Parallel and Distributed Computing
CS3006 (BDS-6A)
Lecture 20

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

13 April, 2023

Previous Lecture

- MPI
  - MPI_Probe and MPI_Get_count
  - MPI_Barrier
  - MPI_Bcast
  - MPI_Reduce (different operations)
  - MPI_Allreduce
  - MPI_Scan (different operations)
  - MPI_Gather and MPI_Scatter
  - SPMD Model
  - MPI_Alltoall

Sorting in the Parallel Era…..

- Can we efficiently apply a Bubble-Sort type of sorting algorithm when the individual values are dispersed across different machines (processes)?

Sorting - Overview

- One of the most commonly used and well-studied Algorithms.
- Sorting can be *comparison-based* or *non-comparison-based*.
- The fundamental operation of comparison-based sorting is *compare-exchange*.
- The lower bound on any comparison-based sort of *n* numbers is $\Theta(n\log n)$.
- Let's explore a comparison-based sorting algorithm.

Sorting – Basics

What is a parallel sorted sequence?

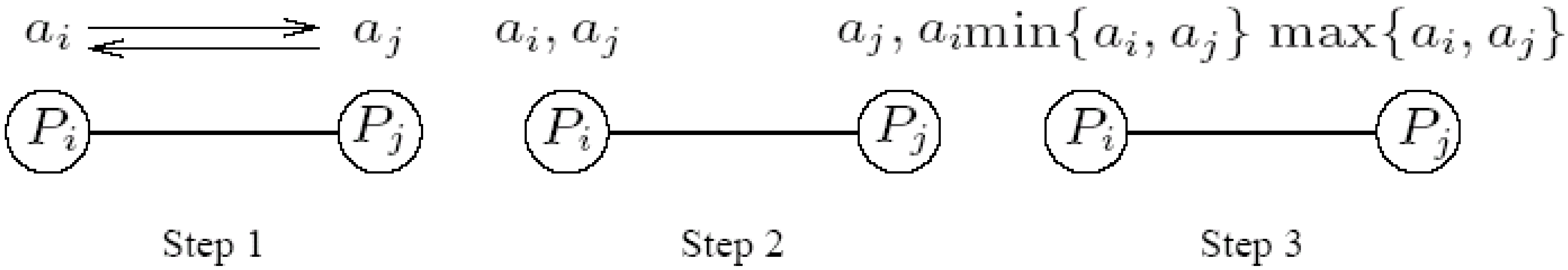
→ Where are the input and output lists stored?

Answers:

- We assume that the input and output lists are distributed.
- The sorted list is partitioned with the property that each partitioned list is sorted and each element in processor P_i's list is less than that in P_j's list if $i < j$.

Sorting: Parallel Compare Exchange Operation

- A parallel compare-exchange operation. Processes P_i and P_j send their elements to each other. Process P_i keeps $\min\{a_i, a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.

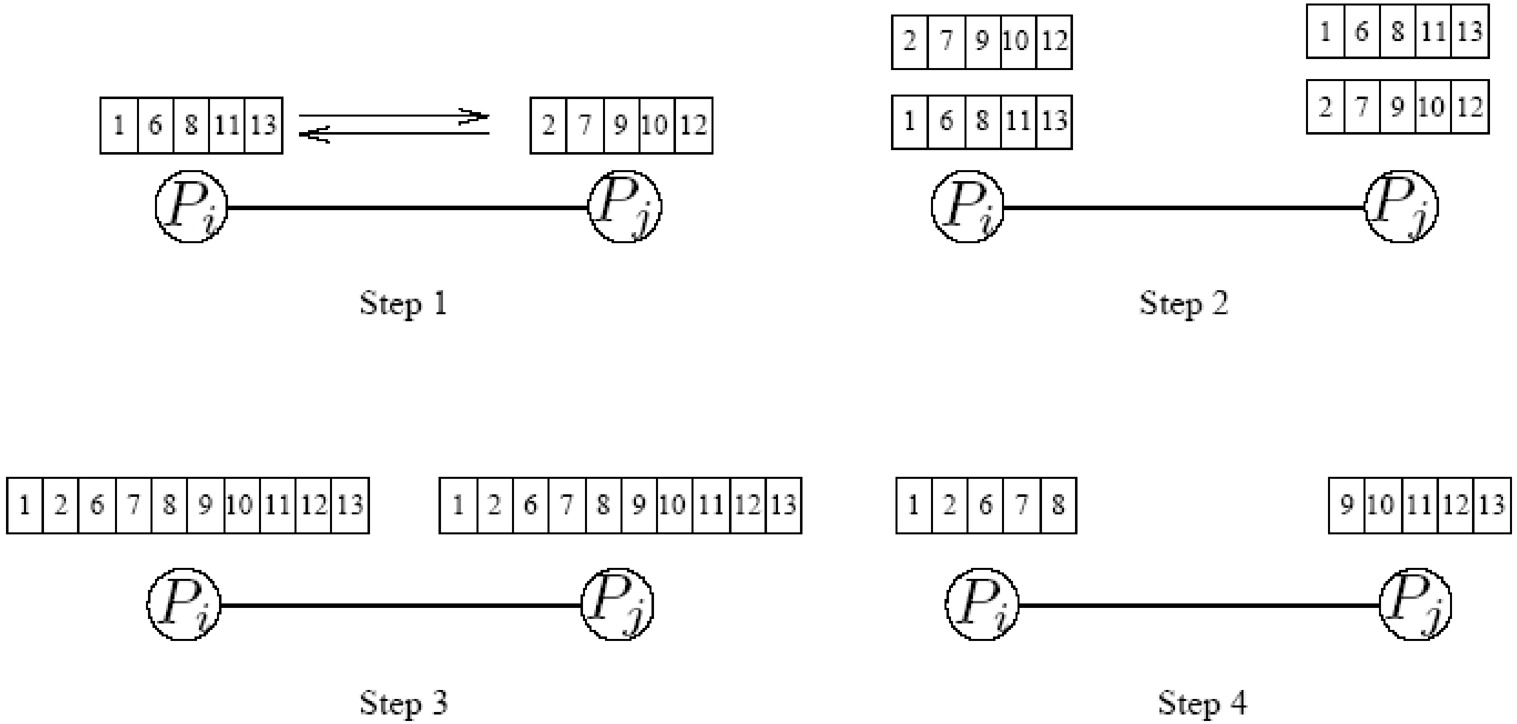


Sorting: Parallel Compare Exchange Operation [cost estimation]

- If each processor has one element, the compare exchange operation stores the smaller element at the processor with smaller id. This can be done in $t_s + t_w$ time.
- If we have more than one element per processor, we call this operation a compare split. Assume each of two processors have *n/p* elements.
- After the compare-split operation, the smaller *n/p* elements are at processor P_i and the larger *n/p* elements at P_j, where $i < j$.
- The time for a compare-split operation is $(t_s + t_w n/p)$, assuming that the two partial lists were initially sorted.
  - Note that this time is only accounting communication costs. Computation and memory complexities are separate things.

Sorting: Parallel Compare Exchange

- A compare-split operation. Each process sends its block of size *n/p* to the other process.
- Each process merges the received block with its own block and retains only the appropriate half of the merged block.
- In this example, process P_i retains the smaller elements and process P_j retains the larger elements.



Bubble Sort and its Variant

- The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted:

procedure BUBBLE_SORT(n)
begin
    for $i := n - 1$ **downto** 1 **do**
        for $j := 1$ **to** i **do**
            compare-exchange(a_j, a_{j+1});
end BUBBLE_SORT

Visualization (first pass, seq. BubbleSort)

First pass →

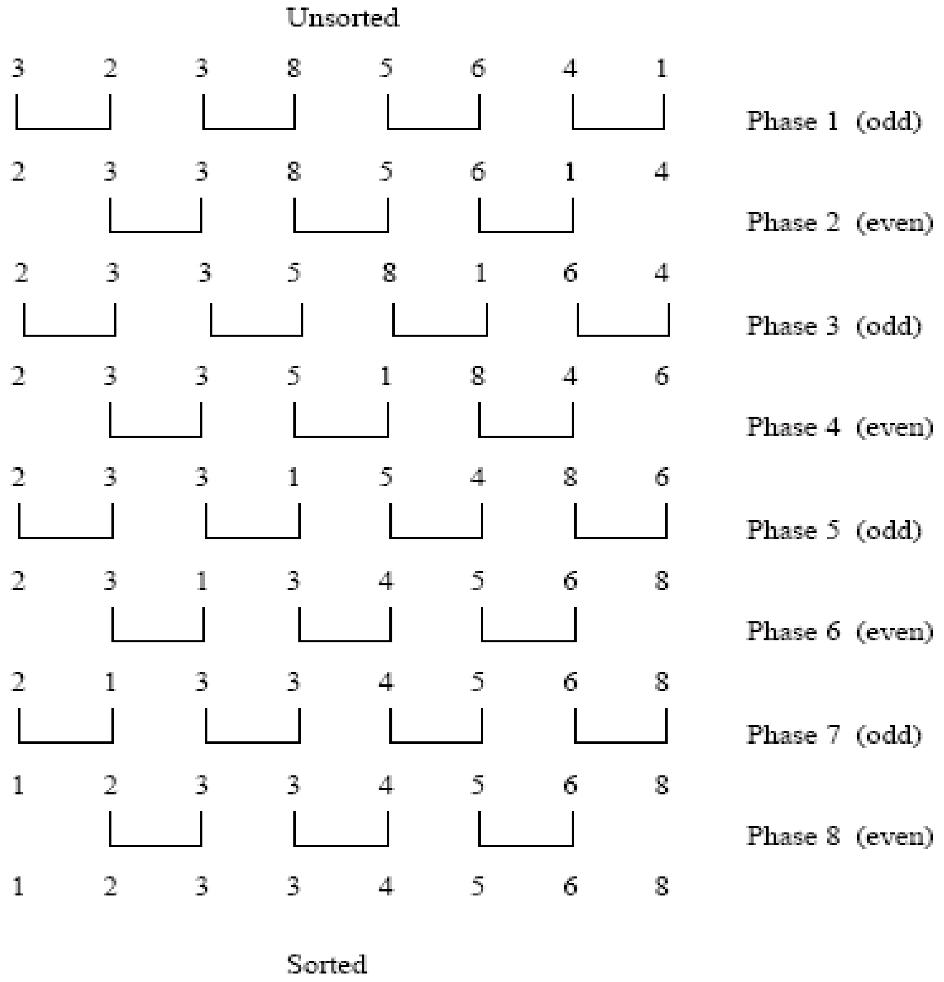
54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

Bubble Sort and its Variant

- The complexity of bubble sort is $\Theta(n^2)$.
- Bubble sort is difficult to parallelize since the algorithm has no concurrency.
- A simple variant, though, uncovers the possible concurrency.

Bubble Sort [Odd-Even Transposition]

- Sorting *n* = 8 elements, using the odd-even transposition sort algorithm.
- During each phase, at most 8 elements are compared.
- **[This is according to the sequential algorithm]**



Bubble Sort [Odd-Even Transposition]

- Sequential odd-even sort algorithm

```
procedure ODD-EVEN(n)
begin
    for i := 1 to n do
    begin
        if i is odd then
            for j := 0 to n/2 - 1 do
                compare-exchange(a_{2j+1}, a_{2j+2});
        if i is even then
            for j := 1 to n/2 - 1 do
                compare-exchange(a_{2j}, a_{2j+1});
    end for
end ODD-EVEN
```

Odd-Even Sort (Seq. Complexity)

- After n phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.

Parallel Odd-Even Sort

Time / Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	7	3	8	5	6
4	2	1	4	3	7	5	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Parallel time complexity: $T_{par} = \mathbf{O}(n)$ (for P=n)

Source: https://www.slideshare.net/richakumari37266/parallel-sorting-algorithm

Parallel Odd-Even Sort

- **Algorithm Through Observations:**

1. There are total **P** phases/steps. Where P is number of processes
2. **For even phases**
   i. If 'myrank' is even → Communication partner is ('myrank'+1)
   ii. If 'myrank' is odd → Communication partner is ('myrank' - 1)

3. **For odd phases:**
   i. If 'myrank' is even → Communication partner is ('myrank' - 1)
   ii. If 'myrank' is odd → Communication partner is ('myrank'+1)
4. Communication partners remain constant
5. If 'myrank' is less-than the partner, then keep lower values in compare-split-operation

Parallel Odd-Even Sort

Complexity when *n* = P

- Consider the one item per processor case.
- There are P iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\boldsymbol{\Theta(n)}$.
- Parallel run time means computation performed by each of the processors in parallel.

Parallel Odd-Even Sort

Complexity when *n* > P

- Consider a block of *n/p* elements per processor.
- The first step is a local sort.
- In each subsequent step, the compare exchange operation is replaced by the compare split operation.
- The parallel run time of the formulation is:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

comm. steps for a single process

```c
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

main(int argc, char *argv[])
{
   int n;           /* The total number of elements to be sorted */
   int npes;        /* The total number of processes */
   int myrank;      /* The rank of the calling process */
   int nlocal;      /* The local number of elements, and the array that stores them */
   int *elmnts;     /* The array that stores the local elements */
   int *relmnts;    /* The array that stores the received elements */
   int oddrank;     /* The rank of the process during odd-phase communication */
   int evenrank;    /* The rank of the process during even-phase communication */
   int *wspace;     /* Working space during the compare-split operation */
```

```
    int i;
    MPI_Status status;

    /* Initialize MPI and get system information */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    n = atoi(argv[1]);
    nlocal = n/npes; /* Compute the number of elements to be stored locally. */

    /* Allocate memory for the various arrays */
    elmnts  = (int *)malloc(nlocal*sizeof(int));
    relmnts = (int *)malloc(nlocal*sizeof(int));
    wspace  = (int *)malloc(nlocal*sizeof(int));
```

```
/* Fill-in the elmnts array with random elements */
srandom(myrank);
for (i=0; i<nlocal; i++)
  elmnts[i] = random();

/* Sort the local elements using the built-in quicksort routine */
qsort(elmnts, nlocal, sizeof(int), IncOrder);

/* Determine the rank of the processors that myrank needs to communicate during */
/* the odd and even phases of the algorithm */
if (myrank%2 == 0) {
  oddrank  = myrank-1;
  evenrank = myrank+1;
}
else {
  oddrank  = myrank+1;
  evenrank = myrank-1;
}
```

```c
  /* Set the ranks of the processors at the end of the linear */
  if (oddrank == -1 || oddrank == npes)
    oddrank = MPI_PROC_NULL;
  if (evenrank == -1 || evenrank == npes)
    evenrank = MPI_PROC_NULL;

  /* Get into the main loop of the odd-even sorting algorithm */
  for (i=0; i<npes-1; i++) {
    if (i%2 == 1) /* Odd phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
          nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
    else /* Even phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
          nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

    CompareSplit(nlocal, elmnts, relmnts, wspace,
                 myrank < status.MPI_SOURCE);
  }

  free(elmnts); free(relmnts); free(wspace);
  MPI_Finalize();
}
```

```c
/* This is the CompareSplit function */
CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
             int keepsmall)
{
  int i, j, k;

  for (i=0; i<nlocal; i++)
    wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */

  if (keepsmall) { /* Keep the nlocal smaller elements */
    for (i=j=k=0; k<nlocal; k++) {
      if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
        elmnts[k] = wspace[i++];
      else
        elmnts[k] = relmnts[j++];
    }
  }
  else { /* Keep the nlocal larger elements */
    for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
      if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
        elmnts[k] = wspace[i--];
      else
        elmnts[k] = relmnts[j--];
    }
  }
}
```

```c
/* The IncOrder function that is called by qsort is defined as follows */
int IncOrder(const void *e1, const void *e2)
{
  return (*((int *)e1) - *((int *)e2));
}
```

Failure Handling

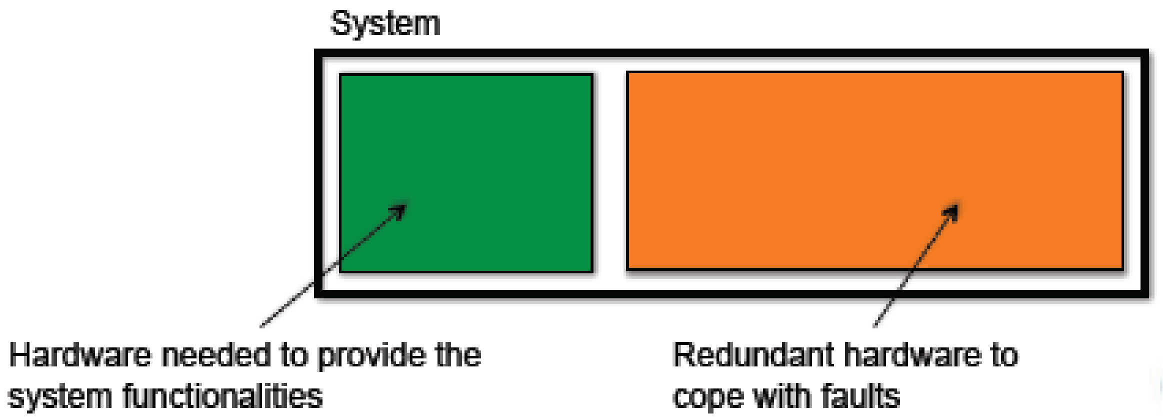
- As the number of components increases, the fault rate also increases
- How to detect failures and make corrections?
- Fault Tolerance
  - fault tolerance or graceful degradation is the property that enables a system to continue operating properly in the event of the failure of some of its components
  - a system continues its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails
- Redundancy
  - redundancy is the duplication of critical components or functions of a system with the intention of increasing reliability of the system

Redundancy design techniques

- Hardware redundancy
- Information redundancy
- Time redundancy

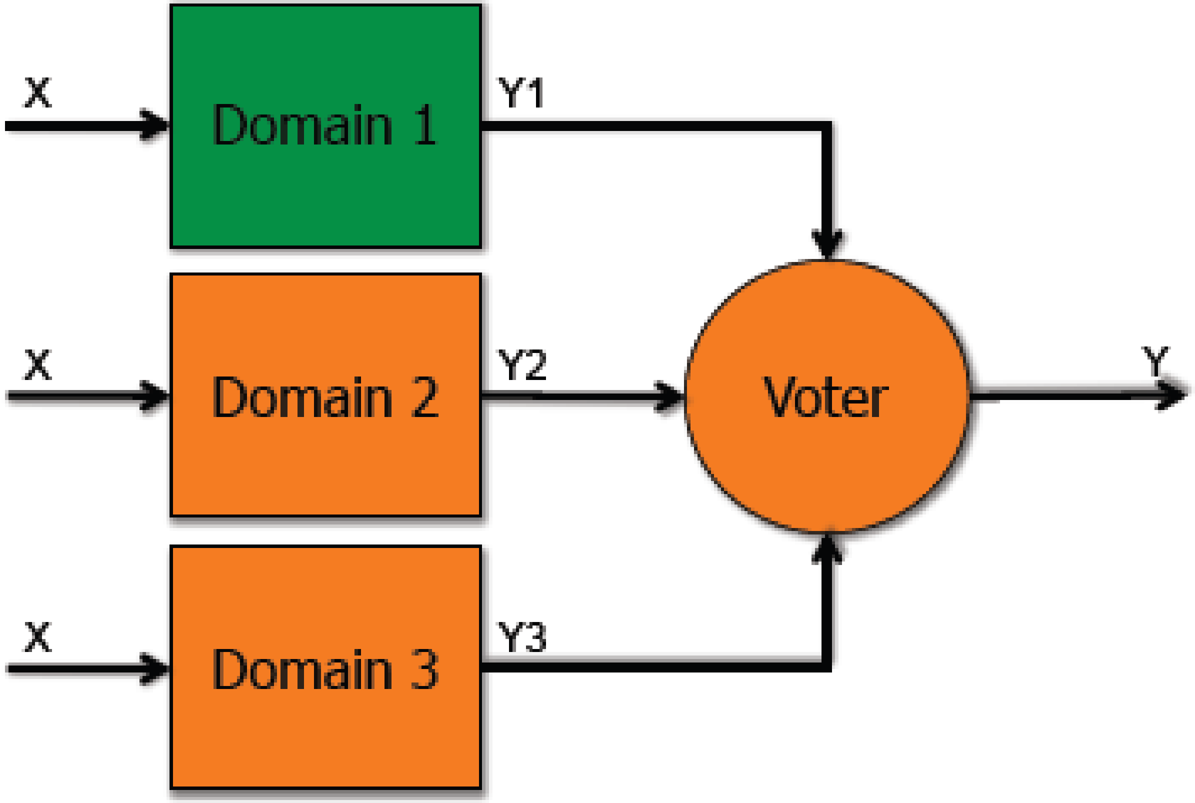
Hardware redundancy

- The system is implemented using *more hardware than that which is needed* for implementing the system functionalities
- The redundant hardware is used for *dealing with faults*



- Types of Hardware redundancy
  - Passive redundancy
  - Active redundancy
  - Hybrid redundancy

Passive Redundancy



- The HW needed to implement the system is replicated 3 times:
  - Triple Module Redundancy (TMR)

Passive Redundancy

- A *majority voter* decides the output to be committed to the user on the basis of the outputs coming from the three domains
- Voter implements the following functionality

```
P_VOTER: process( Y1, Y2, Y3 )
    Begin
      if Y1=Y2 then
            Y = Y1;
      else if Y1=Y3 then
            Y = Y1;
      else if Y2=Y3 then
            Y = Y2;
      else “display error”;
      end if;
    end process;
```

Passive Redundancy

Used to achieve fault tolerance:

- The voter *stops the propagation of faults* that never reach the outputs: error masking

- In case of *permanent fault of a domain no corrective actions are taken*: no error correction

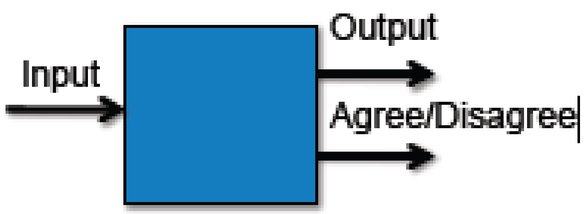
Cost:

- Area: >3x
- Time: negligible

Active Redundancy

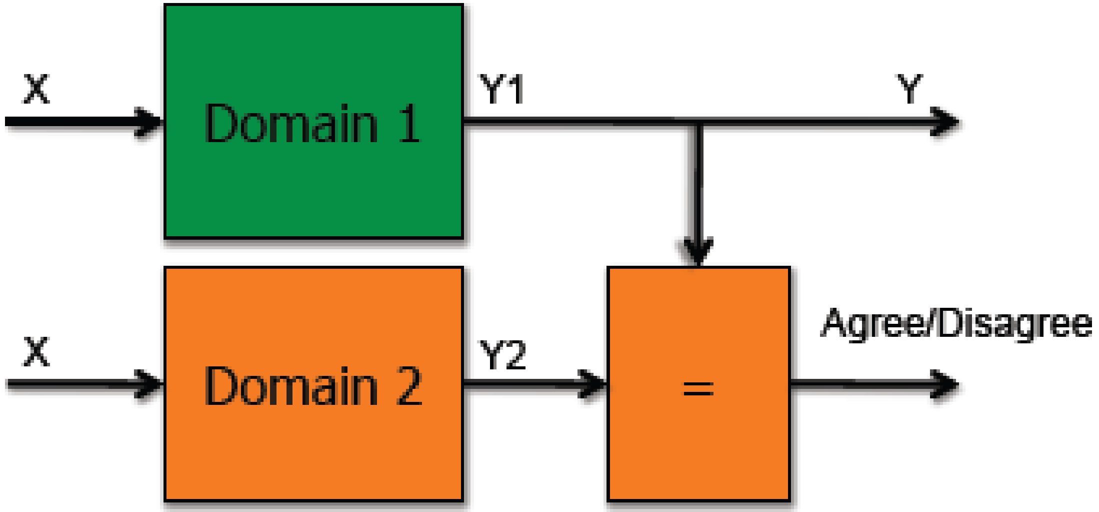
Alternative to passive redundancy

- It implements fault detection
- It may also possibly implement:
  - Error localization
  - Error containment
  - Error recovery



- The purpose of fault detection is to assert a *signal (Agree/Disagree)* every time the system output *differs from the expected* for the given input

Active Redundancy

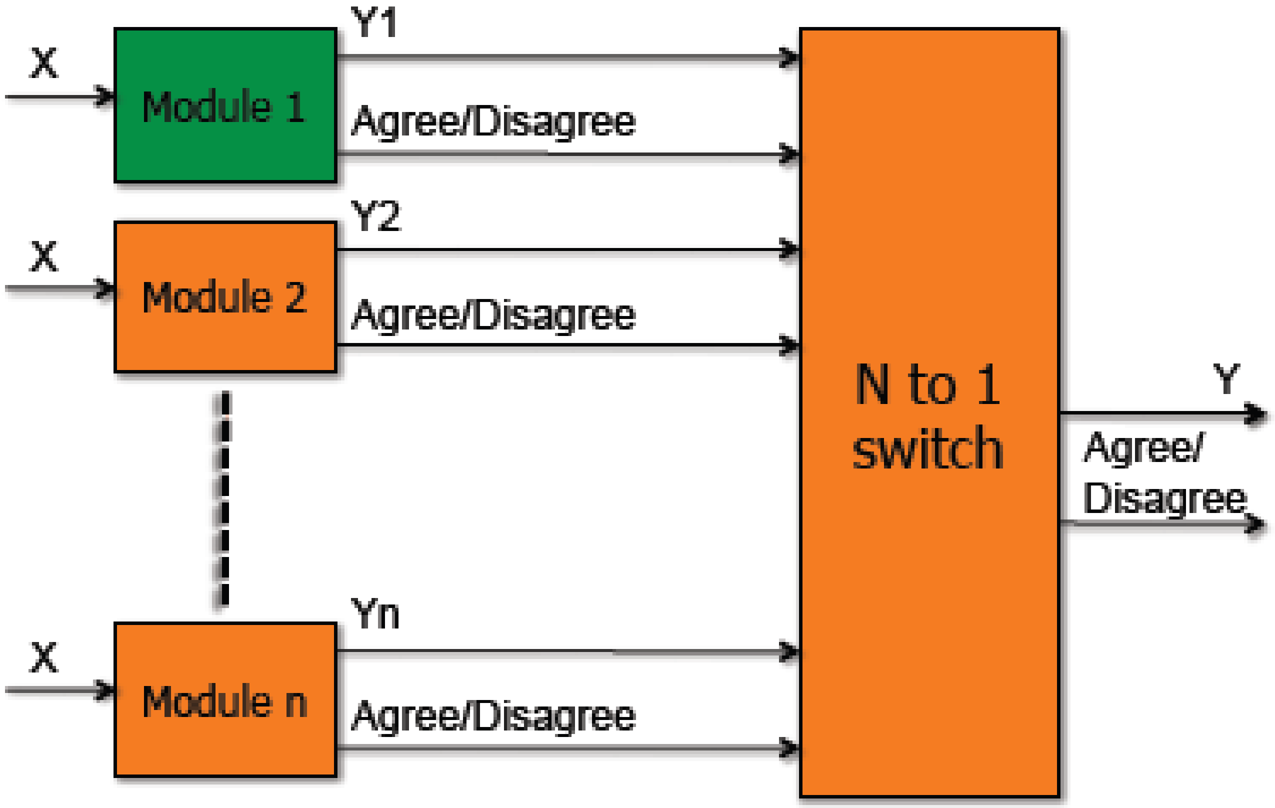


- Duplication with comparison

Active Redundancy

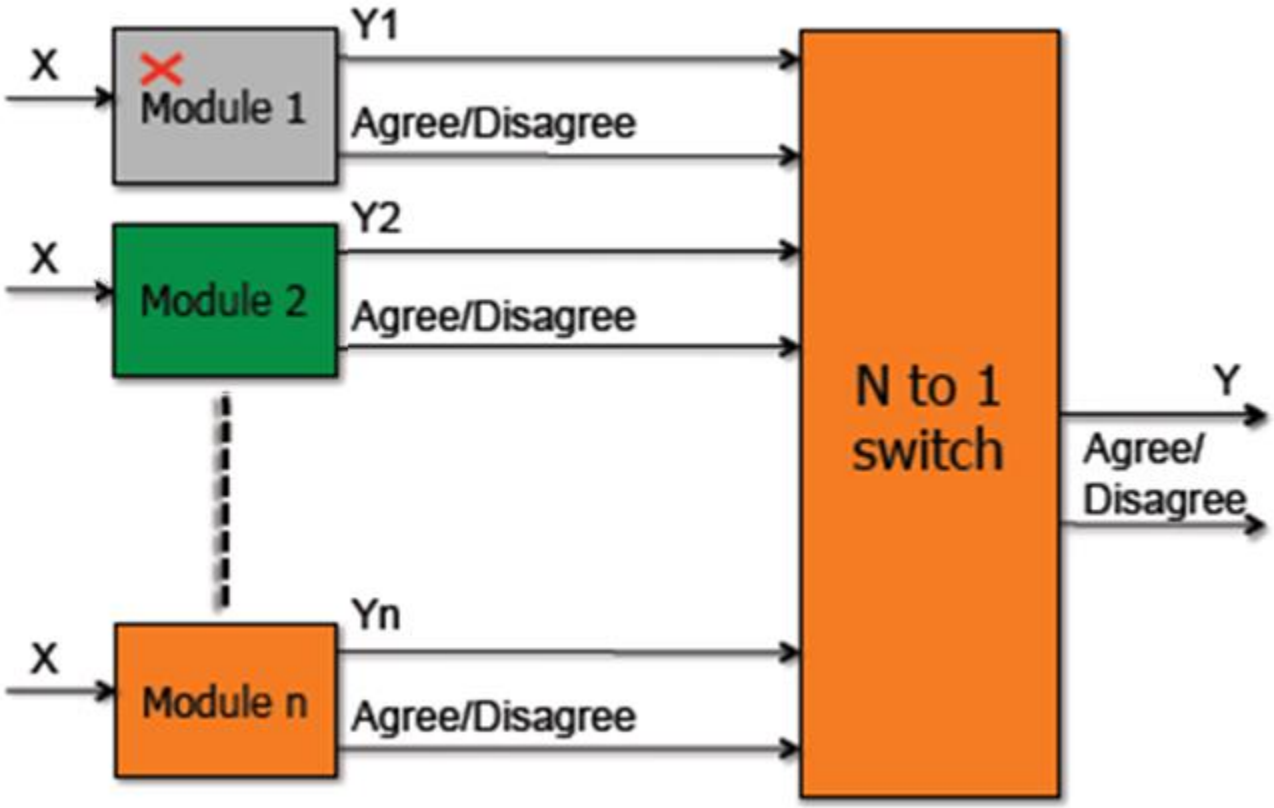
- The purpose of *error localization, containment and recovery* is to reconfigure the system *upon error detection* so that the correct system functionalities are recovered after a certain recovery time
- During recovery the *system is unavailable*:
- The system is not able to respond to inputs
- Maximum allowed recovery time is a function of the application
- Can be implemented using:
  - Hot standby sparing
  - Cold standby sparing

Hot standby sparing



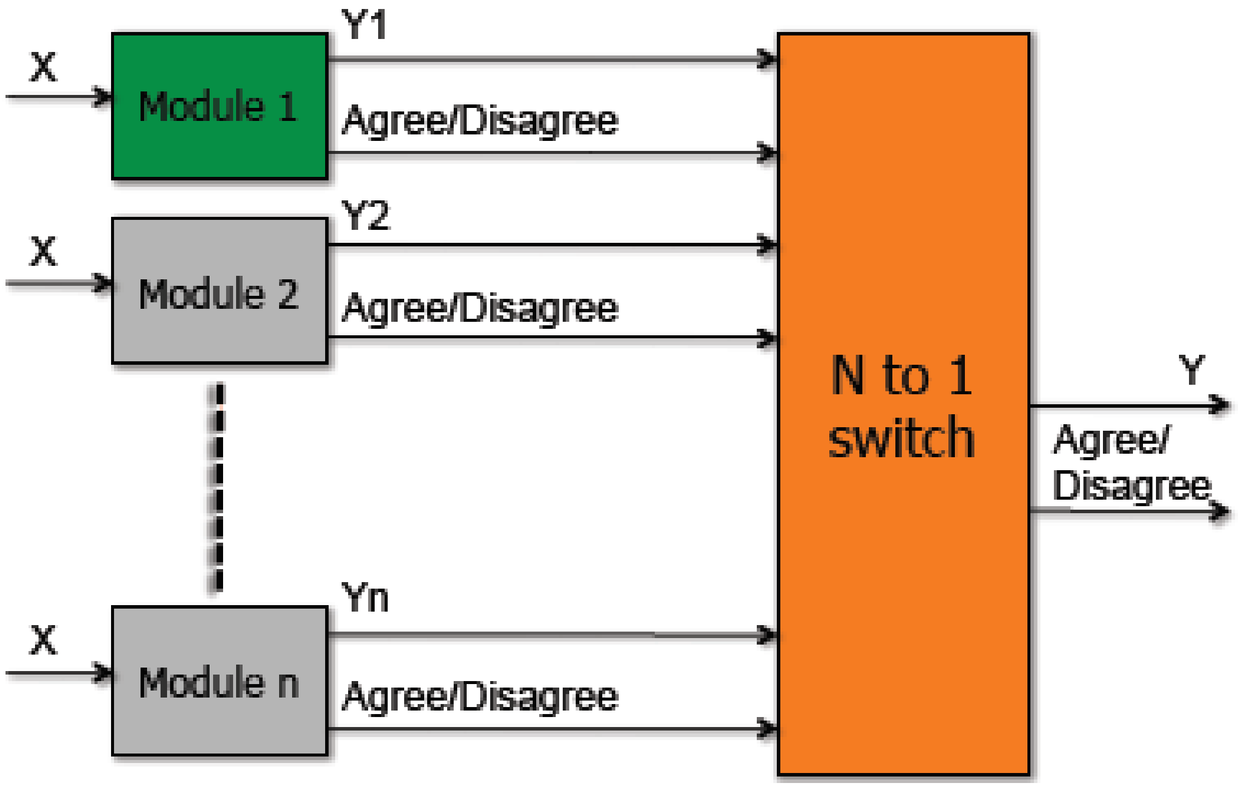
- N modules are used all active at the same time
- One module is the primary module, others are spare modules

Hot standby sparing



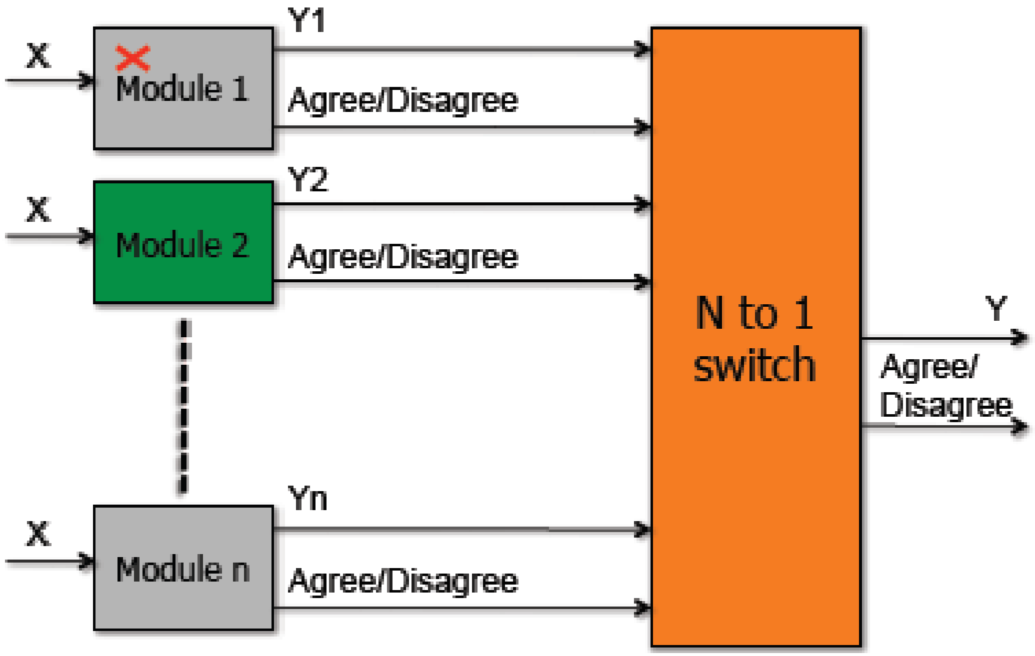
- When an error is detected, the switch sets the faulty primary module to *not-used state*, and it selects a new primary module among the *spares*

Cold standby sparing



- N modules are used: one active, the *primary module*, the others powered-off, the *spare modules*.

Cold standby sparing



- When an error is detected, it *powers-off* the *primary module*, selects a new primary module among the *spares*, and then turns it on.

Dealing with faulty modules

- In hot standby two methods can be used for dealing with the faulty module
- Permanently evicted:
  - When detected as faulty, it is powered-off and no longer used
  - The number of spares decreases over time
- Temporarily evicted:
  - When detected as faulty, it is set to idle but not powered off
  - The switch monitors the status of the idle module and, in case it is no longer affected by faults, it is promoted to the role of spare module
  - In case of transient faults the number of spares remain constant over time

Tradeoffs

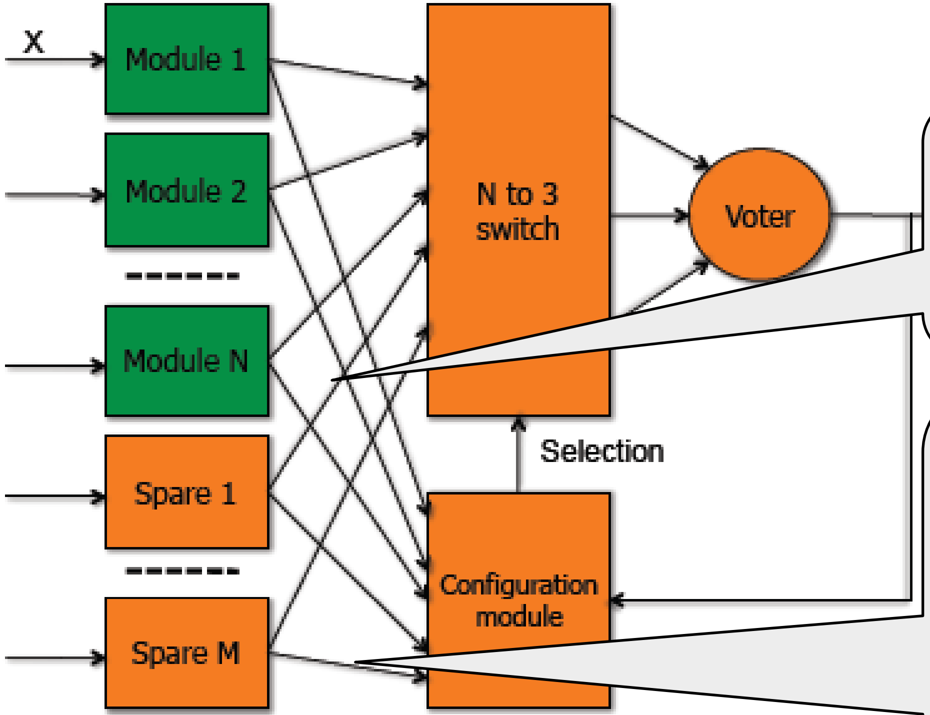
- Active redundancy for error detection only is less expensive than TMR, but does not provide masking
- Standby sparing is more expensive than TMR but it provides error correction:
  - It can survive up to N-1 permanent errors, and provide error detection in case of N permanent faults
  - TMR is able to survive 1 permanent fault, only

Cold sparing vs. hot sparing

- Area: same
- Energy: cold sparing uses 1/n of the energy of hot sparing
- Availability: in hot sparing, availability is higher than in cold one

Hybrid Redundancy

N primary modules and M spare modules



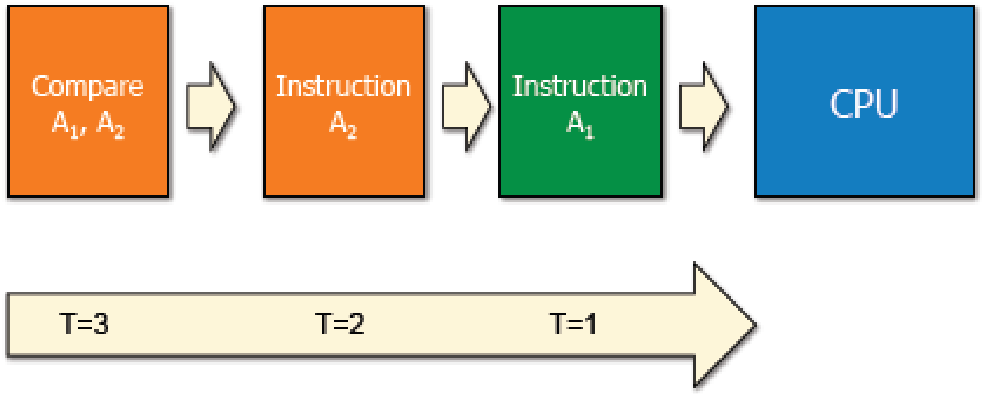
N to 3 switch selects 3 out of N primary outputs to be voted by the voter.

The configuration module detects the primary modules whose outputs differ from the actual one (Y).
Each faulty primary module is replaced by a spare one.

- Combines passive and active redundancies to achieve:
  - Error masking
  - Error correction
- Combining TMR with sparing we can have N modular redundancy with spares

Time Redundancy

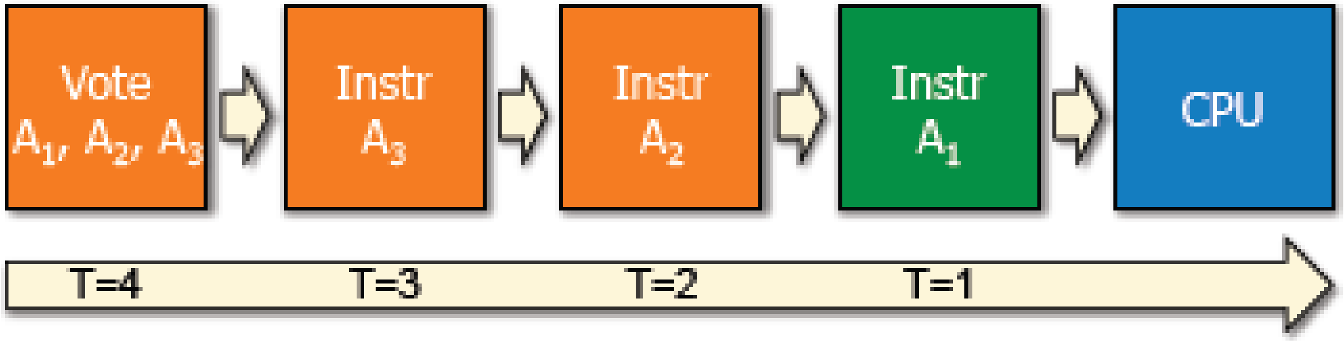
- More time than that needed for processing an input is used
- The additional time is devoted to detect and possibly correct errors occurred during the processing



- The same instruction is executed twice, and a comparison detects the occurrence of errors during the computations

Time Redundancy

- Error Correction:



- Cost:
  - Area: almost negligible
  - Time: ~3x for detection, ~4x for correction

Information Redundancy

- More data than those needed by the application are stored

- The *redundant data* added to the original data is used to detect and possibly *correct errors affecting the original data*
- The redundant data is a function of the original one

Purpose of information redundancy

- Vulnerable components
  - Channels
  - Processes (clients, servers)

- Security properties:
  - Authentication
  - Authorization
  - Confidentiality
  - Integrity
  - Availability

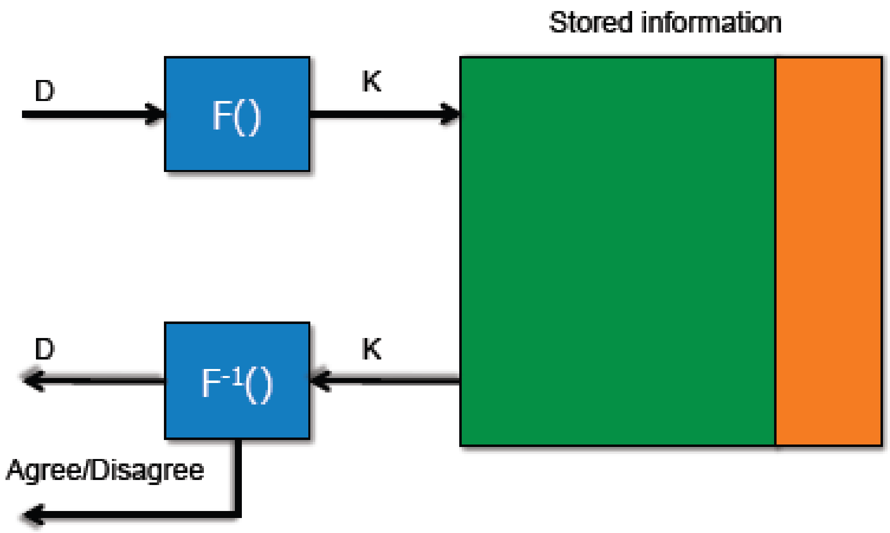
Cryptography

- Encoding operation: K=F(D)
  - D is the original data over N bits
  - F() is the encoding function
  - K is the encoded information, codeword, over M>N bits

- Decoding operation: $D=F^{-1}(K)$
  - $F^{-1}()$ is the decoding function

- In case there is some error during transmission then codeword K will be transformed to K* and therefore for each D, $K^* \neq F(D)$

Information redundancy



- Several redundancy schemes are possible that provides:
  - Error detection
  - Error correction
- Schemes differ in the type of error addressed:
  - Single error
  - Multiple error

Information redundancy

- Examples of information redundancy are:
  - Parity codes
  - Hamming codes
  - Reed Solomon codes

Parity Codes

- N data bits are coded in a N+1 codeword
  - The codeword is such that:
  - The number of ‘1’ in the codeword is even: even parity
  - The number of ‘1’ in the codeword is odd: odd parity
  - Any single error in the codeword can be detected but not corrected

References

1. Slides of Dr. Rana Asif Rehman & Dr. Haroon Mahmood
2. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.
3. Quinn, M. J. Parallel Programming in C with MPI and OpenMP, (2003).

Helpful Links:

1. https://mpitutorial.com/tutorials/mpi-send-and-receive/
2. http://boron.physics.metu.edu.tr/ozdogan/GraduateParallelComputing.old/week11/node2.html (Odd-Even Sort)
3. https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/