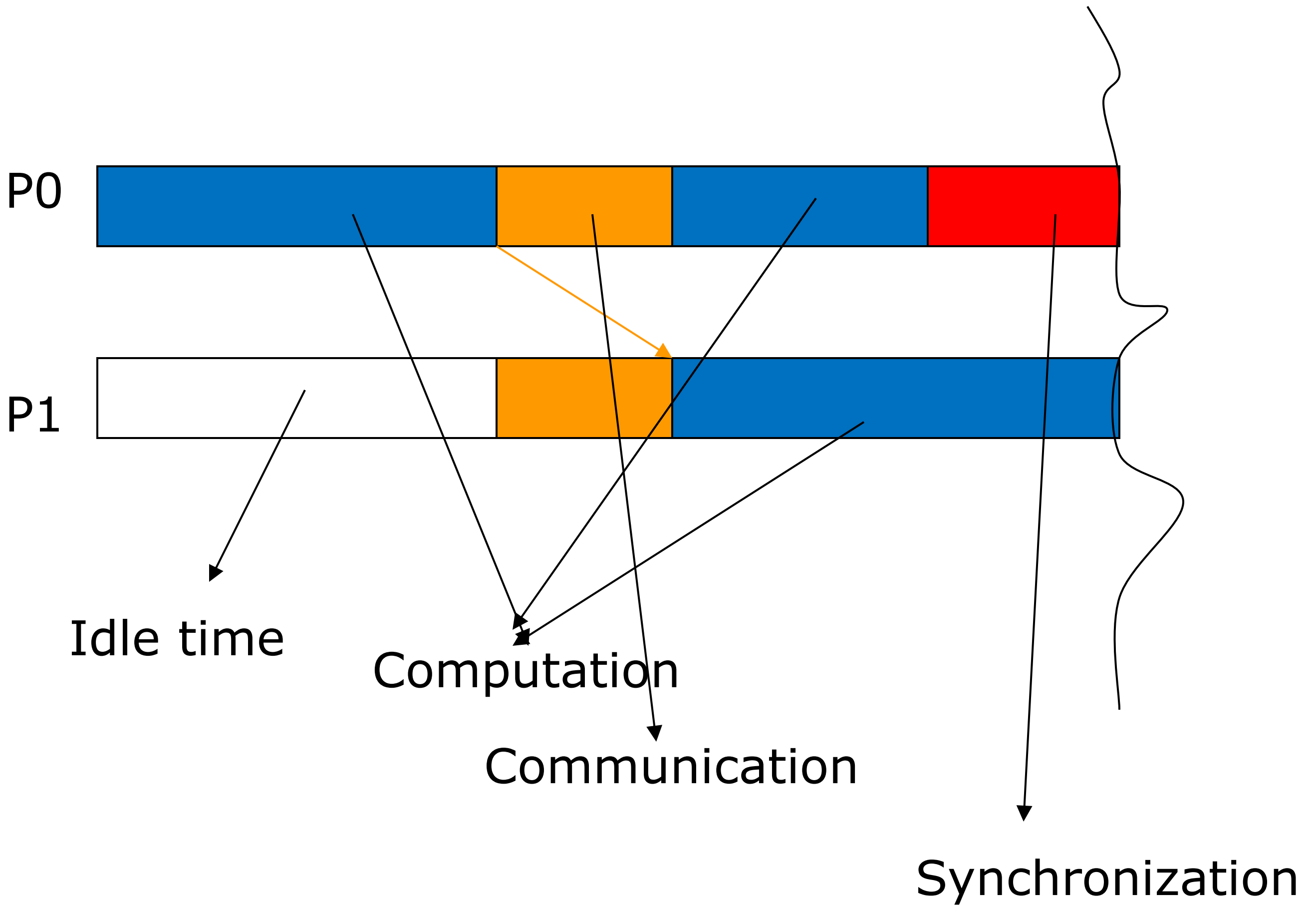
Parallelization Principles

Sathish Vadhiyar

Parallel Programming and Challenges

- Recall the advantages and motivation of parallelism
- But parallel programs incur overheads not seen in sequential programs
  - Communication delay
  - Idling
  - Synchronization

Challenges

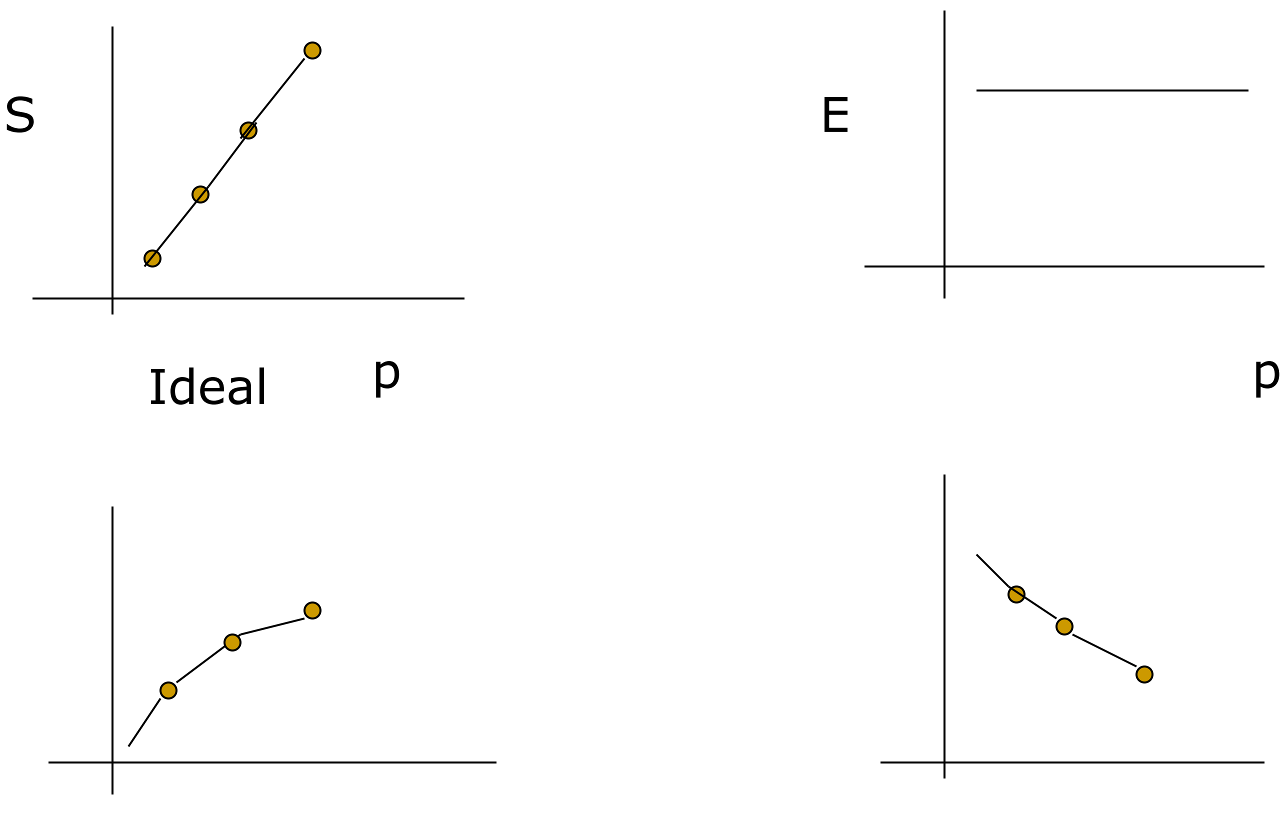


P0
P1
Idle time
Computation
Communication
Synchronization

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.
- $\text{Speedup} = \dfrac{1}{(f_s + (f_p/P))}$

Gustafson’s Law

- Increase problem size proportionally so as to keep the overall time constant
- The scaling keeping the problem size constant (Amdahl’s law) is called **strong scaling**
- The scaling due to increasing problem size is called **weak scaling**

Scalability and Isoefficiency

- 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 computations performed must scale with P to keep E constant
- This function of computation in terms of P is called isoefficiency function.

Example: ScaLAPACK PDGESV

- $$T_{par}(N,P) = \frac{2}{3}\frac{N^3}{P}t_f + \frac{(3+1/4log_2P)N^2}{\sqrt{P}}t_v + (6+log_2P)Nt_m$$
- $$T_{seq}(N) = \frac{2}{3}N^3t_f$$
- $$E = \frac{T_{seq}(N)}{P.T_{par}(N,P)} = (1+\frac{3}{2}\frac{\sqrt(P)(3+1/4log_2P)}{N}\frac{t_v}{t_f} + \frac{3}{2}\frac{P(6+log_2P)}{N^2}\frac{t_m}{t_f})^{-1}$$
- As P is increased, N should be increased by approx. $O(\sqrt(P))$
- As amount of computations is $O(N^3)$, the isoefficiency function is $O(P\sqrt(P))$

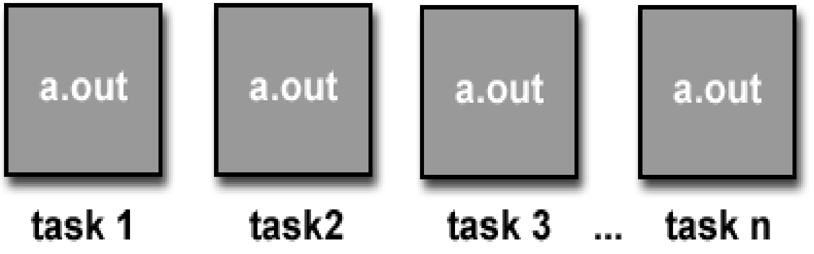
Isoefficiency

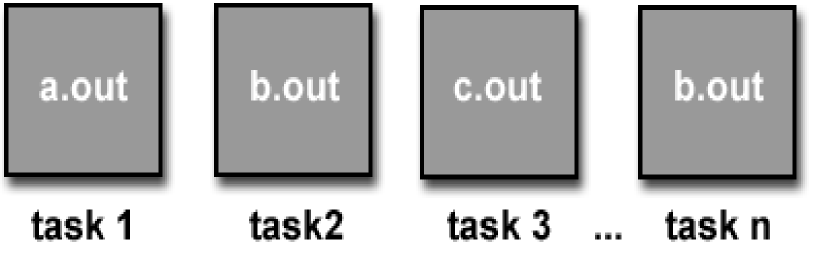
- Smaller isoefficiency functions imply higher scalability
- Consider two parallel algorithms with isoefficiency functions W1=O(P) and W2=O(root-P)
- The second algorithm is considered to be more scalable since only small amount of work needs to be added
- Similarly, an algorithm with an isoefficiency function of O(P) is highly scalable while an algorithm with quadratic or exponential isoefficiency function is poorly 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

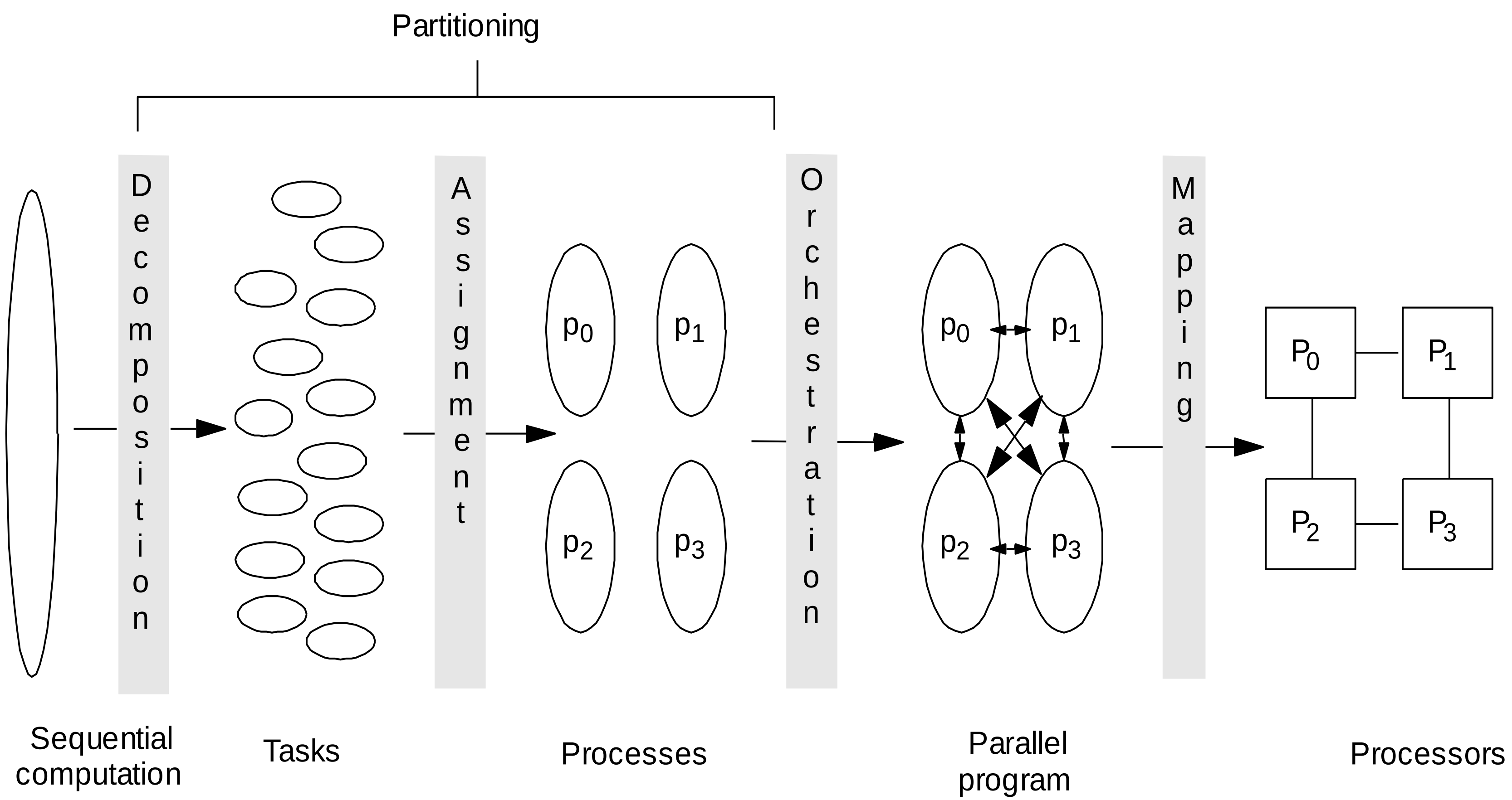
3. Orchestration

   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
- In practical cases, both steps combined into one step, trying to answer the question "What is the role of each parallel processing entity?"

Data Parallelism and Domain Decomposition

- Given data divided across the processing entitites
- Each process owns and computes a portion of the data – owner-computes rule
- Multi-dimensional domain in simulations divided into subdomains equal to processing entities
- This is called **domain decomposition**

Domain decomposition and Process Grids

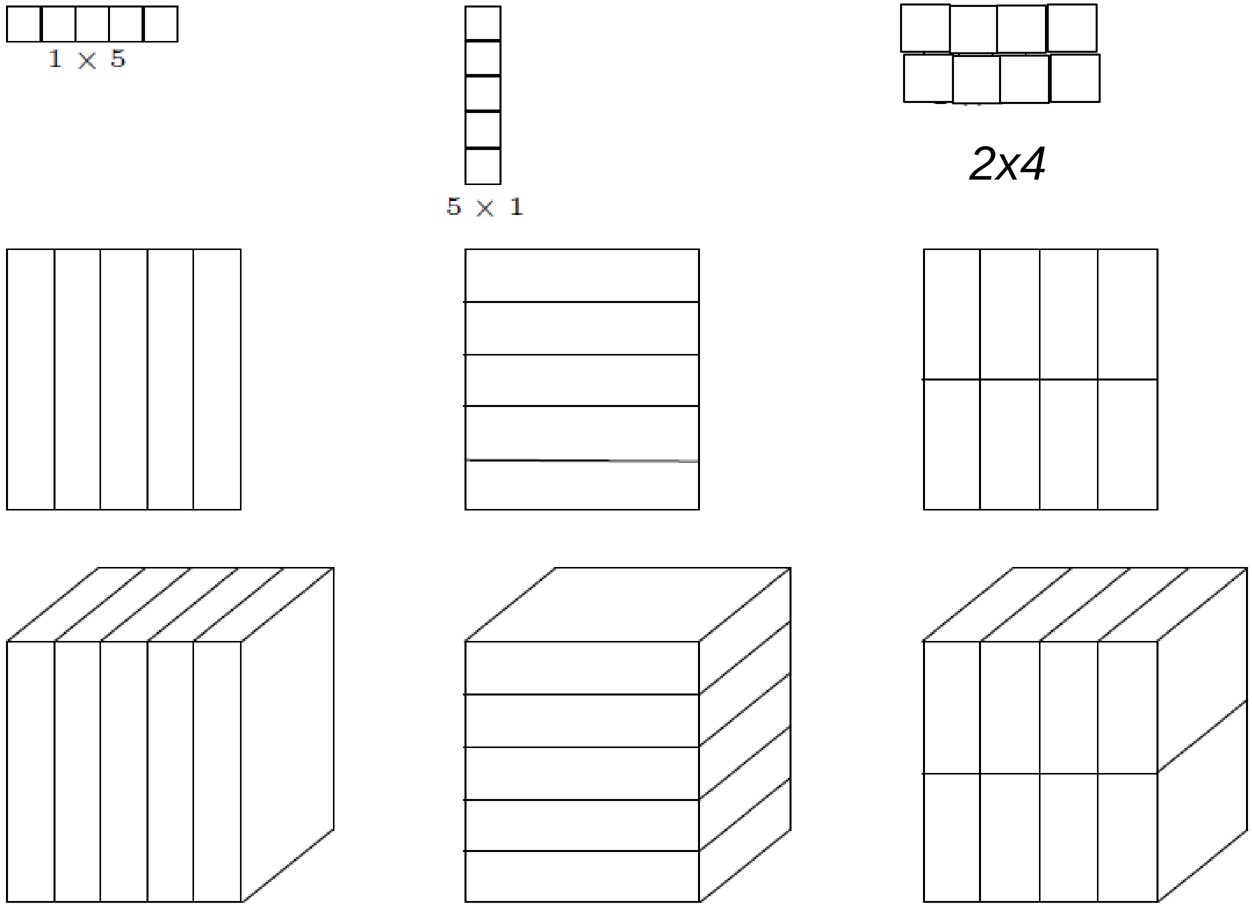
- Process grid used to specify domain decomposition
- The given P processes arranged in multi-dimensions forming a **process grid**
- The domain of the problem divided into process grid

Illustrations

Process grid

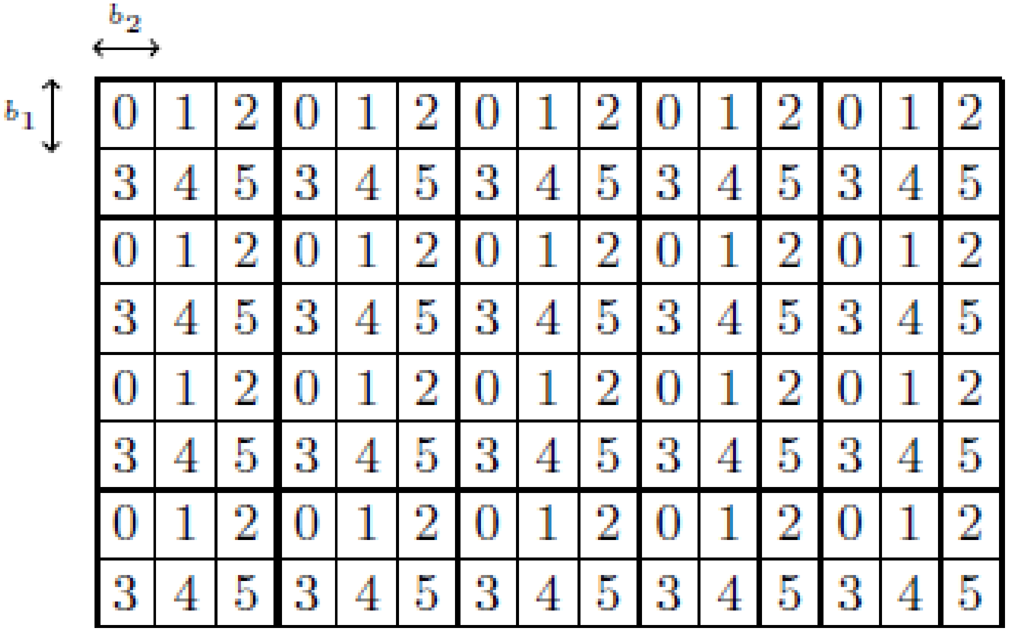
2-D domain decomposed using the process grid

3-D domain decomposed using the process grid



Data Distributions

- For dividing the data in a dimension using the processes in a dimension, **data distribution** schemes are followed
- Common data distributions:
  - Block: for regular computations
  - Block-cyclic: when there is load imbalance across space

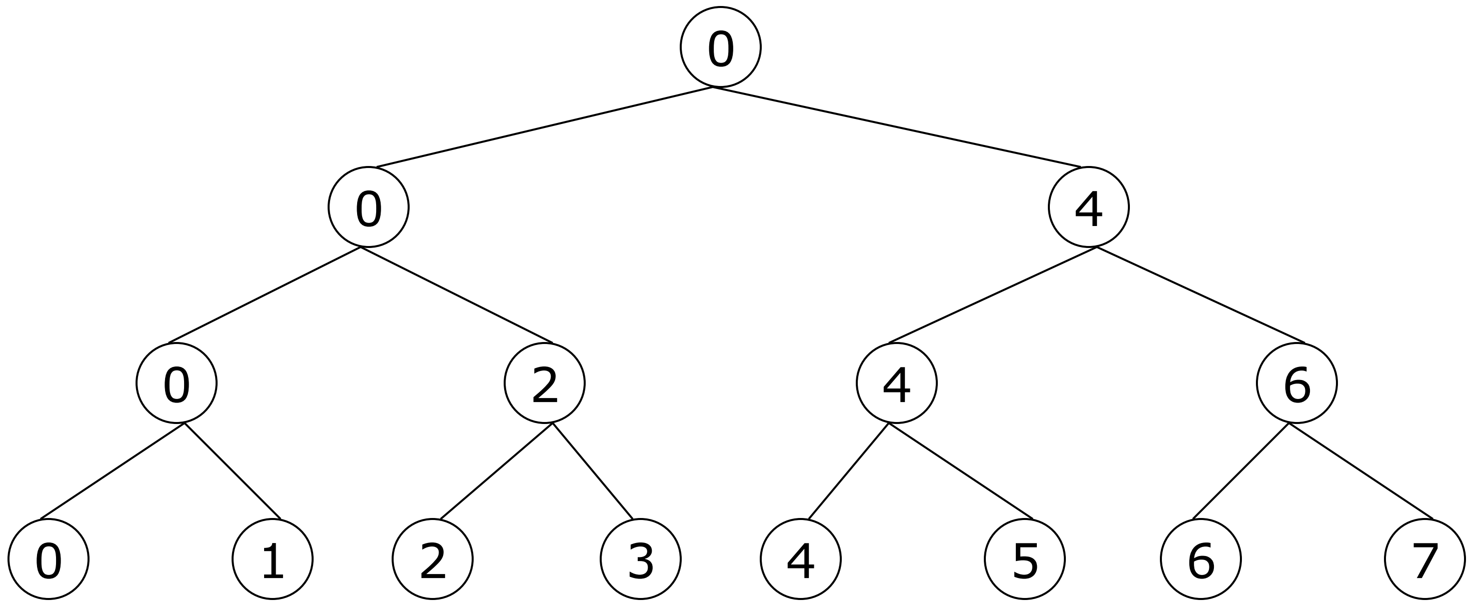


Task parallelism

- Independent tasks identified
- The task may or may not process different data
- The tasks are grouped by a process called **mapping**
- Two objectives:
  - Balance the groups
  - Minimize inter-group dependencies
- Represented as **task graph**
- Mapping problem is NP-hard

Based on Task Partitioning

- Based on **task dependency graph**



- In general the problem is NP complete

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

- All data and task parallel strategies follow static 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

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

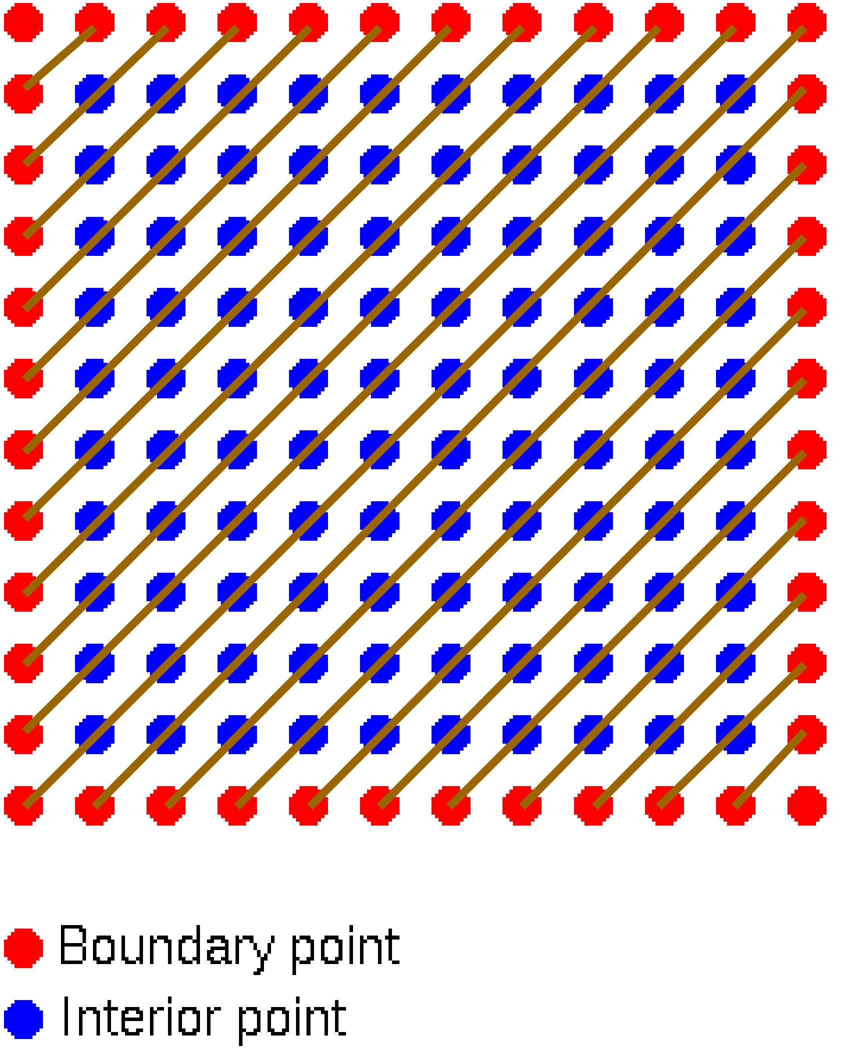
```
do {
    diff = 0.0
    for (i=0; i < n; i++)
        for (j=0; j < n, j++){
            temp = A[i] [j];
            A[i][j] = average (neighbours);
            diff += abs (A[i][j] – temp);
        }
    while  (diff > tolerance) ;
```

	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

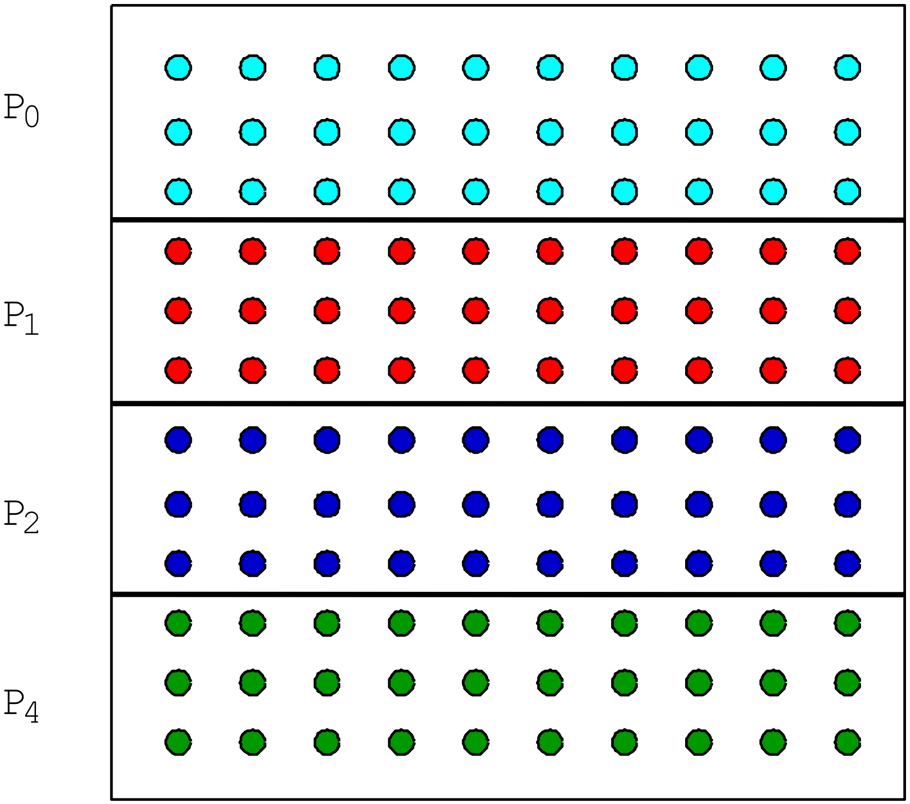
Option 2 - Anti-diagonals



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

```
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*/
             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*/
                mydiff += abs(A[i,j] - temp);
             end for
           end for
           lock (diff-lock);
                   diff += mydiff;
           unlock (diff-lock)
           barrier (barrier1, nprocs);
           if (diff/(n*n) < TOL) then done = 1;
           Barrier (barrier1, nprocs);
        end while
end procedure
```

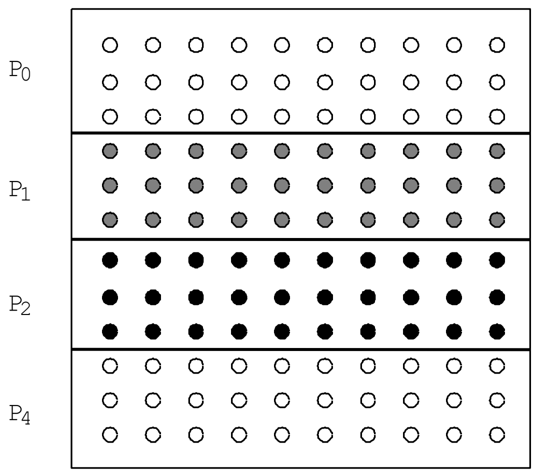
SAS Program

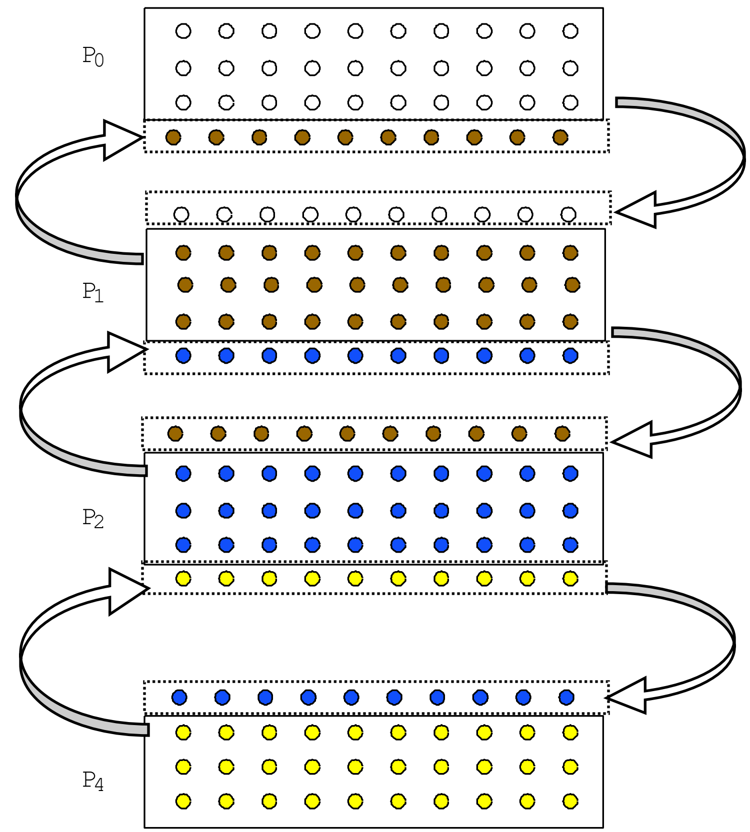
- **done** condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for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

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

Notes on Message Passing Version

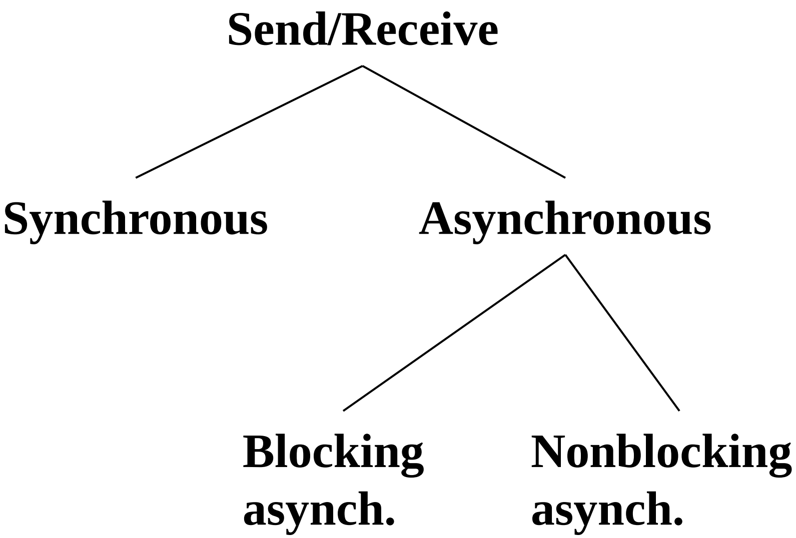
- Receive does not transfer data, send does
  - unlike SAS which is usually receiver-initiated (load fetches data)
- Can there be deadlock situation due to sends?
- Communication done at once in whole rows at beginning of iteration, not grid-point by grid-point
- Core similar, but indices/bounds in local 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