High Performance Data Mining

Ying Liu, Prof., Ph.D

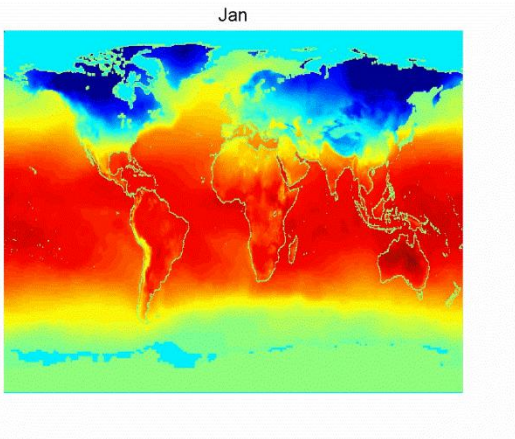
University of Chinese Academy of Sciences
Key Lab of Big Data Mining and Knowledge Management, Chinese Academy of Science

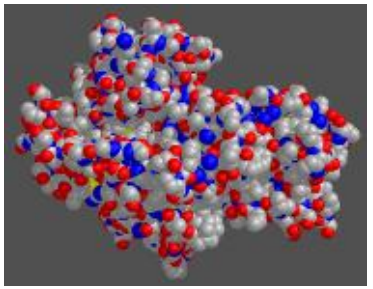
Outline

- Introduction to High Performance Computing
  - Parallel Computing Platforms
  - Parallel Programming
    - Shared Memory Parallel Programming
    - Distributed Memory Message-passing Parallel Programming
    - CUDA-enabled GPU
- Parallel Classification
- Parallel Association Rules Mining
- Parallel Clustering

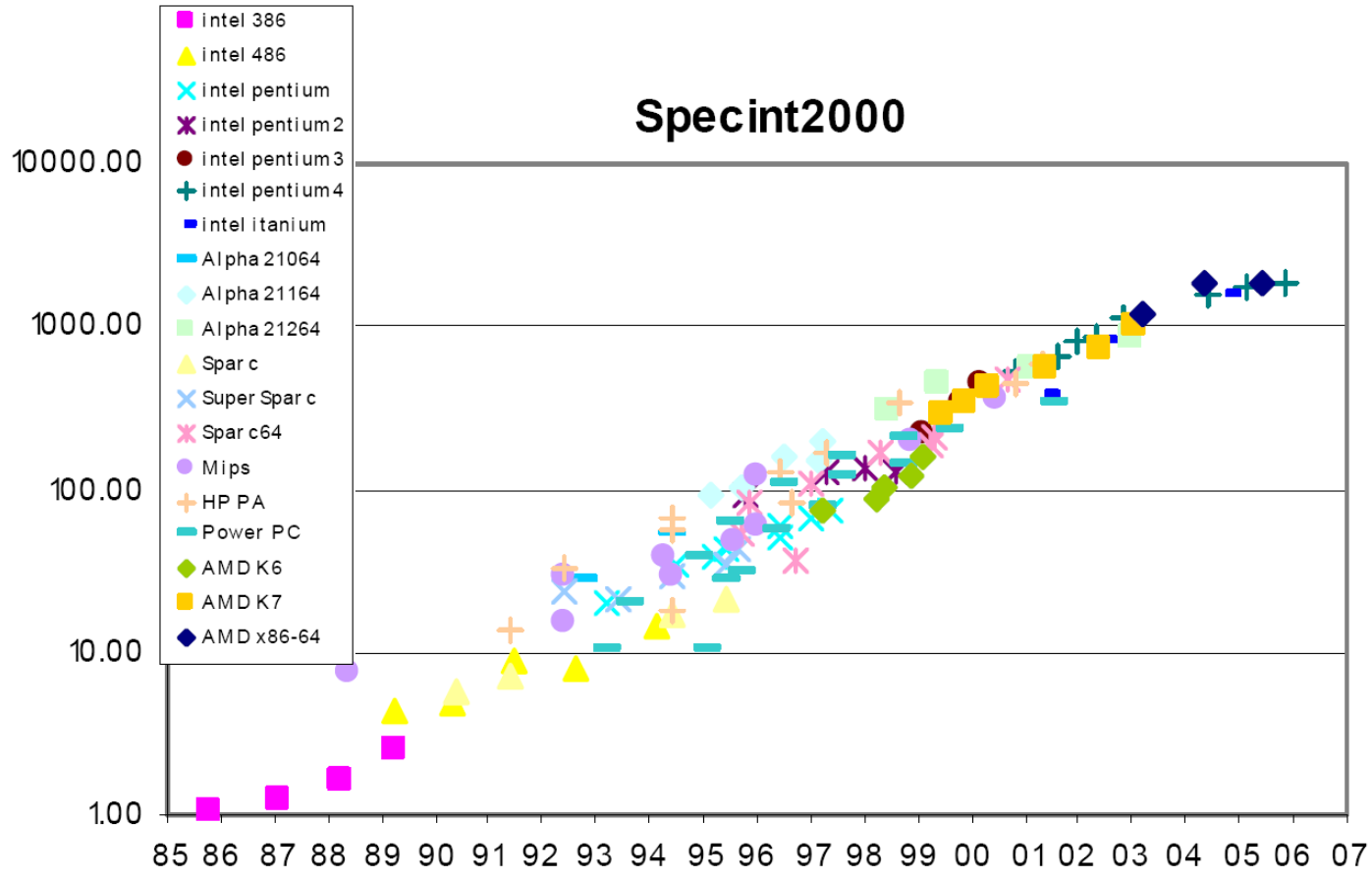
Motivation

- Lots of data being collected in commercial and scientific world, massive data sets
- Strong competitive pressure to extract and use the information from the data, e.g.
  - Climate simulation
  - Astrophysics
  - Molecular biology





Performance of Single-Core CPU



Motivation

- Computing power limits are being approached
- Memory limitations of sequential computers cause sequential algorithms to make multiple expensive I/O passes over data
- Lacking of capability to solve bigger and more realistic distributed applications

Accelerate the computation
Use more memory from multiple machines

Outline

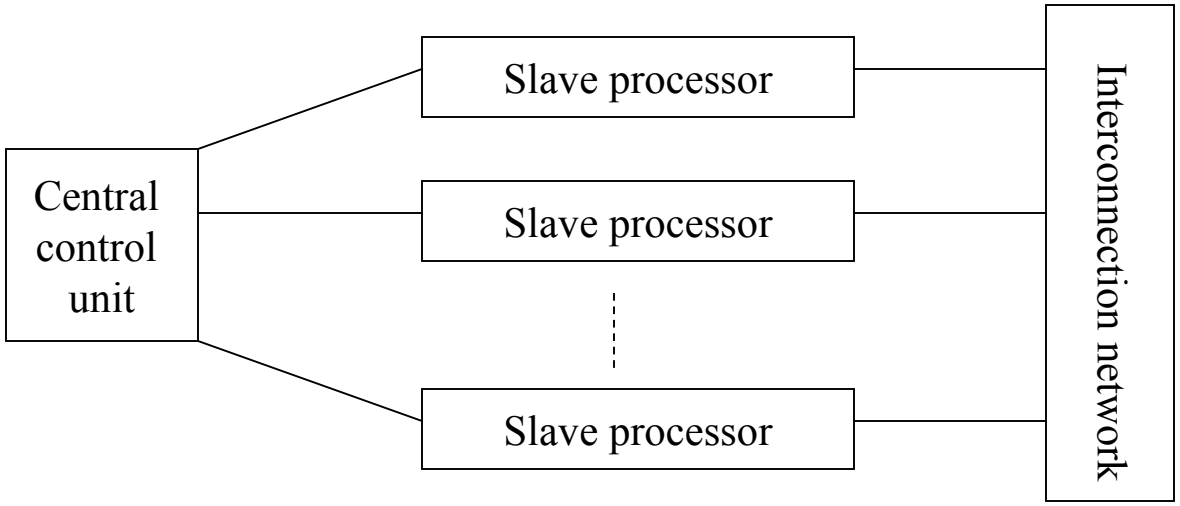
- Introduction to High Performance Computing
  - Parallel Computing Platforms
  - Parallel Programming Model
    - Shared Memory Parallel Programming
    - Distributed Memory Message-passing Parallel Programming
    - CUDA-enabled GPU
- Parallel Classification
- Parallel Association Rules Mining
- Parallel Clustering

Architectures

- Basic components of any architecture:
  - Processors and memory
  - Interconnect
- Logic classification based on:
  - Control mechanism
    - SISD
    - MISD
    - SIMD (Single Instruction Multiple Data stream)
    - MIMD (Multiple Instruction Multiple Data stream)
  - Address space organization
    - Shared Address Space
    - Distributed Address Space

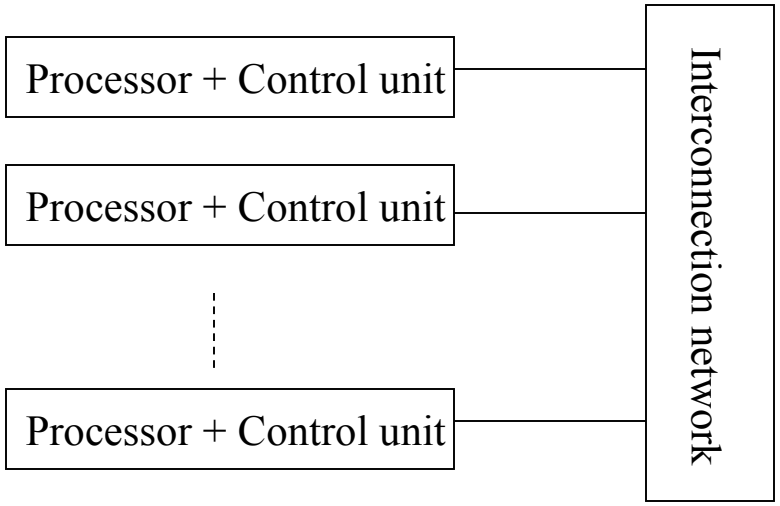
SIMD Architecture

- Multiple processors execute the same instruction
- Data that each processor sees may be different
- Individual processors can be turned on/off at each cycle (“masking”)
- Examples: Illiac IV, Thinking Machines’CM-2, DAP, …
- Specialized architecture limits the popularity



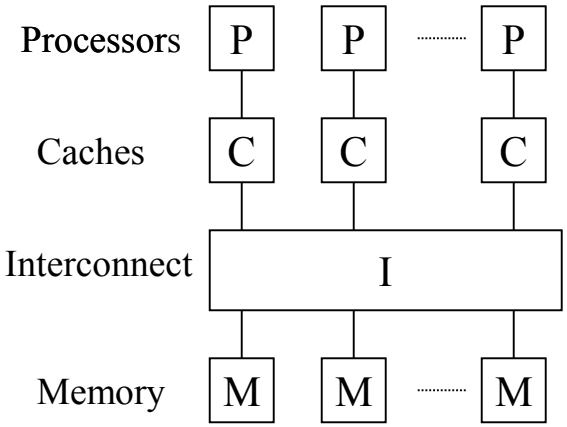
MIMD Architecture

- Each processor executes program independent of other processors
- Processors operate on separate data streams
- May have separate clocks
- Examples: IBM SP, Cray T3D & T3E, SGI Origin, Clusters, Sun Ultra Servers, …

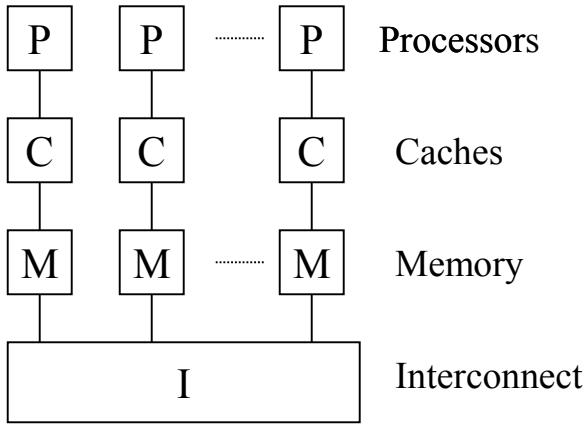


Shared Address Space

- All processors share a single global address space
- Single address space facilitates a simple programming model
- Examples: SGI Origin 2000, IBM SP2



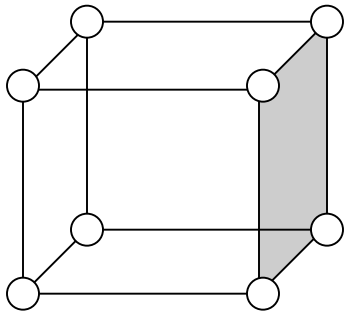
(a) UMA

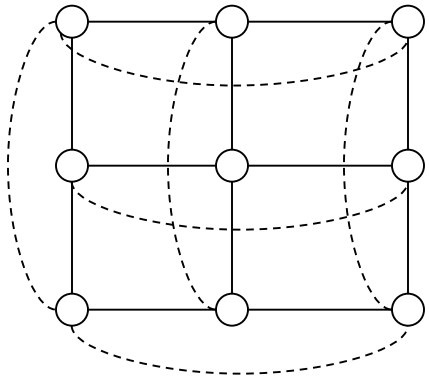


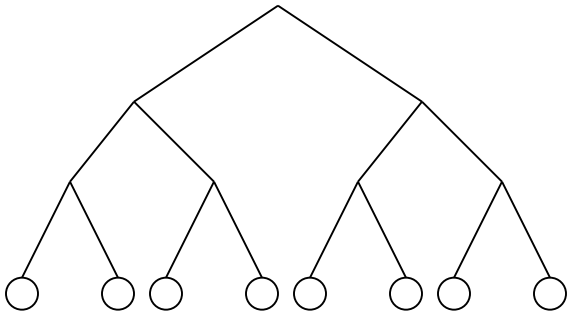
(b) NUMA

Static Interconnects

- Consist of point-to-point links between processors
- Examples: hypercube, mesh/torus, fat tree
- Two major characteristics:
  - § Can make parallel system expansion easy
  - § Some processors may be “closer” than others

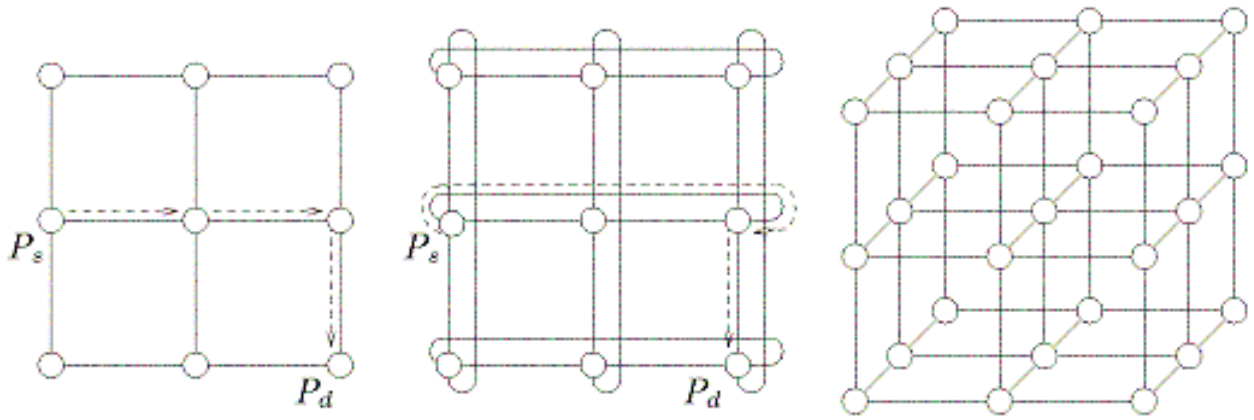






Multidimensional Meshes

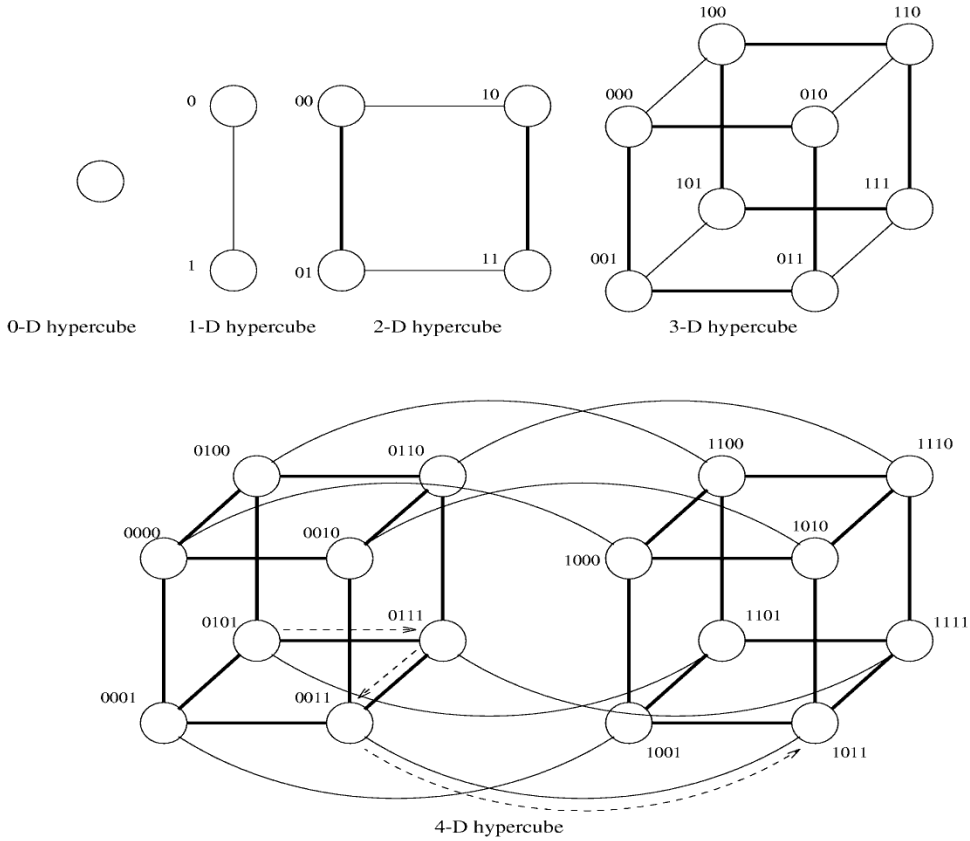
- Each processor in a *d*-dimensional mesh is connected to *2d* other processors
- Data between processors routed via intermediate processors
- In practice, only 2 or 3 dimensional meshes are constructed
- Mesh with wrap around: Torus



(a) (b) (c)

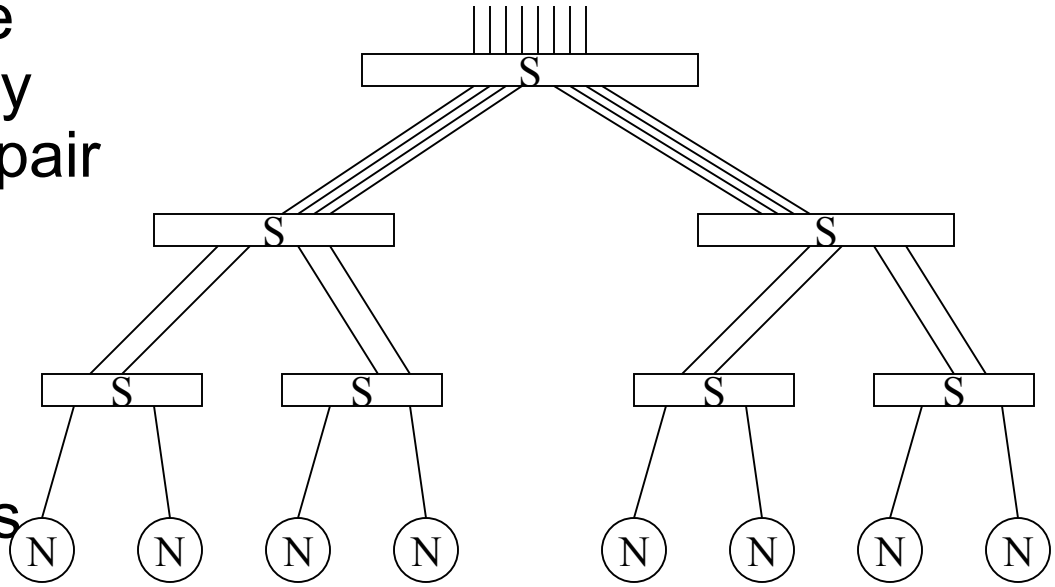
Hypercube

- A hypercube is a multidimensional mesh with exactly 2 processors in each dimension
- In a *d*-dimensional hypercube, each processor is connected with *d* other processors



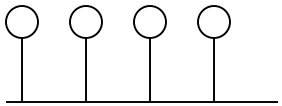
Fat Tree

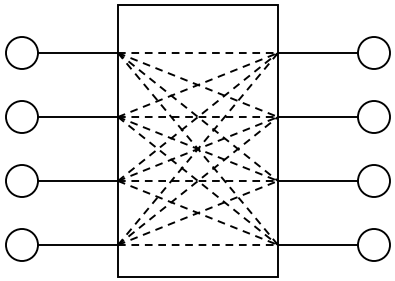
- A tree network is one where there is exactly one path between a pair of processors
- In a simple tree, the bandwidth on higher level links is shared among all processors below creating bottlenecks
- A fat tree solves the problem by using wider links at higher levels in the tree

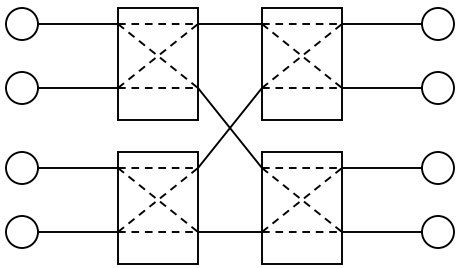


Dynamic Interconnects

- Paths are established as needed between processors
- Examples: bus based, crossbar, multistage networks
- Two major characteristics:
  - System expansion difficult
  - Processors are usually equidistant

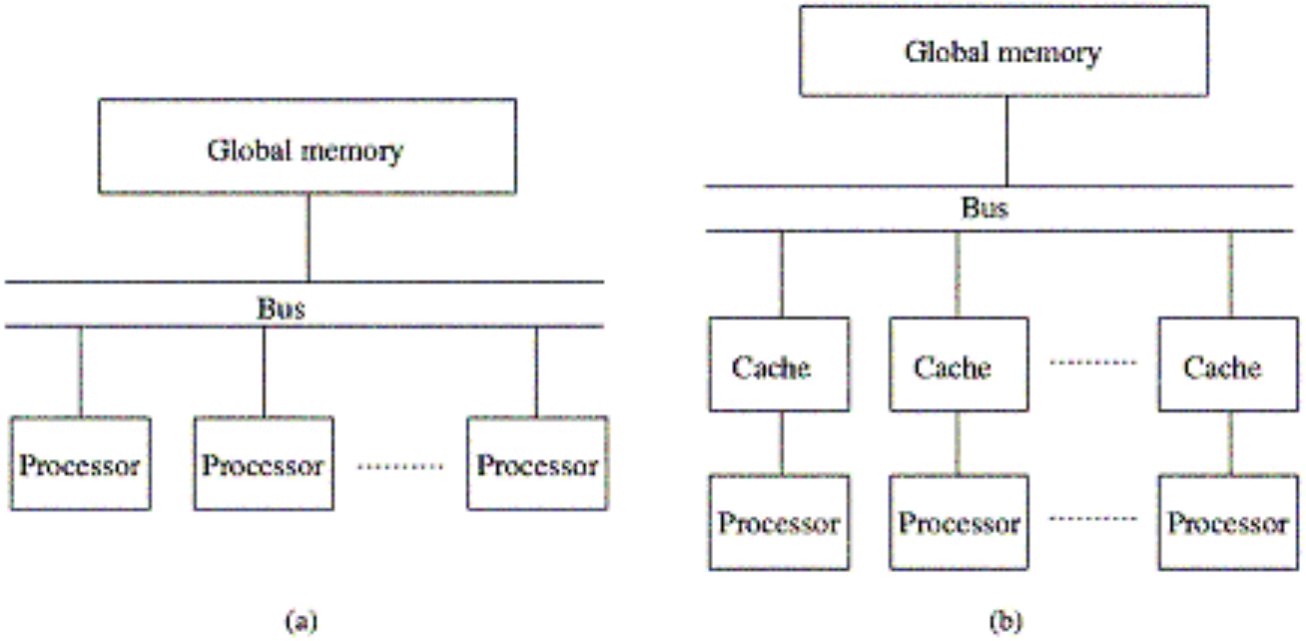






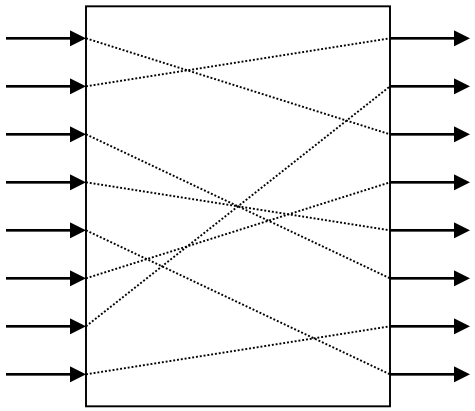
Bus Systems

- Only one pair of processors or processor and memory can exchange data at a time
- Shared medium, good for broadcasting
- Bandwidth limitation, limited to dozens of nodes



Crossbars

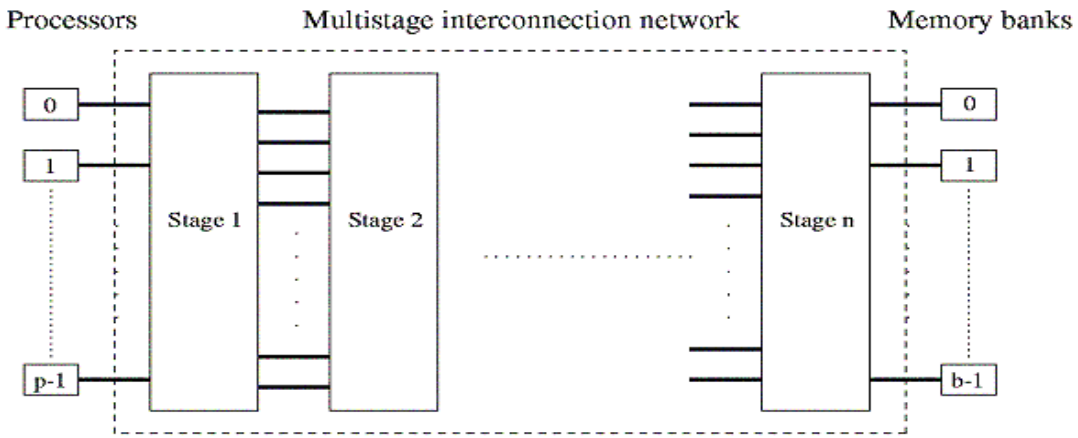
- Designed to increase the traffic between processors and memory
- Given N processors and an N-way interleaved memory, an N*N crossbar provides N simultaneous connections
- Can control crossbar settings to achieve any desired permutation



An 8*8 crossbar switch (8!=40320 permutations)

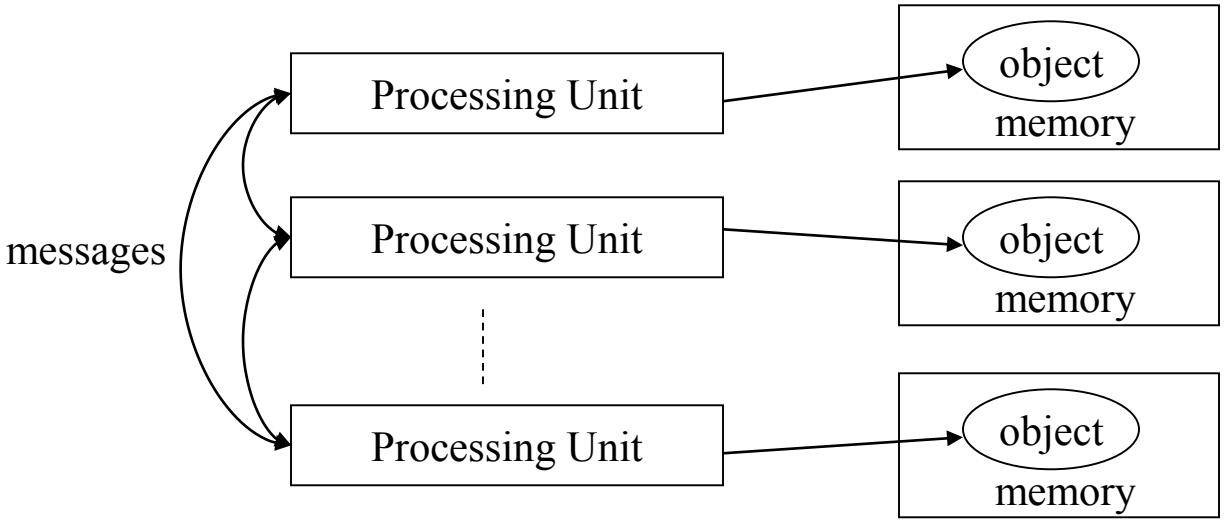
Multistage Network Systems

- Multistage networks use *Nlog(N)* switches
- Use multiple stages of switches to build interconnects
- Design choices in multistage networks:
  - Basic switch type and size
  - Number of stages
  - Connections between stages



Message Passing Platform (Distributed Address Space)

- "Shared nothing:" each processor has a private memory
- Processors can directly access only local data
- Each processing unit can be single processor or a multiprocessor
- Interaction between processors relies on message passing
- Example: clusters

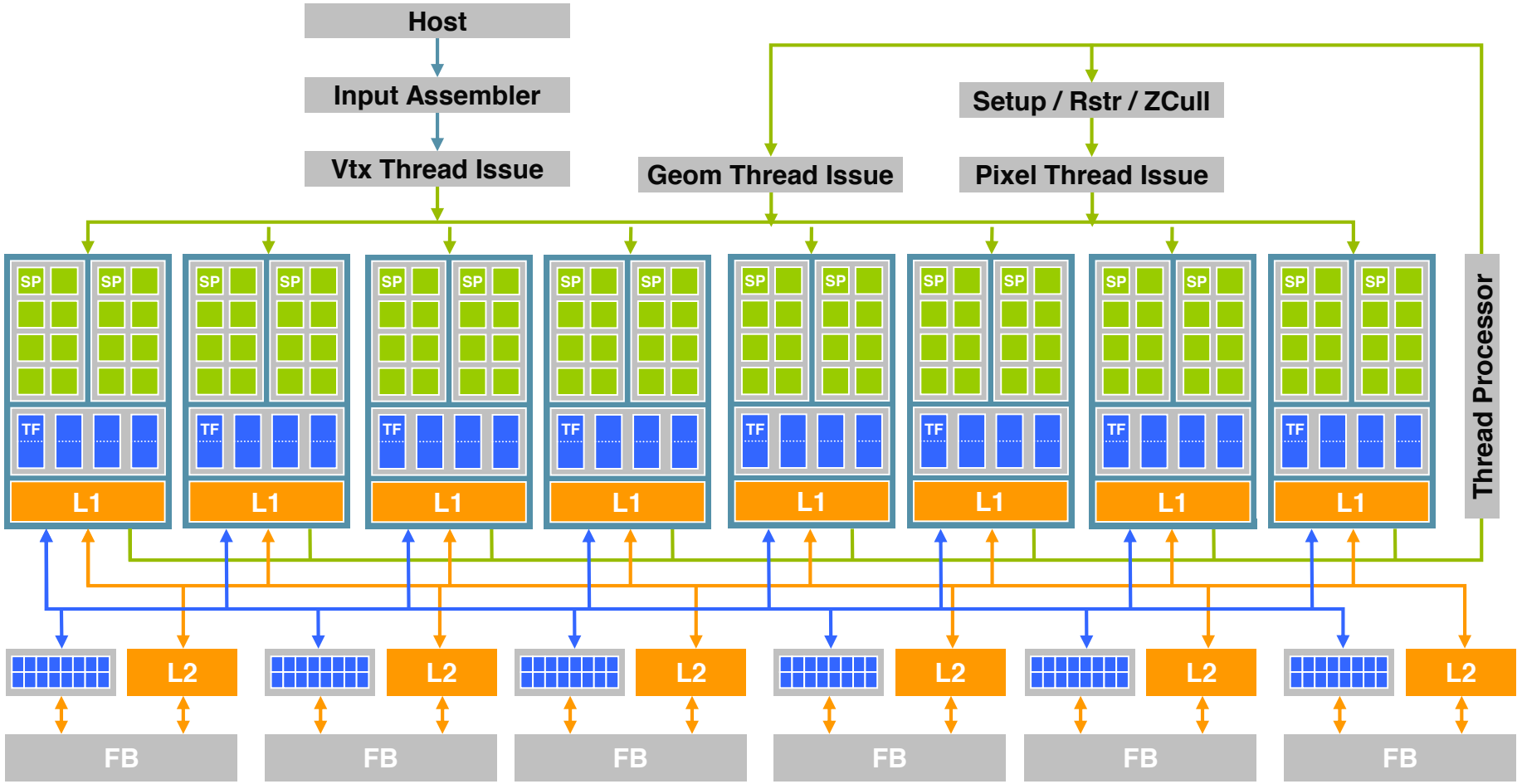


GPU Design Philosophy

- Difference between GPU and CPU
  - More transistors for data processing
  - Many-core (hundreds of cores)

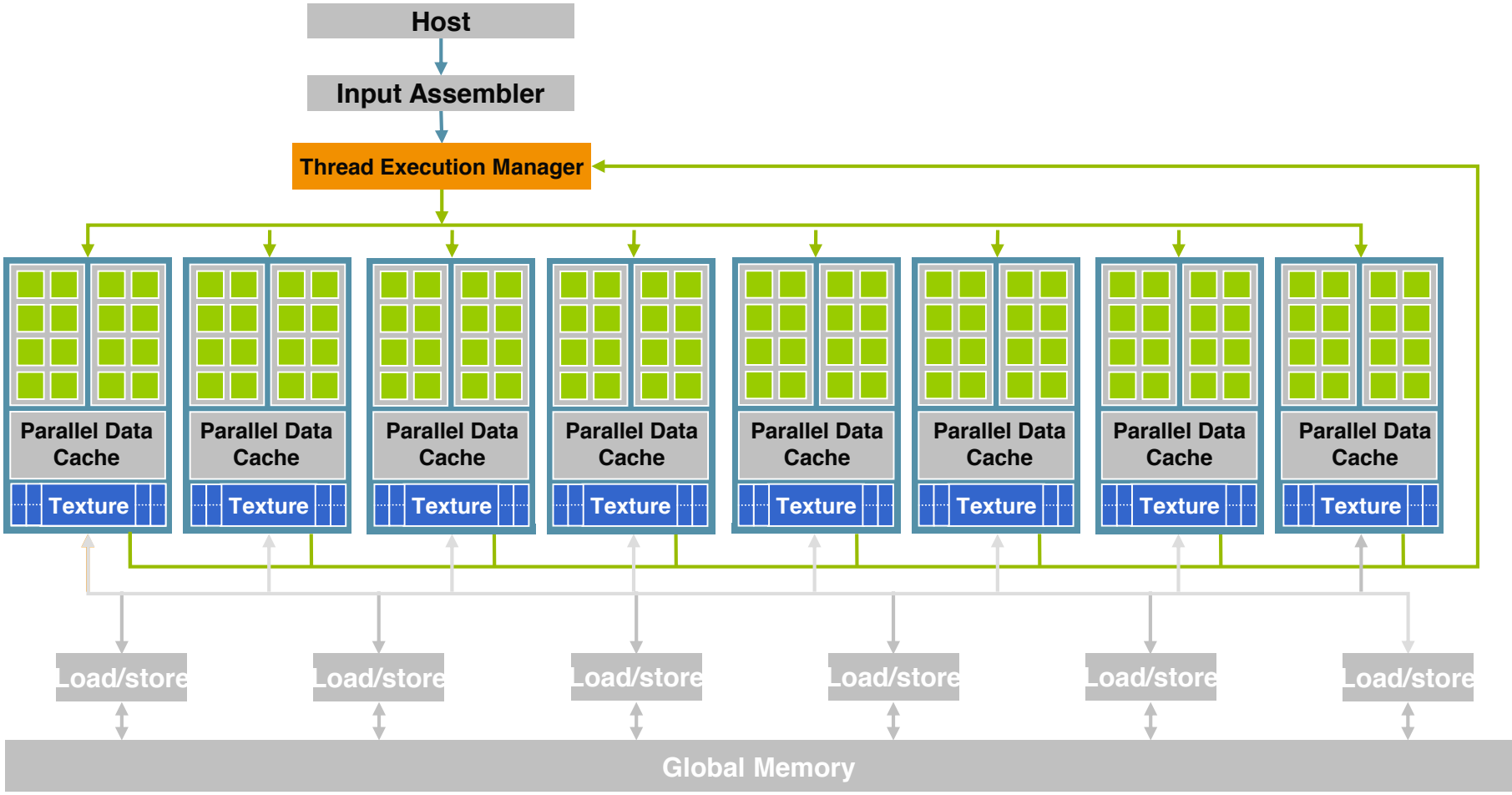
Nvidia's GeForce 8800

G80 – Graphics Mode



Block Diagram of the GeForce 8800

G80 CUDA Mode



TOP 10 Machines 11/2016

Rank	Name	Site	System	#core	R_{max} TF/s	Arch.	Country
1	Sunway TaihuLight	National Supercomputing Center in Wuxi	Sunway SW26010 260C	10,649,600	93,014.6	MPP	China
2	Tianhe-2	National Super Computer Center in Guangzhou	NUDT Cluster, Intel Xeon Phi 31S1P	3120000	33862.7	cluster	China
3	Titan	DOE/SC/Oak Ridge National Laboratory	Cray XK7, Nvidia K20x	560640	17590.0	MPP	USA
4	Sequoia	DOE/NNSA/LLNL	BlueGene/Q	1572864	17173.2	MPP	USA
5	Cori	DOE/SC/LBNL/NERSC	Cray XC40, Intel Xeon Phi	622,336	14,014.7	MPP	USA

TOP 10 Machines 11/2016

Rank	Name	Site	System	#core	R_{max} TF/s	Arch.	Country
6	Oakforest-PACS	Joint Center for Advanced High Performance Computing	PRIMERGY CX1640 M1, Intel Xeon Phi ,Fujitsu	556,104	13,554.6	Cluster	Japan
7	K	RIKEN Advanced Institute for Computational Science (AICS)	SPARC64 VIIIfx	705024	10510	Cluster	Japan
8	Piz Daint	Swiss National Supercomputing Centre	Cray XC50, Nvidia Tesla P100	206,720	9,779.0	MPP	Switzerland
9	Mira	DOE/SC/Argonne National Laboratory	BlueGene/Q	786432	8586.6	MPP	USA
10	Trinity	DOE/NNSA/LANL/SNL	Cray XC40, Xeon E5	301,056	8,100.9	MPP	USA

Operating Systems

- Need to support tasks similar to serial OS like UNIX
  - Memory and process management
  - File system
  - Security
- Additional support needed
  - Job scheduling: assign tasks to idle processors, time sharing, space sharing
  - Parallel programming support: message passing, synchronization

Outline

- Introduction to High Performance Computing
  - Parallel Computing Platforms
  - Parallel Programming Model
    - Shared Memory Parallel Programming
    - Distributed Memory Message-passing Parallel Programming
    - CUDA-enabled GPU
- Parallel Classification
- Parallel Association Rules Mining
- Parallel Clustering

What is Parallel Programming?

- Parallel programming involves constructing or modifying a program for solving a given problem on a parallel machine
  - Start from a serial algorithm for solving the problem
- Goals of parallel programming – Performance!
- Effective parallel programming
  - *Key1:* Minimization of inter-processor synchronization costs
  - *Key 2*: Equal workload between processors

Alternatives to Parallel Programming

- Automatic parallelizing compiler
- Libraries
- Implicit parallel programming:
  - Programmer directives to help compiler analysis
  - Example OpenMP
- Explicit parallel programming
  - Easiest to support
  - Example MPI

Types of Parallelism

- Data parallelism:
  - Partition the data across processors
  - Each processor performs the same operations on its local data partitioning
- Task parallelism:
  - Assign independent modules to different processors
  - Each processor performs different operations

Data Dependence & Parallelization

- Consider the following loop of a C program

```
for (i=0;i<1000;i++)
  a[i]=b[i]+c[i]
```

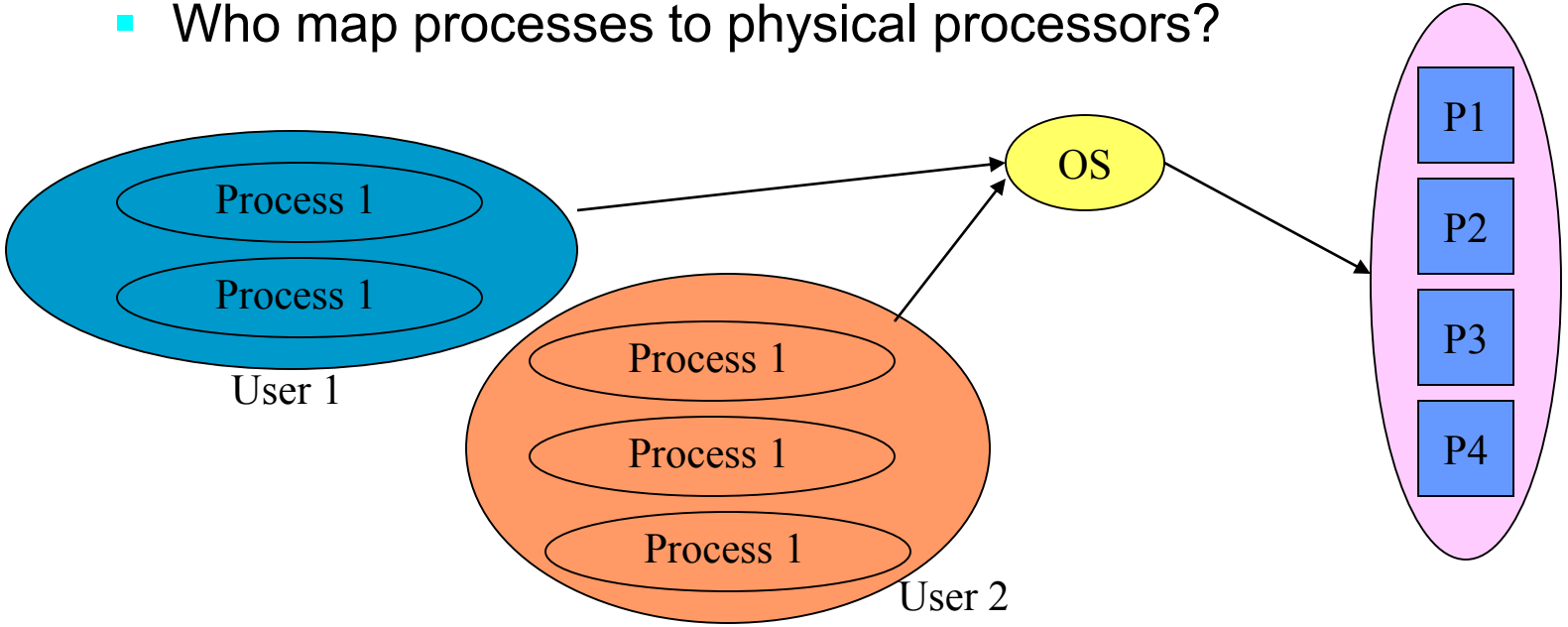
- If one unfolds the loops, the statements would be executed as follows:

```
a[0]=b[0]+c[0];
a[1]=b[1]+c[1];
…..
a[999]=b[999]+c[999];
```

- Can each iteration be executed in parallel?

Processes

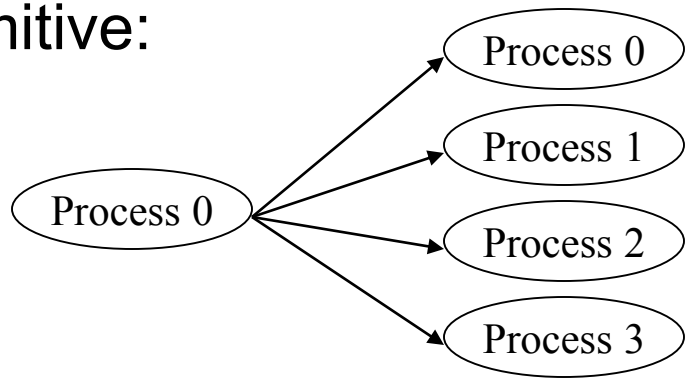
- A process is a program along with all of its environment
  - How to communicate?
  - Who creates it?
  - Who map processes to physical processors?



Process Creation

- Use a process creation primitive:

```
m_set_procs(nproc);
m_fork(func,[args]);
```



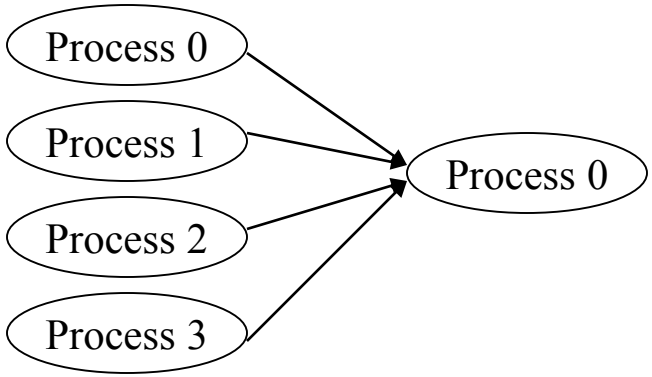
- Each child process is an exact copy of its parent
- All the variables associated with the parent are private for the parent and each child unless explicitly made shared
- The parent had ID=0, children share the other *nproc-1* IDs

```
id=m_get_myid();
nprocs=m_get_procs();
```

Process Destruction

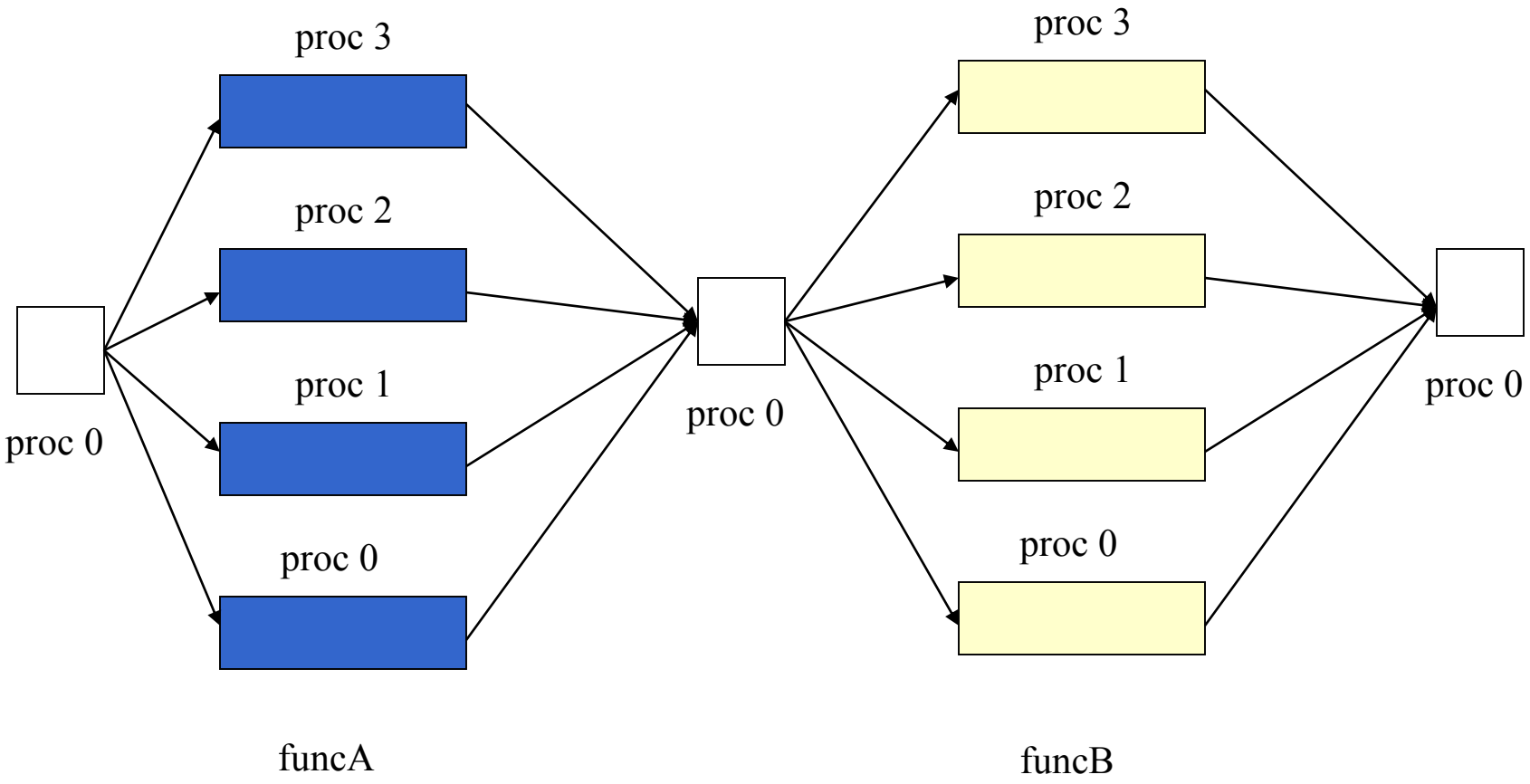
- All child processes are terminated; only the parent remains
- The parent waits for all child processes to terminated before returning
- Use a process destruction primitive

m_kill_procs();

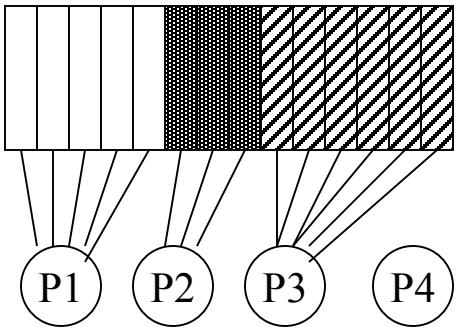


Parallel Program Model

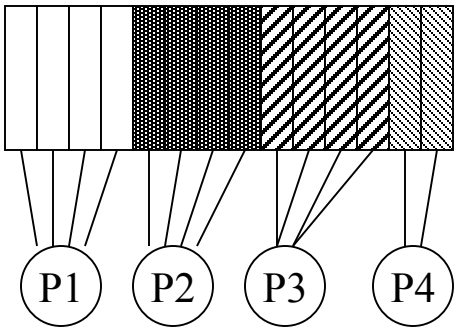
- Processes executing in a data parallel fashion



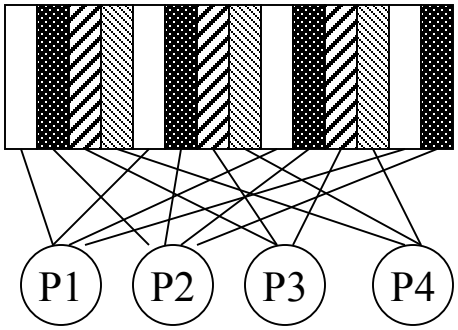
Loop Scheduling



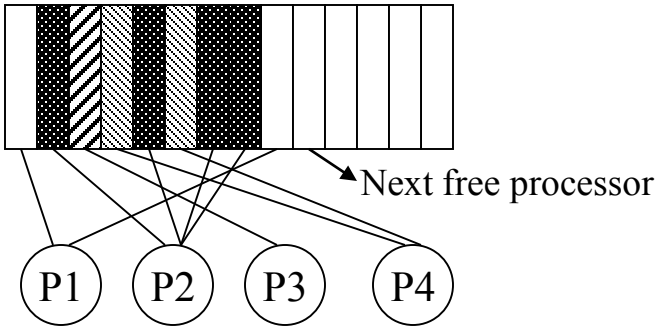
Prescheduling



Static blocked



Static interleaved



Dynamic

Prescheduling

- Assign work using a schedule array

```
float *a, *b, *c;
int low[ ] = {0,50,80}
int high[ ] = {50,80,120}

void parallel_func( ) {
        int id, i;
        id = m_get_myid( );
        for (i=low[id]; i<high[id]; i++)
              a[i] = b[i] + c[i];
}
```

Static Blocked Scheduling

- Assign a contiguous chunk of iterations based on process ID

```
float *a, *b, *c;
void parallel_func( ) {
    int id, i, nprocs;
    int low, high;
    id = m_get_myid( );
    nprocs = m_get_numprocs( );
    low = id*100 / nprocs;
    high = (id+1)*100 / nprocs;
    for (i=low; i<high; i++)
        a[i] = b[i] + c[i];
} /* end of parallel_func */
```

Static Interleaved Scheduling

- Assign iterations in a round-robin way based on process ID

```
float *a, *b, *c;
void parallel_func( ) {
    int id, i, nprocs;
    id = m_get_myid( );
    nprocs = m_get_numprocs( );
    for (i=id; i<100; i+=nprocs)
        a[i] = b[i] + c[i];
}
```

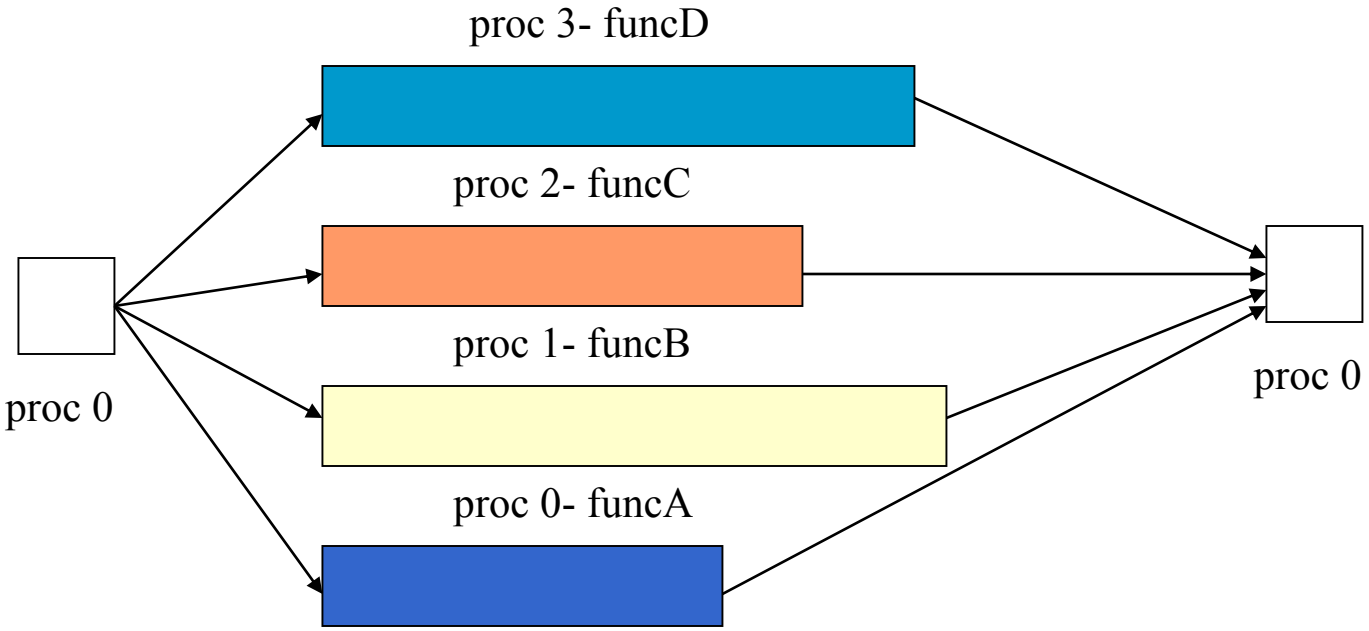
Dynamic Scheduling

- Self scheduling: processes execute iterations using a shared counter

```
float *a, *b, *c;
int i;

i = 0;

void parallel_func( ) {
    int i1, i2;
    int chunk=2;

    while (i1<100) {
        m_lock( );
            i1 = i;
            i += chunk;
        m_unlock( );
        for (i2=i1; i2<min(i1+chunk,100); i2++)
            a[i2] = b[i2] + c[i2];
    } /* end of while */
} /* end of parallel_func */
```

Parallel Program Model

- Processes executing in a function/task parallel fashion

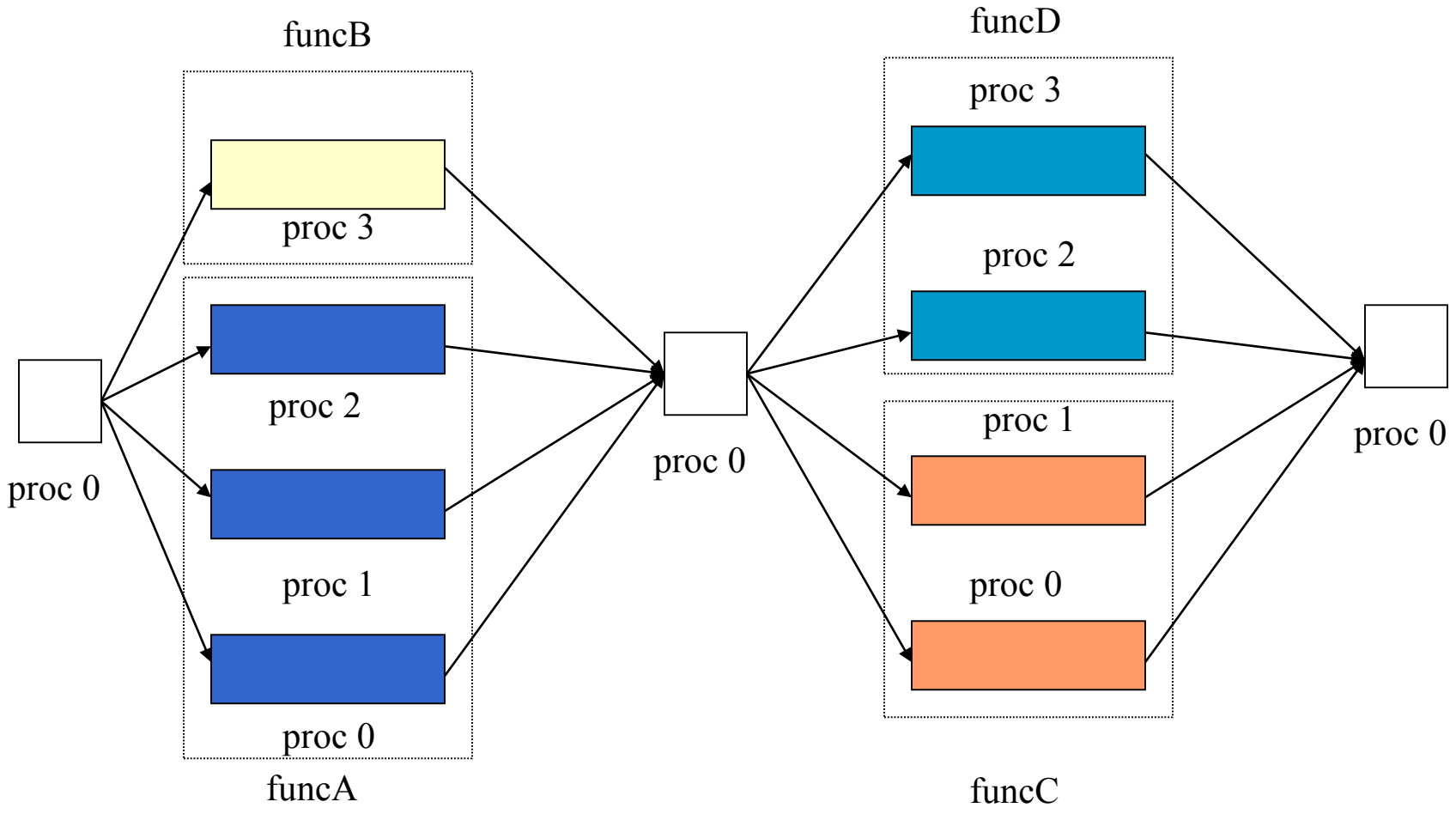


Parallel Program Model

```
main() {
        m_set_procs(4);
        m_fork(func);
        m_kill_procs();
}
void func() {
Int id;
    id = m_get_myid();
    switch (id) {
        case 0: funcA();
                break;
        case 2: funcB();
                break;
        case 2: funcC();
                break;
        case 3: funcD();
                break;
    }
}
```

Parallel Program Model

- Processes executing in a function & data parallel fashion



Parallel Program Model

```
main() {
    m_set_procs(4);
    m_fork(func1);
    m_park_procs();
          …..
     m_rele_procs();
     func2();
     m_kill_procs();
}
```

```
void func1() {
    switch (id) {
        case 0,1,2: funcA();
                    break;
        case 3: funcB();
                break;
    }
}
void func2() {
    switch (id) {
        case 0,1: funcC();
                  break;
        case 2,3: funcD();
                  break;
    }
}
```

Performance of Parallel Programs

- T = execution time of the best serial algorithm
- T_p = execution time of the parallel algorithm with p processors
- Speedup, $S_p = T / T_p$
- Efficiency $E_p = S_p / p$
- However, E_p is realistically never 100% due to:
  - Synchronization and communication cost across processors
  - Less-than-perfect load balancing among processors
  - Assume also that α represents the fraction that cannot be parallelized, i.e., the serial fraction
  - Then, using p processors in parallel and assuming perfect speedups in the parallelized portion, the parallel execution time,

$$T_p = T (\alpha + (1 - \alpha) / p)$$

Example

```
float sum,sum0,sum1;
main() {
    m_set_procs(2);
    m_fork(parallel_func);
    m_kill_procs();
    sum=sum0+sum1;
    printf("total sum is %lf\n",sum);
}
void parallel_func(){
    int id;
    id=m_get_myid();

    if (id==0) sum0=1.0+2.0;
    else sum1=3.0+4.0;
}
```

Example

```
float total_sum;
main() {
    total_sum=0.0;
    m_set_procs(2);
    m_fork(parallel_func);
    m_kill_procs();
    printf("total sum is %f\n",total_sum);
}
void parallel_func() {
    int id;
    float  partial_sum;
     id=m_get_myid();

    if (id==0) partial_sum=1.0+2.0;
    else partial_sum=3.0+4.0;

    total_sum+=partial_sum;
}
```

Example

Possible outputs?

Total sum is 10.0
(ts = 0;
P0: ts = ts(0) + 3;
P1: ts = ts(3) + 7;

Total sum is 7.0
(ts = 0;
P0: ts = ts(0) + 3;
P1: ts = ts(0) + 7;

Total sum is 3.0
(ts = 0;
P1: ts = ts(0) + 7;
P0: ts = ts(0) + 3;

Contention

- Random process scheduling creates problems with respect to modifying shared variables
- The final values of a shared variable can differ from run to run depending on the order in which it was modified
- Root of this contention problem is the simultaneous accessing of a shared variable
- Solution ?

Restrict the access to shared variables

Variable Locks

- Locks are used to provide exclusive access for the modification of a variable

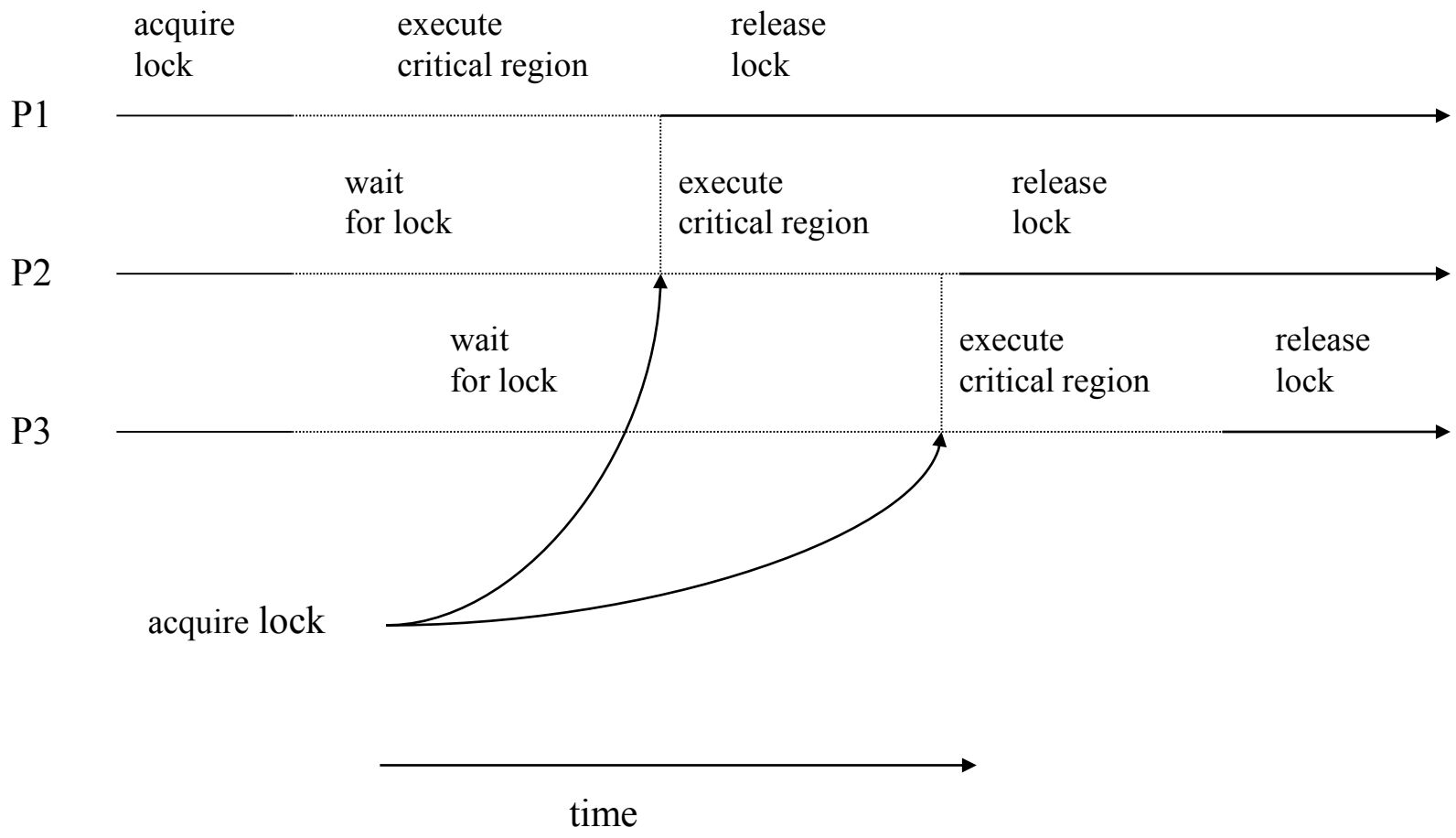
- A lock variable is to be created for every shared variable that needs exclusive access:
  - It must be shared among all processes that need to use it
  - It can either be in a “locked” or “unlocked” state

Variable Locks

- The steps to be followed for exclusive access using locks are:
  - Acquire the lock for a variable
  - Modify the variable
  - Release the lock for the variable

- The use of locks sequentializes execution of program sections and hurts performance

Illustration of Locks



Example

```
float total_sum;
void parallel_func() {
   int id;
   float  partial_sum;
   id=m_get_myid();

   if (id==0) partial_sum=1.0+2.0;
   else partial_sum=3.0+4.0;

   m_lock();
   total_sum+=partial_sum;
   m_unlock();
}
```

Race Conditions

```
float total_sum;
void parallel_func() {
        int id;
        float average;
        float partial_sum;
        id=m_get_myid();

        if (id==0) partial_sum=1.0+2.0;
        else partial_sum=3.0+4.0;

        m_lock();
        total_sum+=partial_sum;
        m_unlock();

        average=total_sum/4.0;
        printf("%d average is %f\n",id, average);
}
```

Race Conditions

Possible output

0 average is 2.5	1 average is 1.75	0 average is 0.75
1 average is 2.5	0 average is 2.5	1 average is 2.5

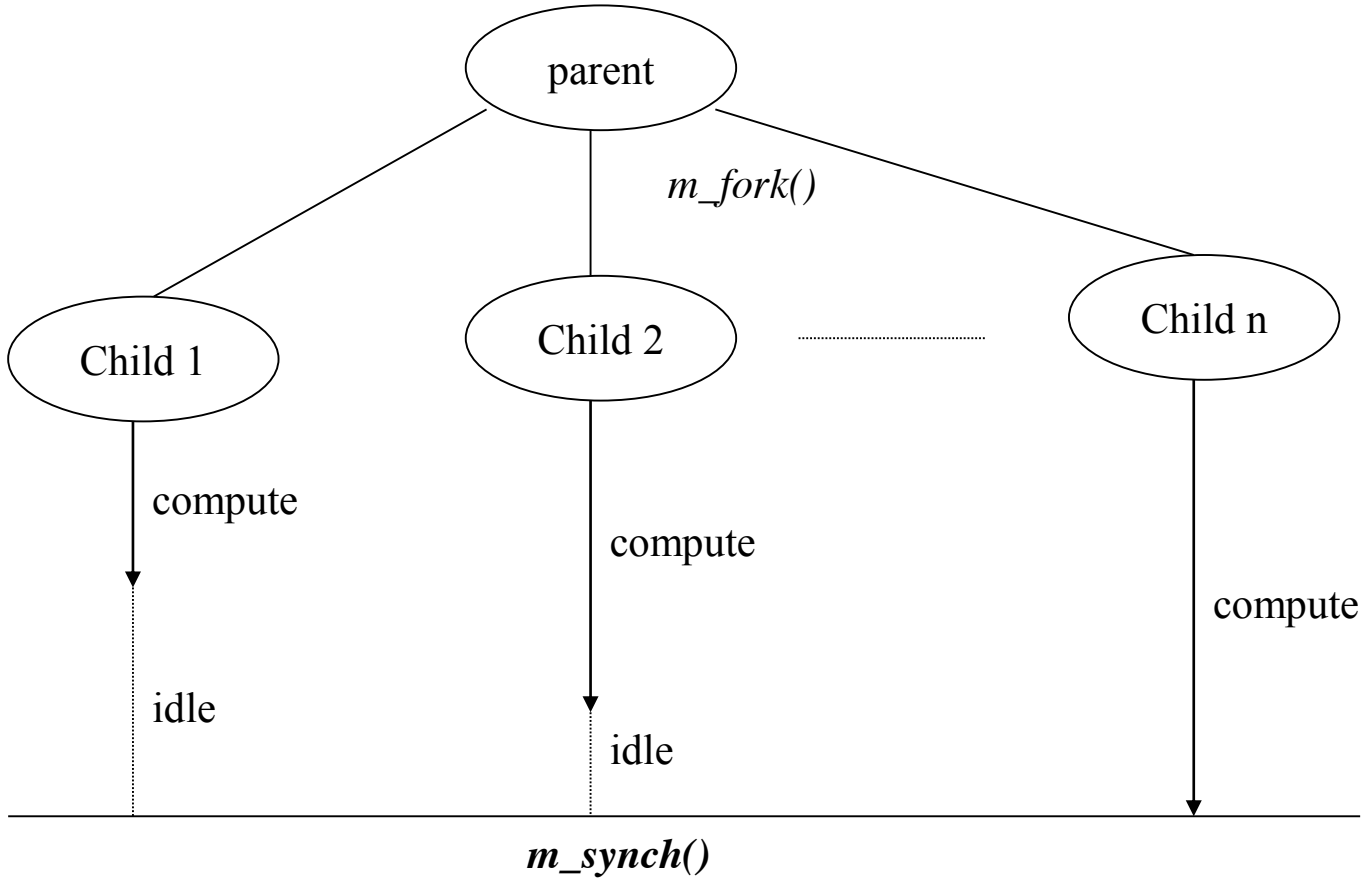
Barrier

- A race condition exists when?

The results of a parallel program depend on the relative execution speed of processes

- Barriers enable processes to synchronize with each other and help avoid race conditions
- When a process enters a barrier, it waits for all other processes involved to reach the barrier before continuing
- Barriers cause performance degradation and therefore must be used carefully

Illustration of Barrier



Barriers

```
float total_sum;
void parallel_func() {
        int id;
        float average;
        float partial_sum;
          id=m_get_myid();
        if (id==0) partial_sum=1.0+2.0;
        else partial_sum=3.0+4.0;
        m_lock();
        total_sum+=partial_sum;
        m_unlock();

        m_synch( );
        average=total_sum/4.0;
        printf("%d average is %f\n",id, average);
}
```

Process Summary

- Contention
  - Locks for exclusive access to shared variables
- Race conditions
  - Barriers for synchronization

Parallel Program Model Summary

- Data parallel programming dominates
- Loops in programs are the source of data parallelism
- Exploitation of parallelism involves sharing work in loops among processes
- Have to use appropriate scheduling techniques for optimal work sharing
- Parallelism in loops not always straightforward to find due to dependence
- Have to perform some transformations to expose parallelism

OpenMP

- A standard for directive based parallel programming
- Higher level than Pthreads, no threads manipulating
- API used in FORTRAN, C, C++
- Support for concurrency, synchronization, etc.

OpenMP Programming

- #pragma omp directive [clause list]
- #pragma omp parallel [clause list]

  /*structured block*/
- Example

```
int a, b;
main() {
    //serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
     {
            //parallel segment
     }
     //serial segment
 }
```

Message Passing Interface (MPI)

- MPI – standard for explicit message passing (C and Fortran)
- Portable
- Widely used, requires minimal underlying hardware knowledge
- Developed by a group of researchers
- Support almost all hardware vendors

MPI Primitives

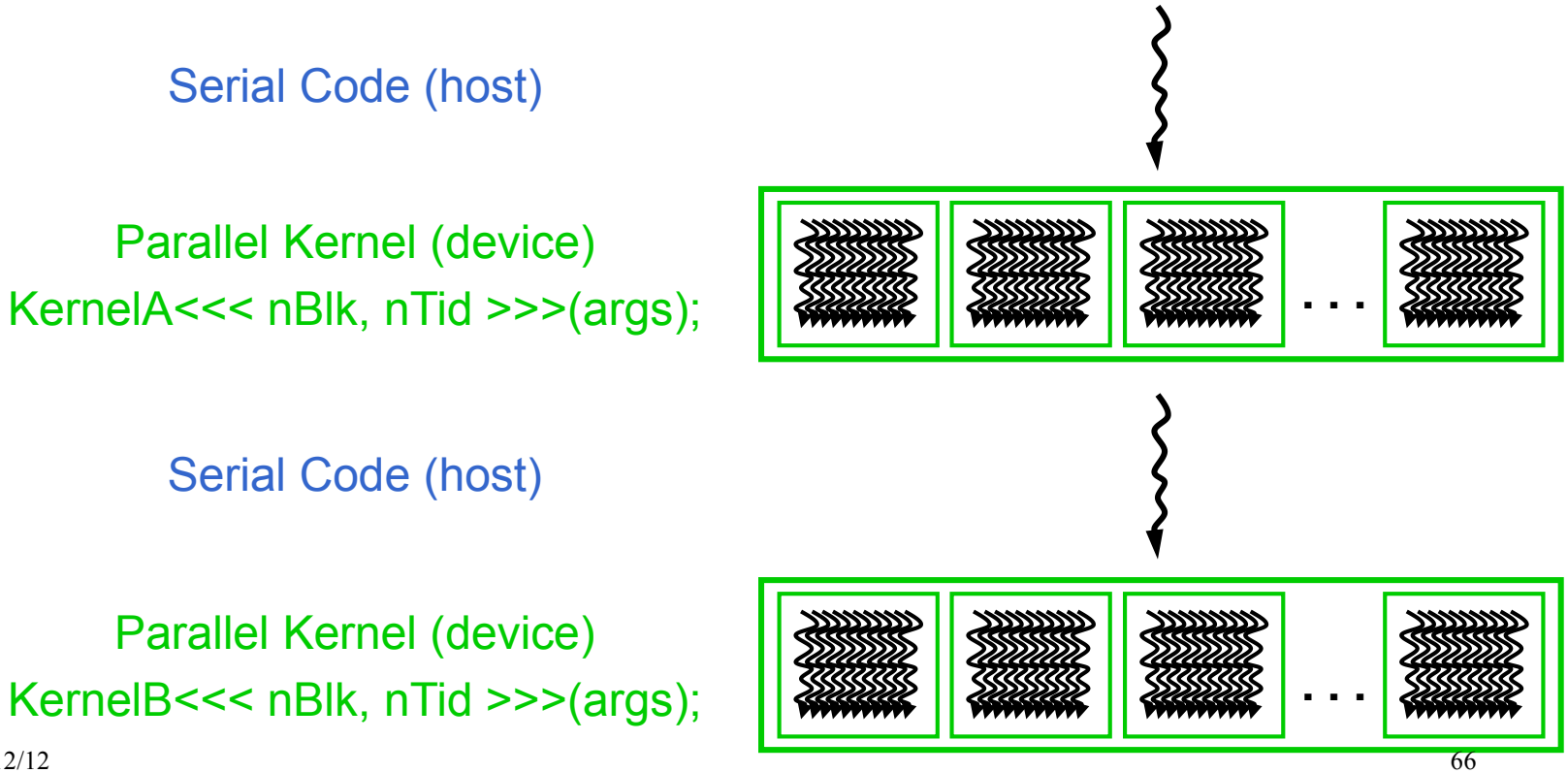
- MPI has over 125 functions, but need to know only 6 to get started on writing real applications
  - MPI_Init: initiate an MPI computation
  - MPI_Finalize: terminate an MPI computation
  - MPI_Comm_Size: determine number of processes
  - MPI_Comm_Rank: determine my process identifier
  - MPI_Send: send a message
  - MPI_Recv: receive a message

CUDA Device

- A compute device
  - Is a co-processor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Kernel — Data-parallel portions of an application which run on many threads

CUDA

- Integrated host+device C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code



Outline

- Introduction to High Performance Computing
  - Parallel Computing Platforms
  - Parallel Programming
    - Shared Memory Parallel Programming
    - Distributed Memory Message-passing Parallel Programming
    - CUDA-enabled GPU
- Parallel Classification
- Parallel Association Rules Mining
- Parallel Clustering

Decision Tree Algorithm

- Build tree (Stage I)
  - Start with data at root node
  - Select an attribute and formulate a logical test on attribute
  - Branch on each outcome of the test, and move subset of examples satisfying that outcome to corresponding child node
  - Recurse on each child node
  - Repeat until leaves are “pure”, i.e., have example from a single class, or “nearly pure”, i.e., majority of examples are from the same class

Decision Tree Algorithm

- Prune tree (Stage II)
  - Remove subtrees that do not improve classification accuracy
  - Avoid over-fitting, i.e., training set specific

Finding Good Splitting Points

- Use Gini index for partition purity

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

$$Gini(S_1, S_2) = \frac{n_1}{n} Gini(S_1) + \frac{n_2}{n} Gini(S_2)$$

- p_j is the relative frequency of class j in S.
- S is split into two subsets S_1 and S_2 with sizes N_1 and N_2 respectively
- If S is pure, Gini(S) = 0
- Find split-point with minimum Gini
- Only need class distributions

Computing Gini Index: Categorical Attributes

- For each distinct value, gather counts for each class in the dataset
- Use the count matrix to make decisions

Multi-way split

	CarType		
	Family	Sports	Luxury
C1	1	2	1
C2	4	1	1
Gini	0.393		

Two-way split
(find best partition of values)

	CarType	
	{Sports, Luxury}	{Family}
C1	3	1
C2	2	4
Gini	0.400	

	CarType	
	{Sports}	{Family, Luxury}
C1	2	2
C2	1	5
Gini	0.419	

Computing Gini Index: Continuous Attributes

- For efficient computation: for each attribute,
  - Sort the attribute on values
  - Linearly scan these values, each time updating the count matrix and computing gini index
  - Choose the split position that has the least gini index

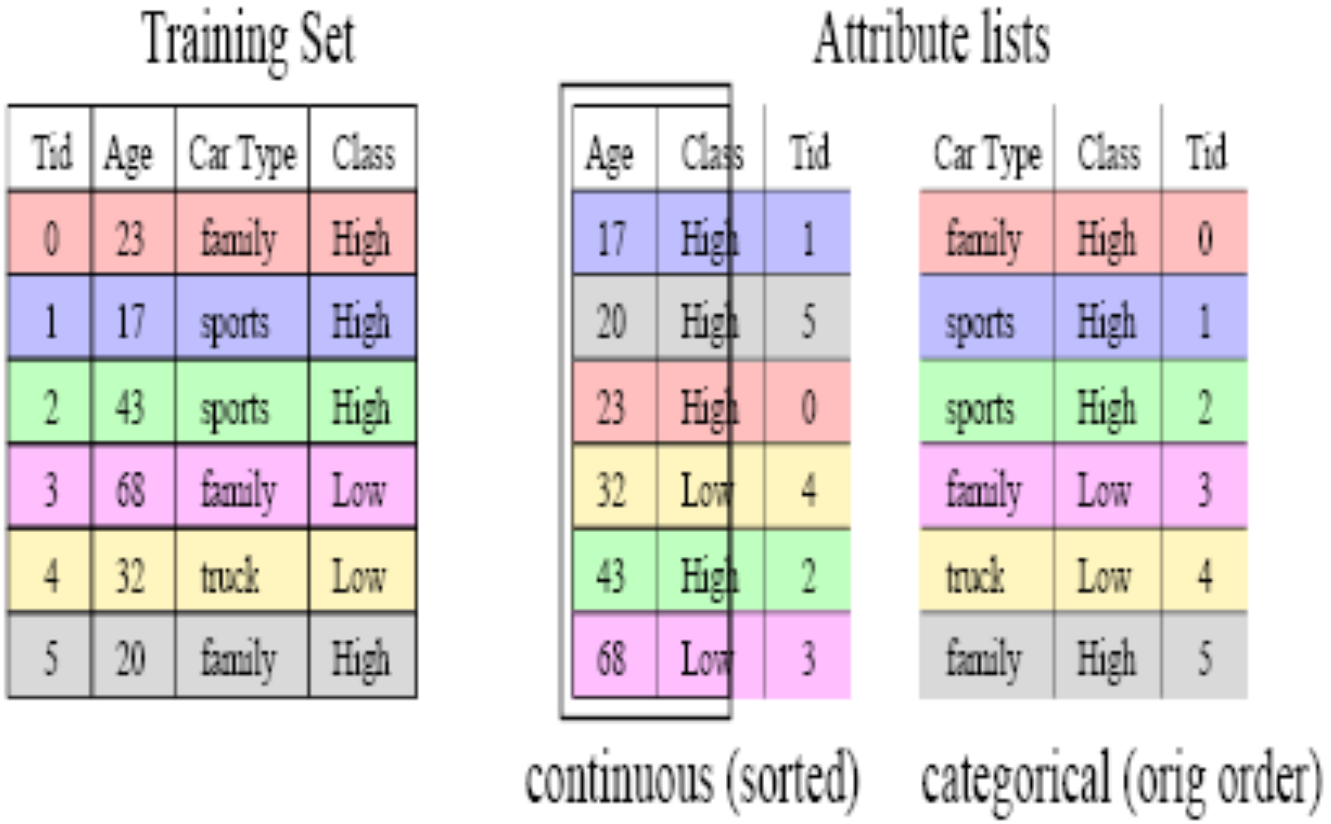
Cheat	No	No	No	Yes	Yes	Yes	No	No	No	No
Taxable Income										
Sorted Values →	60	70	75	85	90	95	100	120	125	220
Split Positions →	55	65	72	80	87	92	97	110	122	172	230
	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >	<= >
Yes	0 3	0 3	0 3	0 3	1 2	2 1	3 0	3 0	3 0	3 0	3 0
No	0 7	1 6	2 5	3 4	3 4	3 4	3 4	4 3	5 2	6 1	7 0
Gini	0.420	0.400	0.375	0.343	0.417	0.400	0.300	0.343	0.375	0.400	0.420

Decision Tree Algorithm

- Evaluate split-points for all attributes
- Select the “best” point and the “winning” attribute ← *Expensive*
- Split the data into two
- Breadth/depth-first construction

BASIC

- Attribute lists



BASIC

```
// Starting with the root node, execute the following code for
    each new tree level
forall attributes in parallel (dynamic scheduling)
     for each leaf
          evaluate attribute (E)
barrier
if (master) then
    for each leaf
          get winning attribute;
          form hash table (W)
barrier
forall attributes in parallel (dynamic scheduling)
     for each leaf
          split attributes (S)
```

BASIC

- Attribute data parallelism
  - *d*/P attributes to each processor
  - Acquire lock, grab attributes, release lock
- Dynamic scheduling
- Each processor works independently on its attributes, calculating *gini*
- Once all processors finish *gini*, enter E phase

BASIC

- Build a hash table for each leaf, keep how records partitioned
- Split attribute lists by checking with the hash table
- Breadth-first
  - Once a processor has been assigned an attribute, it can evaluate the splitting points for that attribute for all the leaves at the current tree level

Problem: when master performs W, all other processors sleep

SPRINT

Attribute list

Tid	Refund	Cheat
1	Yes	No
2	No	No
3	No	No
4	Yes	No
5	No	Yes
6	No	No
7	Yes	No
8	No	Yes
9	No	No
10	No	Yes

Tid	Marital Status	Cheat
1	Single	No
2	Married	No
3	Single	No
4	Married	No
5	Divorced	Yes
6	Married	No
7	Divorced	No
8	Single	Yes
9	Married	No
10	Single	Yes

Tid	Taxable Income	Cheat
1	125K	No
2	100K	No
3	70K	No
4	120K	No
5	95K	Yes
6	60K	No
7	220K	No
8	85K	Yes
9	75K	No
10	90K	Yes

SPRINT

- Class information is included in each attribute list
- The arrays of the continuous attributes are pre-sorted
- The sorted order is maintained during each split
- The classification tree is grown in a breadth-first fashion
- Hashing scheme used in splitting phase
  - tids of the splitting attribute are hashed with the tree node as the key
    - lookup table
  - remaining attribute arrays are split by querying this hash structure

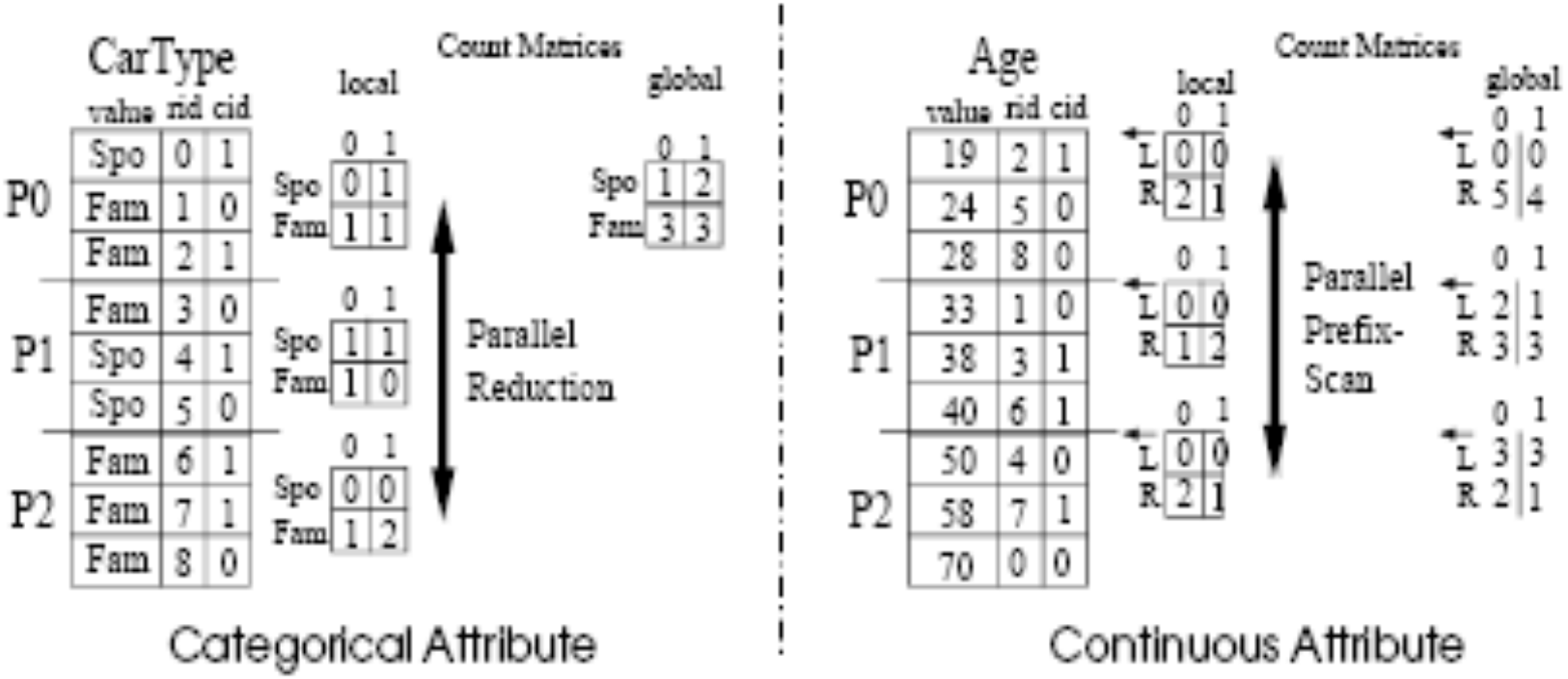
Parallel SPRINT

- Records are partitioned among processors
  - Each processor gets N/p records of each attribute list
  - No more attribute list movement
- Attribute lists of continuous attributes are pre-sorted
- Split Determining Phase (via communication)

Parallel SPRINT

- Split Determining Phase (via communication)
  - Categorical attributes
    - Local construction of count matrices followed by a reduction to add them up
  - Continuous attributes
    - Each processor Gini computation
    - Parallel pre-fix scan, reflecting the class distribution of all sections of an attribute-list assigned to processors of lower/higher rank
    - Find the global minimum gini point

Split Determination Phase



Parallel SPRINT (Contd.)

- Perform splitting
  - Each processor splits the winning attribute list section
  - Build local hash table, communicate, form global hash table
  - Each processor splits the other attribute lists by checking the hash table
- Problem

  Build hash table on all processors by an all-to-all broadcast operation. O(N) data sent & received by each processor --> Not Scalable in Runtime as well as Memory Requirements

Splitting Phase

- Splitting a categorical attribute list requires access exactly to those entries of the hash-table that are created by the same processor

	tid	attr1	Class
P1	1	A	0
	2	B	0
	3	A	1
P2	4	A	1
	5	B	0
	6	A	1
P3	7	A	0
	8	B	0
	9	B	1

Node
L
R
L
L
R
L
L
R
R

	tid	Attr2	Class
P1	1	Y	1
	2	N	0
	3	N	0
P2	4	Y	1
	5	Y	1
	6	N	0
P3	7	Y	0
	8	N	1
	9	N	0

Splitting Phase

- Splitting a continuous attribute, need to enquire hash table
- The hash-table entries required by a processor are generated by other processors

	tid	attr1	Class
P1	1	2454	0
	3	4942	0
	9	6491	1
P2	5	9476	1
	6	9767	0
	2	9881	1
P3	4	12614	0
	7	13683	0
	8	15303	1

Node
L
L
L
L
L
R
R
R
R

	tid	attr2	Class
P1	3	19	0
	6	24	0
	9	28	1
P2	2	33	1
	4	38	0
	7	40	1
P3	5	50	0
	8	58	0
	1	70	1

Outline

- Introduction to High Performance Computing
  - Parallel Computing Platforms
  - Parallel Programming Model
    - Shared Memory Parallel Programming
    - Distributed Memory Message-passing Parallel Programming
    - CUDA-enabled GPU
- Parallel Classification
- Parallel Association Rules Mining
- Parallel Clustering

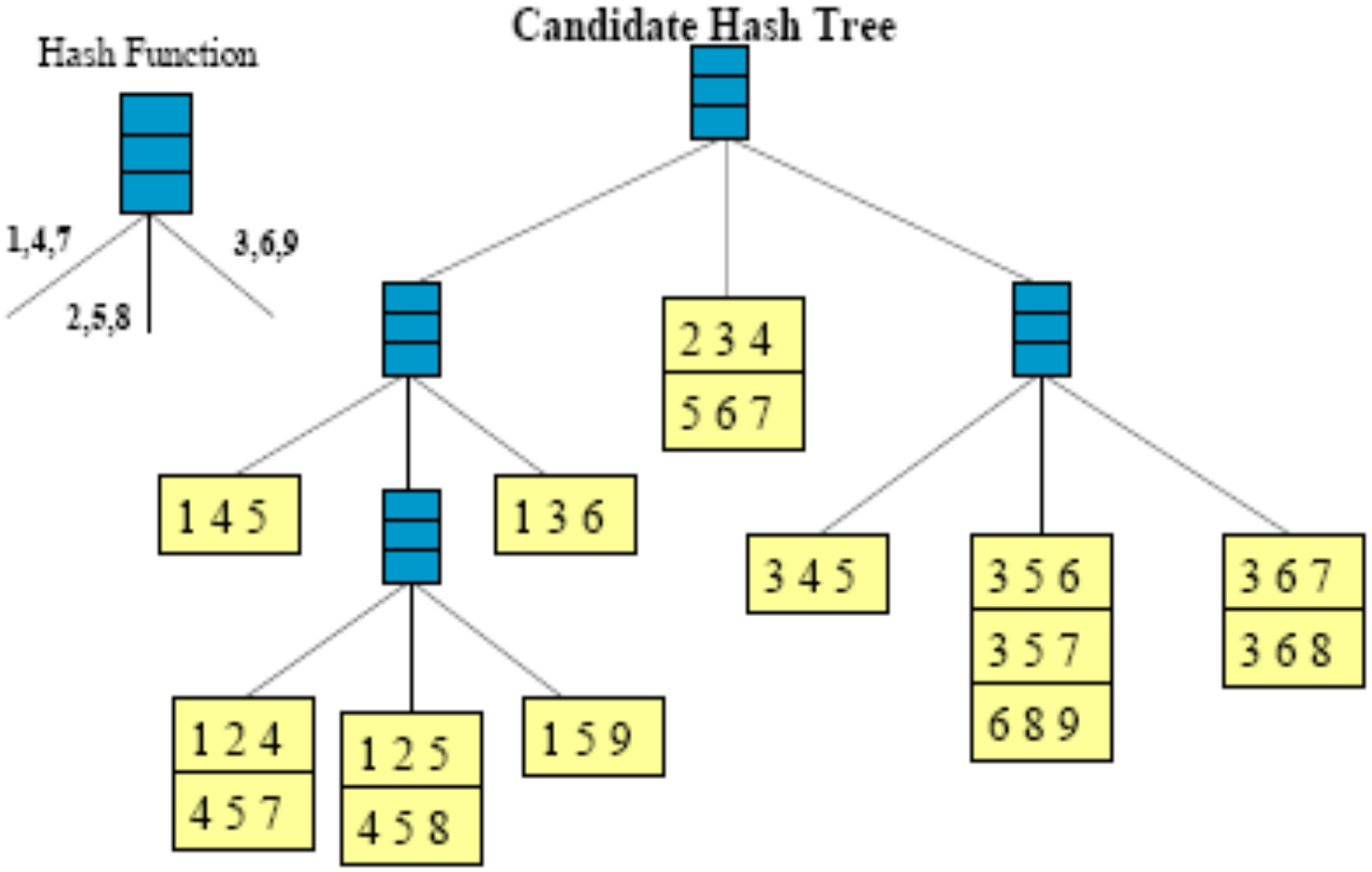
Apriori: A Candidate Generation-and-Test Approach

- Method:
  - Initially, scan DB once to get frequent 1-itemset
  - Generate length (k+1) candidate itemsets from length k frequent itemsets
  - Test the candidates against DB
  - Terminate when no frequent or candidate set can be generated

How to Generate Candidates?

- How to generate candidates?
  - Step 1: self-joining L_k
  - Step 2: pruning
- Example
  - L_3={*abc, abd, acd, ace, bcd*}
  - Self-joining: L_3*L_3
    - *abc* and *abd* -> *abcd*, *acd* and *ace* -> *acde*
  - Pruning:
    - *acde* is pruned because *ade* is not in L_3
  - C_4={*abcd, …*}

Example: Counting Supports of Candidates



CCPD (Common Count Partitioned Data)

- Parallelize candidate generation
  - Each processor works on a disjoint candidate subset
  - Build the hash tree in parallel, CCPD associates a lock with each leaf node
  - When a processor wants to insert a candidate into the tree, it starts at the root, and successively hashes on the items until it reaches a leaf
  - It then acquires the lock and inserts the candidate
  - With this locking mechanism, each processor can insert itemsets in different parts of the hash tree in parallel

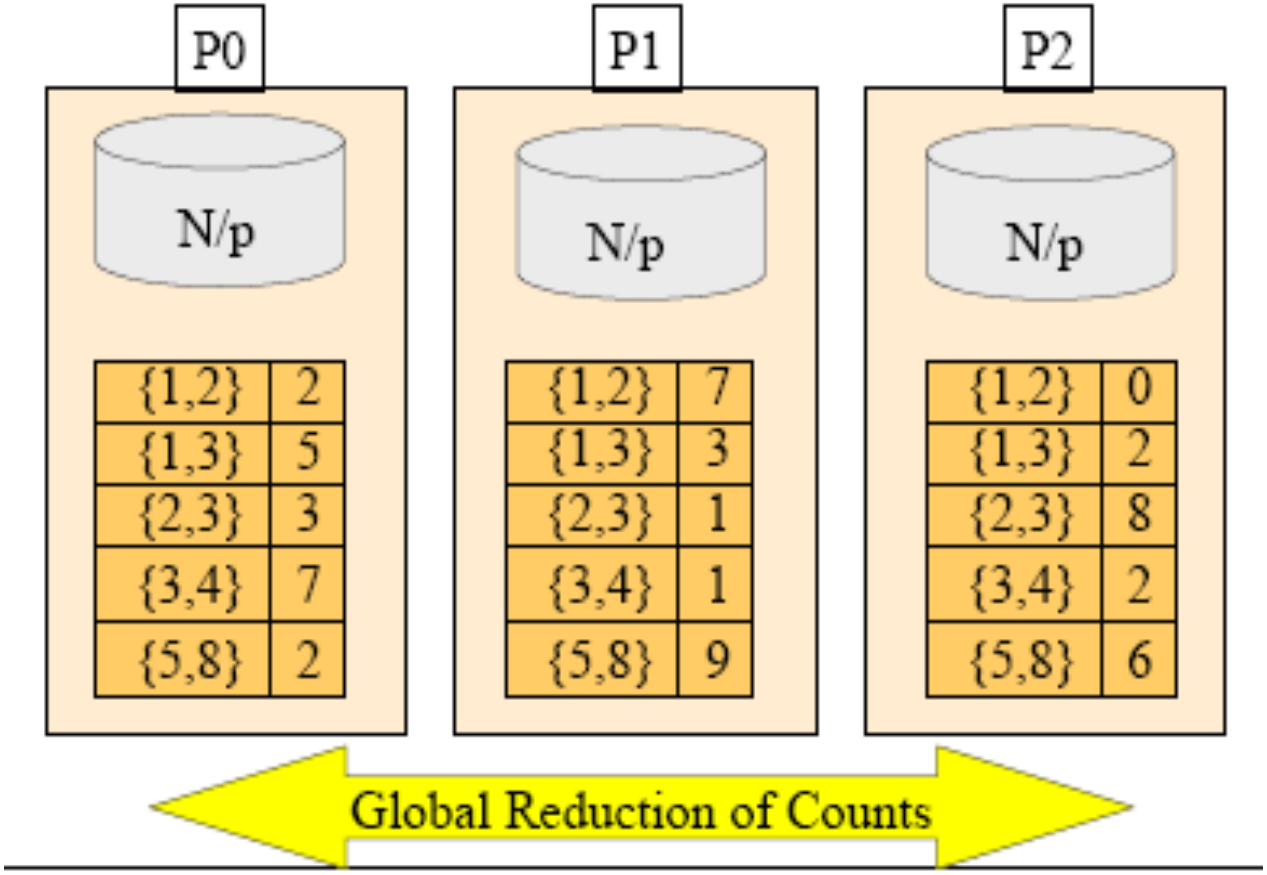
CCPD (Common Count Partitioned Data)

- Parallelize support counting
  - With this locking mechanism, each processor computes frequency from its local data partition, update the counts of each candidate in the hash tree

Count Distribution (CD)

- Apriori-like
- Iterative approach:
  - Each processor has complete candidate hash tree
  - Each processor updates its hash tree with local data
  - Each processor participates in global reduction to get global counts of candidates in the hash tree
- Multiple database scans per iteration are required if hash tree too big for memory

CD Illustration



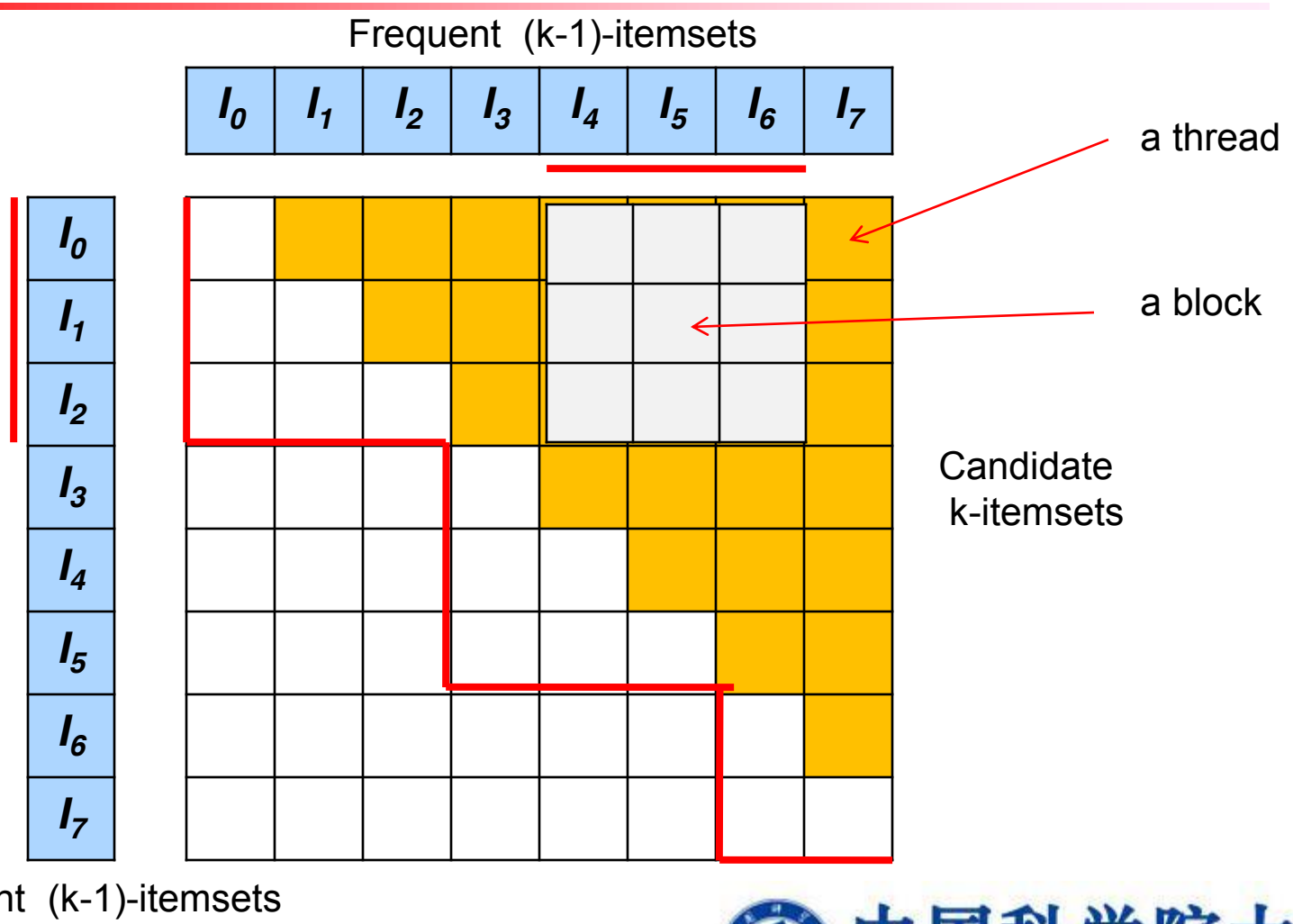
CUDA-enabled Parallel ARM

- Two CUDA kernels:
  - kernel 1: candidate generation
  - kernel 2: support counting

➢ *Liheng Jian, Cheng Wang, Shenshen Liang, Ying Liu, Weidong Yi, Yong Shi, "Parallel Data Mining on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)", Journal of Supercomputing, Vol. 64(3), 2013, pp. 942-967.*

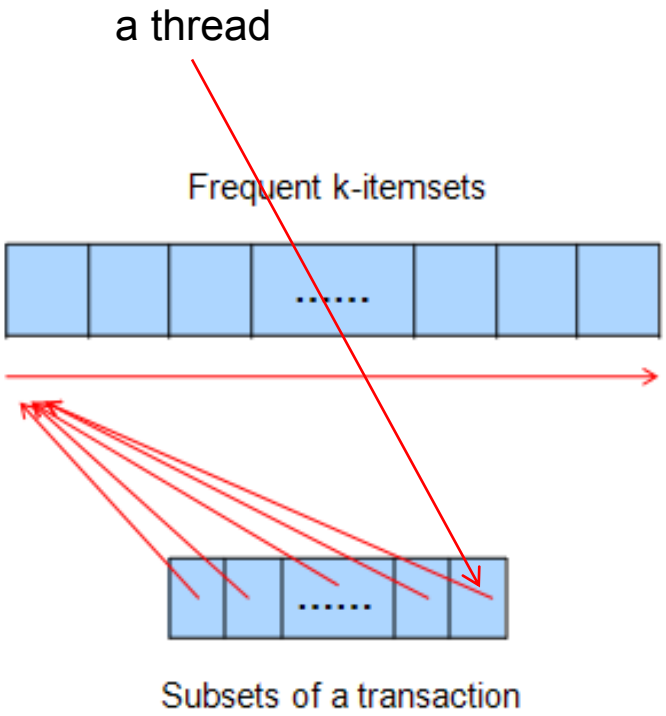
中国科学院大学
University of Chinese Academy of Sciences

Candidate Generation Kernel

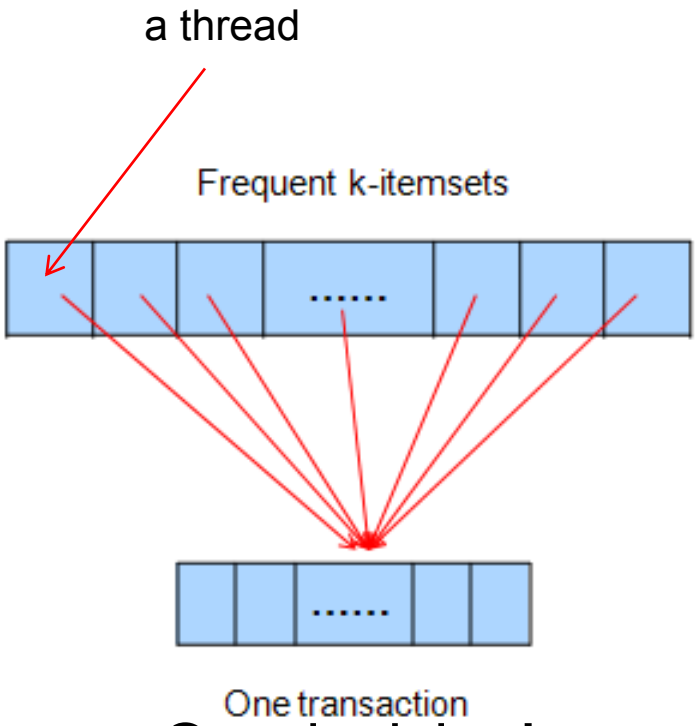


Support Counting Kernel

- Two possible thread models :



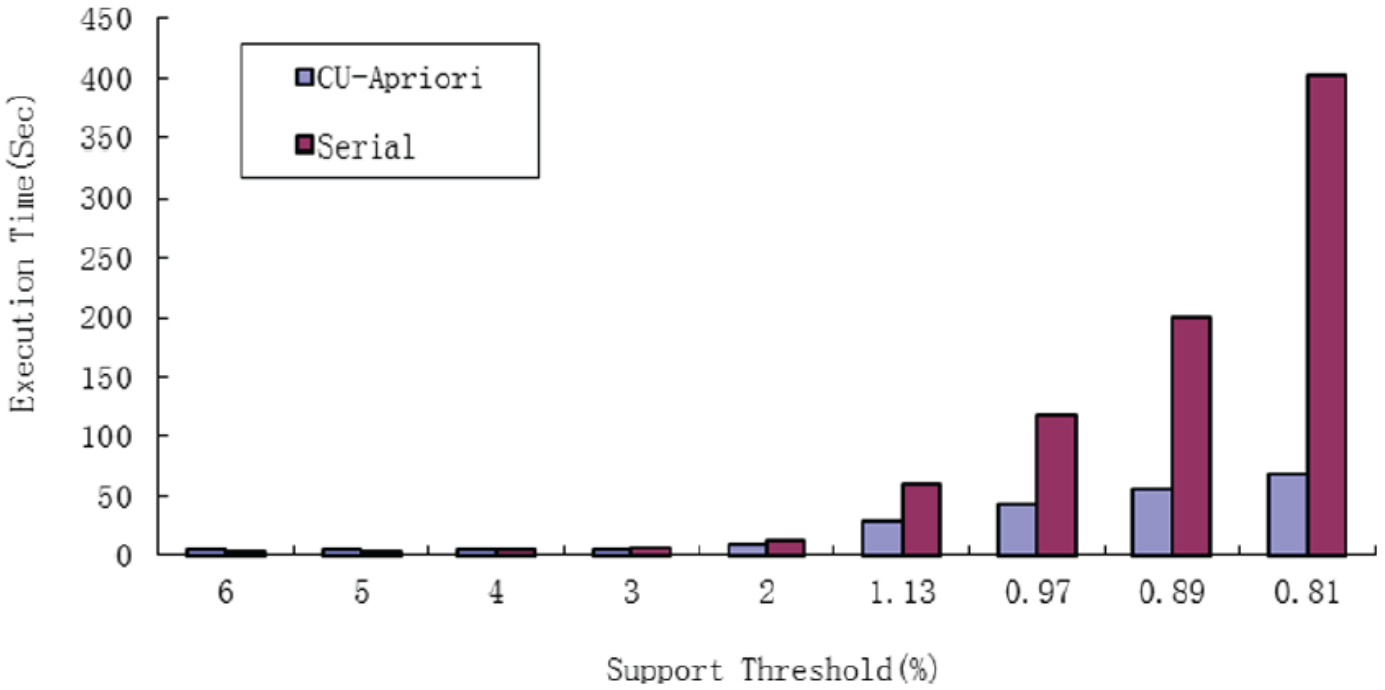
Bad scheduling!



Our decision!

Experimental Results

- Speedup obtained



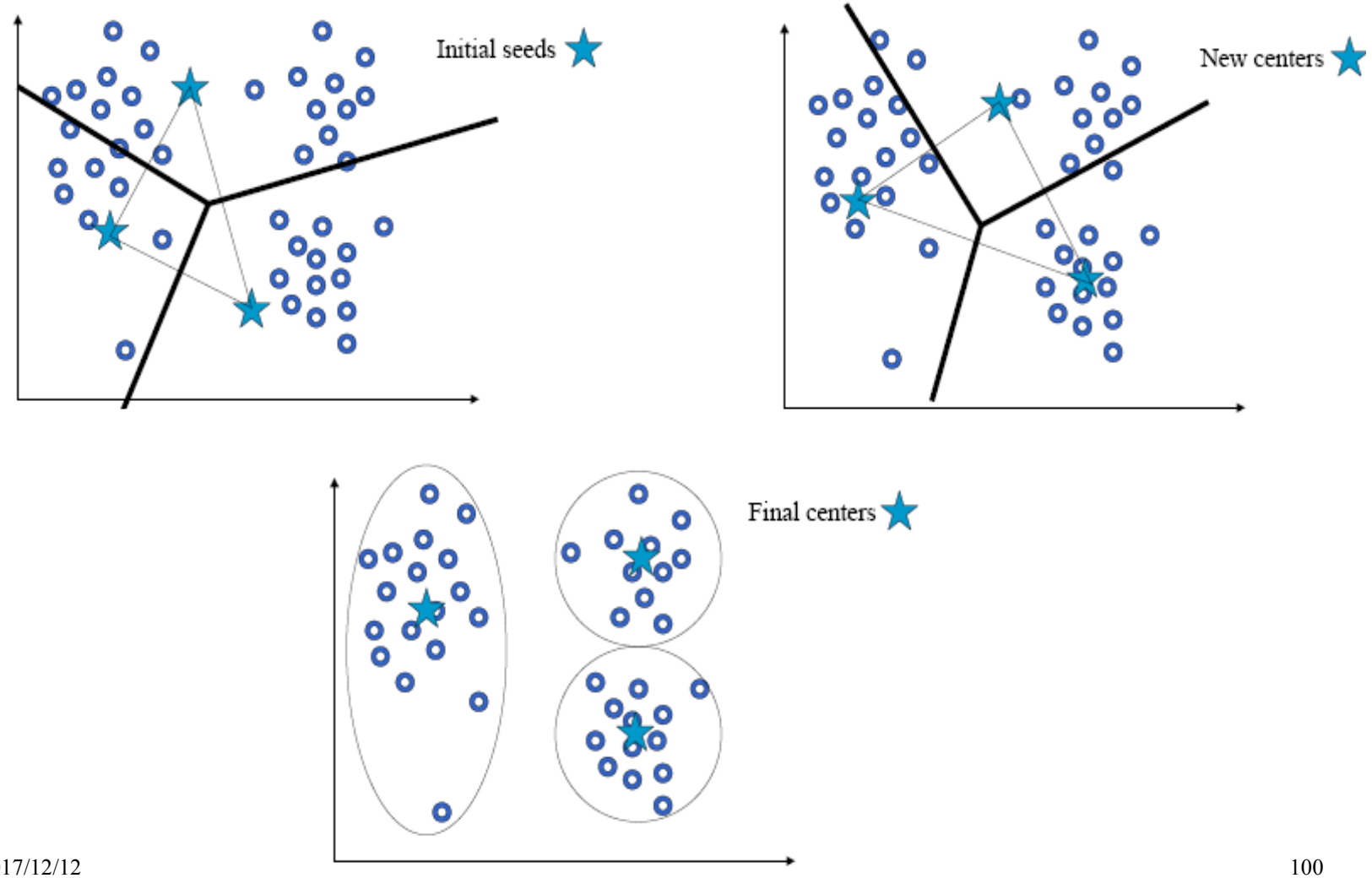
Outline

- Introduction to High Performance Computing
  - Parallel Computing Platforms
  - Parallel Programming Model
    - Shared Memory Parallel Programming
    - Distributed Memory Message-passing Parallel Programming
    - CUDA-enabled GPU
- Parallel Classification
- Parallel Association Rules Mining
- Parallel Clustering

K-Means Clustering

- Given *k*, the *k-means* algorithm is implemented in four steps:
  - Partition objects into *k* nonempty subsets, *k* random seeds as the initial centroids
  - Compute the centroid of the each cluster of the current partition (the centroid is the center, i.e., *mean point*)
  - For each object, compute its distance to the centroids
    - Assign it to the cluster with the nearest centroid
  - Update the *k* centroids
  - Go back to Step 2, stop when no more new assignment

K-Means Clustering



K-Means Clustering

- Main operation
  - Calculate distance to all *k* centroids, find the closest centroid for each point
  - Update the *k* centroids

Parallel *K-Means*

- Divide N points among P processors
- Replicate the *k* centroids
- Each processor computes distance of each local point to the centroids in parallel
- Assign points to closest centroid in parallel
- Perform reduction for global new *k* centroids
- Go back to step 2, until no centroid movement

CUDA-enabled Parallel K-Means

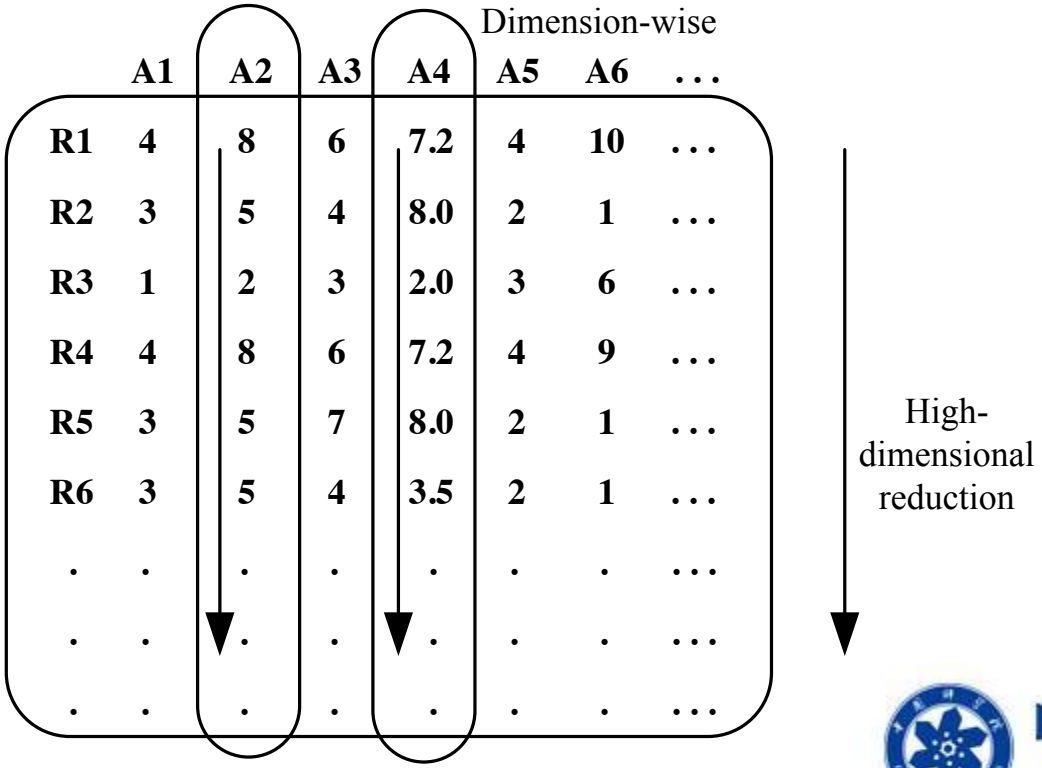
- Cluster label update
  - Load the *k* centers into the shared memory
  - Calculate the distance between point *p* and each of the *k* centers
  - Keep the center *c* closest to *p*
  - Synchronize within a thread block
  - Count the local cluster distribution within a thread block by parallel reduction

➢ *Liheng Jian, Cheng Wang, Shenshen Liang, Ying Liu, Weidong Yi, Yong Shi, "Parallel Data Mining on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)", Journal of Supercomputing, Vol. 64(3), 2013, pp. 942-967.*

中国科学院大学
University of Chinese Academy of Sciences

CUDA-enabled Parallel K-Means

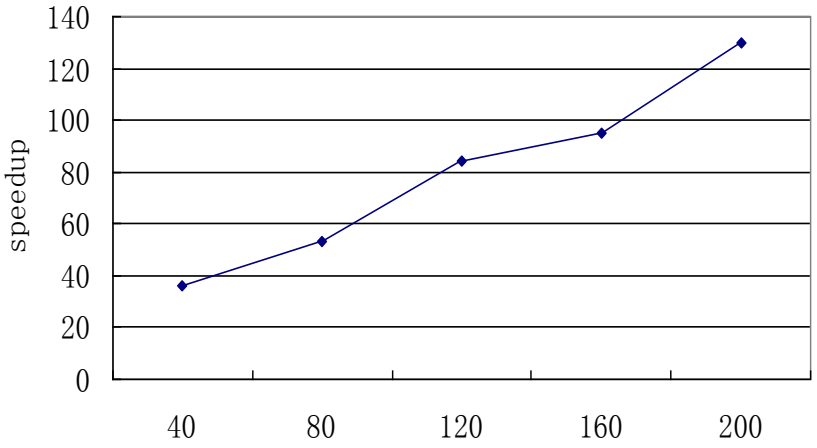
- Centroid update
  - Calculate an attribute value of each new center by high-dimensional parallel reduction



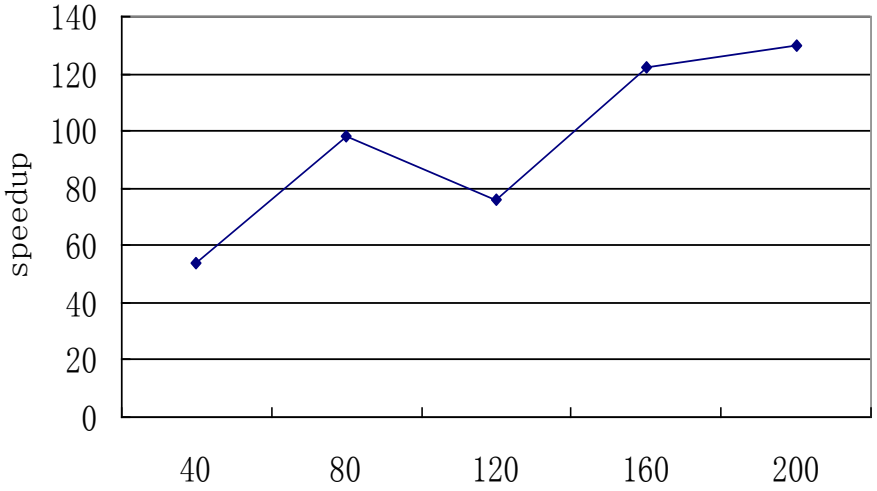
中国科学院大学 University of Chinese Academy of Sciences

Performance Evaluation – CU-K-Means

- Speedups varying with the number of points



- Speedups varying with *k*



- Platform
  - Tesla C1060, 240 cores, 4G global memory
  - HP XW8600 Workstation, quad-core 2.66GHz Intel Xeon CPU，4G Memory