Application and Programming Overview*

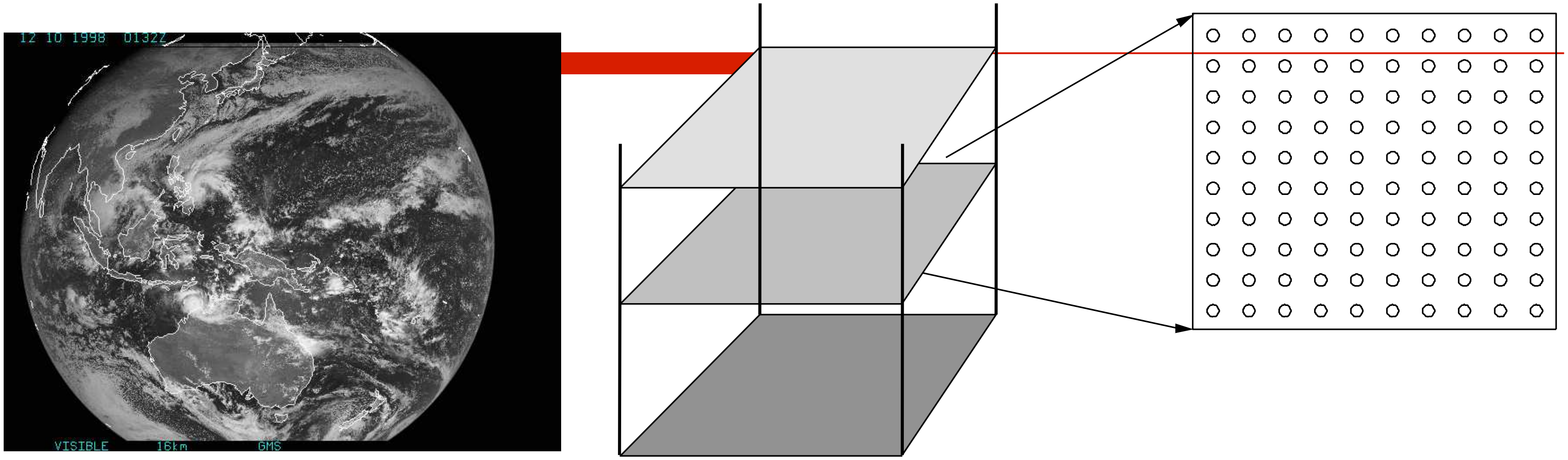
Zeljko Zilic
McConnell Engineering Building
Room 546

* Ch. 2 in the Culler-Singh textbook

Overview

- Motivating Problems (application case studies)
- Process of creating a parallel program
- What a simple parallel program looks like
  - Three major programming models
  - What primitives must a system support?
- *Later*: Performance issues and architectural interactions

Simulating Ocean Currents



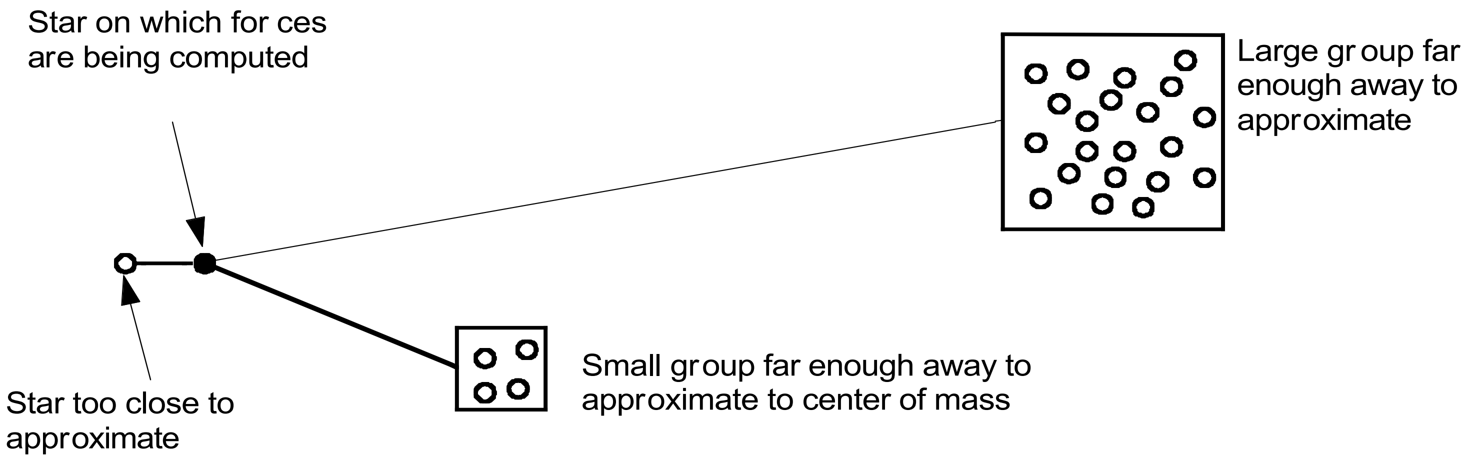
(a) Cross sections

(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
- Static and regular

Simulating Galaxy Evolution

- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
  - $O(n^2)$ brute force approach
  - Barnes-Hut: Hierarchical Method for gravtation: $G \frac{m_1 m_2}{r^2}$



Star on which forces are being computed

Large group far enough away to approximate

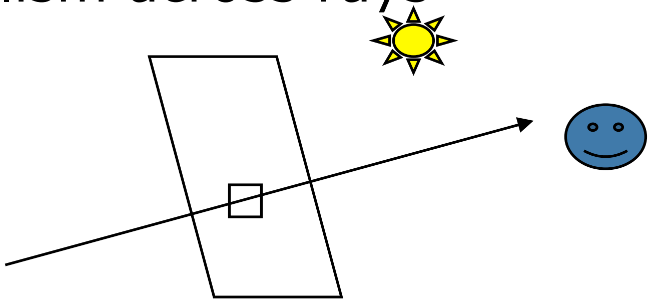
Star too close to approximate

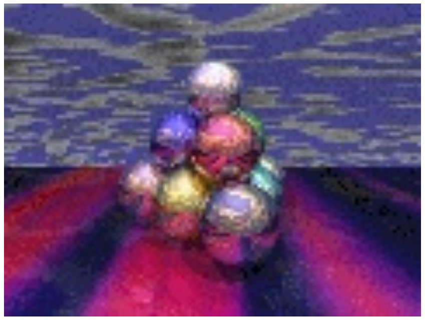
Small group far enough away to approximate to center of mass

•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





- How much concurrency in these examples?

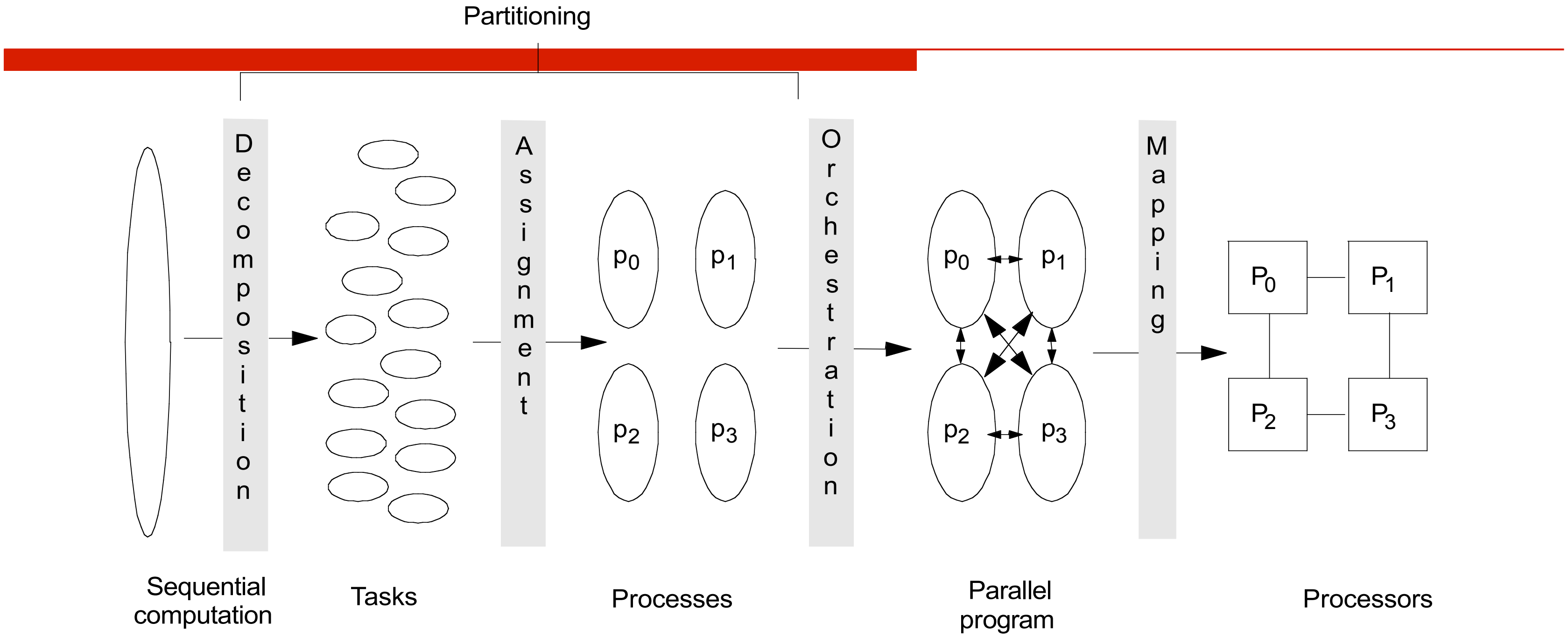
Creating a Parallel Program

- Pieces of the job:
  - 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

Definitions

- *Task*:
  - Arbitrary piece of work (fine- or coarse-grain)
  - Executed sequentially; concurrency is only across tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
- *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
    - Write program in terms of processes, then map to processors

Steps in Making Parallel Programs



- Decomposition of computation in tasks
- Assignment of tasks to processes
- Orchestration of data access, comm, synch.
- Mapping processes to processors

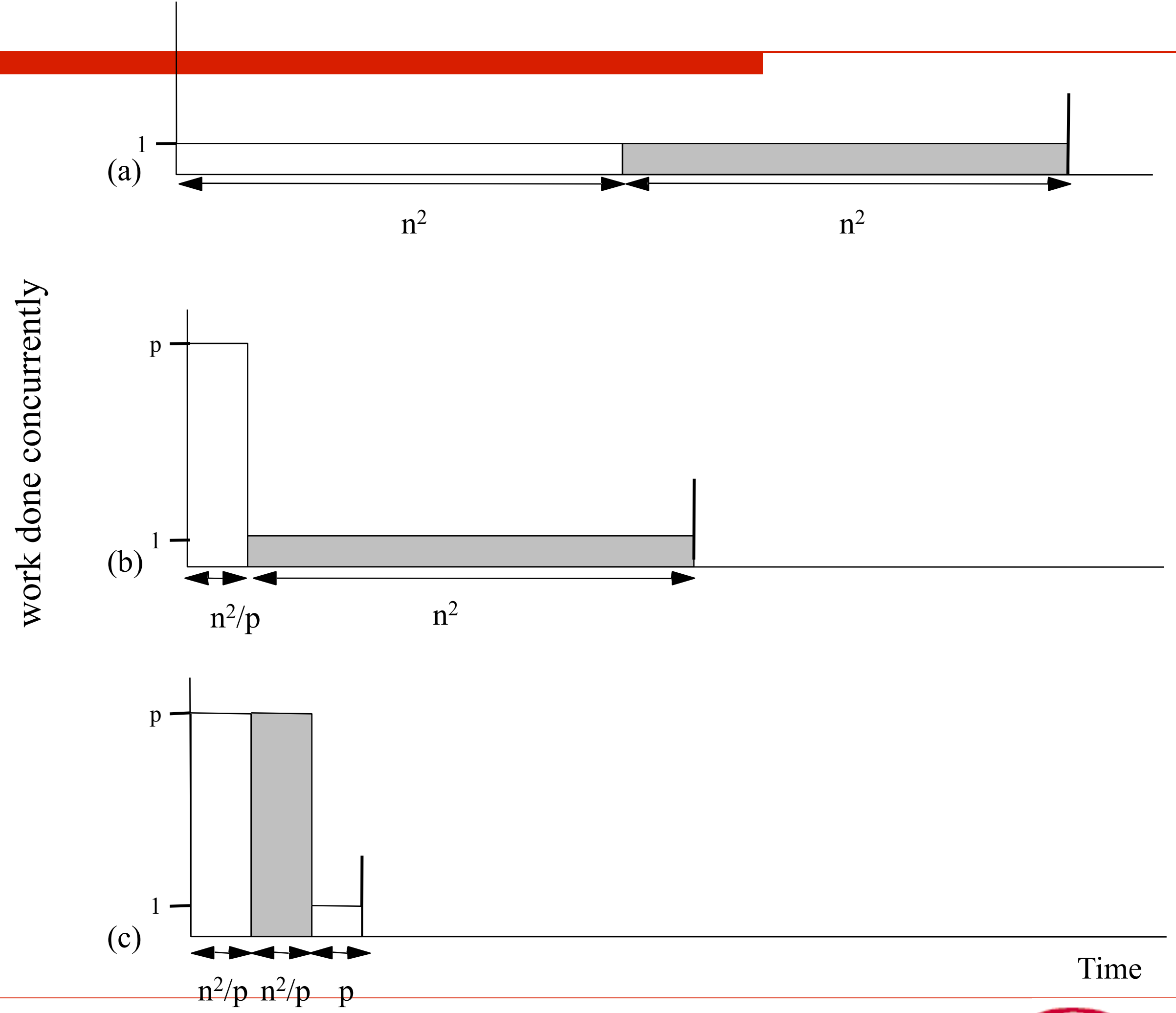
Decomposition

- Identify concurrency and decide level at which to exploit it
- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - No. of available tasks may vary with time
- Goal: Enough tasks to keep processes busy, but not too many
  - Number of tasks available at a time is upper bound on achievable speedup

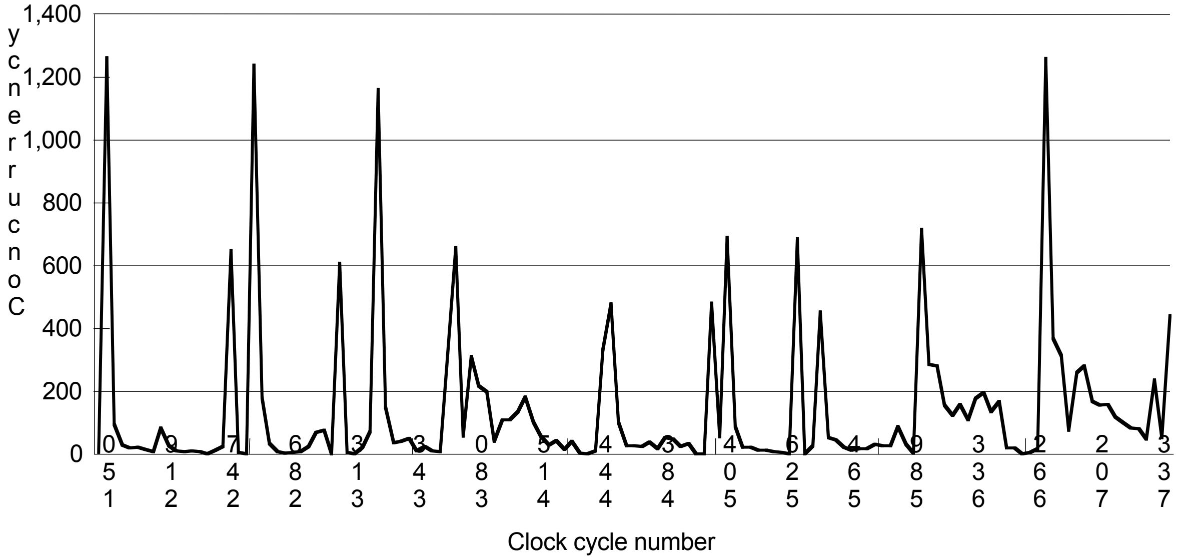
Working Around Amdahl's Law

- If fraction *s* of seq execution is inherently serial, speedup <= *1/s*
- Example: 2-phase calculation
  - sweep over *n*-by-*n* grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase = n^2/p
- Second phase serialized at global variable, so time = n^2
- Speedup <= $\dfrac{2n^2}{\frac{n^2}{p} + n^2}$ or at most 2
- Trick: divide second phase into two
  - Accumulate into private sum during sweep
  - Add per-process private sum into global sum
- Parallel time is $n^2/p + n^2/p + p$, and speedup at best $\dfrac{p*2n^2}{2n^2 + p^2}$

Visualizing Amdahl’s Law



Concurrency Profiles



- Area under curve is total work done (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: $\dfrac{1}{s + \dfrac{1-s}{p}}$

Assignment

- Specify mechanism to divide work up among processes
  - E.g. which process computes forces on which stars, or which rays
  - 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
- Programmers worry about partitioning first
  - *Usually* independent of architecture or prog. model
  - But cost and complexity of using primitives may affect decisions

Orchestration

  - Naming data
  - Structuring communication
  - Synchronization
  - Organizing data structures and scheduling tasks temporally

- Goals
  - Reduce cost of communication and synch.
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Function of Prog. Model, comm. abstraction, primitives
- Architects should provide appropriate primitives efficiently

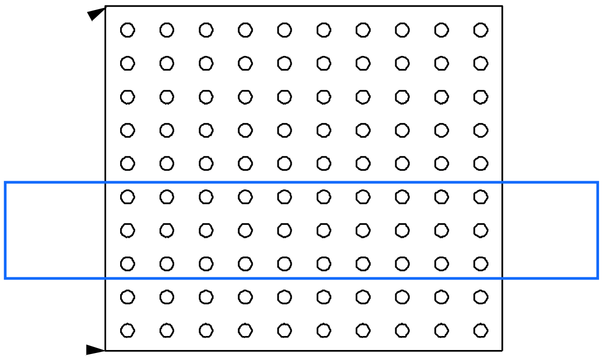
Mapping

- Two aspects:
  - Which process runs on which particular processor?
    - mapping to a network topology
  - 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 desires in some aspects, system handles some
- Usually adopt the view: process <-> processor

Parallelizing Computation vs. Data

- Computation partitioned (decomposed & assigned)

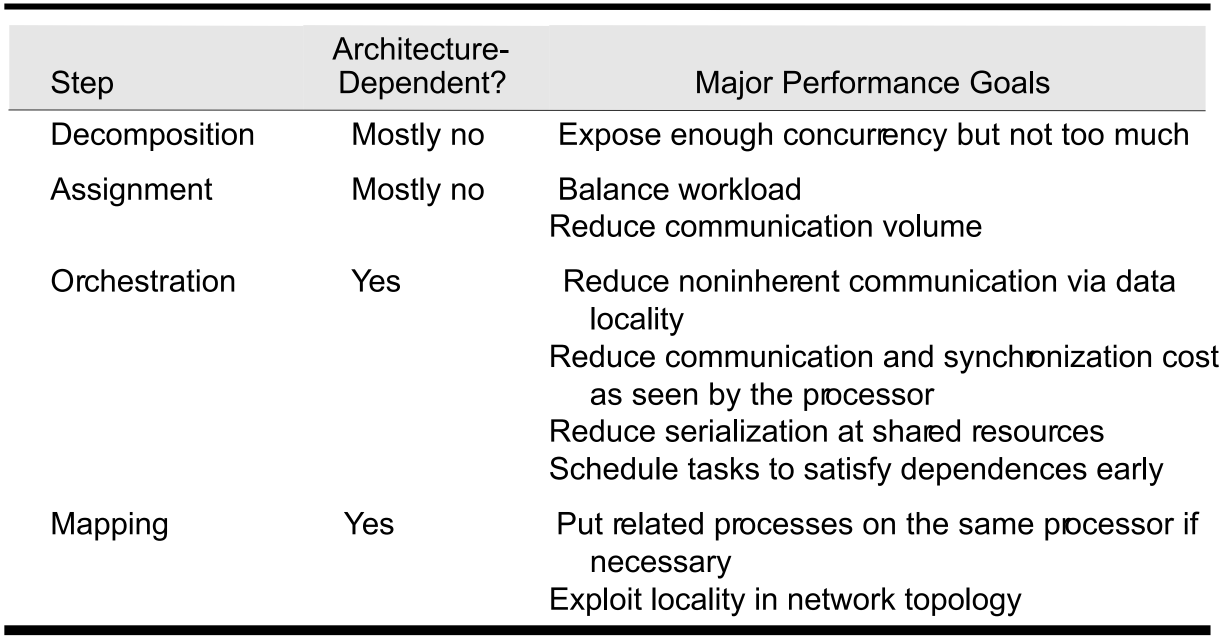
- Partitioning Data is often a natural view too
  - Computation follows data: *owner computes*
  - Grid example; data mining;
- Distinction between comp. and data stronger in many apps
  - Barnes-Hut
  - Raytrace



Architect’ s Perspective

- What can be addressed by hardware design?
- What is fundamentally a programming issue?

High-level Goals



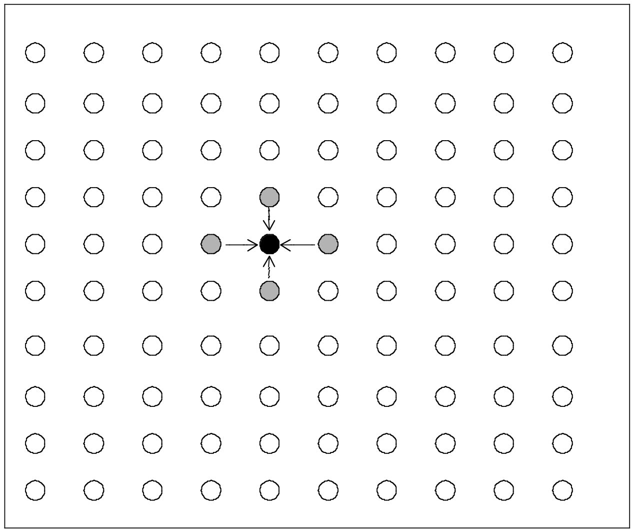
Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost 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

- High performance (speedup over sequential program)
- Low resource usage and development effort
- Implications for algorithm designers and architects?

What Parallel Programs Are Like?

Example: Iterative Equation Solver

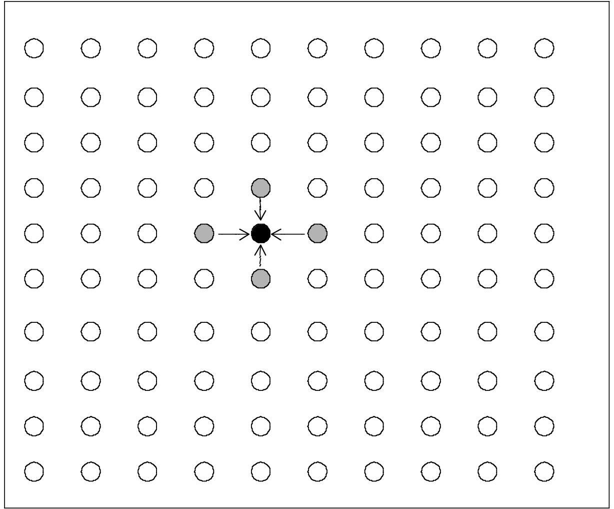
- Simplified version of a piece of Ocean simulation
- Illustrate program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives
  - State of most real parallel programming today



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

Grid Solver



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

- 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
  - Difference from previous value computed
  - Accumulate partial diffs into global diff at end of every sweep
  - Convergence check
    - Within a tolerance parameter

Sequential Version

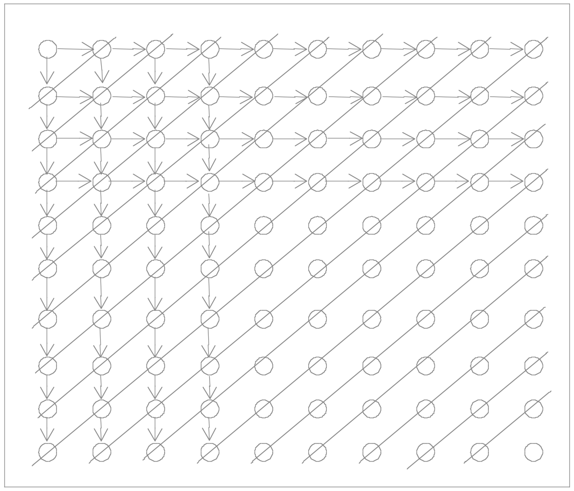
```
int n;                                      /*size of matrix: (n + 2-by-n + 2) elements*/
float **A, diff = 0;

main()
begin
  read(n) ;                                 /*read input parameter: matrix size*/
  A ← malloc (a 2-d array of 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

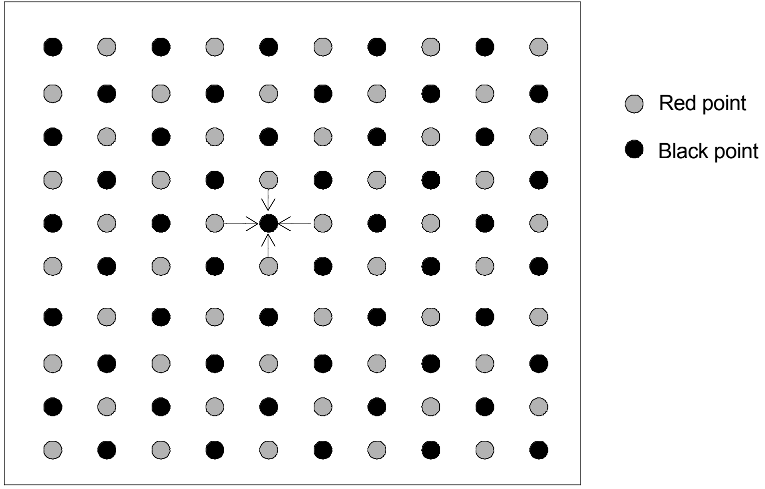
- Simple way to identify concurrency: look at loop iterations
  - *dependence analysis*; if not enough concurrency, look further
- Not much concurrency here at this level (all loops *sequential*)
- Study dependencies



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

Exploit Domain Knowledge

- Reorder grid traversal: red-black ordering



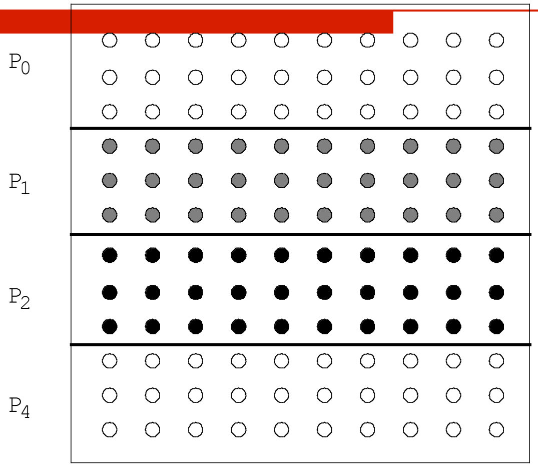
- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel:
- Global synch between them (conservative but convenient)
- Can use simpler, asynchronous one to illustrate
  - No red-black, simply ignore dependences within sweep
  - Program *nondeterministic*

Decomposition

```
while (!done) do                  /*a sequential loop*/
  diff = 0;
  for_all i ← 1 to n do              /*a parallel loop nest*/
    for_all j ← 1 to n do
      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]);
      diff += abs(A[i,j] - temp);
    end for_all
  end for_all
  if (diff/(n*n) < TOL) then done = 1;
end while
```

- Decomposition into elements: degree of concurrency n^2
- Decompose into rows? Degree ?
- `for_all` assignment ??

Assignment



- Static assignment: 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*, ...
- Dynamic assignment
  - Get a row index, work on the row, get a new row, ...
- What is the mechanism?
- Concurrency? Volume of Communication?

Data Parallel Solver

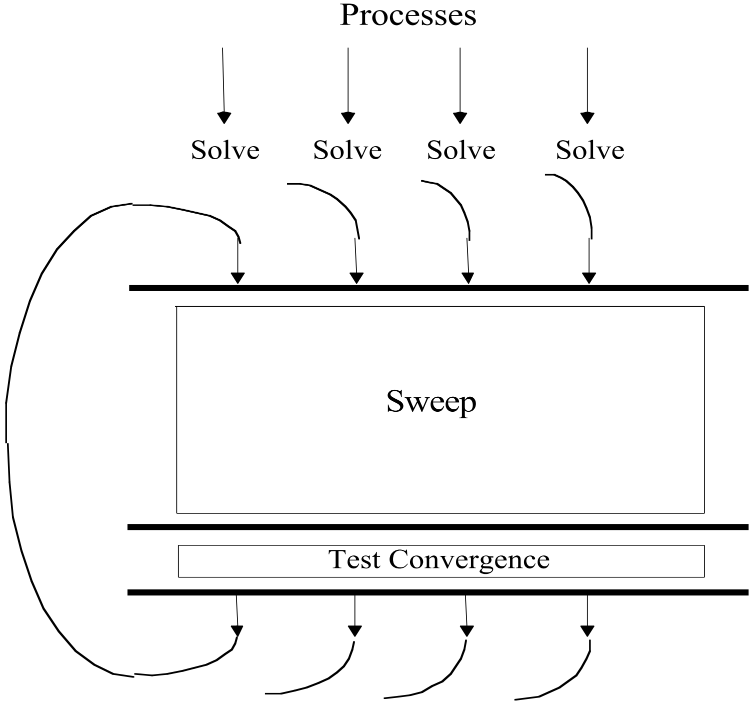
```
int n, nprocs;                               /*grid size (n + 2-by-n + 2) and number of processes*/
float **A, diff = 0;

main()
begin
  read(n); read(nprocs);          ;          /*read input grid size and number of processes*/
  A ← G_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 mydiff = 0, temp;
    DECOMP A[BLOCK,*, nprocs];
  while (!done) do                           /*outermost loop over sweeps*/
    mydiff = 0;                              /*initialize maximum difference to 0*/
    for_all i ← 1 to n do                    /*sweep over non-border points of grid*/
      for_all 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*/
        mydiff[i,j] = abs(A[i,j] - temp);
      end for_all
    end for_all
      REDUCE (mydiff, diff, ADD);
    if (diff/(n*n) < TOL) then done = 1;
  end while
end procedure
```

Shared Memory Solver

Single Program Multiple Data (SPMD)



- Assignment controlled by values of loop bound variables

Generating Threads

```
1.   int n, nprocs;          /*matrix dimension and number of processors to be used*/
2a.  float **A, diff;        /*A is global (shared) array representing the grid*/
                             /*diff is global (shared) maximum difference in current  sweep*/
2b.  LOCKDEC(diff_lock);     /*declaration of lock to enforce mutual exclusion*/
2c.  BARDEC (bar1);          /*barrier declaration for global synchronization between sweeps*/

3.  main()
4.  begin
5.      read(n); read(nprocs);    /*read input matrix size and number of processes*/
6.      A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.      initialize(A);            /*initialize A in an unspecified way*/
8a.     CREATE (nprocs–1, Solve, A);
8.      Solve(A);                 /*main process becomes a worker too*/
8b.       WAIT_FOR_END (nprocs–1);    /*wait for all child processes created to terminate*/
9.  end main

10.  procedure Solve(A)
11.     float **A;                    /*A is entire n+2-by-n+2 shared array,
                                      as in the sequential program*/
12. begin
13. ----
27. end procedure
```

Assignment Mechanism

```
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 sweeps*/
      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
```

SM Program

- SPMD: not lockstep. Not 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?
- Good global reduction?
  - Utility of this parallel accumulate???

Mutual Exclusion

- Why is it needed?

- Provided by LOCK-UNLOCK around *critical section*
  - Set of operations we want to execute atomically
  - Implementation of LOCK/UNLOCK must guarantee mutual excl.

- Serialization?
  - Contention?
  - Non-local accesses in critical section?
  - 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 (Not Used Here)

- One process notifies another of an 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;
a: while (flag is 0) do nothing;
                                              b: flag = 1;
   print A;
```

- *Busy-waiting* or *spinning*

Group Event Synchronization

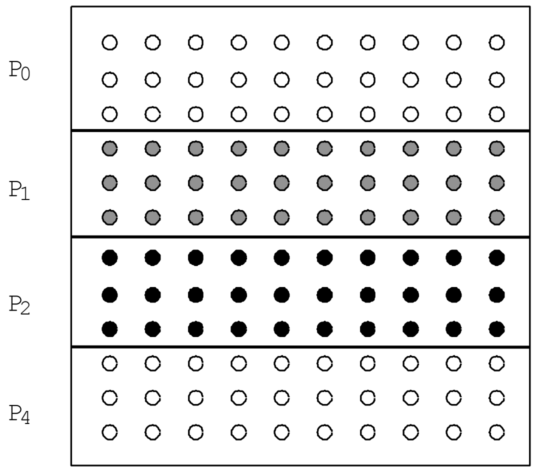
- Subset of processes involved
  - Can use flags or barriers (involving only the subset)
  - Concept of producers and consumers

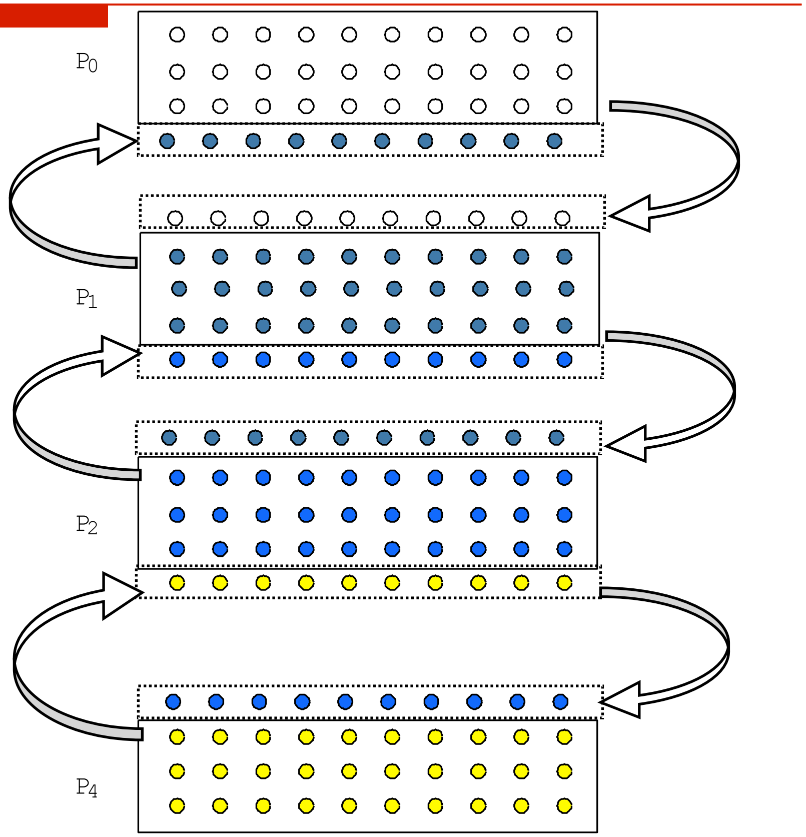
- Major types:
  - Single-producer, multiple-consumer
  - Multiple-producer, single-consumer
  - Multiple-producer, multiple-consumer

Message Passing Grid Solver

- Cannot declare A to be global shared array
  - 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 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

```
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);
                                            /*initialize my rows of A, in an unspecified way*/

while (!done) do
   mydiff = 0;                                 /*set local diff to 0*/
/* Exchange border rows of neighbors into myA[0,*] and myA[n'+1,*]*/
   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);
   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
```

Note 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*/
REDUCE(0,mydiff,sizeof(float),ADD);
if (pid == 0) then
   if (mydiff/(n*n) < TOL) then done = 1;
endif
   BROADCAST(0,done,sizeof(int),DONE);
```

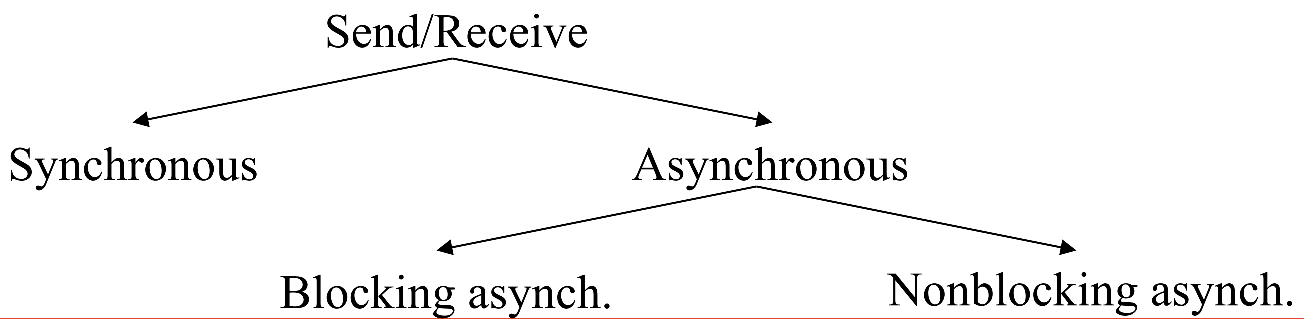
Orchestration: Summary

- 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 communication

- Message passing
  - Data distribution among local address spaces needed
  - No explicit shared structures (implicit in comm. patterns)
  - Communication is explicit
  - Synchronization implicit in comm. (at least in synch. case)
    - mutual exclusion by fiat

Send and Receive Alternatives

- Extended functionality: stride, scatter-gather, groups
- Sychronization semantics
  - Affect when data structures or buffers can be reused at either end
  - Affect event synch (mutual excl. by fiat: only 1 process uses data)
  - Affect ease of programming and performance
- Synchronous messages provide built-in synch. through match
  - Separate event synchronization may be needed with asynch. msgs
- With synch. messages, our code may hang. Fix?



Send/Receive

Synchronous

Asynchronous

Blocking asynch.

Nonblocking asynch.

Correctness in Grid Solver

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

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