I. INTRODUCTION

Multicomputers have gained much attention recently due to their unique scalability in providing massive computing power. Various hypercube multiprocessors, such as the...
Ncube and Intel iPSC families, are notable examples of commercially available multi-
computers [AtSe88]. Design of various parallel algorithms for multicomputers has been
a major research issue in the application domain. However, the design of reliable parallel
algorithms has received little attention. As the size of multicomputers grows into thou-
sands of processors and the corresponding scale of attempted application problems grows
even faster, ensuring the correctness of a parallel algorithm’s computation is a necessary
step in the greater use of parallel processing.

The application-oriented fault tolerance paradigm [McNi88b] is a promising
approach to providing this necessary fault tolerance for the multicomputer environment.
In this approach, testing is performed at a peer level, i.e., processors involved in a calcu-
lation check other processors involved in different parts of the same calculation. The
actual test is comprised of executable assertions [Stuc77] which are embedded in the pro-
gram code to ensure that at each testable stage of a calculation, all tested processors con-
form to the program’s specification, design, and implementation. If a difference between
this expected and observed behavior is detected, then a fault has occurred and a reliable
communication of this diagnostic information is provided to the system so that appropri-
ate actions may be taken.

Executable assertion development for a program in the multicomputer environment
is complicated by several multicomputer specific issues. In a multicomputer, all proces-
sors (nodes) are autonomous with their own private local memory. The nodes are inter-
connected by an interconnection mechanism. There may or may not be a host processor
for program downloading and data downloading/collection. Message passing is usually
the only means to allow interprocessor communication, which takes non-negligible time.
This limits the scope of the test that may be performed by an executable assertion to test-
ing received messages with respect to a testing node’s local state. Thus, code or design
based assertions may not be possible to implement in such an environment.
Previously, we have developed assertions for the multicomputer environment using the constraint predicate paradigm [McNi88b] for applications of matrix multiplication [HoMc91], matrix iterative solution of simultaneous linear equations [McNi88b], parallel relaxation labeling for the computer vision application arena [McNi88a], and parallel sorting [McNi89]. However, this has been largely a manual task with identification of appropriate assertions based on the three basis metrics of progress, feasibility, and consistency (see section II) of application-oriented fault tolerance. What is needed, however, to make the concept viable, is an automated method for generating executable assertions. This automated generation of assertions is the focus of this paper. In particular, we use a proof outline from program verification of a concurrent program as a starting point and generate our fault-tolerant assertions from this proof outline.

[Mili81] was the first to notice the relationship between program verification and fault tolerance through software specified executable assertions. There also exists a strong relationship between the progress and feasibility metrics of application-oriented fault tolerance and the safety and liveliness properties of concurrent program verification. However, unlike the sequential verification environment of [Mili81], complete state knowledge is not available for generating executable assertions. Thus, the additional metric of consistency, which is unique to application-oriented fault tolerance, must be an integral part of the transformation and resulting constraint predicate. We chose the global auxiliary variable axiomatic approach of [LeGr81] as a system for concurrent program verification and use the transformation outlined in [LuMc91] from global auxiliary variables to the communication sequence-based system of [Soun84] to achieve consistency.

The remainder of the paper is organized as follows. Section 2 discusses the application-oriented fault tolerance paradigm. Sections III and IV briefly review program verification concepts and the axiomatic proof system of [LeGr81] respectively. Section V discusses sorting and bitonic sorting. In Section VI, a verification proof outline of the
bitonic sort is given. Section VII presents the main result of the paper, a transformation from the verification proof outline into a fault-tolerant program. We then review the fault-tolerant algorithm’s error coverage and asymptotic complexity as well as experiences surveyed from our earlier implementation work.

II. BACKGROUND ON APPLICATION-ORIENTED FAULT TOLERANCE

Application-oriented techniques rely on the creating and embedding of assertions into program code. Assertions have been used for program verification [Somm82], software fault tolerance [Rand75] through recovery block as well as in our application-oriented techniques [McNi88a]. Development of these assertions is straightforward for the sequential program execution environment (See [YaCh75, Andr79, Somm82, HuAb87] for examples). These works concentrate on generating from the coding and design process.

Assertions integrated into the program code are called executable assertions. Assertions are embedded as follows:

if not <assertion> then ERROR

If an assertion fails due to some abnormality, then the ERROR condition is raised and the appropriate action is taken. The Application-oriented fault tolerance paradigm is the method which provides fault tolerance by having the executable assertions become part of the operational environment.

Earlier work in selecting executable assertions was guided by “Natural Constraints” of the problem[McNi88]. Working from these natural constraints starting at the specification phase provides a global perspective on assertion development. The parallel decomposition of the problem ties in closely with the design of the program to solve the problem. Thus the executable assertions are closely tied to the individual components of the life cycle.
A set of high-level basis metrics were used to extract assertion generating properties from the problem specification. In a solution, each testable intermediate result should satisfy one or more of the subclasses of progress, feasibility, and consistency. The three subclasses form the class of constraint predicates in which each subclass contributes to form a cohesive unit that provides error coverage. The constraint predicate satisfies both the “liveliness” and “safety” properties of [Hail82]. Liveliness ensures that the solution makes progress - i.e. does not stall or deadlock. Safety ensures that the solution obtained will be the correct solution.

Progress is required to be made at each testable step of the solution. Progress means that the state of the solution advances to the goal or final solution of the problem. Each testable step of the solution is defined as a message interchange, sub-message interchange, or multi-message interchange dependent upon the type of solution. If this progress is not made, then faulty behavior may indefinitely postpone the solution.

Each testable result must remain within the defined solution space of the problem. This is feasibility. The feasibility constraints are often immediately apparent from the nature of the problem studied. Problems in physics and engineering, such as equilibrium problems, eigenvalue problems, and to a lesser extent propagation problems, contain feasibility constraints in the form of boundary conditions.

It is necessary to ensure that any two processors that are correct, given the same input values, executing the same executable assertion, reach the same conclusion of error or no error. However, in a system that has Byzantine faults, it is possible for a faulty processor to send inconsistent messages to different processors. It is necessary for this to be detected. This is the purpose of the consistency condition.

This paper provides a method of translating a verification proof for a parallel bitonic sort algorithm into a fault tolerant algorithm.
III. INTRODUCTION TO PROGRAM VERIFICATION

Mathematics provides a sound, objective way to make accurate, precise predictions about the behavior of programs. The process of using mathematics to make an accurate, precise prediction about the end result of a program and then establishing that the program behaves as specified by the mathematics is called program verification. The approach examined is based on an abstraction called assertional reasoning. This approach uses the notion of a state. A program state associates a value with each variable. Execution of program results in a sequence of atomic actions, each of which transforms the state indivisibly. Assertional reasoning characterizes states by describing the properties of states. This is done by using assertions, which are formulas of predicate logic to characterize sets of states. The use of a programming logic allows programs to be understood as implementing a relation between assertions. This is the basis of the axiomatic approach.

In the axiomatic approach, mathematical logic can be used for making assertions about the program variables before, during and after program execution. These assertions characterize properties of program variables and relationships between them at various stages of program execution. Discussion of program verification using assertional reasoning is found in [Deut73], [Floy67], [HaKi57], [King71], [Mann74]. This paper uses the axiomatic proof system found in [LeGr81], which is used for Hoare’s model of concurrent programming, Concurrent Sequential Processes(CSP) [HoAr78].

Now to introduce the basic concepts of axiomatic semantics.

**Definition 3.1:** An assertion is a logical statement.

**Definition 3.2:** An axiom is a true statement about the properties of a mathematical structure. A mathematical structure is defined by a set of axioms.
**Definition 3.3:** A true assertion which can be inferred or determined from the truth of is a *theorem*.

**Definition 3.4:** A *proof* of a theorem is an argument which establishes that the theorem is true for a specified mathematical structure.

**Definition 3.5:** A *rule of inference* specifies the conclusions which can be drawn from axioms and theorems known to be true. They enable the truth of certain assertions to be deduced from the truth of certain other assertions. Specifically, a rule of inference is of the following form:

\[
\frac{H_1, H_2, \ldots, H_n}{H}
\]

where \( H_1, H_2, \ldots, H_n, H \) are all assertions, has the following interpretation:

“given that \( H_1, H_2, \ldots, H_n \) are true, that \( H \) may be deduced”

Axioms, theorems and conclusions we get using a rule of inference are assertions.

A proof is often presented as a sequence of assertions such that each assertion is either an axiom of the mathematical structure, a previous theorem, or a logical inference using one of the rules of inference from previous steps of the proof. Therefore, in order to prove theorems, we must be able to make assertions about mathematical structures and to determine when one assertion follows from others.

**Definition 3.6:** For a program, the sequence of the assertions corresponding to the proof is called the *proof outline*.

**Definition 3.7:** The process of applying the axioms and inference rules to proving theorems is called *deductive reasoning*.

A programming language is a mathematical structure. In addition to the axioms and rules of inference of the predicate logic, the axioms and the rules of inference for this structure must characterize precisely and correctly the effect of executing programs. This
is in addition to the axioms and rules of inference of the predicate logic.

The axiomatic approach to verification is based on a method that is based on assertions about the program variables before, during, and after program execution.

**Definition 3.8:** A program is *partially correct* with respect to an initial assertion I and a final assertion F if, whenever I is true of the program prior to execution of the program, and the program terminates, then F will be true of the program after the execution of the program is terminated.

**Definition 3.9:** A program is *totally correct* if it is partially correct and it can be shown that this program terminates.

Program verification requires proofs of theorems of the following type:

\[ \{P\}S\{Q\} \]

where P and Q are assertions, and S is a statement of the language. The interpretation of the theorem is as follows: if P is true before the execution of S and if the execution of S terminates, then Q is true after the execution of S. P is said to be the *precondition* and Q the *postcondition*.

**4. A PROOF SYSTEM**

The parallel programs examined are made up of component sequential processes executing in parallel. In general, to prove properties about the program, first properties of each component process are derived in isolation. These properties are combined to obtain the properties of the whole program.

The approach to showing correctness is to divide the correctness proof into two parts. The first part of the correctness proof is the sequential proofs of each individual processes that makes assumptions about the effects of the communication commands.
The second part is to ensure that the assumptions are "legitimate". This will be discussed later. We will first examine the axioms and inference rules used for sequential reasoning. These proof rules are similar to those found in [Hoar68].

In addition to the axioms and inference rules of predicate logic, there is one axiom or inference rule for each type of statement, as well as some statement-independent inference rules. The following axioms and rules of inference apply to reasoning about sequential programs. The basis of the axiomatic approach to sequential programming can be found in [Hoar69].

### 4.1 Axioms

The skip axiom is simple, since execution of the skip statement has no effect on any program or auxiliary variables.

\[
\{P\} \text{skip} \{P\}
\]  

(skip)

The axiom states that anything about the program and logical variables that holds before executing skip also holds after it has terminated.

To understand the assignment axiom, consider a multiple assignment statement, \(\bar{x} := \bar{e}\), where \(\bar{x}\) is a list of \(x_1, x_2, \ldots, x_n\) of identifiers and \(\bar{e}\) is a list of \(e_1, e_2, \ldots, e_n\) of expressions. If execution of this statement does not terminate, then the axiom is valid for any choice of postcondition \(P\). If execution terminates, then its only effect is to change the value denoted by each target \(x_i\) to that of the value denoted by the corresponding expression \(e_i\) before execution was begun. Thus, to be able to conclude that \(P\) is true when the multiple assignment terminates, execution must begin in a state in which the assertion obtained by replacing each occurrence of \(x_i\) in \(P\) by \(e_i\) holds. This means that if \(P_{x_i}^{e_i}\) is true before the multiple assignment is executed and execution terminates, then \(P\) will be true after the assignment. Thus we have the following:
It may seem strange at first that the precondition should be derived from the postcondition rather than vice versa, but it turns out that this assignment rule, as well as being simple, is very convenient to apply in constructing proofs about programs.

4.2 Inference Rules

There are also a number of rules of inference, which enable the truth of certain assertions to be deduced from the truth of certain other assertions.

A proof outline for the composition of two statements can be derived from proofs for each of its components.

\[
\begin{align*}
\{P\} S_1 \{Q\}, \{Q\} S_2 \{R\} \\
\{P\} S_1 ; S_2 \{R\} \tag{composition}
\end{align*}
\]

When executing \(S_1; S_2\), if \(Q\) is true when \(S_1\) terminates it will hold when \(S_2\) starts. From the second hypothesis, if \(Q\) is true just before \(S_2\) executes and \(S_2\) terminates, then \(R\) will hold. Thus if \(S_1\) and \(S_2\) are executed one after the other and \(P\) holds before the execution, then \(R\) holds after the execution.

Execution of an alternate command ensures that a statement \(S_i\) is executed only if its guard \(b_i\) is true. Thus, if an assertion \(P\) is true before execution of the alternate command, then \(P \land b_i\) will hold just before \(S_i\) is executed. The second part of the hypothesis assumes that none of the guards are true. If the hypothesis is true and if the alternate statement terminates, then this is sufficient to prove that \(Q\) will hold should the alternate statement terminate.

\[
\forall i: \{P \land b_i\} c_i; S_i \{Q\}, \{P \land \forall i: \neg b_i\} \rightarrow \{Q\} \\
\{P\} \text{ if } \square b_i; c_i \rightarrow S_i \text{ fi } \{Q\} \tag{alteration}
\]

† This stands for predicate \(P\) with each \(x_i\) replaced with \(e_i\)
The consequence rule allows the precondition of a program or part of a program to be strengthened and the postcondition to be weakened, based on deduction possible in the predicate logic.

\[ P \rightarrow P', \{P'\} S \{Q'\}, Q' \rightarrow Q \]
\[ \{P\} S\{Q\} \]  
(consequence)

Parallel programs are composed of a set of communicating sequential processes. In many programs, it is desirable to save part of the communication sequence between processes. This is done with the use of "dummy" or auxiliary variables. The need for such variables has been independently recognized by many. The first reference that shows the usefulness of auxiliary variables is found in [Clin73]. The auxiliary variables must not affect program control during execution. The following rule allows us to draw conclusions from proof outlines of programs annotated with auxiliary variables.

\[ \{P\} S'\{Q\} \]
\[ \{P\} S\{Q\} \]
(auxiliary variables)

where \( S \) is obtained from \( S' \) by deleting all references to auxiliary variables and \( P \) and \( Q \) do not contain any free variables which are auxiliary variables.

The inference rule for the repetition command is based on a loop invariant i.e. an assertion that holds both before and after every iteration of a loop.

\[ \forall i: \{P \land b_i\} c_i; S_i \{P\} \]
\[ \{P\}*[\Box b_i c_i \rightarrow S_i ] \{P \land \forall i: \neg b_i\} \]  
(repetition)

The hypothesis of the repetition rule requires that if execution of \( S_i \) is begun when the assertion \( P \) and \( b_i \) is true, and if execution terminates, then \( P \) will again be true. Hence, if an assertion \( P \) is true just before the execution a repetition command, then \( P \) is true at the beginning and end of each iteration. Thus, \( P \) will hold if the repetition terminates. The repetition ends when no boolean guard is true, so \( \neg b_1 \land \neg b_2 \land \cdots \neg b_n \) will also
hold at that time.

[LeGr81] does not have distributed termination which is contradictory to Hoare’s original version of CSP [Hoar78]. Distributed termination provides the means for automatic termination of a loop in one process because another process has terminated. It is assumed that all termination of loops occurs when all boolean guards are false.

4.3 Axioms and Inference Rules Dealing With Communication

The communication axiom is as follows:

\[
\{P\} \alpha \{Q\} \quad \text{(communication)}
\]

where \(\alpha\) is a communication command.

Remember that \(\{P\}S\{Q\}\) means total correctness if \(S\) terminates. \(S\) terminates in the absence of deadlock. The parallel rule implies that a proof for it is based on the isolated sequential proofs of the programs it comprises. Take any such program \(S\). A sequential proof for it only proves facts about it running in isolation. With only one process running, communication commands deadlock. Thus, any predicate \(Q\) may be assumed to be true upon termination of a communication command because termination never occurs.

The Law of the Excluded Miracle[Dijk76] states that the statement \{false\} should never be derived. This is the requirement to ensure a sound statement. The communication axiom does violate the Law of the Excluded Miracle. This allows us to deduce that the following is true:

\[
\{\text{true}\} A?x\{x = 5 \land x = 6\}
\]

The postcondition, however, is obviously false. Thus, one might come to the conclusion that the proof system is not sound. This is the result of allowing the communication axiom to make assumptions about the behavior of other processes in order to prove properties of an individual process. In order to justify those assumptions a "satisfaction
proof" must be done. This ensures that the proof system is sound. Hence, the parallel interference rule is as follows:

The parallel inference rule is the following:

\[
(\forall i: \{P_i\} S_i \{Q_i\}) \text{ satisfied and inference - free} \quad \frac{\{\forall i: P_i\} \parallel \{\exists i: S_i\} \{\forall i: Q_i\}}{(\forall i: \{P_i\} S_i \{Q_i\})} \quad \text{(parallel)}
\]

The parallel rule implies that we can construct the proof of a parallel program from the partial correctness properties of the sequential programs it comprises.

It has been mentioned that a "satisfaction proof" is needed to ensure soundness of the proof system. Let us examine the proof outline of the matching communication pair.

\[
P1: [\cdots \{P\}P2?x\{Q\}]
\]

\[
P2: [\cdots \{R\}P1!y\{S\}]
\]

The effect of these two communication commands is to assign \(y\) to \(x\). This implies that \(Q \land S\) is true after communication if and only if

\[
\text{implies that } Q \land S \text{ is true after communication if and only if}
\]

A "satisfaction proof" is such that the above is proven for every matching communication pair. This is called the rule of satisfaction.

Earlier we discussed the need for auxiliary variables. An auxiliary variable may affect neither the flow of control nor the value of any non-auxiliary variables. Otherwise, this unrestricted use of auxiliary variables would destroy the soundness of the proof system. Hence, auxiliary variables are not necessary to the computation, but they are necessary for verification. [LeGr81]'s proof system allows for auxiliary variables to be global i.e. variables that can be shared between distinct processes. Auxiliary variables are used to record part of the history of the communication sequence. Shared reference to auxiliary variables allow for assertions relating the different communication sequences. This necessitates the need for a Proof of Non-interference. This consists of showing that for
each assertion $T$ in process $P_i$, it must be shown that $T$ is invariant over any parallel execution. This is the non-interference property of [Owic75].

V. A BITONIC SORT ALGORITHM and ITS VERIFICATION ASSERTIONS

The target multicomputer interconnection topology considered in this paper is the popular hypercube topology. As mentioned above, these systems can grow to over 1000 processors. In general, the topology of an $n$-dimensional hypercube is a graph $G(P,E)$ with $N = 2^n$ vertices called nodes labeled $P_0, P_1, P_2, \ldots, P_{N-1}$. An edge $e_{ij} \in E$ connects $P_i$ and $P_j$ if the binary representations of $i$ and $j$ differ in exactly 1 bit. If we let this bit position be $k$, then $P_i = P_{j+2^k}$. Thus, in an $n$-dimensional hypercube, each processor connects to $n$ neighboring processors. Connections between the host and nodes are mainly used for program/data downloading and result uploading and are not represented in $G$. The algorithm used in this paper as a parallel sorting algorithm was introduced by Batcher in 1968 [Bats68]. This bitonic sort algorithm was introduced as a parallel sorting algorithm that can take advantage of interconnection topologies such as the perfect shuffle and hypercube. As we will see later, there exists a bitonic sort algorithm that maps directly to a hypercube topology.

Sorting is defined in the following:

**Definition 5.1:** Given an input list $I = (I_i)$, $i=0,...,N-1$ a sorting procedure $S$ finds a permutation $\Pi = (\pi)$ such that:

$$I_{\pi_i} \leq I_{\pi_{i+1}}, i = 0, \ldots, N-2$$

or

$$I_{\pi_i} \geq I_{\pi_{i+1}}, i = 0, \ldots, N-2$$

The general idea of a bitonic sort is to build up longer bitonic sequences which eventually lead to a sorted sequence.

**Definition 5.2:** A bitonic sequence is a sequence of elements $O_0, O_1, \cdots, O_{N-1}$ such that

1. There exists a subscript $i$, $0 \leq i \leq N-1$ such that $O_0 \leq O_1 \leq \cdots \leq O_{N-1}$ and $O_0 \geq O_1 \geq \cdots \geq O_{N-1}$
2. There exists a subscript \( i \), \( 0 \leq i \leq N - 1 \) such that \( O_0 \geq O_1 \geq \cdots \geq O_{N-1} \) and \( O_0 \leq O_1 \leq \cdots \leq O_{N-1} \).

The fundamental operation in a bitonic sort is the compare-exchange operation, either \( \min(x,y) \) or \( \max(x,y) \).

**Lemma 5.1:** [Batc68] Given a bitonic sequence \( I_0 \leq I_1 \leq \cdots \leq I_{N/2-1} \) and \( I_{N/2} \geq I_{N/2+1} \geq \cdots \geq I_{N-1} \), each of the subsequences formed by the compare-exchange steps:

\[
\min (I_0, I_{N/2}), \min (I_1, I_{N/2+1}), \ldots, \min (I_{N/2-1}, I_{N-1})
\]

and

\[
\max (I_0, I_{N/2}), \max (I_1, I_{N/2+1}), \ldots, \max (I_{N/2-1}, I_{N-1})
\]

is bitonic with the property that \( O_i \leq O_j \) for all \( i = 0, 1, \ldots, N/2-1 \) and \( j = N/2, N/2+1, \ldots, N-1 \).

Note that the midpoint of the sequence need not be \( N/2 \).

For this paper, we will assume that \( N = 2^k \) for some \( k \). Therefore the initial midpoint will be \( N/2 \). Since each compare-exchange involves only a comparison between elements whose subscripts differ on only one bit and the number of elements is always \( 2^k \), if we have one element per processor, then the bitonic sort can be easily implemented on a hypercube of dimension \( n = \log_2 N \) [Quin87]. As a notational convenience, we define the following:

**Definition 5.3** The home subcube \( SC_{i,j} \) of dimension \( i \) of a processor \( P_j \) is the subcube of size \( 2^i \) that begins with processor \( P_k \), \( k = j - (j \mod 2^i) \) and includes all processors through \( P_l, l = j - (j \mod 2^i) + 2^i - 1 \). Let \( SC_{i,j}^S \) denote the index \( k \) and \( SC_{i,j}^E \) denote the index \( l \).

The Bitonic sort algorithm, instrumented with the assertions necessary for verification, is shown in Figure 5.1 for each node \( node \). Of particular interest are assertions \( Loop_i \) which asserts that at each execution of the outer iteration, a bitonic sequence in the subcube of dimension \( SC_{i,node} \).
Procedure Bitonic Sort;

\[ \text{if} \, \text{node} = \text{a}; \]

\[ \text{a}_\text{node} = \text{a}; \]

\[ \text{p}_\text{node} = \text{a}; \]

\textbf{for} i:=0 \textbf{to} n-1 \textbf{do}

\[ \text{< Loop i >} \quad \text{(assignment, consequence)} \]

\[ \text{p}_\text{node} = \text{a}; \]

\[ \text{a}^{'}_\text{node} = \text{a}; \]

\{

\textbf{for} j:=i \textbf{downto} 0 \textbf{do}

\[ \text{< Loop j >} \quad \text{(assignment, consequence)} \]

\[ \{d:=2^j; \]

\[ \text{< Loop j } \wedge \text{a}_\text{node} = \text{a}_\text{node} \wedge d = 2^j \} \quad \text{(assignment)} \]

\textbf{if} \ (\text{node mod } 2^i < 2^{i+1})

\[ \text{< Loop j } \wedge \text{a}_\text{node} = \text{a}_\text{node} \wedge d = 2^j \wedge \text{data} = \text{a}_\text{node+d} \} \quad \text{(communication)} \]

\textbf{if} \ (\text{node mod } 2^i < 2^{i+1})

\[ \text{a} = \min (\text{a}_\text{node+d}, \text{a}_\text{node}) \quad \forall l_{j, \text{node}} \leq l \leq l_{j, \text{SC}} \quad [a \leq \max (a_{l+d}, a_{l+d})] \quad \text{(assignment, consequence)} \]

\textbf{else}

\[ \text{< Loop j } \wedge \text{a}_\text{node} = \text{a}_\text{node} \wedge d = 2^j \wedge \text{data} = \text{a}_\text{node+d} \wedge (\text{node mod } 2^{i+2} < 2^{i+1}) \} \quad \text{(alternative)} \]

\[ \{b := \max (\text{data}, \text{a}); a := \min (\text{data}, \text{a}); \}

\text{< Loop j } \wedge d = 2^j \wedge \text{data} = \text{a}_\text{node+d} \wedge \text{node mod } 2^{i+2} < 2^{i+1} \wedge b = \max (\text{a}_\text{node+d}, \text{a}_\text{node}) \wedge \text{a} = \min (\text{a}_\text{node+d}, \text{a}_\text{node}) \wedge \forall l_{j, \text{node}} \leq l \leq l_{j, \text{SC}} \quad [a \leq \min (a_{l+d}, a_{l+d})] \quad \text{(assignment, consequence)} \]

\textbf{write} from b to node+d;

\textbf{else} /* Send to neighbor - we are inactive this iteration */

/* Proof outline is symmetrical */

\{ \textbf{write} from a to node-d; \textbf{read} into a from node-d; \}

\[ \text{a}_\text{node} = \text{a}; \]

\[ a^{'}_\text{node} = \text{a}_\text{node}; \]

\} /* End for j */

\} /* End for i */

Figure 5.1. Bitonic Sort Instrumented with Verification Assertions
Where Loop\(_i\) is the following:

\[
i \neq 0 \implies (\text{node mod } 2^{i+1} < 2^i \rightarrow a_{\text{SC}_{i,\text{node}}}^S \leq \cdots \leq a_{\text{SC}_{i,\text{node}}}^E) \wedge
\]

\[
\text{node mod } 2^{i+1} \geq 2^i \rightarrow a_{\text{SC}_{i,\text{node}}}^S \geq \cdots \geq a_{\text{SC}_{i,\text{node}}}^E
\]

\[
\exists l_{0 \leq l \leq N} a_{\text{node}} = ia_l \wedge \exists l_{S_{\text{SC}_{i,\text{node}}} \leq S_{\text{SC}_{i,\text{node}}}^E} a_{\text{node}} = pa_l
\]

and Loop\(_j\) is the following:

\[
\forall m_{j \leq m \leq j-1} [\text{node mod } 2^{i+2} < 2^{i+1} \rightarrow \forall k_{S_{\text{SC}_{m+1,\text{node}}} \leq S_{\text{SC}_{m+1,\text{node}}}^E} a_k \leq a_{i+1}] \wedge
\]

\[
\text{node mod } 2^{i+2} \geq 2^{i+1} \rightarrow \forall k_{S_{\text{SC}_{m+1,\text{node}}} \leq S_{\text{SC}_{m+1,\text{node}}}^E} a_k \geq a_{i+1})]
\]

\[
i = j \implies (\text{node mod } 2^{i+2} < 2^{i+1} \rightarrow a_{\text{SC}_{i,\text{node}}}^S \leq \cdots \leq a_{\text{SC}_{i,\text{node}}}^E \wedge \text{node mod } 2^{i+2} \geq 2^{i+1} \rightarrow a_{\text{SC}_{i,\text{node}}}^S \geq \cdots \geq a_{\text{SC}_{i,\text{node}}}^E)
\]

\[
\exists l_{0 \leq l \leq N} a_{\text{node}} = ia_l \wedge \exists l_{S_{\text{SC}_{i,\text{node}}} \leq S_{\text{SC}_{i,\text{node}}}^E} a_{\text{node}} = pa_l
\]

and the assertion \{Loop\(_j\)\} is constructed by replacing in the assertion Loop\(_j\) each \(a_i\) with \(a_i^{'j}\).

\(a_{\text{node}}, a_{\text{node}}^{'j}, \text{ and } ia_{\text{node}}\) where \(0 \leq \text{node} \leq N\) are the auxiliary variables used in the proof outline.

It is assumed for the sake of simplicity that we have loop synchronization. This simplifies proofs of non-interference. As part of inner loop synchronization, assign \(a_{\text{node}}\) to \(a_{\text{node}}^{'j}\). This is done before any check for loop invariants. Loop invariants are checked at beginning of loops, but loop indices are changed at end. This implies that the loop indices are changed at the end of a loop iteration before invariants are checked.
This paper will not go into the details of the proof outline. However, it is noted that Loop\textsubscript{i} states that the values of the local variables a in each processor at the end of completion of each iteration of the outer loop produces a bitonic subsequence in each subcube of size $2^{i+1}$. Loop\textsubscript{i} also states that the values of the local variables a in each processor at the end of completion of each iteration of the outer loop is a permutation of the initial values.

The most important aspect of this proof outline is to notice that the critical assertion on the sorted behavior of the elements, Loop\textsubscript{i}, contains global auxiliary variables. This assertion is useless in the unreliable distributed environment since we cannot reliably obtain any $a_{l}$ other than $a_{node}$. In the next section, we present an encoding of the global auxiliary variables into communication sequences that can be communicated reliably.

**VI. TRANSLATING THE PROOF OUTLINE OF THE BITONIC SORT TO FAULT TOLERANT CONSTRAINTS**

The procedure for translating a proof outline to fault-tolerant constraints proceeds as follows:

1. Delete all assertions that use only local variables. This leaves only assertions that use the global auxiliary variables.

2. Choose a subset of the of assertions that use global auxiliary variables. It is this subset that will be embedded into the operational environment as an executable assertion. These assertions will become the progress and feasibility constraints.

3. Communicate the global auxiliary variables. Communication must be done in such a way that it is possible to test for communication.

The auxiliary variables are generally used to record part of the history of communication. Assertions that use only local variables could limit the operational fault-tolerant environment to having each node check only itself or limit the search for faults to only node faults as opposed to both node and communication link faults.

Earlier it was stated that a faulty program will violate one of the intermediate assertions used in the proof that the program satisfies its specification. It is necessary to pick out some of the assertions used in the proof outline that will be embedded into the operational environment as an executable assertion.
The assertion Loop_i states that the values of the local variable a in each processor at the end of completion of each iteration of the outer loop produces a bitonic subsequence in each subcube of size $2^{i+2}$ given bitonic subsequences of length $2^{i+1}$ at the start of step i. Therefore, one constraint predicate could be constructed that checks that at the completion of iteration i of the outer loop that the program produces a bitonic subsequence in each subcube of size $2^{i+2}$.

The assertion Loop_i uses the following auxiliary variables $a_0, a_1, \ldots , a_N$. As noted in Section 5, the auxiliary variable $a_{node}$ denotes the value of the local variable a in the process labeled node. At the completion of each iteration of the outer loop, the auxiliary variable $a_{node}$ is equal to the value of a in process node. None of the processes has the "global picture" of the values of the auxiliary variables. Therefore, it becomes necessary to communicate these values. This is done by "piggybacking" values of the auxiliary variables at the end of stage i, in communication that occurs naturally at stage i+1. Mathematically this can be described as follows:

**Definition 6.1:** Let $(x, y)$ denote a tuple.

**Definition 6.2:** Define $node_{h^i_j}$, where $i > 0$, as follows:

**if** node bold mod $2^{j+1} < 2^j$ **then**

$$node_{h^i_j} = \{(node, a_{node}), (node + 2^i, a_{node+2})\} \quad j = i$$

$$node_{h^i_j} = node_{h^i_{j+1}} \cup node_{h^i_{j+1}+2^i} \quad 0 \leq j < i$$

**and if** node bold mod $2^{j+1} \geq 2^j$ **then**

$$node_{h^i_j} = \{(node, a_{node}), (node - 2^i, a_{node-2})\} \quad j = i$$

$$node_{h^i_j} = node_{h^i_{j+1}} \cup node_{h^i_{j+1}-2^i} \quad 0 \leq j < i$$

Definition 6.2 says that $node_{h^i_j}$ denotes the values of the local variables a computed in stage i-1 and collected by the process labelled node in stage i between stages j to i of the inner loop of the bitonic sort algorithm. It should be noted that the values of the local variables a are assigned to the auxiliary variables denoted by $a_0, a_1, \ldots , a_N$.

We now present a series of results about $node_{h^i_j}$. The first result, presented in Lemma 6.1, states that the processes in the same subcube of size $2^{j+1}(j \neq 0)$ have until stage j of the inner loop, collected values of the local variables a from disjoint processes.
Lemma 6.1: For any process labelled node, where $0 \leq \text{node} \leq N - 1$, the following is true:

$$\text{node}^i_j \cap m^i_j = \emptyset$$

where $0 < j \leq i$, $SC^S_j,\text{node} \leq m \leq SC^E_j,\text{node}$, and $m \neq \text{node}$.

Proof: The case of $(\text{node} \bmod 2^{j+1}) < 2^i$, where $0 \leq j \leq i$ is examined. The case of $(\text{node} \bmod 2^{j+1}) \geq 2^i$ is symmetrical. Proof is by induction on $j$.

If $j = i$, then from Definition 6.2, it is trivially true that:

$$\text{node}^i_i \cap m^i_i = \emptyset$$

where $SC^S_i,\text{node} \leq m \leq SC^E_i,\text{node}$.

Assume that $\text{node}^i_j \cap m^i_j = \emptyset$, where $SC^S_j,\text{node} \leq m \leq SC^E_j,\text{node}$ and $m \neq \text{node}$ is true for $j = l+1$.

Assume that $j = l$. Let us choose $m$ such that $SC^S_l,\text{node} \leq m \leq SC^E_l,\text{node}$. We want to show that for any such chosen $m$ that:

$$\text{node}^i_l \cap m^i_l = \emptyset$$

is true.

We have from Definition 6.2 that

$$\text{node}^i_l = \text{node}^{l+1}_i \cup \text{node}^{2^{l+1}}_l$$

$$m^i_l = m^{l+1}_i \cup m^{2^{l+1}}_l$$

From Definition 5.3, we have that

$$SC^S_{l+1,\text{node}} = \text{node} - (\text{node} \bmod 2^{l+1})$$

$$SC^E_{l+1,\text{node}} = \text{node} - (\text{node} \bmod 2^{l+1}) + 2^{l+1} - 1$$

Since $0 \leq \text{node} \bmod 2^{l+1} \leq \text{node}$, then $0 \leq \text{node} - (\text{node} \bmod 2^{l+1}) \leq \text{node}$. Hence, $\text{node} - (\text{node} \bmod 2^{l+1}) \leq \text{node} + 2^l$. Now, since it is assumed that $\text{node} \bmod 2^{l+1} < 2^i$, then it can be concluded that $\text{node} + 2^l \leq \text{node} - (\text{node} \bmod 2^{l+1}) + 2^{l+1} - 1$. Therefore, it can be concluded that $S^S_{l+1,\text{node}} \leq \text{node} + 2^l \leq S^E_{l+1,\text{node}}$. Similarly results can be proven about $m + 2^l$ and $m + 2^{l+1}$. Therefore, since $\text{node}^{l+1}_i, \text{node}^{2^{l+1}}_i, m^{l+1}_i, m^{2^{l+1}}_i$ do not intersect then it can be concluded that
As a corollary to Lemma 6.1, we have that a process only receives one value from another process.

**Corollary 6.1:** If \( t = (n, a_n) \in \text{node}_h^i \), then \( \neg \exists t' \) such that \( t' = (n, a'_n) \in \text{node}_h^i \) and \( a_n \neq a'_n \).

**Proof:** Proof is by induction on \( j \). If \( j = i \), then it is trivially true that if \( t = (n, a_n) \in \text{node}_h^i \), then \( \neg \exists t' \) such that \( t' = (n, a'_n) \in \text{node}_h^i \) and \( a_n \neq a'_n \).

Assume that if \( t = (n, a_n) \in \text{node}_h^i \), then \( \neg \exists t' \) such that \( t' = (n, a'_n) \in \text{node}_h^i \) is true for \( j = i+1 \). If \( j = 1 \) and \( \text{node} \mod 2^{i+1} < 2^i \) (the case of \( \text{node} \mod 2^{i+1} \geq 2^i \) is symmetrical), then

\[
\text{node}_h^i = \text{node}_{h+1}^i \cup \text{node}^{2^i}_{h+1}
\]

By Lemma 6.1, if \( t \in \text{node}_{h+1}^i \) then either \( t \in \text{node}_h^i \) or \( t \in \text{node}^{2^i}_{h+1} \). From the inductive hypothesis, we may then conclude that \( \neg \exists t' \) such that \( t' = (n, a'_n) \in \text{node}_h^i \). \( \Box \)

We now build a series of results to show the values collected by process \( \text{node} \) in stage \( i \) are from processors in the subcube \( SC_{i+1, \text{node}} \).

**Definition 6.3:** The cardinality of any set \( H \) is denoted by \( |H| \).

**Lemma 6.2:** For any process labelled \( \text{node} \), where \( 0 \leq \text{node} \leq N \), the following is true:

\[
| \text{node}_h^i | = 2^{i-j+1}
\]

**Proof:** Proof is by induction. Let \( j = i \). From Definition 6.2, we have that \( | \text{node}_h^i | = 2 = 2^{i-j+1} \). Assume that \( | \text{node}_h^i | = 2^{i-j+1} \) is true for \( j = i+1 \). For \( j = 1 \) we have that

\[
\text{node}_h^i = \text{node}_{h+1}^i \cup \text{node}^{2^i}_{h+1}
\]

From Lemma 6.1, it is known that \( \text{node}_{h+1}^i \) and \( \text{node}^{2^i}_{h+1} \) are disjoint. Therefore,

\[
| \text{node}_h^i | = 2^{i-1-j+1+1} + 2^{i-1-j+1} = 2^{i-j+1}
\]

**Lemma 6.3:** For any process labelled \( \text{node} \), the following is true:

\[
| \text{node}_h^0 | = 2^{i+1}
\]
Proof: This is a direct result of Lemma 6.2. □

We now define a selection operator on $h^i_j$ that returns a set of node identifiers.

**Definition 6.4:**

$$\Pr(\text{node}^i_j) = \{ n | (n, *) \in \text{node}^i_j \}$$

**Lemma 6.4:** For any process labelled node, the following is true:

$$|\Pr(\text{node}^i_j)| = 2^{i-j+1}$$

Proof: This immediately follows Definition 6.2 and Lemma 6.2. □

**Lemma 6.5:** $\Pr(\text{node}^i_j) \subseteq SC_{i+1, \text{node}}$.

Proof: Proof is by induction on $j$. Assume $j=i$. Without loss of generality, it will also be assumed that node $\text{mod} \ 2^{i+1} < 2^i$ (the case of node $\text{mod} \ 2^{i+1} < 2^i$ is symmetrical). Then

$$\text{node}^i_j = \{(n, a_{\text{node}}), (n + 2^i, a_{\text{node} + 2^i})\}$$

This implies that

$$\Pr(\text{node}^i_j) = \{ \text{node}, \text{node} + 2^i \}$$

Therefore, $\Pr(\text{node}^i_j) \subseteq SC_{i+1, \text{node}}$ is true for $j = i$.

Assume that $\Pr(\text{node}^i_j) \subseteq SC_{i+1, \text{node}}$ is true for $j = l+1$. If $j = l$, then

$$\text{node}^i_j = \text{node}^i_{l+1} \cup \text{node} + 2^i \text{node}^i_{l+1}$$

It is known that $\Pr(\text{node}^i_{l+1}) \subseteq SC_{i+1, \text{node}}$ and $\Pr(\text{node} + 2^i \text{node}^i_{l+1}) \subseteq SC_{i+1, \text{node}}$. Hence, it can be concluded that

$$\Pr(\text{node}^i_j) \subseteq SC_{i+1, \text{node}} □$$

**Theorem 6.1:** $\Pr(\text{node}^i_0) = SC_{i+1, \text{node}}$.

Proof: From Lemma 6.5, it was shown that $\Pr(\text{node}^i_j) \subseteq SC_{i, \text{node}}$. Definition 5.2, implies that $|SC_{i, \text{node}}| = 2^{i+1}$. From Lemma 6.3, it was shown that $|\text{node}^i_0| = 2^{i+1}$. It can then be concluded that $\Pr(\text{node}^i_0) = SC_{i+1, \text{node}}$. □
From Theorem 6.1 and Definition 6.2 it can be concluded that \( h_{i,j} \), denotes the values of the local variables \( a \) computed in stage \( i-1 \) and collected by the process labelled node in stage \( i \) between stages \( j \) to \( i \) of the inner loop of the bitonic sort algorithm. Therefore, \( h_{0} \) contains the values of the local variables \( a \) of the subcube \( SC_{i+1,0} \) that is the result of stage \( i-1 \).

It is entirely possible for a Byzantine faulty processor to send different versions of the same message to different processors. Hence, it is necessary for the fault-tolerant algorithm to ensure that every processor receives the same version of a message. This can be accomplished by sending multiple copies of each element via vertex disjoint paths to a "checking" processor, which determines if the incoming message from these two paths is consistent. Mathematically, this can be described as follows:

**Definition 6.5:** Define \( c_{i,j} \), where \( i > 0 \), as follows:

If \( i \) is odd,

\[
\begin{align*}
    c_{i,j} &= \{(n, a_n), (n + 2^j, a_{n+2^j}), (n + 1, a_{n+1}), (n + 2^{j+1}, a_{n+2^{j+1}})\} \quad 0 \leq j < i
\end{align*}
\]

and if \( i \) is even,

\[
\begin{align*}
    c_{i,j} &= \{(n, a_n), (n - 2^j, a_{n-2^j}), (n - 1, a_{n-1}), (n + 2^{j-1}, a_{n+2^{j-1}})\} \quad 0 \leq j < i
\end{align*}
\]

This definition implies that each element considered for inclusion in \( c_{0} \) is reported through at least two vertex disjoint paths.

**Lemma 6.6:** \( \Pr(c_{i,j}) \subseteq SC_{i+1,0} \), node.

*Proof:* Proof is similar to that of Lemma 6.5. \( \Box \).

To achieve consistency, each processor "hears" the same version. This is mathematically defined as follows:

**Definition 6.6:** If there exists \( t_1, t_2 \in c_{i,j} \) and

\[
t_1 = (m_1, a_{m_1})
\]

...
where \( m_1 = m_2 \), but \( a_{m_1} \neq a_{m_2} \), then the system is said to be **inconsistent** else the system is **consistent**.

**Theorem 6.2:** If the system is consistent and for each process labelled node, where \( 0 \leq \text{node} \leq N - 1 \), and \( i > 0 \), the following is true:

\[
\text{node}h_0^i = \text{node}ch_0^i
\]

**Proof:** Assume that \( \text{node}h_0^i \neq \text{node}ch_0^i \). Assume the case of node \( \text{mod} 2^1 < 2^0 \) (the case of node \( \text{mod} 2^1 \geq 2^0 \) is symmetrical). This implies that

\[
\text{node}h_0^i = \text{node}h_1^i \cup \text{node}+2^i h_1^i
\]

Lemma 6.6 and Theorem 6.1 imply the following:

\[
\Pr(\text{node}ch_0^i) \subseteq \text{SC}_{i+1, \text{node}} = \Pr(\text{node}h_0^i)
\]

Then from Corollary 6.1, we can conclude that there exists \( t_1 = (m, a_{m_1}), t_2 = (m, a_{m_2}) \) \( \in \text{node}ch_0^i \) such that \( a_{m_1} \neq a_{m_2} \), but \( m_1 = m_2 \). Yet, this contradicts the consistency assumption. Therefore, we have that \( \text{node}h_0^i = \text{node}ch_0^i \). \( \square \)

**Corollary 6.2:** If the system is inconsistent and for some process labelled node, where \( 0 \leq \text{node} \leq N - 1 \), the following is true:

\[
\text{node}h_0^i \neq \text{node}ch_0^i
\]

**Proof:** Proof is symmetrical to that of Theorem 6.2. \( \square \)

Theorem 6.2 says that each auxiliary element may be collected from two different sets. If the two sets contain a different value for the same auxiliary variable, then there is an error.

Once a processor knows that it has a consistent view of the auxiliary variables, then it is possible to test whether the auxiliary variables satisfy the assertion \( \text{Loop}_i \).

As discussed before the assertion \( \text{Loop}_i \) states that the values of the local variables \( a \) in each processor at the end of completion of each iteration of the outer loop produces a bitonic subsequence in each subcube of size \( 2^{i+1} \). From Theorem 6.1, we have that \( \text{node}h_0^i \)
is the set of values of of the local variables a in the subcube $SC_{i+2, node}$ computed in stage i-1. The progress test then consists of each processor checking that the values associated with $node_{h_0}^i$ forms a bitonic sequence.

**Theorem 6.4:** If $t_1, t_2 \in node_{ch}^i$ where $t_1 = (n, a_n)$ and $t_2 = (n + 1, a_{n+1})$ then if node $mod 2^{i+1} < node mod 2^i$ and $a_n > a_{n+1}$ or if node $mod 2^{i+1} =< node mod 2^i$ and $a_n < a_{n+1}$ then Loop$_i$ is false.

**Proof:** Part of Loop$_i$ is the following:

$$\begin{align*}
i \neq 0 & \Rightarrow (node\ mod\ 2^{i+1} < 2^i \rightarrow a_{SC_{i, node}}^S \leq \cdots \leq a_{SC_{i, node}}^E) \\
node\ mod\ 2^{i+1} \geq 2^i & \Rightarrow a_{SC_{i, node}}^S \geq \cdots \geq a_{SC_{i, node}}^E)
\end{align*}$$

The case of node $mod 2^{i+1} < 2^i$ is examined. The case of node $mod 2^{i+1} \geq 2^i$ is similar. From Theorem 6.1, we have that $n, n + 1 \in SC_{i+1, node}$. Therefore, it can be immediately concluded that if $a_n > a_{n+1}$ then Loop$_i$ is false. □

Loop$_i$ also states that the values of the local variable a in each process at the end of each iteration of the outer loop is a permutation of the original values of the local variables a in each process, which are represented by the auxiliary variables $ia_1, ia_2, \ldots, ia_n$. It is not necessary to explicitly communicate these values to all processes.

**Definition 6.7:** Define $node_{pch}^i$, where $i > 0$, as follows:

if $node \ mod\ 2^{i+1} < 2^i$ then

$$\begin{align*}
node_{pch}^0 & = \{(node, a_{node}), (node + 1, a_{node+1})\} i = 1 \\
node_{pch}^i & = node_{ch}^{i-1} i > 1
\end{align*}$$

and if $node \ mod\ 2^{i+1} \geq 2^i$ then

$$\begin{align*}
node_{pch}^0 & = \{(node, a_{node}), (node - 1, a_{node-1})\} i = 1 \\
node_{pch}^i & = node_{ch}^{i-1} i > 1
\end{align*}$$

From Theorem 6.1, Definition 6.2 and Definition 6.7, it can be concluded that $node_{pch}^i$, contains the values of the local variables a of the subcube $SC_{i, node}$ that is the result of stage i-2.
**Definition 6.8**

\[ A(\text{node pch}_0) = \{a_n|(n, a_n)\} \]

**Theorem 6.5:** If \[ \exists l_{0 \leq l \leq N} a_{\text{node}} = ia_l \land \exists l_{s_{\text{node}} \leq l \leq e_{\text{node}}} a_{\text{node}} = pa_l \] is true for stage i, where \( i > 0 \), then the following is also true:

\[ A(\text{node pch}_{i+1}) \subseteq A(\text{node ch}_{i+1}) \]

**Proof:** By Definition 6.7, we have that \( \text{node pch}_{i+1} = \text{node ch}_0 \), contains the values of the local variables \( a \) of the subcube \( SC_{i+1, \text{node}} \) that is the result of stage \( i-1 \). By definition, these values are denoted by the auxiliary variables \( pa \), where \( SC_{s_{\text{node}}} \leq l \leq SC_{e_{\text{node}}} \). It follows immediately from \( \exists l_{0 \leq l \leq N} a_{\text{node}} = ia_l \land \exists l_{s_{\text{node}} \leq l \leq e_{\text{node}}} a_{\text{node}} = pa_l \) being true for stage i that

\[ A(\text{node pch}_{i+1}) \subseteq A(\text{node ch}_0) \]

is also true. \( \square \)

**Corollary 6.3:** If

\[ \neg[A(\text{node pch}_{i+1}) \subseteq A(\text{node ch}_{i+1})] \]

is true, then Loop \( i \) is false.

**Proof:** Trivially true as a result of Theorem 6.5. \( \square \)

**VII. FAULT-TOLERANT ALGORITHM AND ERROR COVERAGE**

Data that violates the assertion Loop \( i \) is considered an error.

From Theorem 6.4, we can derive the progress test, \( \Phi_p \), for stage \( i \) as follows:

If \( t_1, t_2 \in \text{node ch}_j \) where \( t_1 = <n, a_n> \) and \( t_2 = <n+1, a_{n+1}> \) then

if node \( \mod 2^{i+1} < \) node \( \mod 2^i \) and \( a_n > a_{n+1} \) or

if node \( \mod 2^{i+1} \geq \) node \( \mod 2^i \) and \( a_n < a_{n+1} \) then

ERROR.

From Theorem 6.5, we can derive the feasibility test, \( \Phi_F \), for stage \( i \) as follows:
If $\neg[A(\text{node \, pch}_0^{i-1}) \subseteq A(\text{node \, ch}_0^i)]$ then ERROR.

From Theorem 6.2, we can derive the consistency test, $\Phi_C$, for stage i as follows:

If there exists $t_1, t_2 \in \text{node \, ch}_j^i$ $t_1 = <m_1, a_{m_1}>$ and $t_2 = <m_2, a_{m_2}>$ where $m_1 = m_2$, but $a_{m_1} \neq a_{m_2}$ then ERROR.

The following is the program annotated with $\Phi_P, \Phi_F, \Phi_C$. This indicates where the executable assertions are to be embedded.
/*
* $S_{FT}$, executed by node node, $0 \leq \text{node} \leq N - 1$; Local starting value in a
*/

Procedure $S_{NR}$
for $i := 0$ to $n-1$ do
  for $j := i$ downto $0$ do
    { d := $2^j$; 
      if (node mod (2d)<d)
        {read into data from node + d;}
      if (node mod $2^{i+2} < 2^{i+1}$)
        {b := max(data,a);a := min(data,a);}
      else
        {b := min(data,a);a := max(data,a);}
      write from b to node+d;}
    else /* Send to neighbor - we are inactive this iteration */
      {write from a to node-d;read into a from node-d;}
  } /* End for j */
/* End for i */

Figure 7.1. Fault Tolerant Algorithm

It should be noted that for the last stage, an extra communication must take place in order to transmit the values of the auxiliary variables computed in the last stage of the outer loop. The consistency, progress and feasibility checks are done for this extra communication.

Lemma 7.1: The progress test for stage $i$ detects a non-bitonic sequence in a subcube of size $2^{i+2}$. The feasibility test detects whether the values of the local variable $a$ in each subcube of size $2^{i+2}$ of stage $i$ is a permutation of values of the local variable $a$ in subcubes of size $2^{i+1}$ of stage $i-1$. 
Proof: Trivial from Theorems 6.4 and 6.5. □.

Lemma 7.2: At the end of stage i, at least one processor in SC_{i-1,j} can detect an error made by processor P_j that results in either (1) a non-bitonic subsequence in the subcube containing j of size 2^i or (2) if the the maximum number of faulty nodes in SC_{i-1,j} is 1, the ability to determine if the values of the local variables a computed in stage i-2 is a subset of the values of the local variables a in computed in stage i-1. It is assumed that the values of the local variables a computed in stage i-2 are error free.

Proof: The proof is by induction on i. For i=0, each P_node, P_{node+1} for node even contains the actual correct initial values a_node, a_{node+1}. This forms a complete bitonic sequence of length 2. This then becomes node^{node}pch_{0}.

Assume that the values of the local variables a computed in stage k-1 has been verified to be correct with respect to the values of the local variables a computed in stage k-2. By Definition 6.7, the values of the local variables a computed in stage k-1 are denoted by pch_{k}^{k}. From Definition 6.5, ch_{k+1}^{k+1} denotes the values of local variables a computed in stage k. Since each element considered for inclusion in ch_{k+1}^{k+1} is reported through two vertex disjoint paths, the effects of a single faulty relay are limited to one of these paths. If the two candidate elements differ, an error is signaled. Thus if the sender is faulty, it must send identical values along both paths. If the consistency condition is met, then by Lemma 7.1, an error will be flagged if there is a non-bitonic sequence in the subcube containing j of size 2^i or if the values of the local variables a computed in stage i-2 is not a subset of the values of the local variables computed in stage i-2. □

Theorem 7-1: Algorithm S_{FT} produces either a correct bitonic sort or stops with an error in the presence of at most n-1 faulty nodes.

Proof: By application of Lemma 7.2, at each step i of the for loop in S_{FT}, each bitonic sequence is verified. The final extra stage verifies that last sequence. Since we are allowed 1 faulty node per SC_{i,j}, a processor P_j can detect any faulty behavior. □
VIII. Summary

The parallel bitonic sort algorithm introduced in section 5, $S_{NR}$ has a time and communication complexity of $O(\log_2^2 N)$.

Implementation of the sets $\text{node}ch_i^j$ and $\text{node}ch_i^j$ using one dimensional arrays, where the $i^{th}$ element is the value of the local variable $a$ in stages i-1 and i-2, respectively, results in a fault tolerant algorithm of time complexity of $O(N)$ and communication complexity of $O(\log_2^2 N + N \log_2 N)$[McNi89b]. As expected, there is a performance penalty to pay for the increased reliability. However, it has been shown that this extra cost does result in a more efficient algorithm.

This paper has shown how to translate a verification proof for a parallel bitonic sort algorithm into a fault tolerant algorithm. It is easy to see that the progress and feasibility constraints of [McNi88b] is directly derivable from the assertions of the verification proof, while the consistency constraint is derived from the need to ensure that the each processor has a consistent view of the auxiliary variables used in the assertions of the verification proof.

The translation required the choice of an assertion from the verification proof. One major factor in the choice of the assertion was the granularity. Since processors may only communicate via messages, assertions requiring a great deal of extra communication in order to communicate the values of the variables of that assertion is undesirable. In other words, it is more desirable to choose assertions, in which the communication of values of the variables of that assertion do not require extra communication complexity. The assertions chosen will subsume the assertions not chosen for executable assertions. In other words, if the intermediate results violate assertions not chosen for executable assertions, then the intermediate results will violate those assertions that were chosen for use as executable assertions. On the other hand, it is desirable to check as often as possible in order to decrease the amount of time it takes to detect faults. Further research will examine this problem.
REFERENCES


