Introduction to parallel computing

Rami Melhem
Department of Computer Science

Seminar Organization

1) Introductory lectures (probably 4)

2) Paper presentations by students (2/3 per short/long class)
   - GPU architecture and algorithms
   - Reliability of HPC (High Performance Computing)
   - Distributed graph processing
   - Innovative memory technology for parallel architectures

3) Class projects (typically 2 students per project)

Flynn’s hardware taxonomy:

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

S/M I S/M D

- S for single
- M for multiple

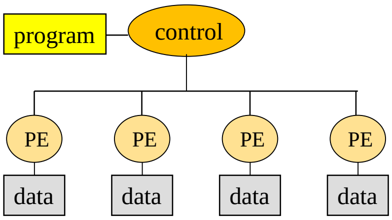
- I for instruction
- D for data.

- 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.

SIMD (two flavors)

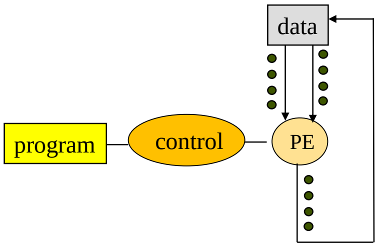
1) Synchronous, lockstep execution

All PEs execute the same instructions on different data

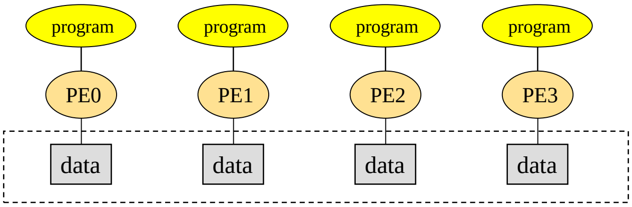


2) Vector processing

The same instruction is repeatedly executed on different data



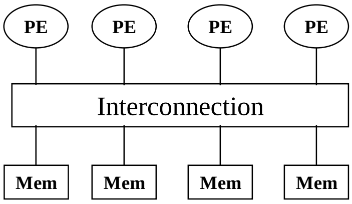
MIMD



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.

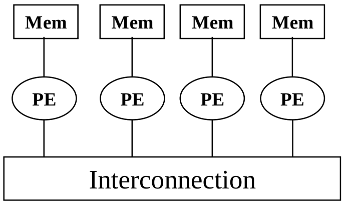
Many flavors depending on the **memory architecture** and the **address space** of each PE (the address space is the range of memory addresses that the PE can access).

Physical memory Architectures



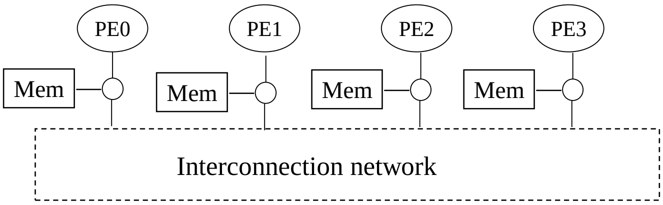
EX: Bus, crossbar, tree, multistage network, mesh, …

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



Distributed memory
Communicate through messages or remote memory access (put/get)

Distributed shared memory systems


PE0

PE1

PE2

PE3

Mem

Mem

Mem

Mem

Interconnection network

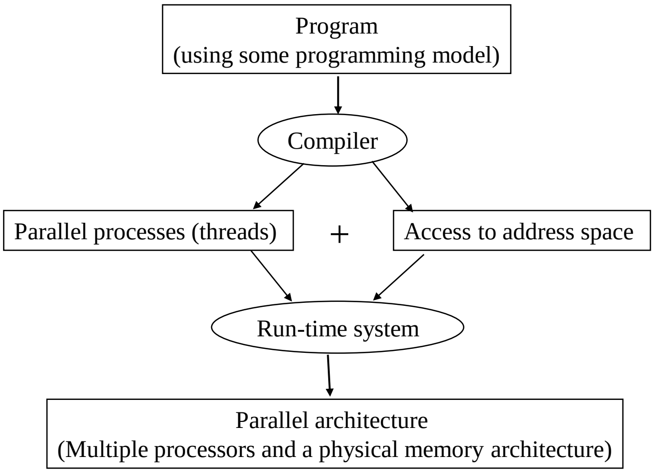
Shared address space, but physically distributed memory.

- 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”)

A system may have shared memory among groups of nodes while communication among groups is through messages

Programming parallel computing systems


Program
(using some programming model)

Compiler

Parallel processes (threads)

+

Access to address space

Run-time system

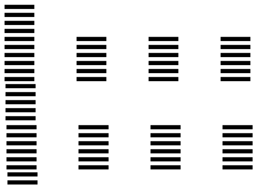
Parallel architecture
(Multiple processors and a physical memory architecture)

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

Parallel Programming Models (control threads - processes).

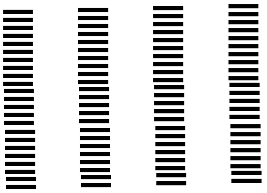
1) Start with one control thread, and create other threads when needed

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.

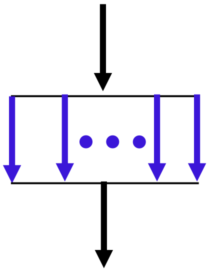
Example - Pthreads

```
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

Example - OpenMP

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

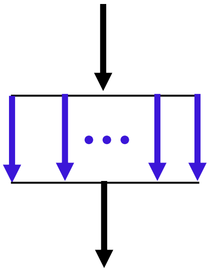


```
% Result of execution
Start
Hello World
Hello World
Hello World
Hello World
Done
```

The user can control the number of parallel threads by setting the environment variable setenv OMP_NUM_THTREADS 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

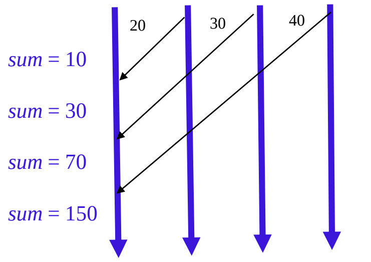
Warning: the loop iterations should be independent (no loop carried dependences)

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

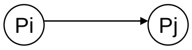
processID =	0	1	2	3
x =	10	20	40	80



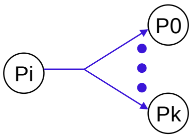
sum = 150 ?? ?? ??

Type of messages

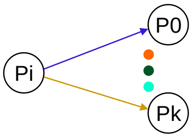
- **Point-to-point**: one processor sends a message to another processor



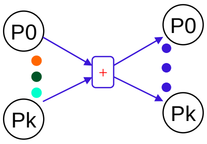
- **One-to-all**: one processor broadcasts a message to all other processors



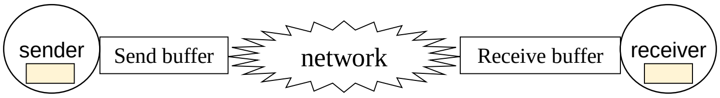
- **One-to-all personalized**: one processor sends a different message to each other processor



- **All-to-all**: each processor broadcasts a message to all other processors
- **Reductions**: the values from each processor are reduced (according to an operator) and broadcast to all processors



Blocking and non-blocking messages

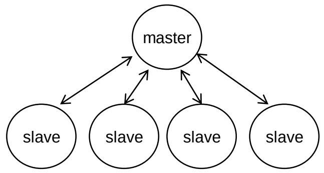


- Depending on the type of call, a process issuing a ***blocking send*** does not continue execution until
  - The message is copied to the *send buffer*
  - The message is sent on the network
  - The message reached the *receive buffer*
  - The message is received by the receiving process.
- A process 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.

The master/slave programming model

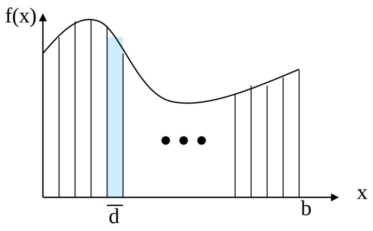
- Master divides the work into work_units ;
- While work is not done {

```
    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 ;
}
```



Numerical integration - a master/slave approach

Example: using distributed memory (may also use shared memory??)

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 ;
```

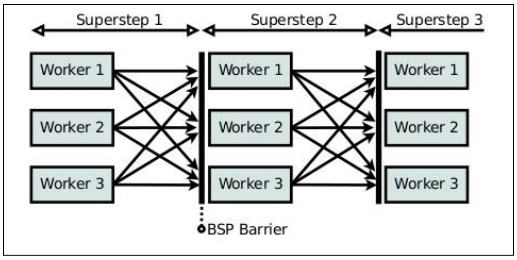
Program for processors 1 , … , *K*-1 (slaves)

```
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?

The BSP programming model

- Introduced by Valiant in the 90’s

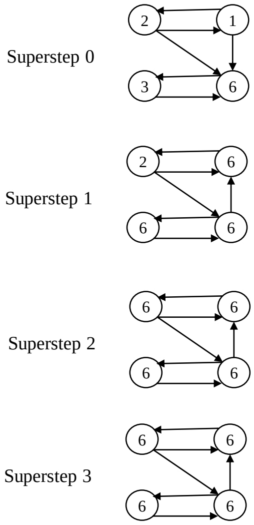


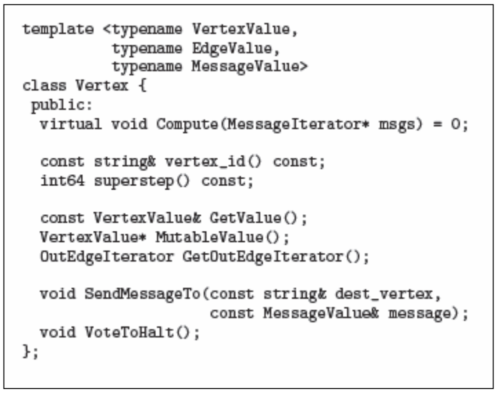
```
Repeat{

    Compute ;
    Send messages ;
    Receive message ;
    Barrier syncronization;
    Reduction to check convergence;

} until converge
```

- Pregel: A System for Large Scale Graph Processing (think like a vertex)

Think like a vertex

- Find the maximum values in the nodes of a graph





```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

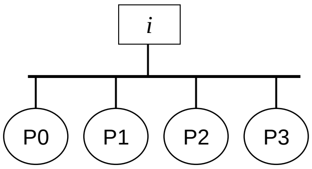
The **Think like a vertex** paradigm may apply to either shared or distributed memory models

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 ;
}
```

Read *i* from memory
Add 1 to *i*
Write *i* to memory



- 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

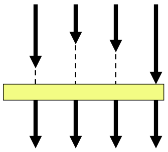
```
Declare a lock
#pragma omp
   {  get the lock ;
        i = i + 1  ;
        release the lock ;
   }
```

Most parallel languages provides ways to declare and use locks or critical sections

Synchronization (barriers)

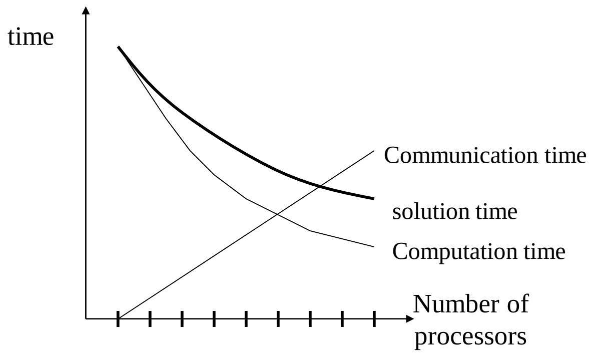
What is the output of the following Pthread program??

```
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] ;
}
```



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

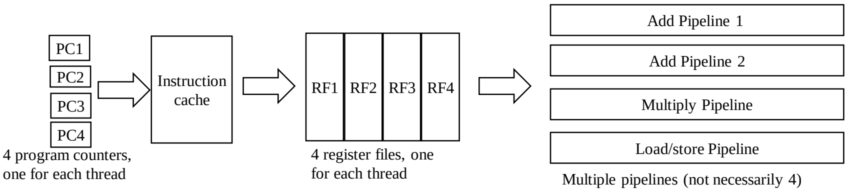
Hardware Multi-threading

- Software-based thread context switching (Posix Threads)
  – Hardware traps on a long-latency operation
  – Software saves the context of the current thread, puts it on hold and starts the execution of another ready thread
  – Relatively large overhead (saving old context and loading new context)
  – Context = registers, PC, stack pointer, pointer to page table, ….

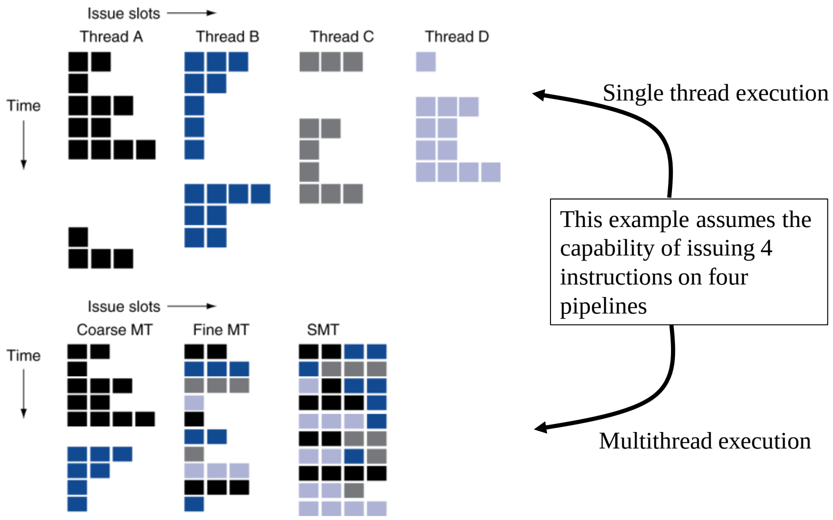
- Hardware-based multithreading
  – Threads = user defined threads or compiler generated threads
  – Replicate registers (including PC and stack pointer)
  – Hardware-based thread-context switching (fast)
  – May multithread independent processes if TLB is replicated

Scheduling multiple threads

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (eg, data hazards)
- SMT: Simultaneous Multi Threading
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when ready
  - Dependencies within each thread are handled separately


PC1
PC2
PC3
PC4
4 program counters, one for each thread
Instruction cache
RF1
RF2
RF3
RF4
4 register files, one for each thread
Add Pipeline 1
Add Pipeline 2
Multiply Pipeline
Load/store Pipeline
Multiple pipelines (not necessarily 4)

SMT Examples

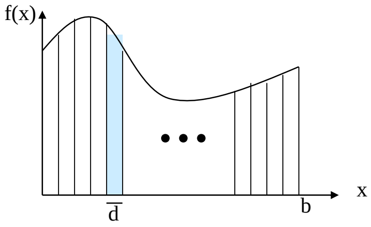


This example assumes the capability of issuing 4 instructions on four pipelines

Example of Parallel algorithms

Parallelizing an algorithm
Numerical integration - a message passing example

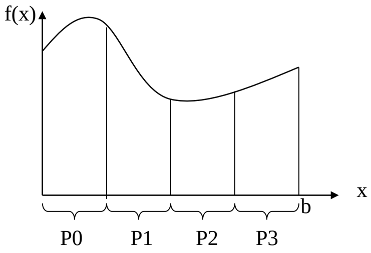
```
n = 1000 ;  d = b / n ;
area = 0 ;
for (i=0 ; i < n ; i++) {
   x = i * d + d / 2;
   area = area + f(x) * d ;
}
```



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.

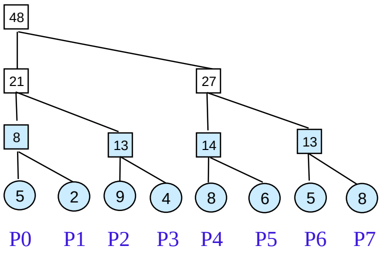


Numerical integration
still need to accumulate the partial areas

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 };
```

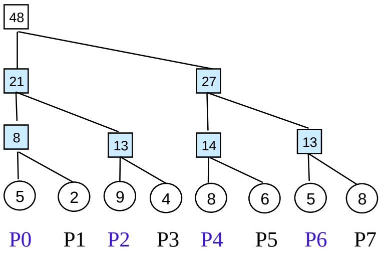


Numerical integration
still need to accumulate the partial areas

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 };
```

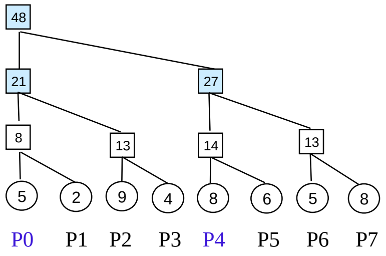


Numerical integration
still need to accumulate the partial areas

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 (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 };
```



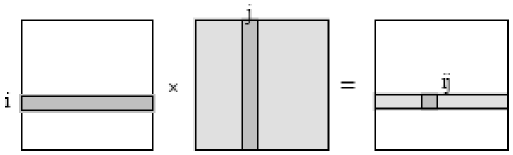
Accumulating the partial areas
Assuming K processors

```
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 $2^i$ == $2^{i-1}$)
          send p_area to processor id - $2^{i-1}$;
elseif (id mod $2^i$ == 0)
        { receive the p_area from processor id + $2^{i-1}$ ;
          add the received value to the local p_area };
}
```

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:

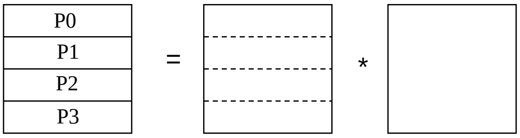
```
Get the processor id    /* 0 <= id < $m^2$ */
i = id / m  ; j = id mod m ;
c[ i,j ] = 0  ;
for k = 0, … , m-1
     c[ i,j ] =+ a[ i,k ] * b[ k,j ]
```

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

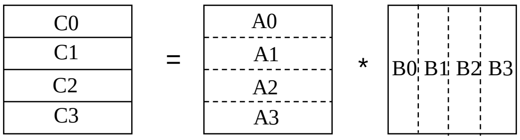
mxm matrix/matrix multiplication using 4 processors
(*m* is a multiple of 4)

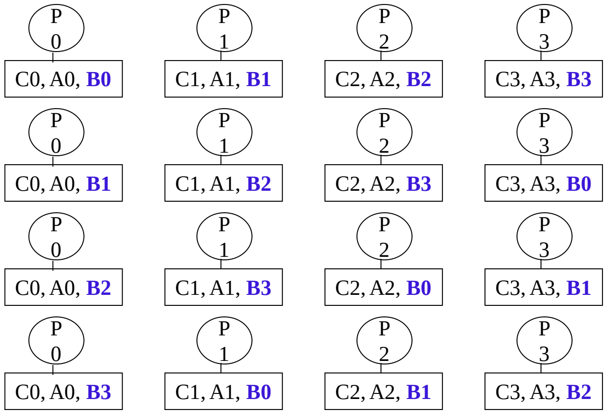
```
for i = 0, … , m-1
  for j = 0, … , m-1 {
    c[ i,j ] = 0 ;
    for k = 0, … , m-1
        c[ i,j ] =+ a[ i,k ] * b[ k,j ]
  }
```

```
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 ]
  }
```



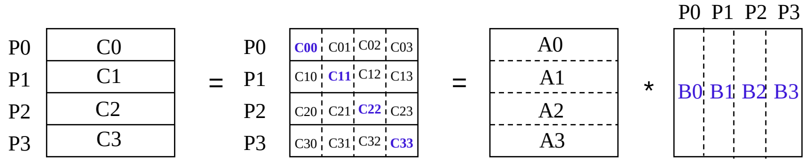
What if we have a distributed memory system?





	P0	P1	P2	P3
Start with this memory allocation	C0, A0, **B0**	C1, A1, **B1**	C2, A2, **B2**	C3, A3, **B3**
Then shift the allocation of B	C0, A0, **B1**	C1, A1, **B2**	C2, A2, **B3**	C3, A3, **B0**
Shift once more	C0, A0, **B2**	C1, A1, **B3**	C2, A2, **B0**	C3, A3, **B1**
And yet, once more	C0, A0, **B3**	C1, A1, **B0**	C2, A2, **B1**	C3, A3, **B2**

The distributed memory program



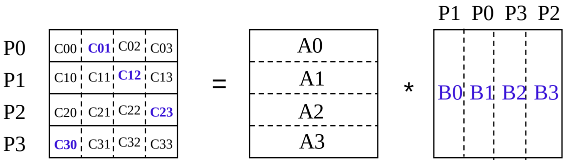
```
float c[m/4,m] = 0   ;
float a[m/4,m] ; b[m,m/4] ;            /* local variables hold initial allocation*/
for i = 0, … , m/4 - 1   /* id is the processor identifier */
  for j = 0 , … , m/4 -1
     for k = 0, … , m-1
        c[ i , j+ id*(m/4)] =+ a[ i,k ] * b[ k,j ]
```

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)]

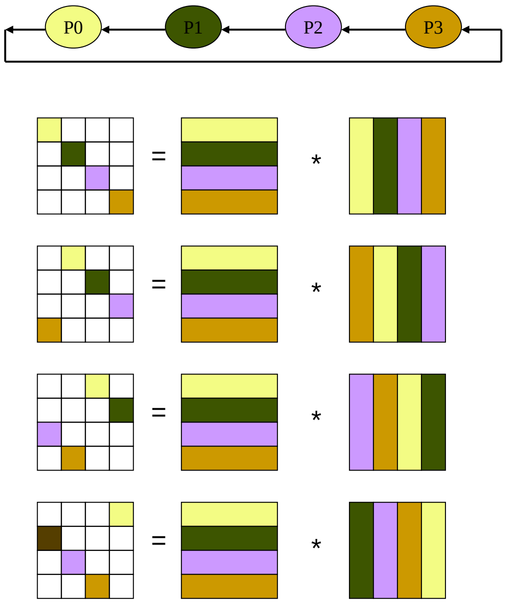
The distributed memory program



```
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 ] ;
```

What happens if we do the receive first?

The distributed memory program



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

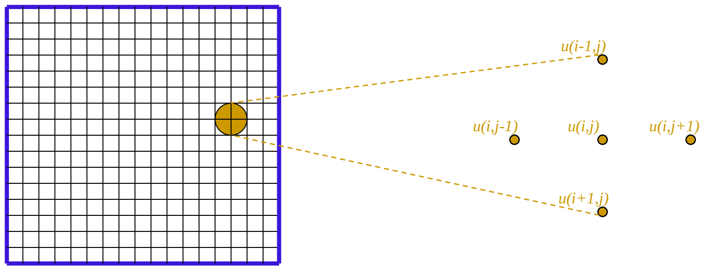
- Can we write an odd-even sort, message passing, algorithm?

```
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?

Parallelizing an algorithm
Finite differences for solving PDEs - an example



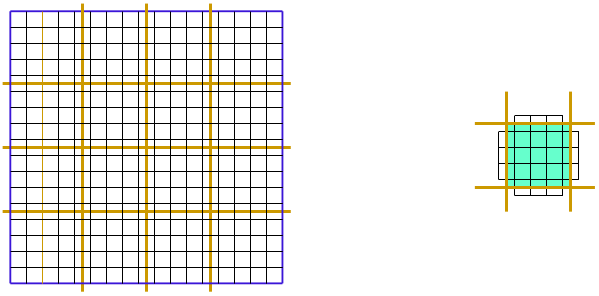
- Discretize the dependent variable into the values $u(i,j)$, $i,j = 0,\ldots,n+1$ (at grid points)
- The values $u(0,*)$, $u(n+1,*)$, $u(*,0)$ and $u(*,n+1)$ are known (boundary conditions)
- To solve, iterate

$$u(i,j) = f\,(u(i,j)\,,\,u(i-1,j)\,,\,u(i,j-1)\,,\,u(i,j+1)\,,\,u(i+1,j))\;, \qquad i,j = 1,\ldots,n$$

until convergence. The function $f()$ depends on the form of the PDE.

- Convergence is when the maximum change in $u(i,j) < \delta$

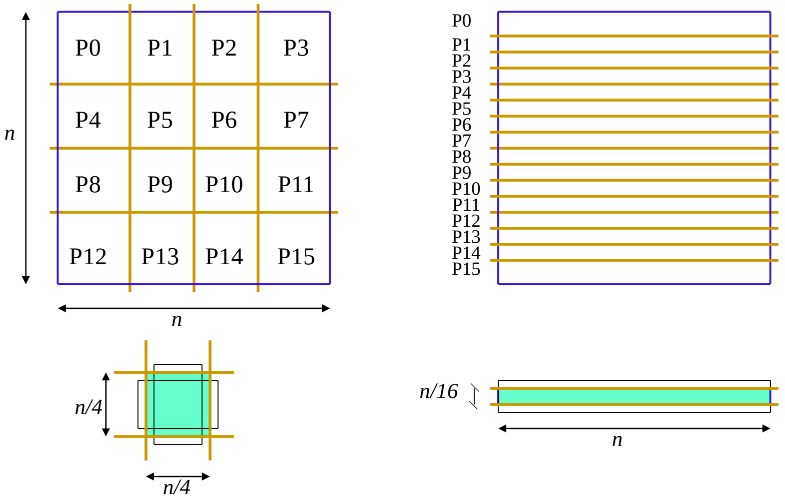
Parallel finite differences for solving PDEs



- 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.

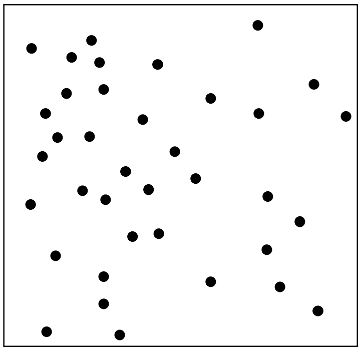
Parallel finite differences for solving PDEs

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



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:

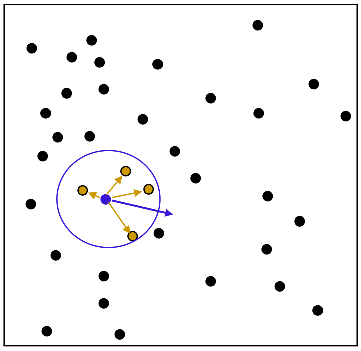
- Divide time into discrete steps, Δ, and iterate over time.
- During each Δ, compute the force on each particle – N^2 forces.
- Compute the velocity and acceleration of each particle, and change its position accordingly.

How should we deal with boundaries?

Particle-particle simulation

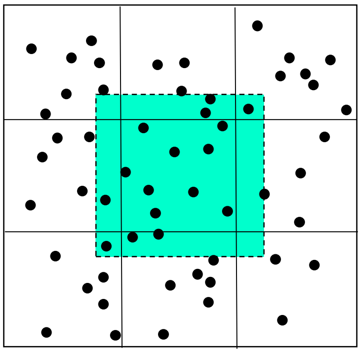
- To reduce the computation in each iteration from $O(N^2)$ to $O(N)$, when computing the force on a particle, *P*, consider only the effect of particles within a given radius, *r*, of *P*.

- Compute the new position of *P* at the end of the interval Δ .



```
for t = 1, 2, 3, …
   for every particle, P, in the domain
    { compute the resultant force on P ;
      change the position of P } ;
```

Parallelizing the particle-particle simulation



- Partition the domain into sub-domains
- Assign one processor to each sub-domain
- Each processors simulates the motion of the particles in its sub-domain.

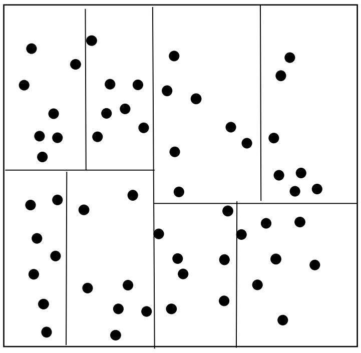
- Processors need to communicate the attributes of the particles in border bands of width *r*.

- 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 inbalance

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.