4 steps:

Decomposition, Assignment, Orchestration, Mapping

+ Scheduling

- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly
Parallelization of An Example Program

Examine a simplified version of a piece of Ocean simulation
  • Iterative equation solver

Illustrate parallel program in low-level parallel language
  • C-like pseudocode with simple extensions for parallelism
  • Expose basic communication and synchronization primitives that must be supported by parallel programming model.

(PCA Chapter 2.3)
Grid Solver Example

- Simplified version of solver in Ocean simulation
- Gauss-Seidel (near-neighbor) sweeps (iterations) to convergence
  - Interior \( n \times n \) points of \((n+2) \times (n+2)\) updated in each sweep
  - Updates done in-place in grid, and diff. from prev. value computed
  - Accumulate partial diffs into global diff at end of every sweep
  - Check if error has converged (to within a tolerance parameter)
    - If so, exit solver; if not, do another sweep (iteration)

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;` /*A is an (n + 2)-by-(n + 2) array*/
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`
19. `temp = A[i,j];` /*save old value of element*/
21. `diff += abs(A[i,j] - temp);`
22. `end for`
23. `end for`
24. `if (diff/(n*n) < TOL) then done = 1;`
25. `end while`
26. `end procedure`
Decomposition

• Simple way to identify concurrency is to look at loop iterations
  
  \textit{Dependence analysis}; if not enough concurrency, then look further

• Not much concurrency here at this level (all loops \textit{sequential})

• Examine fundamental dependences, ignoring loop structure

\begin{itemize}
  \item Concurrency $O(n)$ along anti-diagonals, serialization $O(n)$ along diag.
  \item Retain loop structure, use pt-to-pt synch; Problem: too many synch ops.
  \item Restructure loops, use global synch; imbalance and too much synch
\end{itemize}
Exploit Application Knowledge

• Reorder grid traversal: red-black ordering

![Grid Diagram]

Degree of parallelism = $O(n^2)$
Type of parallelism: Data parallelism
One point update per task
  Computation = 1
  Communication = 4
Communication to computation ratio = 4

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

• 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)
• Ocean uses red-black; we use **simpler**, asynchronous one to illustrate
  – no red-black, simply ignore dependences within sweep
  – Sequential order same as original, parallel program **nondeterministic**
Decomposition Only

15. while (!done) do
16.    diff = 0;
17.   for_all i ← 1 to n do
18.    for_all j ← 1 to n do
19.      temp = A[i, j];
22.      diff += abs(A[i, j] - temp);
23.   end for_all
24. end for_all
25. if (diff/(n*n) < TOL) then done = 1;
26. end while

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

Task = update one grid point

Parallel PRAM $O(1)$

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

Task = grid row
Computation = $O(n)$
Communication = $O(n)$
Communication to Computation ratio = $O(1)$
Assignment

• Static assignments (given decomposition into rows)
  – Block assignment of rows: Row $i$ is assigned to process \( \begin{bmatrix} i \\ p \end{bmatrix} \)
  – Cyclic assignment of rows: process $i$ is assigned rows $i$, $i+p$, and so on

- Dynamic assignment
  – Get a row index, work on the row, get a new row, and so on

- Static assignment into rows reduces concurrency (from $n$ to $p$)
  – Block assign. reduces communication by keeping adjacent rows together

- Let’s dig into orchestration under three programming models
Data Parallel Solver

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

3. main()
4. begin
5. read(n); read(nprocs); /*read input grid size and number of processes*/
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 /*outermost loop over sweeps*/
16. mydiff = 0; /*initialize maximum difference to 0*/
17. for_all i ← 1 to n do /*sweep over non-border points of grid*/
18. for_all j ← 1 to n do /*save old value of element*/
19. temp = A[i,j];
21. mydiff += abs(A[i,j] - temp);
22. end for_all
23. end for_all
24a. REDUCE (mydiff, diff, ADD); /*add all local differences*/
25. if (diff/(n*n) < TOL) then done = 1;
26. end while
27. end procedure

Block decomposition by row

O(n²/p)

- Add all local differences
- Cost depends on architecture
- Best: O(log₂p)
- 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
Pseudocode, Parallel Equation Solver for Shared Address Space (SAS)

```plaintext
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. /*diff is global (shared) maximum difference in current sweep*/
2c. LOCK(diff_lock); /*declaration of lock to enforce mutual exclusion*/
2d. 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; /*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];
22. mydiff += abs(A[i,j] - temp);
23. endfor
24. endfor
25a. LOCK(diff_lock); /*update global diff if necessary*/
25b. diff += mydiff;
25c. UNLOCK(diff_lock);
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
```

# of processors = p = nprocs

$O(n^2/p)$

$O(p)$ Serialzed update of diff
Notes on SAS Program

• **SPMD**: not lockstep or even 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?
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}

- Need the sets of operations to be atomic (mutually exclusive)
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.

Can lead to significant **serialization** if contended

- Especially since expect non-local accesses in critical section
- Another reason to 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
  Barrier (name, nprocs)
  solve eqn system
  Barrier (name, nprocs)
  apply results
  Barrier (name, nprocs)

  set up eqn system
  Barrier (name, nprocs)
  solve eqn system
  Barrier (name, nprocs)
  apply results
  Barrier (name, nprocs)

  set up eqn system
  Barrier (name, nprocs)
  solve eqn system
  Barrier (name, nprocs)
  apply results
  Barrier (name, nprocs)

• Conservative form of preserving dependences, but easy to use

WAIT_FOR_END (nprocs-1)
Point-to-point 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>
<tbody>
<tr>
<td></td>
<td>A = 1;</td>
</tr>
<tr>
<td>a: while (flag is 0) do nothing; b: flag</td>
<td>print A;</td>
</tr>
<tr>
<td></td>
<td>= 1;</td>
</tr>
</tbody>
</table>

- Busy-waiting or spinning
  - Or block process (better for uniprocessors?)
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, single-consumer
**Message Passing Grid Solver**

- Cannot declare A to be 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

- Transfers of entire rows between traversals

- Structurally similar to SAS (e.g. SPMD), but orchestration is different
  - Data structures and data access/naming
  - Communication
  - Synchronization
• 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 = $O\left(\frac{n+p}{(n^2/p)}\right) = O\left(\frac{np + p^2}{n^2}\right)$
1. int pid, n, b; /*process id, matrix dimension and number of processors to be used*/
2. float **myA; # of processors = p = nprocs
3. main()
4. begin
5.  read(n); read(nprocs); /*read input matrix size and number of processes*/
8a.  CREATE(nprocs-1, Solve); /*main process becomes a worker too*/
8b.  Solve(); /*wait for all child processes created to terminate*/
8c.  WAIT_FOR_END(nprocs-1);
9. 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*/
6. myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2); /*my assigned rows of A*/
7. initialize(myA); /*initialize my rows of A, in an unspecified way*/
15. while (!done) do /*set local diff to 0*/
16. mydiff = 0;
16a. if (pid != 0) then SEND(&myA[1,0], n*sizeof(float), pid-1, ROW); /*process 0 holds global total diff*/
16b. if (pid = nprocs-1) then SEND(&myA[n'+1,0], n*sizeof(float), pid+1, ROW);
16c. if (pid != 0) then RECEIVE(&myA[0,0], n*sizeof(float), pid-1, ROW); /*for each of my (nonghost) rows*/
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 all nonborder elements in that row*/
18. for j ← 1 to n do /*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
19. temp = myA[i,j];
21. mydiff += abs(myA[i,j] - temp);
22. endfor
23. endfor
25. if (pid != 0) then /*process 0 does this*/
25a. SEND(mydiff, sizeof(float), 0, DIFF); /*for each other process*/
25b. RECEIVE(done, sizeof(int), 0, DONE);
25c. else /*pid 0 does this*/
25d. for i ← 1 to nprocs-1 do /*accumulate into total*/
25e. RECEIVE(tempdiff, sizeof(float), i, DIFF);
25f. mydiff += tempdiff;
25g. endfor
25h. if (mydiff/(n*n) < TOL) then done = 1;
25i. for i ← 1 to nprocs-1 do /*for each other process*/
25j. SEND(done, sizeof(int), i, DONE);
25k. endfor
25m. endif
26. endwhile
27. end procedure
Notes 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 one element at a time
• Core similar, but indices/bounds in local rather than global space
• Synchronization through sends and receives
  – Update of global difference 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*/
25b.  REDUCE(0, mydiff, sizeof(float), ADD);
25c.  if (pid == 0) then
25i.    if (mydiff/(n*n) < TOL) then done = 1;
25k.    endif
25m.  BROADCAST(0, done, sizeof(int), DONE);
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

- Affect event synch (mutual excl. by fiat: 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 synch. messages, our code is deadlocked. Fix?
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 comm. patterns)
- Communication is explicit
- Synchronization implicit in communication (at least in synch. case)
  - mutual exclusion by fiat
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></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 ...