



## **Introduction to parallel computing**

**Rami Melhem**  
**Department of Computer Science**

**July 23, 2007**

1



## **Evolution of parallel hardware**

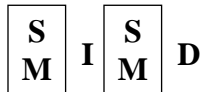
- I/O channels and DMA
- Instruction pipelining
- Pipelined functional units
- Vector processors (ILLIAC IV was built in 1947)
- Multiprocessors (cm\* and c.mmp were built in the 70's)
  
- Massively parallel processors (Connection machine, T3E, Blue Gene, ...)
- Symmetric Multiprocessors
  
- Cluster computing
  
- Multi-core processors
- Chip Multi-Processors

2



## Flynn's hardware taxonomy:

Looks at instructions and data parallelism. Oldest (1960's) and best known of many proposals.



- S for single
- I for instruction
- M for multiple
- D for data.

- SISD is a sequential computer.
- SIMD has one sequence of instructions applied to multiple data.
- MIMD has multiple sequence of instructions executing on multiple data.
- An MISD machine – need to be innovative to define it.

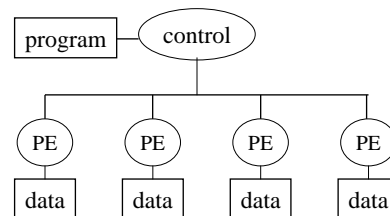
3



## SIMD (two flavors)

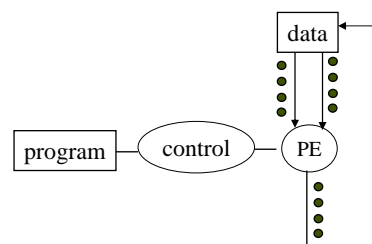
### 1) Synchronous, lockstep execution

All PEs execute the same instructions on different data



### 2) Vector processing

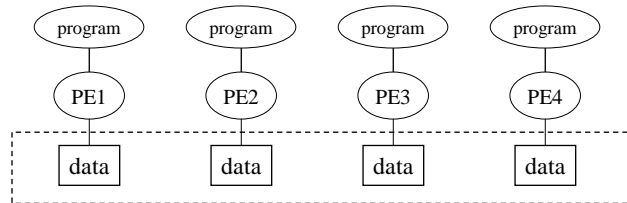
The same instruction is repeatedly executed on different data



4



## MIMD



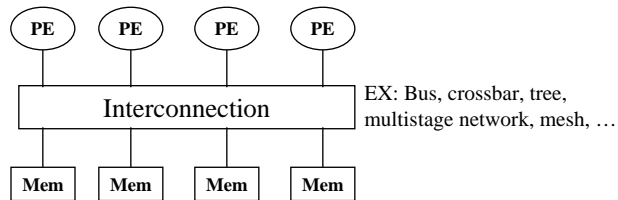
Multiple programs executing on different data – However, if all PEs are to cooperate to solve the problem (as opposed to solving different problems), there should be interaction between the programs and/or the data.

Many flavors depending on the **memory architecture** and the **address space** of each PE (the address space is the range of memory addresses that the PE can access).

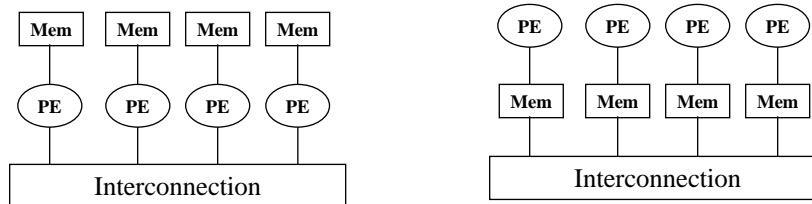
5



## Physical memory Architectures



**Global, shared memory (Symmetric Multi-Processors – SMP)**

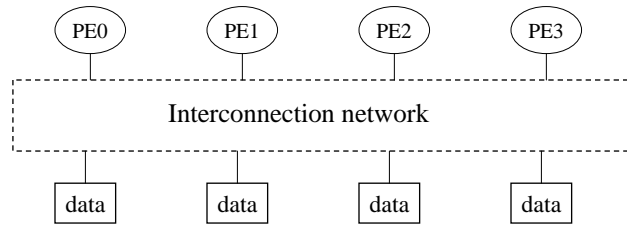


**Distributed memory**

6



## Shared address space



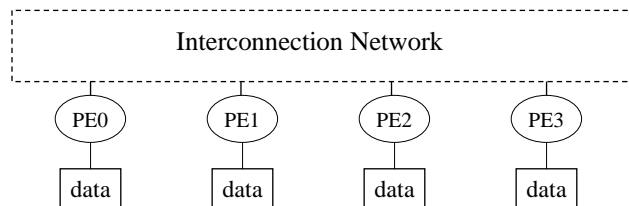
Each of PE0, PE1, PE2 and PE3 can address (directly access) locations  $0, \dots, M-1$

- No need for message passing – communicate through shared memory locations.

7



## Distributed address space



Assuming  $M = 4N$ , then

PE0 can address (directly access) locations  $0, \dots, N-1$

PE1 can address (directly access) locations  $N, \dots, 2N-1$

PE2 can address (directly access) locations  $2N, \dots, 3N-1$

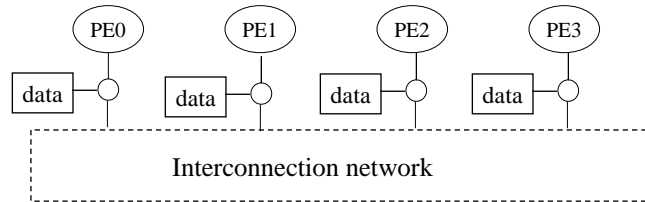
PE3 can address (directly access) locations  $3N, \dots, 4N-1$

- In order for PE<sub>i</sub> to access data in the address space of PE<sub>j</sub>, the two processors have to communicate through explicit messages.

8



## Distributed shared memory systems



Shared address space, but physically distributed memory.

- No need for message passing – communicate through shared memory locations.
- Data is physically distributed, but a runtime system is responsible to access data that do not reside in the local memory.

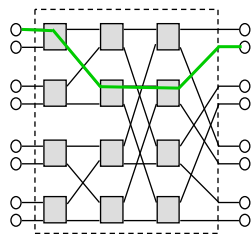
Results in the so called “Non Uniform Memory Access” – NUMA (as opposed to UMA, “Uniform Memory Access”)

9




## Two classes of interconnection networks

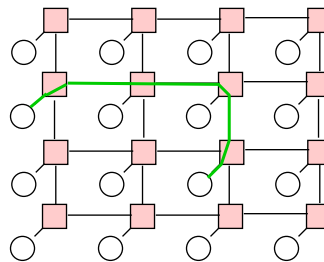
1) Large switch structures built from smaller switches – used for either shared or distributed memory systems

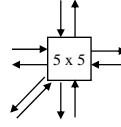



A 2x2 switch or router 

A processor and/or memory 

2) Topology-induced switching structures – typically used in distributed memory system



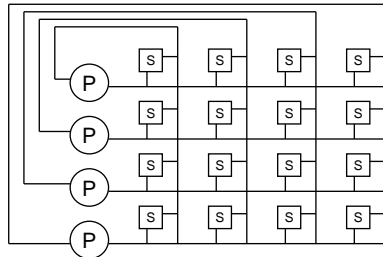
A 5x5 switch or router 

A processor + memory 

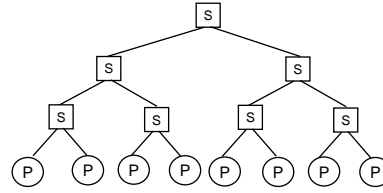
10



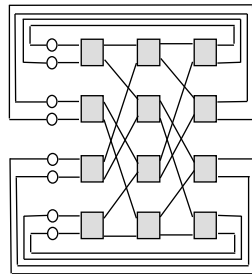
### Common interconnection switches



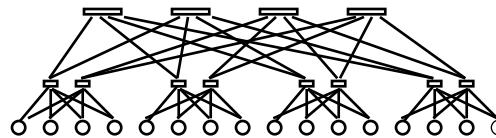
Crossbar



Tree



Multistage



Fat tree

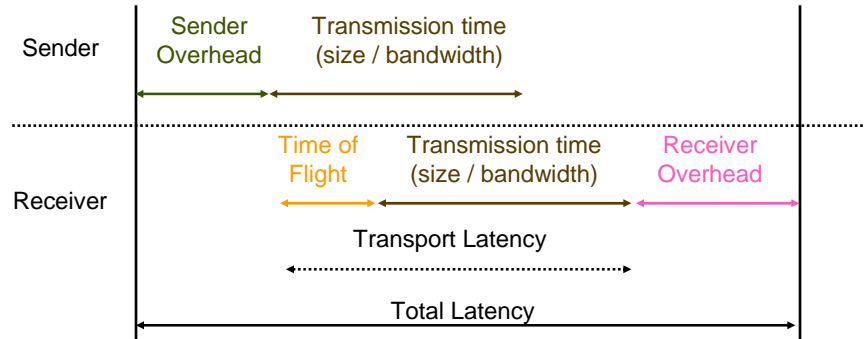


### Common interconnection topologies

						N = 1024	
Type	Degree	Diameter	Av Dist	Bisection	Diam	Av. D	
	2	N-1	N/3	1			
	4	$2(N^{1/2} - 1)$	$2N^{1/2} / 3$	$N^{1/2}$	63	21	
	6	$3(N^{1/3} - 1)$	$3N^{1/3} / 3$	$N^{2/3}$	~30	~10	
	2	N / 2	N/4	2			
	4	$N^{1/2}$	$N^{1/2} / 2$	$2N^{1/2}$	32	16	
	2n	$n(N^{1/n})$ $nk/2$	$nN^{1/n}/2$ $nk/4$	$2k^{n-1}$	15	8 (3D)	
	n	n = LogN	n/2	N/2	10	5	



## Overheads of message passing



**Total Latency =**

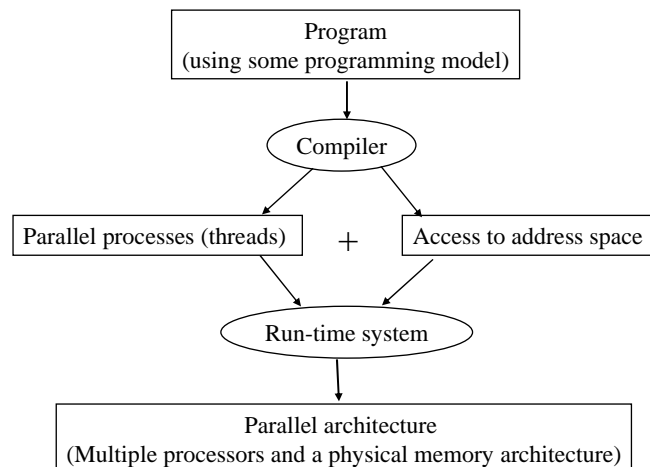
Sender Overhead + Time of Flight + transmission time + Receiver Overhead

**Software overhead at sender/receiver usually dominates**

13



## Programming parallel computing systems



Note the decoupling between the programming model and the physical architecture – For instance, a parallel program can run on a single processor!!!

14



## Two schools for programming parallel systems:

- Automatic detection of parallelism in serial programs and automatic distribution of data and computation.
- User specified parallelism (data distribution, computation distribution, or both).

### Problems:

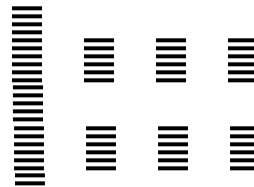
- It is hard to think “parallel.” (is it ???)
- The dusty-deck problem (software inertia) and the absence of good tools for parallelizing serial programs.
- The I/O bottleneck.
- Government funding and commercial success
- Communication and synchronization overhead



## Parallel Programming Models (control threads - processes).

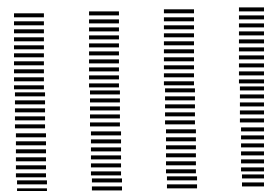
- 1) Start with one control thread, and create other threads when needed

Examples: Pthreads (explicit thread creation) and OpenMP (implicit thread creation).



- 2) Start with multiple control threads – usually multiple copies of the same program (SPMD – single program, multiple data).

How do you make the same program do different things???





## Parallel Programming Models (scope of variables).

- 1) Variables declared shared among threads or processes – any process can read/write to these variables.

Problems with race conditions???

- 2) Variables declared private to a process or thread

To make the value of a private variable available to other processes, one has to either exchange messages, or copy the value to a shared variable.

A programming model can combine private and shared variables, as well as allow message passing.

17



## Example - Pthreads

```
int main(int argc, char *argv) {
double A[100] ; /* global, shared variable*/
int i ;
...
for (i = 0; i < 4 ; i++) pthread_create( ... , DoStuff, int i ) ;
... /* execution continues in parallel with 4 copies of DoStuff*/
...
for (i = 0; i < 4 ; i++) pthread_join ( ... , DoStuff, ... ) ;
...
}

void DoStuff (int threadID) {
int k ; /* k is a local variable – each instance of DoStuff has a copy*/
... /* do stuff in parallel with main */
for (k = threadID*25 ; k < (threadID+1)*25 ; k++) ... do something with A[k] ...
...
}
```

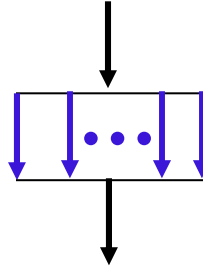
The five threads can be executed on separate CPUs or time\_shared on one CPU

18



## Example - OpenMP

```
int main(){
  print("Start\n");
  ... /* serial code */
  #pragma omp parallel {
    ...
    printf("Hello World\n");
    ...
  }
  ... /* resume serial code */
  printf("Done\n");
}
```



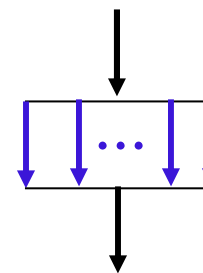
```
% Result of execution
Start
Hello World
Hello World
Hello World
Hello World
Done
```

The user can control the number of parallel threads by setting the environment variable `setenv OMP_NUM_THREADS 4`



## Example - OpenMP

```
#define n 1000
int main(){
  int i, a[n], b[n], c[n] ;
  ...
  ...
  #pragma omp for shared(a,b,c), private(i)
  { for (i = 0; i < n ; i++)
    c[i] = a[i] + b[i] ;
  } /* end of parallel section */
  ... /* resume serial code */
  ...
}
```



The loop will be automatically broken down into smaller loops and each small loop will be given to one thread

**Warning: the loop iterations should be independent (no loop carried dependences)**



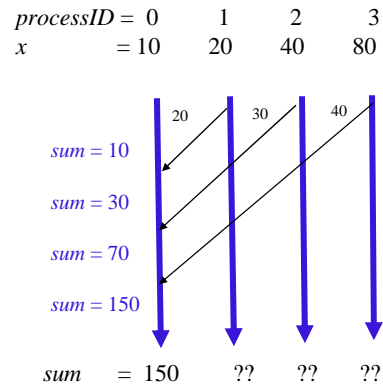
## Example – a message passing program

```

int main(){
int x ,sum, i ; /* local variables */
...
call a function to get the num_processors ;
...
call a function to get your processorID ;
compute a local value for x;
if (processorID > 0)
    send the value of x to processor 0 ;
else {
    sum = x ;
    for (i = 1; i < num_processors ; i++)
    { receive a value from processor i ;
      add that value to sum
    }
};
...
}

```

The number of processors (threads) is specified before execution starts

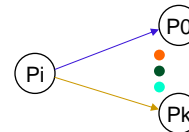
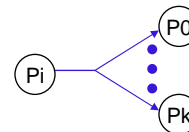
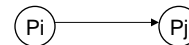


21



## Type of messages

- **Point-to-point:** one processor sends a message to another processor
- **One-to-all:** one processor broadcasts a message to all other processors
- **One-to-all personalized:** one processor sends a different message to each other processor
- **All-to-all:** each processor broadcasts a message to all other processors



22



## Blocking and non-blocking messages



- Depending on the type of call, a process issuing a **blocking send** does not continue execution until
  - The message is copied to the *send buffer*
  - The message is sent on the network
  - The message reached the *receive buffer*
  - The message is received by the receiving process.
- A process issuing a **non-blocking send** continues execution immediately without making sure that the message is sent.
- A process that issues a **blocking receive** does not continue execution before the message is received.
- A process that issues a **non-blocking receive** does continue execution if the message is not in the receive buffer – can check the buffer later.

23

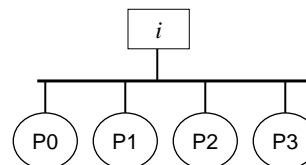


## Synchronization (race conditions)

What is the output of the following OpenMP program??

```

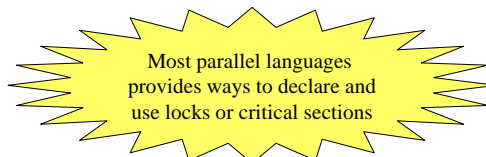
setenv OMP_NUM_THREADS 4
int main(){
int i = 0 ; /*initialized global variable */
#pragma omp
{
  i = i + 1 ;
}
Print the value of i ;
}
  
```



- A **critical section** is a section of code that can be executed by one processor at a time (to guarantee mutual exclusion)
- **locks** can be used to enforce mutual exclusion

```

Declare a lock
#pragma omp
{ get the lock ;
  i = i + 1 ;
  release the lock ;
}
  
```



24

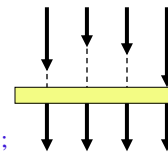


## Synchronization (barriers)

What is the output of the following Pthread program??

```
int main(int argc, char *argv) {
    double A[101], B[101], C[100]; /* global, shared variables*/
    for (i = 0; i < 101; i++) A[i] = B[i] = i;
    for (i = 0; i < 4; i++) pthread_create( ... , DoStuff, int i );
    ...
    for (i = 0; i < 4; i++) pthread_join ( ... , DoStuff, ... );
    Print the values of C ;
}

void DoStuff (int threadID) {
    int k ;
    for (k = threadID*25 ; k < (threadID+1)*25 ; k++) B[k] = 2 * A[k] ;
    Barrier
    for (k = threadID*25 ; k < (threadID+1)*25 ; k++) C[k] = 2 * B[k+1] ;
}
```

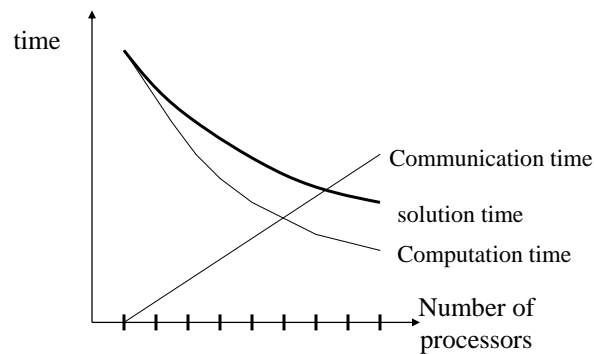


25



## Effect of communication

- Total solution time:
  - 1) Computation time (more processors = faster computation)
  - 2) Communication time (more processors = more communication)



Tradeoff between computation and communication

26



## Parallel algorithms

27



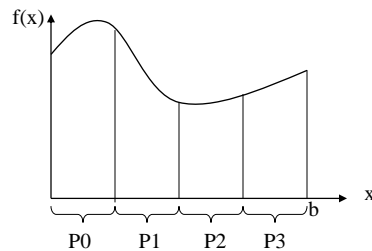
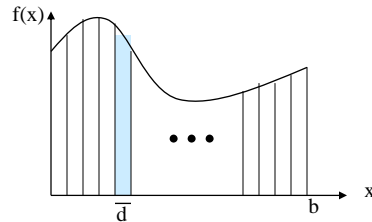
### Parallelizing an algorithm Numerical integration - a message passing example

```
n = 1000 ; d = b / n ;  
area = 0 ;  
for (i=0 ; i < n ; i++) {  
    x = i * d + d / 2 ;  
    area = area + f(x) * d ;  
}
```

Divide the work among 4 processors, so that each processor computes a section of the area

```
n = 1000 ; d = b / n ;  
p_area = 0 ; /* local variable */  
id = my processor id ; /* 0,1,2,3 */  
for (i=id * 250 ; i < (id+1)*250 ; i++) {  
    x = i * d + d / 2 ;  
    p_area = p_area + f(x) * d ;  
}
```

**Exercise: rewrite for K processors.**



28

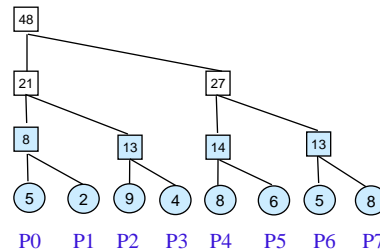


## Numerical integration still need to accumulate the partial areas

Processors 1, ..., K-1 can send their values to processor 0, and processor 0 will do the accumulation – takes K-1 time steps to complete.

**A more efficient way is to use a recursive doubling technique.**

```
If (id mod 2 == 1)
    send p_area to processor id-1 ;
else { receive the p_area from processor id+1 ;
      add the received value to the local p_area ;
```



29

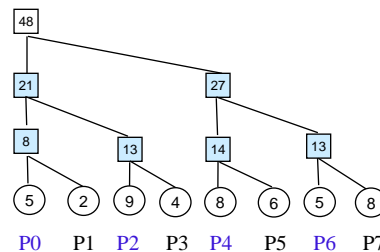


## Numerical integration still need to accumulate the partial areas

Processors 1, ..., K-1 can send their values to processor 0, and processor 0 will do the accumulation – takes K-1 time steps to complete.

**A more efficient way is to use a recursive doubling technique.**

```
If (id mod 2 == 1)
    send p_area to processor id-1 ;
else { receive the p_area from processor id+1 ;
      add the received value to the local p_area ;
If (id mod 4 == 2)
    send p_area to processor id-2 ;
elseif (id mod 4 == 0)
    { receive the p_area from processor id+2 ;
      add the received value to the local p_area ;
```



30



## Numerical integration still need to accumulate the partial areas

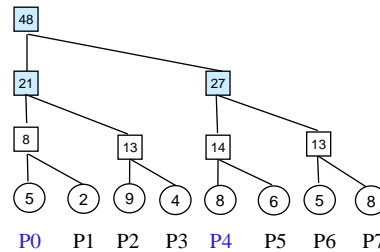
Processors 1, ..., K-1 can send their values to processor 0, and processor 0 will do the accumulation – takes K-1 time steps to complete.

**A more efficient way is to use a recursive doubling technique.**

```

If (id mod 2 == 1)
    send p_area to processor id-1 ;
else { receive the p_area from processor id+1 ;
      add the received value to the local p_area } ;
If (id mod 4 == 2)
    send p_area to processor id-2 ;
elseif (id mod 4 == 0)
    { receive the p_area from processor id+2 ;
      add the received value to the local p_area } ;
If (id mod 8 == 4)
    send p_area to processor id-4 ;
elseif (id mod 8 == 0)
    { receive the p_area from processor id+4 ;
      add the received value to the local p_area } ;

```



## Accumulating the partial areas Assuming K processors

```

If (id mod 2 == 1)
    send p_area to processor id-1 ;
else (id mod 2 == 0)
    { receive the p_area from processor id+1 ;
      add the received value to the local p_area } ;
If (id mod 4 == 2)
    send p_area to processor id-2 ;
elseif (id mod 4 == 0)
    { receive the p_area from processor id+2 ;
      add the received value to the local p_area } ;
If (id mod 8 == 4)
    send p_area to processor id-4 ;
elseif (id mod 8 == 0)
    { receive the p_area from processor id+4 ;
      add the received value to the local p_area } ;

```

```

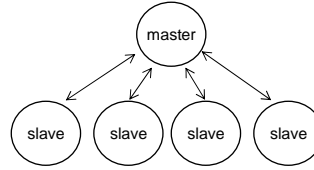
K = number of processors ;
for (i=1 ; i <= log K ; i++) {
If (id mod 2i == 2i-1)
    send p_area to processor id - 2i-1 ;
elseif (id mod 2i == 0)
    { receive the p_area from processor id + 2i-1 ;
      add the received value to the local p_area } ;
}

```



## The master/slave programming approach

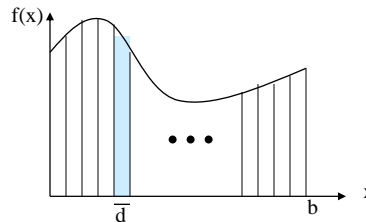
- Master divides the work into *work\_units* ;
- While work is not done {
  - Wait for an available slave ;
  - Send a work unit to the available slave



### The numeric integration example

```

n = 10000 ; d = b / n ;
area = 0 ;
for (i=0 ; i < n ; i++) {
  x = i * d + d / 2 ;
  area = area + f(x) * d ;
}
  
```



33



## Numerical integration - a master/slave approach

### Program for processor 0 (master)

```

next_ld = 0 ; work_unit = 10 ;
for ( i = 1 ; i <= K ; i++ ) {
  send next_ld to processor i ;
  next_ld = next_ld + work_unit
} ;
while ( next_ld + work_unit < n ) {
  wait for a message from any processor ;
  when you get a message from processor i,
  { add the received p_area to area ;
    send next_ld to processor i ;
    next_ld = next_ld + work_unit
  }
}
for ( i = 1 ; i <= K ; i++ )
  send a termination message to processor i ;
  
```

### Program for processors 1, ..., K-1 (slaves)

```

While (true) {
  Receive a message from processor 0 ;
  If not a termination message {
    get the value of next_ld ;
    p_area = 0 ;
    for (i=next_ld; i < next_ld+work_unit; i++) {
      x = i * d + d / 2 ;
      p_area = p_area + f(x) * d ;
    }
    send p_area to processor 0 ;
  }
}
  
```

What is the effect of the *work\_unit* granularity on performance?

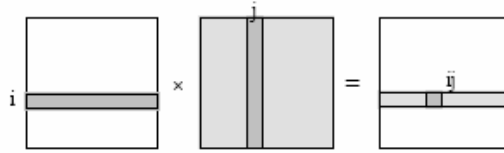
34



## Parallelizing an algorithm Matrix/matrix multiplication - an example

```

for i = 0, ..., m-1
  for j = 0, ..., m-1 {
    c[i,j] = 0 ;
    for k = 0, ..., m-1
      c[i,j] += a[i,k] * b[k,j]
  }
  
```



Assuming  $m^2$  processors with shared memory, each processor executes:

```

Get the processor id /* 0 <= id < m^2 */
i = id / m ; j = id mod m ;
c[i,j] = 0 ;
for k = 0, ..., m-1
  c[i,j] += a[i,k] * b[k,j]
  
```

- What if the number of processors,  $K = m$  and not  $m^2$  ??
- What if  $K = m/q$ , for some integer,  $q$ , ??

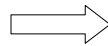
35



## $m \times m$ matrix/matrix multiplication using 4 processors ( $m$ is a multiple of 4)

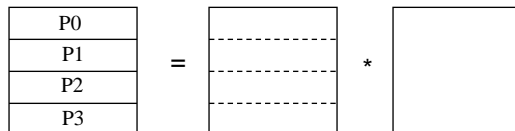
```

for i = 0, ..., m-1
  for j = 0, ..., m-1 {
    c[i,j] = 0 ;
    for k = 0, ..., m-1
      c[i,j] += a[i,k] * b[k,j]
  }
  
```



```

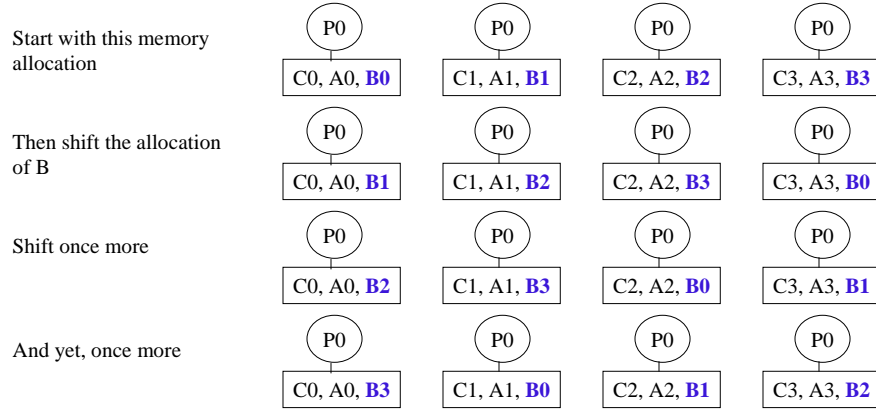
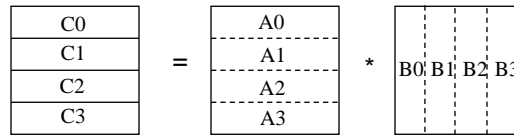
Get processor id, /* 0, 1, 2, 3 */
for i = id*(m/4), ..., (id+1)*(m/4)-1
  for j = 0, ..., m-1 {
    c[i,j] = 0 ;
    for k = 0, ..., m-1
      c[i,j] += a[i,k] * b[k,j]
  }
  
```



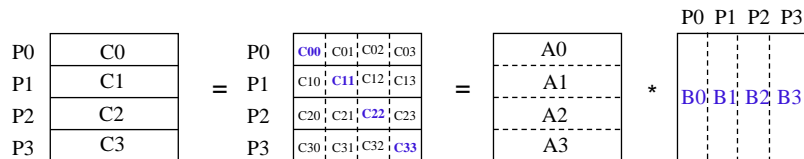
36



### What if we have a distributed memory system?



### The distributed memory program



```
float c[m/4,m] = 0 ;
float a[m/4,m] ; b[m,m/4] ;          /* local variables hold initial allocation*/
for i = 0, ... , m/4 - 1 /* id is the processor identifier */
  for j = 0, ... , m/4 - 1
    for k = 0, ... , m-1
      c[ i , j + id*(m/4)] =+ a[ i,k ] * b[ k,j ]
```

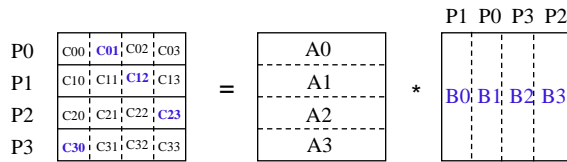
Note the mapping between local arrays a, b & c, and the global arrays A, B & C :

$a[ i , j ] = A[ i + id*(m/4), j ]$        $c[ i , j ] = C[ i + id*(m/4), j ]$

$b[ i , j ] = B[ i , j + id*(m/4) ]$



### The distributed memory program



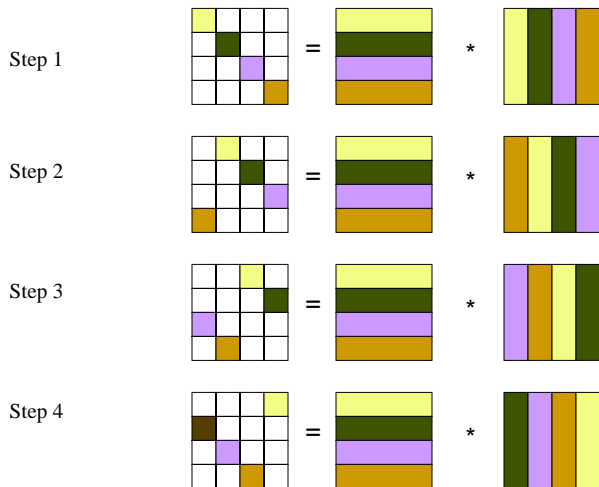
```

float c[m/4,m] = 0 ;
float a[m/4,m] ; b[m,m/4] ; /* local variables hold initial allocation*/
for i = 0, ... , m/4 - 1 /* step 1 */
  for j = 0, ... , m/4 - 1
    for k = 0, ... , m-1
      c[ i, j+ id*(m/4)] += a[ i,k ] * b[ k,j ] ;
  send b[ . , . ] to processor (id-1) mod 4 ;
  receive b[ . , . ] from processor (id+1) mod 4 ;
for i = 0, ... , m/4 - 1 /* step 2 */
  for j = 0, ... , m/4 - 1
    for k = 0, ... , m-1
      c[ i, j + ((id+1) mod 4 )*(m/4)] += a[ i,k ] * b[ k,j ] ;

```



### The distributed memory program





## The distributed memory program

```

float c[m/4,m] = 0 ;
float a[m/4,m] ; b[m,m/4] ; /* local variables hold initial allocation*/
for t = 1 , 2, 3, 4 {
  for i = 0 , ... , m/4 - 1 /* step t */
    for j = 0 , ... , m/4 - 1
      for k = 0 , ... , m-1 {
        offset = ((id+t) mod 4 )*(m/4) ;
        c[ i , j + offset ] += a[ i,k ] * b[ k,j ] } ;
      send b[ . , . ] to processor (id-1) mod 4 ;
      receive b[ . , . ] from processor (id+1) mod 4 ;
    }
}

```

41

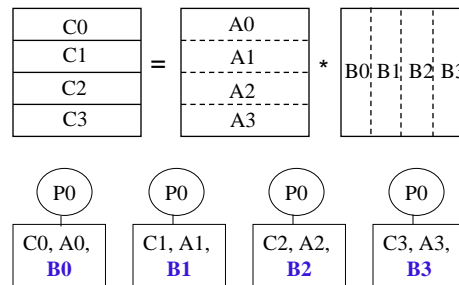


## Is a shared memory program more efficient??

```

Get processor id, /* 0, 1, 2, 3 */
for i = id*(m/4), ... , (id+1)*(m/4)-1
  for j = 0 , ... , m-1 {
    c[ i,j ] = 0 ;
    for k = 0 , ... , m-1
      c[ i,j ] += a[ i,k ] * b[ k,j ]
  }

```

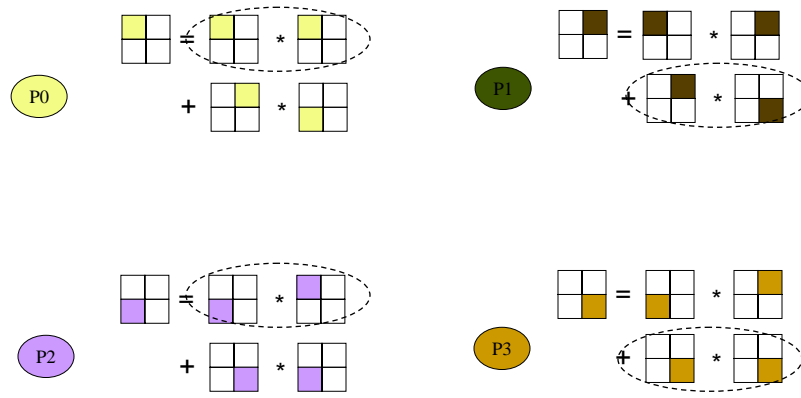


- Usually, the shared variables are physically distributed to local memories
- Many programming languages allow the programmer to specify how to distribute the storage of arrays into local memories:
  - Block distribution along each array dimension
  - Cyclic distribution with a specified length along each dimension
- Hence, user should keep in mind the access pattern when specifying the storage allocation of shared arrays.

42



### An alternate data partitioning scheme (more complex)



### Parallel sorting (odd-even transposition sort)

P0	P1	P2	P3	P4	P5	P6	P7	P8
5	2	8	6	3	7	9	4	1
5	2	8	3	6	7	9	1	4
2	5	3	8	6	7	1	9	4
2	3	5	6	8	1	7	4	9
2	3	5	6	1	8	4	7	9
2	3	5	1	6	4	8	7	9
2	3	1	5	4	6	7	8	9
2	1	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Each processor,  $P_i$ , owns a value,  $x_i$ ,  $i=0, \dots, K-1$

Each  $P_i$  executes

for  $t=1, \dots, K$

if  $(i+t)$  is odd and  $(i > 0)$ , then  $x_i = \max(x_i, x_{i-1})$

else if  $(i+t)$  is even and  $(i < K-1)$  then  $x_i = \min(x_i, x_{i+1})$

**Result:**  $x_0 < \dots < x_{K-1}$



## Discussion

- Can we write an odd-even sort, message passing, algorithm?

Processor,  $P_i$ , is storing a value,  $x$  /\* a local variable \*/

Each  $P_i$  executes

for  $t=1, \dots, K$  {

  if ( $t$  is odd) { if ( $i$  is odd) and ( $i < K-1$ ) {

    send  $x$  to processor  $i+1$  ;

$y$  = the value received from processor  $i+1$  ;

$x = \min(x, y)$  } ;

  if ( $i$  is even) and ( $i > 0$ ) {

    send  $x$  to processor  $i-1$  ;

$y$  = the value received from processor  $i-1$  ;

$x = \max(x, y)$  }

  };

  if ( $t$  is even) .....

**Result:** the value of  $x$  stored in  $P_i$  is smaller than the value of  $x$  stored in  $P_{i+1}$

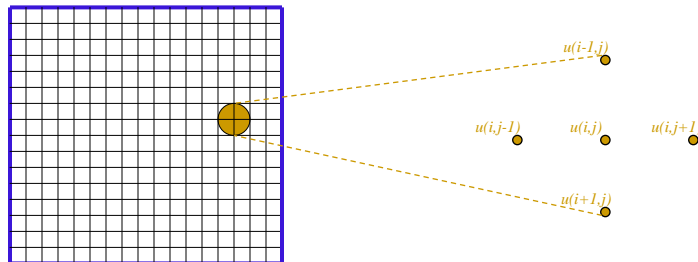
- How can we modify the algorithm if the number of data items to be sorted is much larger than the number of processors?

45



## Parallelizing an algorithm

### Finite differences for solving PDEs - an example

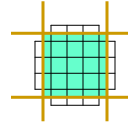
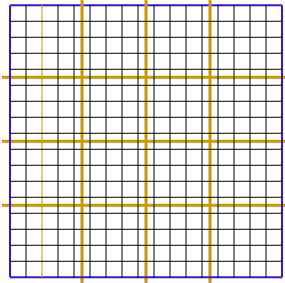


- Discretize the dependent variable into the values  $u(i,j)$ ,  $i,j = 0, \dots, n+1$  (at grid points)
- The values  $u(0,*)$ ,  $u(n+1,*)$ ,  $u(*,0)$  and  $u(*,n+1)$  are known (boundary conditions)
- To solve, iterate
 
$$u(i,j) = f(u(i,j), u(i-1,j), u(i,j-1), u(i,j+1), u(i+1,j)), \quad i,j = 1, \dots, n$$
 until convergence. The function  $f()$  depends on the form of the PDE.
- Convergence is when the maximum change in  $u(i,j) < \delta$

46



## Parallel finite differences for solving PDEs



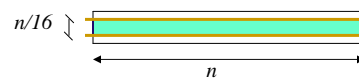
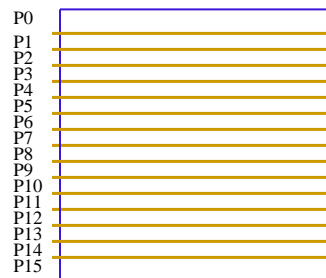
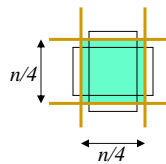
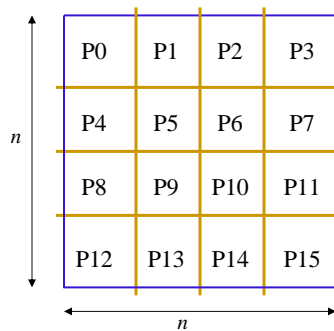
- Subdivide the grid into sub-grids and assign one sub-grid to each processor.
- Each processor will compute the values of  $u()$  in its sub-domain.
- Each processor will have to communicate at the beginning of each iteration to share boundary  $u()$  values with its four neighbors.
- Processors will have to communicate at the end of each iteration to check for convergence.

47



## Parallel finite differences for solving PDEs

- Which of the following two domain partitioning is more efficient??



48

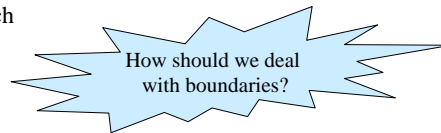
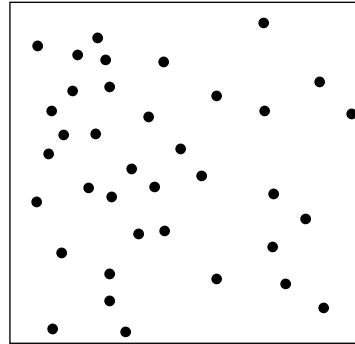


## Parallelizing an algorithm Particle-particle simulation - an example

- $N$  particles in a 2D area (or 3D volume)
- A gravitational force between every pair of particles (can be computed)
- A resultant force on each particle induces a motion.

### Discrete time simulation:

- Divide time into discrete steps,  $\Delta$ , and iterate over time.
- During each  $\Delta$ , compute the force on each particle –  $N^2$  forces.
- Compute the velocity and acceleration of each particle, and change its position accordingly.



49

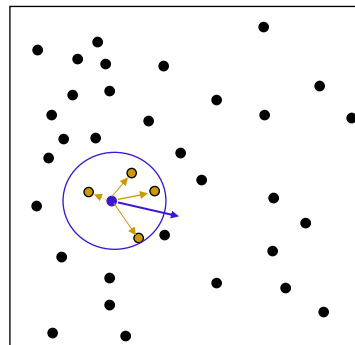


## Particle-particle simulation

- To reduce the computation in each iteration from  $O(N^2)$  to  $O(N)$ , when computing the force on a particle,  $P$ , consider only the effect of particles within a given radius,  $r$ , of  $P$ .
- Compute the new position of  $P$  at the end of the interval  $\Delta$ .

for  $t = 1, 2, 3, \dots$

for every particle,  $P$ , in the domain  
{ compute the resultant force on  $P$  ;  
change the position of  $P$  } ;

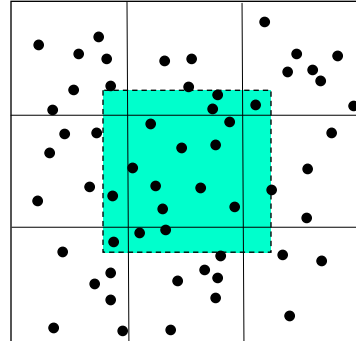


50



## Parallelizing the particle-particle simulation

- Partition the domain into sub-domains
- Assign one processor to each sub-domain
- Each processor simulates the motion of the particles in its sub-domain.
- Processors need to communicate the attributes of the particles in border bands of width  $r$ .
- After the new positions are computed, particles may change sub-domains – may have to move data to reassign particles to processors.



Note that sub-domains may not contain the same number of particles – load imbalance

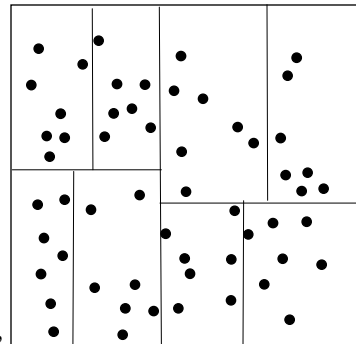
51



## Balanced partitioning of sub-domains

Partition the domain using the nested bisection scheme.

- Divide the domain, **vertically**, into two sub-domains with equal number of particles.
- Divide each of the two sub-domains, **horizontally**, into two sub-domains with equal number of particles.
- Divide each of the four sub-domains, **vertically**, into two sub-domains with equal number of particles.



52



## Speedup and efficiency.

- For a given problem  $A$ , of size  $n$ , let  $t_p(n)$  be the execution time on  $p$  processors, and  $t_1(n)$  be the execution time (of the best algorithm for  $A$ ) on one processor. Then,

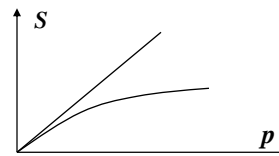
$$\text{Speedup } S_p(n) = t_1(n) / t_p(n)$$

$$\text{Efficiency } E_p(n) = S_p(n) / p$$

Speedup is between 0 and  $p$ , and efficiency is between 0 and 1.

- Linear Speedup means that  $S$  is linear with  $p$  (perfectly scalable machine)
- If speedup is independent of  $n$ , then the algorithm is said to be perfectly scalable.

- Minsky's conjecture:  
Speedup is logarithmic in  $p$



53



## Amdahl's law.

Let  $f$  be the fraction of a program that has to be performed serially, then, using  $p$  processors, the maximum possible speedup is:

$$S < \frac{1}{f + (1-f)/p}$$

Hence, even with unlimited number of processors, the speedup cannot be larger than  $1/f$ .

- Algorithms for the same problem may have different values of  $f$ .
- The above formula ignores the effect of  $n$  on  $f$  (serial portion of code may be fixed, independent of the size of the problem).
- Ignores the effect of memory:
  - Negative effect  $\longrightarrow$  conflict.
  - Positive effect  $\longrightarrow$  more memory, cache and registers.
- Ignores the effect of communication.

54



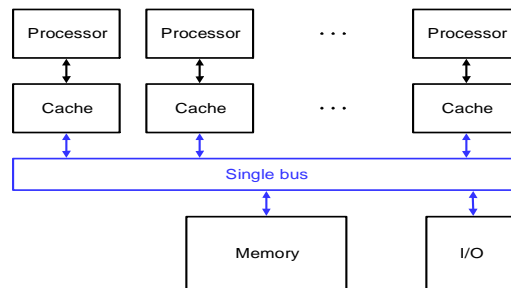
## Cache coherence in SMP's

55



### The problem of cache coherence in SMPs

- Different caches may contain different value for the same memory location.



Time	Event	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
0				1
1	CPU A Reads X	X = 1		1
2	CPU B reads X	X = 1	X = 1	1
3	CPU A stores 0 into X	X = 0	X = 1	0

56



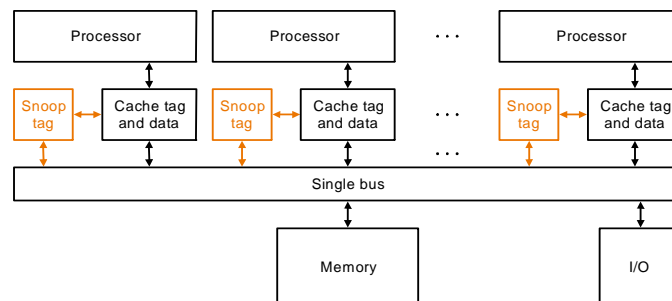
## Approaches to cache coherence

- Do not cache shared data
- Do not cache writeable shared data
- Use snoop caches (if connected by a bus)
- If memory is physically distributed memory and not connected by a bus, then need another solution – directory-based protocols.

57



## Snooping cache coherence protocols



- Each processor monitors the activity on the bus
- **On a read miss**, all caches check to see if they have a copy of the requested block. If yes, they supply the data (will see how).
- **On a write miss**, all caches check to see if they have a copy of the requested data. If yes, they either invalidate the local copy, or update it with the new value.
- Can have either *write back* or *write through* policy (the former is usually more efficient, but harder to maintain coherence).

58



### Example: Write Invalidate

Processor Activity	Bus Activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
				0
CPU A Reads X	Cache Miss for X	0		0
CPU B Reads X	Cache Miss for X	0	0	0
CPU A writes 1 to X	Invalidation for X	1		0
CPU B Reads X	Cache Miss for X	1	1	1



### Example: Write update

Processor Activity	Bus Activity	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
				0
CPU A Reads X	Cache Miss for X	0		0
CPU B Reads X	Cache Miss for X	0	0	0
CPU A writes 1 to X	update for X	1	1	1
CPU B Reads X	Cache hit for X	1	1	1



## An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory can be:
  - Clean in all caches and up-to-date in memory (Read-Only), or
  - Dirty in exactly one cache (Read/Write), or
  - Not in any caches
- Correspondingly, we record the state of each block in a cache as one of:
  - Shared : block can be read (clean, read-only)
  - Exclusive : cache has only copy, its writeable, and dirty
  - Invalid : block contains no data
- Read misses: cause all caches to snoop bus
  - If a cache has the block “exclusively”, it supplies the block and changes its state to “share”
- A write to a clean block is treated as a write miss -- all other caches that have the block should invalidate that block.

61



## Example

- Assumes A1 and A2 map to same cache block B.
- Initial cache state is invalid

Event	In P1's cache	In P2's cache
	B = invalid	B = invalid
P1 writes 10 to A1	A1 = 10 (exclusive)	B = invalid
P1 reads A1	A1 = 10 (exclusive)	B = invalid
P2 reads A1	A1 = 10 (shared)	A1 = 10 (shared)
P2 write 20 to A1	B = invalid	A1 = 20 (exclusive)
P2 writes 40 to A1	B = invalid	A2 = 40 (exclusive)

62



## Directory-based coherence protocols

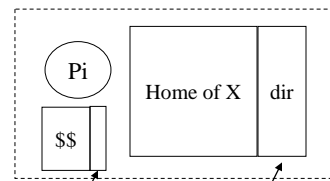
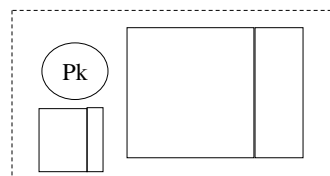
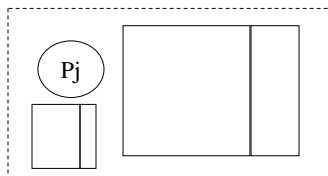
- For shared address space over physically distributed memory
- The home of each block is determined by its address.
- A controller decides if access is Local or Remote
- As in snooping caches, the state of every block in every cache is tracked in that cache (exclusive/dirty, shared/clean, ...)
- In addition, with each block in memory keep a directory to track the state of each block:
  - *Shared/clean*: cached in one or more processors, and memory is up-to-date
  - *Uncached*: no processor has it (not valid in any cache)
  - *Exclusive/dirty*: 1 processor (*owner*) has data; memory out-of-date
- The directory also keep track of *which processor(s)* cached the block

63



## Directory-based coherence protocols

Case 1:  
X is in the *uncached* state in home directory



Keeps track of  
state of cached blocks

Keeps track of  
where X is cached

### Possible scenario:

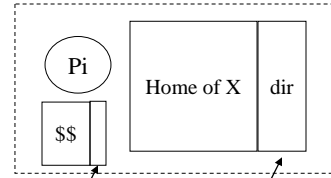
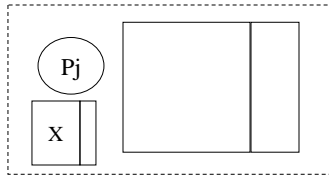
- $P_j$  reads X
- Then  $P_j$  writes to X

64



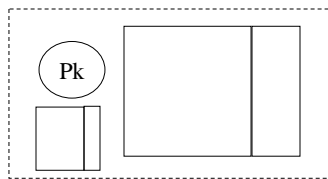
## Directory-based coherence protocols

Case 2:  
X is *exclusive* in home directory and  
*exclusive* in the cache of Pj



Keeps track of  
State of cached blocks

Keeps track of  
where X is cached



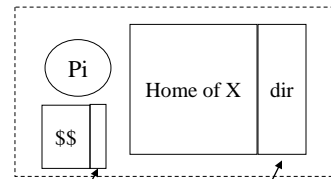
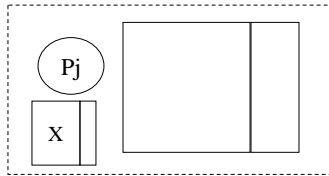
**What happens if:**

- Pj reads X
- Then Pj writes to X
- Then Pk reads X



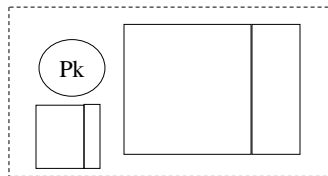
## Directory-based coherence protocols

Case 2:  
X is *exclusive* in home directory and  
*exclusive* in the cache of Pj



Keeps track of  
State of cached blocks

Keeps track of  
where X is cached



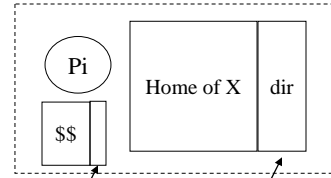
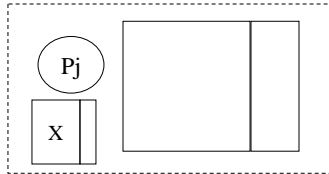
**What happens if:**

- Pk writes to X



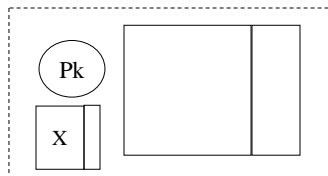
## Directory-based coherence protocols

Case 3:  
X is *shared* in home directory and  
*clean* in the caches of Pj and PK



Keeps track of  
State of cached blocks

Keeps track of  
where X is cached

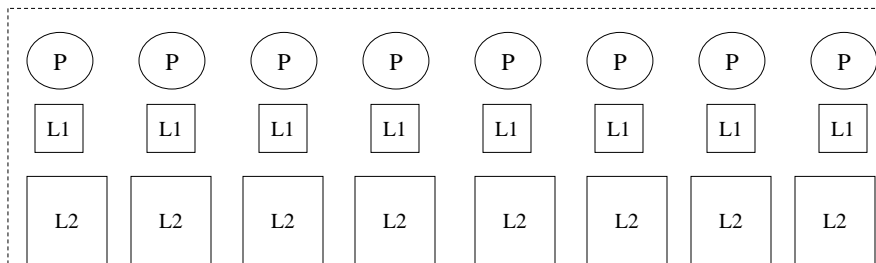


### Possible scenario:

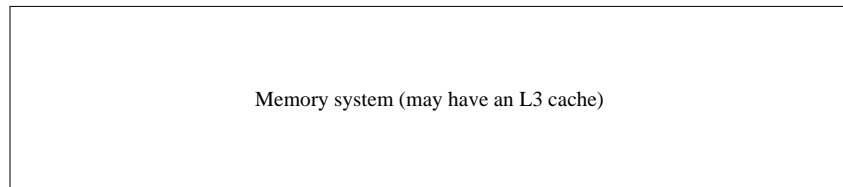
- Pj reads X
- Then Pk writes into X



## The cache configuration in CMPs



The L2 cache modules can be either shared or private to the processors on the chip.





## A request for a phase-oriented application

Phase I

Phase II

Phase III