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

Vs. implicitly by parallelizing compiler

(PCA Chapter 2.3)
Example Motivating Problem: Simulating Ocean Currents/Heat Transfer ...

- Model as two-dimensional “n x n” grids
- Discretize in space and time
  - finer spatial and temporal resolution \(\Rightarrow\) greater accuracy
- Many different computations per time step, \(O(n^2)\) per grid.
  - set up and solve linear equations iteratively (Gauss-Seidel).
- Concurrency across and within grid computations per iteration
  - \(n^2\) parallel computations per grid \(\times\) number of grids

Expression for updating each interior point:
\[
\]

Maximum Degree of Parallelism (DOP) or concurrency: \(O(n^2)\) data parallel computations per grid per iteration

When one task updates/computes one grid element

More reading: PP Chapter 11.3 (Pages 352-364)

From last lecture

(PCA Chapter 2.3)
Solution of Linear System of Equation By Synchronous Iteration

*Iterations are sequential – Parallelism within an iteration $O(n^2)$*

Initialize all points

**Setup**

Iterate over (update) all interior $n^2$ points $O(n^2)$

Expression for updating each interior point:


Find Error (Global Difference)

Is Error < Tolerance Limit ? (Threshold)

Converged?

For one 2D nxn grid

One Iteration Or Sweep

Or maximum number of iterations is reached

Yes

Done

No

Iterate Again
Parallelization of An Example Program

Examine a simplified version of a piece of Ocean simulation

- Iterative (Gauss-Seidel) linear equation solver

One 2D Grid, \( n \times n = n^2 \) points (instead of 3D – \( n \) grids)

Illustrate parallel program in low-level parallel language:

- C-like pseudo-code with **simple extensions for parallelism**
- Expose basic communication and synchronization primitives that must be supported by parallel programming model.

Three parallel programming models targeted for orchestration:

- Data Parallel
- Shared Address Space (SAS)
- Message Passing

(PCA Chapter 2.3)
2D Grid Solver Example

- Simplified version of solver in Ocean simulation
- Gauss-Seidel (near-neighbor) sweeps (iterations) to convergence:
  - Interior n-by-n points of (n+2)-by-(n+2) updated in each sweep (iteration)
  - Updates done in-place in grid, and difference from previous value is computed
  - Accumulate partial differences into a global difference at the end of every sweep or iteration
  - Check if error (global difference) has converged (to within a tolerance parameter)
    - If so, exit solver; if not, do another sweep (iteration)
    - Or iterate for a set maximum number of iterations.

Expression for updating each interior point:

Computation = \( O(n^2) \) per sweep or iteration

\[ n^2 = n \times n \]

interior grid points
Pseudocode, Sequential Equation Solver

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

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

10. procedure Solve (A) /*solve the equation system*/
11. float **A;
12. begin
13. int i, j, done = 0;
14. float diff = 0, temp;
15. while (!done) do /*outermost loop over sweeps*/
16.   diff = 0; /*initialize maximum difference to 0*/
17.   for i ← 1 to n do /*sweep over nonborder points of grid*/
18.     for j ← 1 to n do /*save old value of element*/
19.       temp = A[i,j];
21.     end for
22.     end for
23.     diff += abs(A[i,j] - temp);
24.   end for
25.   if (diff/(n*n) < TOL) then done = 1; /*sweep over nonborder points of grid*/
26. end while
27. end procedure

Setup
- Initialize grid points
- Call equation solver

Iterate until convergence
- i.e one iteration
- Sweep O(n^2) computations

Update Points
- Global Difference
- Old value
- Done?

TOL, tolerance or threshold
**Decomposition**

- Simple way to identify concurrency is to look at loop iterations.
- Dependency analysis; if not enough concurrency is found, then look further into application.

1. Not much concurrency here at this level (all loops sequential).
2. Examine fundamental dependencies, ignoring loop structure.

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

Expression for updating each interior point:

Exploit Application Knowledge

- Reorder grid traversal: red-black ordering

\[ i \quad j \]

\[ \text{Red point} \quad \text{Black point} \]

- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel:
- Global synchronization between them (conservative but convenient)
- Ocean uses red-black; here we use simpler, asynchronous one to illustrate

- No red-black sweeps, simply ignore dependencies within a single sweep (iteration) all points can be updated in parallel \( \text{DOP} = \frac{n^2}{2} = O(n^2) \)

Maximum Degree of parallelism = \( \text{DOP} = O(n^2) \)
Type of parallelism: Data parallelism

One point update per task (\( n^2 \) parallel tasks)
- Computation = 1
- Communication = 4
- Communication-to-Computation ratio = 4

For PRAM with \( O(n^2) \) processors:
- Sweep = \( O(1) \)
- Global Difference = \( O(\log_2 n^2) \)
Thus: \( T = O(\log_2 n^2) \)

- Iterations may converge slower than red-black ordering
- Sequential order same as original.

Two parallel sweeps
Each with parallel \( \frac{n^2}{2} \) points updates

i.e. Max Software \( \text{DOP} = n^2 = O(n^2) \)
Decomposition Only

- **Decomposition into elements:** degree of concurrency $n^2$
- **To decompose into rows**, make line 18 loop sequential; degree of parallelism (DOP) = $n$
- `for_all` leaves assignment left to system
  - but implicit global synch. at end of `for_all` loop

The “`for_all`” loop construct imply parallel loop computations

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];
21.    diff += abs(A[i,j] - temp);
22. end for_all
23. end for_all
24. if (diff/(n*n) < TOL) then done = 1;
25. end while

Parallel PRAM $O(1)$
O($n^2$)
Parallel Computations (tasks)

Global Difference PRAM $O(\log_2 n^2)$

Degree of Parallelism (DOP)

- **Fine Grain:** $n^2$ parallel tasks each updates one element
  - DOP = $O(n^2)$
- **Coarser Grain:** $n$ parallel tasks each update a row
  - Task = grid row
  - Computation = $O(n)$
  - Communication = $O(n) \sim 2n$
  - Communication to Computation ratio = $O(1) \sim 2$
Assignment: (Update n/p rows per task)

i.e Task Assignment

\[ p = \text{number of processes or processors} \]

\[ \text{i.e } n^2/p \text{ rows per task} \]

- **Static assignments** (given decomposition into rows)
  - Block assignment of rows: Row \( i \) is assigned to process \( \frac{i}{p} \)
  - Cyclic assignment of rows: process \( i \) is assigned rows \( i, i+p, \) and so on

\[ p = \text{number of processors} < n \]
\[ p \text{ tasks or processes} \]

Task = updates \( n/p \) rows = \( n^2/p \) elements

Computation = \( O(n^2/p) \)

Communication = \( O(n) \) ~ \( 2n \) (2 rows)

Communication-to-Computation ratio = \( O \left( \frac{n}{(n^2/p)} \right) = O(p/n) \)

Lower C-to-C ratio is better

- **Dynamic assignment (at runtime):**
  - Get a row index, work on the row, get a new row, and so on

- Static assignment into rows reduces concurrency (from \( n^2 \) to \( p \))
  - concurrency (DOP) = \( n \) for one row per task \( C\text{-to-C} = O(1) \)
  - Block assign. reduces communication by keeping adjacent rows together

- Let’s examine **orchestration** under three programming models:
  - 1- Data Parallel
  - 2- Shared Address Space (SAS)
  - 3- Message Passing

\[ \text{Why Block Assignment} \]

p tasks

Instead of \( n^2 \)
Data Parallel Solver

1. int n, nprocs;
2. float **A, diff = 0;

3. main()
4. begin
5. read(n); read(nprocs);
6. A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7. initialize(A);
8. Solve(A);
9. end main

10. procedure Solve(A)
11. float **A;
12. begin
13. int i, j, done = 0;
14. float mydiff = 0, temp;
14a. DECOMP A[BLOCK, *, nprocs];
15. while (!done) do
16. mydiff = 0;
17. for_all i ← 1 to n do
18. for_all j ← 1 to n do
19. temp = A[i,j];
22. mydiff += abs(A[i,j] - temp);
23. end for all
24. end for all
24a. REDUCE (mydiff, diff, ADD);
25. if (diff/(n*n) < TOL) then done = 1;
26. end while
27. end procedure

\[ O(n^2/p + \log_2 p) \leq T(\text{iteration}) \leq O(n^2/p + p) \]
Shared Address Space Solver

Single Program Multiple Data (SPMD) Still MIMD

- Assignment controlled by values of variables used as loop bounds and individual process ID (PID)

As shown next slide
Pseudo-code, Parallel Equation Solver for Shared Address Space (SAS)

```
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*/
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. int i,j, pid, done = 0;
14. float temp, mydiff = 0;
14a. int mymin = 1 + (pid * n/nprocs); /*assume that n is exactly divisible by*/
14b. int mymax = mymin + n/nprocs - 1 /*nprocs for simplicity here*/
15. while (!done) do /*outer loop over all diagonal elements*/
16. mydiff = diff = 0; /*set global diff to 0 (okay for all to do it*/
16a. BARRIER(bar1, nprocs); /*ensure all reach here before anyone modifies diff*/
17. for i ← mymin to mymax do /*for each of my rows*/
18. for j ← 1 to n do /*for all nonborder elements in that row*/
19. temp = A[i,j];
21. mydiff += abs(A[i,j] - temp);
22. endfor
23. endwhile
24. end procedure
```

# of processors = p = nprocs
pid = process ID, 0 …. P-1

Array “A” is shared (all grid points)

Loop Bounds/Which Rows?
mymin = low row
mymax = high row

Private Variables

Sweep:
T = O(n^2/p)

T(p) = O(n^2/p + p)

Serialized update of global difference

Check/test convergence:
all processes do it

Barrier 1 → (Start sweep)
Barrier 2 → (sweep done)
Barrier 3 →

Global Difference

Mutual Exclusion (lock) for global difference

Critical Section: global difference

Done?
Notes on SAS Program

- **SPMD: not lockstep** (i.e. still MIMD not SIMD) or even necessarily same instructions.

- **Assignment controlled by values of variables used as loop bounds and process ID (pid)** (i.e. mymin, mymax)
  - Unique pid per process, used to control assignment of blocks of rows to processes.

- **Done condition (convergence test) evaluated redundantly by all processes**

- Code that does the update identical to sequential program
  - Each process has private mydiff variable

  **But**
  - Why?

  - Most interesting special operations needed are for synchronization
    - Accumulations of local differences (mydiff) into shared global difference have to be **mutually exclusive**
    - **Why the need for all the barriers?**

  Otherwise each process must enter the shared global difference critical section \( \frac{n^2}{p} \) times (\( n^2 \) times total) instead of just \( p \) times per iteration for all processes

  Using `LOCK()` .... `UNLOCK()`

SPMD = Single Program Multiple Data
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
  r1 ← diff
  r1 ← r1+r2
  diff ← r1

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

  {P1 gets 0 in its r1}
  {P2 also gets 0}
  {P1 sets its r1 to 1}
  {P2 sets its r1 to 1}
  {P1 sets cell_cost to 1}
  {P2 also sets cell_cost to 1}
  \[ r2 = mydiff = \text{Local Difference} \]

• Need the sets of operations to be \text{atomic (mutually exclusive)}

Fix?
Mutual Exclusion

Provided by LOCK-UNLOCK around critical section

- Set of operations we want to execute atomically
- Implementation of LOCK/UNLOCK must guarantee mutual exclusion. However, no order guarantee

Can lead to significant serialization if contended (many tasks want to enter critical section at the same time)

- Especially costly since many accesses in critical section are non-local
- Another reason to use private mydiff for partial accumulation:
  - Reduce the number times needed to enter critical section by each process to update global difference:
    - Once per iteration vs. $\frac{n^2}{p}$ times per process without mydiff
      - i.e $O(n^2)$ total number of accesses to critical section by all processes

No order guarantee provided

i.e one task at a time in critical section

Lock — Enter — Critical Section — Exit — Unlock
Global (or group) Event Synchronization

BARRIER(nprocs): wait here till nprocs processes get here
- Built using lower level primitives \textit{i.e} locks, semaphores
- Global sum example: wait for all to accumulate before using sum
- Often used to separate phases of computation

<table>
<thead>
<tr>
<th>Process P_1</th>
<th>Process P_2</th>
<th>Process P_nprocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>set up eqn system</td>
<td>set up eqn system</td>
<td>set up eqn system</td>
</tr>
<tr>
<td>\textbf{Barrier} (name, nprocs)</td>
<td>\textbf{Barrier} (name, nprocs)</td>
<td>\textbf{Barrier} (name, nprocs)</td>
</tr>
<tr>
<td>solve eqn system</td>
<td>solve eqn system</td>
<td>solve eqn system</td>
</tr>
<tr>
<td>\textbf{Barrier} (name, nprocs)</td>
<td>\textbf{Barrier} (name, nprocs)</td>
<td>\textbf{Barrier} (name, nprocs)</td>
</tr>
<tr>
<td>apply results</td>
<td>apply results</td>
<td>apply results</td>
</tr>
<tr>
<td>\textbf{Barrier} (name, nprocs)</td>
<td>\textbf{Barrier} (name, nprocs)</td>
<td>\textbf{Barrier} (name, nprocs)</td>
</tr>
</tbody>
</table>

- Conservative form of preserving dependencies, but easy to use
Point-to-point (Ordering) Event Synchronization

(Not Used Here)

One process notifies another of an event so it can proceed:

- Needed for **task ordering** according to **data dependence between tasks**
- Common example: producer-consumer (bounded buffer)
- Concurrent programming on uniprocessor: semaphores
- Shared address space parallel programs: semaphores, or use ordinary variables as **flags**

Initially flag = 0

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
</tr>
</thead>
</table>

A = 1;  

b: flag

a: while (flag is 0) do nothing;  

print A;

*Busy-waiting (i.e. spinning)*

- Or block process (better for uniprocessors?)

Or compute using “A” as operand

i.e busy-wait (or spin on flag)
Message Passing Grid Solver

- Cannot declare A to be a shared array any more
  Thus
  No shared address space

- 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
  n/p rows in this case

- Explicit transfers (communication) of entire border or “Ghost” rows between tasks is needed (as shown next slide)

- Structurally similar to SAS (e.g. SPMD), but orchestration is different
  - Data structures and data access/naming e.g Local arrays vs. shared array
  - Communication
  - Synchronization
    } Via Send/receive pairs +

Explicit
Implicit
Message Passing Grid Solver

- Parallel Computation = $O(n^2/p)$
- Communication of rows = $O(n)$
- Communication of local DIFF = $O(p)$
- Computation = $O(n^2/p)$
- Communication = $O(n + p)$
- Communication-to-Computation Ratio = $O((n+p)/(n^2/p)) = O((np + p^2) / n^2)$

$n/p$ rows or $n^2/p$ points per process or task

Same block assignment as before

Time per iteration:

\[ T = T(\text{computation}) + T(\text{communication}) \]
\[ T = O(n^2/p + n + p) \]

nprocs = number of processes = number of processors = p
Pseudo-code, Parallel Equation Solver for Message Passing

1. int pid, n, b; /*process id, matrix dimension and number of processors to be used*/
2. float **myA;
3. main()
4. begin
5.  read(n);  read(nprocs); /*read input matrix size and number of processes*/
6a. CREATE (nprocs-1, Solve); /*main process becomes a worker too*/
6b. Solve();
6c. WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
7. end main

10. procedure Solve()
11. begin
13. int i,j, pid, n' = n/nprocs, done = 0;
14. float temp, tempdiff, mydiff = 0; /*private variables*/
15. while (!done) do
16.  mydiff = 0; /*set local diff 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[n',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[n'+1,0],n*sizeof(float), pid+1,ROW);
/*border rows of neighbors have now been copied into myA[0,*] and myA[n'+1,*]*/
17.  for i ← 1 to n' do /*for each of my (nonghost) rows*/
18.    for j ← 1 to n do /*for all nonborder elements in that row*/
19.      temp = myA[i,j];
21.      mydiff += abs(myA[i,j] - temp);
22.  endfor
23.  endfor
/*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
25a. if (pid != 0) then /*process 0 holds global total diff*/
25b.  SEND(mydiff,sizeof(float),0,DIFF);
25c.  RECEIVE(done,sizeof(int),0,DONE);
25d. else /*pid 0 does this*/
25e.    for i ← 1 to nprocs-1 do /*for each other process*/
25f.      RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.      mydiff += tempdiff; /*accumulate into total*/
25h.  endfor
25i. if (mydiff/(n*n) < TOL) then done = 1;
25j. for i ← 1 to nprocs-1 do /*for each other process*/
25k.    SEND(done,sizeof(int),i,DONE);
25l.  endfor
25m.  endif
26. endwhile
27. end procedure

# of processors = p = nprocs

Message Passing

Initialize local rows myA

Send one or two ghost rows

Exchange ghost rows (send/receive)

Receive one or two ghost rows

Sweep over n/p rows = n^2/p points per task
T = O(n^2/p)

Communicate O(n) exchange ghost rows

Before start of iteration

Computation O(n^2/p)

Local Difference

Only Pid 0 tests for convergence

Pid 0: Done?
calculate global difference and test for convergence
send test result to all processes

Send mydiff to pid 0
Receive test result from pid 0

T = O(n^2/p + n + p)
Notes on Message Passing Program

- Use of ghost rows.  
  - Receive does not transfer data, send does (sender-initiated)
    - Unlike SAS which is usually receiver-initiated (load fetches data)
- Communication done at beginning of iteration (exchange of ghost rows).
- Explicit communication in whole rows, not one element at a time
- Core similar, but indices/bounds in local space rather than global space.
- Synchronization through sends and blocking receives (implicit)
  - Update of global difference and event synch for done condition
  - Could implement locks and barriers with messages
- Only one process (pid = 0) checks convergence (done condition).
- Can use REDUCE and BROADCAST library calls to simplify code:

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

Tell all tasks if done
Message-Passing Modes: **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

Send/Receive

- Synchronous
  - Blocking
    - Send: Wait until message is sent
    - Receive: Wait until message is received
  - Easy to create
  - Deadlock

- Asynchronous
  - Non-blocking
    - Immediate
    - Return immediately (both)

  - Affect event synch (mutual exclusion implied: only one process touches data)

  - Affect ease of programming and performance

Synchronous messages provide built-in synch. through match

- Separate event synchronization needed with async. messages

With synchronous messages, our code is deadlocked. Fix?

**Use asynchronous blocking sends/receives**
Message-Passing Modes: Send and Receive Alternatives

Synchronous Message Passing:  
Process X executing a synchronous send to process Y has to wait until process Y has executed a synchronous receive from X.

Asynchronous Message Passing:

Blocking Send/Receive:  
A blocking send is executed when a process reaches it without waiting for a corresponding receive. Returns when the message is sent. A blocking receive is executed when a process reaches it and only returns after the message has been received.

Non-Blocking Send/Receive:  
A non-blocking send is executed when reached by the process without waiting for a corresponding receive. A non-blocking receive is executed when a process reaches it without waiting for a corresponding send. Both return immediately.

Most Common Type

In MPI: MPI_Ssend()  MPI_Srecv()

In MPI: MPI_Send()  MPI_Recv()

In MPI: MPI_Isend()  MPI_Irecv()

MPI = Message Passing Interface
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 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 synch. case)
  - Mutual exclusion implied

No SAS
Correctness in Grid Solver Program

Decomposition and Assignment similar in SAS and message-passing

Orchestration is different:

- Data structures, data access/naming, communication, synchronization

<table>
<thead>
<tr>
<th>AKA shared?</th>
<th>SAS</th>
<th>Msg-Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment indept of data layout?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Requirements for performance are another story ...