Parallel Programming Principle and Practice

Lecture 4 — Parallel Programming Methodology

Outline

- □ **Motivating Problems**
- □ **Steps in Creating a Parallel Program**
- □ **What a Simple Parallel Program Looks Like**

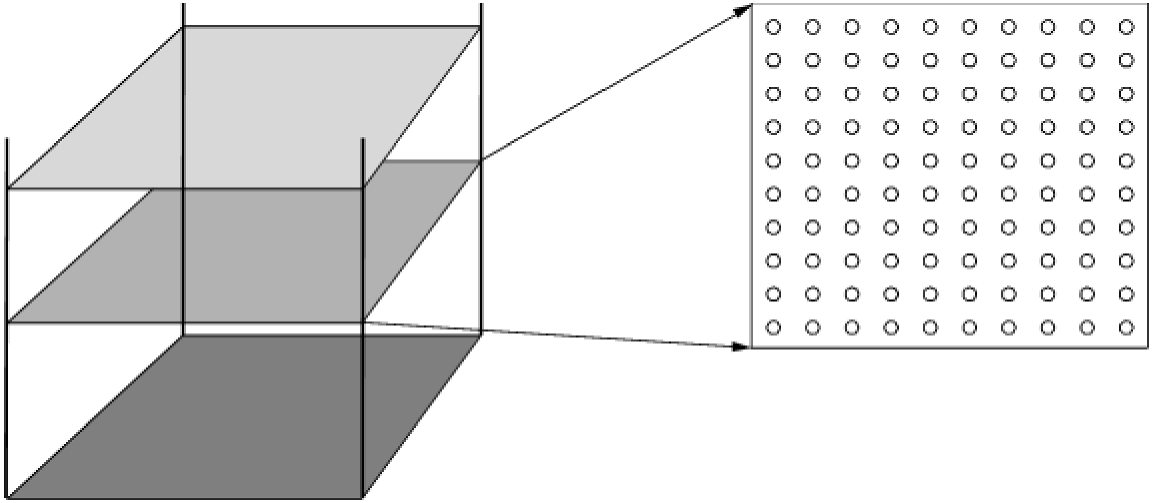
Parallel programming methodology

MOTIVATING PROBLEMS

Motivating Problems

- Simulating Ocean Currents
  - Regular structure, scientific computing
- Simulating the Evolution of Galaxies
  - Irregular structure, scientific computing
- Rendering Scenes by Ray Tracing
  - Irregular structure, computer graphics

Simulating Ocean Currents



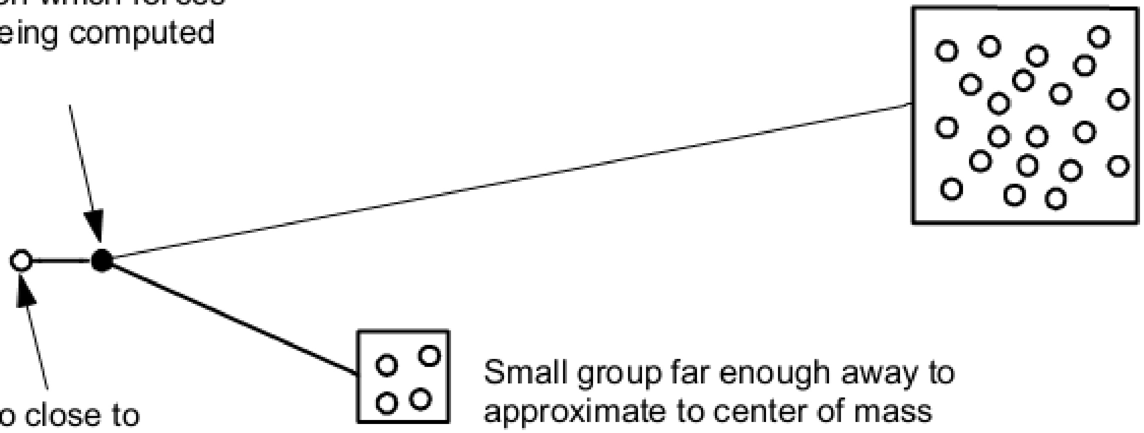
(a) Cross sections

(b) Spatial discretization of a cross section

- Model as two-dimensional grids
- Discretize in space and time
  - finer spatial and temporal resolution -> greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations

Simulating Galaxy Evolution

- □ Simulate the interactions of many stars evolving over time
- □ Computing forces is expensive
- □ $O(n^2)$ brute force approach
- □ Hierarchical methods take advantage of force law: $G\frac{m_1 m_2}{r^2}$



Star on which forces are being computed

Large group far enough away to approximate

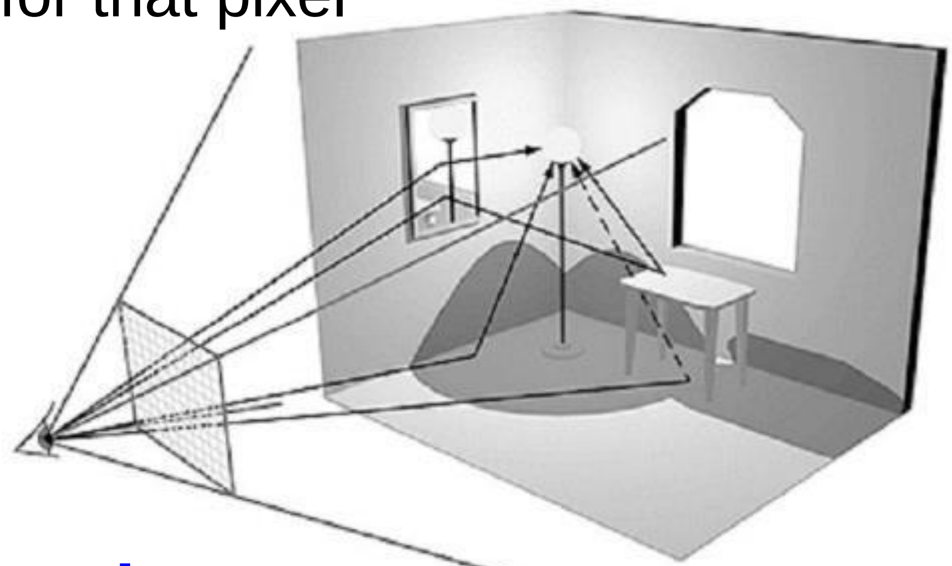
Star too close to approximate

Small group far enough away to approximate to center of mass

- □ Many time-steps, plenty of concurrency across stars within one

Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane
- Follow their paths
  - They bounce around as they strike objects
  - They generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays



All case studies have abundant concurrency

Copyright Mladen Stamenic

Creating a Parallel Program

- Assumption: Sequential algorithm is given
  - Sometimes need very different algorithm, but beyond scope
- Pieces of the job
  - Identify work that can be done in parallel
  - Partition work and perhaps data among processes
  - Manage data access, communication, and synchronization
  - *Note:* work includes computation, data access, and I/O
- Main goal: Speedup (plus low prog. effort and resource needs)

$$Speedup\ (p) = \frac{Performance(p)}{Performance(1)}$$

- For a fixed problem

$$Speedup\ (p) = \frac{Time(1)}{Time(p)}$$

Parallel programming methodology

STEPS IN CREATING PARALLEL PROGRAM

Some Important Concepts

- □ Task
  - ➢ Arbitrary piece of undecomposed work in parallel computation
  - ➢ Executed sequentially; concurrency is only across tasks
  - ➢ e.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - ➢ Fine-grained versus coarse-grained tasks
- □ Process (thread)
  - ➢ Abstract entity that performs the tasks assigned to processes
  - ➢ Processes communicate and synchronize to perform their tasks
- □ Processor
  - ➢ Physical engine on which process executes
  - ➢ Processes virtualize machine to programmer
    - first write program in terms of processes, then map to processors

Limited Concurrency: Amdahl's Law

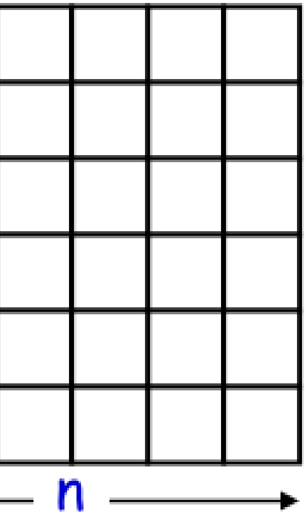
□ Fundamental limitation on parallel speedup

□ If *s* = fraction of sequential execution that is inherently serial

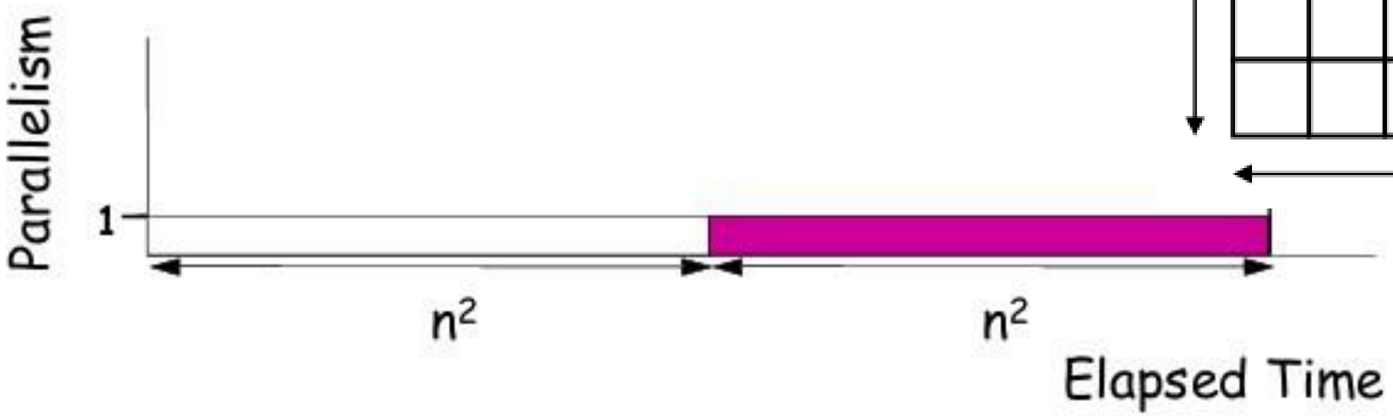
then speedup ≤ 1/*s*

Amdahl’s Law Example

- 2-phase computation over an *n*-by-*n* grid
  - Phase 1: perform an independent computation on each grid element
    - easy to parallelize
  - Phase 2: add a value from each grid element into a global sum
    - more difficult to parallelize; serial by default

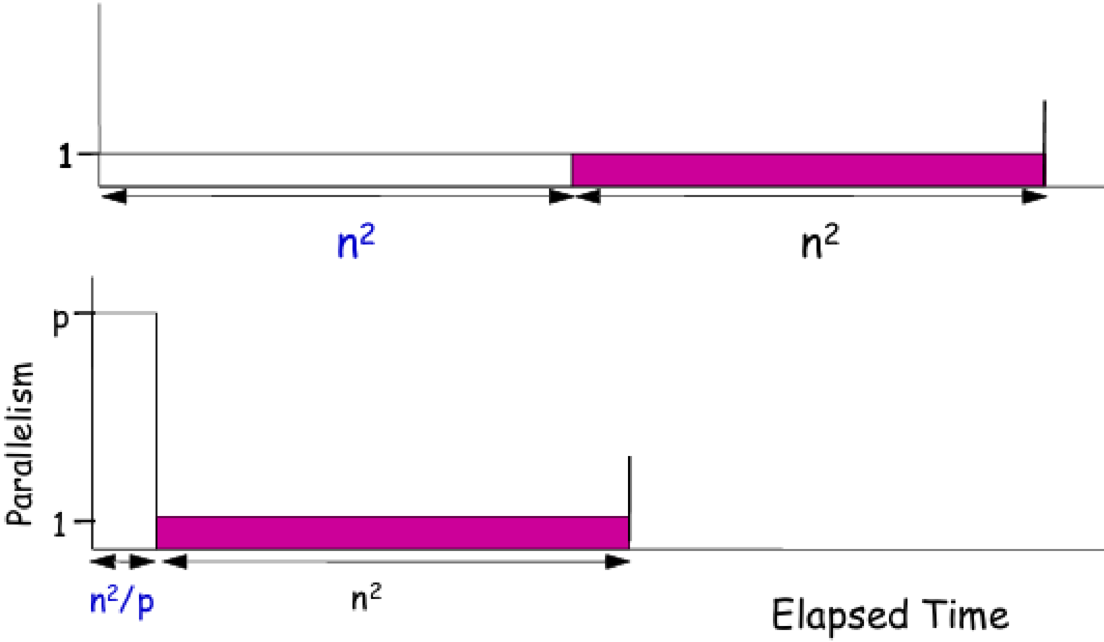


- Sequential Execution
  - both phases take n^2 time; $2n^2$ total



First Attempt at Parallelization

- Strategy
  - Phase 1: execute in parallel
    - time for phase 1 = n^2/p
  - Phase 2: execute serially
    - time for phase 2 = n^2



- Overall Performance
  - Speedup <= $\dfrac{2n^2}{\dfrac{n^2}{p} + n^2}$
  - i.e. no more than 2

Parallelizing Phase 2

- □ Trick: divide second phase into two steps
  - ➢ Step 1: accumulate into *private* sum during sweep
  - ➢ Step 2: add per-process private sum into global sum

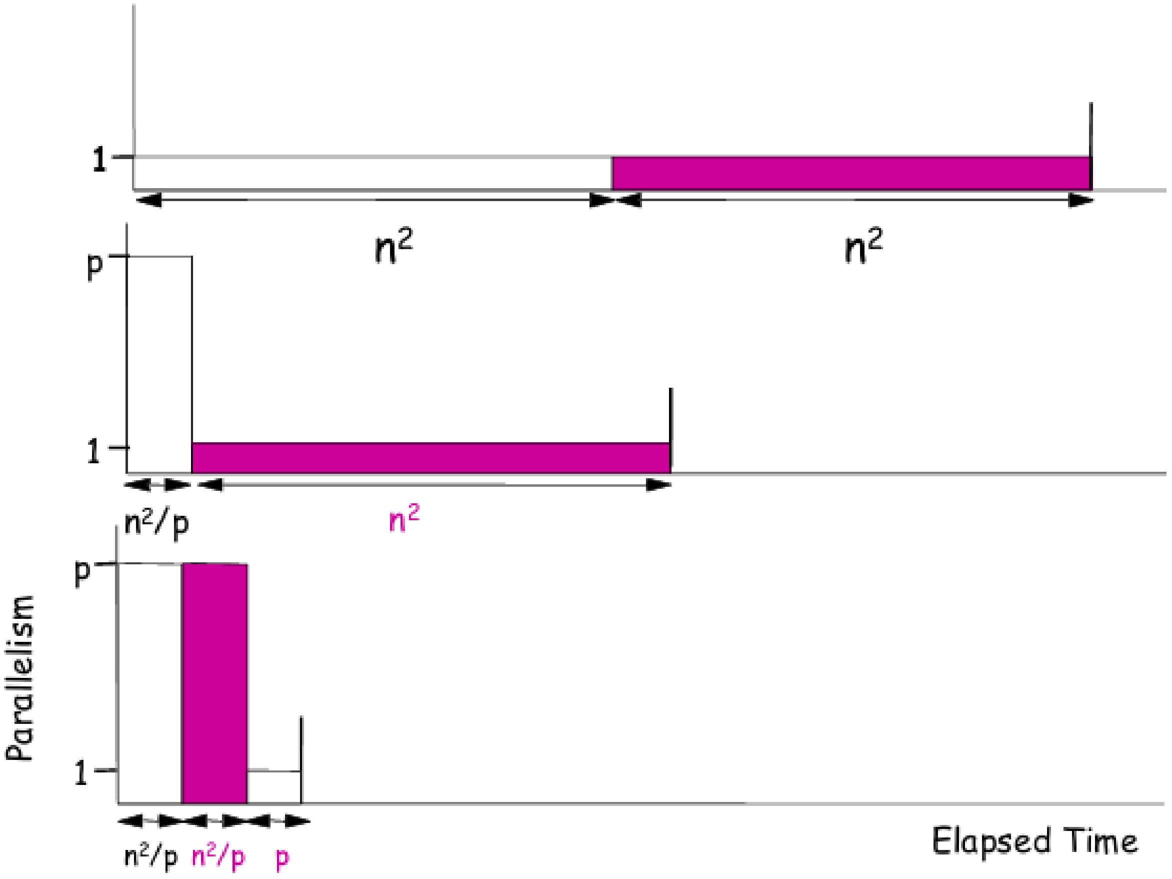
- □ Overall Performance:
  - ➢ Parallel time =

*n*²/*p* + *n*²/*p* + *p*

  - ➢ Speedup

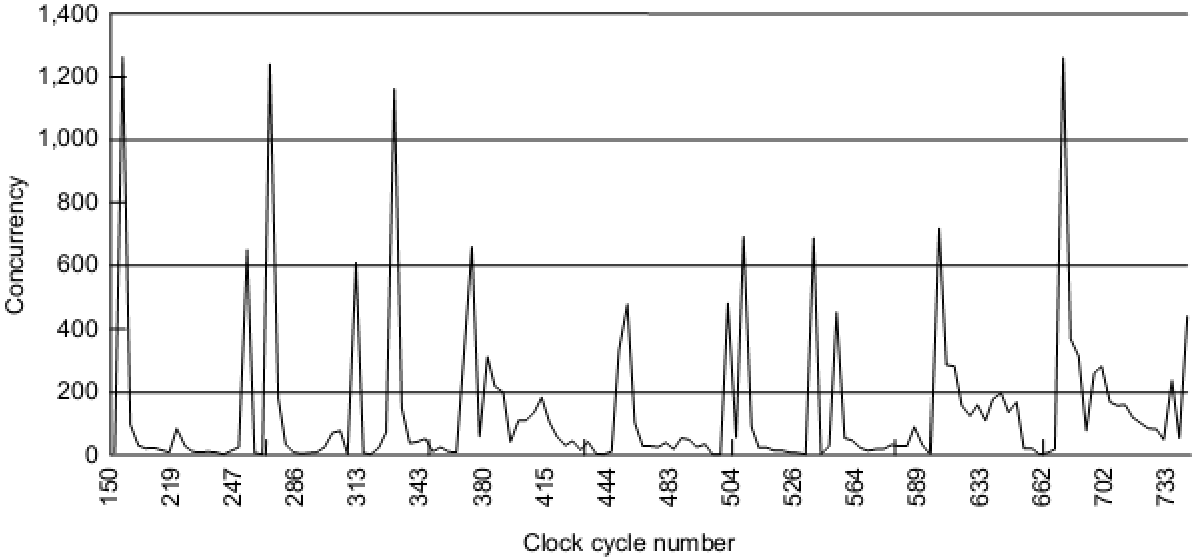
$$<= \frac{p2n^2}{2n^2 + p^2}$$

close to *p* if *n*>>*p*



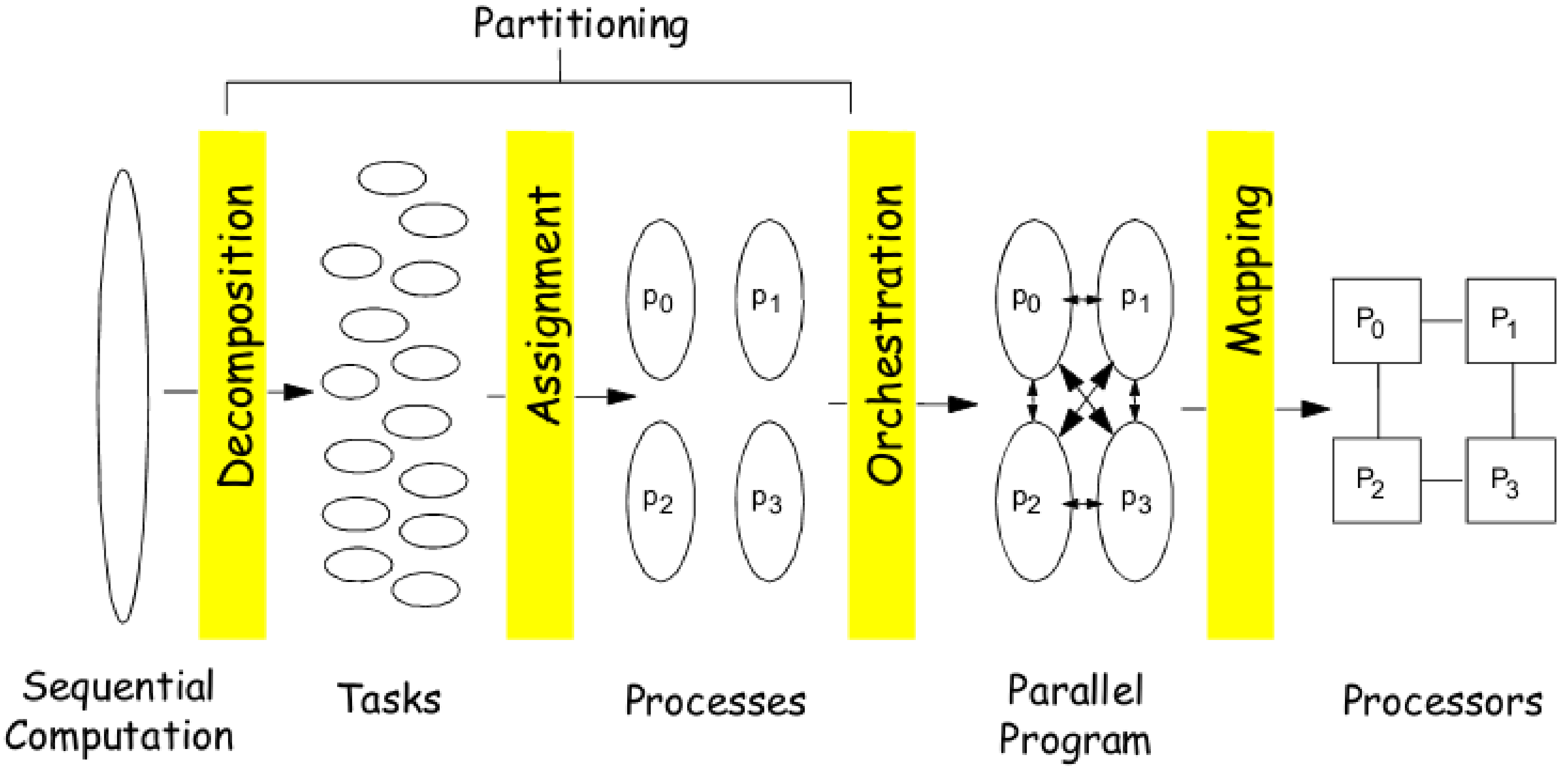
Concurrency Profiles

- □ Cannot usually divide into serial and fully parallel parts



- □ Area under curve is total work done, or time with 1 processor
- □ Horizontal extent is lower bound on time (infinite processors)
- □ Speedup is the ratio: $\dfrac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \dfrac{k}{p} \right\rceil}$, base case: $\dfrac{1}{s + \dfrac{1-s}{p}}$
- □ Amdahl’s law applies to any overhead, not just limited concurrency

Steps in Creating a Parallel Program



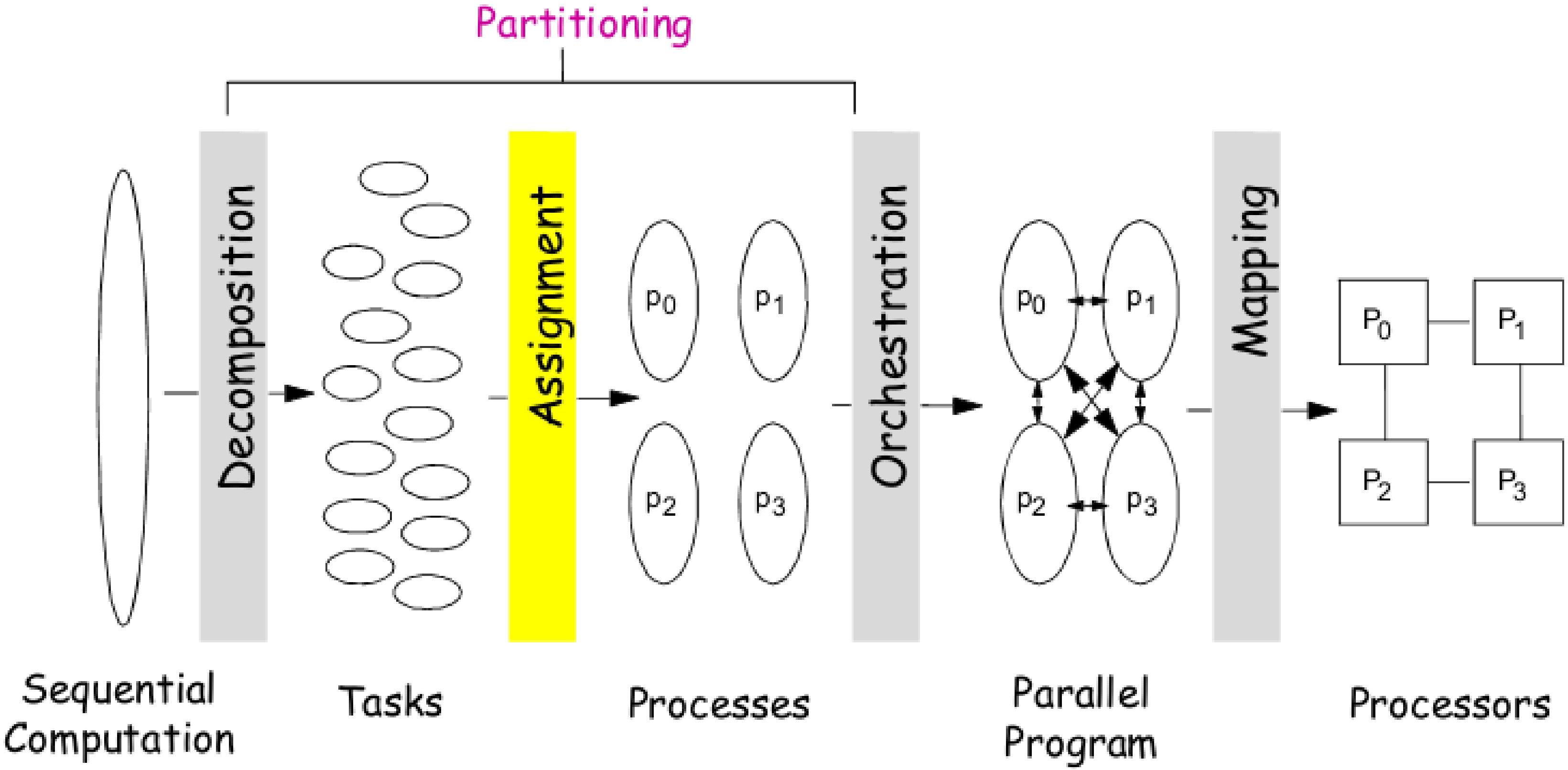
4 steps: Decomposition, Assignment, Orchestration, Mapping

- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly

Decomposition

- Break up computation into tasks to be divided among processes
  - i.e. Identify concurrency and decide level at which to exploit it
- Tasks may or may not be defined statically
  - Tasks may become available dynamically
  - Lots of available tasks may vary with time
- Goal: Enough tasks to keep processes busy, but not too many
  - Lots of tasks available at a time is upper bound on achievable speedup

Steps in Creating a Parallel Program

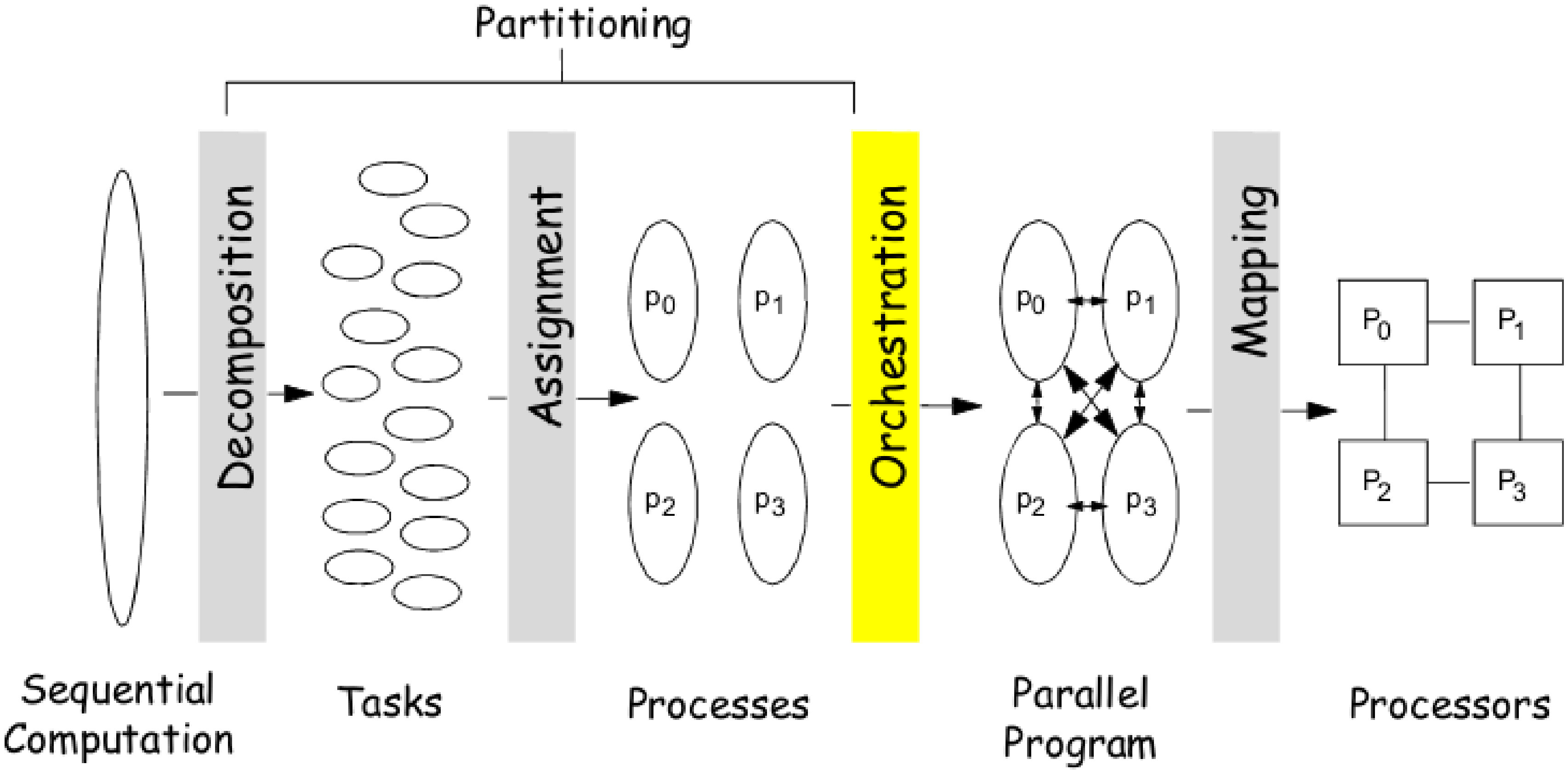


- 4 steps: Decomposition, Assignment, Orchestration, Mapping

Assignment

- Specifying mechanism to divide work among processes
  - e.g. which process computes forces on which stars, or which rays
  - Together with decomposition, also called *partitioning*
  - Goals: balance workload, reduce communication, and management cost
- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
  - Static versus dynamic assignment
- As programmers, we worry about partitioning first
  - Usually independent of architecture or programming model
  - But cost and complexity of using primitives may affect decisions
- As architects, we assume program does reasonable job of it

Steps in Creating a Parallel Program

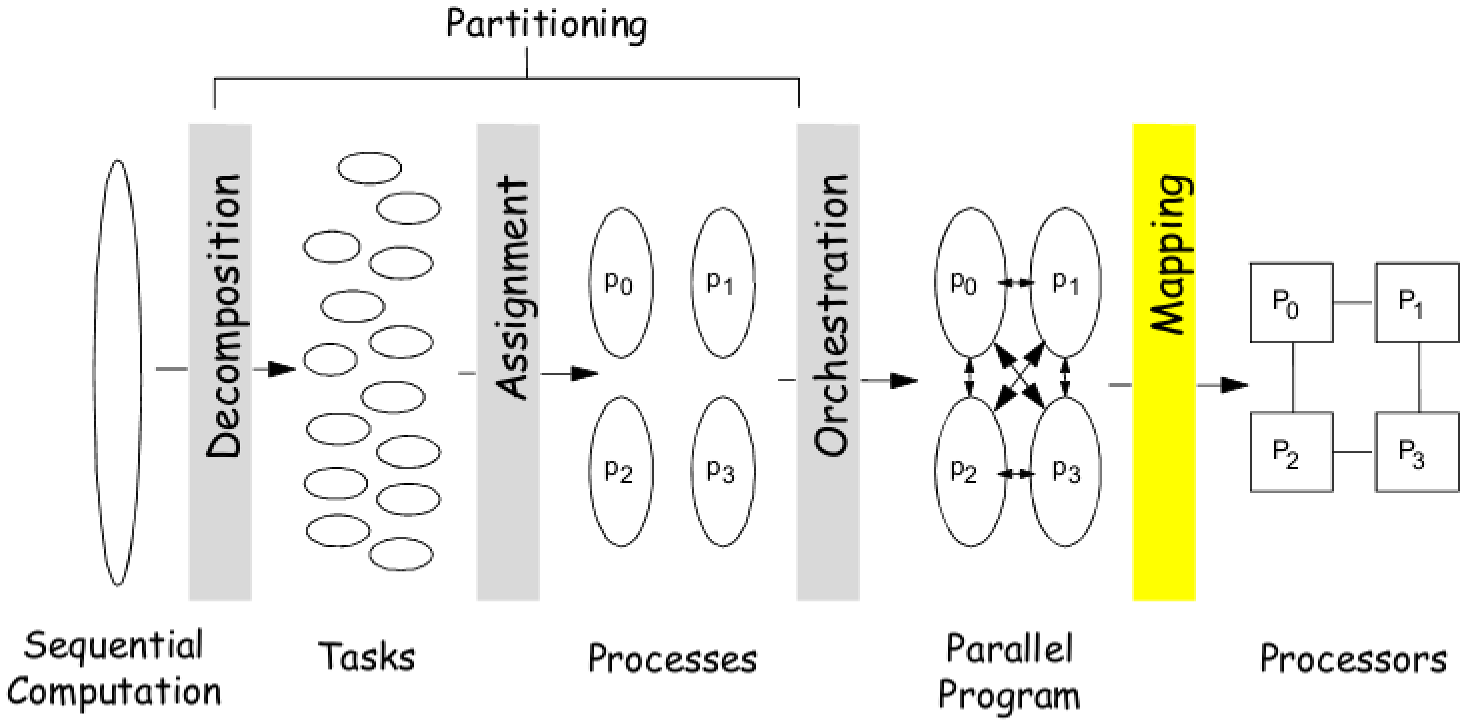


- 4 steps: Decomposition, Assignment, Orchestration, Mapping

Orchestration

- Main task
  - Naming data
  - Structuring communication
  - Synchronization
  - Organizing data structures and scheduling tasks temporally
- Goals
  - Reduce cost of communication and synchronization as seen by processors
  - Preserve locality of data reference (incl. data structure organization)
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Closest to architecture (and programming model & language)
  - Choices depend a lot on communication abstraction, efficiency of primitives
  - Architects should provide appropriate primitives efficiently

Steps in Creating a Parallel Program



- □ 4 steps: Decomposition, Assignment, Orchestration, Mapping

Mapping

- After orchestration, already have parallel program
- Two aspects of mapping
  - Which processes will run on same processor, if necessary
  - Which process runs on which particular processor
    - mapping to a network topology
- One extreme: space-sharing
  - Machine divided into subsets, only one application at a time in a subset
  - Processes can be pinned to processors, or left to OS
- Another extreme: complete resource management control to OS
  - OS uses the performance techniques we will discuss later
- Real world is between the two
  - User specifies desires in some aspects, system may ignore
- Usually adopt the view: process <-> processor

Parallelizing Computation vs. Data

- Above view is centered around computation
  - Computation is decomposed and assigned (partitioned)
- Partitioning data is often a natural view too
  - Computation follows data: *owner computes*
  - Grid example; data mining; High Performance Fortran (HPF)
- But not general enough
  - Distinction between computation and data stronger in many applications
  - Retain computation-centric view
  - Data access and communication is part of orchestration

High-level Goals

- □ High performance (speedup over sequential program)

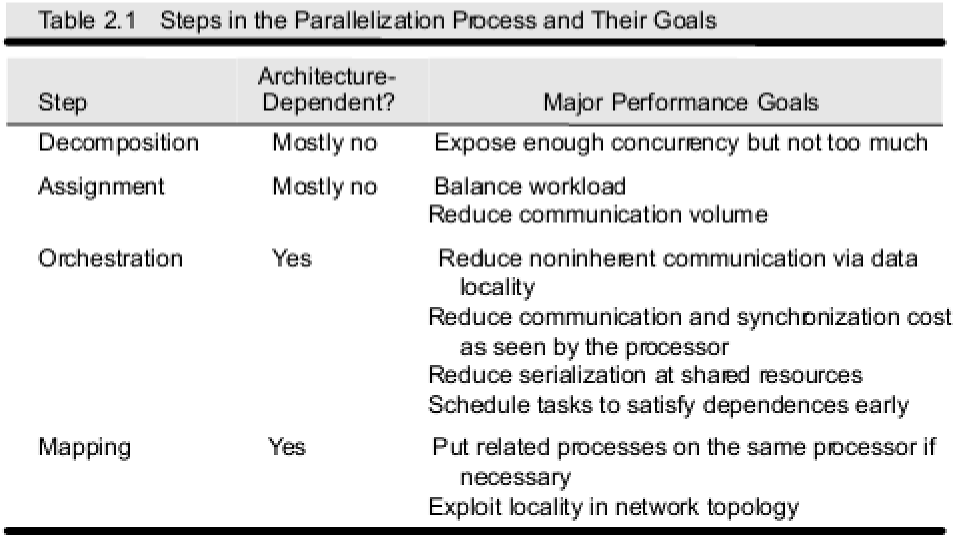


Table 2.1 Steps in the Parallelization Process and Their Goals

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

- □ But low resource usage and development effort
- □ Implications for algorithm designers and architects
  - ➢ Algorithm designers: high-performance, low resource needs
  - ➢ Architects: high-performance, low cost, reduced programming effort
    - • e.g. gradually improving performance with programming effort may be preferable to sudden threshold after large programming effort

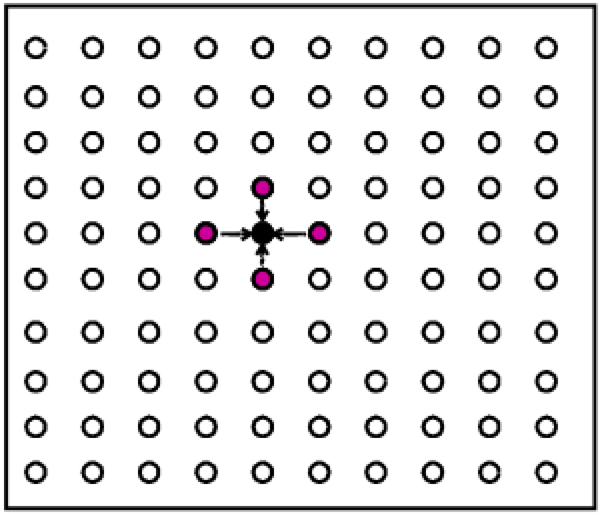
Parallel programming methodology

WHAT PARALLEL PROGRAMS LOOK LIKE

Parallelization of an Example Program

- Motivating problems all lead to large, complex programs
- Examine simplified version of a piece of Ocean simulation
  - Iterative equation solver
- Illustrate parallel program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives that must be supported
  - State of most real parallel programming today

Grid Solver Example



Expression for updating each interior point:

```
A[i,j] = 0.2 x (A[i,j]+A[i,j-1]+A[i-1,j]+
                     A[i,j+1]+A[i+1,j])
```

- Simplified version of solver in Ocean simulation
- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior *n*-by-*n* points of (*n*+2)-by-(*n*+2) updated in each sweep
  - updates done in-place in grid, and diff from previous value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if error has converged to (within a tolerance parameter)
  - if so, exit solver; if not, do another sweep

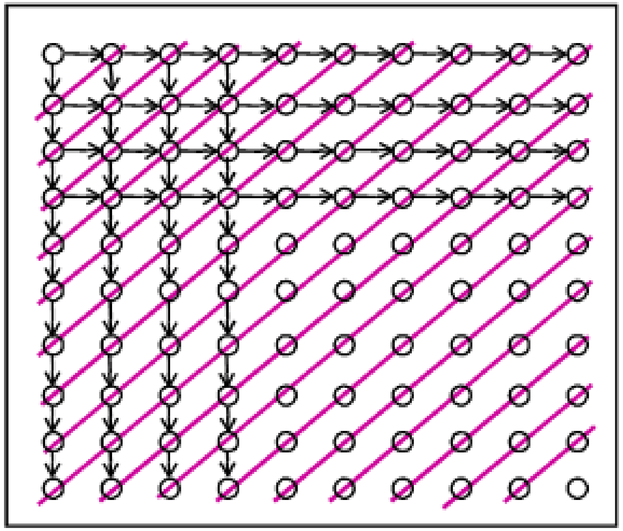
```
int n;                                          /*size of matrix: (n + 2-by-n + 2) elements*/
float **A, diff = 0;

main()
begin
   read(n) ;                                    /*read input parameter: matrix size*/
   A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
   initialize(A);                               /*initialize the matrix A somehow*/
   Solve (A);                                   /*call the routine to solve equation*/
end main

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 nonborder points of grid*/
         for j ← 1 to n 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 for
      end for
      if (diff/(n*n) < TOL) then done = 1;
   end while
end procedure
```

Decomposition

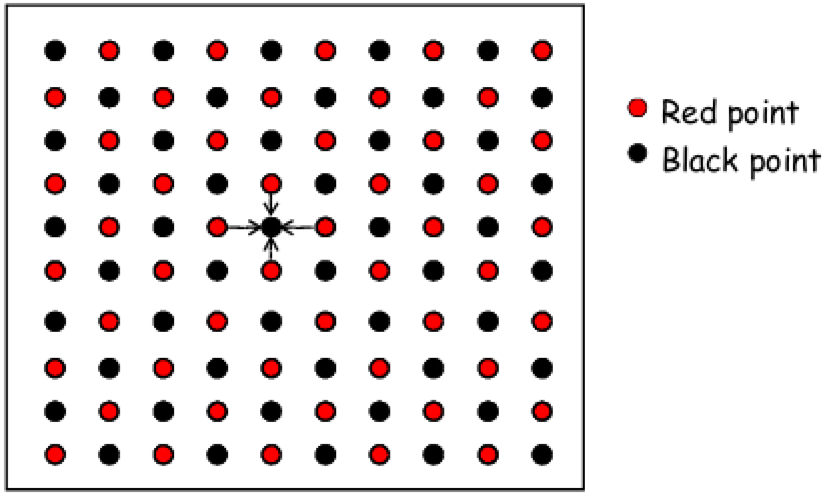
- □ Simple way to identify concurrency is to look at loop iterations
  - ➢ *dependence analysis*; if not enough concurrency, then look further
- □ Not much concurrency here at this level (all loops *sequential*)
- □ Examine fundamental dependences, ignoring loop structure



- □ Concurrency *O(n)* along anti-diagonals, serialization *O(n)* along diag
- □ Retain loop structure, use pt-to-pt synch; Problem: too many synch ops
- □ Restructure loops, use global synch; imbalance and too much synch

Exploit Application Knowledge

- Reorder grid traversal: red-black ordering



- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel
- Global synch between them (conservative but convenient)
- Ocean uses red-black; we use simpler, asynchronous one to illustrate
  - no red-black, simply ignore dependences within sweep
  - sequential order same as original, parallel program *nondeterministic*

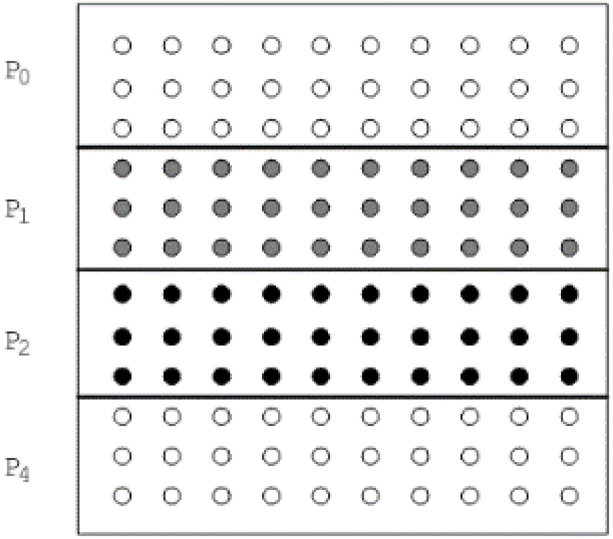
Decomposition Only

```
15. while (!done) do                         /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do                  /*a parallel loop nest*/
18.     for_all j ← 1 to n do
19.       temp = A[i,j];
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]);
22.       diff += abs(A[i,j] - temp);
23.     end for_all
24.   end for_all
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
```

- □ Decomposition into elements: degree of concurrency n^2
- □ To decompose into rows, make line 18 loop sequential; degree n
- □ for_all leaves assignment to the system
  - ➢ but implicit global synch. at end of for_all loop

Assignment

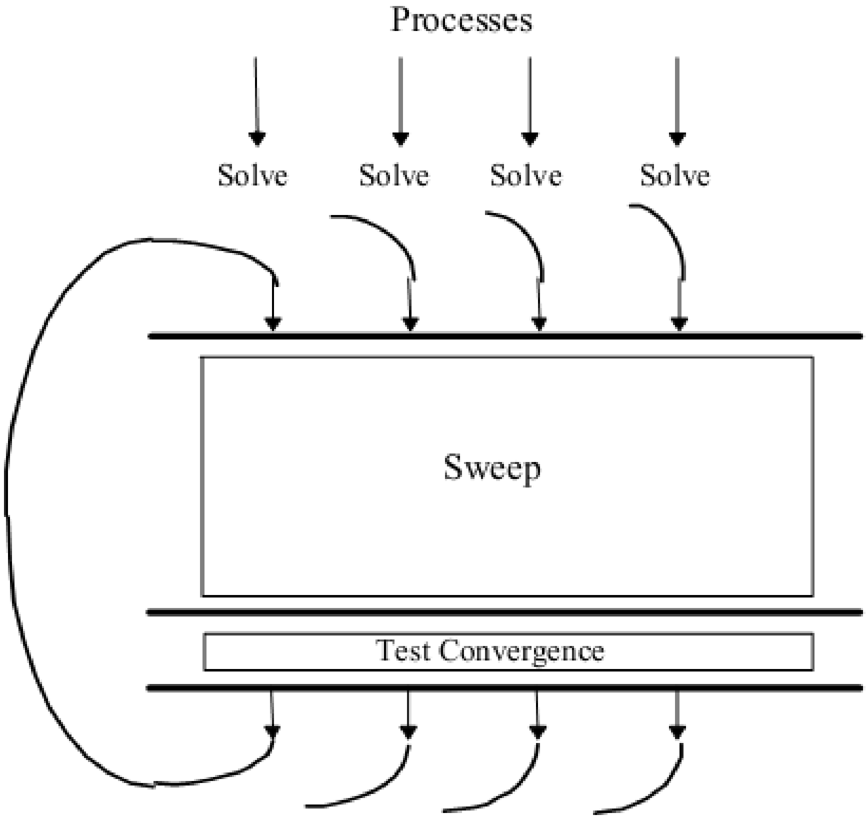
- Static assignments (given decomposition into rows)
  - block assignment of rows: Row *i* is assigned to process $\left\lfloor \frac{i}{p} \right\rfloor$
  - cyclic assignment of rows: process *i* is assigned rows *i*, *i+p* , and so on



- Dynamic assignment
  - get a row index, work on the row, get a new row, and so on
- Static assignment into rows reduces concurrency (from *n* to p)
  - block assignment reduces communication by keeping adjacent rows together
- Let's dig into orchestration under two programming models

Shared Address Space Solver

Single Program Multiple Data (SPMD)



- Assignment controlled by values of variables used as loop bounds

```
    int n, nprocs;              /*matrix dimension and number of processors to be used*/
    float **A, diff;            /*A is global (shared) array representing the grid*/
                                /*diff is global (shared) maximum difference in current
                                sweep*/
    LOCKDEC(diff_lock);         /*declaration of lock to enforce mutual exclusion*/
    BARDEC (bar1);              /*barrier declaration for global synchronization between
                                sweeps*/

main()
begin
        read(n); read(nprocs);          /*read input matrix size and number of processes*/
        A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
        initialize(A);                  /*initialize A in an unspecified way*/
        CREATE (nprocs-1, Solve, A);
    Solve(A);                           /*main process becomes a worker too*/
        WAIT_FOR_END (nprocs-1);        /*wait for all child processes created to terminate*/
end main

procedure Solve(A)
    float **A;                                  /*A is entire n+2-by-n+2 shared array,
                                                as in the sequential program*/
begin
    int i,j, pid, done = 0;
    float temp, mydiff = 0;                     /*private variables*/
    int mymin = 1 + (pid * n/nprocs);           /*assume that n is exactly divisible by*/
    int mymax = mymin + n/nprocs - 1            /*nprocs for simplicity here*/

    while (!done) do                        /*outer loop over all diagonal elements*/
        mydiff = diff = 0;                  /*set global diff to 0 (okay for all to do it)*/
    BARRIER(bar1, nprocs);                  /*ensure all reach here before anyone modifies diff*/
        for i ← mymin to mymax do           /*for each of my rows*/
                for j ← 1 to n do           /*for all nonborder elements in that row*/
                temp = A[i,j];
                A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                    A[i,j+1] + A[i+1,j]);
                mydiff += abs(A[i,j] - temp);
            endfor
        endfor
        LOCK(diff_lock);                    /*update global diff if necessary*/
        diff += mydiff;
        UNLOCK(diff_lock);
        BARRIER(bar1, nprocs);              /*ensure all reach here before checking if done*/
        if (diff/(n*n) < TOL) then done = 1;        /*check convergence; all get
                                                    same answer*/
        BARRIER(bar1, nprocs);
    endwhile
end procedure
```

Notes on SAS Program

- SPMD: not lockstep or even necessarily same instructions
- Assignment controlled by values of variables used as loop bounds
  - Unique *pid* per process, used to control assignment
- “*Done*” condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - Each process has private *mydiff* variable
- Most interesting special operations are for synchronization
  - Accumulations into shared *diff* have to be mutually exclusive
  - Why the need for all the barriers?

Need for Mutual Exclusion

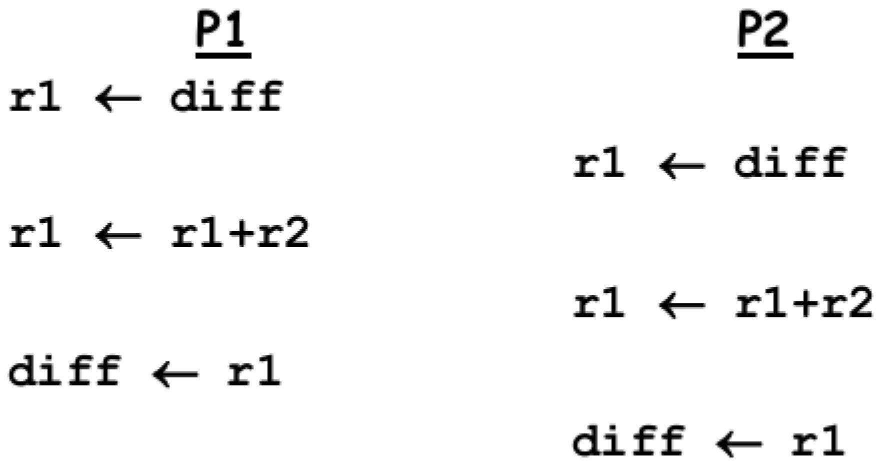
- Code each process executes

  load the value of *diff* into register *r1*

  add the register *r2* to register *r1*

  store the value of register *r1* into diff

- A possible interleaving



P1	P2
r1 ← diff	
	r1 ← diff
r1 ← r1+r2	
	r1 ← r1+r2
diff ← r1	
	diff ← r1

- Need the sets of operations to be atomic (mutually exclusive)

Mutual Exclusion

- Provided by LOCK-UNLOCK around *critical section*
  - Set of operations we want to execute atomically
  - Implementation of LOCK/UNLOCK must guarantee mutual exclusive
- Can lead to significant serialization if contended
  - Especially since expect non-local accesses in critical section
  - Another reason to use private *mydiff* for partial accumulation

Global Event Synchronization

- BARRIER(*nprocs*): wait here till *nprocs* processes get here
  - Built using lower level primitives
  - Global sum example: wait for all to accumulate before using sum
  - Often used to separate phases of computation

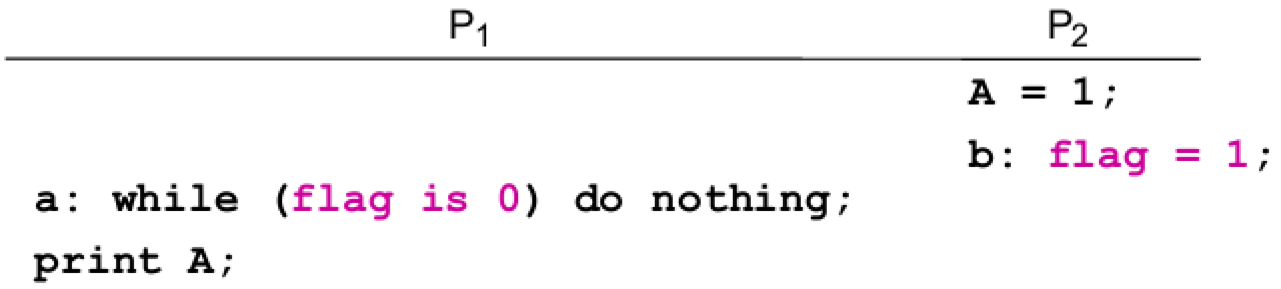
Process P_1	*Process P_2*	*Process P_nprocs*
set up eqn system	set up eqn system	set up eqn system
Barrier (name, nprocs)	Barrier (name, nprocs)	Barrier (name, nprocs)
solve eqn system	solve eqn system	solve eqn system
Barrier (name, nprocs)	Barrier (name, nprocs)	Barrier (name, nprocs)
apply results	apply results	apply results
Barrier (name, nprocs)	Barrier (name, nprocs)	Barrier (name, nprocs)

  - Conservative form of preserving dependences, but easy to use

- WAIT_FOR_END (*nprocs*-1)

Pt-to-pt Event Synch

- One process notifies another event so it can proceed
  - Common example: producer-consumer (bounded buffer)
  - Concurrent programming on uniprocessor: semaphores
  - Shared address space parallel programs: semaphores , or use ordinary variables as flags



```
                     P1                                  P2
------------------------------------------------------------------
                                              A = 1;
                                              b: flag = 1;
a: while (flag is 0) do nothing;
print A;
```

  - Busy-waiting or spinning

Group Event Synchronization

- Subset of processes involved
  - Can use flags or barriers (involving only the subset)
  - Concept of producers and consumers
- Major types
  - Single-producer, multiple-consumer
  - Multiple-producer, single-consumer
  - Multiple-producer, multiple-consumer

Message Passing Grid Solver

- Cannot declare *A* to be shared array any more
- Need to compose it logically from per-process private arrays
  - Usually allocated in accordance with the assignment of work
  - Process assigned a set of rows allocates them locally
- Transfers of entire rows between traversals
- Structurally similar to SAS (e.g. SPMD), but orchestration different
  - Data structures and data access/naming
  - Communication
  - Synchronization

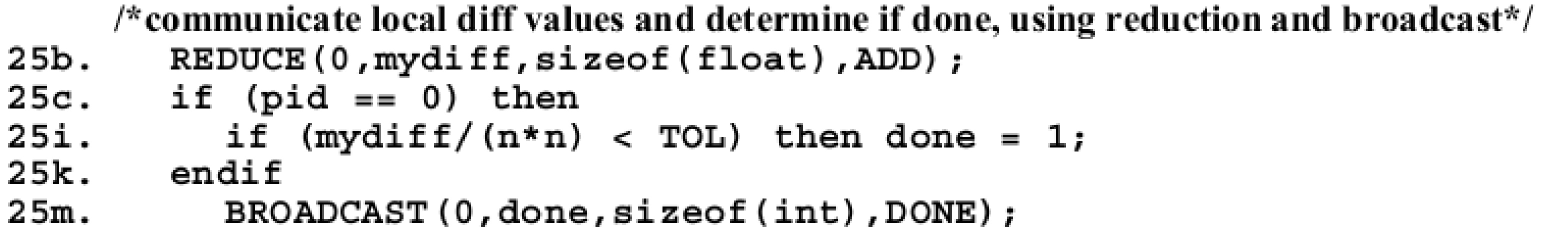
```
int pid, n, b;                /*process id, matrix dimension and number of
                                processors to be used*/
float **myA;
main()
begin
    read(n);   read(nprocs);  /*read input matrix size and number of processes*/
    CREATE (nprocs-1, Solve);
    Solve();                  /*main process becomes a worker too*/
    WAIT_FOR_END (nprocs-1);  /*wait for all child processes created to terminate*/
end main

  procedure Solve()
  begin
    int i,j, pid, n' = n/nprocs, done = 0;
    float temp, tempdiff, mydiff = 0;   /*private variables*/
  myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                              /*my assigned rows of A*/
initialize(myA);              /*initialize my rows of A, in an unspecified way*/

  while (!done) do
    mydiff = 0;               /*set local diff to 0*/
    if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
    if (pid != nprocs-1) then
      SEND(&myA[n',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[n'+1,0],n*sizeof(float), pid+1,ROW);
                              /*border rows of neighbors have now been copied
                              into myA[0,*] and myA[n'+1,*]*/
    for i ← 1 to n' do        /*for each of my (nonghost) rows*/
      for j ← 1 to n do       /*for all nonborder elements in that row*/
        temp = myA[i,j];
        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
          myA[i,j+1] + myA[i+1,j]);
        mydiff += abs(myA[i,j] - temp);
      endfor
    endfor
                              /*communicate local diff values and determine if
                              done; can be replaced by reduction and broadcast*/
      if (pid != 0) then            /*process 0 holds global total diff*/
        SEND(mydiff,sizeof(float),0,DIFF);
        RECEIVE(done,sizeof(int),0,DONE);
      else                          /*pid 0 does this*/
        for i ← 1 to nprocs-1 do    /*for each other process*/
          RECEIVE(tempdiff,sizeof(float),*,DIFF);
        mydiff += tempdiff;         /*accumulate into total*/
      endfor
      if (mydiff/(n*n) < TOL) then    done = 1;
        for i ← 1 to nprocs-1 do    /*for each other process*/
          SEND(done,sizeof(int),i,DONE);
        endfor
    endif
  endwhile
  end procedure
```

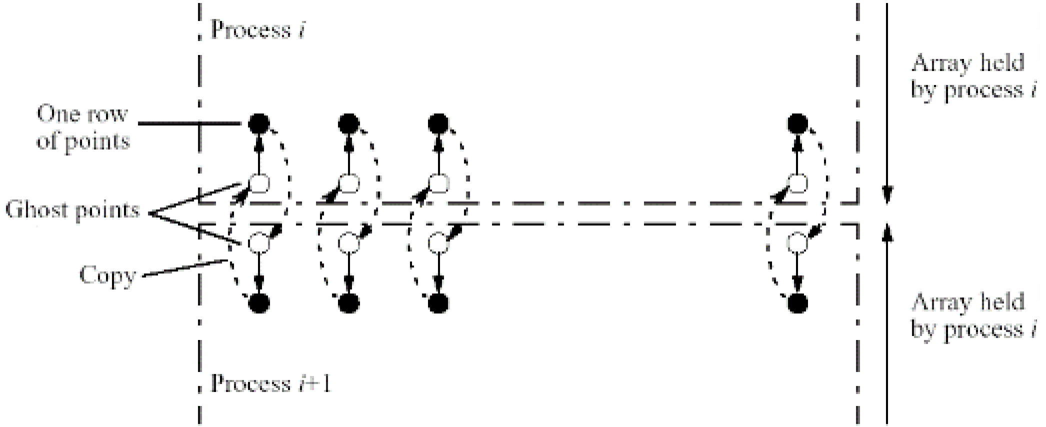
Notes on Message Passing Program

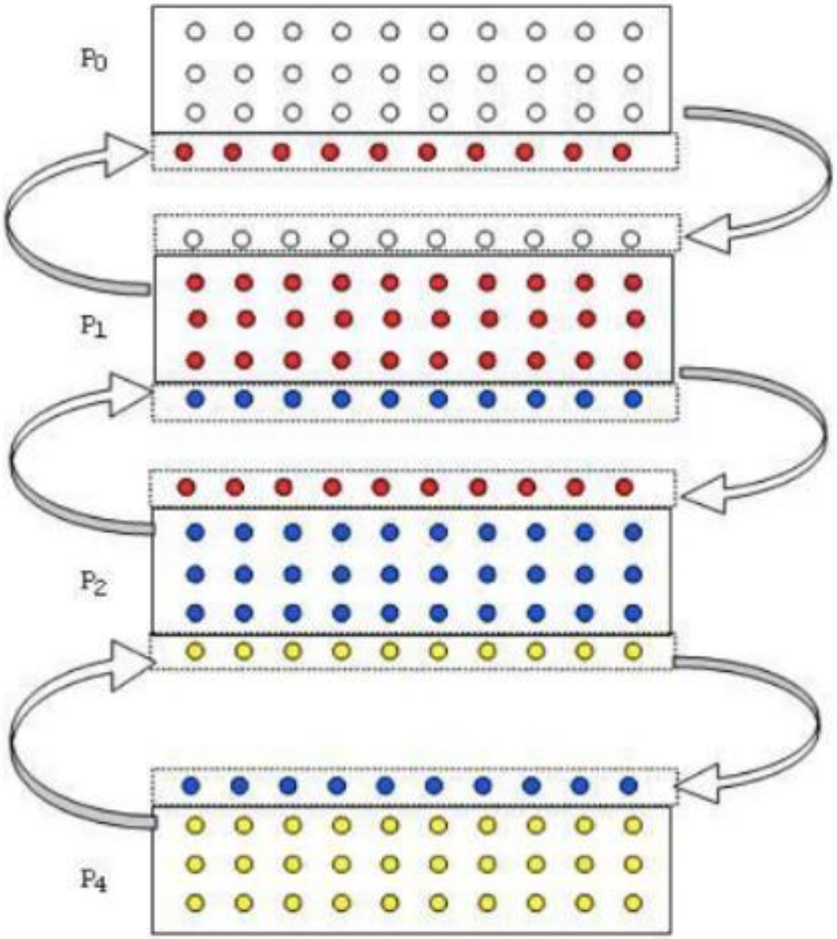
- Use of ghost rows
- Receive does not transfer data, send does
  - Unlike SAS which is usually receiver-initiated (load fetches data)
- Communication done at beginning of iteration, so no asynchrony
- Communication in whole rows, not element at a time
- Core similar, but indices/bounds in local rather 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



```
          /*communicate local diff values and determine if done, using reduction and broadcast*/
25b.    REDUCE(0,mydiff,sizeof(float),ADD);
25c.    if (pid == 0) then
25i.       if (mydiff/(n*n) < TOL) then done = 1;
25k.    endif
25m.       BROADCAST(0,done,sizeof(int),DONE);
```

Ghost Points and Ghost Row

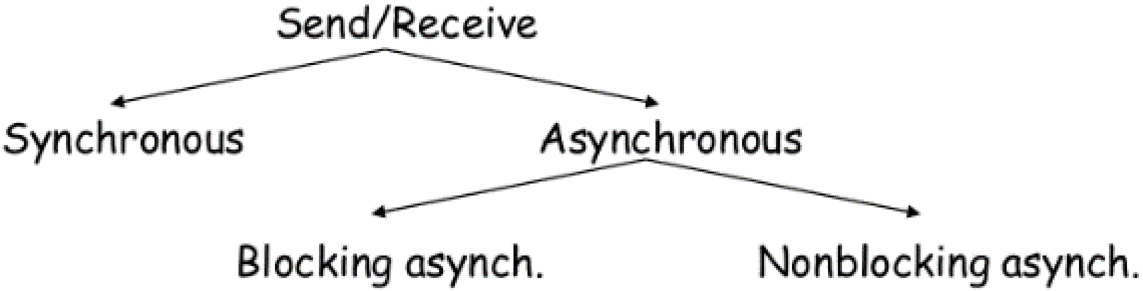




Send and Receive Alternatives

- Can extend functionality: stride, scatter-gather, groups
- Semantic flavors: based on when control is returned

Affect when data structures or buffers can be reused at either end



  - Affect event synch (mutual exclusive: only one process touches data)
  - Affect ease of programming and performance
- Synchronous messages provide built-in synchronous through match
  - Separate event synchronization needed with asynchronous messages
- With synchronous messages, our code is deadlocked

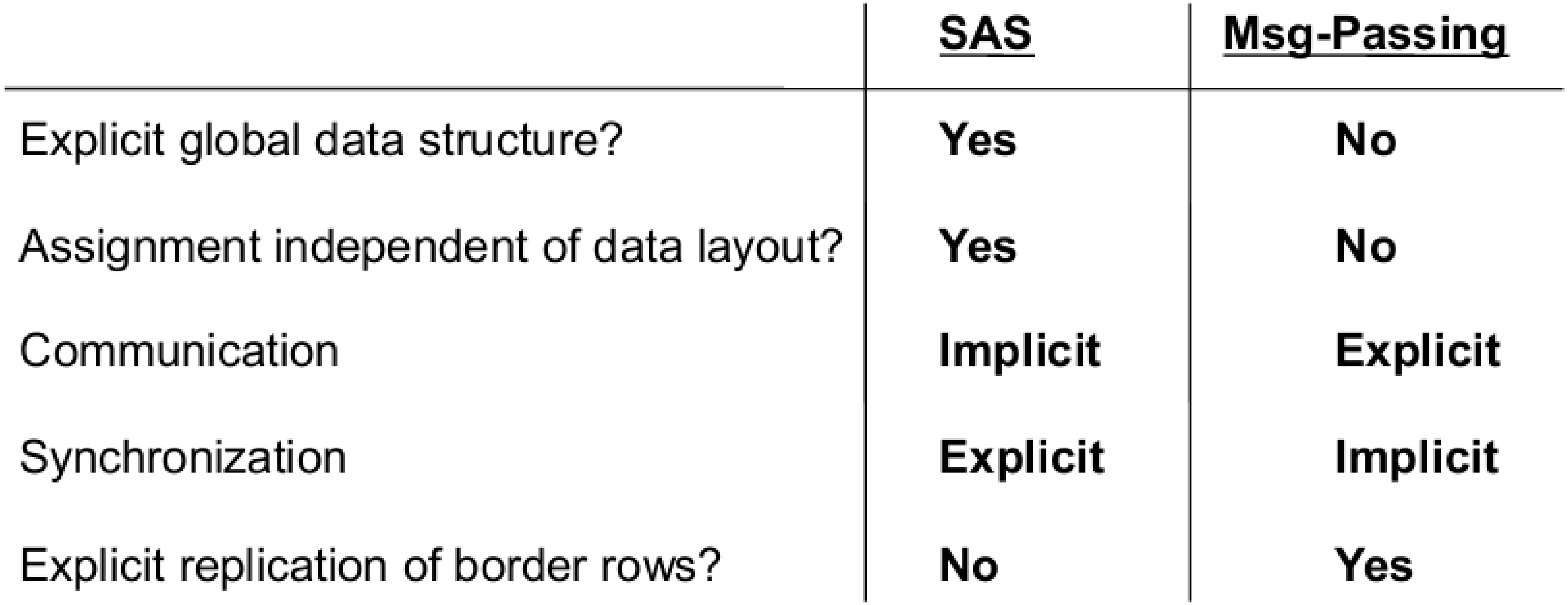
Orchestration: Comparison

- Shared address space
  - Shared and private data explicitly separate
  - Communication implicit in access patterns
  - No correctness need for data distribution
  - 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 communication patterns)
  - Communication is explicit
  - Synchronization implicit in communication (at least in synchronous case)

Summary in Grid Solver Program

- Decomposition and assignment similar in SAS and message-passing
- Orchestration is different
  - Data structures, data access/naming, communication, synchronization



	SAS	Msg-Passing
Explicit global data structure?	Yes	No
Assignment independent of data layout?	Yes	No
Communication	Implicit	Explicit
Synchronization	Explicit	Implicit
Explicit replication of border rows?	No	Yes