Parallel Architectures & Parallelization Principles

R. Govindarajan
CSA/SERC, IISc
govind@iisc.ac.in

Overview

- Introduction
- Parallel Architecture
- Parallelization Steps
- Example
  - Shared Address Space
  - Distributed Address Space

Acknowledgments:

Slides for this tutorial are taken from presentation materials available with the book "Parallel Computing Architecture: A Hardware/Software Approach" (Culler, Singh and Gupta, Morgan Kaufmann Pub.) and the associated course material. They have been suitably adapted.

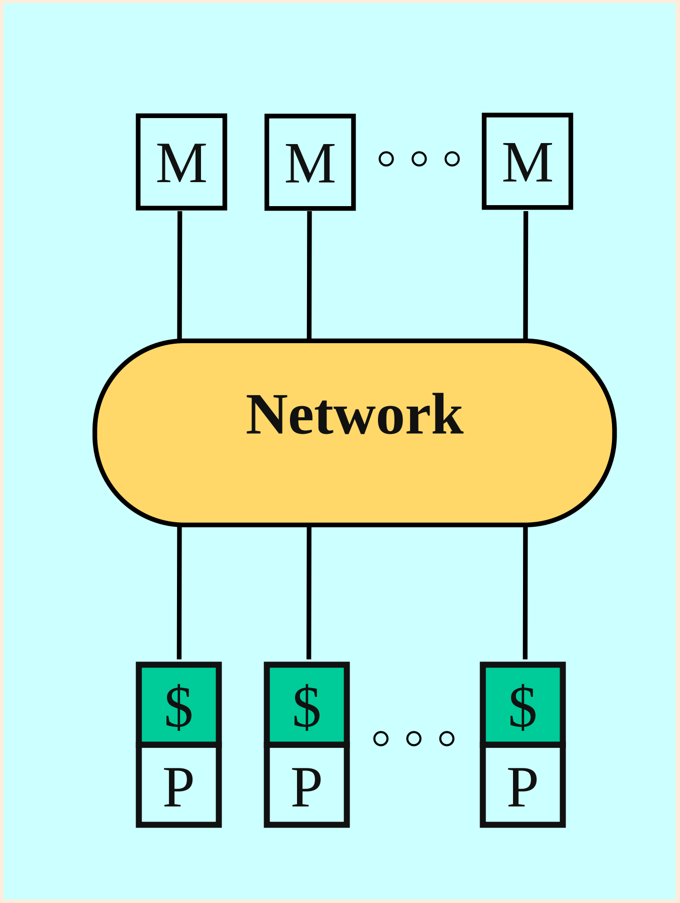
Introduction

Parallel Machine: a computer system with

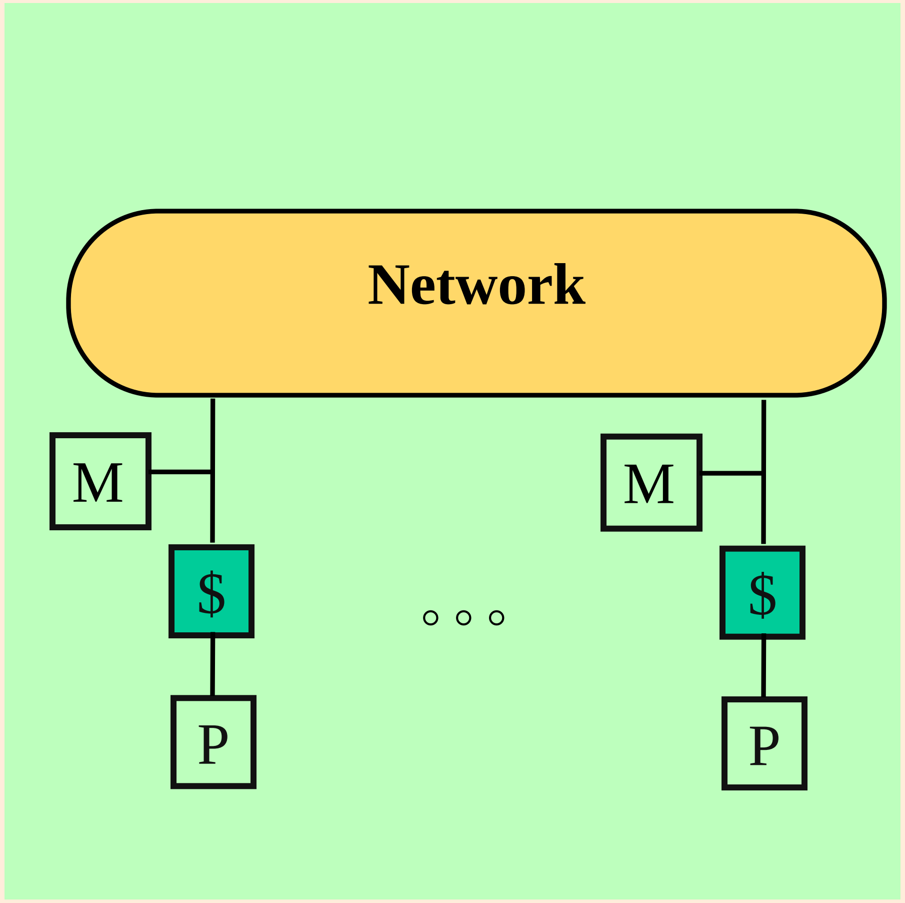
- More than one processor
- Processors interacting with each other
- Typically solving a single problem

- Multiprocessors
- Multicores
- Clusters
- Network of Workstations

Parallel Architecture: Shared Memory



Centralized Shared Memory



Distributed Shared Memory

Shared Memory Architecture

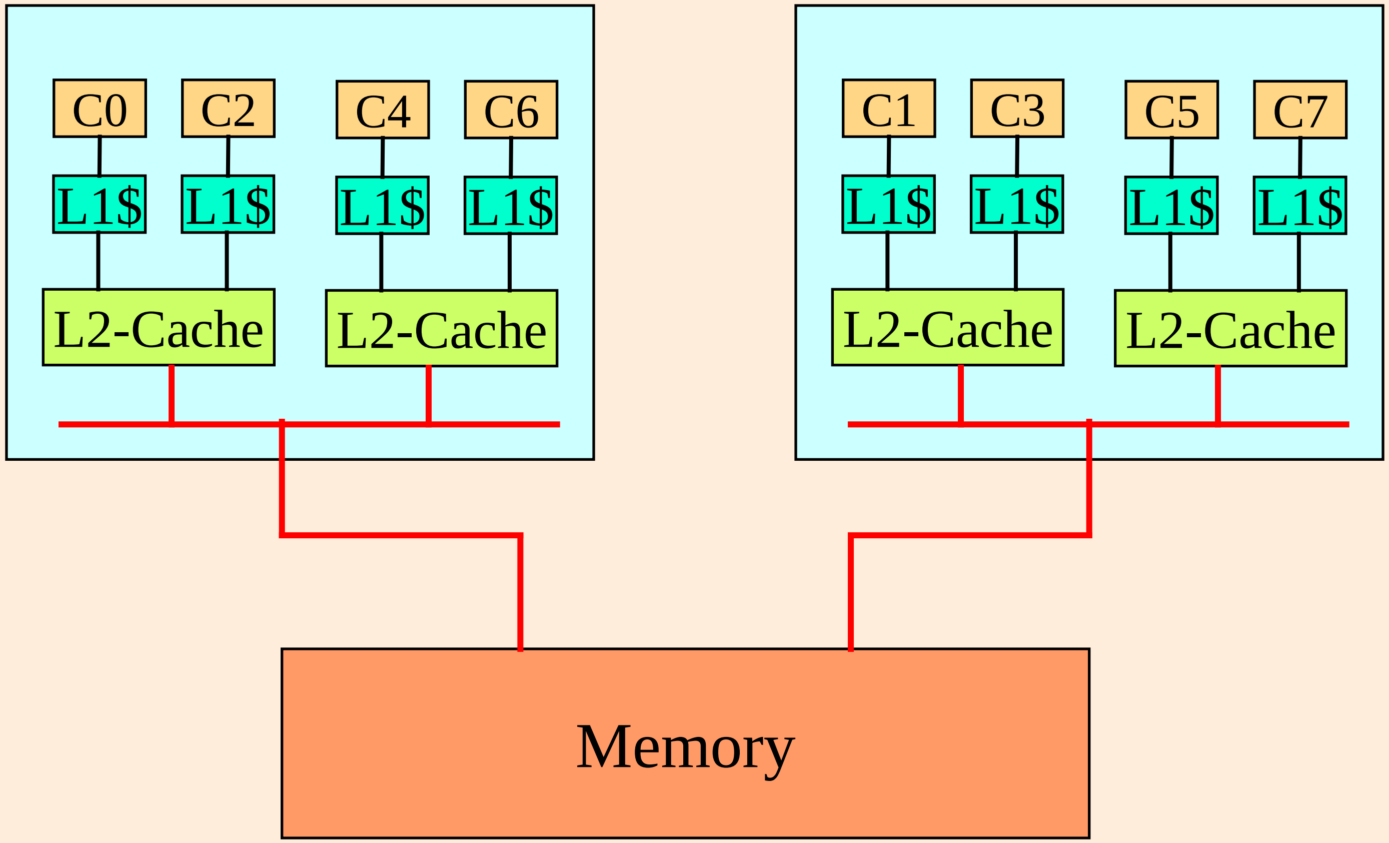
Uniform Memory Access (UMA) Architecture

Non-Uniform Memory Access (NUMA) Architecture

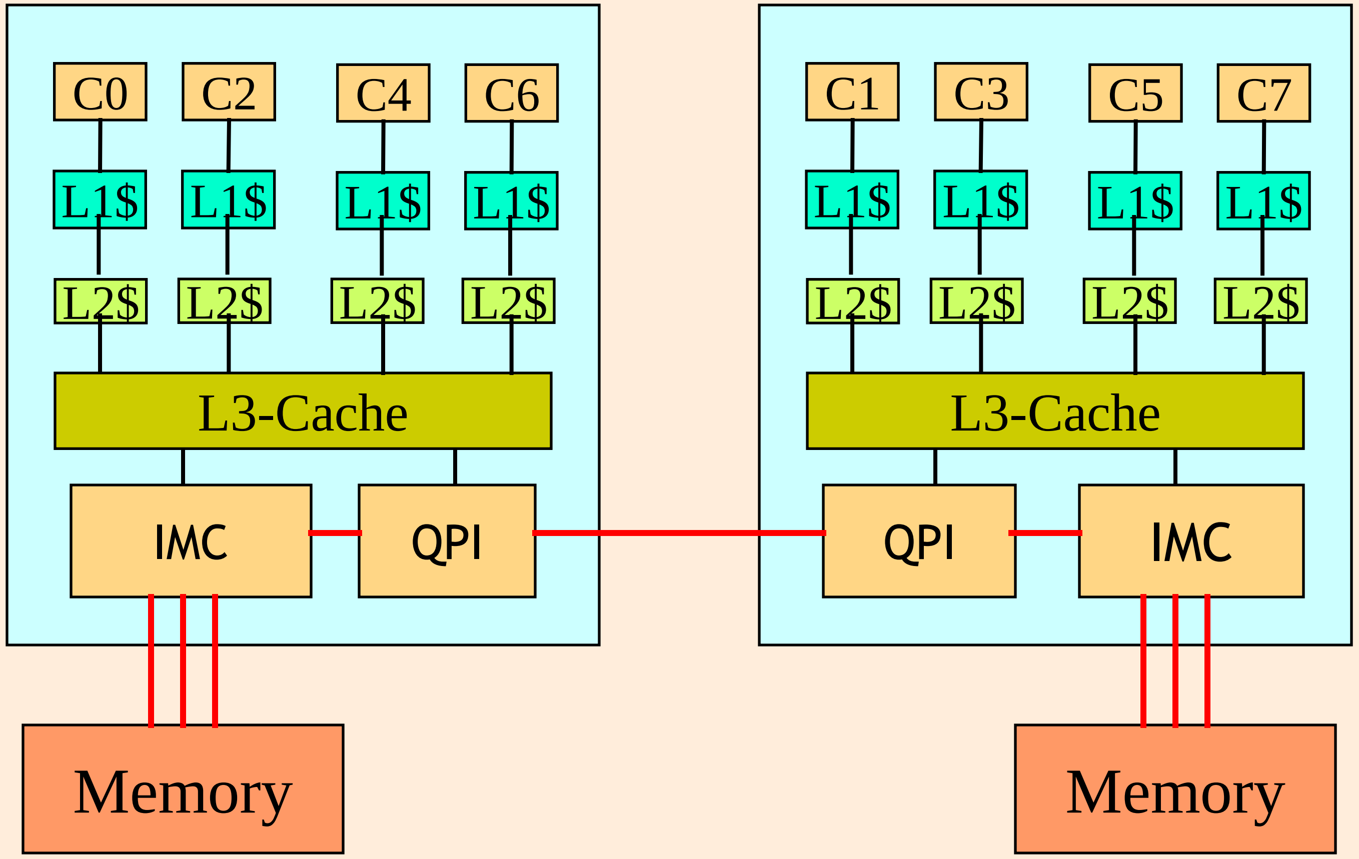
Centralized Shared Memory

Distributed Shared Memory

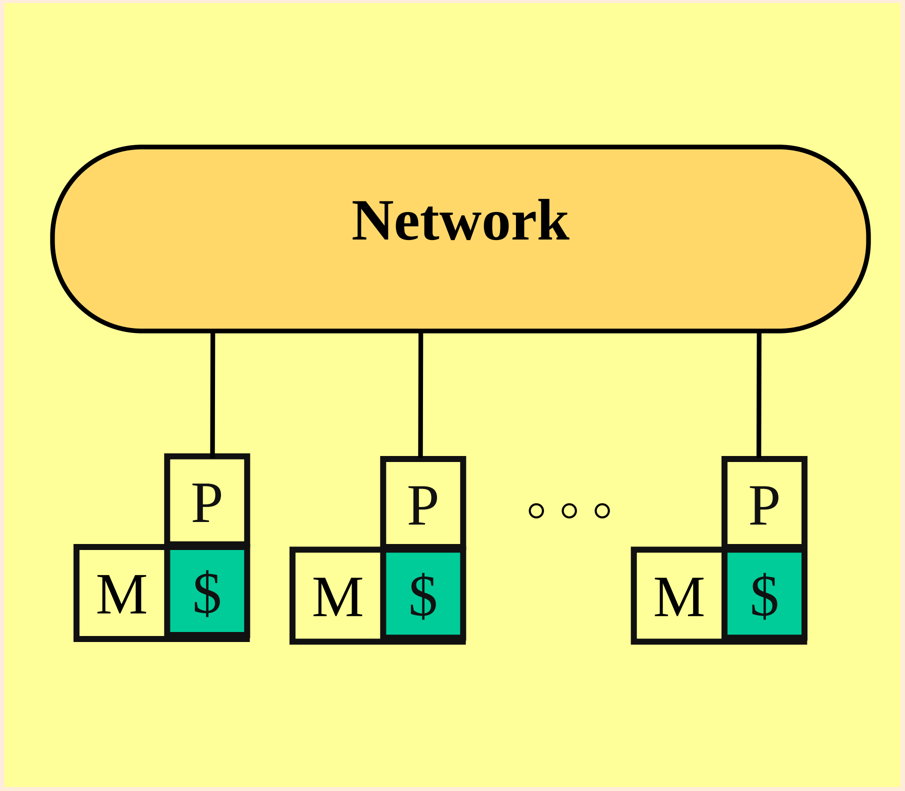
UMA Architecture



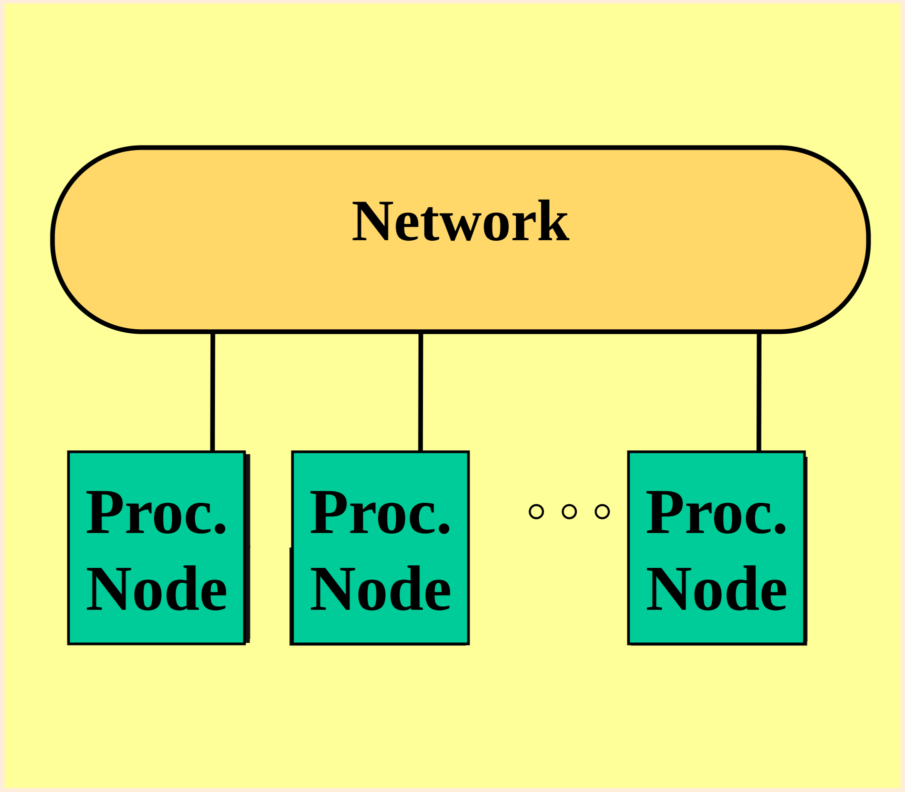
NUMA Architecture



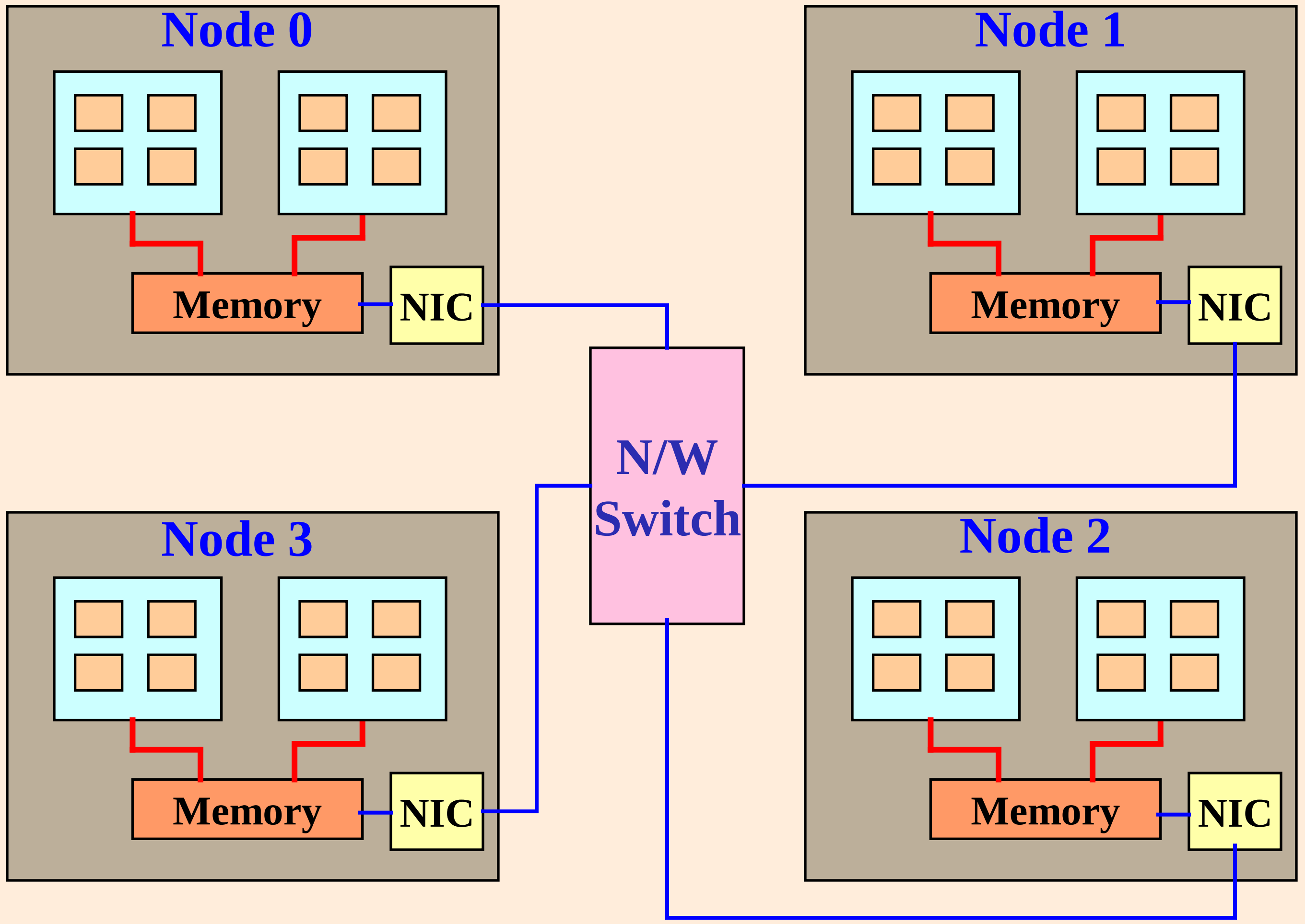
Distributed Memory Architecture



Distributed Memory Architecture



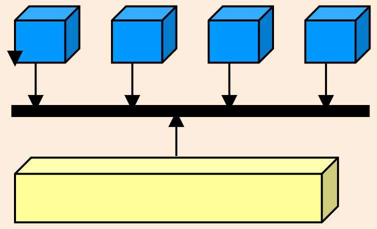
Hybrid Architecture

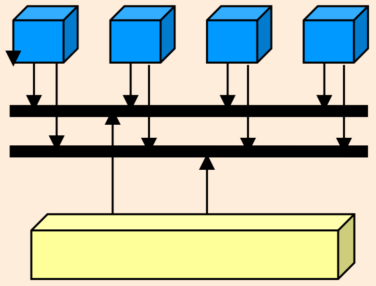


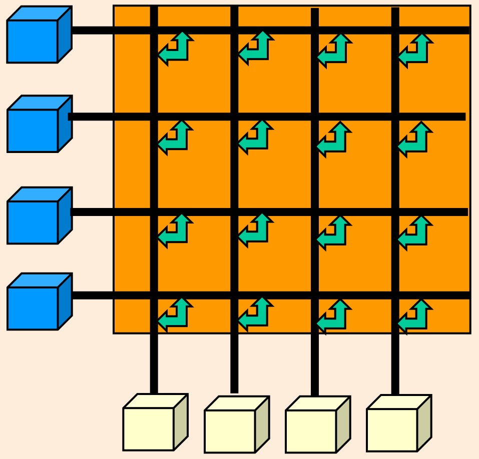
Parallel Architecture: Interconnection Network

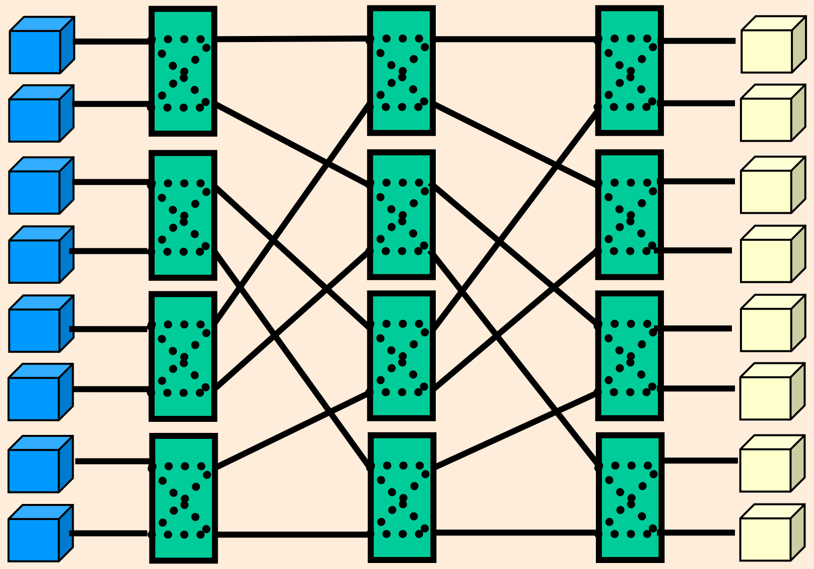
- Indirect interconnects: nodes are connected to interconnection medium, not directly to each other
  - ➢Shared bus, multiple bus, crossbar, MIN
- Direct interconnects: nodes are connected directly to each other
  - ➢Topology: linear, ring, star, mesh, torus, hypercube
  - ➢Routing techniques: how the route taken by the message from source to destination is decided

Indirect Interconnects


Shared bus

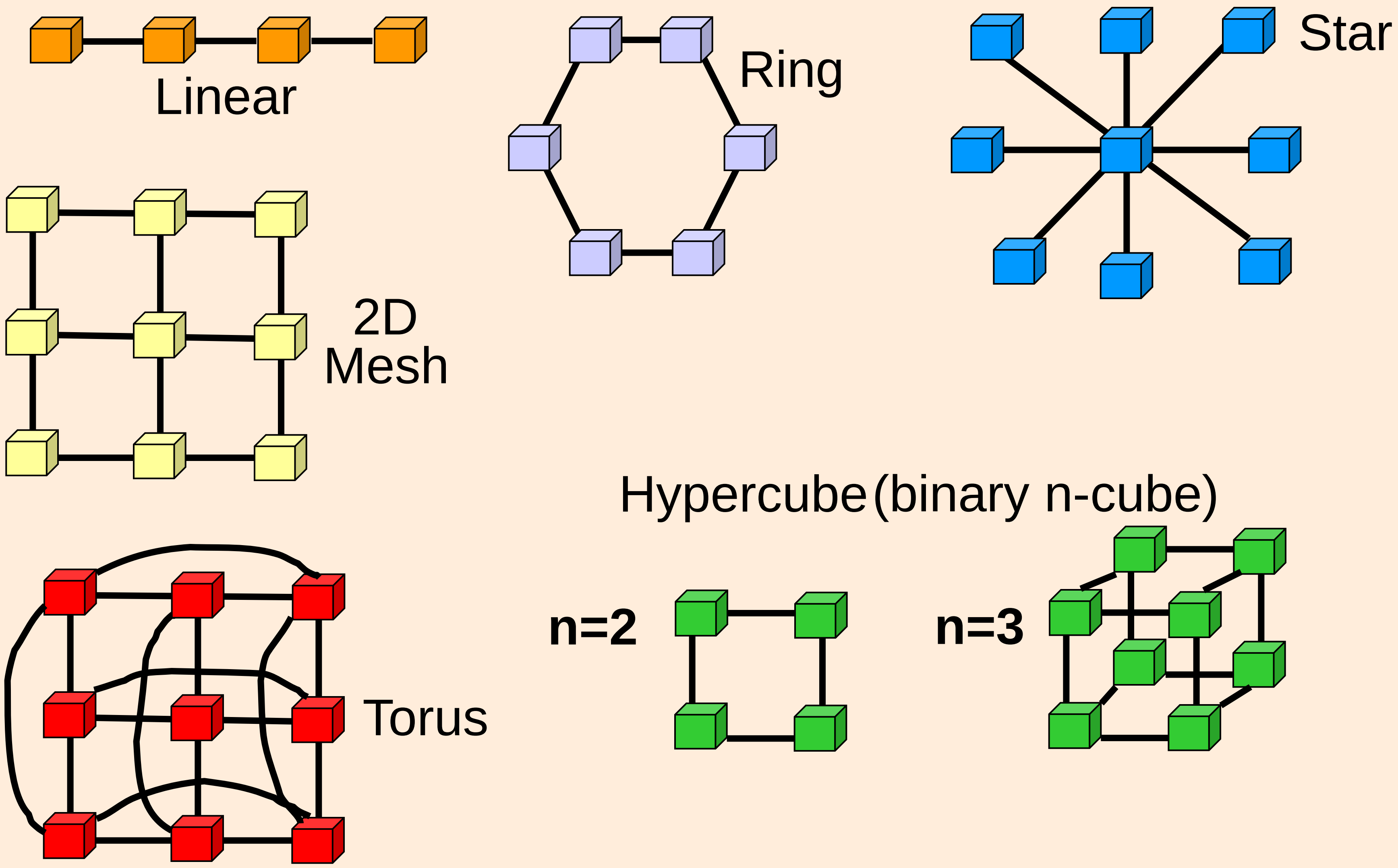

Multiple bus


Crossbar switch



Multistage Interconnection Network

Direct Interconnect Topologies



Space of Parallel Computing

Parallel Architecture

- Shared Memory
  - Centralized shared memory (UMA)
  - Distributed Shared Memory (NUMA)
- Distributed Memory
  - A.k.a. Message passing
  - E.g., Clusters

Programming Models

- What programmer uses in coding applns.
- Specifies synch. And communication.
- Programming Models:
  - Shared address space, e.g., OpenMP
  - Message passing, e.g., MPI

Parallel Programming

- Shared, global, address space, hence called *Shared Address Space*
  - Any processor can ***directly*** reference any memory location
  - Communication occurs implicitly as result of loads and stores
- Message Passing Architecture
  - Memory is private to each node
  - Processes communicate by messages

Definitions

- Speedup = $\frac{Exec.Time\ \ in\ \ UniProcesor}{Exec.Time\ in\ \boldsymbol{n}\ processors}$

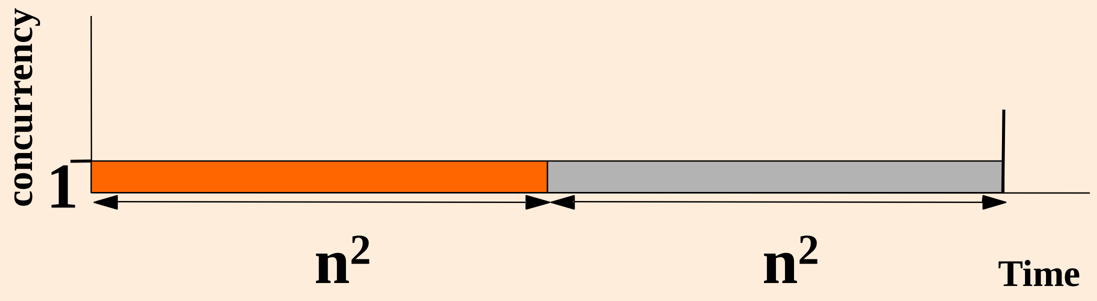
- Efficiency = $\frac{Speedup}{\boldsymbol{n}}$

- Amdahl’s Law:
  - For a program with s part sequential execution, speedup is limited by 1/s .

Understanding Amdahl's Law

Example: 2-phase calculation

- sweep over *n* x *n* grid and do some independent computation
- sweep again and add each value to global sum



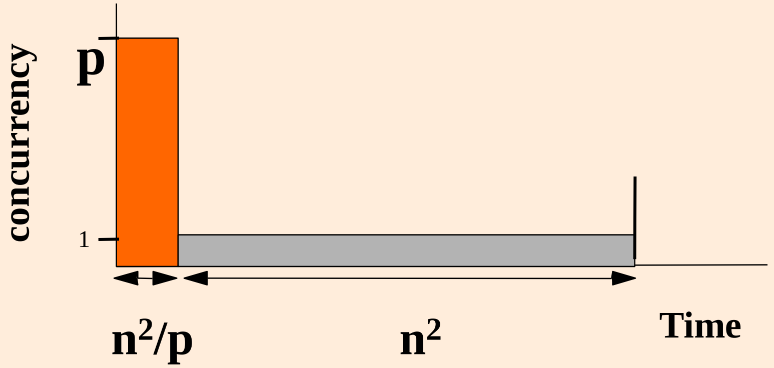
(a) Serial

Understanding Amdahl's Law

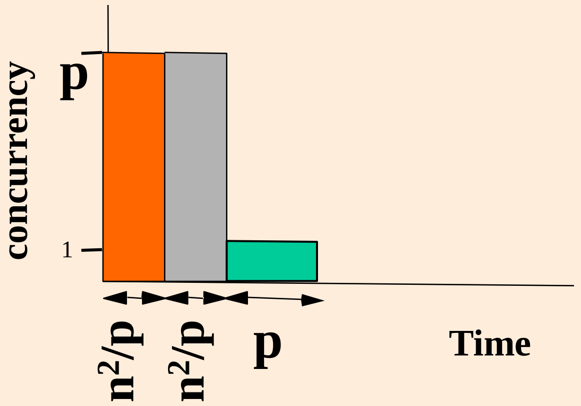
Execution time:

- Time for first phase = n^2/p
- Second phase serialized at global variable = n^2;
- Speedup = $(2n^2/(n^2 + n^2/p))$ or at most 2

- Localize the sum in p procs and then do serial sum.



(b) Naïve Parallel



(c) Parallel

Definitions

- Task
  - Arbitrary piece of work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - Fine-grained versus coarse-grained tasks
- Process (thread)
  - Abstract entity that performs the tasks
  - Communicate and synchronize to perform the tasks
- Processor:
  - Physical engine on which process executes

Tasks involved in Parallelizaton

- Identify work that can be done in parallel
  - work includes computation, data access and I/O
- Partition work and perhaps data among processes
- Manage data access, communication and synchronization

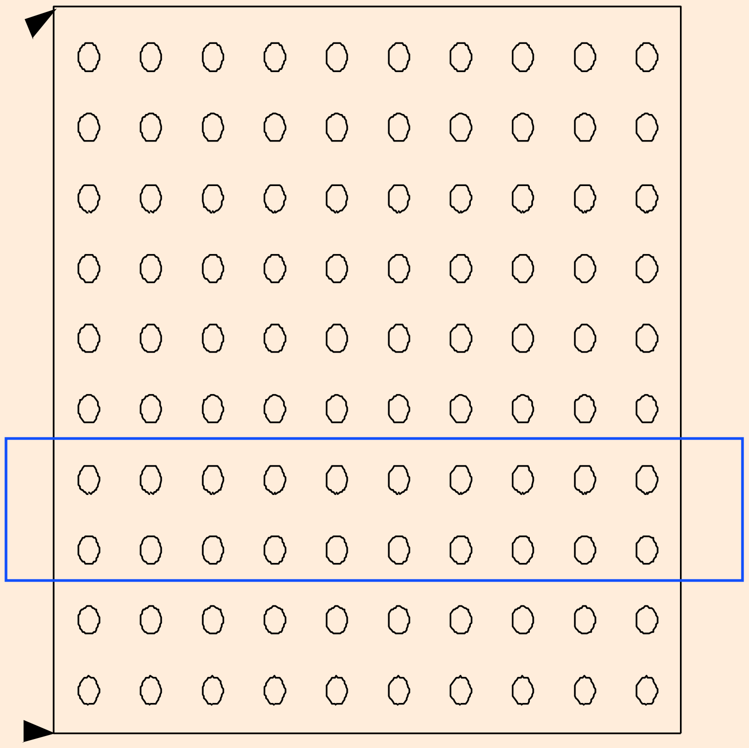
Parallelizing Computation vs. Data

- Computation is decomposed and assigned (partitioned) – ***task decomposition***
  - Task graphs, synchronization among tasks
- Partitioning Data is often a natural view too – ***data or domain decomposition***
  - Computation follows data: *owner computes*
  - Grid example; data mining;

Domain Decomposition: Example

- Some computation performed on all elts. of the array

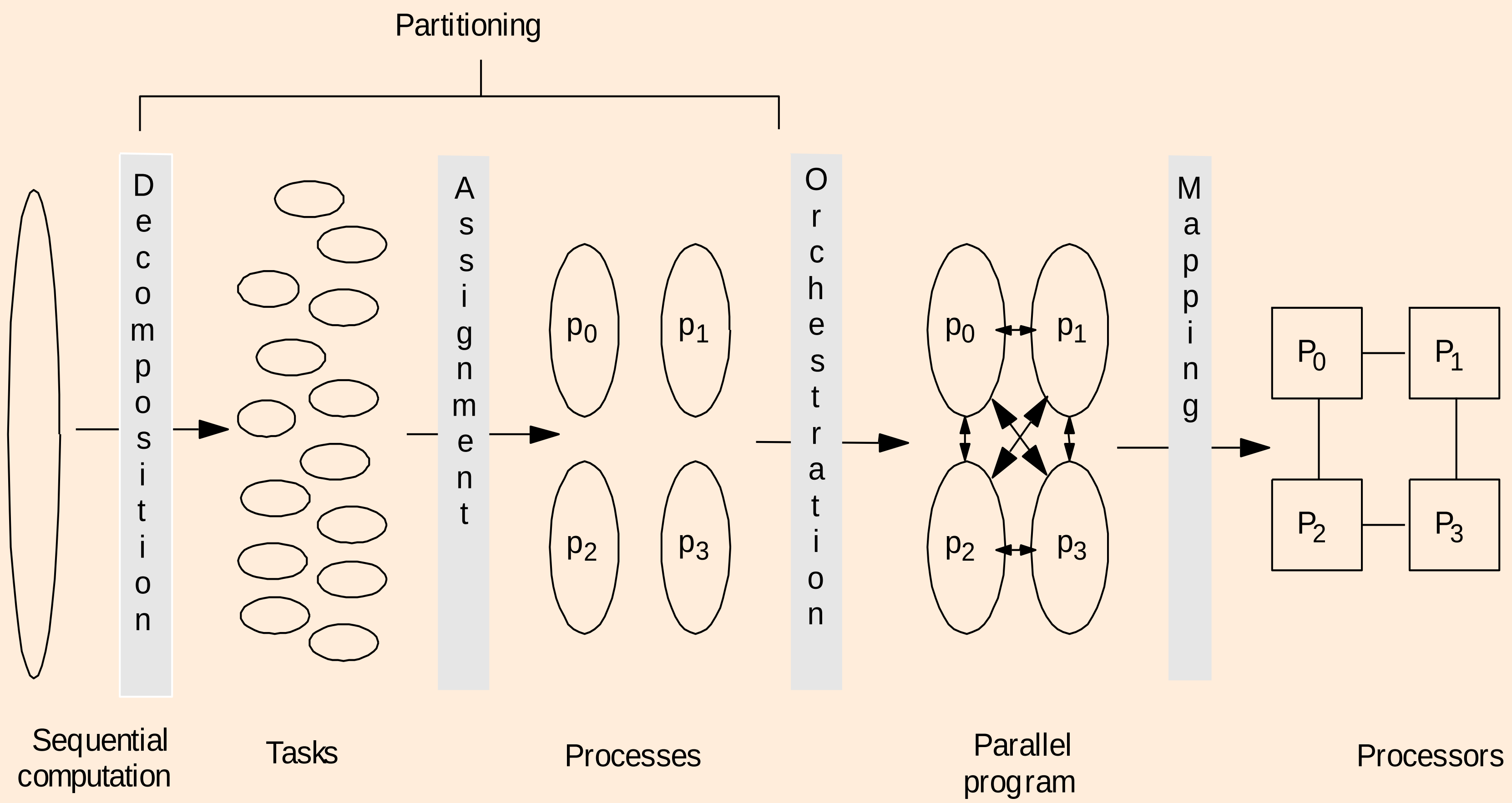
```
for i=1 to m
 for  j= 1 to n
    a[i,j] = a[i,j] + v[i]
```



Steps in Creating a Parallel Program

- Decomposition of computation into tasks
- Assignment of tasks to processes
- Orchestration of data access, communication, and synchronization.
- Mapping processes to processors

Steps in Creating a Parallel Program



Decomposition

- Identify concurrency
- Break up computation into tasks to be divided among processes
    - Tasks may become available dynamically
    - No. of available tasks may vary with time
- Goal: Expose available parallelism → enough tasks to keep processes busy

Assignment

- Specifies how to group tasks together for a process
  - Balance workload, reduce communication and management cost
- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - *Static* versus *dynamic* assignment
- Both decomposition and assignment are ***usually*** independent of architecture or prog model
  - But cost and complexity of using primitives may affect decisions

Orchestration

- Goals
  - Reduce cost of communication and synch.
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Choices depend on Programming Model, Communication abstraction, and efficiency of primitives
- Architecture should provide appropriate primitives efficiently

Mapping

- Two aspects:
  - Which process runs on which particular processor?
  - Will multiple processes run on same processor?
- Space-sharing
  - Machine divided into subsets, only one app at a time in a subset
  - Processes can be pinned to processors, or left to OS
- System allocation
- Real world
  - User specifies some aspects, system handles some

High-level Goals

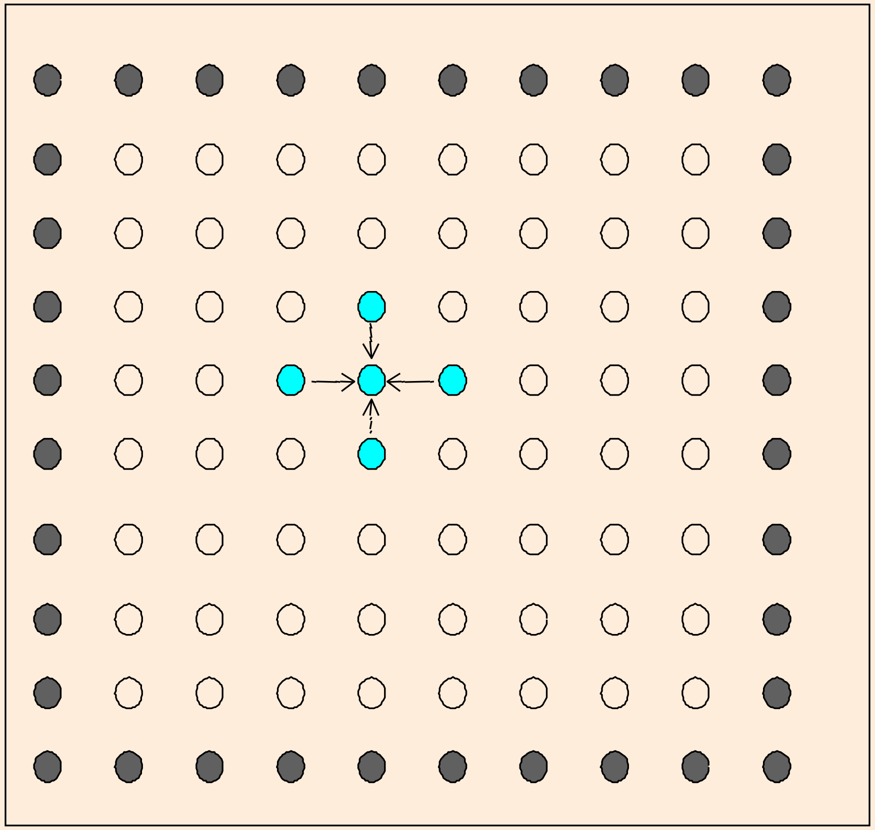
Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Example: Grid Solver

- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - difference from previous value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if it has converged
    - to within a tolerance parameter
  - updates array and iterate

Grid solver (Simple Version)



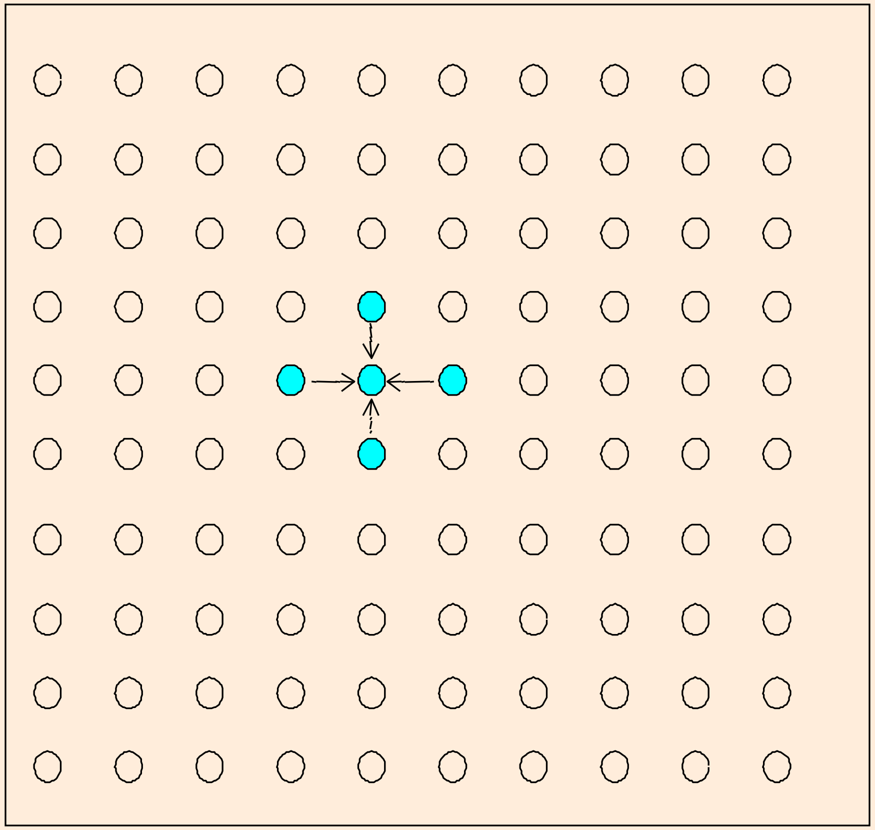
```
for i = 1 to n
    for j = 1 to n
    {
        B[i,j] = 0.2 * (A[i,j] +
               A[i-1,j] + A[i+1,j]+
               A[i,j-1] + A[i,j+1]);
        diff += abs(B[i,j] - A[i,j]);
    }
for i = 1 to n
    for j = 1 to n
        A[i,j] = B[i,j] ;
```

Decomposition & Assignment

```
for i = 1 to n
  for j = 1 to n
  {
     B[i,j] = 0.2 * (A[i,j] +
         A[i-1,j] + A[i+1,j]+
         A[i,j-1] + A[i,j+1]);
     diff += abs(B[i,j] - A[i,j]);
  }
for i = 1 to n
  for j = 1 to n
      A[i,j] = B[i,j] ;
```

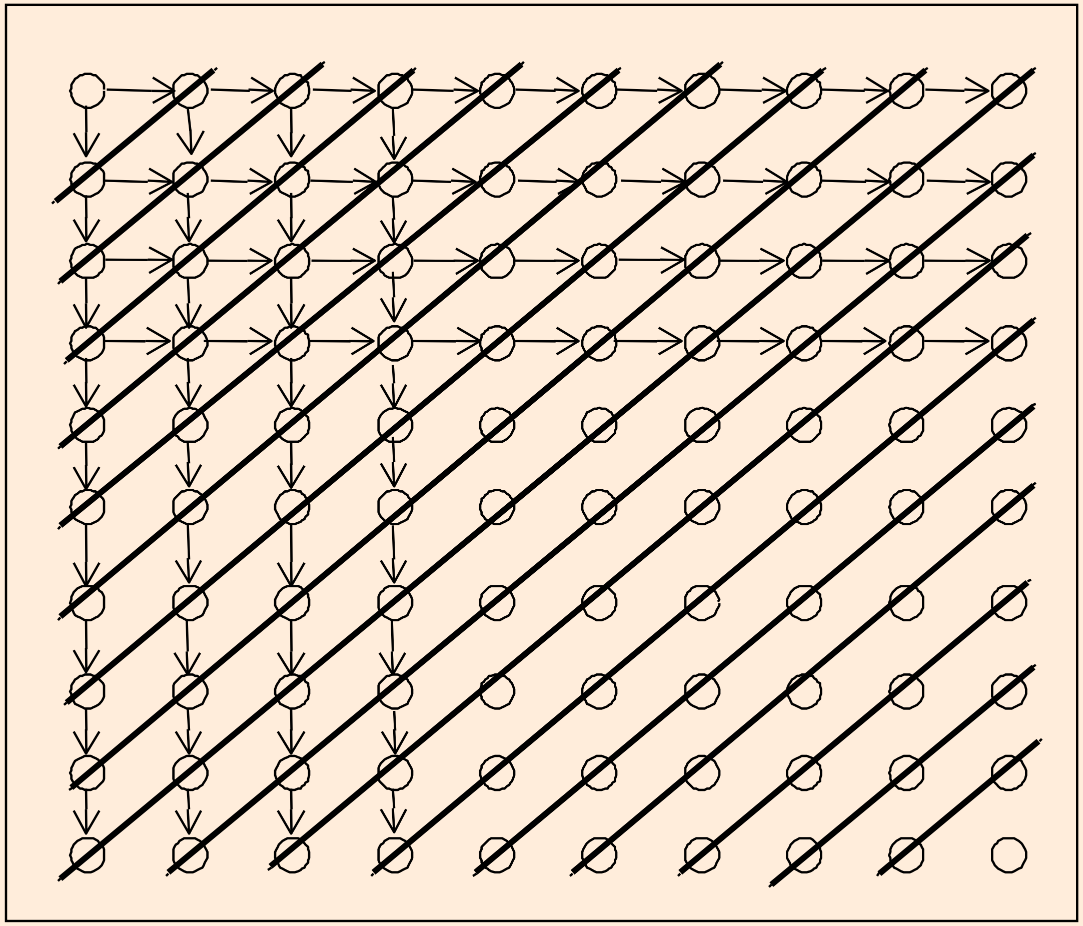
- Decomposition
  - Both i and j loops can be parallelized – no data dependences
  - Each grid point can be a task
  - To compute diff, some coordination would be required!
- Assignment
  - Each grid point
  - Each row or column
  - A group of rows or columns

Grid solver (Update-in-place Version)



```
for i = 1 to n
   for j = 1 to n
   {
      temp = A[i,j];
       A[i,j] = 0.2 * (A[i,j] +
          A[i-1,j] + A[i+1,j]+
          A[i,j-1] + A[i,j+1]);
      diff += abs(temp - A[i,j]);
    }
```

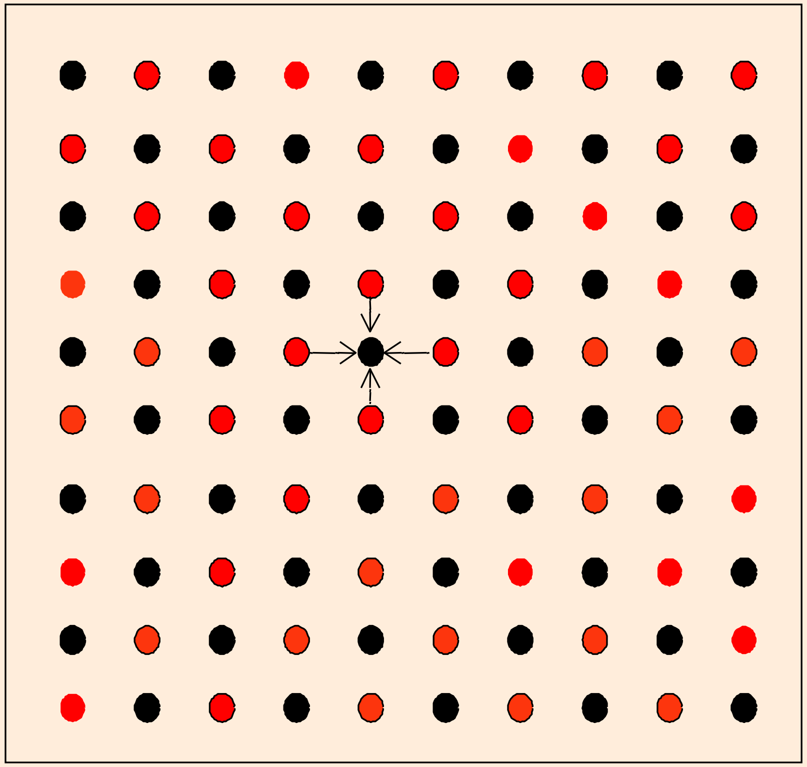
Decomposition & Assignment



- Decomposition
  - Dependence on both i and j loops
  - Each grid point can be a task
  - Need point-to-point synchronization -- Very expensive
- Assignment
  - Grid points along diagonal can a task
  - Restructure loop and global synchronization
  - Load imbalance

Exploiting Application Knowledge

- Reorder grid traversal: red-black ordering
- Red sweep and black sweep are each fully parallel:
- Global synch between them (conservative but convenient)
- Different ordering of updates: may converge slower



Red-Black Parallel Version

```
procedure Solve (A)   /*solve the equation system*/
    float **A;        /*A is an (n + 2)-by-(n + 2) array*/
begin
    int i, j, done = 0;
    float diff = 0, temp;
    while (!done) do  /*outermost loop over sweeps*/
        diff = 0;     /*initialize maximum difference to 0*/
        forall  i ← 1 to n  step 2 do/*sweep black points of grid*/
            forall j ← 2 to n+1 step 2 do
                temp = A[i,j]; /*save old value of element*/
                A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                      A[i,j+1] + A[i+1,j]);        /*compute average*/
                diff += abs(A[i,j] - temp);
            end forall
        end forall
        /* similarly forall loop for red points of grid
        if (diff/(n*n) < TOL) then done = 1;
    end while
end procedure
```

Ensure computation for all black points are complete!

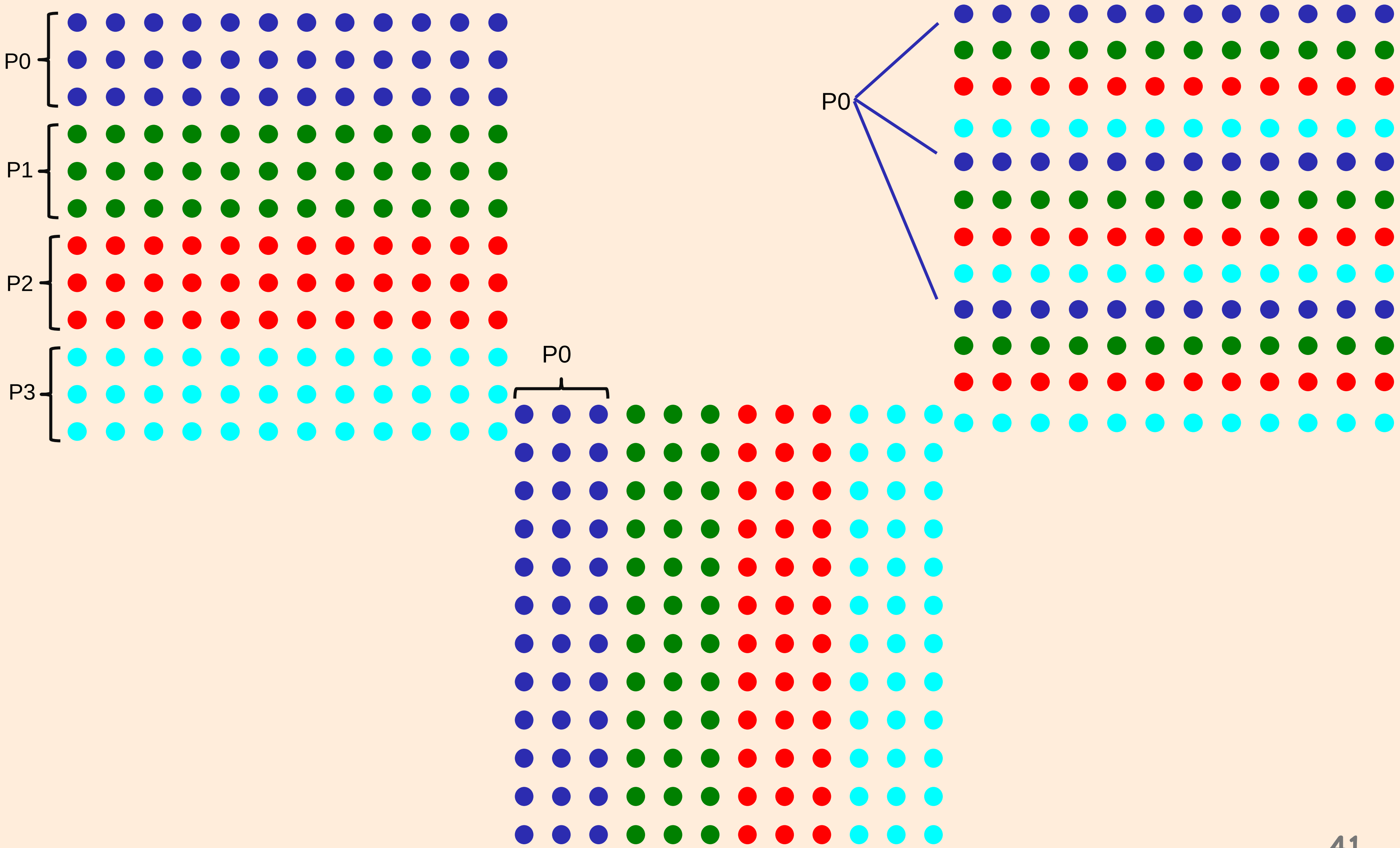
Red-Black Parallel Version (contd.)

- Decomposition into elements: degree of concurrency $n^2/2$; 2 global synchronizations per n^2 computation
- **forall** loop to express the parallelism.
- Too fine-grain parallelism $\Rightarrow$ group tasks to form a process.
- Decompose into rows?   Computation vs. communication overhead?

Assignment

- Static assignment: decomposition into rows
  - ***Block*** assignment of rows: Rows *i*(n/p), *(n/p) +1, … , (i+1)*(n/p) - 1* are assigned to process *i*
  - ***Cyclic*** assignment of rows: process *i* is assigned rows *i, i+p, …*
- Dynamic assignment
  - get a row index, work on the row, get a new row, …
- Concurrency? Volume of Communication?

Assignment (contd.)



Orchestration

- Different for different programming models/architectures
  - Shared address space
    - Naming: global addr. Space
    - Synch. through barriers and locks
  - Distributed Memory /Message passing
    - Non-shared address space
    - Send-receive messages + barrier for synch.

Shared Memory Version

```
1.   int n, nprocs;         /* matrix: (n + 2-by-n + 2) elts.*/
2.   shared float **A, diff = 0;
2a.  LockDec (diff_lock);
2b.  BarrierDec (barrier1);
3.   main()
4.   begin
5.       read(n) ;    /*read input parameter: matrix size*/
5a.       Read (nprocs);
6.       A ← g_malloc (a 2-d array of (n+2) x (n+2)  doubles);
6a.       Create (nprocs -1, Solve, A);
7.       initialize(A);       /*initialize the matrix A somehow*/
8.       Solve (A);          /*call the routine to solve equation*/
8a.      Wait_for_End (nprocs-1);
9.   end main
```

Shared Memory Version

```
10.  procedure Solve (A)    /*solve the equation system*/
11.      float **A;          /*A is an (n + 2)-by-(n + 2) array*/
12.  begin
13.      int i, j, pid, done = 0;
14.      float mydiff, temp;
14a.              mybegin  = 1 + (n/nprocs)*pid;
14b.              myend = mybegin + (n/nprocs);
15.      while (!done) do    /*outermost loop over sweeps*/
16.           mydiff = diff = 0;    /*initialize local difference to 0*/
16a.          Barrier (barrier1, nprocs);
17.           for  i ← mybeg to myend do/*sweep for all points of grid*/
18.               for j ← 1 to n do
19.                  temp = A[i,j];  /*save old value of element*/
20.                  A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);       /*compute average*/
22.                  mydiff += abs(A[i,j] - temp);
23.               end for
24.           end for
24a               lock (diff_lock);
24b.              diff += mydiff;
24c               unlock (diff_lock);
24d.          barrier (barrier1, nprocs);
25.           if (diff/(n*n) < TOL) then done = 1;
26.      end while
27.  end procedure
```

- No red-black, simply ignore dependences within sweep
- Simpler asynchronous version, may take longer to converge!

Shared Memory Version

```
procedure Solve (A)   /*solve the equation system*/
    float **A;          /*A is an (n + 2)-by-(n + 2) array*/
begin
    int i, j, pid, done = 0;
    float mydiff, temp;
             mybegin  = 1 + (n/nprocs)*pid;
             myend = mybegin + (n/nprocs);
    while (!done) do    /*outermost loop over sweeps*/
         mydiff = diff = 0;    /*initialize local difference to 0*/
        Barrier (barrier1, nprocs);
        for  i ← mybeg to myend do/*sweep
            for j ← 1 to n do
               temp = A[i,j];  /*save old value of eler
               A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-
                       A[i,j+1] + A[i+1,j]);        /*cor
               mydiff += abs(A[i,j] - temp);
            end for
         end for
              lock (diff_lock);
              diff += mydiff;
              unlock (diff_lock);
         barrier (barrier1, nprocs);
         if (diff/(n*n) < TOL) then done = 1;
    end while
end procedure
```

- No red-black, simply ignore dependences within sweep
- Simpler asynchronous version, may take longer to converge!

Why do we need this barrier?

Why do we need this barrier?

Shared Memory Version

```
10.  procedure Solve (A)    /*solve the equation system*/
11.      float **A;          /*A is an (n + 2)-by-(n + 2) array*/
12.  begin
13.      int i, j, pid, done = 0;
14.      float mydiff, temp;
14a.             mybegin  = 1 + (n/nprocs)*pid;
14b.             myend = mybegin + (n/nprocs);
15.      while (!done) do    /*outermost loop over sweeps*/
16.          mydiff = diff = 0;    /*initialize local difference to 0*/
16a.         Barrier (barrier1, nprocs);
17.          for  i ← mybeg to myend do/*sweep
18.              for j ← 1 to n do
19.                  temp = A[i,j];  /*save old value of eler
20.                  A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-
21.                          A[i,j+1] + A[i+1,j]);       /*cor
22.                  mydiff += abs(A[i,j] - temp);
23.              end for
24.          end for
24a
24b.
24c
24d.         barrier (barrier1, nprocs);
25.          if (diff/(n*n) < TOL) then done = 1;
26.      end while
27.  end procedure
```

- No red-black, simply ignore dependences within sweep
- Simpler asynchronous version, may take longer to converge!

Why do we need this barrier?

Reduce (mydif, diff);

Why do we need this barrier?

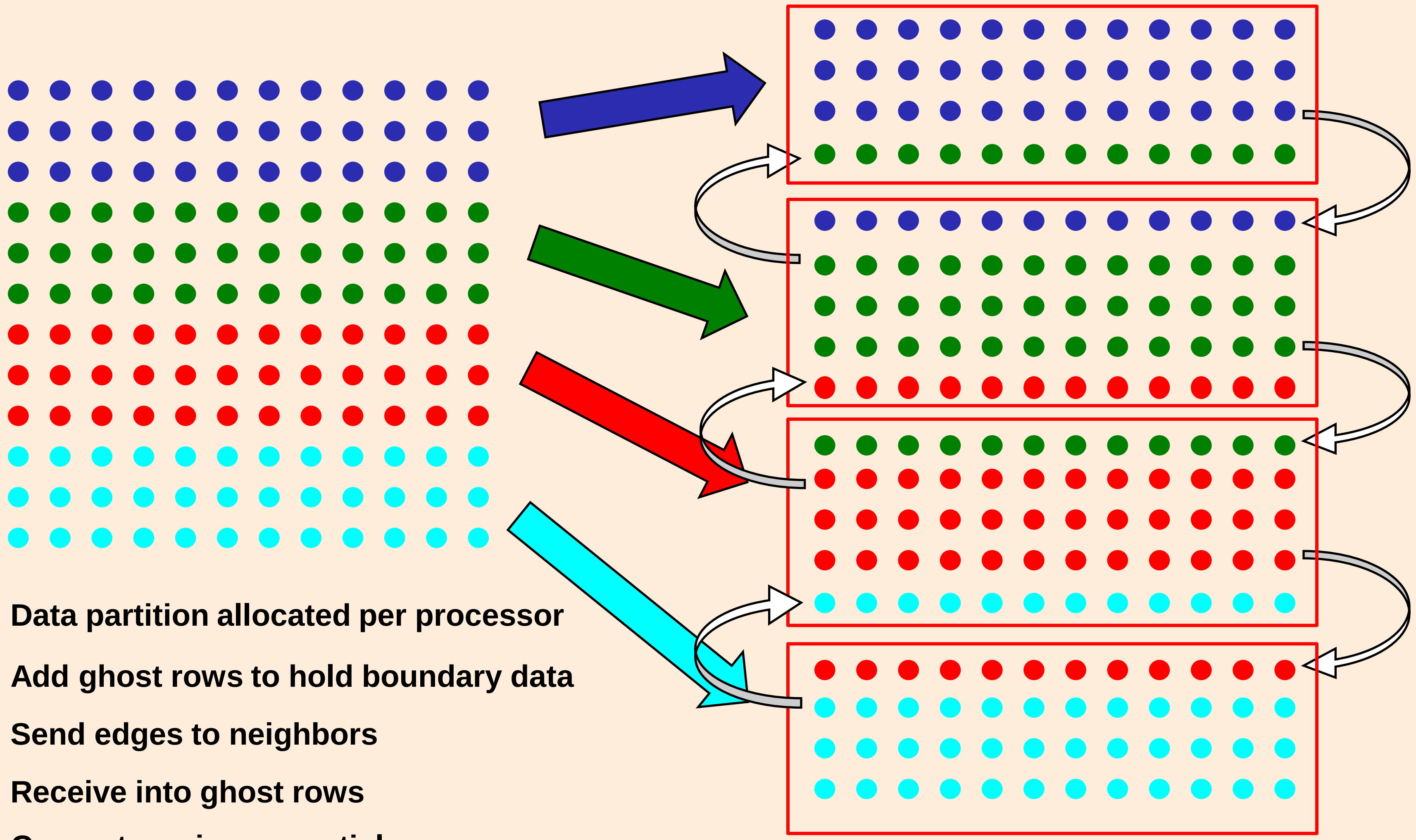
Shared Memory Program : Remarks

- **done** condition evaluated redundantly by all
- Each process has private mydiff variable
- Most interesting special operations are for synchronization provided by LOCK-UNLOCK around *criticalsection*
  - Set of operations we want to execute atomically
  - accumulations into shared diff have to be mutually exclusive
- Good global reduction?

Message Passing Version

- Cannot declare A to be global shared array
  - compose it from per-process private arrays
  - usually allocated in accordance with the assignment of work -- owner-compute rule
    - process assigned a set of rows allocates them locally
- Structurally similar to SPMD Shared Memory Version
- Orchestration different
  - data structures and data access/naming
  - communication
  - synchronization
- Ghost rows

Data Layout and Orchestration



Data partition allocated per processor

Add ghost rows to hold boundary data

Send edges to neighbors

Receive into ghost rows

Compute as in sequential program

Message Passing Version

```
1.   int n, nprocs;        /* matrix: (n + 2-by-n + 2) elts.*/
2.   float **myA;
3.   main()
4.   begin
5.       read(n) ;    /*read input parameter: matrix size*/
5a.       read (nprocs);
/* 6. A ← g_malloc (a 2-d array of (n+2) x (n+2) doubles); */
6a.       Create (nprocs -1, Solve, A);
/* 7.    initialize(A);      */ /*initialize the matrix A somehow*/
8.       Solve (A);          /*call the routine to solve equation*/
8a.       Wait_for_End (nprocs-1);
9.   end main
```

Message Passing Version

```
10.  procedure Solve (A)   /*solve the equation system*/
11.      float A[n+2][n+2];          /*A is an (n + 2)-by-(n + 2) array*/
12.  begin
13.      int i, j, pid, done = 0;
14.      float mydiff, temp;
14a.     myend = (n/nprocs) ;
14b.     myA = malloc  (array of ((n/nprocs)+2) x (n+2) floats );
14c.     If (pid == 0)
               Initialize (A)
14d.     GetMyArray (A, myA);    /* get n x (n+2) elts. from proess 0 */
15.      while (!done)  {    /*outermost loop over sweeps*/
16.          mydiff = 0;    /*initialize local difference to 0*/
16a.         if (pid != 0) then
                   SEND (&myA[1,0] , n*sizeof(float), (pid-1), row);
16b.         if (pid != nprocs-1) then
                  SEND (&myA[myend,0], n*sizeof(float), (pid+1), row);
16c.         if (pid != 0) then
                   RECEIVE (&myA[0,0], n*sizeof(float), (pid -1), row);
16d.         if (pid != nprocs-1) then
                   RECEIVE (&myA[myend+1,0], n*sizeof(float), (pid -1), row);
16e.       ...      ...      ...
```

Message Passing Version – Solver

```
12.begin
           …      …      …
15.        while (!done) do    /*outermost loop over sweeps*/
             …     …      …
17.              for  i ←  1 to myend do/*sweep for all points of grid*/
18.                 for j ← 1 to n do
19.                    temp = myA[i,j];          /*save old value of element*/
20.                    myA[i,j] ← 0.2 * (myA[i,j] + myA[i,j-1] +myA[i-1,j] +
21.                           myA[i,j+1] + myA[i+1,j]);     /*compute average*/
22.                    mydiff += abs(myA[i,j] - temp);
23.                 end for
24.              end for
24a              if (pid != 0) then
24b.                 SEND (mydif, sizeof (float), 0, DIFF);
24c.                 RECEIVE (done, sizeof(int), 0, DONE);
24d.             else
24e.                   for k ← 1 to nprocs-1 do
24f.                       RECEIVE (tempdiff, sizeof(float), k   ,  DIFF);
24g.                       mydiff += tempdiff;
24h.                     Endfor
24i.                   if (diff/(n*n) < TOL) then done = 1;
24j.                   for k ← 1 to nprocs-1 do
24k.                       SEND (done, sizeof(float), k , DONE);
26.        end while
27.end procedure
```

Message Passing Version : Remarks

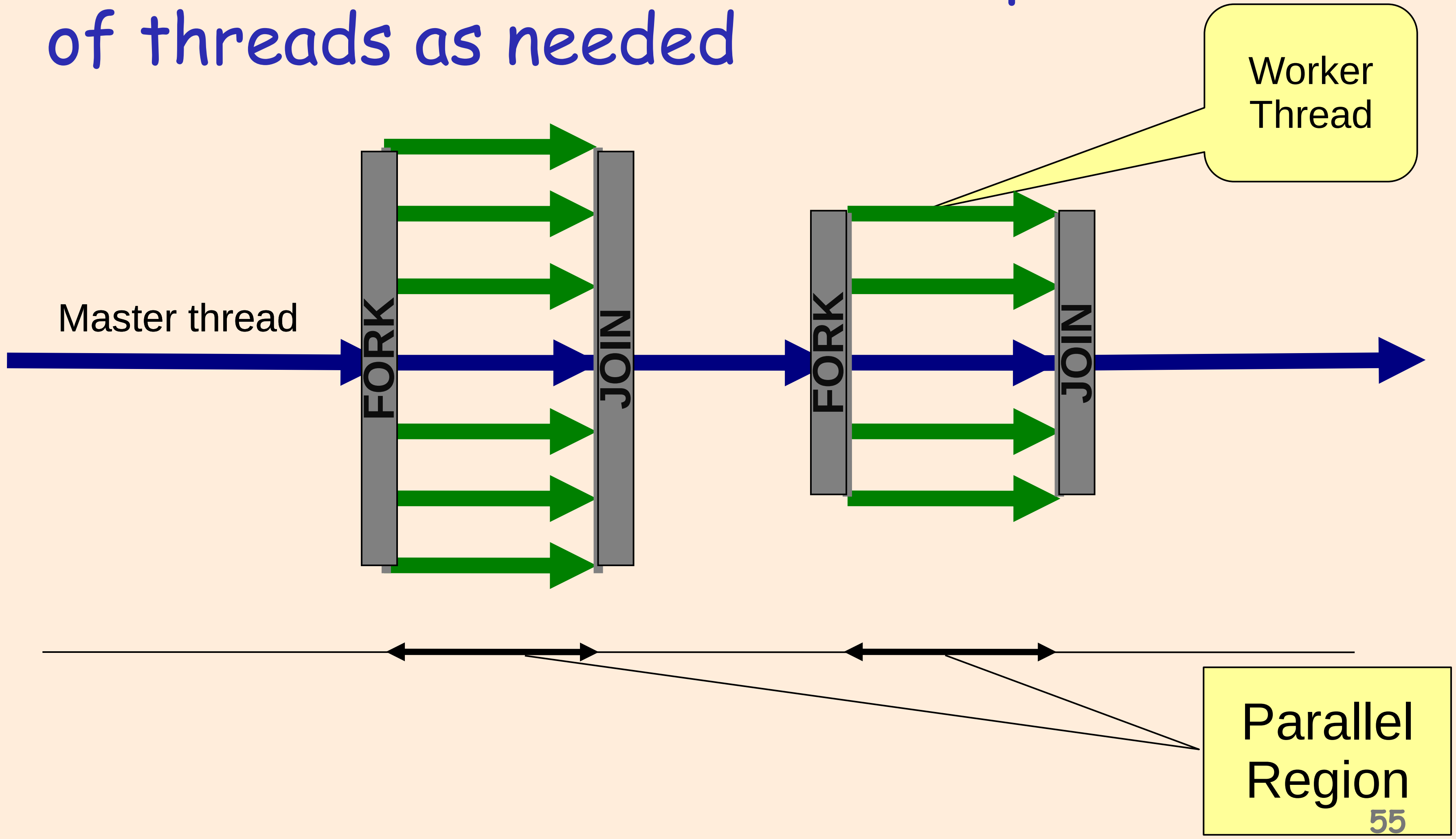
- Communication in whole rows, not element at a time
- Code similar, but indices/bounds in local rather than global space
- Synchronization through sends and receives
  - Update of global diff and event synch for done condition
  - Could implement locks and barriers with messages
- Can use REDUCE and BROADCAST library calls to simplify code
- Communication done at beginning of iteration, synchronization only between neighboring processes

What is OpenMP?

- What does OpenMP stands for?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP is an Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism.***
  - API components:
    - Compiler Directives
    - Runtime Library Routines
    - Environment Variables

OpenMP execution model

Fork and Join: Master thread spawns a team of threads as needed



OpenMP syntax

- Most of the constructs of OpenMP are pragmas
  - #pragma omp construct [clause [clause] …]
  - An OpenMP construct applies to a structural block (one entry point, one exit point)
- Categories of OpenMP constructs
  - Parallel regions
  - Work sharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- In addition:
  - Several omp_<something> function calls
  - Several OMP_<something> environment variables

Parallel Regions - Example

- "omp parallel" pragma to indicates next structured block is executed by all threads (forks)
- For example:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Each thread executes a copy of the the code within the structured block

Runtime function to request a certain number of threads

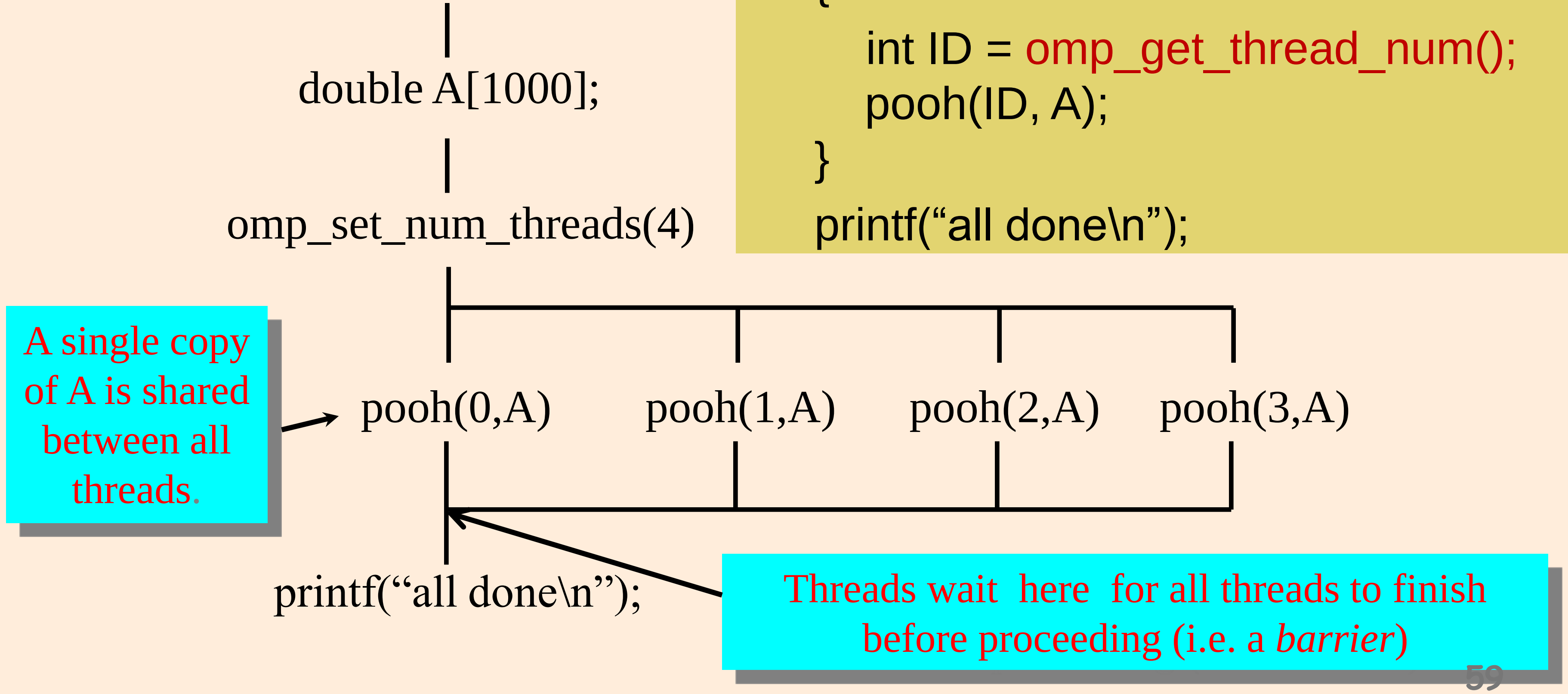
Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for ID = 0 to 3

Parallel Regions – Another Example

- Each thread executes the same code redundantly.

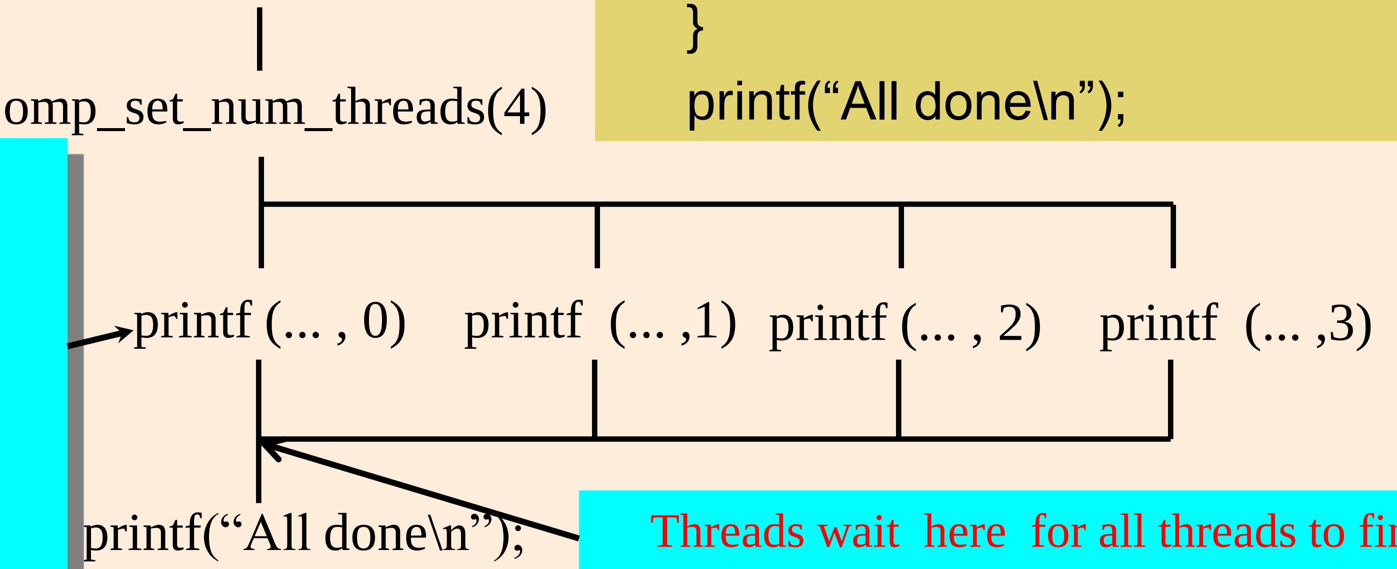
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```


double A[1000];
omp_set_num_threads(4)
pooh(0,A)
pooh(1,A)
pooh(2,A)
pooh(3,A)
printf("all done\n");
A single copy of A is shared between all threads.
Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

Parallel Regions – Yet Another Example

- Each thread executes the same code redundantly.

```
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf ("Hello World %d\n", ID);
}
printf("All done\n");
```



Prints in some order!

```
Hello Word 0
Hello World 2
Hello World 3
Hello World 1
All Done
```

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

OpenMP: Work Sharing Constructs

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int Nthr = omp_get_num_threads();
  int istart = id*N/Nthr; iend = (id+1)*N/Nthr;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

Automatic parallelization of the for loop

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

OpenMP: Work Sharing Constructs

OpenMP* shortcut: Put the "parallel" and the work-share on the same line

```
#pragma omp parallel
#pragma omp for schedule(static)
{
    for (int i=0; i<N; i++)
     { a[i] =b[i]+c[i]; }
}
```

```
#pragma omp parallel for schedule(static)
{
  for (int i=0; i<N; i++)
  { a[i] =b[i]+c[i]; }
}
```

OpenMP For construct: The Schedule Clause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,csize])
    - Deal-out blocks of iterations of size “csize” to each thread.
    - Default: chunks of approximately equal size, one to each thread
    - If more chunks than threads: assign in round-robin to the threads

Problems of schedule static for

- Load balancing
  - If all the iterations execute at the same speed, the processors are used optimally
  - If some iterations are faster than others, some processors may get idle, reducing the speedup
  - We don't always know the distribution of work, may need to re-distribute dynamically
- Granularity
  - Thread creation and synchronization takes time
  - Assigning work to threads on per-iteration resolution may take more time than the execution itself!
  - Need to coalesce the work to coarse chunks to overcome the threading overhead
- Trade-off between load balancing and granularity!

OpenMP For construct: The Schedule Clause

- Use dynamic schedule clause for load balancing
  - schedule(dynamic[,csize])
    - Each thread grabs “csize” iterations from a queue until all iterations have been handled.
    - Threads receive chunk assignments dynamically
    - Default csize = 1

OpenMP Section : Work Sharing Construct

- The Sections work-sharing construct gives a different structured block to each thread.

```
#pragma omp parallel
#pragma omp sections
{
   #pragma omp section
         X_calculation();
   #pragma omp section
         y_calculation();
   #pragma omp section
         z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

PI Program: The sequential program

```
static long num_steps = 100000;
double step;
void main ()
{           int i;    double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=1;i<= num_steps; i++){
                    x = (i-0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

PI Program: OpenMP Version

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 4
void main ()
{     int i;  double x, pi, sum[NUM_THREADS] ={0};
      step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel
     {          double x;     int id, i;
                id = omp_get_thread_num();
                 #pragma omp for
                      for (i=id;i< num_steps; i++ )
                      {       x = (i+0.5)*step;
                              sum[id] += 4.0/(1.0+x*x);
                       }
     }
     for(i=0, pi=0.0;i<NUM_THREADS;i+
            pi += sum[i] * step;
}
```

Any synchronization required?

Any synchronization required?

OpenMP: Data Environment

- Shared Memory programming model
  - Most variables are shared by default
- Global variables are shared
  - File scope variables, static variables
- Some variables can be private
  - Automatic variables inside the statement block
  - Automatic variables in the called functions
  - Variables can be explicitly declared as private: A local copy is created for each thread

Overriding Storage attributes

- private:
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies
- firstprivate:
  - Same, but the initial value of the variable is copied from the main thread
- lastprivate:
  - Same, but last sequential value of the variable is copied to the main thread

```
int idx=1;
int x = 10;
#pragma omp parallel for
        private (i,idx, x)
for (i=0; i<n; i++) {
    if (data[i] == x)
        idx = i; x++;
}
printf (“%d\n, idx);
```

x is not initialized

Value of idx is not from the for loop

```
int idx=1;
int x = 10;
#pragma omp parallel for
     firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i]==x)
        idx = i; x++;
}
printf (“%d\n, idx);
```

OpenMP Synchronization

```
X = 0;
#pragma omp parallel
    X = X+1;
```

What should be the result (assume 2 threads)?

Could be 1 or 2!

- OpenMP assumes that the programmer knows what (s)he is doing
  - Regions of code that are marked to run in parallel are independent
  - Race conditions are possible, it is the programmer's responsibility to insert protection

Synchronization Mechanisms

- Many of the existing mechanisms for shared programming
  - Critical sections, Atomic updates
  - Barriers
  - Nowait (turn synchronization off!)
  - Single/Master execution
  - Ordered
  - Flush (memory subsystem synchronization)
  - Reduction

Critical Sections

- #pragma omp critical [name]
  - Standard critical section functionality
- Critical sections are global in the program
  - Can be used to protect a single resource in different functions
- #pragma omp atomic
  update_statement

Reduction Motivation

- How to parallelize this code?

```
for (i=0; i<N; j++) {
    sum = sum+a[i]*b[i];
}
```

  - accessing it atomically is too expensive
  - Have a private copy in each thread, then add them up
- OpenMP clause Reduction: data environment clause that affects the way variables are shared:

  reduction (op : list)

  - The variables in "list" must be shared in the enclosing parallel region
- Use the reduction clause!
  #pragma omp parallel for reduction(+: sum)

OpenMP: Reduction Example

```
#include <omp.h>
#define NUM_THREADS 4
void main ()
{
    int i;
    int A[1000], B[1000];  sum=0;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum) private(tmp)
    for (i=0; i< 1000; i++){
        tmp = A[i] * B[i] ;
        sum = sum + tmp;
    }
}
```

Barrier synchronization

- #pragma omp barrier
- Performs a barrier synchronization between all the threads in a team *at the given point.*
- Example:

```
#pragma omp parallel
{
  int result = heavy_computation_part1();
  #pragma omp atomic
     sum += result;
  #pragma omp barrier
  heavy_computation_part2(sum);
}
```

OpenMP: Implicit Synchronization

- Barriers are implied on the following OpenMP constructs:
  - end parallel
  - end sections
  - end single
- Use NoWait to avoid synchronization

Controlling OpenMP behavior

- omp_set_num_threads(int)
  - Control the number of threads used for parallelization
  - Must be called from sequential code
  - Also can be set by OMP_NUM_THREADS environment variable
- omp_get_num_threads()
  - How many threads are currently available?
- omp_get_thread_num()
- omp_set_nested(int)/omp_get_nested()
  - Enable nested parallelism
- omp_in_parallel()
  - Am I currently running in parallel mode?
- omp_get_wtime()
  - A portable way to compute wall clock time

Message Passing Interface (MPI)

Standard API

- Hides sw/hw details, portable, flexible

Implemented as a library

Your program	
MPI Library	
Custom SW	**Standard TCP/IP**
Custom HW	**Standard network HW**

Making MPI Programs

- Executable must be built by compiling program and linking with MPI library
  - Header files (mpi.h) provide definitions and declarations
- MPI commonly used in SPMD mode
  - One executable
  - Multiple instances of it executed concurrently
- Implementations provide command to initiate execution of MPI processes (mpirun)
  - Options: number of processes, which processors they are to run on

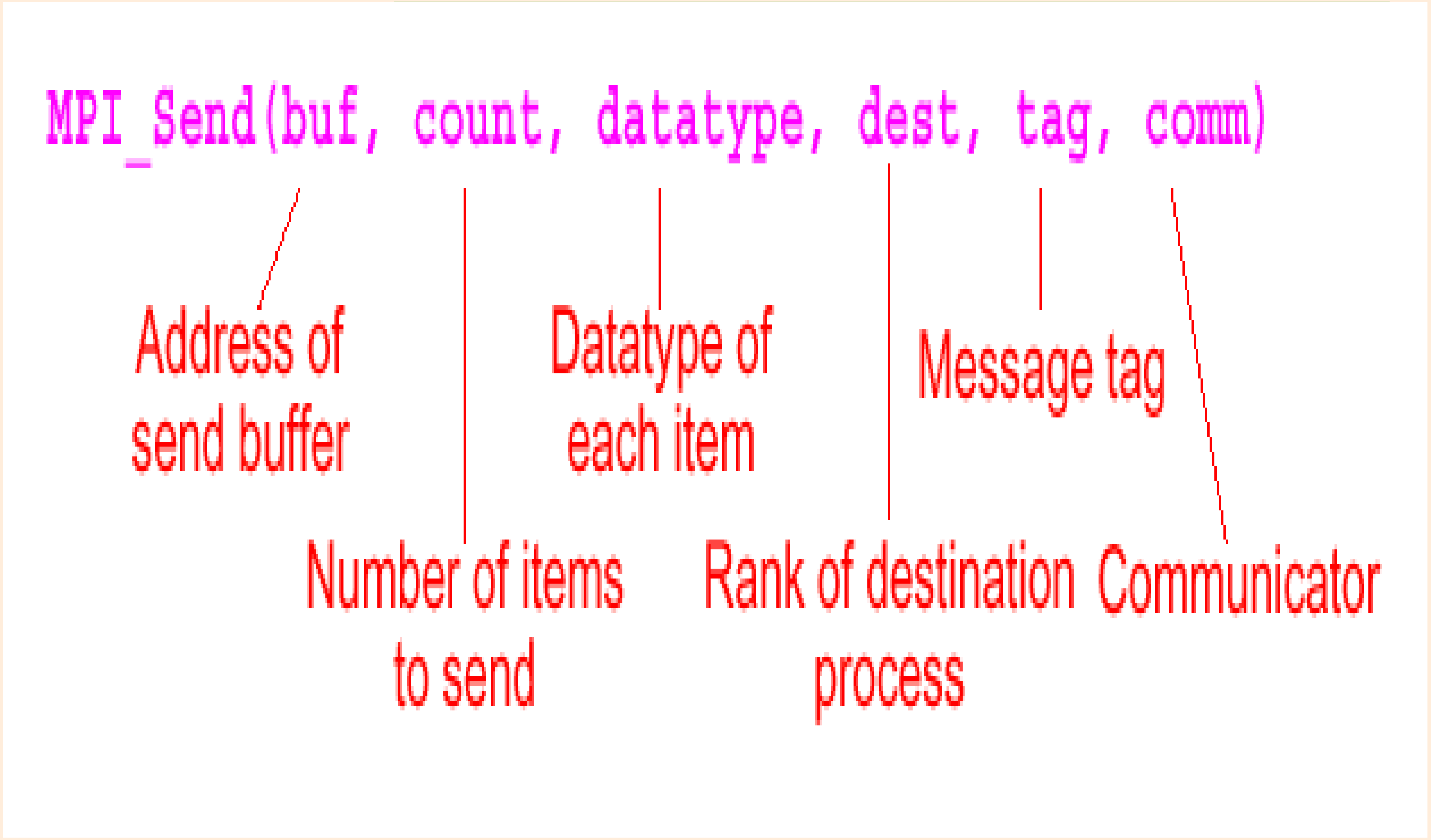
Key MPI Functions and Constants

- MPI_Init (int *argc, char ***argv)
- MPI_Finalize (void)
- MPI_Comm_rank (MPI_COMM comm, int *rank)
- MPI_Comm_size (MPI_COMM comm, int *size)
- MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- MPI_CHAR, MPI_INT, MPI_LONG, MPI_BYTE
- MPI_ANY_SOURCE, MPI_ANY_TAG

MPI: Matching Sends and Recvs

- Sender always specifies destination and tag, Addr., size, type of the data
- Receiver specifies source, tag, location, size and type of data
- Receive can specify for exact match or using wild cards (any source, any tag)
- Send/Receive : Standard, Buffered, Synchronous and Ready modes
- Send/Receive : Blocking or Non-Blocking

Parameters of Blocking Send



```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

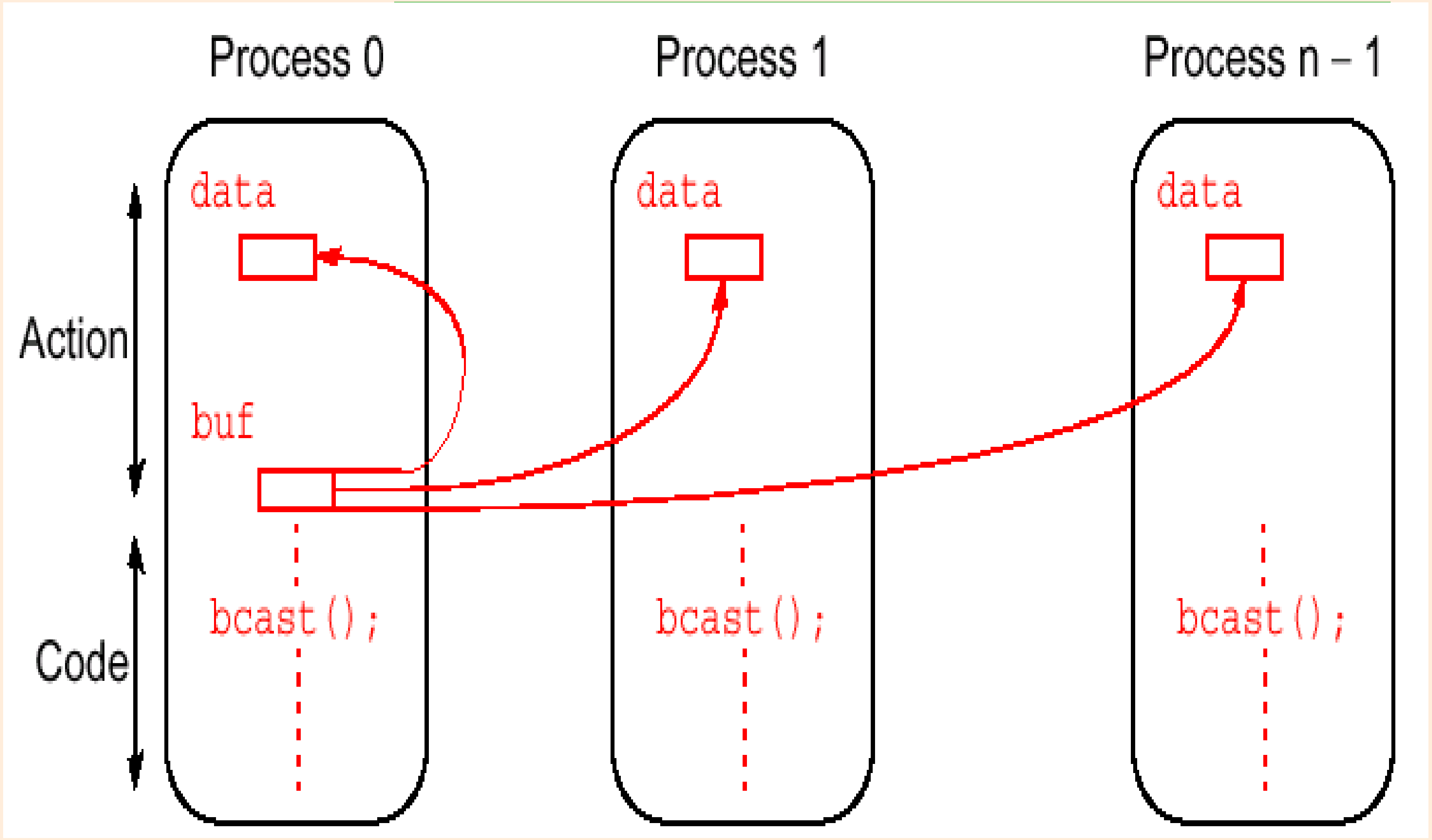
MPI Blocking and Non-blocking

- Blocking - return after local actions complete, though the message transfer may not have been completed
- Non-blocking - return immediately
  - Assumes that data storage to be used for transfer is not modified by subsequent statements prior to being used for transfer
  - Implementation dependent local buffer space is used for keeping message temporarily

MPI Group Communication

- Until now: point-to-point messages
- MPI also provides routines that sends messages to a group of processes or receives messages from a group of processes
  - Not absolutely necessary for programming
  - More efficient than separate point-to-point routines
- Examples: broadcast, multicast, gather, scatter, reduce, barrier
  - MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Alltoall, MPI_Scatter, MPI_Gather, MPI_Barrier

Broadcast


Process 0
Process 1
Process n – 1
Action
Code
data
data
data
buf
bcast();
bcast();
bcast();

MPI Broadcast

```
MPI_Bcast(void *buf, int count,
  MPI_Datatype datatype, int root,
  MPI_Comm Comm)
```

Example: MPI Pi Calculating Program

```
MPI_Init (&argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank( MPI_COMM_WORLD, &myid);
MPI_Bcast(&nsteps,1,MPI_INT,0,  MPI_COMM_WORLD);
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid+1; i <= n; i += numprocs)  {
          x = h * ((double) i - 0.5);
          sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
If (myrank !=0)
    MPI_Send (&mypi, &pi, 1, MPI_DOUBLE, MPI_tag, MPI_COMM_WORLD);
else
   for (j = 1 ; j < num_procs; j++ ) {
    MPI_Recv (&temp, &pi, 1, MPI_DOUBLE, MPI_tag, MPI_COMM_WORLD);
    mypi += temp
}
MPI_Finalize();
```

Example: MPI Pi Calculating Program

```
MPI_Init (&argc, &argv);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank( MPI_COMM_WORLD, &myid);
MPI_Bcast(&nsteps,1,MPI_INT,0,  MPI_COMM_WORLD);
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid+1; i <= n; i += numprocs)  {
            x = h * ((double) i - 0.5);
            sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
```

```
}
MPI_Finalize();
```

Beware of Deadlock

- Suppose a process *P*i needs to be synchronized and to exchange data with process P_{i-1} and process P_{i+1} before continuing

  Pi:

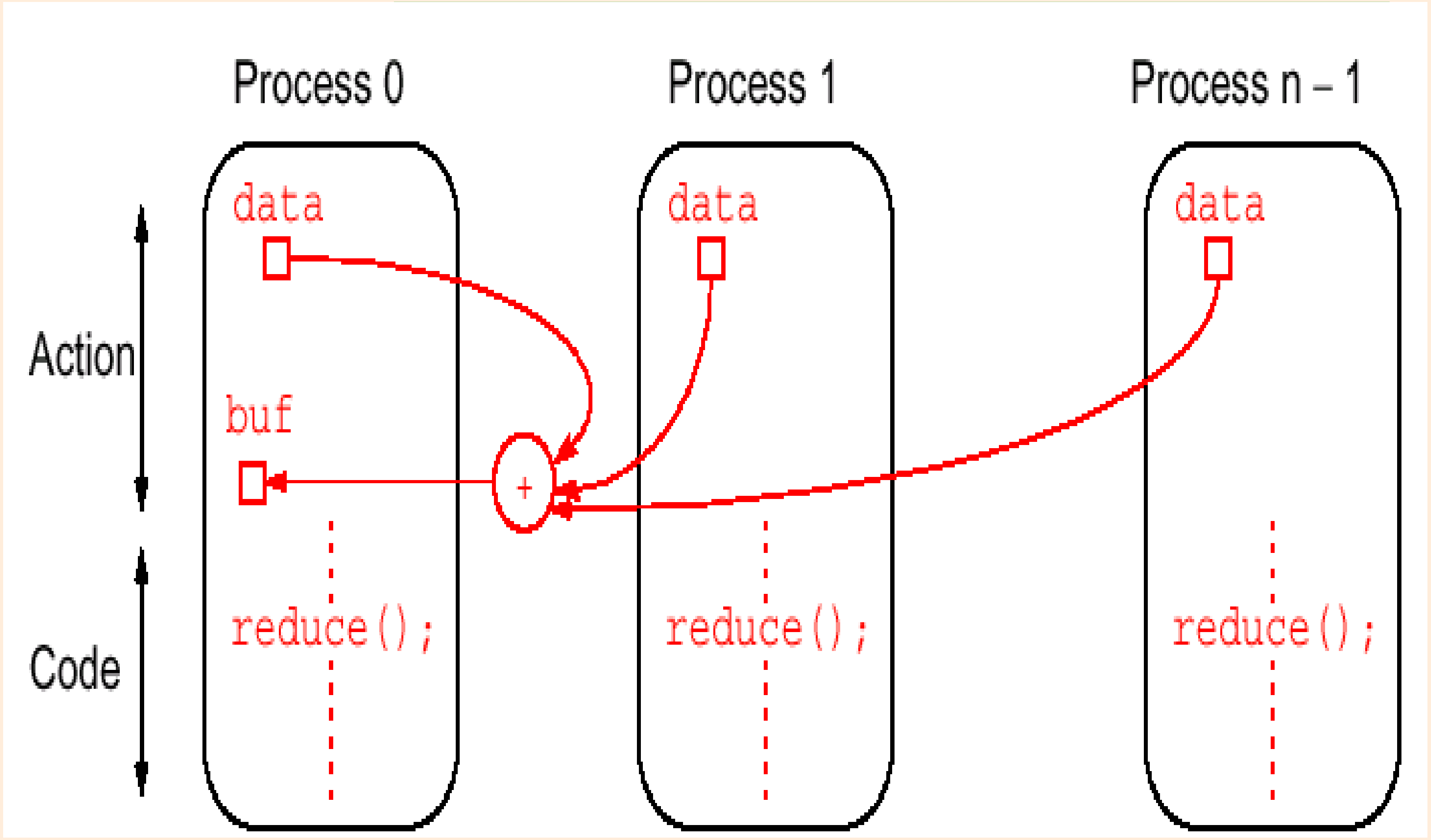
```
    send(P_{i-1});
    send(P_{i+1});
    recv(P_{i-1} );
    recv(P_{i+1});
```

MPI Reduce

MPI_Reduce (void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

- Operations: MPI_SUM, MPI_MAX
- Reduction includes value coming from root

Reduce


Process 0
Process 1
Process n – 1
Action
Code
data
data
data
buf
+
reduce();
reduce();
reduce();