Parallelization Principles

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

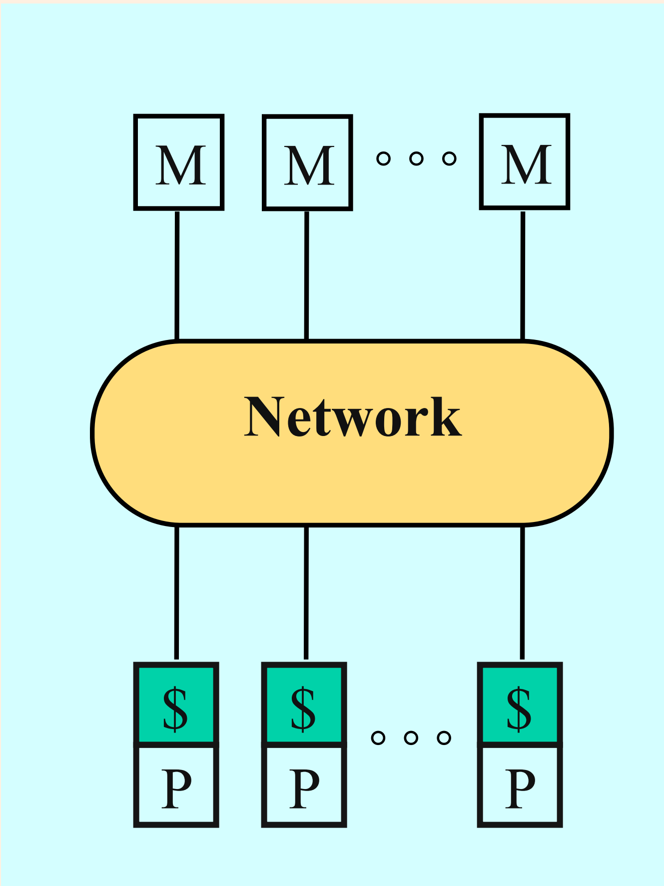
Overview

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

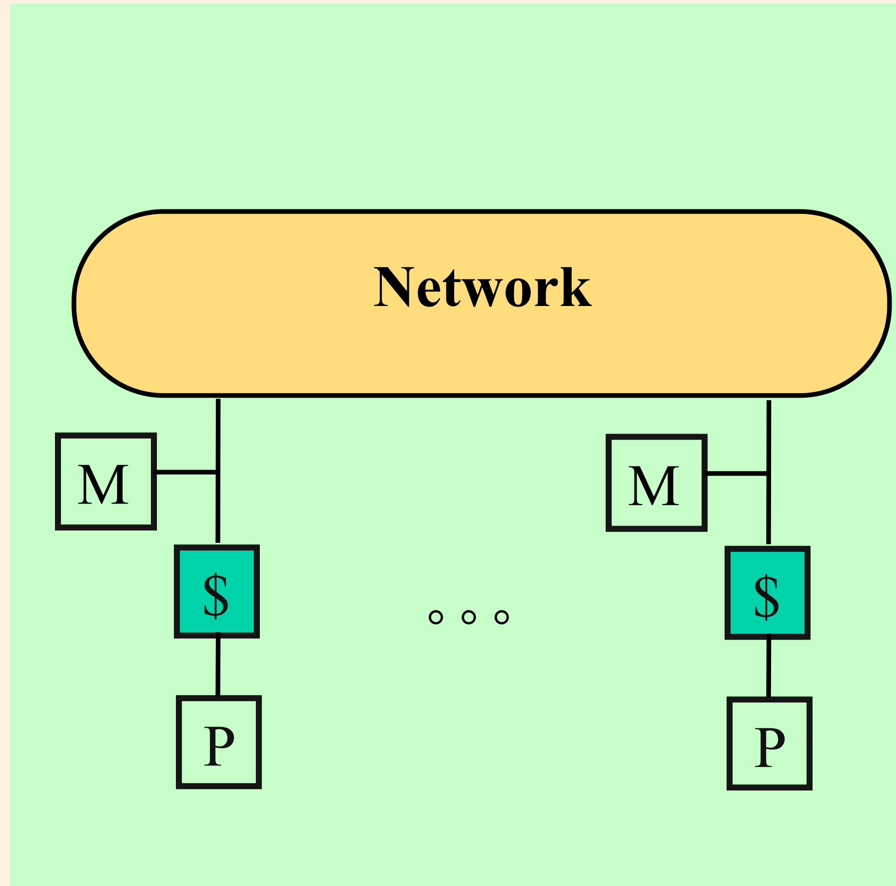
Acknowledgments:

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.

Parallel Architecture: Shared Memory

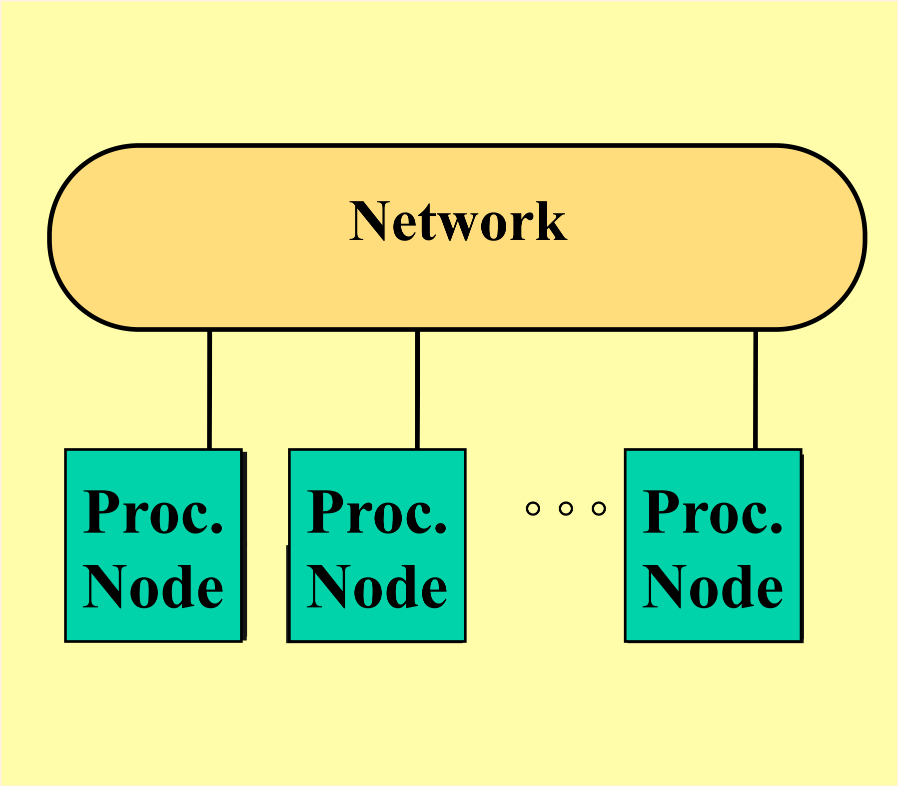


Centralized Shared Memory

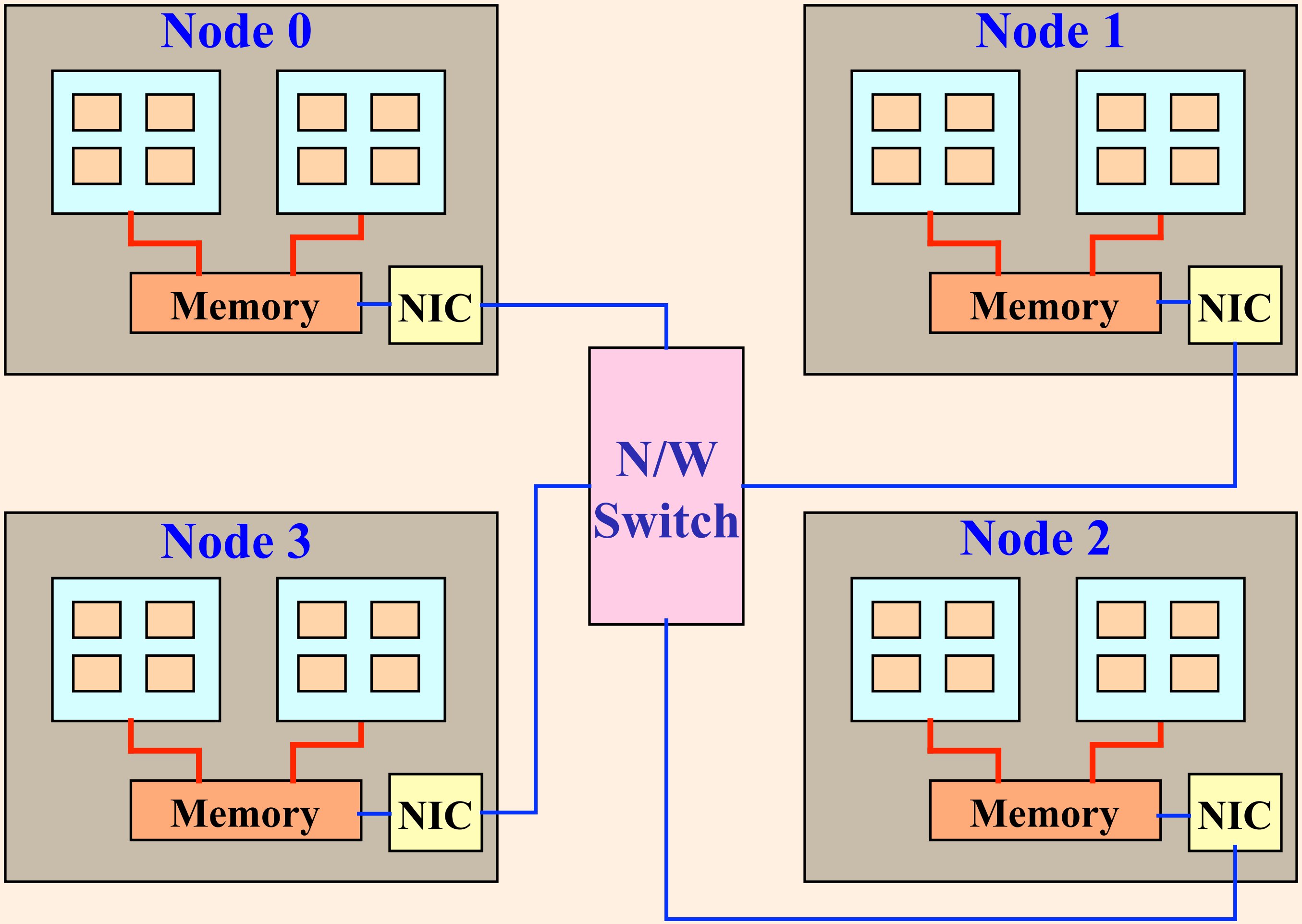


Distributed Shared Memory

Distributed Memory Architecture



Hybrid Architecture



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 = *Exec. Time in UniProcesor / Exec. Time in **n** processors*

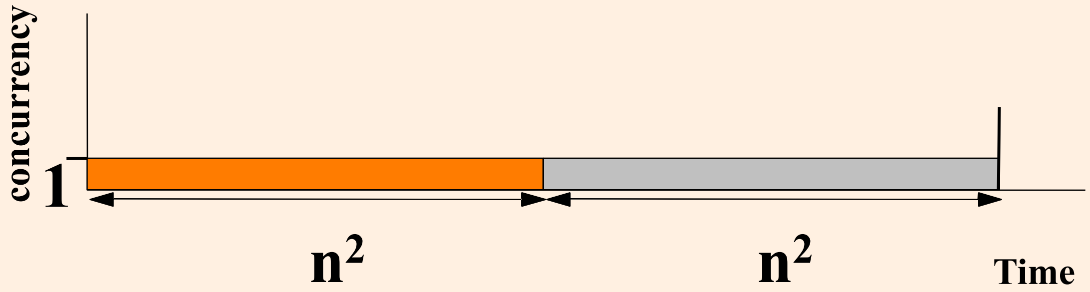
- Efficiency = *Speedup / **n***

- 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

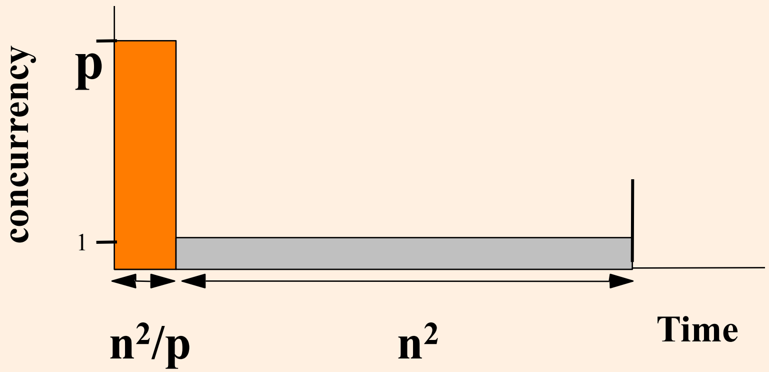
- Serial Execution Time = $n^2 + n^2 = 2n^2$

Understanding Amdahl's Law

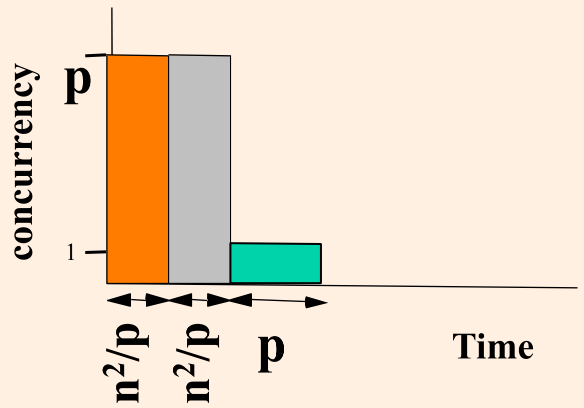
Parallel Execution time:

- 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

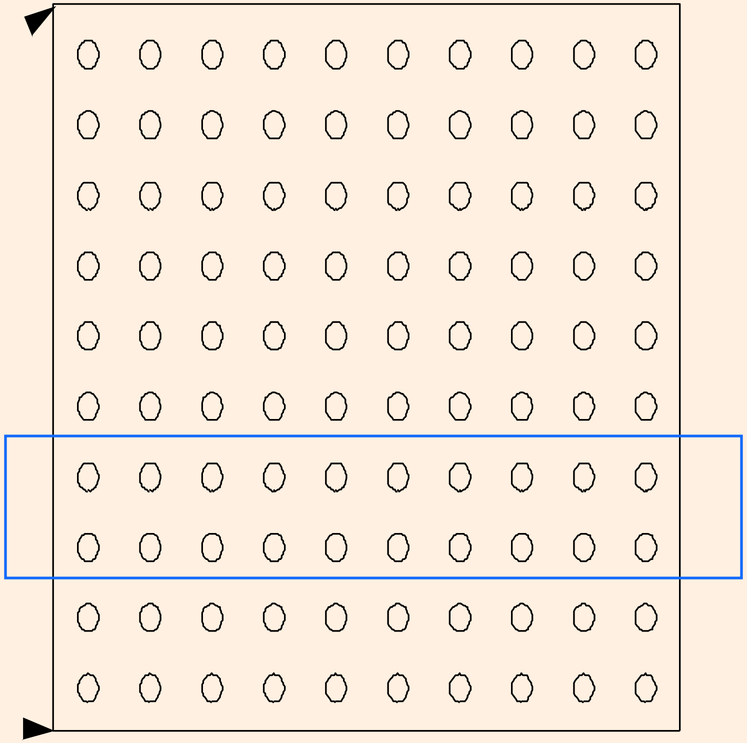
Parallelizing Computation vs. Data

- 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

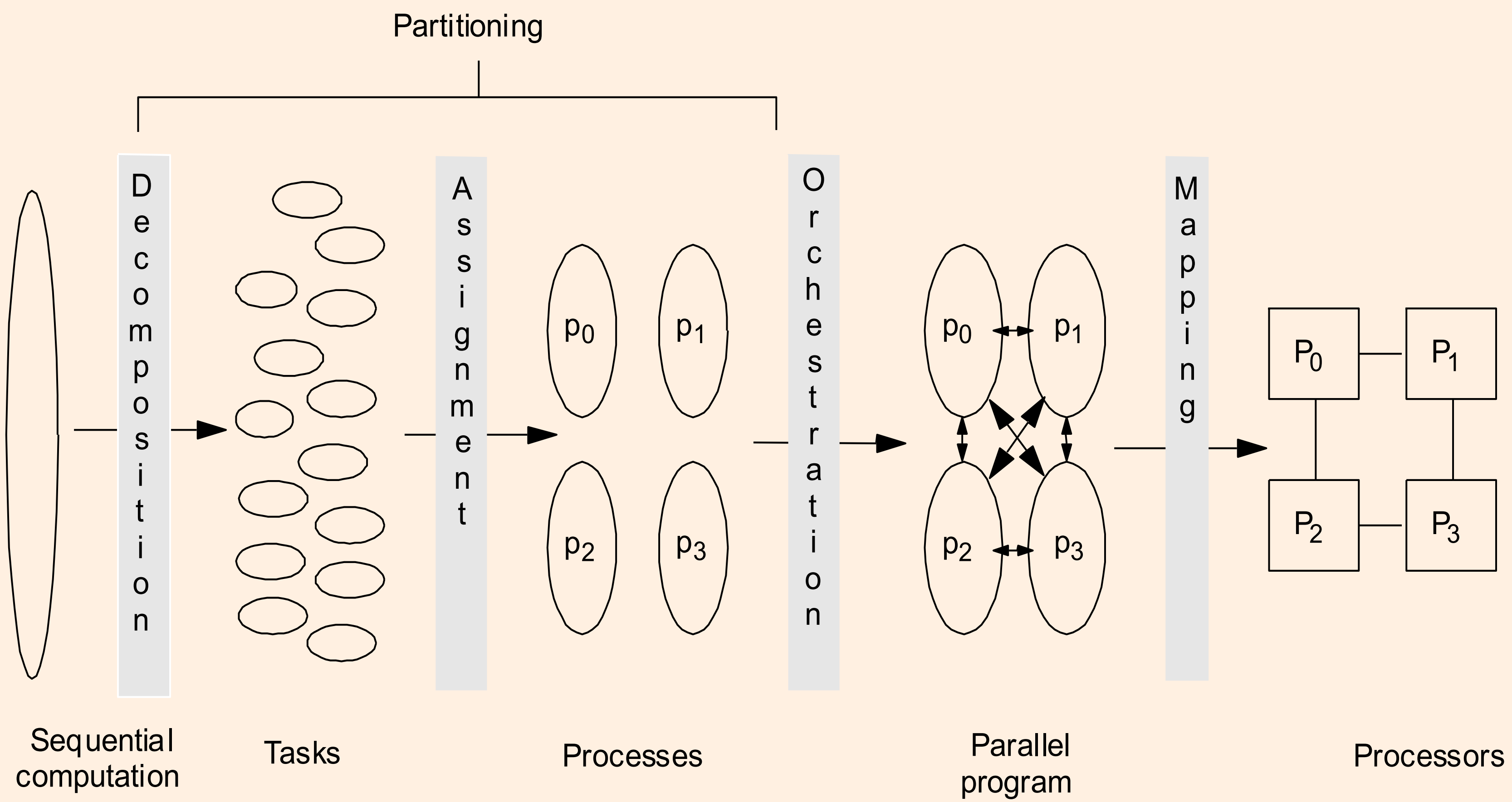
```
for i=1 to m
  for  j= 1 to n
     a[i,j] = a[i,j] + v[i]
```



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 all processors 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

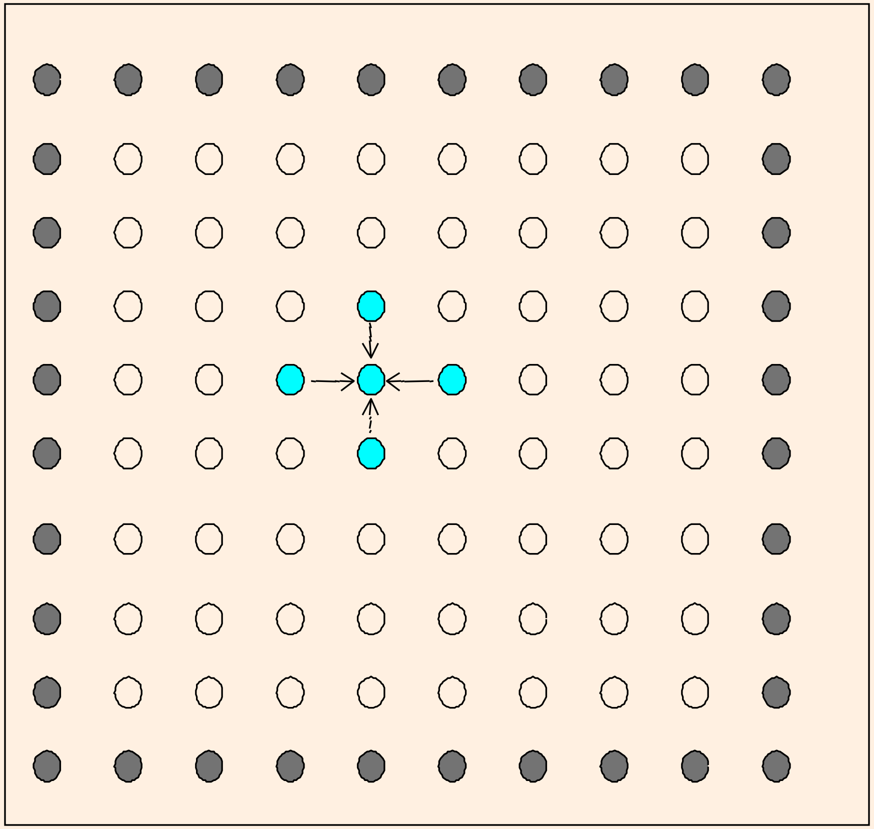
Table 2.1 Steps in the Parallelization Process and Their Goals

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 x n points of (n+2) x (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] ;
```

Sequential Version

```
int n;        /*size of matrix: (n + 2-by-n + 2) elements*/
float **A, diff = 0;
main()
begin
    read(n) ;   /*read input parameter: matrix size*/
    A ← malloc (a 2-d array of (n+2) x (n+2)  doubles);
    B ← malloc (a 2-d array of (n+2) x (n+2)  doubles);
    initialize(A);      /*initialize the matrix A somehow*/
    Solve (A);          /*call the routine to solve equation*/
end main
```

Sequential Version (contd.)

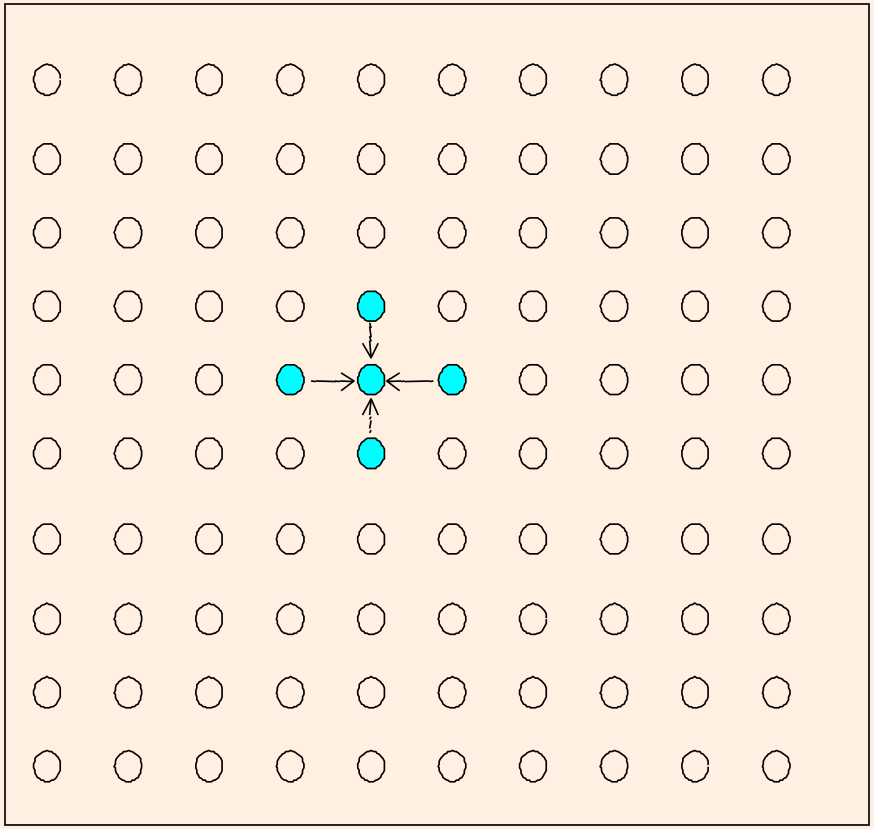
```
procedure Solve (A)   /*solve the equation system*/
      float **A;           /*A is an (n + 2)-by-(n + 2) array*/
begin
      int i, j, done = 0;
      float diff = 0, temp;
      while (!done) do    /*outermost loop over sweeps*/
            diff = 0;      /*initialize maximum difference to 0*/
            for i ← 1 to n do/*sweep over non-border points of grid*/
                  for j ← 1 to n do
                        B[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(B[i,j] – A[i,j]);
                  end for
              end for
              if (diff/(n*n) < TOL) then done = 1;
              else    Copy_Array (A ← B)
      end while
end procedure
```

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] ;
```

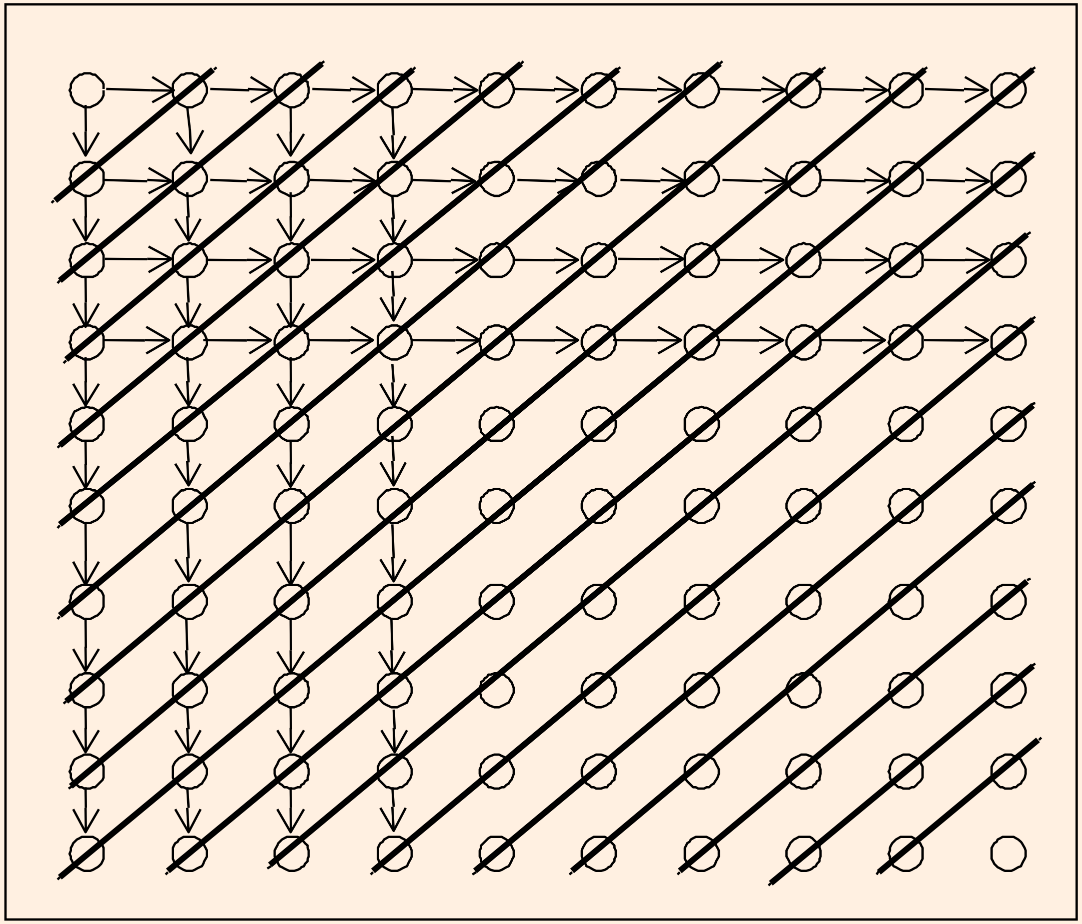
- Decomposition
  - Both i and j loops can be parallelized – no data dependences
  - Each grid point can be a task
  - To compute diff, some coordination would be required!
- Assignment
  - Each grid point
  - Each row or column
  - A group of rows or columns

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]);
  }
```

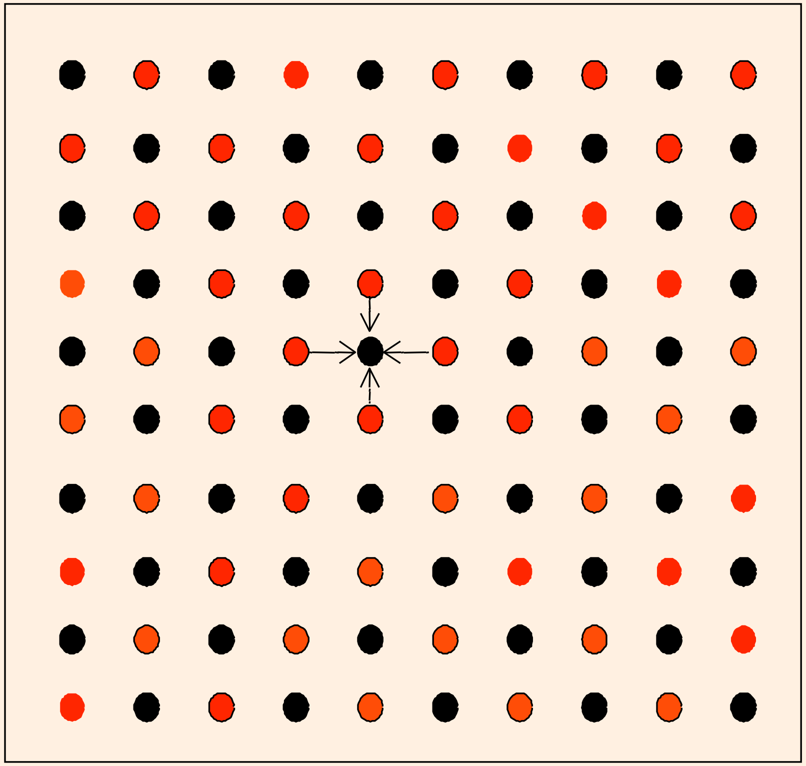
Decomposition & Assignment



- 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 p
        if (diff/(n*n) < TOL) then done = 1;
    end while
end procedure
```

Ensure computation for all black points are complete!

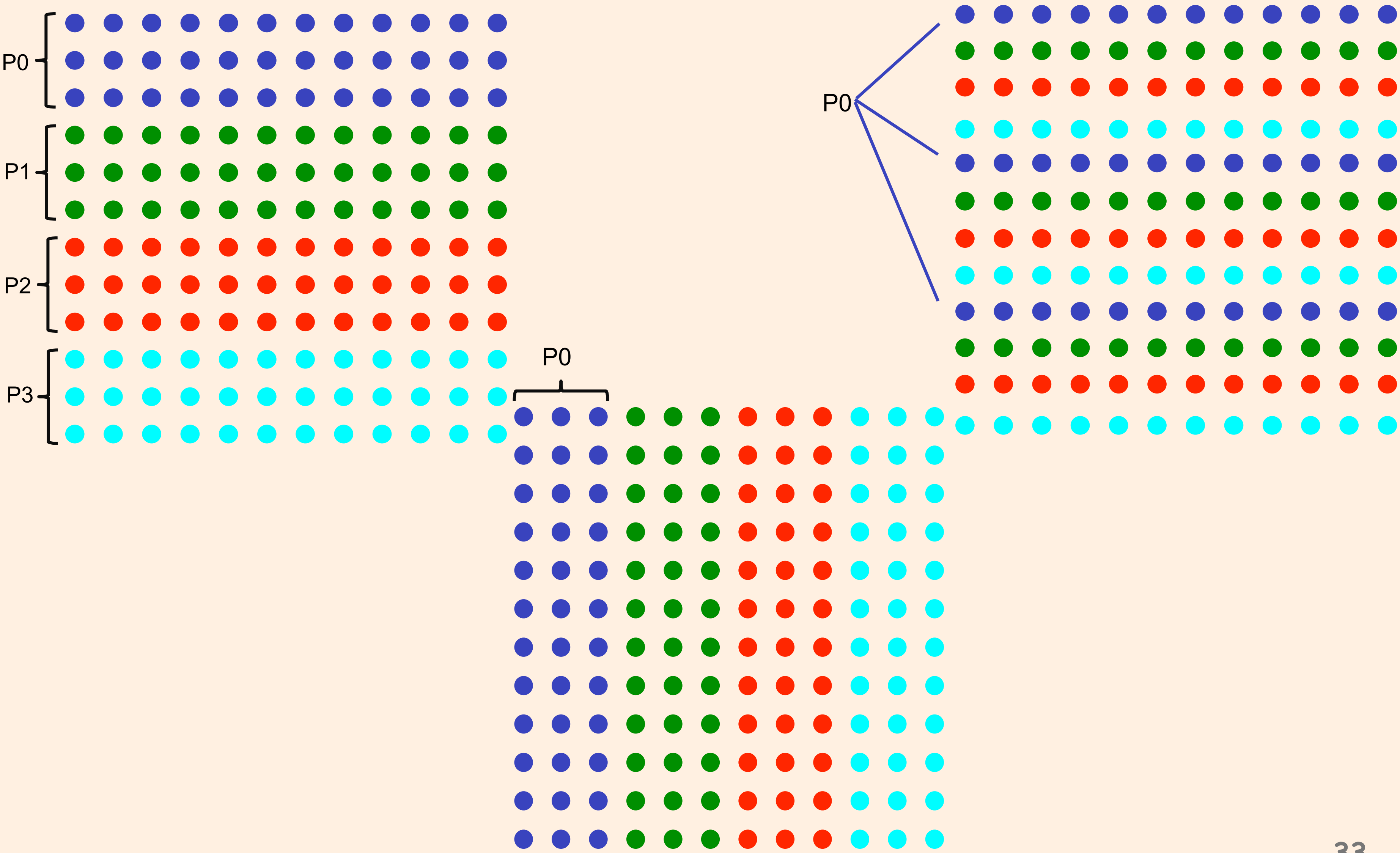
Red-Black Parallel Version (contd.)

- 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 ⇒ 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), … , (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.  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

```
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
            Reduce (mydif, diff);
        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?

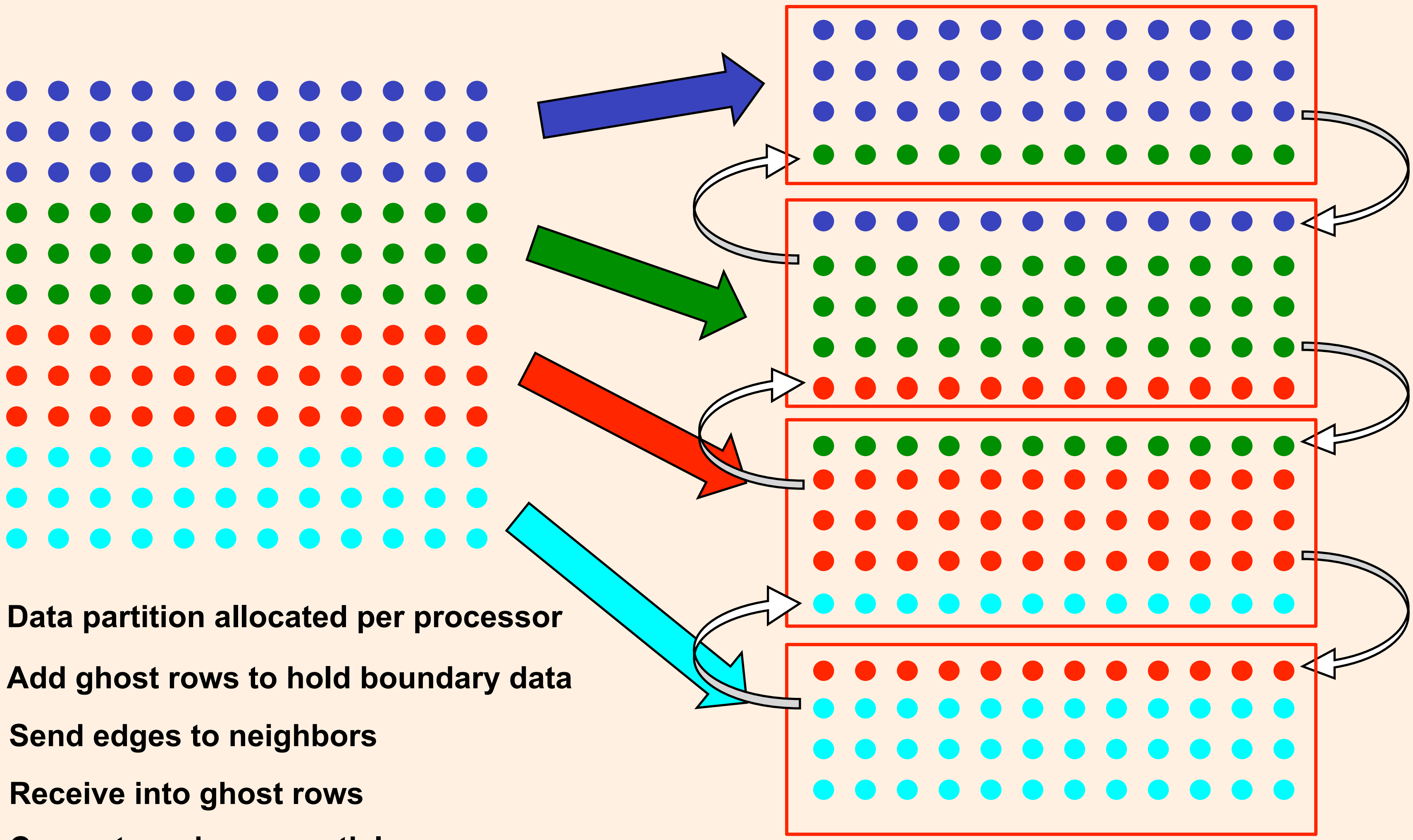
Shared Memory Program : Remarks

- **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

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

Message Passing Version – Solver

```
begin
        …     …     …
        while (!done) do    /*outermost loop over sweeps*/
          …     …     …
             for  i ←  1 to myend do/*sweep for all points of grid*/
                for j ← 1 to n do
                   temp = myA[i,j];          /*save old value of element*/
                   myA[i,j] ← 0.2 * (myA[i,j] + myA[i,j-1] +myA[i-1,j] +
                         myA[i,j+1] + myA[i+1,j]);     /*compute average*/
                   mydiff += abs(myA[i,j] - temp);
                end for
             end for
             if (pid != 0) then
                  SEND (mydif, sizeof (float), 0, DIFF);
                  RECEIVE (done, sizeof(int), 0, DONE);
             else
                   for k ← 1 to nprocs-1 do
                       RECEIVE (tempdiff, sizeof(float), k   ,  DIFF);
                       mydiff += tempdiff;
                    Endfor
                   if (diff/(n*n) < TOL) then done = 1;
                   for k ← 1 to nprocs-1 do
                       SEND (done, sizeof(float), k , DONE);
        end while
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

Orchestration: Summary

- Shared address space
  - Shared and private data explicitly separate
  - Communication implicit in access patterns
  - Synchronization via atomic operations on shared data
  - Synchronization explicit and distinct from data communication
- Message passing
  - Data distribution among local address spaces needed
  - No explicit shared structures (implicit in comm. patterns)
  - Communication is explicit
  - Synchronization implicit in communication (at least in synch. case)