Philosophy

Parallel processing is the normal mode for most human activities.

Driving a car,
cooking a meal,
listening to a lecture

But in programming we have been led astray by history.

Von Neumann machines are sequential.

Parallel programming should be taught first!
Sequential programming should be taught as a special case of parallel programming.

Pervasiveness of Parallelism

1. Parallelism is a central conceptual issue across most of computer science.

2. Every subfield of computer science uses a different and parochial view of its problem domain.

3. The “wheel” is constantly being reinvented and restated with confusion and duplication of effort.

4. We will discuss parallelism from the viewpoint of programming but with connections to other domains.

Pervasiveness of Parallelism

Machine Architecture Oriented MOPC
- **SIMD, MIMD, etc.**

Theoretical Performance Oriented MOPC
- **PRAM, CREW, EREW, LogP, LogGP**

Theoretical-Practical Bridge MOPC
- **Bulk Synchronous Protocol - BSP**

Theoretical Analysis Oriented MOPC
- **Petri nets**

Memory Consistency MOPC
- **Strong Consistency, weak consistency**

Programming Oriented MOPC
- **Programming Languages - data flow, data partitioning MOPC, etc.**

Operating Systems - processes, threads, mutual exclusion

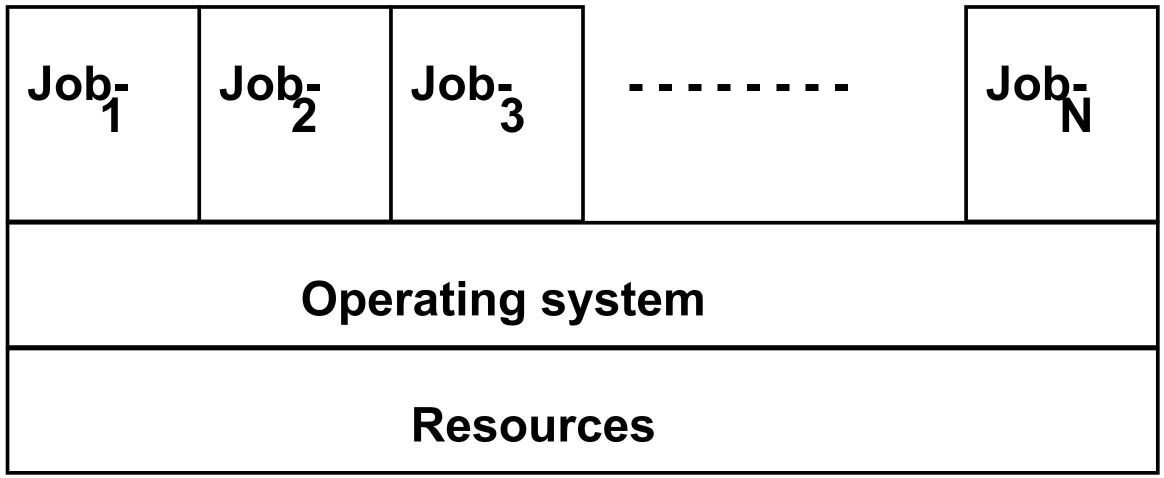
Data Base - transaction concurrency MOPC

AI - Parallel Production Rules, Neural Nets, Parallel Logic Programming

HISTORICAL SURVEY

1946 -	Vannevar Bush - Concept
1962 -	Petri - Model Of Parallel Computation
1963 -	Conway - Fork/join
1966 -	Bernstein - Single Assignment
1966 -	Karp & Miller - Model Of Parallel Computation
1960 -	Multiprogramming Operating Systems
1964 -	Array Processors
1975 -	Database Transaction Systems
1975 -	Scientific Computation
1972 -	Language Theory

Multi-programming operating system



OS must be conceptually parallel (or concurrent)

It has a thread of execution for each job and each active resource

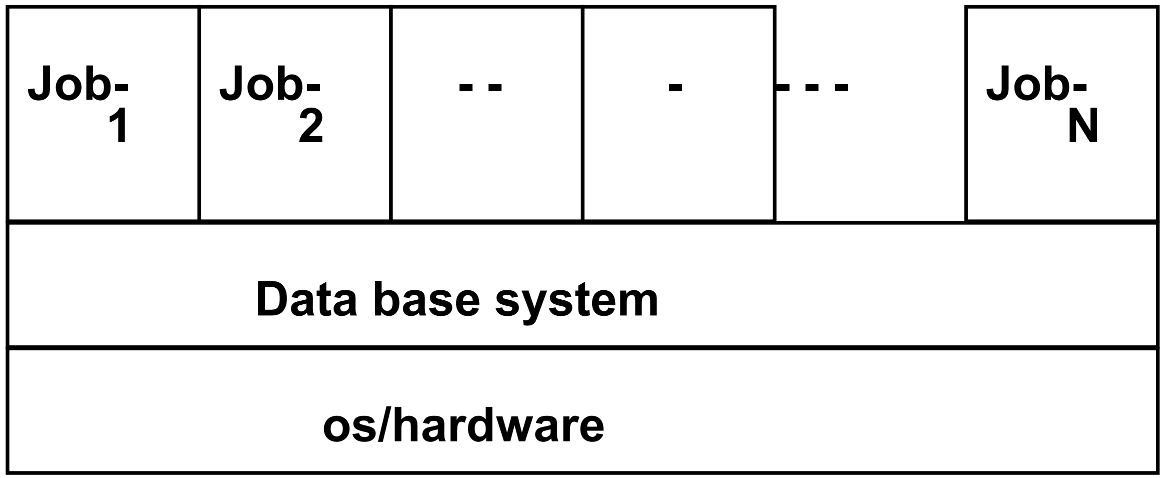
Requirements - partitioning of system state

- Consistency of shared state data under concurrent update
- Specification of action sequences

Concepts - processes

- Mutual exclusion
- Message systems

Database systems

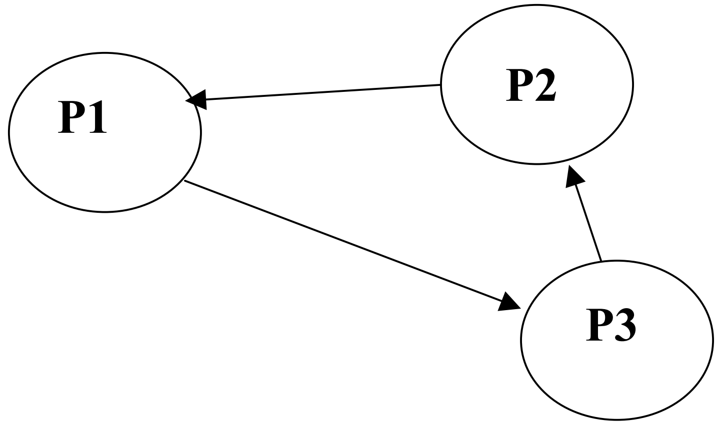


Requirement - updates to database must be consistent under concurrent update.

Concepts - transactions (indivisible actions)

Mechanisms - lock/unlock protocols
- commit protocols

DISTRIBUTED SYSTEMS



Requirement - Management Of Distributed And Partitioned System State

Concepts - Partitioned State Management Algorithms

- Forced Recognition Of Issues

Mechanisms - Versioning Of Data
- Distributed Algorithms

Approach

From
Models of Parallel Computation
to
Methods of Formulation
to
Parallel Programming Languages
to
Programs and Debugging
To
Architectures and Executions

What is a computation?

1. A Computation is a specification for transformation of one instance set of data structures or objects to some other instance set of data structures or objects.
2. A computation is a process which leads to the satisfaction of a set of constraint specifications.
3. A computation is an ordered sequence of execution of a (possibly dynamic) set of primitive units of computation.
4. Computation is a recursive and hierarchical concept. A computation may be primitive in one view and composed in another view.

Parallel Computations - Semi-Formal Model

Computation = (V,S,T,O)

V = set of variables

S = Set of states or set of assignments of values to variables.

T = Set of transformations on members of V to transition among members of S.

O = Ordering relation on the application of t in T to v in V to generate s in S.

O is the subject of study of this course.

There are concepts for specification of orderings.

There are mechanisms for specification of orderings

There are representations of concepts and mechanisms for ordering.

Ordering Relations for Sequential Programs

- a) next - positional
- b) or - if () then {} else {}
- c) one of - case (i): 1{};2{};
- d) ordered list - for i =1, n {}
- e) members of a set - while () {};

Ordering Relations for Parallel Programs

- a) after - <
- b) in parallel - ||
- c) mutual exclusion - <>

Let $c_1 = t_1 : v_1$ - be a computation - application of an operator to a set of variables.

The simplest sequential ordering is uses only positional ordering

$$c_1, c_2 \ldots\ldots\ldots\ldots c_n$$

Let us look at some possible parallel orderings

$$c_1 < [c_2 \parallel c_3 \parallel c_4] < c_5 \ldots\ldots\ldots\ldots c_n$$

c_1 must precede c_2 , c_3 and c_4 , c_2 , c_3 and c_4 can be executed in any order but all must complete before c_5 is begun.

The following are all correct sequential executions:

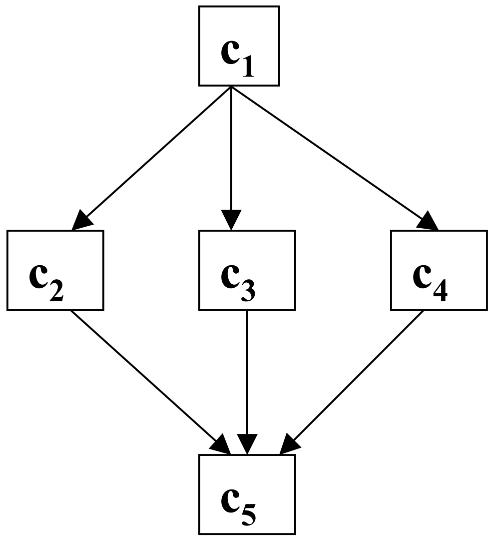
$$c_1 < c_2 < c_3 < c_4 < c_5$$

$$c_1 < c_3 < c_2 < c_4 < c_5$$

$$c_1 < c_4 < c_3 < c_2 < c_5$$

………………………………..

Graphical Specification of Parallel Computations



c_1 => x = 2;
c_2 => y= x;
c_3 => z = x;
c_4 => w = x;
c_5 => v = y+z+w;

Let us represent the ordering relations as a directed graph where the arrows specify precedence of execution among the nodes.

Let us assume that c_1 generates an output needed by each of c_2, c_3 and c_4 and c_5 requires input from each of its immediate predecessors in the graph.

Let us then specify that the arcs connecting the nodes carrying the output of the execution of the computation done at the node.

What else do we have to specify at each node?

Local versus Distributed Execution

1. The ordering specifications say nothing about where the computations are executed or how the output-input relations among the computations are implemented.

2. Therefore this type of specification of parallel execution is independent of the execution environment of the program.

3. If we could compile this type of specification to an efficient parallel implementation then parallel programming would be little more difficult than sequential programming.

4. Even though compilation of these declarative specifications of orderings to efficient parallel programs is possible there is little use of these declarative representations.

5. Attention is still focused on procedural mechanisms for implementation of ordering relations by procedural specification of communication or synchronization.

Models of Parallel Computation

Ordering Relation \ Name Space	Single	Multiple	
Sequential	Sequential Composition.	Remote Procedural Calls	
Parallel	Synchronization Operators	Communication Operators - Messages, etc.	

Implementation Mechanisms for Single and Multiple Name Spaces.

Let us examine the execution behavior of Program P which consists of a main program M which during its first execution, executes procedures P, Q, R and S in that order.

M is resident on computer A

P is resident on computer B

Q is resident on computer C

R and S are resident on computer D.

The second time M is executed the procedures are executed in the order P, R, Q and S. The answer is correct both times.

Is P a sequential program or a parallel program?

What can you infer about the relationship among procedures R and Q?

Parallel Programming systems are most often discussed or classified not by the ordering mechanism which is used but rather by the execution environment (shared or distributed memory) or by the mechanisms used to formulate the program (data parallelism, pipeline parallelism or task parallelism) or by the mechanisms for implementation of ordering (wait and signals or messages.)

We will (reluctantly) follow conventional practice and discuss parallel programming in terms of the conventional classifications.

COMPUTATION OF BINOMIAL COEFFICIENTS

Let C[n,k] be array of binomial coefficients

ALGORITHM

1. **INITIALIZE C[I,0] AND C[I,I] TO 1 FOR 0 <= I <= N**

2. **COMPUTE**

   C[K,I], 2 <= K <= N, 1 <= I <= K-1

   FROM

   C[K,I] := C[K-1,I-1] + C[K-1,I]

PROGRAM

```
      DO 10 I = 0,N
10    C[I,0] := C[I,I] := 1
      DO 11 K = 2,N
      DO 12 I = 1,K-1
12    C[K,I] := C[K-1,I-1] + C[K-1,I]
11    CONTINUE
```

Specification of Execution Sequences

Strict or Partial Orders
Sequencing by output/input dependencies
- **Implementation depends on name space specifications**
  - **partitioned - messages - data flow**
  - **shared - synchronized access**
  - **call/return**

Sequence by Schedule
- **Sequential**
- **Priority**

Sequence by functional dependencies
- **functional and logic languages**

Sequence by data dependencies
Sequence by logical expression
- **first order linear logic of events**

Properties of Execution Schedules for Parallel Computations

1. There are often many valid sequences.
2. There exist some sequences which result in minimal computation.
   - event specification
   - "data flow" specification
   - sequential specification

A sequence which results in a correct transformation with minimum number of operations insures that when a computation is executed for the first time all of its dependence relations have been satisfied.

3. There exist correct sequences which do not result in minimum computation.

Partial Orders and "weak fairness" condition.

- A) No sequencing other than "weak fairness."
- B) Partial or "optimistic" sequence specifications

FIRING RULES OR GUARDS

(1) Define for each UC a predicate (firing rule or guard) which is a function of the state of the computation

$$\{t,f\} \equiv F(S)$$

(2) The UC may execute whenever

$$F(S) \Rightarrow t$$

(3) If F and S are "complete" for each UC a minimum schedule results

(4) if F and S are "optimistic" then the UC's must follow "commit" protocols to validate execution order to obtain a minimum schedule

Distributed databases - cost of assembly of "complete" S

Distributed/parallel simulations

COMPUTATION OF BINOMIAL COEFFICIENTS

LET THERE EXIST A VALUE "UNDEFINED"

```
COBEGIN (0<=I<=N, 0<=J<=N)
      C[I, J] : = "UNDEFINED"
COEND
COBEGIN (0<=I<=N)
      C[I, O]:=C[I, I] := 1
COEND
COBEGIN (2<=K<=N)
      COBEGIN (1<=I<=K-1)
            {IF(C[K-1, I-1] = "DEFINED"
                       AND
            (C[K-1, I] = "DEFINED")
        THEN C[K, I]:=C[K-1, I-1] + C [K-1, I]}
      COEND
COEND
```

COMPUTATION OF BINOMIAL COEFFICIENTS

LARGE GRANULARITY PARALLELISM

```
COBEGIN
         {DO 10 I = 0, N
     1 0 C[I, 0] :=1}
         {DO 11 I = O, N
     1 1 C[I, I] :=1}
COEND

COBEGIN (2<=K<=N)
     { IF C[K-1, K-1] = "DEFINED"
     THEN
     DO 12  I = 1, K-1
1 2 C[K, I] := C[K-1, I-1] + C[K-1, I]}
COEND
```

```
import java.lang.*;
public class BinomialWeakFairness
{public static void main(String[] args)
{int i,j,k,l;
Integer NBC = new Integer(args[0]);
int  nbc= NBC.intValue();
Integer IT = new Integer(args[1]);
int it = IT.intValue();
int [][] b= new int[100][100];
int [][] c= new int[100][100];
for (i = 0; i <=nbc;i = i+1)
        {b[i][0] = 1;
         b[i][i] = 1;}
for ( j = 1; j <=it; j = j+1)
        { RandomIntGenerator rk = new RandomIntGenerator(2,nbc);
        k = rk.draw();
        RandomIntGenerator rl = new RandomIntGenerator(1,k-1);
        l = rl.draw();
        b[k][l] = b[k-1][l] +b[k-1][l-1];
        }
```

If the order of execution is left totally unspecified (random selection)

Then attainment of specifications requires "weak fairness"

"Weak fairness"

The order of execution of UC's is chosen at random with the constraint that each UC will be executed infinitely often during the total execution of the computation

This is the end point of "no" specification of order

Intermediate points include "optimistic" protocols

Optimistic protocols

UC initiates execution without certain knowledge that a given execution is in a minimum schedule - validates sequence before emitting outputs. Example - Database transaction processing.

Approaches and Languages

Distributed Memory – MPI

Coordination Models – Linda and Associative Interactions

Shared Memory –OpenMP, Direct Threads Programming (Java or Posix)

Parallelizing Compilers and Dependence Relations

Graphical/Visual Programming

Internet/Grid Programming – Globus/RSL, Web Services