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

(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 => 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 x 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

From last lecture

More reading: PP Chapter 11.3 (Pages 352-364)
Solution of Linear System of Equation By Synchronous Iteration

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

- **Setup**
  - Initialize all points
  - Expression for updating each interior point:

- **Iterate over (update) all interior n^2 points**
  - \(O(n^2)\)

- **Converged?**
  - Find Error (Global Difference)
  - Is Error < Tolerance Limit? (Threshold)
  - Or maximum number of iterations is reached

- **Iterate Again**
  - No
  - One Iteration Or Sweep

- **Done**
  - Yes
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 x 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:
  1. Interior n-by-n points of (n+2)-by-(n+2) updated in each sweep (iteration)
  2. Updates done in-place in grid, and difference from previous value is computed
  3. Accumulate partial differences into a global difference at the end of every sweep or iteration
  4. 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
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 \leftarrow \text{malloc} \text{ (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 \leftarrow 1 to n do /*sweep over nonborder points of grid*/
18. for j \leftarrow 1 to n do /*save old value of element*/
19. temp = A[i,j];
21. end for
22. end for
23. if (diff/(n*n) < TOL) then done = 1;
24. end while
25. end procedure

Setup

Call equation solver

Iterate until convergence

i.e one iteration

Sweep $O(n^2)$ computations

Global Difference

Update Points

Done?

TOL, tolerance or threshold

Initialize grid points
Decomposition

• Simple way to identify concurrency is to look at loop iterations

  2. Dependency analysis; if not enough concurrency is found, then look further into application

  • Not much concurrency here at this level (all loops sequential)

  • 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.

  or • Restructure loops, use global synch; load imbalance and too much synch

Expression for updating each interior point:

Decomposition: **Exploit Application Knowledge**

- Reorder grid traversal: red-black ordering
  - Two parallel sweeps
    - Each with parallel \( \frac{n^2}{2} \) points updates

  ![Grid traversal diagram]

  - Maximum Degree of parallelism = 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: \( T = O(\log_2 n^2) \)

- 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 DOP = \( n^2 = O(n^2) \)

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

  i.e. Max Software 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 $\leftarrow$ 1 to n do /*a parallel loop nest*/
18. for_all j $\leftarrow$ 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
Assignment: (Update \( \frac{n}{p} \) rows per task)

i.e Task Assignment
\[ p = \text{number of processes or processors} \]

\[ \text{i.e n/p points per task} \]

• 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

  \[ p = \text{number of processors < n} \]
  \[ p \text{ tasks or processes} \]
  \[ \text{Task = updates } \frac{n}{p} \text{ rows} = \frac{n^2}{p} \text{ elements} \]
  \[ \text{Computation} = O(\frac{n^2}{p}) \]
  \[ \text{Communication} = O(n) \]
  \[ \sim 2n \text{ (2 rows)} \]
  \[ \text{Communication-to-Computation ratio} = O\left( \frac{n}{(n^2/p)} \right) = O(p/n) \]
  \[ \text{Lower C-to-C ratio is better} \]

• 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 \( \text{C-to-C} = O(1) \)
  - Block assign. reduces communication by keeping adjacent rows together

• Let’s examine orchestration under three programming models:

<table>
<thead>
<tr>
<th>1- Data Parallel</th>
<th>2- Shared Address Space (SAS)</th>
<th>3- Message Passing</th>
</tr>
</thead>
</table>

Why Block Assignment

\[ p \text{ tasks} \]
\[ \text{Instead of } n^2 \]
Data Parallel Solver

```c
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;
15.   DECOMP A[BLOCK,*, nprocs];
16.   while (!done) do
17.     mydiff = 0;
18.     for_all i ← 1 to n do
19.       for_all j ← 1 to n do
20.         temp = A[i,j];
23.         mydiff += abs(A[i,j] - temp);
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) ≤ T(iteration) ≤ O(n^2/p + p)
```

**Setup/Initialize Points**
- `nprocs` = number of processes = p

**Block decomposition by row**
- n/p rows per processor

**Sweep/Iteration:**
- \( T = O(n^2/p) \)

Add all local differences (REDUCE)
- cost depends on architecture
- and implementation of REDUCE
- best: \( O(\log_2 p) \) using binary tree reduction
- Worst: \( O(p) \) sequentially
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

- Array of grid points “A” in shared memory
- ...
Pseudo-code, Parallel Equation Solver for Shared Address Space (SAS)

1. int n, nprocs; /*matrix dimension and number of processors to be used*/
2. float **A, diff; /*A is global (shared) array representing the grid*/
2a. LOCKDEC(diff_lock); /*declaration of lock to enforce mutual exclusion*/
2c. BARDEC (bar1); /*declaration of lock to enforce mutual exclusion*/
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; /*private variables*/
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. endfor
25a. LOCK(diff_lock); /*update global diff if necessary*/
25b. diff += mydiff;
25c. UNLOCK(diff_lock); /*Critical Section: global difference*/
25d. BARRIER(bar1, nprocs); /*ensure all reach here before checking if done*/
25e. if (diff/(n*n) < TOL) then done = 1; /*check convergence; all get same answer*/
26. endwhile
27. end procedure

SAS

Array “A” is shared (all grid points)

- # of processors = p = nprocs
- pid = process ID, 0 .... P-1
- T = O(p)
- T(p) = O(n^2/p + p)
-Serialized update of global difference
- Check/test convergence: all processes do it

Main process or thread

Create p-1 processes

Setup

Loop Bounds/Which Rows?

mymin = low row

mymax = high row

Private Variables

Done?

Global Difference

Mutual Exclusion (lock) for global difference

Barrier 1 (Start sweep)

Barrier 2 (sweep done)

Barrier 3
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
  
  **But**
  - Each process has private mydiff variable

  **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?**

SPMD = Single Program Multiple Data

Which n/p rows?

Otherwise each process must enter the shared global difference critical section for each point, n²/p times (n² times total) instead of just p times per iteration for all processes

Using LOCK () .... UNLOCK ()
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 = Local Difference*

• Need the sets of operations to be **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. $n^2/p$ times per process without mydiff

\[ O(p) \text{ total number of accesses to critical section} \]

\[ O(n^2) \text{ total number of accesses to critical section by all processes} \]
Global (or group) Event Synchronization

BARRIER(nprocs): wait here till nprocs processes get here
  • Built using lower level primitives \( \text{i.e locks, semaphores} \)
  • 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_{\text{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

SAS (Not Used or Needed 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>
<tbody>
<tr>
<td>A = 1;</td>
<td>i.e P2 computed A</td>
</tr>
<tr>
<td>b: flag</td>
<td></td>
</tr>
</tbody>
</table>

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

- **Busy-waiting** (i.e. spinning)
  - Or block process (better for uniprocessors?)

Or compute using “A” as operand
Message Passing Grid Solver

- Cannot declare A to be a 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
  - 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
  - Communication
  - Synchronization

\[ \text{Via Send/receive pairs} + \]

Explicit

Implicit

\{ e.g. Local arrays vs. shared array \}

\{ n/p rows in this case \}

\{ myA \}

\{ myA arrays \}

\{ Each n/p rows in local memory \}

\{ No shared address space \}

\{ At start of each iteration \}
- 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

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

```c
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*/
6.   CREATE (nprocs-1, Solve);  /*main process becomes a worker too*/
7.   Solve();
8.   WAIT_FOR_END (nprocs-1);  /*wait for all child processes created to terminate*/
9. end main
10. procedure Solve()
11.   begin
12.     int i,j, pid, n' = n/nprocs, done = 0;
13.     float temp, tempdiff, mydiff = 0;  /*private variables*/
14.     myA = malloc(a 2-d array of size [n/nprocs + 2] by n+2);  /*my assigned rows of A*/
15.     initialize(myA);  /*initialize my rows of A, in an unspecified way*/
16.     mydiff = 0;
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
24.     myA = realloc(myA, sizeof(float)*n'*(n+2));  /*for all processes*/
25.     if (pid != 0) then  /*process 0 holds global total diff*/
26.         SEND(mydiff,sizeof(float),0,DIFF);
27.     else  /*pid 0 does this*/
28.         for i = 1 to nprocs-1 do  /*for each other process*/
29.             RECEIVE(tempdiff,sizeof(float),*,DIFF);
30.             mydiff += tempdiff;  /*accumulate into total*/
31.         endfor
32.     endif
33.     if (mydiff/(n*n) < TOL) then done = 1;
34.     for i = 1 to nprocs-1 do  /*for each other process*/
35.         SEND(done,sizeof(int),i,DONE);
36.     endfor
37.     if (done) then done = 1;
38. endprocedure
```

### Communication

- **O(n)** exchange ghost rows
- **O(p)** communication

### Computation

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

---

**Pseudocode Summary**

- **Create p-1 processes**
- **Initialize local rows myA**
- **Send one or two ghost rows**
- **Exchange ghost rows**
- **Receive one or two ghost rows**
- **Sweep over \( n/p \) rows**
- **Send mydiff to pid 0**
- **Receive test result from pid 0**
- **Pid 0: Done?**
  - calculate global difference and test for convergence
  - send test result to all processes

---

**Parallel Message Passing**

- **# of processors = p = nprocs**
Notes on Message Passing Program

• Use of ghost rows.  **Or border 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:

  /*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

<table>
<thead>
<tr>
<th>Send/Receive</th>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blocking</strong></td>
<td>Send: Wait until message is sent</td>
<td>Non-blocking</td>
</tr>
<tr>
<td>Receive: Wait until message is received</td>
<td></td>
<td>Return immediately (both)</td>
</tr>
</tbody>
</table>

- 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 asynch. 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.

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 ...