Algorithm-Based Error-Detection Schemes for Iterative Solution of Partial Differential Equations

Amber Roy-Chowdhury, *Student Member, IEEE*, Nikolas Bellas, *Student Member, IEEE*, and Prithviraj Banerjee, *Fellow, IEEE*

Abstract—Algorithm-based fault tolerance is an inexpensive method of achieving fault tolerance without requiring any hardware modifications. Algorithm-based schemes have been proposed for a wide variety of numerical applications. However, for a particular class of numerical applications, namely those involving the iterative solution of linear systems arising from discretization of various PDEs, there exist almost no fault-tolerant algorithms in the literature. In this paper, we first describe an error-detecting version of a parallel algorithm for iteratively solving the Laplace equation over a rectangular grid. This error-detecting algorithm is based on the popular successive overrelaxation scheme with red-black ordering. We use the Laplace equation merely as a vehicle for discussion; later in the paper we show how to modify the algorithm to devise error-detecting iterative schemes for solving linear systems arising from discretizations of other PDEs, such as the Poisson equation and a variant of the Laplace equation with a mixed derivative term. We also discuss a modification of the basic scheme to handle situations where the underlying solution domain is not rectangular. We then discuss a somewhat different error-detecting algorithm for iterative solution of PDEs which can be expected to yield better error coverage.

We also present a new way of dealing with the roundoff errors which complicate the check phase of algorithm-based schemes. Our approach is based on error analysis incorporating some simplifications and gives high fault coverage and no false alarms for a large variety of data sets. We report experimental results on the error coverage and performance overhead of our algorithm-based error-detection schemes on an Intel iPSC/2 hypercube multiprocessor.

The timing overheads of our error-detecting algorithms over the basic iterative algorithms involving no error detection decrease with increasing problem dimension and become small for large data sizes.

Index Terms—Algorithm-based fault-tolerance, parallel algorithms, partial differential equations, error analysis, fault injection.

1 Introduction

Algorithm-based fault tolerance (ABFT) is a well established technique which is used to develop reliable versions of numerical algorithms [1], [2]. Basically, an algorithm is modified so that the computed data elements preserve a certain property as the computation progresses. A lack of preservation of the property at the end of the algorithm indicates the presence of errors involving data computations. The algorithm-based fault tolerance technique for a particular application may be so designed so that the property is preserved not only at the end of all computation on data elements but also at suitable intermediate points during the course of the algorithm which may then serve as intermediate checkpoints. Often, variables are introduced to store the sums of various portions of computed data. At intermediate points and at the end of the algorithm, the computed data elements are summed and compared with the variables storing the checksum. A discrepancy would indicate an error. The low cost of algorithm-based schemes derives from the fact that it usually involves far fewer operations to update values of checksum variables than in the original algorithm. Reliable versions of such numerical algorithms as matrix multiplication [2], [3], Gaussian elimination [3], fast Fourier transform [4], [5], QR factorization [6], singular value decomposition [7], and several others have been developed in the past. Recently, ABFT techniques have been applied to a parallel bitonic sort algorithm [8], thus demonstrating that nonnumerical applications can also be made reliable by the use of algorithm-based checks.

In this paper, we develop low-overhead, error-detecting versions of iterative algorithms for solving the regular, sparse linear systems which arise from discretizations of various partial differential equations (PDEs). First, we discuss in detail an error-detecting version of the popular parallel algorithm-based on successive overrelaxation (SOR) with red-black ordering [9] for iteratively solving the linear system arising from a discretization of the Laplace equation. We merely use the Laplace equation as a representative of a class of PDEs for which similar low-overhead error-detecting algorithms may be devised; modifications of the error-detecting algorithm for the Laplace equation are discussed for other PDEs with a structure similar to the Laplace equation such as the Poisson equation and a vari-

• *The authors are with the Coordinated Science Lab, University of Illinois, 1308 W. Main St., Urbana, IL 61801.*
E-mail: {amber,bellas,banerjee}@crhc.uiuc.edu.

Manuscript received Nov. 8, 1993; revised Apr. 12, 1994. A short version of this paper appeared in Proceedings of ICCP '93.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number C96012.

ant of the Laplace equation with a mixed derivative term [10]. In our derivation of the error-detecting algorithm for solving the Laplace equation, we restrict our solution domain to be a rectangular grid for ease of discussion; in a later section we discuss a simple modification to the algorithm to deal with problems with differently shaped solution domains. We also discuss an alternative algorithm with higher overhead which may be used in situations where higher error coverage is desired.

We also use our reliable version to demonstrate a new technique for dealing with roundoff errors which can complicate the invariant check step in the algorithm. Our method for dealing with roundoff error applies error analysis techniques to derive expressions for error accumulation which are used to accumulate the roundoff error at runtime. This scheme gives better error coverage results than earlier techniques [7], [3], [11], and has been discussed in detail for other algorithms [12], [13], [14].

The organization of this paper is as follows. We first discuss the simple serial SOR algorithm for solving the Laplace equation in order to motivate the reader for the reliable parallel algorithm which is our goal. We demonstrate our low overhead invariant calculation and error analysis for the reliable version of the serial algorithm. We then present the reliable version of the parallel algorithm involving invariant and error bound computation. Next, we discuss how the error-detecting SOR algorithm presented for the Laplace equation may be modified to obtain error-detecting algorithms for solving other similar PDEs and the modification necessary to deal with irregular solution domains. We then discuss an alternative error detecting scheme which may be applied to the same class of iterative schemes which possesses higher overheads but which can also be expected to yield higher error coverage. We present experimental results on an Intel iPSC/2 hypercube indicating error coverage and timing overhead for both the error detecting parallel algorithms introduced in this paper. Finally, we compare our approach to deriving an error-detecting SOR algorithm to an earlier approach reported in [15] before concluding our paper.

2 Serial Algorithm for Solving the Laplace Equation Using Successive Overrelaxation

2.1 Serial SOR Algorithm

The Laplace equation is a second-order elliptic partial differential equation described by the following equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (1)$$

We may "solve" the Laplace equation numerically over a region by discretizing it in the x and y directions to obtain a grid of points and then computing the approximate solution values at these points. Assuming that the distances between neighboring points in the grid in the x and y directions is h, we may replace the partial derivatives appearing in (1) by finite difference approximations to obtain the following equation approximating (1) at the grid points.

$$u(x_{i+1}, y_j) + u(x_i, y_{j+1}) + u(x_{i-1}, y_j) + u(x_i, y_{j-1}) - 4u(x_i, y_j) = 0 \quad (2)$$

In (2), we have $x_i = ih$ and $y_j = jh$. Boundary conditions are usually specified by providing the values of $u(x_i, y_j)$ at the boundary points of the grid. Equation (2) then forms a basis for developing an algorithm for iteratively solving for the values of $u(x_i, y_j)$ at interior points in the grid. The algorithm is shown in Fig. 1 and is often referred to as the iterative Gauss-Jacobi technique [9]. Here the value $u(x_i, y_j)$ is assumed to be stored in the array element u[i][j]. It is assumed that the grid is discretized so that it has $n + 2$ points along each dimension, though the more general case of having a different number of grid points along each dimension does not add any complication to subsequent analyses. In Section 4, we discuss a modification by which even irregular solution domains may be handled. It is also assumed that rows 0 and $n + 1$ and columns 0 and $n + 1$ are initialized to store the boundary conditions. The termination condition is determined at runtime by specifying that the outer loop continue until the maximum difference over all grid points of a point value at the current iteration from its value at the previous iteration drops below a threshold. Note that the algorithm in Fig. 1 is assumed to execute the outermost loop *iter* times. We omit the convergence check in subsequent discussions since it has no bearing on the development of the error-detecting algorithm.

```
for(k=0;k<iter;k++)
{
   for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      t[i][j] = 0.25*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);

   for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      u[i][j] = t[i][j];
}
```

Fig. 1. Code for the solution of the Laplace equation by the iterative Gauss-Jacobi technique.

We may use the updated values of the u[i][j]s as soon as they are available in the update statement of the Gauss-Jacobi loop to obtain the code in Fig. 2, which is referred to as the iterative Gauss-Seidel technique. This modification results in asymptotically faster convergence to the steady state values [10]. We may further accelerate the rate of convergence to the steady state value by introducing a parameter ω to overrelax the computation of Fig. 2, so that each updated point is now computed as an average of its old value and the values of its four neighboring points, as shown in Fig. 3. The code of Fig. 3 is often referred to as a successive overrelaxation iterative method (SOR) and ω is referred to as the relaxation parameter. A careful choice of ω results in much faster convergence rates than the previous two schemes. The parameter ω lies between 1 and 2 for an overrelaxation technique; in some cases it is useful to choose ω between 0 and 1 instead—this is referred to as an

underrelaxation technique. The correctness of the error-detecting algorithm we derive in this paper is not dependent on the range of ω in any way, so that both overrelaxation and underrelaxation algorithms may be modified to become error-detecting in an identical manner. Thus the term overrelaxation in the rest of the paper may refer to both overrelaxation and underrelaxation techniques. The code of Fig. 3 is hard to parallelize since each update of u[i][j] involves new values of u[i][j – 1] and u[i – 1][j] and old values of u[i][j + 1] and u[i + 1][j]. However, a modified SOR algorithm may be devised with the same asymptotic rate of convergence as the algorithm of Fig. 3 which is well suited for parallel implementation. In this algorithm, the grid points are divided into two disjoint sets. We refer to those u[i][j] for which $i + j$ is even as red points and those for which $i + j$ is odd as black points. Then, each update of Fig. 3 may be split up into two updates, one involving only the red points and the other involving only the black points, as shown in Fig. 4. The code of Fig. 4 is referred to as a Red-Black SOR method. The update of each red point utilizes the values of its four neighbors, which are black points, and the update of each black point utilizes the values of its four neighbors, which are red points. Thus, parallelizing the code of Fig. 4 poses no problem since we may first update all red points in parallel and then update all black points in parallel. We refer to this algorithm as the Red-Black SOR algorithm. The code of Fig. 4 forms the basis of our fault-tolerant serial and parallel algorithms.

```
for(k=0;k<iter;k++)
{
   for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      u[i][j] = 0.25*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);
}
```

Fig. 2. Code for the solution of the Laplace equation by the iterative Gauss-Seidel technique.

```
for(k=0;k<iter;k++)
{
   for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      u[i][j] = (1-w)*u[i][j]+0.25*w*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);
}
```

Fig. 3. Successive overrelaxation code for the solution of the Laplace equation.

2.2 Serial Red-Black SOR Algorithm with Checks

The idea for a fault-tolerant algorithm for the Red-Black SOR algorithm stems from the idea of maintaining checksums based on the linearity property [7]. We assume that n is even, though the case when n is odd may be treated similarly. For each red-black update, we make the following observations:

1) Every red interior point is used in the update of exactly one red point and every black interior point is used in the update of exactly one black point.
2) Every black boundary point is used in the update of exactly one red point and every red boundary point is used in the update of exactly one black point.
3) The black points u[1][n] and u[n][1] are used in the update of exactly two red points. The red points u[1][1] and u[n][n] are used in the update of exactly two black points.
4) The black points for which i = 1 or i = n or j = 1 or j = n except for u[1][n] and u[n][1] are used in the update of exactly three red points. The red points for which i = 1 or i = n or j = 1 or j = n except for u[1][1] and u[n][n] are used in the update of exactly three black points.
5) All other black points are used in the update of exactly four red points. All other red points are used in the update of exactly four black points.

Observation 1) follows directly from the statement for updating the red points in Fig. 4. The remaining observations are illustrated in Fig. 5 for the black points. Observations 1) through 5) lead us to maintain a sum on all the red points and update this sum at the end of each update of the red points. Let us denote the sum of all red points by S_R, the sum of all red boundary points by S_{RB}, the sum of u[1][1] and u[n][n] by S_{RC} the sum of the red points of 4) by S_{RO} and the sum of the red points of 5) by S_{RI}. The corresponding sums of the black points are denoted by S_B, S_{BB}, S_{BC}, S_{BO}, and S_{BI}. Each red interior point contributes $(1 - \omega)$ times its value to the updated value of a red point while each black point contributes 0.25ω times its value to each red point update it is involved in. Similarly, each black interior point contributes $(1 - \omega)$ times its value to the updated value of a black point while each red point contributes 0.25ω times its value to each black point update it is involved in. These last two observations, along with observations 1) through 5), allow us to update S_R and S_B following the updates of the red and black points respectively in terms of the other sums as follows:

$$S_R \leftarrow (1 - \omega)S_R + \omega(S_{BI} + 0.75S_{BO} + 0.5S_{BC} + 0.25S_{BB}) \quad (3)$$

$$S_B \leftarrow (1 - \omega)S_B + \omega(S_{RI} + 0.75S_{RO} + 0.5S_{RC} + 0.25S_{RB}) \quad (4)$$

Following the update of the red points, besides updating S_R as shown in (3), we also need to recompute S_{RO}, S_{RC}, and S_{RI} since these have changed also. Similarly, following the update of the black points we need to recompute S_{BO}, S_{BC}, and S_{BI}. The boundary elements are constant and so we do not need to recompute S_{RB} or S_{BB}. (Note that in the parallel implementation, the boundary elements for each processor's portion of the grid may need to be recomputed, since these may be interior points in the global grid.) S_{RO} and S_{BO} require $O(n)$ operations to compute, while computing S_{RC} and S_{BC} require just one addition operation. Since S_R is the sum of all red interior points, i.e., $S_R = S_{RO} + S_{RC} + S_{RI}$, and similarly S_B is the sum of all black interior points, i.e., $S_B = S_{BO} + S_{BC} + S_{BI}$, we may compute S_{RI} and S_{BI} in constant time once the red and black updates, respectively, and the updates of all other red and black sums, respectively, have been completed, using the following equations

$$S_{RI} \leftarrow S_R - S_{RO} - S_{RC} \quad (5)$$

$$S_{BI} \leftarrow S_B - S_{BO} - S_{BC} \quad (6)$$

After a predefined number of iterations has been executed, the sum of the red interior points is compared with S_R, and the sum of the black interior points is compared with S_B. The check could be applied after every iteration but in the interests of low overhead the check could be applied after a user specified number of iterations have been completed. In the absence of computational faults, these values should be equal to within a tolerance. (The tolerance is necessary due to roundoff error accumulation). The code for the Red-Black SOR algorithm with additional checksums for error detection is shown in Fig. 6.

```
for(k=0;k<iter;k++)
{
  /* update red points */
  for(i=1;i<=n;i++)
  {
      for(j=2-(i%2);j<=n;j+=2)
      u[i][j] = (1-w)*u[i][j] + 0.25*w*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);
  }
  /* update black points */
  for(i=1;i<=n;i++)
  {
      for(j=1+(i%2);j<=n;j+=2)
      u[i][j] = (1-w)*u[i][j] + 0.25*w*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);
  }
}
```

Fig. 4. Red-Black SOR code.

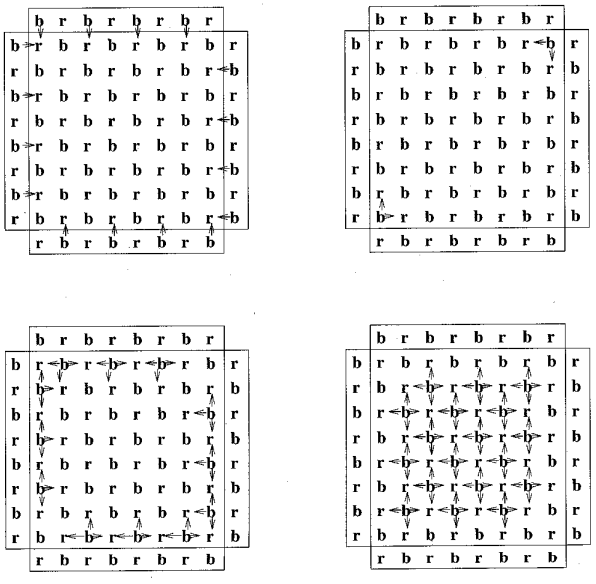


Fig. 5. Figure showing couplings of black points with red points.

```
for(k=0;k<iter;k++)
{
  /* update red points */
  for(i=1;i<=n;i++)
  {
      for(j=2-(i%2);j<=n;j+=2)
      u[i][j] = (1-w)*u[i][j] + 0.25*w*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);
  }
  /* update red sums */
  SR = (1-w)*SR + w*(SBI+0.75*SBO+0.5*SBC+0.25*SBB);
  SRO = 0;
  for(j=3;j<=n;j+=2)
  SRO += u[1][j];
  for(j=2;j<n;j+=2)
  SRO += u[n][j];
  for(i=3;i<=n;i+=2)
  SRO += u[i][1];
  for(i=2;i<n;i+=2)
  SRO += u[i][n];
  SRC = u[1][1]+u[n][n];
  SRI = SR-SRO-SRC;
  /* similarly update black points and black sums */
                .
                .
                .
}
/* check sum of red points */
SUM = 0;
for(i=1;i<=n;i++)
    for(j=2-(i%2);j<=n;j+=2)
    SUM += u[i][j];
if(abs(SUM-SR)>tolerance)
error();
/* similarly check sum of black points */
                .
                .
                .
```

Fig. 6. Red-Black SOR code with checksums for error detection.

2.3 Error Bound Derivation

Due to differences in roundoff error accumulation in the red and black points and S_R and S_B, one has to allow for a threshold when comparing between the sum of the red points and S_R and the sum of the black points and S_B. Previous researchers have either suggested an experimental evaluation of the threshold [7], [3] which is at best useful for data sets of fixed size and limited data range or a mantissa checksum test [1] which suffers from the disadvantage of not being able to check floating point additions without recourse to duplication or experimental determination of the threshold. We deal with the thresholding problem in a different manner by computing the error expressions for the variables involved in the computation. In order to simplify our error analysis, we compute the global error at the end of one iteration of the outer loop in terms of the local error for the loop and the global error accumulation upto the end of the previous iteration. A more detailed discussion of this approach and its application to several other algorithms can be found in [12]. Error analysis of the code in Fig. 6 forms the subject of the following paragraphs.

The statements which we need to analyze for error are of four types

1) $a = b + c$
2) $a = b - c - d$
3) $a = Ab + BC(c + d + e + f)$
4) $a = Ab + B(c + Cd + De + Ef)$

where lowercase letters denote variables and capitals denote constants. In subsequent discussions, we use the notation $|err(v)|$ to denote an upper bound on the absolute error associated with a variable v. We further use $\hat{v}$ to denote the value of a variable v including the floating point errors accumulated in computing v. Thus, we can relate $\hat{v}$, v, and $err(v)$ by

$$|\hat{v}| \leq |v| + |err(v)| \tag{7}$$

We now proceed to derive error bounds for the expressions in 1), 2), 3), and 4) above using the following fundamental lemma concerning error accumulation in floating point computations.

LEMMA 1. *A floating point computation of z, denoted by $z = fl(x \oplus y)$, (where $\oplus$ denotes any floating point operator) results in an error of $err(z) = (x \oplus y)\delta$ (i.e., $fl(x \oplus y) = (x \oplus y)(1 + \delta)$), where $|\delta| \leq \epsilon = 2^{-t}$, where t denotes the mantissa size.*

PROOF. Refer to [16], page 113. □

In subsequent derivations we will assume that all elements taking part in the floating point computations are positive. Thus, the absolute values of each element are the same as their actual values. Also, in this case, the δ of Lemma 1 is always positive. We will indicate in Section 2.4 how to initially transform the values on the boundary and interior of the grid so that at all subsequent steps, the value of each grid point is positive.

The error expression for 1) is given by the following theorem.

THEOREM 1. *The error accumulation in the computation of the floating point expression*

$$\hat{a} = fl(\hat{b} + \hat{c}) \tag{8}$$

is given by

$$err(a) \leq a\epsilon + err(b) + err(c) \tag{9}$$

PROOF. By Lemma 1 we then have $\hat{a} = (\hat{b} + \hat{c})(1 + \delta_1)$. Using Lemma 1 to get the bound $\delta_i \leq \epsilon$ and discarding the $O(\epsilon^2)$ terms which arise under the assumption that they are negligible compared to the $O(\epsilon)$ terms, we obtain the following relation

$$\hat{a} \leq \hat{b} + \hat{c} + (b + c)\epsilon \tag{10}$$

Using (7) to substitute for the hatted variables in (10) and once again using the simplification of discarding the $O(\epsilon^2)$ terms which arise, we get the following bound on the value of $\hat{a}$

$$\hat{a} \leq a + (b + c)\epsilon + err(b) + err(c) \tag{11}$$

from where we obtain the following relation for $err(a)$

$$err(a) \leq (b + c)\epsilon + err(b) + err(c) \tag{12}$$

Substituting $a = b + c$ in the above expression completes the proof. □

The error expression for 2) is given by the following theorem

THEOREM 2. *The error accumulation in the computation of the floating point expression*

$$\hat{a} = fl(\hat{b} - \hat{c} - \hat{d}) \tag{13}$$

is given by

$$err(a) \leq (2b + 2c + d)\epsilon + err(b) + err(c) + err(d) \tag{14}$$

PROOF. Let us denote by t_1 the floating point computation of $\hat{b} - \hat{c}$, i.e., we have $t_1 = (\hat{b} - \hat{c})$. By Lemma 1, we then have $t_1 = (\hat{b} - \hat{c})(1 + \delta_1)$. We now have $\hat{a} = fl(t_1 - \hat{d})$. By applying Lemma 1 once again we get $\hat{a} = (t_1 - \hat{d})(1 + \delta_2)$. By expanding out t_1 and using Lemma 1 to get the bound $|\delta_i| \leq \epsilon$ and discarding the $O(\epsilon^2)$ terms which arise under the assumption that they are negligible compared to the $O(\epsilon)$ terms, we obtain the following relation

$$\hat{a} \leq \hat{b} - \hat{c} - \hat{d} + (2\hat{b} + 2\hat{c} + \hat{d})\epsilon \tag{15}$$

Using (7) to substitute for the hatted variables in (15) and once again using the simplification of discarding the $O(\epsilon^2)$ terms which arise, we get the following bound on the value of $\hat{a}$

$$\hat{a} \leq a + (2b + 2c + d)\epsilon + err(b) + err(c) + err(d) \tag{16}$$

from where we obtain the following relation for $err(a)$

$$err(a) \leq (2b + 2c + d)\epsilon + err(b) + err(c) + err(d) \tag{17}$$

which completes the proof. □

The error expression for 3) is given by the following theorem:

THEOREM 3. *The error accumulation in the computation of the floating point expression*

$$\hat{a} = fl(A\hat{b} + BC(\hat{c} + \hat{d} + \hat{e} + \hat{f})) \tag{18}$$

is given by

$$err(a) \leq 2a\epsilon + BC(4c + 4d + 3e + 2f)\epsilon + Aerr(b) + BC(err(c) + err(d) + err(e) + err(f)) \tag{19}$$

PROOF. Let us denote by t_1, t_2, and t_3 the floating point computations $A\hat{b}$, BC, and $\hat{c} + \hat{d} + \hat{e} + \hat{f}$, respectively. Then, applying Lemma 1, we have $t_1 = A\hat{b}(1 + \delta_1)$ and $t_2 = BC(1 + \delta_2)$ while we may show

$$t_3 \leq \hat{c} + \hat{d} + \hat{e} + \hat{f} + (3\hat{c} + 3\hat{d} + 2\hat{e} + \hat{f})\epsilon$$

by repeated applications of Lemma 1 in a manner similar to the proof of Theorem 2. Next let us denote by t_4 the floating point product $fl(t_2 t_3)$. Applying Lemma 1, we obtain $t_4 = t_2 t_3(1 + \delta_3)$. Thus, we may write $\hat{a} = fl(t_1 + t_4)$ which may be similarly written as $\hat{a} = (t_1 + t_4)(1 + \delta_4)$ by an application of Lemma 1. Now expanding out each of the t_is, using Lemma 1 to bound each δ_i and discarding the $O(\epsilon^2)$ terms which

arise, we get the following equation

$$\hat{a} \leq A\hat{b} + BC(\hat{c} + \hat{d} + \hat{e} + \hat{f}) + (2A\hat{b} + 3BC(\hat{c} + \hat{d} + \hat{e} + \hat{f})\epsilon + BC(3\hat{c} + 3\hat{d} + 2\hat{e} + \hat{f})\epsilon \quad (20)$$

We now use (7) to substitute for the hatted variables in (20) to obtain the following equation

$$\hat{a} \leq (a + (2Ab + 3BC(c + d + e + f))\epsilon + BC(3c + 3d + 2e + f)\epsilon + Aerr(b) + BC(err(c) + err(d) + err(e) + err(f))\epsilon \quad (21)$$

from where we obtain the relation

$$err(a) \leq (2Ab + 3BC(c + d + e + f))\epsilon + BC(3c + 3d + 2e + f)\epsilon + Aerr(b) + BC(err(c) + err(d) + err(e) + err(f)) \quad (22)$$

Sustituting $a = Ab + BC(c + d + e + f)$ in the above expression proves the theorem. □

The error expression for the floating point computation of 4) is as follows

THEOREM4. *The error accumulation in the computation of the floating point expression*

$$\hat{a} = A\hat{b} + B(\hat{c} + C\hat{d} + D\hat{e} + E\hat{f}) \quad (23)$$

is given by

$$err(a) \leq (2a + B(3c + 4Cd + 3De + 2Ef))\epsilon + Aerr(b) + B(err(c) + Cerr(d) + Derr(e) + Eerr(f)) \quad (24)$$

PROOF. The derivation is similar to the proof of Theorem 3 and is omitted. □

2.4 Modified Algorithm with Checks and Error Bounding

The derivation of the error expressions now enables us to write a fault-tolerant version of the algorithm in Fig. 7 which also computes the error bounds on the fly. We make two initial transformations to the original problem. Initially, before starting the iterative solution process, we determine the value of the negative boundary element of largest magnitude, say M and add $|M|$ to all points on the grid (boundary and interior). This results in all points on the grid taking on positive initial values. It may easily be verified that from this point on, the u[i][j]s computed in each iteration exceed the corresponding u[i][j]s of the unmodified problem by exactly $|M|$. Thus, the solution to the original problem may be recovered by subtracting $|M|$ from each of the u[i][j]s computed in the modified problem. Also note that the u[i][j] values in the modified problem are always positive, so that the error expressions of Section 2.3 may be directly applied. Another initial modification which we introduce is scaling the elements by dividing each element by the largest boundary element after translating them by the value of the negative boundary element of largest magnitude. Again, it may be easily verified that following the latter modification, the magnitude of every element in subsequent iterations of the algorithm lies between 0 and 1, leading to an easy correctness check at intermediate points in the algorithm. The solution to the original problem at any point may be recovered by simply scaling the elements with the reciprocal of the boundary element of largest magnitude.

```
for(k=0;k<iter;k++)
{
  /* update red points */
   for(i=1;i<=n;i++)
   {
      for(j=2-(i%2);j<=n;j+=2)
      u[i][j] = (1-w)*u[i][j] + 0.25*w*(u[i][j-1]+u[i][j+1]+u[i-1][j]+u[i+1][j]);
   }
   /* update red sums and error variables */
   SR = (1-w)*SR + w*(SBI+0.75*SBO+0.5*SBC+0.25*SBB);
   errSR_ = 2*SR+3.25*w*SB+(1-w)*errSR_+w*errSB_;
   errSR = (1-w)*errSR+w*(errSBI+0.75*errSBO+0.5*errSBC+0.25*errSBB)
           +2*w*SR+w*(2*SBI+3*SBO+1.5*SBC+0.5*SBB);
   SRO = errSRO = 0;
   for(j=3;j<=n;j+=2)
   {
      SRO += u[1][j];
      errSRO += SRO;
   }
   for(j=2;j<n;j+=2)
   {
      SRO += u[n][j];
      errSRO += SRO;
   }
   for(i=3;i<=n;i++)
   {
      SRO += u[i][1];
      errSRO += SRO;
   }
   for(i=2;i<n;i+=2)
   {
      SRO += u[i][n];
      errSRO += SRO;
   }
   SRC = u[1][1]+u[n][n];
   errSRC  = SRC;
   SRI = SR-SRO-SRC;
   errSRI = errSR+errSRO+errSRC+2*(SR+SRO)+SRC;

   /* similarly update black points and black sums */
                .
                .
                .
}

/* check sum of red points */
errSUM =  SUM = 0;
for(i=1;i<=n;i++)
   for(j=2-(i%2);j<=n;j+=2)
   {
      SUM += u[i][j];
      errSUM += SUM;
   }

   if(abs(SUM-SR)>(errSR+errSR_+errSUM)*pow(2.0,-mantissa_size))
   error();

/* similarly check sum of black points */
                .
                .
                .
```

Fig. 7. Error detecting SOR code incorporating error expression computation.

Using Theorems 1 through 4 to compute error expressions, we get the code shown in Fig. 7. There are three points to note here.

First, we store the error bound for a variable called v in a variable called errv. For each of the sum variables, the error variables may be updated by using one of the three error

expressions derived in Section 2.3. Thus, to keep track of the error in updating the value of the checksum variables storing the sum of the red and the black points (denoted by SR and SB, respectively), we have the error variables errSR and errSB. However, we need to introduce two extra error variables errSR_ and errSB_ which keep track of the total error accumulation of the red and black points respectively (as opposed to the error in updating the variables which store the checksums). The expressions for updating errSR_ and errSB_ in each iteration shown in Fig. 7 may be derived by summing over all red and black points, respectively, the error expressions for each individual red or black point which may be obtained by an application of Theorem 3. The sums of the errors over the red points are bounded by the old value of errSR_ while the sums of the errors over the black points are bounded by the old value of errSB_ since the values of the black boundary points never change and thus do not incorporate any roundoff errors.

Second, wherever error bounds for individual elements of u[i][j] arise in our error expressions, we drop them since maintaining error bounds for individual grid elements would require too much computation overhead. This approximation still leads to the error expressions acting as upper bounds on the error in practice.

Third, we postpone the multiplication of the error variables by ϵ to the very end, when the error variables are actually used to compute the tolerance for the check on the sums of the red and black points. This saves some extra computations.

3 Parallel Algorithm for Solving the Laplace Equation

3.1 Parallel SOR Algorithm

The serial Red-Black SOR code with checks shown in Fig. 6 lends itself easily to efficient implementation on a parallel machine. We assume that the underlying architecture is a 2-dimensional mesh with N_1 and N_2 processors in each dimension. For simplicity, we assume that the grid dimensions (assumed to be n in each direction) are divisible by $2N_1$ and $2N_2$. Then, the initial data distribution is blockwise with the processor in the mth row and lth column in the mesh receiving the block containing the interior points u[i][j], $\frac{mn}{N_1}+1 \le i \le \frac{(m+1)n}{N_1}$, $\frac{ln}{N_2}+1 \le j \le \frac{(l+1)n}{N_2}$. The boundary points are distributed to processors holding the adjacent interior points. This is shown in Fig. 8 for a 4×4 mesh of processors. At the end of each iteration, each processor in the mesh receives from its north and south neighbors (if they exist) messages containing the elements of u adjoining the elements in its top and bottom rows respectively. Similarly, each processor also receives from its east and west neighbors messages containing the elements of u adjoining the elements in its rightmost and leftmost columns respectively. Each processor then updates its portion of the grid in a manner identical to the serial algorithm, and then proceeds with the next iteration of the algorithm.

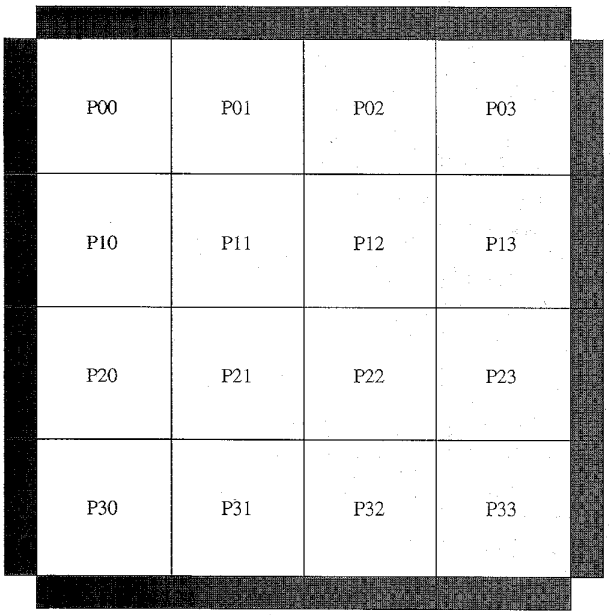


Fig. 8. Data distribution for parallel SOR code.

3.2 Parallel Algorithm with Checks and Error Bounding

From the discussion in Section 3.1, we note that from the point of view of a single processor, the computations performed by it are exactly the same as the serial algorithm being used to solve a problem of smaller dimensions, except possibly for nonconstant boundary elements received from neighboring processors. Thus we may incorporate fault tolerance into the parallel algorithm in the same manner as for the serial algorithm by introducing the variables for keeping track of the sums of the red and black points of each processor's local block of u and using the same expressions as for the serial algorithm. Our error bound expressions for the sum variables which were developed for the serial algorithm via Theorems 1 through 4 are valid here as well since now each processor performs the same kind of operations as in the serial case, except that the boundary elements for the processor may not be fixed in this case. Thus, we need to recompute the sum of the red and black boundary elements following the red and black updates in each iteration, unlike the serial algorithm where these sums only needed to be computed once before the beginning of the iterative updates. Besides dropping error variables involving individual elements of u, as in the serial algorithm, we make the further simplification of dropping error variables involving the boundary elements wherever they arise in updating the error expressions for a particular processor since otherwise we would have to maintain error variables for each element of u, an unacceptably large overhead. (Recall that in the serial algorithm, the boundary elements were not computed on and thus did not accumulate any roundoff errors). We have found that even with this simplification, our error expressions act as an upper bound on the error in practice. Also, there is a possibility that boundary elements received from a neighbor might be in error. We

therefore translate and scale the initial data as for the serial algorithm. Recall that this ensures that the values of all elements involved in the computations lie between 0 and 1 at any point in the algorithm. This property may be used to check the correctness of the boundary data received from neighbors. Gross errors which cause any element to take on values less than 0 or greater than 1 can be immediately detected. More subtle errors are likely to be detected when we compare the values of SR and SB and the sum over all red and black points of u, respectively, on every processor at the conclusion of the algorithm. In the interests of high error coverage, processors could be paired so that each processor maintains checksums on not its own data, but on the data of its pair processor. This would double the communication at the start of each iteration, since each communicated data element would now have to be sent to the pair processor as well in order to enable it to perform the checksum updates. Alternatively, if the algorithm is devised to detect errors caused by transient faults, it would suffice for each processor to maintain checksums on its own data, preserving the communication requirements of the original algorithm. In Section 5, we describe an alternative error-detecting approach where communicated data is subjected to a more rigorous check prior to its use in subsequent updates.

4 Modifications to Error-Detection Algorithm

In this section we discuss how the error-detecting parallel algorithm for solving the Laplace equation may be modified to handle other PDEs. We also discuss a modification to handle the case where the solution domain is not rectangular.

4.1 Application of the Error-Detection Scheme to Other PDEs

4.1.1 Poisson Equation

The Poisson equation has a form similar to the Laplace equation except that the right hand side of the equation, instead of being zero, is some function f of x and y, as shown below.

$$u_{xx} + u_{yy} = f \tag{25}$$

As was the case for the Laplace equation, we may obtain a discrete analog of the Poisson equation over a grid by replacing the partial derivatives in (25) by centered difference approximations to obtain the following equation for computing the value at a grid point in terms of the values of its neighbors and the value of the function f at that point

$$u(x_{i+1}, y_j) + u(x_i, y_{j+1}) + u(x_{i-1}, y_j) + u(x_i, y_{j-1}) - 4u(x_i, y_j) = h^2 f(x_i, y_j) \tag{26}$$

where, as before $x_i = ih$, $y_j = jh$, and h is the spacing between adjacent points in the grid in both the x and y dimensions of the grid.

As was the case for (2), (26) may also be used to derive an iterative SOR method using Red-Black ordering in which the updates of red and the black points proceed in an almost identical fashion to the updates shown in Fig. 4 except that now we have an additional term of $-\omega h^2 f[i][j]/4$ (assuming that the value $f(x_i, y_j)$ is stored in the array element f[i][j]) added to u[i][j] during the update of u[i][j]. At the start of the algorithm, we compute the sum of f[i][j]s for the red points and the black points and store them in the variables F_R and F_B respectively. Since the f[i][j]s are not modified during the course of the algorithm, the values of F_R and F_B remain unchanged throughout the algorithm. The equations for modifying S_R and S_B at the end of each iteration of the algorithm are then changed as follows

$$S_R = (1 - \omega)S_R + \omega(S_{BI} + 0.75S_{BO} + 0.5S_{BC} + 0.25S_{BB}) + 0.25h^2\omega F_R \tag{27}$$

$$S_B = (1 - \omega)S_B + \omega(S_{RI} + 0.75S_{RO} + 0.5S_{RC} + 0.25S_{RB}) + 0.25h^2\omega F_B \tag{28}$$

where the S_Xs are defined as earlier. Note that the only difference between (3) and (4) and (27) and (28) is the addition of the terms involving F_R and F_B. The equations for updating S_{RI} and S_{BI} are exactly the same as (5) and (6).

4.1.2 Laplace Equation with Mixed Derivative Term

Consider a variant of the Laplace equation given by the following equation

$$u_{xx} + u_{yy} + au_{xy} = 0 \tag{29}$$

where a is a constant. Upon using standard finite difference approximations for the partial derivatives as before in the above equation, we obtain the following equation

$$u(x_{i+1}, y_j) + u(x_{i-1}, y_j) + u(x_i, y_{j+1}) + u(x_i, y_{j-1}) - 4(u(x_i, y_j) + \frac{a}{4}[u(x_{i+1}, y_{j+1}) - u(x_{i-1}, y_{j+1}) - u(x_{i+1}, y_{j-1}) + u(x_{i-1}, y_{j-1})] = 0 \tag{30}$$

where x_i, y_j, and h have their earlier meanings. If we use (30) as a basis for the iterative solution of (29) over the grid, we notice that an update of u[i][j] (where u[i][j] is assumed to be defined as earlier) requires the values of not only u[i + 1][j], u[i – 1][j], u[i][j – 1], and u[i][j + 1] as for the Poisson and Laplace equations, but also the values of u[i + 1][j + 1], u[i + 1][j – 1], u[i – 1][j + 1], and u[i – 1][j – 1] as well. This is illustrated in Fig. 9. A parallel SOR algorithm may be developed based on (30). Multiple colors are introduced and each grid point is assigned one color so that each point requires the values of only points colored differently from itself for its update. One SOR update is then composed of multiple Jacobi-like sweeps, one for each color, where first we update all points of a particular color in parallel, then we update all points of the next color, and so on until we update all points of the last color in the last sweep [10]. For (30), four colors in the ordering indicated by Fig. 10 suffice to decouple one SOR update step into four Jacobi-like update steps for the four colors (We use the letters R, B, G, W to indicate the colors Red, Black, Green and White respectively). We will refer to this algorithm as the 4-color SOR algorithm. Fig. 11 shows the coupling of a red point to its neighboring points, none of which is a red point. It may be similarly verified that no point in Fig. 10 is coupled with a point of the same color. To derive an error detecting version, each sweep in the 4-color SOR algorithm for the modified Laplace equation may then be treated in a manner similar to the red or the black sweep for the Red-Black SOR

algorithm for the basic Laplace equation. We need to introduce variables S_R, S_B, S_G, and S_W for keeping track of the sums of points of every color. In order to derive simple expressions based on the linearity of (30), we need to introduce additional variables to keep track of the sum of various subsets of points within each color. Two points belonging to the same subset have the same color and are involved in the updates of the same number of points of another color for every other color. For example, in Fig. 12 each red point with a box around it belongs to the same subset of red points since each is involved in the updates of three black, three white, and two green points. For each color, we may categorize these subsets into boundary points, corner periphery points, other periphery points and interior points. These categories are analogous to the categories S_{XB}, S_{XC}, S_{XO}, and S_{XI} (X = R or B) for the error-detecting Red-Black SOR algorithm, except that for the 4-color algorithm we now have X = R, B, G, or W. It is then a simple matter to derive the equations for updating each S_X in terms of the various S_{XT} (X = R, B, G, or W and T = B, C, O, or I) based on (30). The derivation proceeds in a manner very similar to the derivations of (3) and (4) and is not detailed here. The values of the various S_{XT}s need to be computed once at the start of the algorithm; subsequently, at the end of each iteration of the algorithm, we need to recompute S_{XB} (may be omitted for a serial implementation since boundary elements do not change), S_{XC} and S_{XO} (X = R, B, G, or W) pertaining to categories of elements on the boundary and periphery of the grid (analogous to the computation of S_{RB}, S_{RC}, S_{RO}, S_B, S_{BB}, S_{BC}, and S_{BO} for the Red-Black case) by summing the new values of the elements in each of these categories. This summing, however, involves only $O(N)$ operations for an $N \times N$ grid, while each iteration of the 4-color SOR algorithm involves $O(N^2)$ operations. Thus the overhead due to these computations diminishes with increasing problem size. Once the sums for various categories of boundary and periphery elements have been computed, the new sums of the interior elements of each color may be computed by subtracting off the new values of the sums of the boundary and periphery elements for each color from the newly computed sum for all elements of that color, in a manner analogous to (5) and (6) for the Red-Black case.

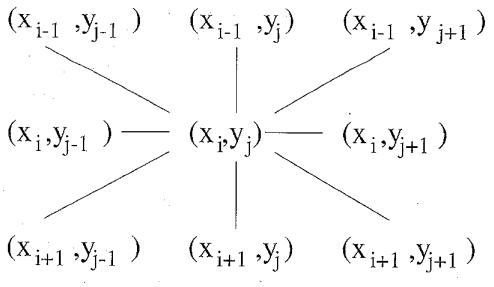


Fig. 9. Coupling of a point to its neighbors in the modified Laplace equation.

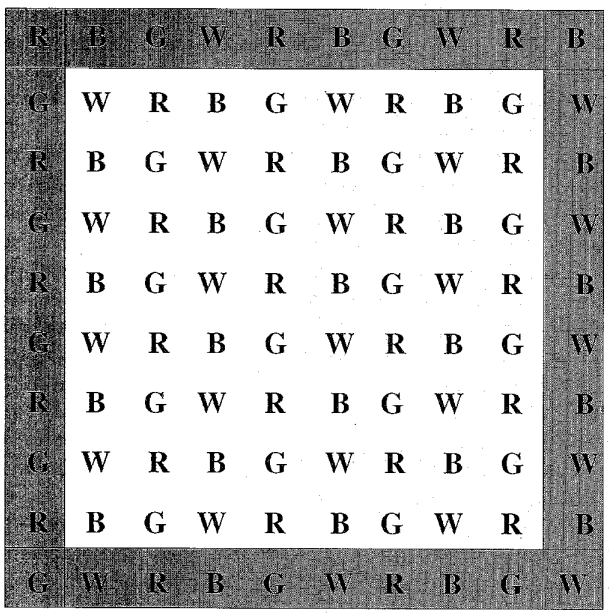


Fig. 10. Coloring scheme for the modified Laplace equation.

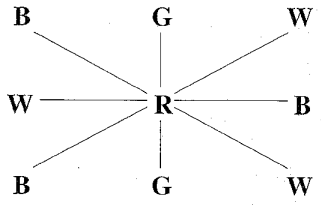


Fig. 11. Coupling of a red point to its neighbors using the coloring scheme of Fig. 10 for the modified Laplace equation.

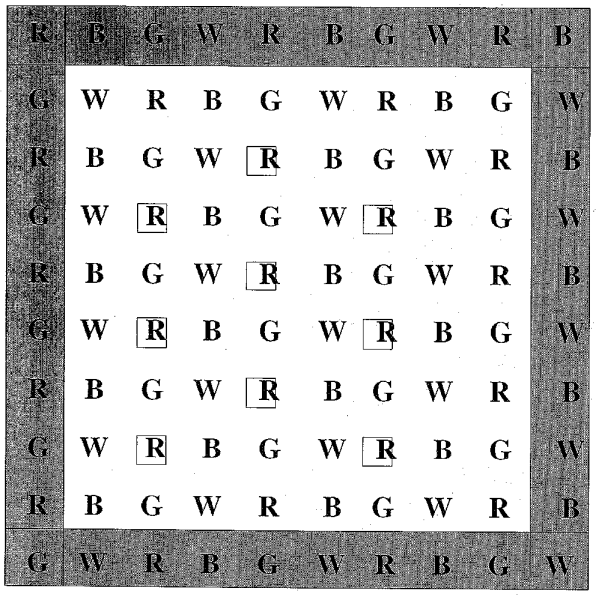


Fig. 12. Red points belonging to the same subset in the 4-color SOR algorithm.

4.1.3 Nonrectangular Solution Domain

Let us consider a problem where we need to solve the Laplace equation over a nonrectangular solution domain such as the one shown in Fig. 13. One may derive an error-detecting Red-Black SOR algorithm by introducing the same categories of points as in Section 2.2, namely S_{XB}, S_{XC}, S_{XO}, and S_{XI}, (X = R or B). As before, each of these categories of points contributes to the update of one, two, three and four points of opposite color, respectively. The points belonging to each of these categories for the nonrectangular domain of Fig. 13 are shown in Fig. 14. As before, variables S_X, (X = R or B), are introduced to keep track of the sums of the red and the black points separately. The update equations for the S_Xs and the S_{XI}s are identical to (3), (4) and (5), (6) in Section 2.2, and we have to update sums of the remaining categories of points by summing all points in each of these categories at the end of each iteration, as before. Thus once we have identified the various categories of points within the irregular solution domain, the remainder of the error-detecting algorithm is identical to the algorithm described in Section 2.2 for obtaining the solution over a rectangular domain.

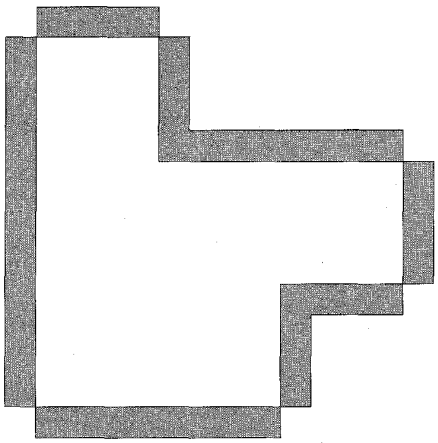


Fig. 13. A nonrectangular solution domain for the Laplace equation.

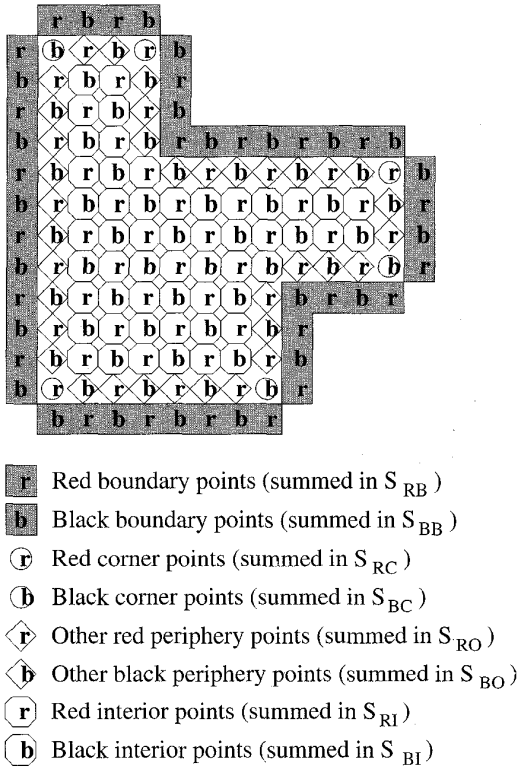


Fig. 14. Various categories of points for the nonrectangular domain of Fig. 13.

5 An Alternative SOR Algorithm With Higher Error Coverage

Although the parallel error-detecting algorithm for solving the Laplace Equation described in Section 3 offers very low overheads over the basic parallel algorithm, the check involving the communicated elements at the end of each iteration of the algorithm is not a very good one. This is because we only perform a range check over the values of the communicated data elements. It is possible that faults can cause some of the communicated data elements to take on erroneous values which do not violate the range check. Such errors would pass undetected and be used in subsequent updates on nonfaulty processors. Also, since communicated data is used in the updates of both grid elements as well as their checksums, undetected errors in communicated data could corrupt both the grid elements as well as the checksums on a nonfaulty processor, which could result in the passing of the final checksum check. A stronger check on the integrity of the communicated data prior to their use in updates would result in higher error coverages. In this section we describe a somewhat different approach than the algorithm of Section 3 in which each communicated data element is subjected to two separate checksum checks prior to its use in the subsequent update. Although this approach is expected to lead to higher error coverages, it also entails higher overheads than the algorithm of Section 3. However, as in the algorithm of Section 3, the overheads for the following approach also diminish with increasing problem size, becoming negligible for large problem sizes.

The algorithm employs the same data distribution as earlier. Each processor is paired with a neighboring processor. Each processor maintains checksums over the red and black points of every row and column owned by its pair processor. At the start of the algorithm, each processor receives its own as well as its pair processor's data. Thus, at the start of the algorithm, all nonfaulty processors compute the correct values for their pair's checksums. The key to the error detecting algorithm is to observe that each data element communicated by a processor at the start of an iteration can be subjected to checksum checks on a different processor prior to its use in updates for that iteration. To describe the new algorithm, we focus on the additional computations performed by a particular processor p on behalf of its pair processor p'. We assume that at some point, processor p' may become faulty, and indicate how its faulty

behavior may be detected on processor p. Assume that the grid points owned by processor p' are denoted by u[i][j], $a \le i < a + q$, $b \le j < b + r$. Let us also assume that u[a][b] is a red point. (The treatment when u[a][b] is a black point is similar.) Let us denote the row checksums over the red points of the ith row owned by processor p' by $RCSR_i^b$ and the row checksums over the black points of the ith row owned by processor p' by $RCSB_i^b$. Analogously, column checksums for the red and black points of column j owned by processor p' are denoted by $CCSR_j^a$ and $CCSB_j^a$ (for notational convenience we assume a, b, q, and r are all even). That is, by definition, we have

$$RCSR_i^b = \sum_{\frac{b}{2} \le j < \frac{b+r}{2}} (u[i][2j] \text{ or } u[i][2j+1]),$$
$$\text{where } (i - a \text{ is even or odd}) \; a \le i < a + q \qquad (31)$$

The equations defining the other checksums are similar. The update equations for the row and column checksums may be derived as before and are as follows

$$RCSR_i^b = (1-\omega)RCSR_i^b + 0.25\omega(RCSB_{i-1}^b + RCSB_{i+1}^b + 2RCSB_i^b + (u[i][b-1] \text{ or } u[i][b+r]) - (u[i][b+r-1] \text{ or } u[i][b])),$$
$$\text{where } (i - a \text{ is even or odd}) \qquad (32)$$

$$RCSB_i^b = (1-\omega)RCSB_i^b + 0.25\omega(RCSR_{i-1}^b + RCSR_{i+1}^b + 2RCSR_i^b + (u[i][b+r] \text{ or } u[i][b-1]) - (u[i][b] \text{ or } u[i][b+r-1])),$$
$$\text{where } (i - a \text{ is even or odd}) \qquad (33)$$

$$CCSR_j^a = (1-\omega)CCSR_j^a + 0.25\omega(CCSB_{j-1}^a + CCSB_{j+1}^a + 2CCSB_j^a + (u[a-1][j] \text{ or } u[a+q][j]) - (u[a+q-1][j] \text{ or } u[a][j])),$$
$$\text{where } (j - b \text{ is even or odd}) \qquad (34)$$

$$CCSB_j^a = (1-\omega)CCSB_j^a + 0.25\omega(CCSR_{j-1}^a + CCSR_{j+1}^a + 2CCSR_j^a + (u[a+q][j] \text{ or } u[a-1][j]) - (u[a][j] \text{ or } u[a+q-1][j])),$$
$$\text{where } (j - b \text{ is even or odd}) \qquad (35)$$

Since we assume that at the start of the algorithm, processor p receives both its own as well processor p''s data, the row and column checksums of processor p' for the next iteration may be computed correctly. However, the computation of the row and column checksums for subsequent iterations is complicated by the fact that processor p requires the row checksums $RCSR_{a-1}^b$, $RSCB_{a-1}^b$, $RSCR_{a+q}^b$, $RCSB_{a+q}^b$, the column checksums $CCSR_{b-1}^a$, $CCSB_{b-1}^a$, $CCSR_{b+r}^a$, $CCSB_{b+r}^a$ and the new values of the grid points u[a − 1][j], u[a][j], u[a + q − 1][j], u[a + q][j], $b \le j < b + q$, u[i][b − 1], u[i][b], u[i][b + q − 1], u[i][b + q], $a \le i < a + q$ for computing the checksum updates using (32) through (35). Some or all of these quantities may not be local to processor p. However, each of these quantities is subjected to a checksum check on its pair processor. Only if all the checks pass are these quantities used in updates involved in the next iteration. For example, the grid elements communicated by processor p' may be checked before they are used in the subsequent iteration since checksums involving these elements are available on processor p (Each communicated grid element of p' is involved in a row or a column checksum check on p). Concurrently, the checksums to be communicated by processor p' to other processors to enable them to perform their checksum updates for the next iteration (following the pair processor approach, each checksum on processor p' corresponds to a row or column of the grid elements computed by processor p) are also communicated to processor p which owns the actual grid elements for which the checksums have been computed. Processor p verifies that the communicated checksums equal the sum over the corresponding grid elements. If the checks pass, we may assume that processor p''s communicated data is free from errors and the checksum update on p yields the correct values of the checksums for the next iteration. If, on the other hand, the data communicated by p' is corrupted, this will cause the checksum check to fail on processor p and cause it to flag an error. Finally, at the end of the algorithm, each processor checks all the elements owned by its pair processor by exchanging data and verifying that the row and column checksums match for every row and column owned by the pair processor (Note that during the course of the algorithm, only the communicated data is checked). The complete check over all the grid elements may also be done after a fixed number of iterations has been completed, in the interests of lower error latency. Note that in the final check, each element is subjected to a row as well as a column checksum check. Thus, a pattern of four compensating errors is required for an error to pass undetected, which is very unlikely in practice. This is illustrated in Fig. 15.

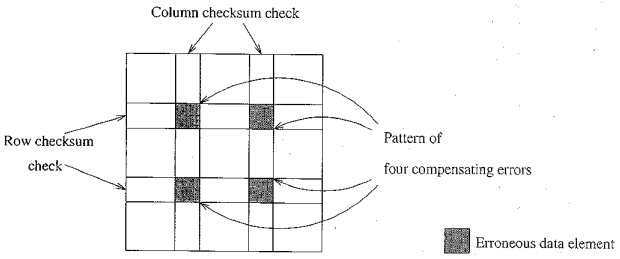


Fig. 15. Pattern of four compensating errors which can cause checks to pass.

The scheme described here also performs only $O(n)$ operations to check the communicated data and compute the checksum updates, while $O(n^2)$ operations are required for the updates of the grid elements. Thus, as in the scheme of Section 3, the overheads for error-detection diminish with increasing problem size. However, actual overheads for this scheme can be expected to be greater because of the larger number of checksum updates involved ($O(n)$ compared to $O(1)$ for the previous algorithm) and the larger number of messages communicated due to the pair processor approach. However, the error coverage of the scheme can be expected to be higher owing to the more rigorous checks communicated elements are subjected to and also due to the fact that each element is subjected to two separate checks, a row and a column checksum check. Additional memory required is $O(n)$ due to the introduction of row and column checksums, compared to only $O(1)$ for the algorithm of Section 3. However, since the memory required

by the original algorithm is $O(n^2)$, memory overheads are not significant for either algorithm.

6 Experimental Results

The effectiveness of an algorithm-based fault tolerance method can be judged by evaluating it according to three criteria. First, the percentage of *false alarms*, which are defined as the percentage of times roundoff errors caused an algorithm-based check to fail, must be low. Second, the error coverage, which may be defined as the percentage of time hardware errors were detected due to a failed check, must be high. Third, the overhead of the fault-tolerant algorithm over the basic algorithm due to extra computations involving the checks and their tolerance expressions, must be as small as possible.

We implemented the both the error-detecting parallel Red-Black SOR algorithms introduced in Sections 3 and 5 for solving the Laplace equation on a 16-processor Intel iPSC/2 hypercube. Meshes of various dimensions were simulated on the hypercube. Section 6.1 gives the false alarm and error coverage results. Section 6.2 gives the timing overhead of the fault-tolerant algorithm over the basic algorithm for various grid and mesh dimensions.

6.1 Error Coverages

In order to determine the false alarm percentage, we ran our error-detecting parallel versions of the two SOR algorithms on around 2,000 data sets in all without injecting hardware errors. Any failed checks on these data sets could then be assumed to be false alarms caused by the magnitude of roundoff errors exceeding the upper bound estimate on roundoff errors provided by the tolerance expression. We varied grid sizes from 32×32 to 256×256, the number of iterations from 100 to 1,000 and mesh dimensions from 1×1 to 4×4. (Recall that our scaling and translating of initial data ensured that the data range being computed on was between 0 and 1 in all cases.) However, not even a single case of false alarm was observed for either algorithm over this set of runs involving data sets with widely different characteristics.

Error coverages were determined by simulation of transient and permanent bit and word level errors over 2,500 runs in all over different data sets. Permanent word (bit) level errors were simulated by corrupting the entire word (flipping a chosen bit) of the result with a probability of 0.5 every time a floating point computation was performed. To simulate transient errors, we performed the data corruption for the result of a floating point computation with only a probability of 0.01. Error coverages are reported as the total number of runs in which errors were detected, divided by the total number of runs in which errors were injected. The grid size and mesh dimensions were varied over the same range as for the false alarm calculation, while the number of iterations were varied from 100 to 500. For the algorithm of Section 3, we were interested in developing a version with as low overhead as possible, and therefore we did not adopt the pair processor approach. (The pair processor approach was adopted instead for the algorithm of Section 5, since in this case we were interested in as high error coverages as possible.) For the algorithm of Section 3, error coverage of word level errors was always 100%, whether the errors were permanent or transient, indicating the effectiveness of our this algorithm for detecting errors of a gross nature. Error coverages of transient and permanent bit level errors were around 65%. However, most of these undetected errors were in the less significant bits. In order to illustrate the effectiveness of this algorithm in detecting serious errors, we computed new error coverages for only those errors which caused a change of 0.1% or more in the element values they were injected in. Subject to this restriction, we found error coverages of bit level errors to be around 98%, showing the effectiveness of our method in detecting errors that resulted in significant data corruption. The results are summarized in Table 1. The column heading "Significant Error Coverage" refers to errors which caused a change of more than 0.1% in the elements they were injected in and were detected by our algorithm. The notion of "Significant Errors" for ABFT techniques was introduced in [12], [13] although the criterion for an error to be significant is somewhat different from the one used in this paper.

TABLE 1
Error Coverage Results for Error-Detecting SOR Algorithm of Section 3

	Error Coverage	Significant Error Coverage
Transient Bit-level	65	98
Transient Word-level	100	100
Permanent Bit-level	69	98
Permanent Word-level	100	100

For the algorithm described in Section 5, error coverages were expectedly higher (Error expressions to determine the tolerance for the checks were calculated in a manner similar to those for the algorithm of Section 3 and are not described in this paper). The error coverage results are indicated in Table 2. We observe that error coverages are extremely high even for the transient and bit level cases, indicating that this method is truly effective in detecting even slight data corruption.

TABLE 2
Error Coverage Results for Error-Detecting SOR Algorithm of Section 5

	Error Coverage	Significant Error Coverage
Transient Bit-level	98	100
Transient Word-level	100	100
Permanent Bit-level	98	100
Permanent Word-level	100	100

6.2 Timing Overheads

Fig. 16 shows the timing overheads of the error-detecting Red-Black SOR algorithm with checks and error bounding over the basic Red-Black SOR algorithm incorporating no error detection for various mesh dimensions and grid sizes. We did not use the pair processor approach for this method (as illustrated by the results in Section 6.2, error coverages were high even though each processor was responsible for checking its own data). We performed a sum over all red and black points and compared with the values stored in the checksum variables once for every 100 iterations. The overheads drop to around 1% as we increase the grid size,

which is only to be expected since we perform only $O(n)$ check operations in each iteration of Red-Black update, compared to $O(n^2)$ operations required for the updates of each red and black point in the grid.

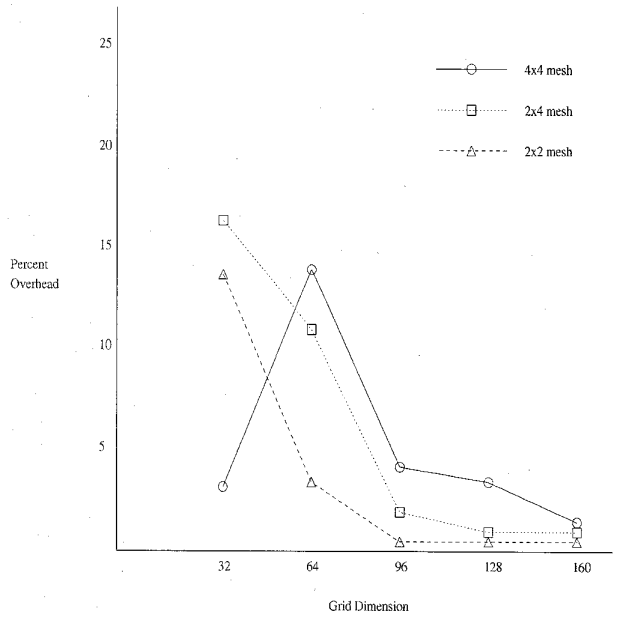


Fig. 16. Timing overhead of the error-detecting algorithm of Section 3 over the basic algorithm.

Fig. 17 depicts the overheads of the error-detecting algorithm of Section 5 over the basic algorithm. The pair processor approach was employed for this algorithm since we were interested in obtaining as high error coverages as possible. The overheads are considerably greater than the previous case, and are contributed to mainly by the fact that message communication more than doubles over the basic algorithm since the pair processor approach is employed, while for our implementation of the previous algorithm, the pair processor approach was not adopted and thus the message count remained the same. However, as expected, the overheads become smaller for larger data sizes, and drop to around 25% for grids of dimensions 256×256.

7 Relation to Prior Work

At this point we would like to mention an earlier work on developing an error-detecting SOR algorithm [15]. The algorithm is based on suitably choosing the initial values on the grid so that at subsequent time steps, solutions at each point are monotonically increasing. The algorithm uses a fine-grained data distribution in which each processor computes on only two adjacent data points and is thus unsuitable for implementation on a general purpose distributed memory parallel computer such as the testbed we ran our experiments on. Also, the monotonic behavior is preserved only for a certain range of the relaxation parameter ω, unlike our algorithm which does not restrict the range of ω in any way in order to achieve fault-tolerance. The SOR

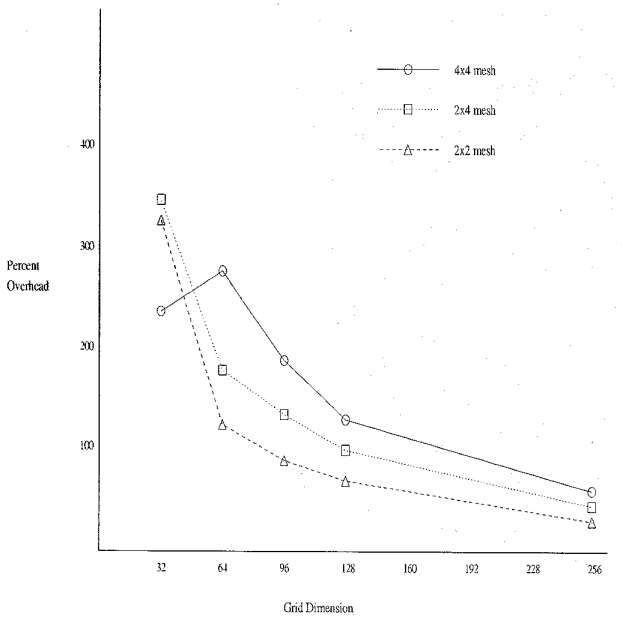


Fig. 17. Timing overhead of the error-detecting algorithm of Section 5 over the basic algorithm.

algorithm in [15] requires individual checking of each element of the grid following each SOR iteration to determine whether the monotonicity property holds, which represents a large and constant overhead. In contrast, both the algorithms introduced in this paper only require checking the boundary elements of the grid subbblock on each processor, which requires only $O(n)$ steps as compared to $O(n^2)$ SOR update operations for each iteration, and a final check step after convergence has been achieved (or the specified number of SOR iterations have been performed), which requires $O(n^2)$ steps. Thus the checking overhead for our algorithms diminishes both with increasing grid size and increasing the number of iterations between checks of all the grid elements (Recall that the communicated data is subjected to checks at every iteration). We would like to mention here that the algorithm in [15] was not evaluated for error coverage or time overhead.

8 Conclusions

In this paper, we have discussed in detail an error-detecting algorithm-based on the Red-Black SOR algorithm for solving the Laplace equation. Minor modifications of the algorithm can be used to derive error-detecting versions of iterative algorithms for solving other PDEs with a similar form such as the Poisson equation. We have discussed extensions to deal with multicoloring and nonrectangular solution domains. We have also discussed an alternative error-detecting approach for the same class of problems which may be used when higher error coverage is desired at the cost of higher overheads. We have presented a new method of dealing with the roundoff accumulation problem which complicates the invariant check step in algorithm-

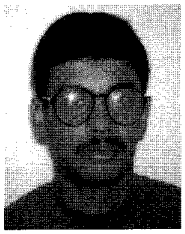
based fault-tolerant encodings based on error analysis incorporating some simplifications in the interest of keeping overheads low. We have shown by our results on a wide variety of data sets that our algorithms possess low overhead and demonstrate high error coverage.

ACKNOWLEDGMENTS

This research was supported in part by the U.S. Office of Naval Research under contract N00014-91-J-1096 and in part by the U.S. Joint Services Electronics Program under contract N00014-90-J-1270.

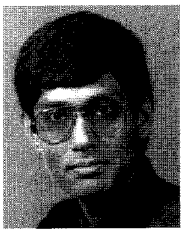
REFERENCES

[1] K.-H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers*, vol. 33, no. 6, pp. 518-528, June 1984.
[2] J.-Y. Jou and J.A. Abraham, "Fault-Tolerant Matrix Operations on Multiple Processor Systems Using Weighted Checksums," *SPIE Proc.*, vol 495, Aug. 1984.
[3] 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. Computers*, vol. 39, no. 9, pp. 1,132-1,145, Sept. 1990.
[4] J.-Y. Jou and J.A. Abraham, "Fault-Tolerant fft Processor," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 548-561, May 1988.
[5] Y.-H. Choi and M. Malek, "A Fault-Tolerant fft Processor," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 617-621, May 1988.
[6] V. Balasubramanian and P. Banerjee, "Tradeoffs in the Design of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors," *IEEE Trans. Software Engineering*, vol. 16, pp. 183-194, Feb. 1990.
[7] V. Balasubramanian, "The Analysis and Synthesis of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors," PhD dissertation, Univ. of Illinois, Urbana-Champaign, Feb. 1991, Technical Report no. CRHC-91-6, UILU-ENG-91-2210.
[8] B.M. McMillin and L.M. Ni, "Reliable Distributed Sorting Through the Application-Oriented Fault Tolerance Paradigm," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, pp. 411-420, July 1992.
[9] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*. New York: Plenum Publishing Corp., 1988.
[10] G. Golub and J.M. Ortega, *Scientific Computing: An Introduction with Parallel Computing*. San Diego: Academic Press, 1993.
[11] F.T. Assad and S. Dutt, "More Robust Tests in Algorithm-Based Fault-Tolerant Matrix Multiplication," *Proc FTCS-22*, pp. 430-439, June 1992.
[12] A. Roy-Chowdhury, "Evaluation of Algorithm-based Fault-Tolerance Techniques on Multiple Fault Classes in the Presence of Finite Precision Arithmetic," MS thesis, Univ. of Illinois, Urbana-Champaign, Aug. 1992, Technical Report no. CRHC-92-15, UILU-ENG-92-2228.
[13] A. Roy-Chowdhury and P. Banerjee, "Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques," *Proc. FTCS-23*, June 1993.
[14] A. Roy-Chowdhury and P. Banerjee, "A New Error Analysis Based Method for Tolerance Computation for Algorithm-Based Checks," *IEEE Trans. Computers*, vol. 45, no. 2, pp. 238-243, Feb. 1996.
[15] K.-H. Huang, "Fault-Tolerant Algorithms for Multiple Processor Systems," PhD dissertation, Univ. of Illinois, Urbana-Champaign, Nov. 1983, Technical Report no. CSG-20.
[16] J.H. Wilkinson, *The Algebraic Eigenvalue Problem*. Oxford: Clarendon Press, 1965.



Amber Roy-Chowdhury (S'90) received the BTech (Hons) degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, in 1990. He received the MS degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1992, where he is currently working toward a PhD. During the summers of 1993 and 1994, he was employed by the Fault-Tolerant Systems Software group at IBM T.J. Watson Research Center, New York. His research interests include fault-tolerance for parallel computers, parallel algorithms, numerical analysis, and parallelizing compilers. He is a student member of the IEEE and the ACM.

Nikolas Bellas (S'93) received his Diploma in computer engineering and informatics from the University of Patras, Greece, in 1992. Since the fall of 1993, he has been a research assistant in the Department of Electrical Engineering at the University of Illinois, Urbana-Champaign, where he received his MS degree in 1995. He is currently pursuing his PhD degree from the same university. His research interests include low power systems, testing and verification of digital circuits, and fault tolerant computing.



Prithviraj Banerjee (F'95) received his BTech degree in electronic and electrical engineering from the Indian University of Technology, Kharagpur, India, in August 1981, and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in December 1982 and December 1984, respectively.

Dr. Banerjee is currently director of the Computational Science and Engineering program and a professor in the Departments of Electrical Engineering and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois, Urbana-Champaign. His research interests are in distributed memory parallel architectures and parallel algorithms for VLSI design automation. He is the author of more than 150 papers in these areas. He is the author of the book *Parallel Algorithms for VLSI CAD* (Prentice Hall, 1994).

Dr. Banerjee was elected a fellow of the IEEE in 1995. He was the recipient of the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981, the IBM Young Faculty Development award in 1986, the National Science Foundation's Presidential Young Investigator award in 1987, IEEE senior membership in 1990, the Senior Xerox Research award in 1992, and the University Scholar award from the University of Illinois for 1992-1993.

Dr. Banerjee was the program chair for the International Conference on Parallel Processing for 1995. He served on the program and organizing committees for the 1988, 1989, 1993, and 1996 Fault Tolerant Computing Symposia, the 1992, 1994, 1995, and 1996 International Parallel Processing Symposia, the 1991, 1992, and 1994 International Symposia on Computer Architecture, and the 1990, 1993, 1994, 1995, and 1996 International Symposia on VLSI Design. He also served as general chairman of the International Workshop on Hardware Fault Tolerance in Multiprocessors, 1989. He is an associate editor of the *Journal of Parallel and Distributed Computing*, and of the *IEEE Transactions on VLSI Systems*. In the past, he served as the editor of the *Journal of Circuits, Systems, and Computers*. He is also a consultant to AT&T, Westinghouse Corporation, the Jet Propulsion Laboratory, General Electric, the Research Triangle Institute, and the United Nations Development Program.