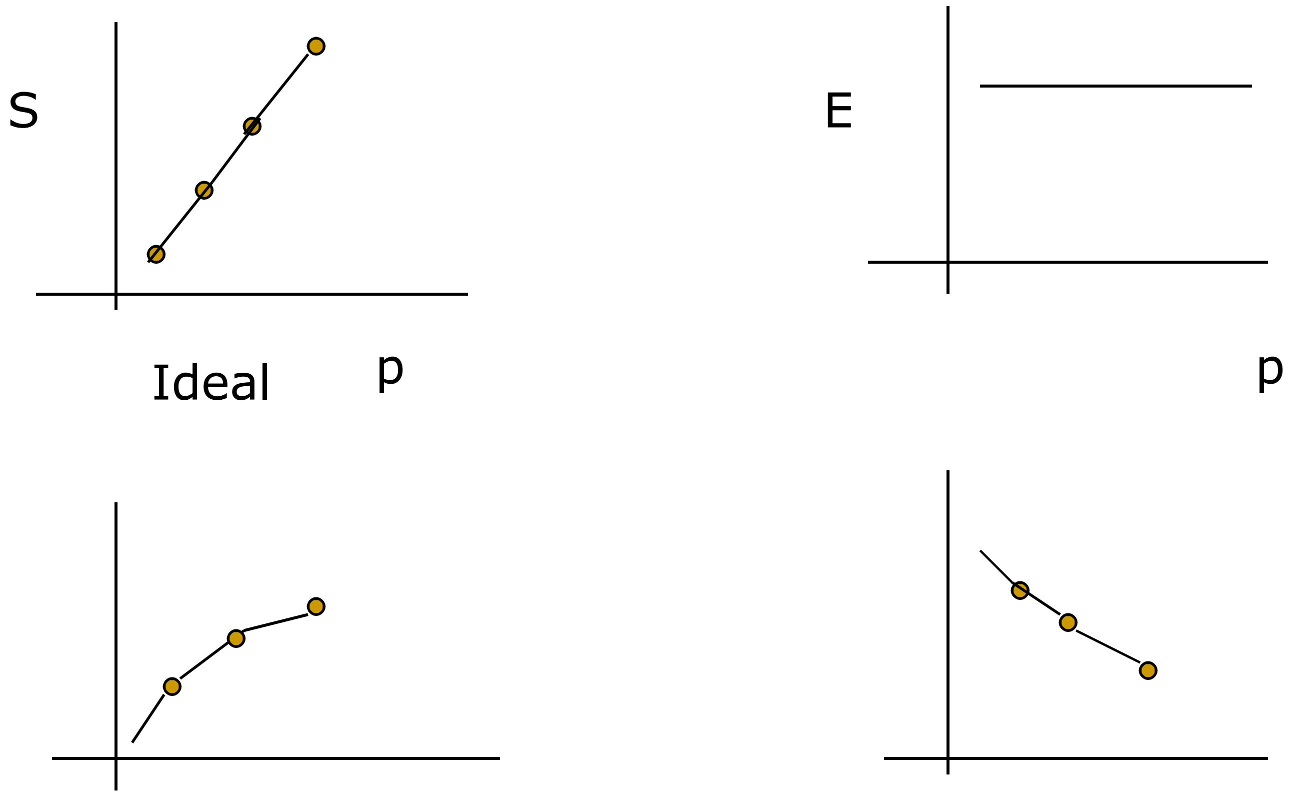
Scalability

How do we evaluate a parallel program?

- Execution time, T_p
- Speedup, S
  - $S(p, n) = T(1, n) / T(p, n)$
  - Usually, $S(p, n) < p$
  - Sometimes $S(p, n) > p$ (superlinear speedup)
- Efficiency, E
  - $E(p, n) = S(p, n)/p$
  - Usually, $E(p, n) < 1$
  - Sometimes, greater than 1
- **Scalability** – Limitations in parallel computing, relation to *n* and *p*.

Speedups and efficiency



Practical

Limitations on speedup – Amdahl’s law

- Amdahl's law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Overall speedup in terms of fractions of computation time with and without enhancement, % increase in enhancement.
- Places a limit on the speedup due to parallelism.
- Speedup = $\dfrac{1}{(f_s + (f_p/P))}$

Scalability

- Suppose we run a parallel program with a fixed number of processes / threads and a fixed input size, and we obtain an efficiency E
- Suppose we now increase the number of processes / threads that are used by the program.
  - If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is **scalable**.

Scalability

- Efficiency decreases with increasing P; increases with increasing N
- How effectively the parallel algorithm can use an increasing number of processors
- How the amount of computation performed must scale with P to keep E constant
- This function of computation in terms of P is called isoefficiency function.
- An algorithm with an isoefficiency function of O(P) is highly scalable while an algorithm with quadratic or exponential isoefficiency function is poorly scalable

Scalability

- Example: suppose that $T_{serial} = n$, where the units of T_{serial} are in microseconds and n is also the problem size.

- Also suppose that $T_{parallel} = n/p + 1$. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}$$

- To see if the program is scalable, we increase the number of processes/threads by a factor of k, and we want to find the factor x that we need to increase the problem size by so that E is unchanged.

Scalability

- The number of processes / threads will be kp and the problem size will be xn, and we want to solve the equation for x :

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}$$

- Well, if x=k, there will be a common factor of k in the denominator xn + kp = kn+kp=k(n+p)

  and we can reduce the fraction to get

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}$$

Scalability

- In other words, if we increase the problem size at the same rate that we increase the number of processes / threads, then the efficiency will be unchanged, and our program is scalable.

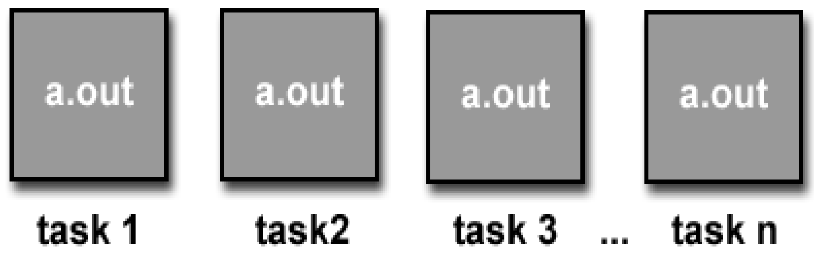
Scalability : 2 cases

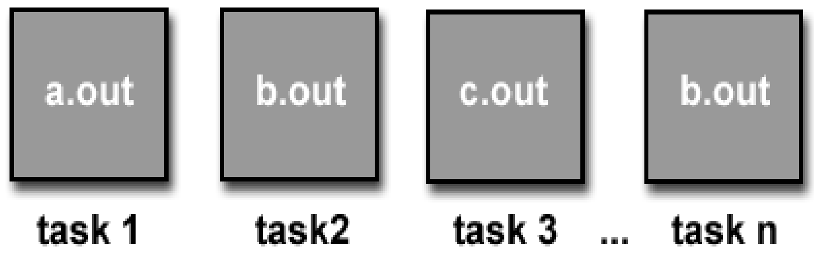
- Case 1: when we increase the no. of processes / threads, we can keep the efficiency fixed without increasing the problem size, the program is said to strongly scalable.
- Case 2: When we keep the efficiency fixed by increasing the problem size at the same rate as we increase the no. of processes / threads, then the program is said to weakly scalable.

PARALLEL PROGRAMMING CLASSIFICATION AND STEPS

Parallel Program Models

- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)





Courtesy: http://www.llnl.gov/computing/tutorials/parallel_comp/

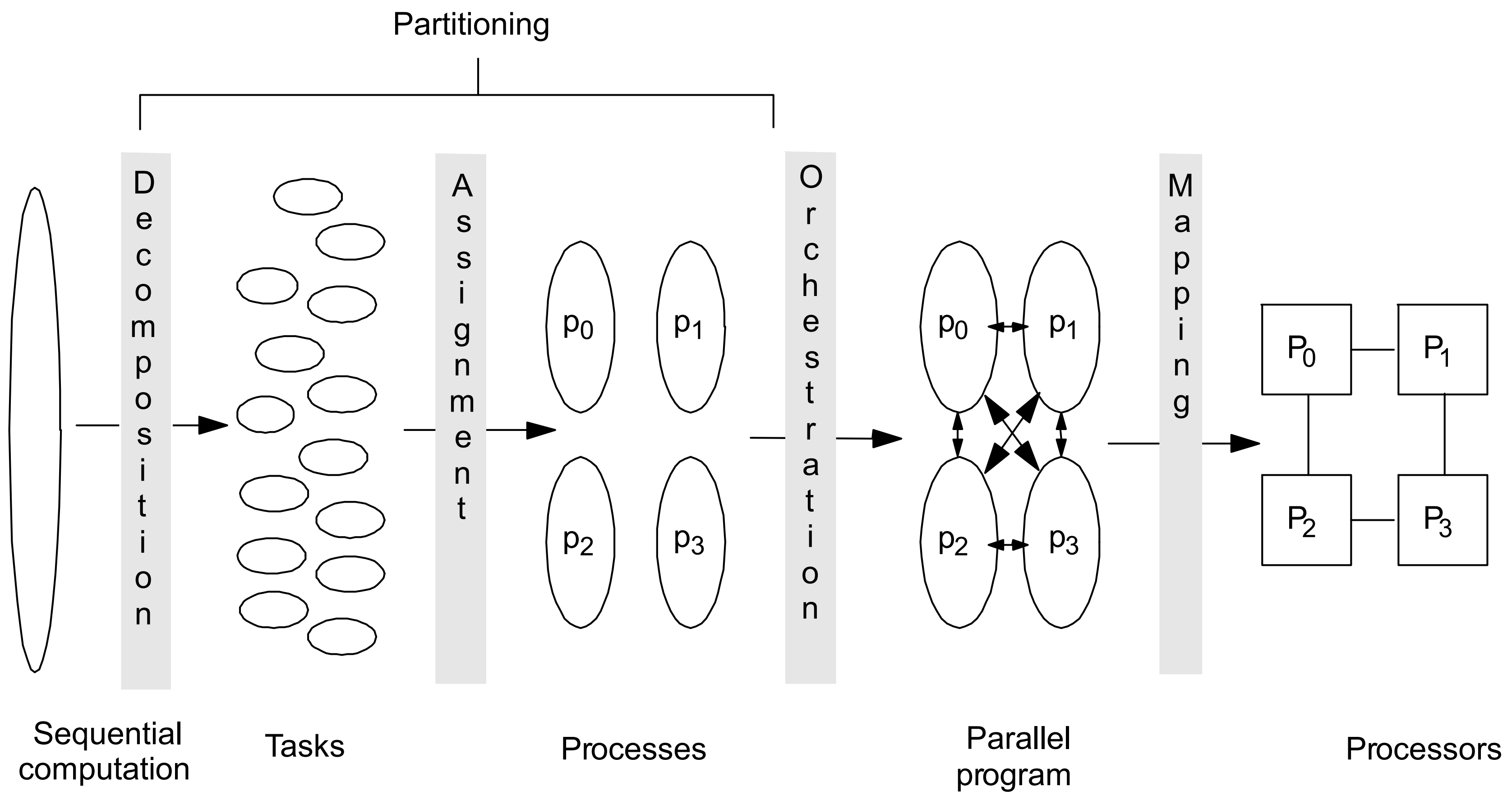
Programming Paradigms

- Shared memory model – Threads, OpenMP, CUDA
- Message passing model – MPI

Parallelizing a Program

- Given a sequential program/algorithm, how to go about producing a parallel version
- Four steps in program parallelization
  1. Decomposition
     - Identifying parallel tasks with large extent of possible concurrent activity; splitting the problem into tasks
  2. Assignment
     - Grouping the tasks into processes with best load balancing
  3. Orchestration
     - Reducing synchronization and communication costs
  4. Mapping
     - Mapping of processes to processors (if possible)

Steps in Creating a Parallel Program



Decomposition and 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
  - Structuring communication
  - Synchronization
- Challenges
  - Organizing data structures – packing
  - Small or large messages?
  - How to organize communication and synchronization ?

Orchestration

- Maximizing data locality
  - Minimizing volume of data exchange
    - Not communicating intermediate results – e.g. dot product
  - Minimizing frequency of interactions - packing
- Minimizing contention and hot spots
  - Do not use the same communication pattern with the other processes in all the processes
- Overlapping computations with interactions
  - Split computations into phases: those that depend on communicated data (type 1) and those that do not (type 2)
  - Initiate communication for type 1; During communication, perform type 2
- Replicating data or computations
  - Balancing the extra computation or storage cost with the gain due to less communication

Mapping

- Which process runs on which particular processor?
  - Can depend on network topology, communication pattern of processes
  - On processor speeds in case of heterogeneous systems

Mapping

- Static mapping
  - Mapping based on Data partitioning
    - Applicable to dense matrix computations
    - Block distribution

0	0	0	1	1	1	2	2	2

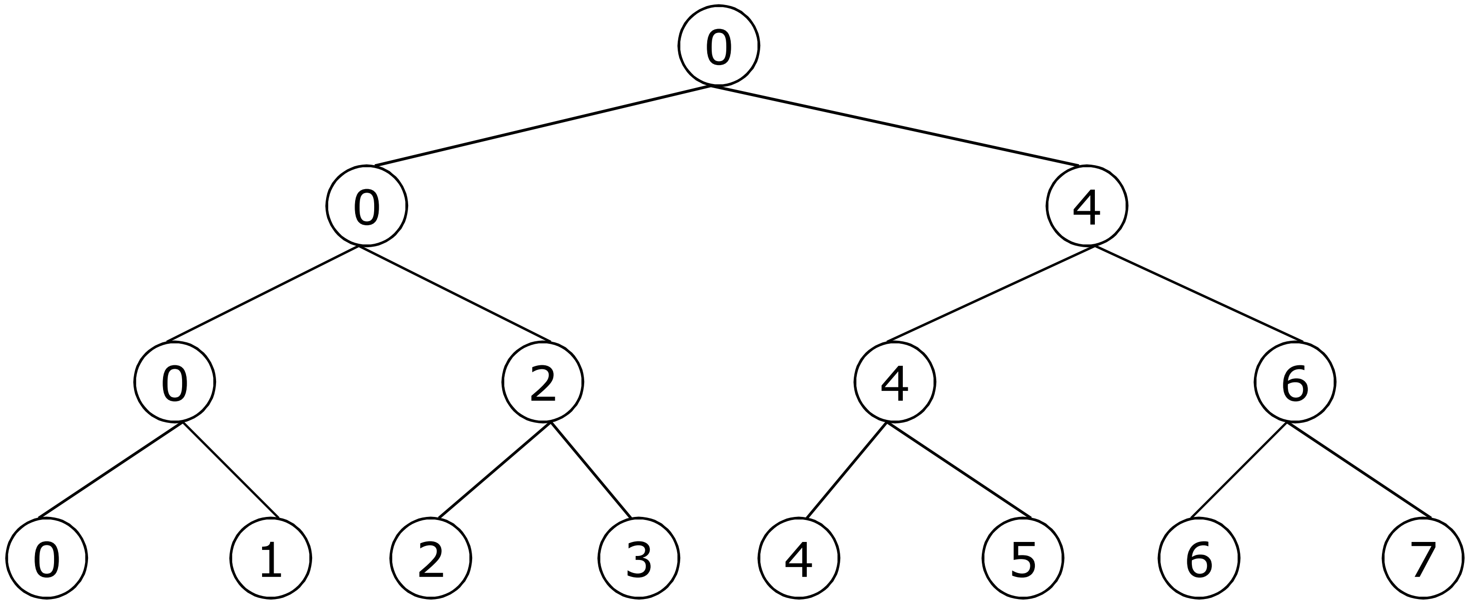
    - Block-cyclic distribution

0	1	2	0	1	2	0	1	2

  - Graph partitioning based mapping
    - Applicable for sparse matrix computations
  - Mapping based on task partitioning

Based on Task Partitioning

- Based on **task dependency graph**



- In general the problem is NP complete

Mapping

- Dynamic Mapping
  - A process/global memory can hold a set of tasks
  - Distribute some tasks to all processes
  - Once a process completes its tasks, it asks the coordinator process for more tasks
  - Referred to as *self-scheduling, work-stealing*

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

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;

Example

Given a 2-d array of float values, repeatedly average each elements with immediate neighbours until the difference between two iterations is less than some tolerance value

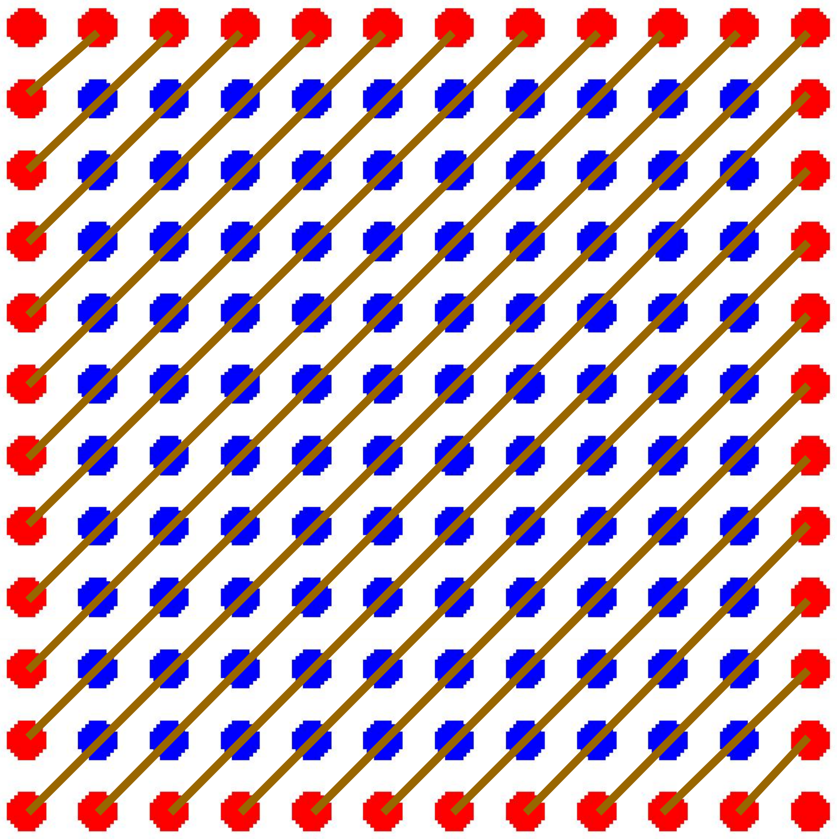
```
do {
    diff = 0.0
    for (i=0; i < n; i++)
        for (j=0; j < n, j++){
            temp = A[i] [j];
            A[i][j] = average (neighbours);
            diff += abs (A[i][j] – temp);
        }
    while  (diff > tolerance) ;
```

	A[i-1][j]	
A[i][j-1]	A[i][j]	A[i][j+1]
	A[i+1][j]	

Assignment Options

1. A concurrent task for each element update
   - Max concurrency: n**2
   - Synch: wait for left & top values
   - High synchronization cost
2. Concurrent tasks for elements in anti-diagonal
   - No dependence among elements in a diagonal
   - Max concurrency: ~ n
   - Synch: must wait for previous anti-diagonal values; less cost than for previous scheme

Option 2 - Anti-diagonals



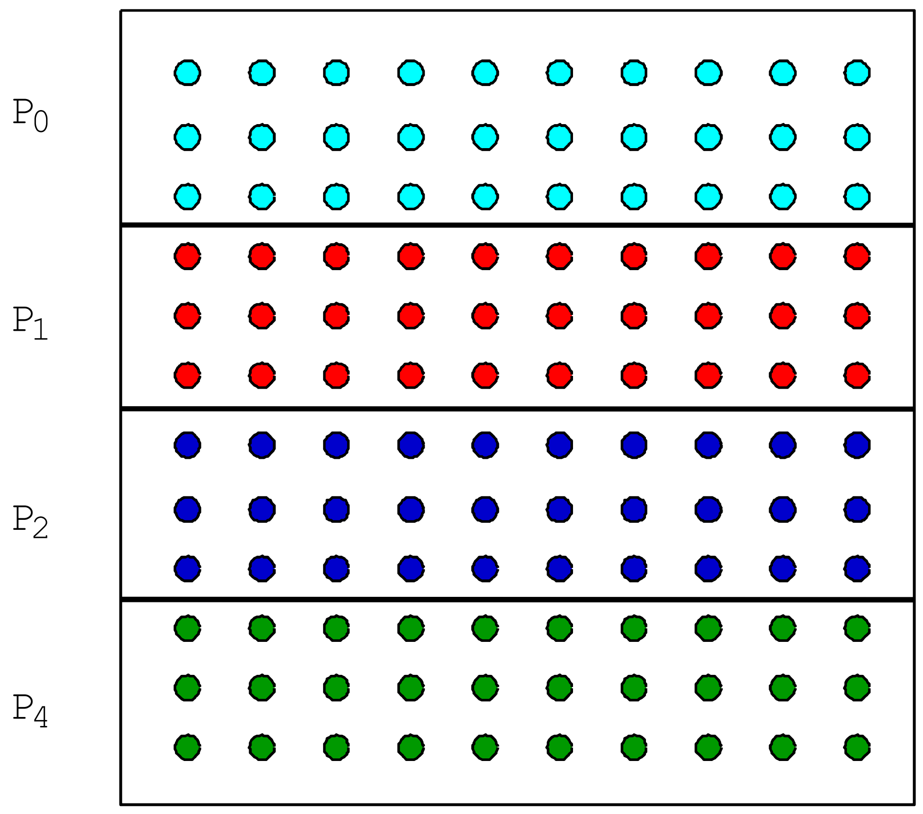
● Boundary point

● Interior point

Assignment Options

1. A concurrent task for each element update
   - Max concurrency: n**2
   - Synch: wait for left & top values
   - High synchronization cost
2. A concurrent task for each anti-diagonal
   - No dependence among elements in task
   - Max concurrency: ~ n
   - Synch: must wait for previous anti-diagonal values; less cost than for previous scheme
3. A concurrent task for each block of rows

Assignment -- Option 3



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.

SAS Version – Generating Processes

```
1.   int n, nprocs;        /* matrix: (n + 2-by-n + 2) elts.*/
2.   float **A, diff = 0;
2a.  LockDec (lock_diff);
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
```

SAS Version -- Solve

```
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 temp;
                    mybegin  = 1 + (n/nprocs)*pid;
                    myend = mybegin + (n/nprocs);
        while (!done) do    /*outermost loop over sweeps*/
           diff = 0;            /*initialize difference to 0*/
                 Barriers (barrier1, nprocs);
            for  i ← mybeg to myend do/*sweep for all 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
```

SAS Version -- Issues

- SPMD program
- Wait_for_end – all to one communication
- How is diff accessed among processes?
  - Mutex to ensure diff is updated correctly.
  - Single lock $\Rightarrow$ too much synchronization!
  - Need not synchronize for every grid point. Can do only once.
- What about access to A[i][j], especially the boundary rows between processes?
- Can loop termination be determined without any synch. among processes?
  - Do we need any statement for the termination condition statement

SAS Version -- Solve

```
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.              Barriers (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;
25a.           Barrier (barrier1, nprocs);
26.          end while
27.  end procedure
```

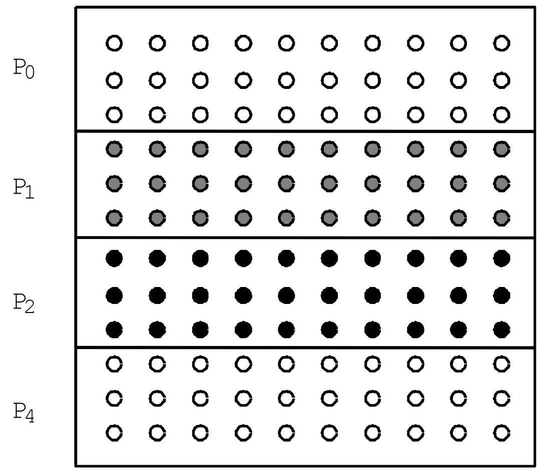
SAS Program

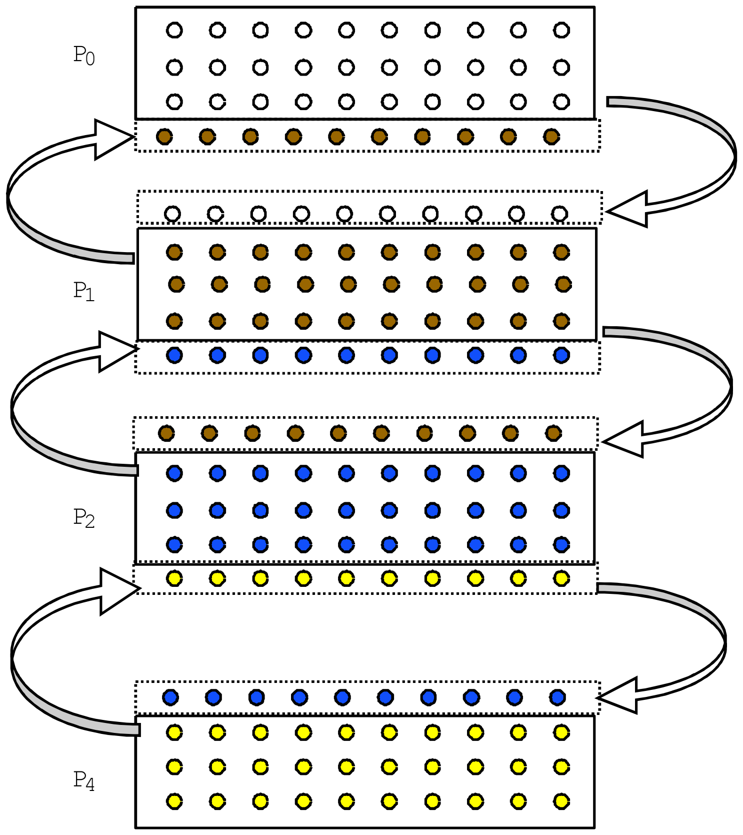
- **done** condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for all the barriers?
- Good global reduction?
  - Utility of this parallel accumulate??

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 SAS
- 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 – Generating Processes

```
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 – Array allocation and Ghost-row Copying

```
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.          myend = (n/nprocs) ;
6.            myA = malloc  (array of (n/nprocs) x n floats );
7.            initialize (myA);   /* initialize myA LOCALLY */
15.           while (!done) do     /*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);
```

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 (mydiff, 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(mydiff/(n*n) < TOL) then done = 1;
24j.                  for k ← 1 to nprocs-1 do
24k.                          SEND (done, sizeof(float), k   ,  DONE);
24l.                   endfor
25.         end while
26.   end procedure
```

Notes on Message Passing Version

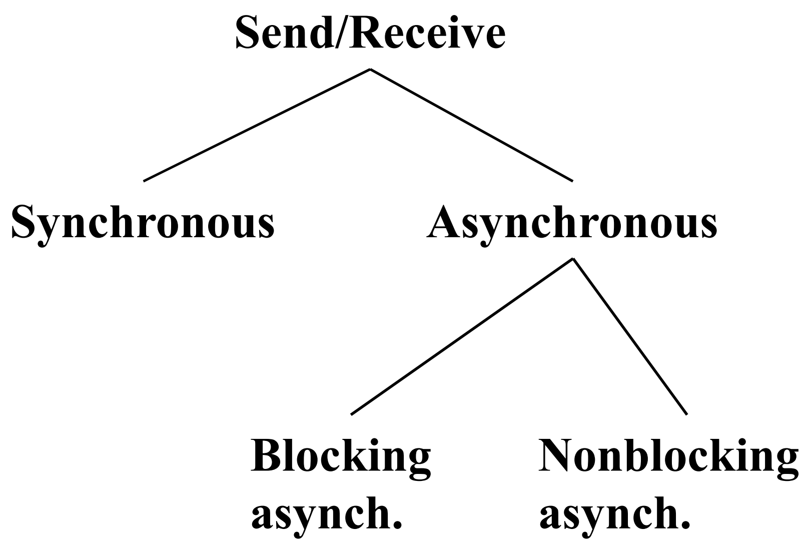
- Receive does not transfer data, send does
  - unlike SAS which is usually receiver-initiated (load fetches data)
- Can there be deadlock situation due to sends?
- Communication done at once in whole rows at beginning of iteration, not grid-point by grid-point
- Core similar, but indices/bounds in local rather than global space
- Synchronization through sends and receives
  - Update of global diff and event synch for done condition – mutual exclusion occurs naturally
- Can use REDUCE and BROADCAST library calls to simplify code

Notes on Message Passing Version

```
/*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
25j.            done = 1;
25k.        endif
25m.            BROADCAST(0,done,sizeof(int),DONE
```

Send and Receive Alternatives

- Semantic flavors: based on when control is returned
- Affect when data structures or buffers can be reused at either end
- Synchronous messages provide built-in synch. through match
- Separate event synchronization needed with asynch. Messages
- Now, deadlock can be avoided in our code.



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

Orchestration: Summary

- 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)

Grid Solver Program: Summary

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

Grid Solver Program: Summary

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