01 Jan 1989

Reliable Distributed Sorting Through the Application-oriented Fault Tolerance Paradigm

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

L. M. Ni

Recommended Citation

B. M. McMillin and L. M. Ni, "Reliable Distributed Sorting Through the Application-oriented Fault Tolerance Paradigm," *Proceedings of the 9th International Conference on Distributed Computing Systems, 1989*, Institute of Electrical and Electronics Engineers (IEEE), Jan 1989.
The definitive version is available at https://doi.org/10.1109/ICDCS.1989.37983

RELIABLE DISTRIBUTED SORTING THROUGH THE APPLICATION-ORIENTED FAULT TOLERANCE PARADIGM†

Bruce M. McMillin
Computer Science Department
University of Missouri-Rolla
Rolla, MO 65401-0249
ff@cs.umr.edu

Lionel M. Ni
Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027
ni@cps.msu.edu

ABSTRACT: Reliability is an important concern in large scale applications such as distributed sorting. An error made due to a hardware or software fault during the application execution can corrupt, perhaps undetectably, the final result of the calculation. Rather than implement the necessary hardware system level reliability by such traditional (and expensive) methods as replication and masking, we appeal to *Application-Oriented Fault Tolerance*. The Application-Oriented Fault Tolerance Paradigm relies on properties of expected algorithm behavior to form a test for faulty behavior thus also detecting software failures. The test is performed on the actual application's intermediate values by the peers of a particular tested processor. This paper describes the design and implementation of a reliable version of the distributed bitonic sorting algorithm using this paradigm on a commercial multicomputer.

1. INTRODUCTION

Multicomputers have gained much attention recently due to their unique scalability in providing massive computing power. Various hypercube multiprocessors, such as the Ncube, the iPSC 2, and the Symult 2010, are notable examples of commercially available multicomputers [1]. 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 thousands 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* [7] 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 calculation check other processors involved in different parts of the same calculation. The actual test is comprised of executable assertions [9] which are embedded in the program code to ensure that at each testable stage of a calculation, all tested processors conform 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 appropriate 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 processors (nodes) are autonomous with their own private local memory. The nodes are interconnected 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 testing 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.

To overcome the problem of assertion development for the multicomputer environment, the authors proposed the *constraint predicate* paradigm [7]. In the constraint predicate paradigm, assertion development begins with the specification phase of the software life cycle. Three basis metrics of *progress, feasibility*, and *consistency* are used to generate assertions which 1) preserve the global nature of the problem, and 2) can be implemented locally in each node. The collection of such assertions forms the constraint predicate.

It is appropriate to contrast our constraint predicate view of application-oriented fault tolerance with the algorithm-based fault tolerance of Abraham et. al. (for example, [4]) in which checksum encodings are embedded in the calculation for fault detection purposes. Algorithm-based fault tolerance imposes an additional structure on the problem which, while detecting hardware failures, may not be applicable to detecting software failures. The constraint predicate, by contrast, ensures that the program conforms to its specifications such that both hardware and software failures are detected.

The constraint predicate paradigm has been successfully applied to such diverse problems as parallel matrix iterative solution of simultaneous linear equations [7] and parallel relaxation labeling for the computer vision application arena [6]. However, while diverse, both of these problems have their underlying basis in relaxation techniques which contain the necessary local information with which to implement a successful constraint predicate. This paper treats a completely different type of problem from these first two, the problem of parallel sorting in the multicomputer environment.

Sorting in the multicomputer environment is not likely to be the sole application but rather a sub-problem of some other parallel application. This distinction is important for two reasons. First, since the data is already in the node processors of the multicomputer, there is no central point through which all data will pass or had passed. Since no such central point exists, there is no opportunity for centralized fault diagnosis, therefore, the fault-detection problem must be solved in a distributed manner. A second, more pragmatic, consideration is that if the data must be loaded from the external environment, it may be more efficient to simply sort the

† This work was supported in part by the DARPA ACMP project, in part by the GTE Graduate Research Fellowship program, and in part by the AMOCO faculty development program.

data sequentially in the host rather than incur the communication cost necessary to distribute/collect the data to/from the nodes.

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 $P_0,P_1,P_2,...,P_{N-1}$. An edge $e_{i,j}\in E$ 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 $P_i=P_{j\oplus 2^k}$. 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 goal of this research is not to propose a new parallel algorithm for sorting, there are a plethora of these. Rather, the algorithm developed in this paper is a reliable version of a known bitonic sort. It is reliable in the sense that it tolerates a certain degree of hardware failures, never producing an incorrect result in the presence of these failures.

The remainder of the paper is organized as follows. Section 2 discusses sorting assertions in general and introduces the bitonic sort algorithm. Section 3 discusses faulty behavior and presents a fault-tolerant parallel bitonic sort developed using this paradigm. In Section 4, the error coverage and the response of the fault tolerant algorithm to faulty behavior is presented. Section 5 presents both asymptotic complexity and the results of run-time experimental measurements on an Ncube multicomputer [3].

2. SORTING AND BITONIC SORTING

Sorting is defined in the following:

Definition 1: Given an input list $\mathbf{I}=(I_i),i=0,\cdots,N-1$ a sorting procedure S finds a permutation $\Pi=(\pi_i)$ such that:

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

or

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

Let the output list delivered by S be $\mathbf{O}=(O_i)=I_{\pi_i}$ and assume that an ascending sort is performed. The following theorem immediately follows.

Theorem 1:

If there exists no j such that $O_i=I_j$ for each i or there exists no i such that $I_i=O_j$ for each j

or $O_j>O_{j+1}$ for some j,

then the result $\mathbf{O}$ produced by S is incorrect.

Proof: For part (1) if either there is no j such that $O_i=I_j$ for some i or there is no i such that $I_i=O_j$ for some j, then the output list $\mathbf{O}$ is not a permutation of $\mathbf{I}$ thus violating Definition 1. For part (2), the output list must appear in a non-decreasing manner as an ascending sort it being performed. □

The assertion suggested by Theorem 1 can be used in the sequential environment to verify the output of sorting procedure S. However, in the parallel environment, in general, neither the input list $\mathbf{I}$ nor the output list $\mathbf{O}$ is available to a processor implementing the assertion. Furthermore, the assertion (in general) is only applicable at the termination of the sorting procedure. There is no opportunity to flag an error that occurs earlier then the termination phase in S. These difficulties with general sorting foster consideration of a particular parallel sorting algorithm, the *Bitonic Sort* [2].

The bitonic sort algorithm was introduced as a parallel sorting algorithm that can take advantage of interconnection topologies such as the perfect shuffle and hypercube. For our purposes the algorithm has properties that make it amenable to assertion development in the parallel environment. Furthermore, there exists a bitonic sort algorithm that maps directly to a hypercube topology.

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

Definition 2: A Bitonic Sequence is a sequence of elements $O_0,O_1,\cdots,O_{N-1}$ such that

There exists a subscript i, $0\le i\le N-1$ such that $O_0\le O_1\le\cdots\le O_i$ and $O_{i+1}\ge O_{i+2}\ge\cdots\ge O_{N-1}$

or There exists a subscript i, $0\le i\le N-1$ such that $O_0\ge O_1\ge\cdots\ge O_i$ and $O_{i+1}\le O_{i+2}\le\cdots\le O_{N-1}$

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

Lemma 1: [2] Given a bitonic sequence $I_0\le I_1\le\cdots\le I_{N/2-1}$ and $I_{N/2}\ge I_{N/2+1}\ge\cdots\ge I_{N-1}$, each of the subsequences formed by the compare-exchange steps:

$$\min(I_0,I_{N/2}),\min(I_1,I_{N/2+1}),\cdots\min(I_{N/2-1},I_{N-1})$$
$$=O_0,O_1,\cdots,O_{N/2-1}$$

and

$$\max(I_0,I_{N/2}),\max(I_1,I_{N/2+1}),\cdots\max(I_{N/2-1},I_{N-1})$$
$$=O_{N/2},O_{N/2+1},\cdots,O_{N-1}$$

is bitonic with the property that $O_i\le O_j$ for all $i=0,1,\cdots,N/2-1$ and $j=N/2,N/2+1,\cdots,N-1$.

Note that the midpoint of the sequence need not be $N/2$. However, assume that the midpoint is $N/2$ and $N=2^k$ for some k. Pictorially, this splitting and merging is shown in Figure 1. It is easy to see that a bitonic sequence of length 2^k can be sorted by recursively applying the compare-exchange operation k times.

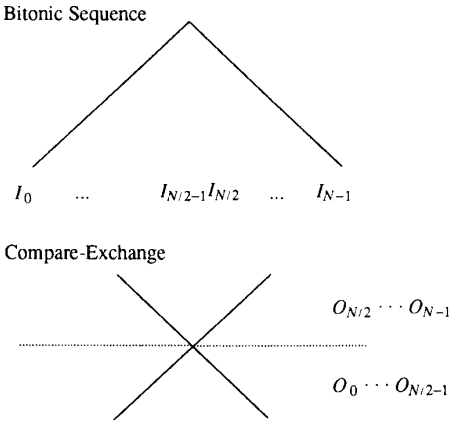

Figure 1. Compare-Exchange Step

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 $n=\log_2 N$ [8]. The algorithm is shown in Figure 2 as Algorithm S_{NR}.

```
/*
* S_NR, executed by node node, 0≤node≤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 2. Algorithm S_{NR}

The completion of each iteration of the **for** i is called a *stage*.

Lemma 2: Algorithm S_{NR} produces a bitonic subsequence in each subcube of size 2^{i+2} at the end of stage i given bitonic subsequences of length 2^{i+1} at the start of step i if no errors occur.

Proof: By induction on i, the subcube index under consideration. Initially we are given degenerate bitonic sequences of length 2. For i=0, since $d=2^0=1$, by Lemma 1 a single parallel compare-exchange step forms two bitonic sequences of length 1 in each subcube of dimension 2, either ascending or descending based on the subcube indices. This forms bitonic sequences of length $2^{i+2}=4$ in each 2-cube.

Assume that for $i=k$, algorithm S_{NR} has produced a bitonic sequence in each subcube of size 2^{k+2} at the end of step k. Then for $i=k+1$, since $d=2^{k+1}$, a single compare-exchange step, by Lemma 1, forms two bitonic sequences each of length 2^{k+1}, one high and one low. Since the next iteration of j is a bitonic sort on these two subcubes of dimension 2^{k+1}, by the inductive hypothesis at the termination of step $k+1$, the subcube of size 2^{k+3} contains a bitonic sequence. □

Theorem 2: Algorithm S_{NR} sorts a list of items arranged in a cube of size $N=2^n$.

Proof: Immediate from Lemma 2 by iteration of the **for** i loop from 0 up to $n-1$ in algorithm S_{NR}. □

Since the code executed by the inner **for** j loop is clearly $O(1)$, the runtime of this parallel algorithm is $O(n^2)=O(\log_2^2 N)$.

3. FAULT-TOLERANT BITONIC SORTING

Our goal in development of this algorithm is to implement a system which can sort in parallel and never produce an incorrect output. To perform this task, the notion of a fault must be clearly defined.

Definition 3: A *failure* is a deviation from the correct sequence of a calculation that produces an incorrect result. A system is said to be *fail-stop* if, upon the occurrence of a fault, no output is produced and the system simply halts. A component is said to exhibit *Byzantine* [5] behavior if, upon the occurrence of a failure, the effect of the fault on the overall calculation is to produce an incorrect result. Furthermore, this will be done in the most malicious manner possible. A node P_i is said to be *faulty* under the following cases:

1) A failure occurs in processor P_i and any number of its incident communication links.
2) A failure occurs in one or more of P_i's incident communication links but not in P_i. There are two cases.
   a) The failure occurs in exactly one incident communication link $e_{i,j}$. If a failure has also occurred in P_j, then P_j is faulty. If no failure has occurred in P_j, then arbitarily P_i is declared faulty and P_j is declared non-faulty.
   b) The failure occurs in two or more incident communication links. Then P_i is declared to be faulty.

The following assumptions are made in the development of the parallel sorting algorithms.

Environmental Assumptions:

1) Inter-node communications and processors are subject to Byzantine faults.
2) The host processor is reliable as are the links from the host processor to each node.
3) Message transmission is over point-to-point links and no atomic broadcast exists.
4) The absence of a message can be detected and constitutes an error.
5) All nodes are non-faulty at initiation of the algorithm and remain non-faulty through the first message exchange.

Definition 4: 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 $SC^S_{i,j}$ denote the index k and $SC^E_{i,j}$ denote the index l. If the bitonic sequence in $SC_{i,j}$ is comprised of an ascending sequence followed by a descending sequence each of length 2^{i-1}, then the flag *ascending* is true and false otherwise.

The reliable bitonic sorting algorithm S_{FT} is given in Figure 3. The individual components of the constraint predicate Φ_P,Φ_F,Φ_C corresponding to the progress, feasibility, and consistency predicates subclasses are discussed in the following paragraphs.

```
/* S_FT, executed by node, 0≤node≤N-1 - Local starting value in a */

Procedure S_FT

  LBS[j]:=a; lmask:=2^j;
  for i:=0 to n-1 do
    {limit:=min(SC^E_{i+1,node},N);
    for j:=i downto 0 do
      {d:=2^j;
      if (node mod (2d)<d)
        {read into (data,lbuf) from node+d;
        lmask:=Φ_C(LBS_i,j,lmask);
        if (node mod 2^(i+2)<2^(i+1))
          b := max(data.a,a);a := min(data.a,a);
        else
          b := min(data.a,a);a := max(data.a,a);
        data:=(a,b);
        write from data,LBS to node+d;}
      else
        {data.a:=a;
        write from data,LBS to node-d;
        read into (data,lbuf) from node-d;
        lmask:=Φ_C(LBS_i,j,lmask);
        (a,b):=data;}
      }   /* End for j */
```

```
/* Verify both the validity of the sorted sequence in LBS with
   respect to LLBS and the bitonic nature of the sequence in LBS */

  if (i≠0)
    if (not bit_compare(LLBS_i, LBS_i)
      signal ERROR to host;
    for m:=SC^S_{i,node} to limit /* Update LLBS */
      LLBS[m]:=LBS[m];
  LBS[node]:=a;lmask:=2^node;
  }   /* End for i */

/* Verify Last Stage by pure exchange of final LBS */

  i:=n-1;
  for j:=i downto 0 do
    {d:=2^j;
    if (node mod (2d)<d)
      read into lbuf from node+d;lmask:=Φ_C(LBS_i,j,lmask);
      write from LBS to node+d;
    else
      write from LBS to node-d;read into lbuf from node-d;
      lmask:=Φ_C(LBS_i,j,lmask);
    }   /* End for j */
    if (not bit_compare(LLBS_i,LBS_i)
        signal ERROR to host;
}
```

Figure 3. Algorithm S_{FT}

(1) *Progress*

The progress component Φ_P ensures that algorithm termination will occur and that at each testable step of the solution, the state of the solution advances to the goal or final solution of the problem. If this progress is not made, then faulty behavior may indefinitely postpone the solution - any solution, even an incorrect solution.

For iterative convergent problems, the progress component involves reduction of error. For the bitonic sorting problem, the number of steps is known a priori to all participants in the algorithm. Thus any early termination is considered an error. The testable step for Φ_P is at the bottom of the outer loop i in algorithm S_{NR}. By Lemma 2 this must be a bitonic sequence of length 2^{i+1}.

The progress test Φ_P is given in Figure 4a. The sequence BS_i is a bitonic sequence of length 2^{i+1} in $SC_{i+1,node}$.

Procedure $\Phi_P(BS_i)$

```
  for k := SC^S_{i,j} to SC^E_{i,j}
    if (ascending)
      if (BS[k+1]<BS[k]) then ERROR;
    else
      if (BS[k+1]>BS[k]) then ERROR;
  if (i ≠ n)
    for k := SC^S_{i,j}+2^i to SC^E_{i+1,j}
      if (ascending)
        if (BS[k+1]>BS[k]) then ERROR;
      else
        if (BS[k+1]<BS[k]) then ERROR;
```

Figure 4a. Φ_P - Progress Component

(2) *Feasibility*

Each testable result must remain within the defined solution space of the problem. Formally, consider a solution space H_i and any intermediate result $\mathbf{U}^{(i)}$. Then for any stage i,

$$\mathbf{U}^{(i)} \varepsilon H_i$$

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. Indeed, these boundary conditions are exactly the class of natural problem constraints. The natural constraint here, and indeed the reason for the choice of the bitonic sort procedure, is that at each stage i of the computation, the bitonic sequence $\mathbf{U}^{(i)}$ formed must contain only the elements to be sorted, no more, no less.

Definition 5: The bitonic sequence $LLBS_i$ is a bitonic sequence of length 2^i resulting from stage $i-2$ of algorithm S_{FT}. The bitonic sequence LBS_i is a bitonic sequence of length 2^{i+1} resulting from stage $i-1$ of algorithm S_{FT}.

The feasibility test Φ_F is given in Figure 4b. Note the "C" conditional statement syntax in the **for** loop. In this notation, x?y:z indicates that if x is true, execute y, otherwise execute z.

Procedure $\Phi_F(LLBS_i,LBS_i)$

```
  l:=SC^S_{i,node};
  u:=SC^E_{i,node};
  for m := ascending?SC^S_{i,node}:SC^E_{i,node};
    ascending?to:downto ascending?SC^E_{i,node}):SC^S_{i,node}

    if (LBS_i[m]=LLBS_i[l] && l≤2^{i-1}+SC^S_{i,j})
      l := l+1;
    else if (LBS_i[m]=LLBS_i[u] && m≥2^{i-1}+SC^S_{i,j})
      u := u-1;
    else
      ERROR;
```

Figure 4b. Φ_F Feasibility Component

(3) *Consistency*

Many intermediate calculations contain additional properties that are indirectly obtainable from the problem's natural constraints. These are defined as *Consistency Conditions*. In the sequential programming environment, all facets of each testable step can be checked. As noted previously, in the multicomputer environment, a consistency predicate may be applied only to the received information and locally known information.

In the case of bitonic sorting, the bitonic subsequence must be distributed to checking processors. Since it is entirely possible for a Byzantine faulty processor to send different versions of the same message to different checking processors, each version of which satisfies the feasibility test locally but is incorrect globally. This situation is said to be *inconsistent*. To achieve consistency, we require that each processor "hears" the same version of a message, in this case a bitonic subsequence. This can be accomplished by sending two copies of each bitonic sequence via vertex disjoint paths to each checking processor. The message routing is constructed in such a way that each message must pass though processors that are capable of checking parts of each message locally. The intersection of these tests, given the same input message, form a global test. When the locally checked messages meet at a checking processor, they must contain the same information. If not, then the message has been altered to satisfy each individual processor locally. This situation is then detected at the checking processor and an error flagged.

The consistency test Φ_C is given in Figure 4c. Note that in keeping with application-oriented paradigm, the test for faulty

behavior is closely intertwined with the actual message delivery of the last bitonic sequence LBS_i. *lmask* is initially set to the binary representation of j and *source* is the address of the sender of the message in *lbuf*.

```
Procedure Φ_C(LBS_i,j,lmask)

/* y>>x is a bitwise right shift of y by x bits */

  limit=min(SC^S_{i+1,j},N)
  mask:=vect_mask(i,j,source);
  omask:=mask;
  mask:=mask>>SC^S_{i+1,j};
  lmask:=lmask>>SC^S_{i+1,j};

  for k:=SC^S_{i+1,j} to limit
    {if (mask&01 && !(lmask&01))
     LBS[k]:=lbuf[k];
     else if (mask&01 && lmask&01)
     if (LBS[k] ≠ lbuf[K])
       ERROR;
     mask:=mask>>1;
     lmask:=lmask>>1;}
   return(omask);

Procedure vect_mask(i,j,node)
  d=2^j;
  if (j=i)
    if (node mod (d<<1)<d)
      mask=1<<node | 1<<node+d;
    else
      mask=1<<node | 1<<node-d;}
  else
    if (node mod (d<<1)<d)
      mask=vect_mask(i,j+1,node+d) | vect_mask(i,j+1,node);
    else
      mask=vect_mask(i,j+1,node-d) | vect_mask(i,j+1,node);
  return(mask);
}
```

Figure 4c. Φ_C Consistency Component

Briefly the concept of the fault-tolerant bitonic sorting algorithm is that as we build up longer bitonic sequences, we (1) check that these sequences are bitonic and (2) check that these bitonic sequences are permutations of the the earlier, shorter, bitonic sequences. The communication of a bitonic sequence from stage i is "piggybacked" in the communication that occurs naturally at stage $i+1$. This affords the resulting fault-tolerant algorithm no increase in message complexity over the non fault-tolerant algorithm S_{NR} (although, as we show in Section 5, the length of the messages increases).

Definition 6: A bitonic (sub)-sequence LBS_i is *complete* if for each $I_j \in LLBS_i$ there exists a permutation Π on the elements of $LLBS_i$ to the lower or upper half of LBS_i determined by the range of $SC_{i-2,j}$ that is bijective.

Lemma 3: Algorithm vect_mask(i,j,k) provides a bit_vector in which a 1 in bit position l indicates that $LBS_i[l]$ has been collected from node l in a message exchange at node k from iteration i and one or more of the iterations $j, j-1, \cdots, 0$.

Proof: By induction on j, we show that the bit vector returned by vect_mask represents the actual message traffic of algorithm S_{FT} (and S_{NR}). If $i=j$, then this is the start of stage i of the message exchange and mask is set to processor P_k and $P_{k \oplus 2^j}$.

Assume that vect_mask$(i,l+1,k)$ returns a correct (as in the lemma statement) mask. Then for $j=l$, the mask returned is the $j+1$ mask for P_k or'ed with the $j+1$ mask for $P_{k \oplus 2^j}$, each of which is the correct mask. □

```
Procedure bit_compare(LLBS_i,LBS_i)

  Φ_P(LBS_i);
  Φ_F(LLBS_i,LBS_i)
```

Lemma 4: Algorithm bit_compare detects non-bitonic LBS_i and non-complete LBS_i with respect to $LLBS_i$ given a bitonic $LLBS_i$ as input.

Proof: Trivial from the previous discussion and bit_compare code. □

Lemma 5: At the end of stage 0 at least one processor in $SC_{0,j}$ can detect an error made by processor P_j that results in either (1) a non-bitonic LBS_0 or (2) if the the maximum number of faulty nodes in $SC_{0,j}$ is 1, a non-complete LBS_0 given a bitonic, complete $LLBS_i$.

Proof: For stage $i=0$, each P_j, P_{j+1} for j even contains the actual correct initial values I_j, I_{j+1} in LBS_0. This forms a complete bitonic sequence of length 2. This then becomes $LLBS_0$ which is bitonic. □

Lemma 6: At the end of stage $i, i>0$ 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 LBS_i or (2) if the the maximum number of faulty nodes in $SC_{i-1,j}$ is i, a non-complete LBS_i given a bitonic, complete $LLBS_i$.

Proof: By induction on i, the step count. By Lemma 5, at $i=0$, each P_j in $SC_{0,j}$ holds a complete, correct $LLBS_0$,

Assume that LBS_{k-1} has been verified complete and correct with respect to $LLBS_{k-1}$ at the end of step $k-1$. Then during step k, LBS_k is filled by partial LBS_k's according to the bit sequence of vect_mask(k,j). Since each element considered for inclusion in LBS_k is reported to at least one processor through k vertex disjoint paths in the graph G, the effects of a $k-1$ faulty relays are limited to one of these paths. If any two of the candidate elements differ, an error is signaled. Thus if the sender is faulty, it must send identical values along all paths. If these values destroy the bitonic nature of LBS_k or if they are not complete with respect to $LLBS_k$, then, by Lemma 4, bit_compare will flag an error. Otherwise, no error has occurred. □

An example of S_{FT} is shown for $n=3$ in Figure 5. The list to be sorted, {10,8,3,9,4,2,7,5} is stored in processors P_0–P_7. Note that for stage 1, the LBS and LLBS are shown only for $SC_{1,2}$ and $SC_{1,6}$ and for stage 2, only $SC_{2,0}$ is shown.

4. ERROR COVERAGE AND RESILIENCE

Analysis of the expected error coverage is key for any fault-tolerant algorithm.

Theorem 3: 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 6 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 $i-1$ faulty nodes per $SC_{i,j}$ (i now = n in bit_compare), a processor P_j can detect any faulty behavior. □

As Theorem 3 shows, the constraint predicate Φ formed by Φ_P, Φ_F, Φ_C detects all errors from the Byzantine fault class committed by one faulty processor. Thus the reliable bitonic sort algo-

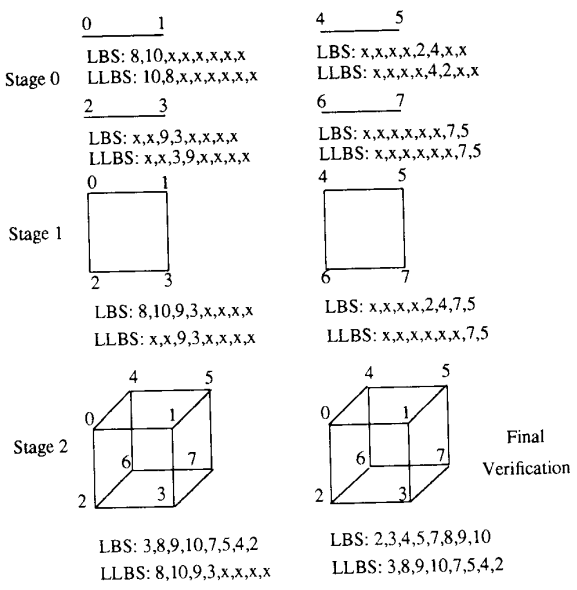

Figure 5. Example of S_{FT} for $n=3$.

rithm is fail-stop using components which may fail in Byzantine ways. The result of the calculation is either completely correct, or the entire system halts with an error condition. We are guaranteed that under a single processor failure, we will never receive an incorrect sorting result. Furthermore, unless an error occurs, the host is never involved in the calculation. This is ideal from a performance standpoint since, as mentioned earlier, the host can become a bottleneck.

Clearly, there will be a performance penalty to pay for the increased reliability of algorithm S_{FT} over the unreliable S_{NR}. The reader may question how much overhead is introduced and whether it might be better to simply send all the data to the host, let the host sort the data, and return the final result to the node processors. Another possibility is to send all the data to the host, sort the data in the node processors, and send the results to the host for verification. As we show in the next section, the performance of both these possibilities becomes unreasonable for even moderately large problems.

5. COMPLEXITY

As mentioned in Section 2, algorithm S_{NR} has a time complexity of $O(\log_2^2 N)$. Sequential sorting, on the other hand, has a lower bound of $O(N\log_2 N)$. The communication complexity (since the data must be transferred from the nodes to the host) is $O(N)$ for sequential sorting. The communication complexity for S_{NR} is $O(\log_2^2 N)$. Thus it is expected that the fault tolerant algorithm S_{FT} will have time/communication complexity between these two.

Lemma 7: Algorithm vect_mask(i,j) has time complexity of $O(2^{i-j})$.

Proof: Let the running time of vect_mask(i,j) be $T_{VM}(i,j)$. If $i=j$, then four logical bit operations are performed and $T_{VM}(i,i)=O(2^0)=O(1)$. If $i>j$, then two executions of vect_mask$(i,j+1)$ are performed. Thus we have the linear recurrence

$$T_{VM}(i,j) = 2T_{VM}(i,j+1)$$

Solving this yields:

$$T_{VM}(i,j) = 2^{i-j}T_{VM}(i,i) = O(2^{i-j}) \square$$

Lemma 8: Algorithm bit_compare() has a time complexity of $O(2^i)$ for a calling node k at step i.

Proof: Finding the size and location of the subcube is $O(1)$ by simple application of Definition 4. Verifying the sorted nature of each half of LBS_i can be done in $O(2^i)$ time. Verification of the completeness of LBS_i with respect to $LLBS_i$ can be done in $O(2^i)$. Thus the total time complexity is $O(3\cdot 2^i)=O(2^i)$. □

Lemma 9: Algorithm $\Phi_C(i,j)$ has a time complexity of $O(2^{j+1}+2^{i-j})$.

Proof: There are at most 2^{j+1} non-zero entries reported by vect_mask at step j plus the time for vect_mask(i,j) of $O(2^{i-j})$ by Lemma 7 gives the bound. □

Theorem 4: Algorithm S_{FT} has a computational time complexity of $O(N)$ and a communication complexity of $O(\log_2^2 N+N\log_2 N)$.

Proof: For a particular value of i, j ranges from 0 to i. By Lemma 9, each Φ_C(i,j) has complexity $O(2^{j+1}+2^{i-j})$. Since each iteration contains time for 2 Φ_C iterations (because the computations cannot be overlapped), we have

$$2\sum_{j=0}^{i} 2^{j+1}+2^{i-j}=2\cdot 2^{i+2}+2^{i+1}=O(2^{i+3})$$

By Lemma 8, bit_compare has complexity 2^i and updating LLBS takes 2^i so for a single iteration of i, we have a run time of $O(2^{i+3}+2^i+2^i)$. Summing over all i from 0 to $n-1$ and adding in the final verification step, we have

$$\sum_{i=0}^{n-1}(2^{i+3}+2^i+2^i)+2^{n+2}=2^{n+3}+2\cdot 2^{n-1}+2^{n+2}=O(2^{n+3})$$

Since $n=\log_2 N$, we have the run time of $S_{FT}=O(2^{\log_2 N+2})=O(N)$

The communication complexity of the main loop is composed of $O(\log_2^2 N)$ of S_{NR} plus $O(N\log_2 N)$ based on the length of the message sent. □

For comparison purposes, a sequential "sorting" algorithm was constructed for the host. Sort is quoted since we implement this "sort" as a single if statement executed $N\log_2 N$ times to achieve the theoretical minimum. $O(N)$ communication is required to send/receive the sorted data. Additionally, a sequential verification was constructed. In this algorithm, the initial data is sent to the host, sorted by the node processors, and the sorted data also sent to the host. The host then implements Theorem 1 to verify the results. This process takes $O(N)$ communication complexity and $O(N\log_2 N)$ computational complexity since the matching of the ordered and unordered list becomes equivalent to finding a permutation in the sense of Definition 1. Thus, for the following discussion, the best sequential algorithm is $O(N\log_2 N)$.

The algorithms S_{NR}, S_{FT}, and a sequential sort were implemented on a 32 nodes of a 64 node Ncube computer to sort 32-bit integers into ascending order (implementation constraints forced consideration of the smaller subcube). Timings were obtained for for all three algorithms for problem sizes of 4, 8, 16, and 32 nodes. This is shown in Figure 6.

The execution (observed) results are inconclusive since the cube we have available is very small. Asymptotically, we certainly expect S_{FT} to be more efficient than the host sort, but the constant multiplier in the run time order dominates for these small problem sizes. Measurement of the running time for each component of the two algorithms yields the following table (measured in clock ticks).

Algorithm	Communication Time	Computation Time
S_{FT}	$8\log_2^2 N+.05N\log_2 N$	$11.5N$
Sequential	$14N$	$0.45N\log_2 N$

This behavior is plotted as (Theoretical) in Figure 6. Comparison with the observed values indicates this approximation is close to the actual run time for smaller cube sizes. In the projected run times of Figure 7, S_{FT} rapidly becomes more efficient for the size of cubes that we are concerned with in a real multicomputer application. The portion of this plot covered by Figure 6 is highlighted in the lower left corner of Figure 7. These projected run times indicate that the penalty paid for fault tolerance in parallel sorting is less than the cost for sequential sorting in the host. Furthermore, it is easy to see in the limit that the cost of reliable parallel sorting becomes 11% the cost of sequential sorting.

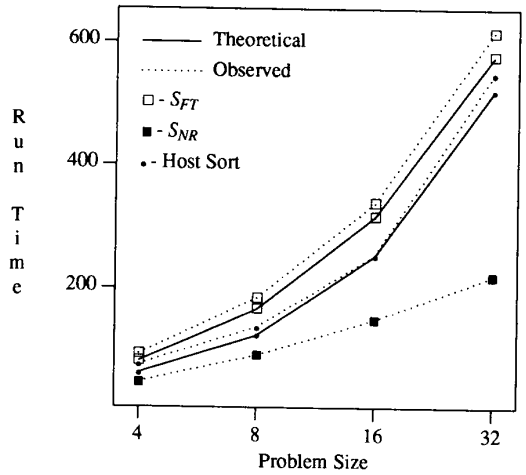

Figure 6. Sorting Time Comparisons

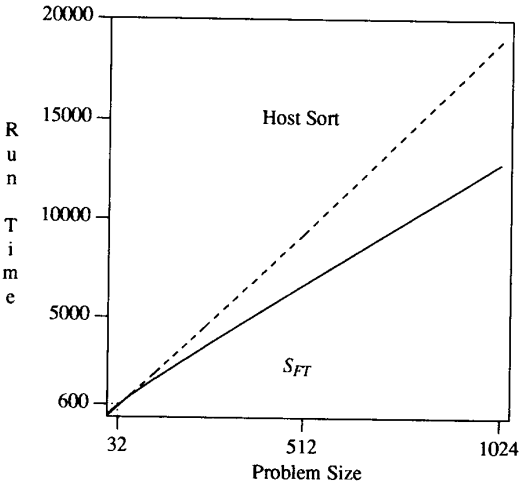

Figure 7. Projected Sorting Time Comparisons - Large Systems

The next logical step is to consider bitonic sort/merge in which each processor hold m elements to be sorted instead of 1. The message exchange structure is preserved. At each compare/exchange step, however, half of the processors must do a compare/exchange of $2m$ elements and then each processor must sort these m elements locally. These two operations add $O(m+m\log_2 m)$ to the computation time for both S_{NR} and S_{FT}. It is easy to see (we omit the details) that each of the predicates Φ scales by m. Thus for block sorting blocks of size m, we have the following run time comparisons between S_{FT} and host sequential sort plotted in Figure 8 for a representative value of m. As is to be expected this plot is virtually a right shift of focus on the plot of Figure 6 due to the scale by m.

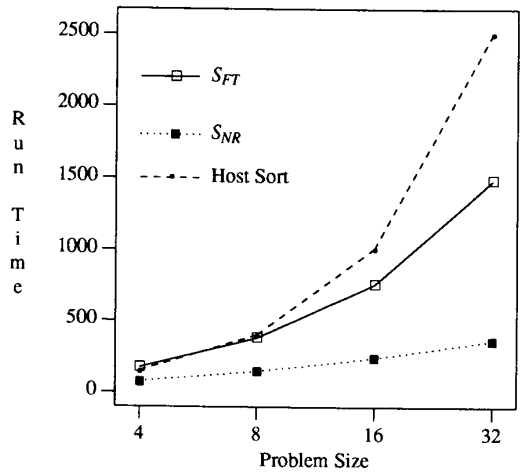

Figure 8. Sorting Time Comparisons for m=8

7. CONCLUDING REMARKS

This paper has presented a fault-tolerant parallel sorting algorithm developed using the application-oriented fault tolerance paradigm. The algorithm is tolerant of $\log_2 N-1$ processor/link failures.

The addition of reliability to the sorting algorithm results in a performance penalty. Asymptotically, the developed fault-tolerance algorithm is less costly than host sorting. Experimentally we have shown that fault-tolerant sorting becomes quickly more efficient than host sorting when the bitonic sort/merge is considered.

The main contribution of this paper is the demonstration that the application-oriented fault tolerance paradigm is applicable to problems of a non-iterative nature. Again, all that is necessary for successful algorithm development is a sufficient set of natural problem constraints; in this case the bitonic nature of the intermediate results.

REFERENCES

1 Athas, W. and Seitz, C., "Multicomputers: Message-Passing Concurrent Computers," *Computer*, August, 1988, pp. 9-25.

2 Batcher, K., "Sorting Networks and Their Applications," *Proc. of the 1968 Spring Joint Computer Conference*, vol. 32, AFIPS Press, Reston, VA, pp. 307-314.

3 Hayes, J., Mudge, T., Stout, Q., Colley, S. and Palmer, J., "Architecture of a Hypercube Supercomputer," *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986, pp. 653-660.

4 Jou, J. and Abraham, J., "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proceedings of the IEEE*, May 1986, pp. 732-741.

5 Lamport, L., Shostak, R., Pease, M., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol 4., No. 3, July 1982, pp. 382-401.

6 McMillin, B. and Ni, L., "A Reliable Parallel Algorithm for Relaxation Labeling," *Proceedings of the 1988 International Conference on Parallel Processing for Computer Vision and Display*, Leeds, U.K., Addison-Wesley, January, 1988.

7 McMillin, B. and Ni, L., "Executable Assertion Development for the Distributed Parallel Environment," *Proceedings of the 12th Intl. COMPSAC*, Chicago, IL, October, 1988, pp. 284-291.

8 Quinn, M., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

9 Stucki, L. "New directions in automated tools for improving software quality," *Current Trends in Programming Methodology*, Vol. 2., R.T. Yeh (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 80-111.