*, pp. II:1–10, August 1994.

[43] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2 manual," Tech. Rep., CSRD, Univ. of Illinois, August 1990, Available via www from http://www.csrd.uiuc.edu:80/parafrase2.

[44] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten, "Parafrase–2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors," *Proc. 18th Int. Conf. Parallel Proc.*, pp. II:39–48, August 1989.

[45] E. W. Hodges IV, "High performance fortran support for the paradigm compiler," M.S. thesis, Dept. of Elec. and Comput. Engg., Univ. of Illinois, Urbana-Champaign, October 1995, Tech. Rep. CRHC–95–23.

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

[47] U. Banerjee, *Loop Transformations for Restructuring Compilers: Loop parallelization.* Boston, MA: Kluwer Press, 1993.

[48] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations.* Boston, MA: Kluwer Press, 1993.

[49] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York, NY: ACM Press, 1992.

[50] W. H. Brainerd, C. H. Goldberg, and J. C. Adams, *Programmer's Guide to Fortran 90.* New York, NY: McGraw Hill, 1990.

APPENDIX A

COMPILER OUTPUT

In this appendix, we show the code for three programs that were transformed by our compiler to generate error-detecting programs. For each of the programs, we show the input to our compiler, which is a Fortran program with HPF data distribution annotations, and the output generated by our compiler, which is an augmented version of the original serial program with additional code inserted for manipulating shadow arrays and checksums and additional data distribution directives specifying the distributions of the new data introduced by the compiler.

A.1 Matrix Multiplication

A.1.1 Input

```
        program mmul


        integer i,j,k
        double precision A(64,64),B(64,64),C(64,64)

!hpf$ processors p(4)
!hpf$ distribute (block,*) onto p :: A,C
!hpf$ distribute (*,block) onto p :: B

        do i = 1,64
        do j = 1,64
        C(i,j) = 0
        do k = 1,64
        C(i,j) = C(i,j) + A(i,k)*B(k,j)
        enddo
        enddo
        enddo

        end
```

A.1.2 Output

```
       PROGRAM mmul
       IMPLICIT NONE
       DOUBLE PRECISION $T_0(4,64)
       DOUBLE PRECISION $T(4,64)
       INTEGER $p
       INTEGER $i2
       INTEGER $i1
```

```
      DOUBLE PRECISION $cs1_c(4,64)
      DOUBLE PRECISION $cs1_a(4,64)
      DOUBLE PRECISION $cs2_c(64)
      DOUBLE PRECISION $b(64,64)
      DOUBLE PRECISION $a(64,64)
      DOUBLE PRECISION $c(64,64)
      INTEGER i, j, k
      DOUBLE PRECISION a(64,64)
      DOUBLE PRECISION b(64,64)
      DOUBLE PRECISION c(64,64)

!HPF$ PROCESSORS :: p(4)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO p :: a, c
!HPF$ DISTRIBUTE (*, BLOCK) ONTO p :: b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,hpf$0) WITH template$0(hpf$0 + 16,:) WRAP :: $b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$1(64, 64)
!HPF$ ALIGN (hpf$0,:) WITH template$1(hpf$0 + 16,:) WRAP :: $a
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$2(64, 64)
!HPF$ ALIGN (hpf$0,:) WITH template$2(hpf$0 + 16,:) WRAP :: $c
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ONTO p :: TEMPLATE$3(4,64)
!HPF$ ALIGN $cs1_c(hpf$0,:) WITH TEMPLATE$3(hpf$0+1,:) WRAP
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ONTO p :: TEMPLATE$4(4,64)
!HPF$ ALIGN $cs1_a(hpf$0,:) WITH TEMPLATE$4(hpf$0+1,:) WRAP

      DO $i1 = 1,64
        DO $i2 = 1,64
          $c($i1,$i2) = c($i1,$i2)
        END DO
      END DO
      DO $i1 = 1,64
        DO $i2 = 1,64
          $a($i1,$i2) = a($i1,$i2)
        END DO
      END DO
      DO $i1 = 1,64
        DO $i2 = 1,64
          $b($i1,$i2) = b($i1,$i2)
        END DO
      END DO
      DO i = 1,64
        $cs2_c(i) = (64 - 1 + 1) * 0
        DO j = 1,64
          c(i,j) = 0
        END DO
```

```
  END DO
  DO $p = 1,4
    DO $i2 = 1,64
      $cs1_a($p,$i2) = 0
    END DO
  END DO
  DO $i1 = 1,16
    DO $p = 1,4
      DO $i2 = 1,64
        $cs1_a($p,$i2) = $cs1_a($p,$i2) + a(($p - 1) * 16 + $i1,$i2
1       )
      END DO
    END DO
  END DO
  DO $p = 1,4
    DO $i2 = 1,64
      $cs1_c($p,$i2) = 0
    END DO
  END DO
  DO $i1 = 1,16
    DO $p = 1,4
      DO $i2 = 1,64
        $cs1_c($p,$i2) = $cs1_c($p,$i2) + c(($p - 1) * 16 + $i1,$i2
1       )
      END DO
    END DO
  END DO
  DO $i1 = 1,64
    DO $i2 = 1,64
      IF (compare($a($i1,$i2),a($i1,$i2)) .EQ. 1) CALL error_handle
1     r()
    END DO
  END DO
  DO $i1 = 1,64
    DO $i2 = 1,64
      IF (compare($b($i1,$i2),b($i1,$i2)) .EQ. 1) CALL error_handle
1     r()
    END DO
  END DO
  DO j = 1,64
    DO k = 1,64
      DO $p = 1,4
        $cs1_c($p,j) = $cs1_c($p,j) + $cs1_a($p,k) * b(k,j)
      END DO
    END DO
```

```
  END DO
  DO i = 1,64
    DO j = 1,64
      DO k = 1,64
        c(i,j) = c(i,j) + a(i,k) * b(k,j)
      END DO
    END DO
  END DO
  DO $p = 1,4
    DO $i2 = 1,64
      IF (compare($T($p,$i2),$cs1_a($p,$i2)) .EQ. 1) CALL error_han
1     dler()
    END DO
  END DO
  DO $p = 1,4
    DO $i2 = 1,64
      $T_0($p,$i2) = 0
    END DO
  END DO
  DO $i1 = 1,16
    DO $p = 1,4
      DO $i2 = 1,64
        $T_0($p,$i2) = $T_0($p,$i2) + c(($p - 1) * 16 + $i1,$i2)
      END DO
    END DO
  END DO
  DO $p = 1,4
    DO $i2 = 1,64
      IF (compare($T_0($p,$i2),$cs1_c($p,$i2)) .EQ. 1) CALL error_h
1     andler()
    END DO
  END DO
  DO $i1 = 1,64
    DO $i2 = 1,64
      IF (compare($b($i1,$i2),b($i1,$i2)) .EQ. 1) CALL error_handle
1     r()
    END DO
  END DO
  DO $i1 = 1,64
    DO $i2 = 1,64
      IF (compare($a($i1,$i2),a($i1,$i2)) .EQ. 1) CALL error_handle
1     r()
    END DO
  END DO
  END
```

A.2 Jacobi Solver

A.2.1 Input

```
      program jacobi
        double precision a(1000, 1000), b(1000, 1000)

!hpf$ processors p(2,2)
!hpf$ distribute (block,block) onto p :: A,B

C
        do k = 1, 100
          do j = 2, 999
            do i = 2, 999
              a(i, j) = (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) + b(i,
     *j + 1)) / 4
            enddo
          enddo
          do j = 2, 999
            do i = 2, 999
              b(i, j) = a(i, j)
            enddo
          enddo
        enddo
      end
```

A.2.2 Output

```
       PROGRAM jacobi
       DOUBLE PRECISION $T_0(1000,2)
       DOUBLE PRECISION $T(1000,2)
       INTEGER $p
       INTEGER $i2
       INTEGER $i1
       DOUBLE PRECISION $cs2_a(1000,2)
       DOUBLE PRECISION $cs2_b(1000,2)
       DOUBLE PRECISION $a(1000,1000)
       DOUBLE PRECISION $b(1000,1000)
       INTEGER p(2,2)
       DOUBLE PRECISION a(1000,1000)
       DOUBLE PRECISION b(1000,1000)
       INTEGER template$0(1000,1000)
       INTEGER template$1(1000,1000)
       INTEGER k, j, i

!HPF$ PROCESSORS :: p(2,2)
```

```
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO p :: a, b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, BLOCK) ONTO p :: template$0(1000, 1000)
!HPF$ ALIGN (hpf$0,hpf$1) WITH template$0(hpf$0 + 500,hpf$1) WRAP :: $a
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, BLOCK) ONTO p :: template$1(1000, 1000)
!HPF$ ALIGN (hpf$0,hpf$1) WITH template$1(hpf$0 + 500,hpf$1) WRAP :: $b
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO p :: TEMPLATE$2(1000,2)
!HPF$ ALIGN $cs2_a(hpf$0,hpf$1) WITH TEMPLATE$2(hpf$0+500,hpf$1+1) WRAP
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO p :: TEMPLATE$3(1000,2)
!HPF$ ALIGN $cs2_b(hpf$0,hpf$1) WITH TEMPLATE$3(hpf$0+500,hpf$1+1) WRAP

      DO $i1 = 2,999
        DO $i2 = 2,999
          $a($i1,$i2) = a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,1000
          $b($i1,$i2) = b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,2
          $cs2_a($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,500
          $cs2_a($i1,1) = $cs2_a($i1,1) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 501,999
          $cs2_a($i1,2) = $cs2_a($i1,2) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,2
          $cs2_b($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,500
          $cs2_b($i1,1) = $cs2_b($i1,1) + b($i1,$i2)
        END DO
      END DO
```

```
  DO $i1 = 2,999
    DO $i2 = 501,999
      $cs2_b($i1,2) = $cs2_b($i1,2) + b($i1,$i2)
    END DO
  END DO
  DO k = 1,100
    DO i = 2,999
      $cs2_a(i,1) = ($cs2_b(i - 1,1) + $cs2_b(i + 1,1) + $cs2_b(i,1
1     ) + (b(i,2 - 1) - b(i,500 - 1)) + $cs2_b(i,1) + (b(i,500 + 1)
2      - b(i,2 + 1))) / 4
      $cs2_a(i,2) = ($cs2_b(i - 1,2) + $cs2_b(i + 1,2) + $cs2_b(i,2
1     ) + (b(i,501 - 1) - b(i,999 - 1)) + $cs2_b(i,2) + (b(i,999 +
2     1) - b(i,501 + 1))) / 4
    END DO
    DO j = 2,999
      DO i = 2,999
        a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
1       ) / 4
      END DO
    END DO
    DO i = 2,999
      $cs2_b(i,1) = $cs2_a(i,1)
      $cs2_b(i,2) = $cs2_a(i,2)
    END DO
    DO j = 2,999
      DO i = 2,999
        b(i,j) = a(i,j)
      END DO
    END DO
  END DO
  DO $i1 = 2,999
    DO $p = 1,2
      $T($i1,$p) = 0
    END DO
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,500
      $T($i1,1) = $T($i1,1) + a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    DO $i2 = 501,999
      $T($i1,2) = $T($i1,2) + a($i1,$i2)
    END DO
  END DO
```

```
      DO $i1 = 2,999
        DO $p = 1,2
          IF (compare($T($i1,$p),$cs2_a($i1,$p)) .EQ. 1) CALL error_han
     1    dler()
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,2
          $T_0($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,500
          $T_0($i1,1) = $T_0($i1,1) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 501,999
          $T_0($i1,2) = $T_0($i1,2) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,2
          IF (compare($T_0($i1,$p),$cs2_b($i1,$p)) .EQ. 1) CALL error_h
     1    andler()
        END DO
      END DO
      END
```

A.3 ADI Integration

A.3.1 Input

```
        program ADI2d
        implicit none
        integer N, maxiter
        parameter (N = 256, maxiter = 100)
        double precision u(256,256), uh(256,256), b(256,256), alpha
        integer i, j, k

!hpf$ processors p(8)
!hpf$ distribute (block,*) onto p :: u,uh,b

c       *** Initial value for u
        do j = 1, 256
```

```
            do i = 1, 256
               u(i,j) = 0.0
            enddo
            u(1,j) = 30.0
            u(256,j) = 30.0
         enddo

c        *** Initialize uh
         do j = 1, 256
            do i = 1, 256
               uh(i,j) = u(i,j)
            enddo
         enddo

         alpha = 4 * (2.0 / 256)
         do k = 1, 100
c           *** Forward and backward sweeps along columns
            do j = 2, 255
               do i = 2, 255
                  b(i,j) = (2 + alpha)
                  uh(i,j) = (alpha - 2) * u(i,j) + u(i,j+1) + u(i,j-1)
               enddo
            enddo
            do j = 2, 255
               uh(2,j) = uh(2,j) + u(1,j)
               uh(255,j) = uh(255,j) + u(256,j)
            enddo

            do j = 2, 255
               do i = 3, 255
                  b(i,j) = b(i,j) - 1 / b(i-1,j)
                  uh(i,j) = uh(i,j) + uh(i-1,j) / b(i-1,j)
               enddo
            enddo
            do j = 2, 255
               uh(255,j) = uh(255,j) / b(255,j)
            enddo
            do j = 2, 255
               do i = 254, 2, -1
                  uh(i,j) = (uh(i,j) + uh(i+1,j)) / b(i,j)
               enddo
            enddo

c           *** Forward and backward sweeps along rows
            do j = 2, 255
```

```
      do i = 2, 255
         b(i,j) = (2 + alpha)
         u(i,j) = (alpha - 2) * uh(i,j) + uh(i+1,j) + uh(i-1,j)
      enddo
   enddo
   do i = 2, 255
      u(i,2) = u(i,2) + uh(i,1)
      u(i,255) = u(i,255) + uh(i,256)
   enddo

   do j = 3, 255
      do i = 2, 255
         b(i,j) = b(i,j) - 1 / b(i,j-1)
         u(i,j) = u(i,j) + u(i,j-1) / b(i,j-1)
      enddo
   enddo
   do i = 2, 255
      u(i,255) = u(i,255) / b(i,255)
   enddo
   do j = 254, 2, -1
      do i = 2, 255
         u(i,j) = (u(i,j) + u(i,j+1)) / b(i,j)
      enddo
   enddo
enddo
end
```

A.3.2 Output

```
PROGRAM adi2d
IMPLICIT NONE
DOUBLE PRECISION $T_3(256)
DOUBLE PRECISION $T_2(8,256)
DOUBLE PRECISION $T_1(256)
DOUBLE PRECISION $T_0(8,256)
DOUBLE PRECISION $T(256)
INTEGER $p
INTEGER $i2
INTEGER $i1
DOUBLE PRECISION $cs1_u(8,256)
DOUBLE PRECISION $cs1_uh(8,256)
DOUBLE PRECISION $cs2_b(256)
DOUBLE PRECISION $cs2_uh(256)
DOUBLE PRECISION $cs2_u(256)
DOUBLE PRECISION $b(256,256)
```

```
      DOUBLE PRECISION $uh(256,256)
      DOUBLE PRECISION $u(256,256)
      INTEGER n
      INTEGER maxiter
      PARAMETER(n = 256, maxiter = 100)
      DOUBLE PRECISION u(256,256)
      DOUBLE PRECISION uh(256,256)
      DOUBLE PRECISION b(256,256)
      DOUBLE PRECISION alpha
      INTEGER i, j, k

!HPF$ PROCESSORS :: p(8)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO p :: u, uh, b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$0(256, 256)
!HPF$ ALIGN (hpf$0,:) WITH template$0(hpf$0 + 32,:) WRAP :: $b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$1(256, 256)
!HPF$ ALIGN (hpf$0,:) WITH template$1(hpf$0 + 32,:) WRAP :: $uh
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$2(256, 256)
!HPF$ ALIGN (hpf$0,:) WITH template$2(hpf$0 + 32,:) WRAP :: $u
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ONTO p :: TEMPLATE$3(8,256)
!HPF$ ALIGN $cs1_u(hpf$0,:) WITH TEMPLATE$3(hpf$0+1,:) WRAP
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ONTO p :: TEMPLATE$4(8,256)
!HPF$ ALIGN $cs1_uh(hpf$0,:) WITH TEMPLATE$4(hpf$0+1,:) WRAP

      DO $i1 = 2,255
        DO $i2 = 1,256
          $uh($i1,$i2) = uh($i1,$i2)
        END DO
      END DO
      DO $i1 = 1,256
        DO $i2 = 1,256
          $u($i1,$i2) = u($i1,$i2)
        END DO
      END DO
      DO $i1 = 1,256
        $cs2_u($i1) = 0
      END DO
      DO $i1 = 1,256
        DO $i2 = 1,256
          $cs2_u($i1) = $cs2_u($i1) + u($i1,$i2)
        END DO
      END DO
      DO i = 1,256
        $cs2_u(i) = (256 - 1 + 1) * 0.0
      END DO
```

```
      DO j = 1,256
        DO i = 1,256
          u(i,j) = 0.0
        END DO
      END DO
      $cs2_u(256) = (256 - 1 + 1) * 30.0
      DO j = 1,256
        u(256,j) = 30.0
      END DO
      $cs2_u(1) = (256 - 1 + 1) * 30.0
      DO j = 1,256
        u(1,j) = 30.0
      END DO
      DO i = 1,256
        $cs2_uh(i) = $cs2_u(i)
      END DO
      DO j = 1,256
        DO i = 1,256
          uh(i,j) = u(i,j)
        END DO
      END DO
      alpha = 4 * (2.0 / 256)
      DO k = 1,100
        DO i = 2,255
          $cs2_b(i) = ((255 - 2 + 1) * (2 + alpha))
        END DO
        DO j = 2,255
          DO i = 2,255
            b(i,j) = (2 + alpha)
          END DO
        END DO
        DO $i1 = 2,255
          $cs2_u($i1) = 0
        END DO
        DO $i1 = 2,255
          DO $i2 = 2,255
            $cs2_u($i1) = $cs2_u($i1) + u($i1,$i2)
          END DO
        END DO
        DO i = 2,255
          $cs2_uh(i) = (alpha - 2) * $cs2_u(i) + ($cs2_u(i) + u(i,256)
     1    - u(i,2)) + ($cs2_u(i) - u(i,255) + u(i,1))
        END DO
        DO j = 2,255
          DO i = 2,255
```

```
    uh(i,j) = (alpha - 2) * u(i,j) + u(i,j + 1) + u(i,j - 1)
  END DO
END DO
$cs2_uh(1) = $cs2_uh(1) - $uh(1,1)
$cs2_u(1) = 0
DO $i2 = 2,255
  $cs2_u(1) = $cs2_u(1) + u(1,$i2)
END DO
$cs2_uh(2) = $cs2_uh(2) + $cs2_u(1)
DO j = 2,255
  uh(2,j) = uh(2,j) + u(1,j)
END DO
DO $i1 = 1,254
  $cs2_uh($i1) = $cs2_uh($i1) - $uh($i1,1)
END DO
$cs2_u(256) = 0
DO $i2 = 2,255
  $cs2_u(256) = $cs2_u(256) + u(256,$i2)
END DO
$cs2_uh(255) = $cs2_uh(255) + $cs2_u(256)
DO j = 2,255
  uh(255,j) = uh(255,j) + u(256,j)
END DO
DO $i1 = 2,255
  DO $i2 = 2,255
    $b($i1,$i2) = b($i1,$i2)
  END DO
END DO
DO $i1 = 2,255
  DO $i2 = 2,255
    $uh($i1,$i2) = uh($i1,$i2)
  END DO
END DO
DO $i1 = 2,255
  $T($i1) = 0
END DO
DO $i1 = 2,255
  DO $i2 = 2,255
    $T($i1) = $T($i1) + b($i1,$i2)
  END DO
END DO
DO $i1 = 2,255
  IF (compare($T($i1),$cs2_b($i1)) .EQ. 1) CALL error_handler()
END DO
DO j = 2,255
```

```
      DO i = 3,255
        $b(i,j) = $b(i,j) - 1 / $b(i - 1,j)
        b(i,j) = b(i,j) - 1 / b(i - 1,j)
        $uh(i,j) = $uh(i,j) + $uh(i - 1,j) / $b(i - 1,j)
        uh(i,j) = uh(i,j) + uh(i - 1,j) / b(i - 1,j)
      END DO
    END DO
    DO j = 2,255
      $uh(255,j) = $uh(255,j) / $b(255,j)
      uh(255,j) = uh(255,j) / b(255,j)
    END DO
    DO j = 2,255
      DO i = 254,2, -1
        $uh(i,j) = ($uh(i,j) + $uh(i + 1,j)) / $b(i,j)
        uh(i,j) = (uh(i,j) + uh(i + 1,j)) / b(i,j)
      END DO
    END DO
    DO i = 2,255
      $cs2_b(i) = ((255 - 2 + 1) * (2 + alpha))
    END DO
    DO j = 2,255
      DO i = 2,255
        b(i,j) = (2 + alpha)
      END DO
    END DO
    DO $i1 = 2,255
      $cs2_uh($i1) = 0
    END DO
    DO $i1 = 2,255
      DO $i2 = 2,255
        $cs2_uh($i1) = $cs2_uh($i1) + uh($i1,$i2)
      END DO
    END DO
    DO $i1 = 2,255
      DO $i2 = 2,255
        IF (compare($uh($i1,$i2),uh($i1,$i2)) .EQ. 1) CALL error_ha
1       ndler()
      END DO
    END DO
    DO i = 2,255
      $cs2_u(i) = (alpha - 2) * $cs2_uh(i) + $cs2_uh(i + 1) + $cs2_
1     uh(i - 1)
    END DO
    DO j = 2,255
      DO i = 2,255
```

```
    u(i,j) = (alpha - 2) * uh(i,j) + uh(i + 1,j) + uh(i - 1,j)
  END DO
END DO
DO $p = 1,8
  $cs1_uh($p,1) = 0
END DO
DO $i1 = 2,32
  $cs1_uh(1,1) = $cs1_uh(1,1) + uh($i1,1)
END DO
DO $i1 = 1,32
  DO $p = 2,7
    $cs1_uh($p,1) = $cs1_uh($p,1) + uh(($p - 1) * 32 + $i1,1)
  END DO
END DO
DO $i1 = 225,255
  $cs1_uh(8,1) = $cs1_uh(8,1) + uh($i1,1)
END DO
DO $p = 1,8
  $cs1_u($p,2) = 0
END DO
DO $i1 = 2,32
  $cs1_u(1,2) = $cs1_u(1,2) + u($i1,2)
END DO
DO $i1 = 1,32
  DO $p = 2,7
    $cs1_u($p,2) = $cs1_u($p,2) + u(($p - 1) * 32 + $i1,2)
  END DO
END DO
DO $i1 = 225,255
  $cs1_u(8,2) = $cs1_u(8,2) + u($i1,2)
END DO
$cs1_u(1,2) = $cs1_u(1,2) + $cs1_uh(1,1)
DO $p = 2,7
  $cs1_u($p,2) = $cs1_u($p,2) + $cs1_uh($p,1)
END DO
$cs1_u(8,2) = $cs1_u(8,2) + $cs1_uh(8,1)
DO i = 2,255
  u(i,2) = u(i,2) + uh(i,1)
END DO
DO $p = 1,8
  $cs1_uh($p,256) = 0
END DO
DO $i1 = 2,32
  $cs1_uh(1,256) = $cs1_uh(1,256) + uh($i1,256)
END DO
```

```
    DO $i1 = 1,32
      DO $p = 2,7
        $cs1_uh($p,256) = $cs1_uh($p,256) + uh(($p - 1) * 32 + $i1,
1       256)
      END DO
    END DO
    DO $i1 = 225,255
      $cs1_uh(8,256) = $cs1_uh(8,256) + uh($i1,256)
    END DO
    DO $p = 1,8
      $cs1_u($p,255) = 0
    END DO
    DO $i1 = 2,32
      $cs1_u(1,255) = $cs1_u(1,255) + u($i1,255)
    END DO
    DO $i1 = 1,32
      DO $p = 2,7
        $cs1_u($p,255) = $cs1_u($p,255) + u(($p - 1) * 32 + $i1,255
1       )
      END DO
    END DO
    DO $i1 = 225,255
      $cs1_u(8,255) = $cs1_u(8,255) + u($i1,255)
    END DO
    $cs1_u(1,255) = $cs1_u(1,255) + $cs1_uh(1,256)
    DO $p = 2,7
      $cs1_u($p,255) = $cs1_u($p,255) + $cs1_uh($p,256)
    END DO
    $cs1_u(8,255) = $cs1_u(8,255) + $cs1_uh(8,256)
    DO i = 2,255
      u(i,255) = u(i,255) + uh(i,256)
    END DO
    DO $i1 = 2,255
      DO $i2 = 2,255
        $b($i1,$i2) = b($i1,$i2)
      END DO
    END DO
    DO $i1 = 2,255
      DO $i2 = 2,255
        $u($i1,$i2) = u($i1,$i2)
      END DO
    END DO
    DO $p = 1,8
      $T_0($p,2) = 0
    END DO
```

```
      DO $i1 = 2,32
        $T_0(1,2) = $T_0(1,2) + u($i1,2)
      END DO
      DO $i1 = 1,32
        DO $p = 2,7
          $T_0($p,2) = $T_0($p,2) + u(($p - 1) * 32 + $i1,2)
        END DO
      END DO
      DO $i1 = 225,255
        $T_0(8,2) = $T_0(8,2) + u($i1,2)
      END DO
      DO $p = 1,8
        IF (compare($T_0($p,2),$cs1_u($p,2)) .EQ. 1) CALL error_handl
     1  er()
      END DO
      DO $i1 = 2,255
        $T_1($i1) = 0
      END DO
      DO $i1 = 2,255
        DO $i2 = 2,255
          $T_1($i1) = $T_1($i1) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,255
        IF (compare($T_1($i1),$cs2_b($i1)) .EQ. 1) CALL error_handler
     1  ()
      END DO
      DO j = 3,255
        DO i = 2,255
          $b(i,j) = $b(i,j) - 1 / $b(i,j - 1)
          b(i,j) = b(i,j) - 1 / b(i,j - 1)
          $u(i,j) = $u(i,j) + $u(i,j - 1) / $b(i,j - 1)
          u(i,j) = u(i,j) + u(i,j - 1) / b(i,j - 1)
        END DO
      END DO
      DO i = 2,255
        $u(i,255) = $u(i,255) / $b(i,255)
        u(i,255) = u(i,255) / b(i,255)
      END DO
      DO j = 254,2, -1
        DO i = 2,255
          $u(i,j) = ($u(i,j) + $u(i,j + 1)) / $b(i,j)
          u(i,j) = (u(i,j) + u(i,j + 1)) / b(i,j)
        END DO
      END DO
```

```
  END DO
  DO $i1 = 225,255
    $T_2(8,1) = $T_2(8,1) + uh($i1,1)
  END DO
  DO $p = 1,8
    IF (compare($T_2($p,1),$cs1_uh($p,1)) .EQ. 1) CALL error_handle
1   r()
  END DO
  DO $i1 = 2,255
    $T_3($i1) = 0
  END DO
  DO $i1 = 2,255
    DO $i2 = 2,255
      $T_3($i1) = $T_3($i1) + uh($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,255
    IF (compare($T_3($i1),$cs2_uh($i1)) .EQ. 1) CALL error_handler(
1   )
  END DO
  DO $i1 = 2,255
    DO $i2 = 2,255
      IF (compare($uh($i1,$i2),uh($i1,$i2)) .EQ. 1) CALL error_hand
1     ler()
    END DO
  END DO
  DO $i1 = 2,255
    DO $i2 = 2,255
      IF (compare($u($i1,$i2),u($i1,$i2)) .EQ. 1) CALL error_handle
1     r()
    END DO
  END DO
  END
```

VITA

Amber Roy-Chowdhury received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, in 1990. He was awarded the Master of Science degree in Electrical Engineering in 1992 from the University of Illinois, Urbana-Champaign, where he is currently completing a doctoral degree. During the summers of 1993 and 1994, he interned with the Fault Tolerant Systems Software Division at IBM T. J. Watson Research Center.

After completing his doctoral dissertation, Mr. Roy-Chowdhury will be joining IBM T. J. Watson Research Center as a Research Staff Member. His present research interests include high-performance and fault tolerant computing and performance analysis.